Eötvös Loránd Tudományegyetem
Informatikai Kar
Algoritmusok és Alkalmazásaik Tanszék

Rendező algoritmusok működését szemléltető program

dr. Ásványi Tibor
Egyetemi docens

Bacsa Roland
Programtervező Informatikus BSc

Budapest, 2017

TARTALOMJEGYZÉK

1. BEVEZETÉS

Rendező algoritmusok ismerete elengedhetetlen egyes programozási problémák hatékony megoldása érdekében. Az első lépés ahhoz, hogy megértsük miért is fontos a megfelelő rendezés használata, tisztáznunk kell mi is az **algoritmus**? Az algoritmus egy jól definiált procedúra (vagy más szóval eljárás), mely adott lépéssorozat után a bemeneti értékek alapján egy elvárt kimenetet ad vissza.

Egy problémát a való életben számos módon megoldhatunk. Például az ELTE Lágymányosi Campusra a XII. kerületből eljuthatunk a Puskás Ferenc Stadion érintésével, pár átszállással, az 1-es villamos segítségével. Azonban, lehet nem lenne célszerű ekkora kerülőúton mennünk, ha a záróvizsga fél óra múlva kezdődik, és van közvetlen buszjárat is oda.

Egy-egy problémát a programozásban is számos algoritmussal megoldhatunk, azonban nem mindegy, hogy **mennyi idő alatt** kapjuk meg az elvárt eredményt. A rendező algoritmusok hatékonyságát a **műveletigényükkel** tudjuk jellemezni. A műveletigény egy nagyságrendi becslés az algoritmus által végzett műveletek számára. Rendező algoritmusok esetén a **kulcs összehasonlítások** –, és az **elemek mozgatásának száma** jól jellemzi az algoritmus tényleges futási idejét.

Megkülönböztetünk legrosszabb, várható és legjobb eseteket, melyek mind különböző nagyságrendűek lehetnek. Például a gyors rendezés, ami $\theta(n \log(n))$ átlagos műveletigénnyel rendelkezik, előfordulhat, hogy rosszabbul teljesít egy nála sokkal nagyobb, $\theta(n^2)$ átlagos műveletigényű társánál, a beszúró rendezésnél. Egy rendezett 30 elemű tömbön ugyanis a beszúró rendezés 29 összehasonlítás után, míg a gyors rendezés csak 435 összehasonlítás után végez.

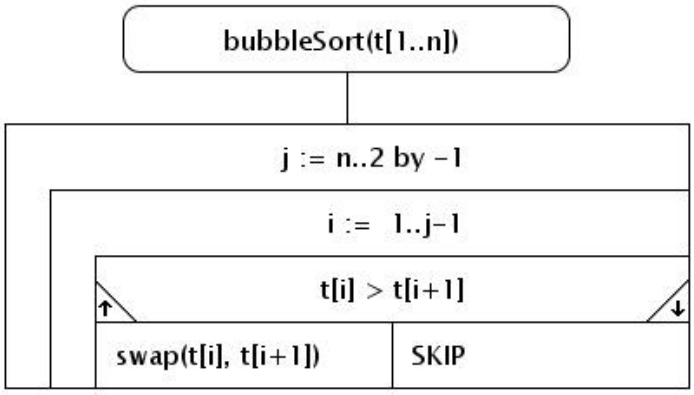
Szakdolgozatomban hat különböző, tömb adatszerkezeten rendező algoritmus működését szemléltetem. Három $\theta(n^2)$ átlagos műveletigénnyel rendelkező algoritmusét: buborék, beszúró, maximum kiválasztó rendezés, valamint három $\theta(n \log(n))$ átlagos műveletigényűét: kupac, gyors, összefésülő rendezés.

2. RENDEZŐ ALGORITMUSOK ISMERTETÉSE

A következőkben néhány lényegre törő mondattal ismertetem a szakdolgozat témáját adó rendező algoritmusok működését. A leírás alatt találhatóak az algoritmusok jellemzésére használt struktogramjuk, melyek segíthetik a leírtak megértését.
Átfogóbb, részletesebb ismeretekért az irodalomjegyzékben mellékelt [1], valamint [2] azonosítóval ellátott hivatkozásokat ajánlom.

2.1. Buborék rendezés

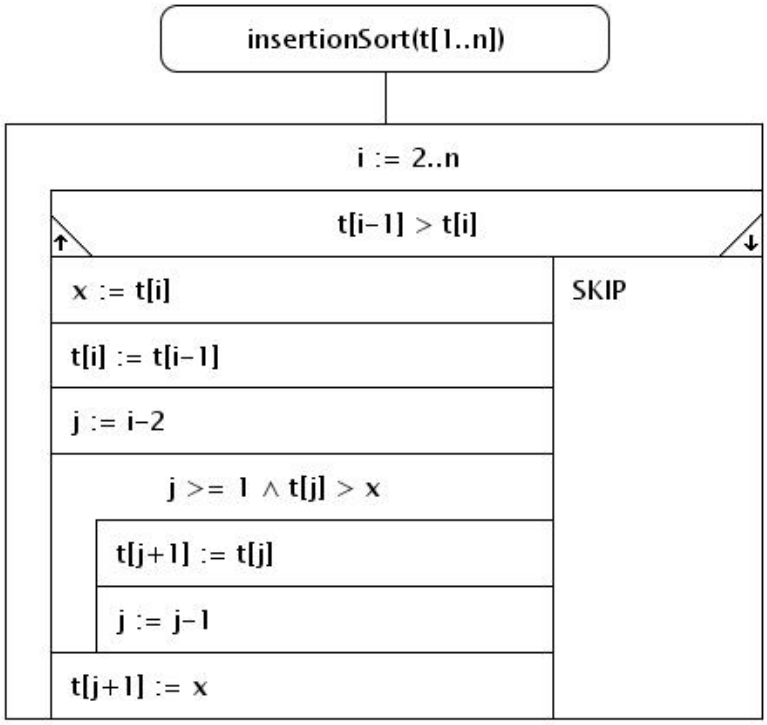
A buborék rendezés lényege, hogy a tömb egymás mellett lévő elemein párosával haladunk végig, és a rossz sorrendben lévőket megcseréljük. Miután a tömb végére érünk, a legnagyobb elem a helyére kerül. Ezután a lépéseket megismételjük egyel rövidebb hosszúságú tömbre addig, amíg elő nem áll a rendezett tömb.



A buborék rendezés struktogramja.

2.2. Beszúró rendezés

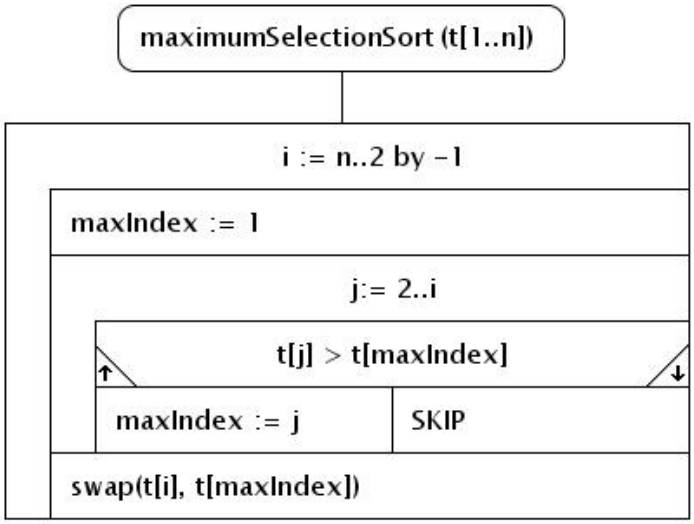
A beszúró rendezés során a tömböt egy rendezett és egy rendezetlen részre osztjuk. Kezdetben a rendezett rész az első tömbelemből áll. A tömb elemeit sorba véve, azokat beszúrjuk a rendezett rész elemei közé. n-1 beszúrás után a tömb elemei rendezett sorrendben fognak állni.



A beszúró rendezés struktogramja.

2.3. Maximum kiválasztó rendezés

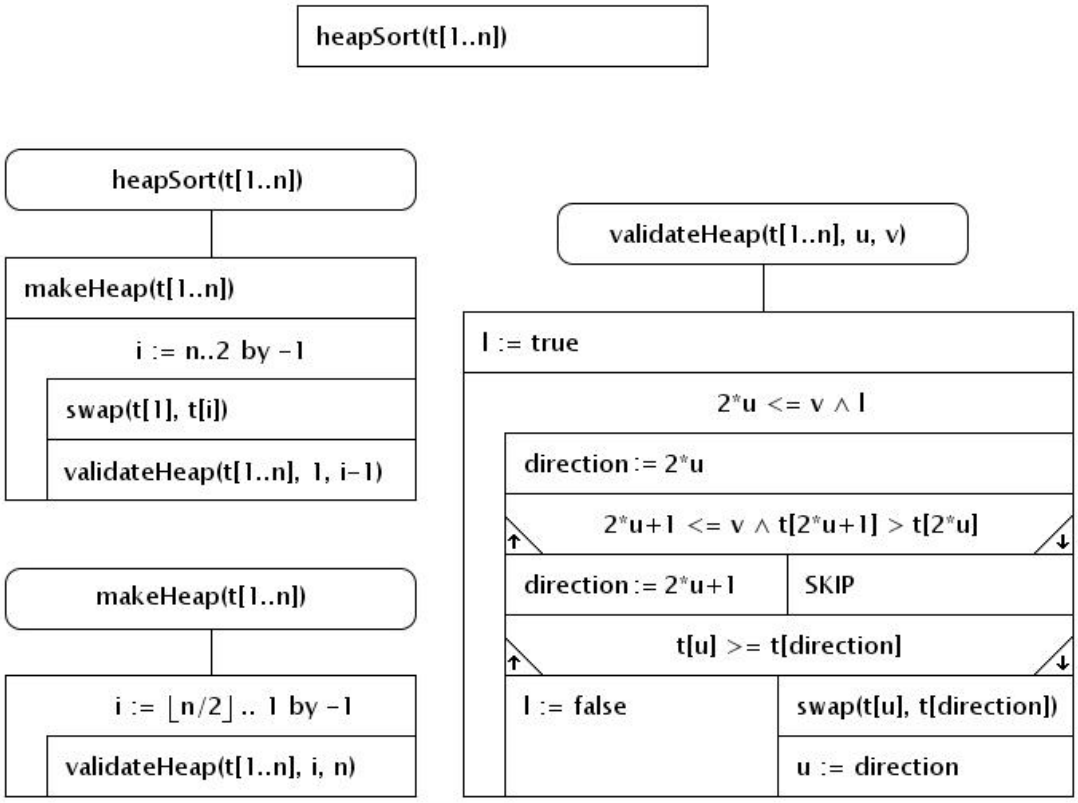
A maximum kiválasztó rendezés lényege, hogy a tömb elemeit végignézve, megkeressük a legnagyobbat. Ezután a legnagyobb elemet megcseréljük a legutolsó elemmel. A lépéseket megismételjük egyel rövidebb hosszúságú tömbre addig, amíg elő nem áll a rendezett tömb.



A maximum kiválasztó rendezés struktogramja.

2.4. Kupac rendezés

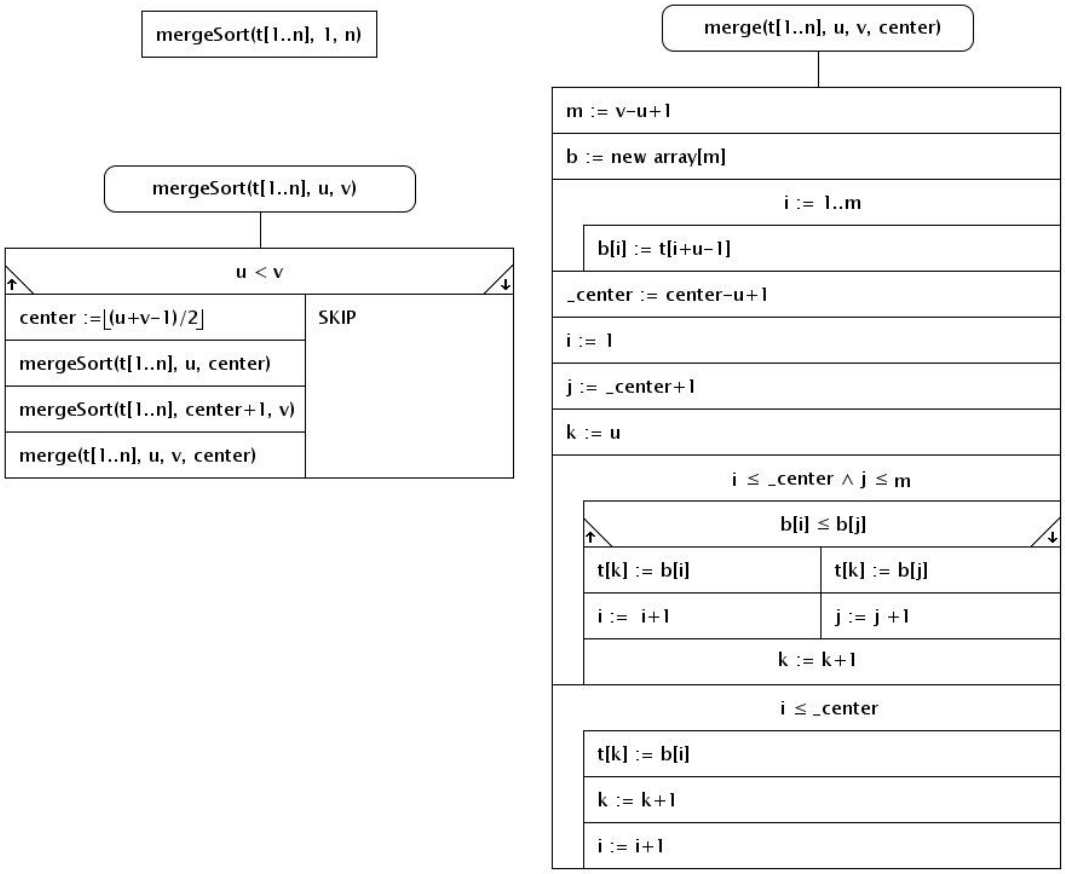
A kupacrendezést úgy lehet elképzelni, mint egy továbbfejlesztett maximum kiválasztó rendezést. A továbbfejlesztés a kupac adatszerkezet használatában nyilvánul meg. Működése során előbb felépíti a kupacot, majd egyesével kiemeli a gyökérelemet. Definíciója miatt a kupacot reprezentáló résztömb első eleme (gyökér elem) mindig a kupac legnagyobb eleme lesz. A kiemelt elem a résztömb végére (a rendezett rész elejére) kerül, így az egyre kisebb méretű rendezetlen résztömb végül eltűnik, és az elemek rendezett sorrendbe kerülnek.



A kupac rendezés stuktogramja.

2.5. Összefésülő rendezés

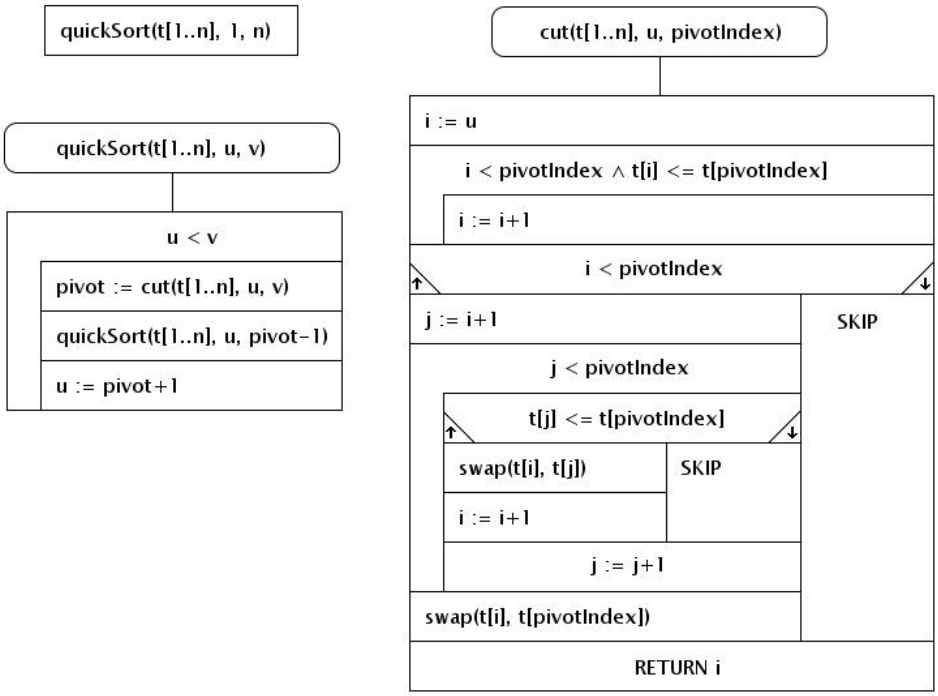
Az összefésülő rendezés során az algoritmus a rendezendő tömböt rekurzívan kettő részre bontja, majd a résztömböket összefésüli. Így a rekurzió során az 1 hosszú rendezett sorozatokból végül egy (minden elemet magában foglaló) rendezett sorozat jön létre.



Az összefésülő rendezés stuktogramja.

2.6. Gyorsrendezés

Gyorsrendezés során az algoritmus minden lépésben egy elemet választ ki a tömbből, majd azt a helyére rakja. Bal oldalára kerülnek a tőle kisebb elemek, a jobb oldalára pedig a nagyobb elemek. A helyrekerült elemmel már nincs dolga az algoritmusnak, a továbbiakban csak a fennmaradó részekkel kell foglalkoznia.



A gyorsrendezés struktogramja.

3. FELHASZNÁLÓI DOKUMENTÁCIÓ

3.1. A program rövid ismertetése

A feladat egy olyan program fejlesztése volt, mellyel a felhasználó a **tömb** adatszerkezeten **rendező** (legismertebb) algoritmusok működését lépésről-lépésre nyomon követheti. Az **összehasonlítások** és **kulcs mozgatások** száma folyamatosan megjelennek a felhasználó számára, ezáltal könnyen eldöntheti, hogy melyik rendezés milyen esetekben hatékony, és melyekben kevésbé.

A szemléltetés eszközeként **oszlopokat** használok, melyek egy-egy tömbelemet képviselnek. Magasságuk a tömbelemtől függőek, színük pedig az algoritmus lépései alapján változnak. Így a felhasználó számára az algoritmus minden lépése jól elkülöníthető, ami segítséget nyújthat a működésének megértésében is.

3.2. Telepítés

A program futtatásához elengedhetetlen a **Java Runtime Environment** telepítése, mely minden Java programozási nyelven írt alkalmazás futtatásához szükséges. A legfrissebb 8-as verziójú Java Runtime Environment az Oracle honlapjáról tölthető le a következő link segítségével:

http://oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html

A Java Runtime Environment telepítése után már a program futtatását lehetővé tévő környezet rendelkezésre áll. A programot külön telepíteni nem szükséges, indításához egyszerűen a *SAV.jar* nevű futtatható állományt kell megnyitni.

3.3. Rendszerkövetelmények

A program megfelelő működéséhez és megjelenítéséhez az alábbi paraméterekkel rendelkező számítógép javasolt:

	Minimális	**Ajánlott**
Operációs rendszer:	Bármilyen Java Runtime Environment futtatására alkalmas rendszer.	
Felbontás:	800x600	2560x1440
Processzor:	1 magos, 3 GHz órajelű	2 magos, 2.5 GHz órajelű
Memória	1 GB	
Videokártya:	Nem szükséges dedikált videokártya.	
Háttértár:	128 MB szabad terület	
Bemeneti perifériák:	Egér	

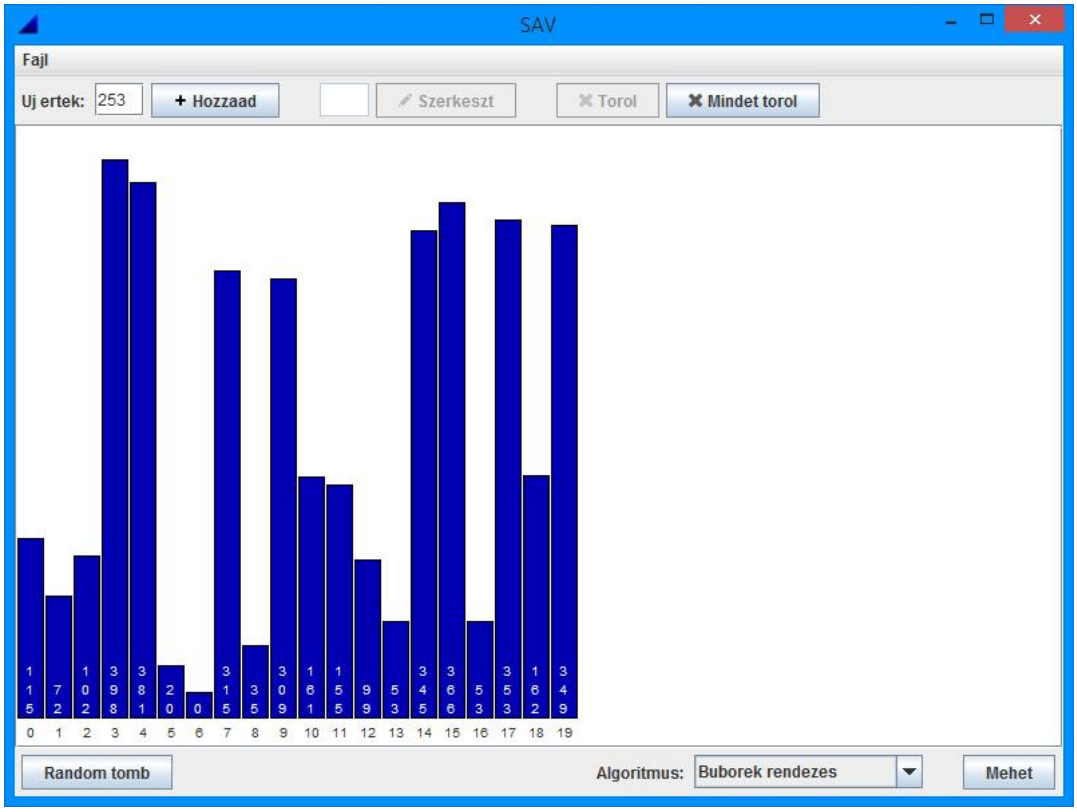
Megjegyzés: a program minimálisan 800x600 pixel méretű ablakot használ. Ez a grafikai elemek problémamentes megjelenítéséhez szükséges. Habár a program 800x600 pixel méretű felbontás esetén is megfelelően működik, azonban egyes rendezéseknek (nagyméretű tömbök esetén) ennél nagyobb szélességű területre lehet szükségük, ezért az algoritmus nyomonkövethetőségének érdekében 2560x1440 pixel méretű felbontás javasolt.

A Java Runtime Environment telepítéséhez 124 MB háttértárra van szükség. Az alkalmazás ennél sokkal kevesebb tárterületet igényel.

3.4. A program használata

A program kettő megjelenítési móddal rendelkezik: az egyik a **szerkesztő felület**, ahol a rendezendő tömböt szabhatjuk testre; a másik a rendező algoritmus lépéseit mutató **szemléltető felület**.

A program elindítása után a szerkesztő felület töltődik be alapértelmezetten, melyen egy előre generált 20 elemű tömb is megjelenik véletlenszerű értékekkel.



A szerkesztő felület.

Az ablak mérete a program indításakor 800x600 pixel méretű, azonban ha szükséges lehetőség van átméretezni az ablakot egy tetszőlegesen nagyobb méretűre.

3.4.1. Szerkesztő felület

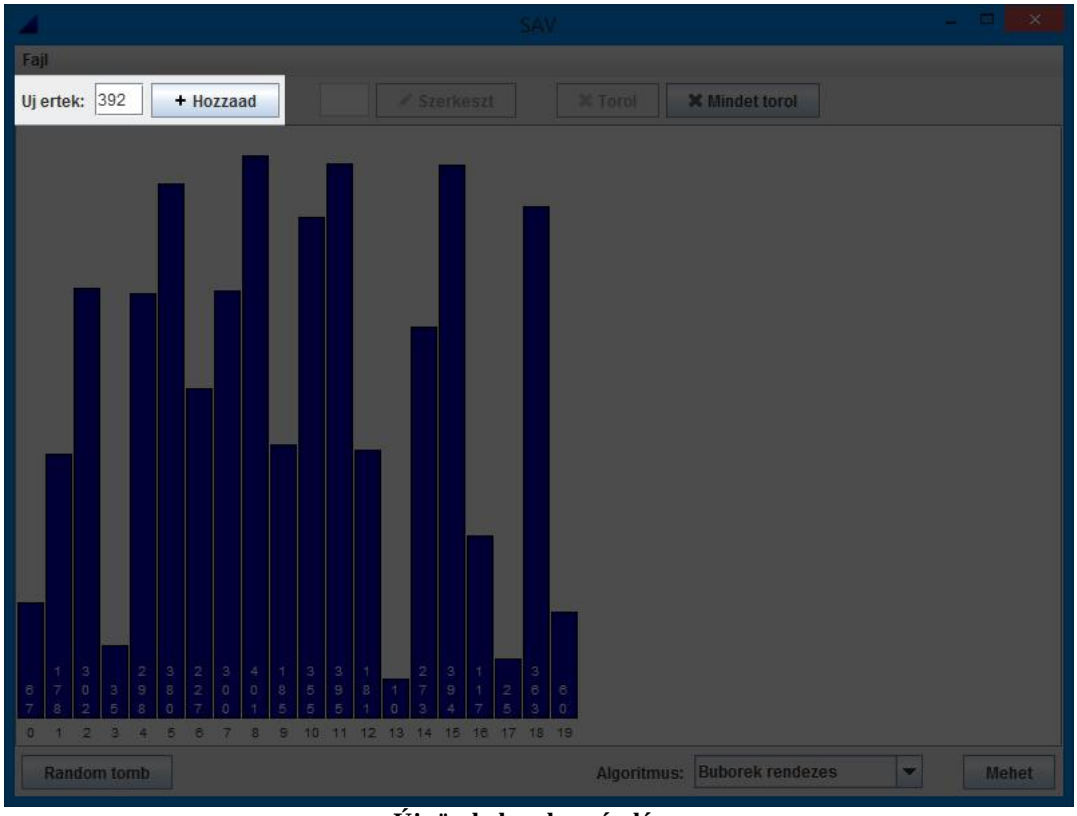
A szerkesztő felület részei.

Ahogy az ábra is mutatja négy nagyobb elkülöníthető részből áll:

1. A **szerkesztő területből**, amin a szemléltetés alapját alkotó oszlopok találhatóak. Minden oszlop egy-egy tömbelemet reprezentál. Minél nagyobb a tömbelem értéke, annál magasabb az oszlop. A tömbelem pontos értéke az oszlopon belül, alulra helyezve, függőlegesen olvasható. Az oszlopok alatt a tömbelem indexe olvasható, amely egyes rendezéseknél az algoritmus jobb követhetőségének érdekében került fel.
2. A **menüből**, ahol betölthetünk, illetve (ha a szemléltető felület aktív éppen) menthetünk rendező algoritmusokat.
3. A **felső szerkesztő panelből**, amin a fontosabb szerkesztői elemek helyezkednek el. Itt adódik lehetőség új oszlopokat helyezni a szerkesztő területre, azokat szerkeszteni és törölni.
4. Az **alsó szerkesztő panelből**, melyen egy random tömböt generáló gomb található, az algoritmus kiválasztásához szükséges lenyíló mező (ún. „combo box”), valamint a szemléltető felületre továbbító *Mehet* felirattal rendelkező gomb.

3.4.2. Új tömbelem hozzáadása

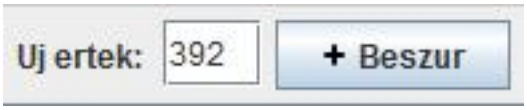
Új tömbelem hozzáadására a felső szerkesztő panelen van lehetőség. A művelet végrehajtásához szükséges input mező és hozzá tartozó gomb rögtön a panel bal szélén helyezkednek el.



Új tömbelem hozzáadása.

Az input mező alapértelmezetten tartalmaz egy random generált számot, de ha ez az érték nem szimpatikus, át lehet írni. A *Hozzáad* feliratú gomb megnyomását követően az új érték a szerkesztő területhez adódik, közvetlenül a legutolsó tömbelem után.

Új elem **beszúrására** (már meglévő oszlopok elé) is van lehetőség. Ehhez először egy kattintással ki kell jelölni azt az oszlopot, ami elé majd az új elem fog kerülni. A kijelölést követően a *Hozzáad* gomb helyét egy *Beszúr* feliratú gomb veszi át.



Beszúr gomb.

A *Beszúr* feliratú gomb megnyomását követően az új érték a szerkesztő területhez adódik, közvetlenül a kijelölt tömbelem elé.

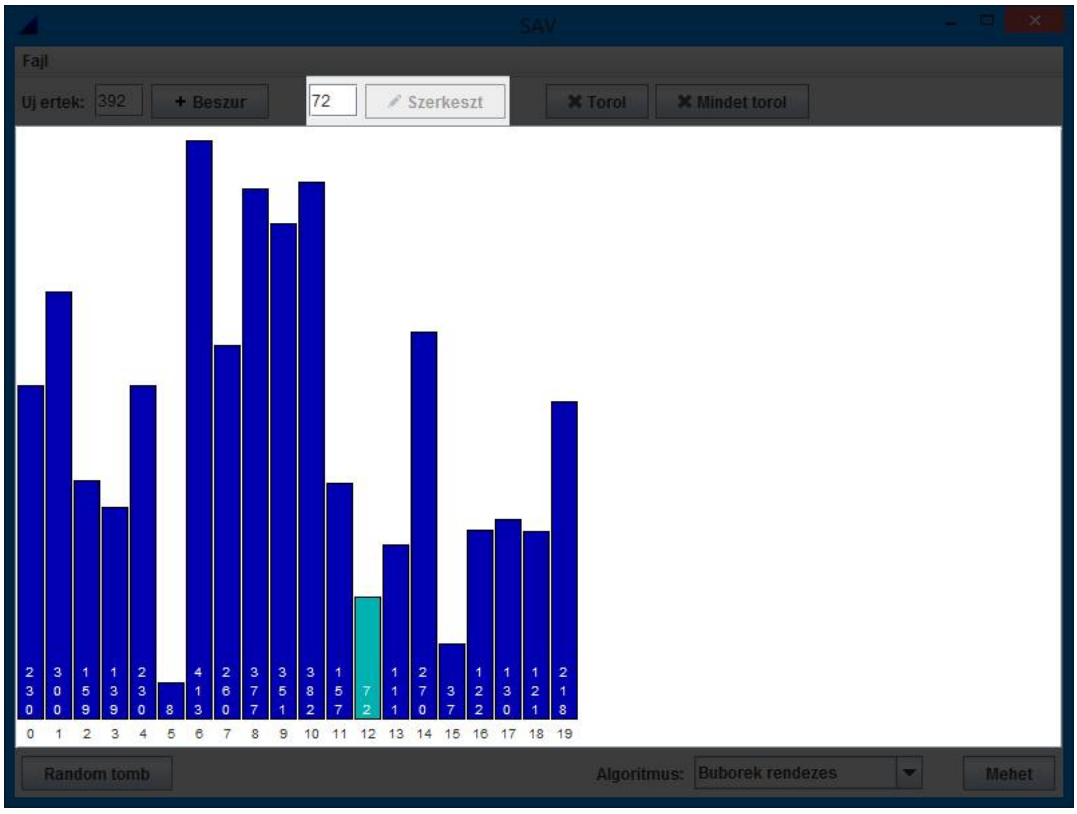
Az input mezőben engedélyezett számok: 0..420. Nem engedélyezett érték beírását a program a gomb **letiltásával** jelzi, így helytelen adatokat nem lehet bevinni.

Gomb letiltása helytelen adat esetén.

Maximum 50 tömbelem kerülhet a szerkesztő területre. Ennél több elem a szemléletesség megtartása érdekében nem támogatott.

3.4.3. Tömbelem szerkesztése

Tömbelem szerkesztésére szintén a felső szerkesztő panelen van lehetőség. A szerkesztő területen egy oszlopra kattintva tudjuk **kijelölni** a szerkeszteni kívánt elemet. A kijelölés alatt lévő oszlop az alapértelmezett sötétkék helyett egy világosabb kék színnel jelenik meg.



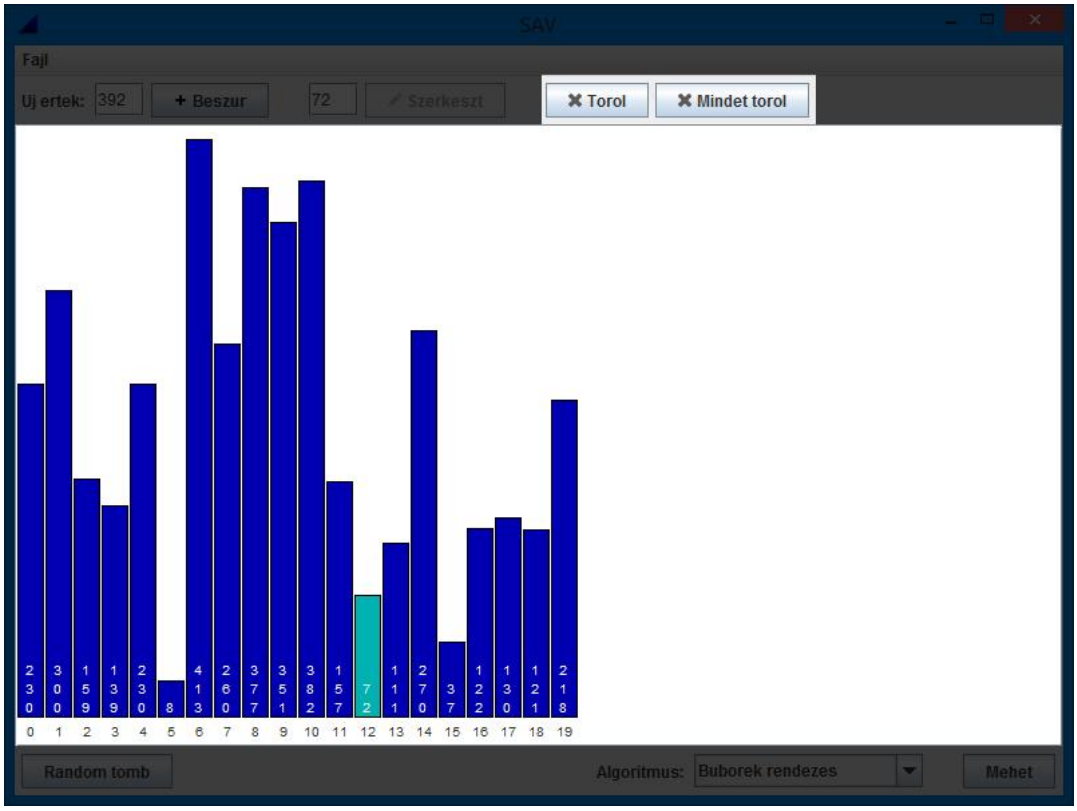
Tömbelem szerkesztése.

Miután kijelöltünk egy oszlopot a szerkesztéshez szükséges input mező aktívvá válik. Egy másik értéket beírva a gomb is aktiválódik, és a szerkesztést végre lehet hajtani.

Természetesen ebbe az input mezőbe is csak helyes adatok adhatóak meg. A helytelen adatokat továbbra is a gomb letiltásával jelzi a program.

3.4.4. Tömbelem törlése

A tömbelemek törléséhez szükséges gombok is a felső szerkesztő panelen kaptak helyet.

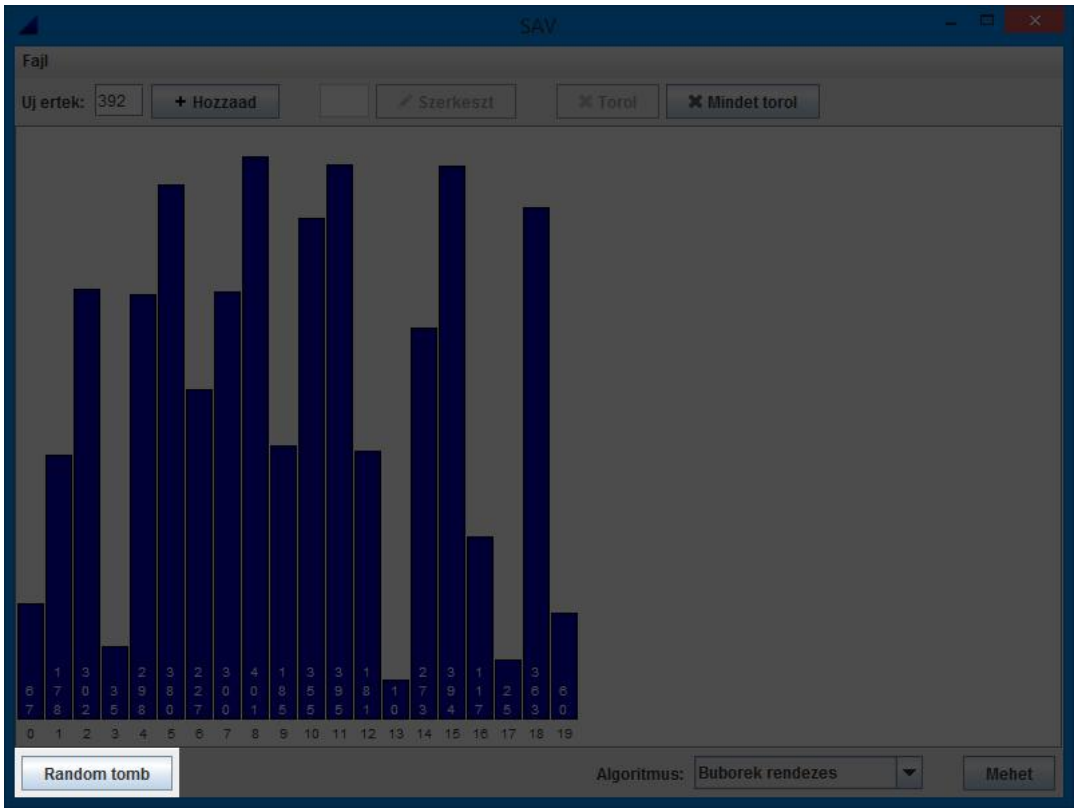


Tömbelem törlése.

A *Töröl* feliratú gomb egyetlen kijelölt oszlop törlésére ad lehetőséget, még a *Mindent torol* gomb az összes eddig létrehozott oszlop törlésére. A *Töröl* gomb akkor aktív, ha van kijelölt oszlop; a *Mindent töröl* gomb pedig, ha már legalább egy oszlop szerepel a szerkesztő területen.

3.4.5. Random tömb generálása

A random tömb generálásáért felelős gomb az alsó szerkesztő panelen kapott helyet, rögtön a panel bal szélén.



Random tömböt generáló gomb.

A gomb megnyomásával a program egy új, 20 elemből álló tömböt generál véletlenszerű értékekkel.

3.4.6. Algoritmus kiválasztása, és a szemléltető felületre lépés

A tömb módosításának befejezése után a **szemléltető felületre** az alsó szerkesztő panelen található, jobb oldalra helyezett *Mehet* feliratú gombbal lehet navigálni. A továbblépés előtt lehetőség van a rendező algoritmust kiválasztani, amit a gomb bal oldalán található „combo box”-al tehet meg a felhasználó. Az elemre való kattintás után megjelenik a hat rendezés, ami közül választani lehet.

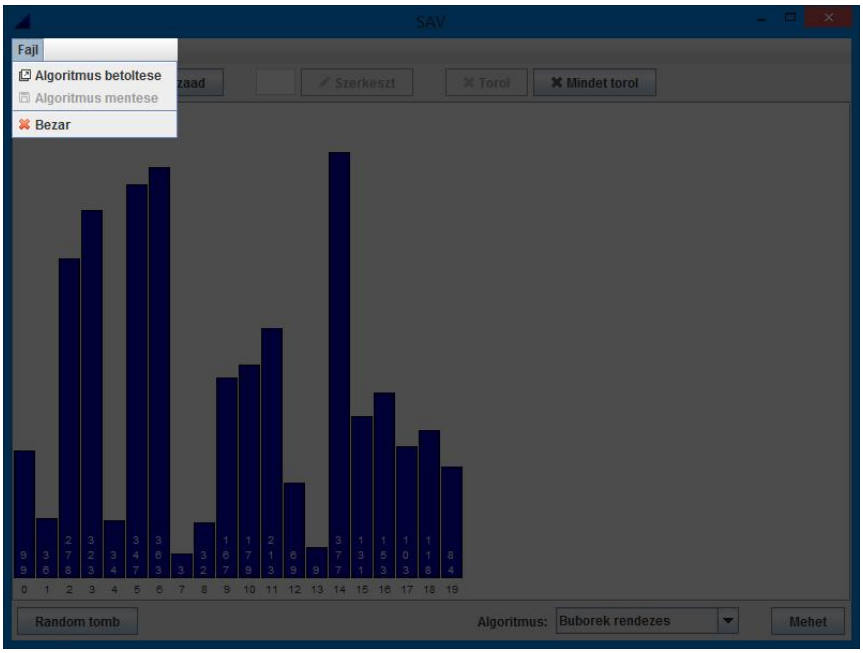
Algoritmus kiválasztása.

Ha nem választunk ki másik rendezést, alapértelmezetten a buborék rendezés fog betöltődni.

A *Mehet* gomb aktív, ha legalább 2 elemből álló tömböt hoztunk létre.

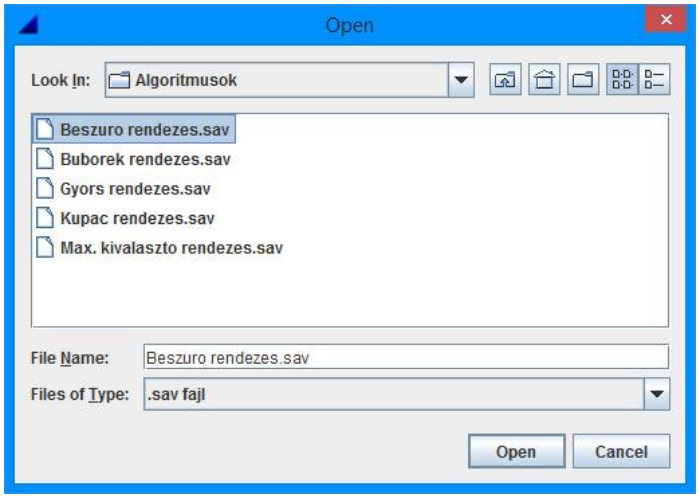
3.4.7. Algoritmus betöltése

Korábban elmentett algoritmusokat a **menüből** van lehetőség betölteni. A *Fájl* menüt kiválasztva megjelenik az *Algoritmus betöltése* opció.



Algoritmus betöltése.

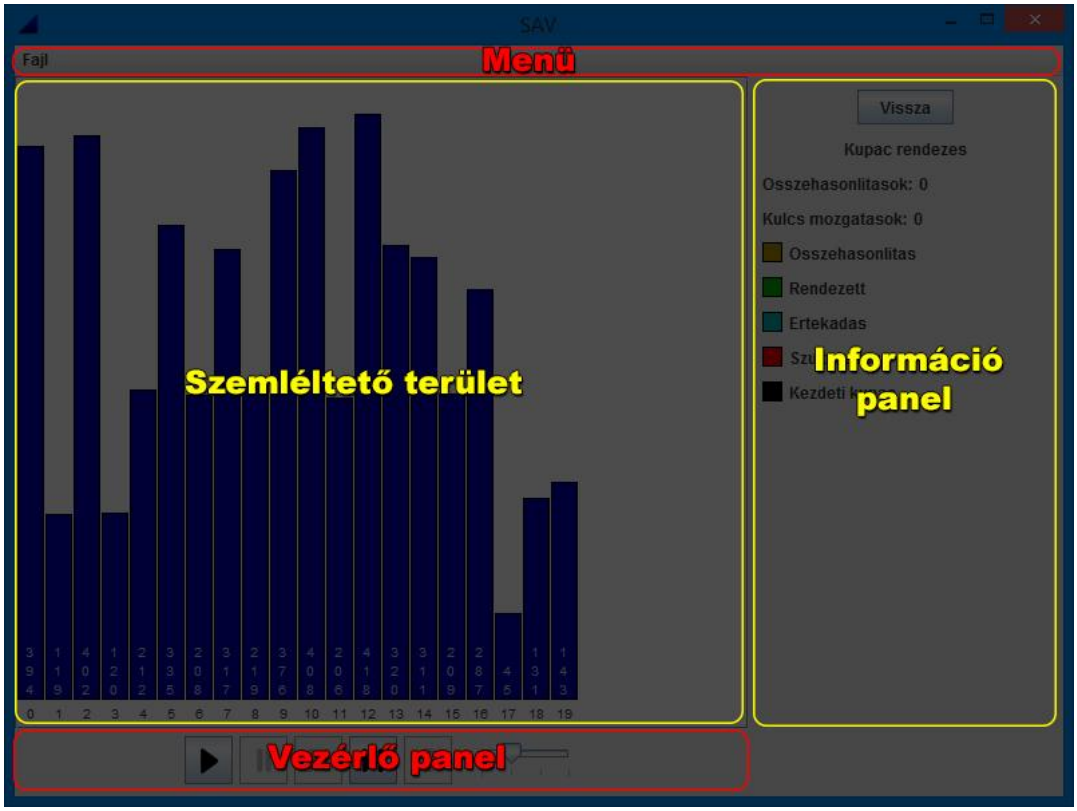
A menüelemet kiválasztva előugrik a **fájlkiválasztó ablak**, ahol a korábban mentett algoritmusok mappájába navigálva, kiválasztható az algoritmus.



Fájlkiválasztó ablak.

A program a könnyebb használhatóság érdekében **saját fájlkiterjesztést** használ, ami a három betűből álló *.sav* nevet viseli. Ennek előnyeként a fájlkiválasztó ablakban csak a releváns, *.sav* kiterjesztésű fájlok jelennek meg.

3.4.8. Szemléltető felület



A szemléltető felület részei.

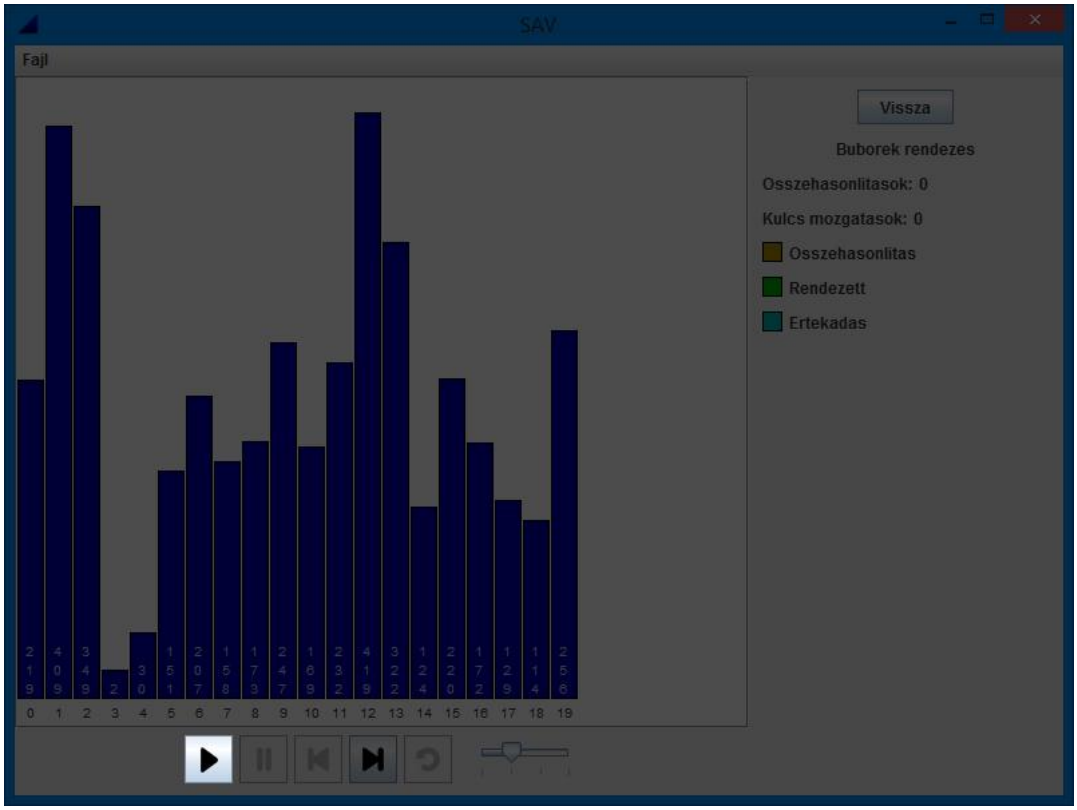
A szemléltető felületre a szerkesztés befejezése után lehet navigálni, ahol a felhasználó az elkészített tömbön hajthatja végre a kiválasztott rendező algoritmus lépéseit.

A szemléltető felület a szerkesztő felülethez hasonlóan négy nagyobb részből tevődik össze:

1. A **szemléltető területből**, amely megegyezik a szerkesztő felületen látható szerkesztő területtel, azzal a különbséggel, hogy ott kattintásra az oszlopokat ki lehetett jelölni. Erre itt már nincs lehetőség.
2. A **menüből**, ahol ezen a felületen aktiválódik az algoritmus mentése funkció. Így a létrehozott tömböt a felhasználó el tudja menteni, melyet később szintén a menüből vissza tud tölteni.
3. Az **információ panelből**, melyen a szerkesztő felületre visszalépő gomb, az algoritmus neve és az algoritmus lépéseinek megértését segítő színmagyarázat kapott helyet. Valamint az összehasonlítások és a kulcs mozgatások száma is itt olvasható, amelyeket a program lépésről-lépésre számon tart.
4. A **vezérlőpanelből**, melyen a felhasználónak lehetősége van az algoritmus léptetésére.

3.4.9. Algoritmus léptetése, vezérlőpanel ismertetése

Az algoritmus lépéseit kettő különböző módon lehet vezérelni. Választható a **folyamatos léptetés**, mely során a program bizonyos időközönként a következő lépésre lép. Az automatikus léptetés a médialejátszó alkalmazásokból ismert, „play” gombhoz hasonló *Folyamatos léptetés* gombbal indítható.



Folyamatos léptetést elindító gomb.

A **léptetés sebessége** lejátszás közben is állítható a vezérlőpanel jobb szélén található „csúszka” segítségével.

Léptetés sebességét állító csúszka.

Négy különböző sebesség választható, a nagyon lassútól a nagyon gyorsig terjedően. Alapértelmezetten a csúszka a 2. ponton áll, ami 1 másodperces időközönkénti léptetést jelent.

Az automatikus léptetést a *Szünet* gombra kattintva lehet megállítani.

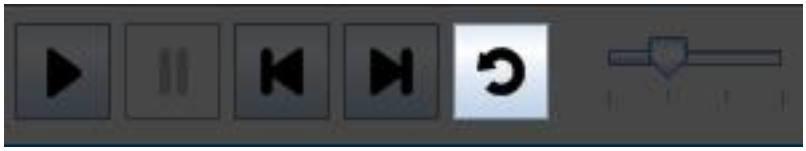
Szünet gomb.

Manuális léptetés esetén a 3. és 4. helyen lévő *Előző lepés*, illetve *Következő lépés* gombok használatosak.

Előző lépés és Következő lépés gomb.

Érelemszerűen a gombok megnyomásával a program az algoritmus előző, illetve következő lépésére lép.

Az **algoritmus elejére** való lépéshez nem szükséges az *Előző lépés* gombot számtalanszor megnyomni. Ehhez elég a vezérlőpanel utolsó gombjára, a *Vissza az elejére* gombra kattintani, ami azonnal visszaugrik a legelső lépéshez.

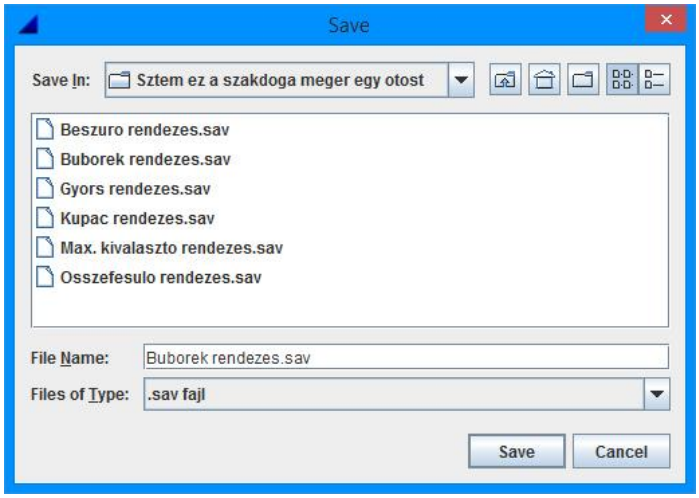


Vissza az elejére gomb.

3.4.10. Algoritmus mentése

Ahhoz, hogy **később** is megtekinthető legyen az algoritmus a jelenleg használt tömbbel, a felhasználónak lehetősége van **elmenteni** az adatokat egy fájlba. Ehhez a *Fájl* menüt kiválasztva, az *Algoritmus mentése* opciót kell választani.

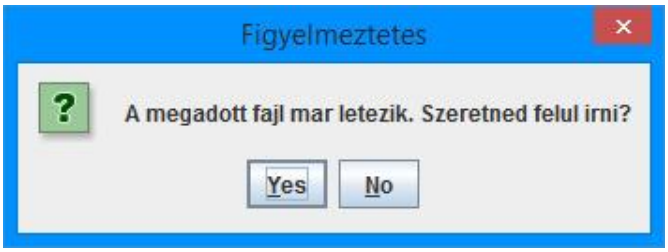
Ezt követően a fájlok mentésére szolgáló ablak jelenik meg, ahol a kívánt mappába navigálhatunk.



Fájlok mentésére szolgáló ablak.

A fájl nevének a helyén alapértelmezetten a menteni kívánt algoritmus neve szerepel. Természetesen ez igény szerint változtatható, azonban **fontos**, hogy *.sav* kiterjesztése legyen a fájlnak, ugyanis, ha más formátumban kerül mentésre, később a betöltésnél **nem fog megjelenni** a fájlkiválasztó ablakban!

Amennyiben a kiválasztott fájl már létezik, a program egy figyelmeztető ablakkal értesíti a felhasználót, melyen kiválasztható, hogy felül szeretné-e írni a régi fájlt.



Figyelmeztető ablak fájl felülírásánál.

4. FEJLESZTŐI DOKUMENTÁCIÓ

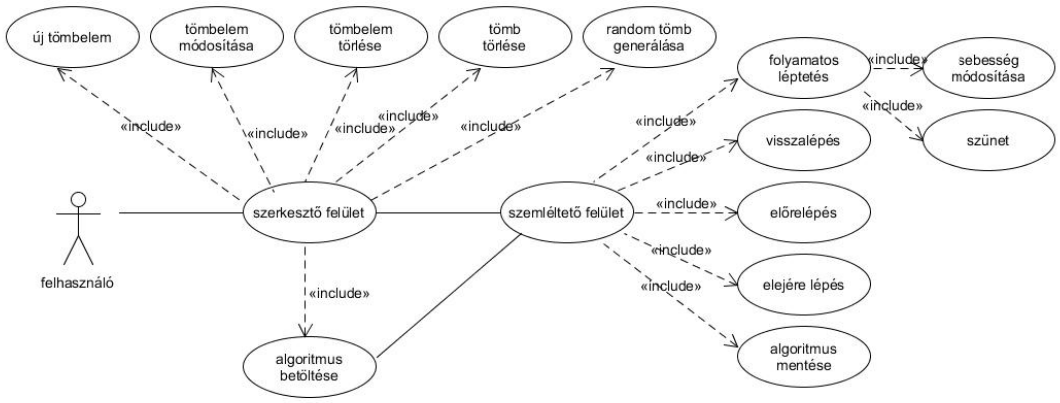
4.1. Tervezés

4.1.1. Követelményanalízis

- A legfontosabb feladat az algoritmusok vezérlésének megvalósítása. A programnak képesnek kell lennie lépésről-lépésre irányítani az algoritmus működését, a visszalépés lehetőségét is magában foglalva.
- Egy átlag felhasználó mindennapos számítógép használat közben valószínűleg nem gyakran találkozik terminállal / parancssorral. Éppen ezért az algoritmusok vezérelésének megvalósítása után, ezt grafikusan is meg kell jeleníteni a felhasználó számára.
- A szemléltetésnek lényegre törekvőnek, áttekinthetőnek kell lennie. Mivel a rendező algoritmusok különböző módon közelítik meg a rendezett tömb előállításának folyamatát, a szemléltetést egyedien, egyes algoritmusokra szabva kell megvalósítani.
- Lehetőséget kell biztosítani a felhasználónak egyedi tömböket létrehozni, melyeket tetszőlegesen módosíthat, elmenthet, újra betölthet. Mindezt időigényes almenükben való kutakodás nélkül kell a felhasználó rendelkezésre bocsátani. A grafikus felületnek áttekinthetőnek és egyértelműnek kell maradnia.
- A programnak jól karbantarthatónak kell lennie. Igény szerint később új rendező algoritmusok implementálása ne okozzon problémát.

4.1.2. Felhasználói esetek

A programot egyidejűleg egyetlen felhasználó használja, aki hozzáfér annak minden funkciójához. A következő UML diagramon látható az összes felhasználói eset.



Felhasználói eset diagram.

4.1.3. Megvalósítási terv

A program elkészítésére a **Java** programozási nyelv adott volt. Korábbi Java programozással eltöltött éveim alatt megismerkedtem az **objektumorientáltság** előnyeivel. A program egységeinek osztályokba szervezésével megfelelő eszközöket biztosít a karbantarthatóság megteremtésére. Azonban a nyelv melletti a döntésem leginkább a **grafikus szoftver** létrehozását támogató beépített könyvtárak indokolták. A *javax.swing* könyvtár minden eszközt biztosít a programozó számára, ami grafikus alkalmazás fejlesztése során szükséges lehet. A *swing* könyvtár nem tartalmaz platform-specifikus kódokat, ezért jó eséllyel minden platformon egységes kinézetet kap a felhasználó.

Az algoritmusok vezérlésének megvalósítására a kezdeti fázisban több megoldás is született, melyek a későbbiek folyamán **nem** bizonyultak elég **hatékonynak**.

Az első ötlet egy *nextStep* nevű függvény létrehozása volt, amelyet lefuttatva a rendező algoritmus **egyetlen műveletet** hajt végre. A művelet összehasonlítás vagy értékadás lehet. A függvény *true* vagy *false* értékekkel tér vissza annak jelzésére, hogy az algoritmus befejeződött, vagy még nem futott le teljesen. Egyetlen műveletet csakis akkor fog végrehajtani a függvény, ha a ciklusokat szétbontjuk **állapotokra** és **elágazásokra**, a ciklus léptetéséért felelős változókat pedig osztály szintre emeljük, hogy később hozzáférhessünk a függvény újbóli futtatásával.

```
public boolean nextStep() {
    if (j < 1) {
        return false;
    }

    if (i > j-1) {
        i = 0;
        --j;
    } else {
        sumCompare += 1;
        if (t[i] > t[i+1]) {
            sumSwap += 1;
            int temp = t[i];
            t[i] = t[i+1];
            t[i+1] = temp;
        }
        ++i;
    }

    return true;
}
```

Buborék rendezés kezdeti implementációja.

A buborék rendezés implementációja kisebb gondolkodás után értelmezhető, azonban a következő ábrán látható beszúró rendezésről ez már csak kisebb túlzásokkal mondható.

```
public boolean nextStep() {
    if (status == Status.OUTER) {
        if (i >= t.length) return false;
        if (t[i-1] > t[i]) {
            x = t[i];
            t[i] = t[i-1];
            j = i-2;
            status = Status.INNER;
        } else {
            ++i;
        }
    } else {
        if (j >= 0 && t[j] > x) {
            t[j+1] = t[j];
            --j;
        } else {
            t[j+1] = x;
            status = Status.OUTER;
            ++i;
        }
    }

    return true;
}
```

Beszúró rendezés kezdeti implementációja.

A beszúró rendezésnél **számon kell tartani**, hogy az előző lépésben a külső vagy a belső ciklusban fejezte be a működését a függvény. Az algoritmus kódja ezzel teljesen **átláthatatlan** lett. A másik probléma, hogy a buborék és beszúró rendezés nem tartalmaz rekurziót. A rekurzív rendező algoritmusok *nextStep* függvényét megírni szinte **lehetetlen** feladatnak tűnt. Éppen ezért az ötlet **elvetetésre került**.

Egy olyan módszerre van szükség tehát (az előzőekből tanulva), ami a rendező algoritmusok kódját csak olyan mértékben módosítja, amivel az még áttekinthető, felismerhető lesz.

A megoldást ezután a **többszálúságban** láttam. Az ötlet az volt, hogy a rendezést végrehajtó függvényt külön szálon futtatjuk, majd a szál futását lépésenként **blokkoljuk** egy *waitForNextStep* nevű metódussal.

```
void waitForNextStep() throws InterruptedException {
    status = Status.SLEEP;
    while(status == Status.SLEEP) {
        Thread.sleep(10);
    }
}
```

waitForNextStep metódus.

A szál futásának engedélyezését a másik szálból **vezéreljük**, amihez csak egy adattagot kell módosítani a rendezést tartalmazó osztályban.

```
public void execute() throws InterruptedException {
    for (int j = t.length-1; j >= 1; --j) {
        for (int i = 0; i <= j-1; ++i) {
            if (t[i] > t[i+1]) {
                int _x = t[i];
                t[i] = t[i+1];
                t[i+1] = _x;
                waitForNextStep();
            }
        }
        System.out.println(toString());
    }
}
```

A buborék rendezés kódja waitForNextStep metódussal.

Az előző ábrán látható, hogy a buborék rendezés kódja így csak egyetlen sorral lett kiegészítve.

Azonban az algoritmus előző lépéseit ezzel a módszerrel sem tudjuk megtekinteni.

Végül a többszálúság ötlete is **elvetésre került**, ugyanis az előző probléma mellett az algoritmus folyamatos léptetését nem volt képes követni a külön szál. Ha kicsit nagyobb terhelés jutott éppen a processzorra az egyik időzítő lekéste a másikat, így **megállt** léptetés.

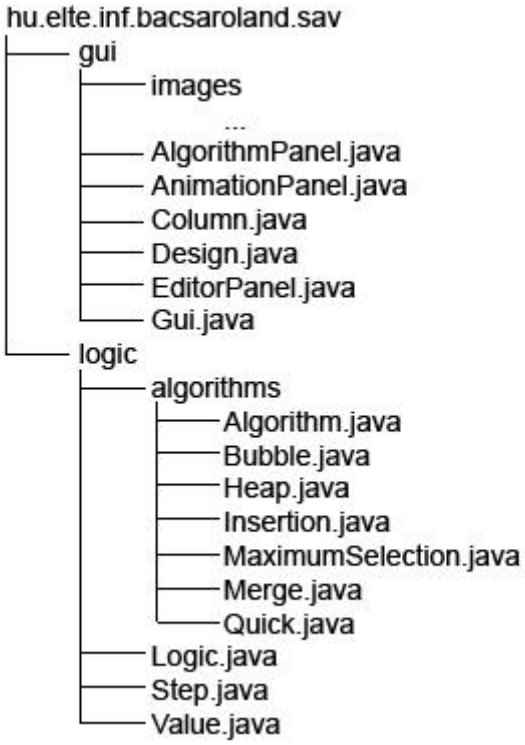
A megoldást a *Step* osztály bevezetése jelentette. Segítségével könnyen lemásolhatóak az algoritmust tartalmazó osztály (*Algorithm*) adattagja. Amint a felhasználó a szemléltető felületre érkezik, a rendezés lefut a háttérben. A rendezés futása alatt a program (az *Algorithm* osztályban definiált függvények segítségével) feltölt egy *Step* listát az algoritmus lépéseivel. Ez a program működését **nem lassítja**, ugyanis a maximális 50 elemű tömbön még minden rendezés képes a másodperc töredék része alatt lefutni. A feltöltött lista elemeit később **bármilyen irányú** léptetésre lehet használni.

4.1.4. A program felépítése

A programot **kétrégetű** architektúrában valósítjuk meg. Ennek megfelelően a program kettő nagyobb csomagra van osztva.

A *gui* csomag, mely a felhasználói interakciókat kezelő –, valamint a grafikus felület létrehozásáért felelős osztályokat tartalmazza. Továbbá ebben a csomagban találhatóak a program által használt képfájlok is.

A *logic* csomag, melyben a program logikájáért felelős –, valamint a rendező algoritmusokat implementáló osztályok kapnak helyet.

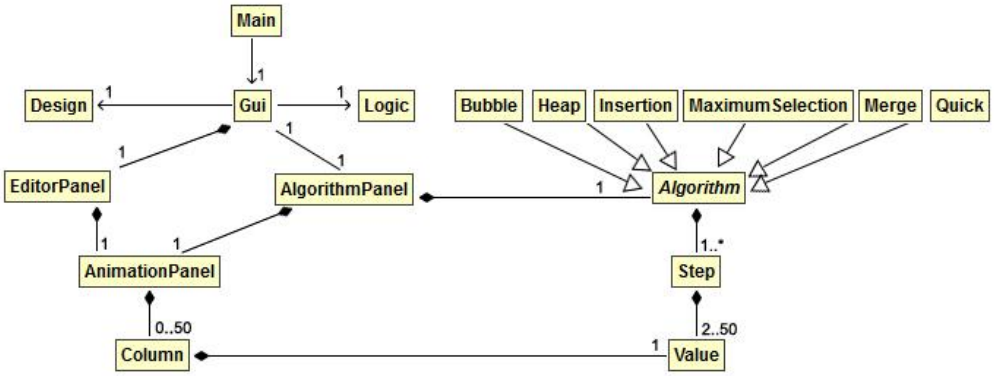


A program felépítése.

4.2. Megvalósítás

4.2.1. Az osztályok kapcsolata

Az objektumorientált programozás elveit követve a metódusokat, függvényeket és adattagokat osztályokba szervezve tároljuk. Az osztályok közötti kapcsolat az alábbi UML osztálydiagramon látható.



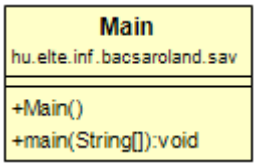
UML osztálydiagram.

Az osztályok és a hozzájuk tartozó metódusok, függvények és adattagok részletes leírása lentebb található.

4.2.2. Az osztályok részletes leírása

Main osztály:

A program **belépési pontja** (*main* metódus) ebben az osztályban található.



Main osztály UML diagramja.

main metódus: létrehoz egy példányt a *Main* osztályból.

Konstruktor: létrehoz egy példányt a grafikus felületért felelős *Gui* osztályból, majd megjeleníti azt.

Gui osztály:

A grafikus felhasználói felületet megjelenítéséért, módosításáért felelős osztály. Alap szerkezetét a *javax.swing.JFrame* osztály adja, melyben az **ablak kezeléséhez** elengedhetetlen osztályelemek már definiálva vannak.

Gui osztály UML diagramja.

Adattagok:

- logic: a *Logic* osztály egyetlen példánya.
- ICON_IMAGE, MENU_LOAD_ICON, MENU_SAVE_ICON, MENU_EXIT_ICON: a menü elemeinek ikonjai.
- frame: a program főablaka.
- centerPanel: az aktuálisan megjelenített panel, mely lehet *AlgorithmPanel*, vagy *EditorPanel* típusú. A panelek közötti váltást a *changeCenterPanelToEditor*, valamint a *changeCenterPanelToAlgo* nevű metódusok végzik.
- editorPanel: az *EditorPanel* osztály példánya, amely a program szerkesztőfelületének megjelenését adja. Az osztály tárolja a felhasználó által létrehozott tömböt, ezért a *centerPanel* váltásakor a régi állapot töltődik vissza.

- saveAlgorithm: a menü egy komponense. A használata az *AlgorithmPanel* osztálytól függ, ezért adattagként szerepel az osztályban, hogy később módosítható legyen a viselkedése.

Konstruktor: meghívja az *init* metódust.

Függvények és metódusok:

- init: létrehozza a program ablakát és beállítja az osztály adattagjait a *create** nevű függvények segítségével.
- createMenuBar: létrehozza a menüt, valamint a menüelemek viselkedését definiálja.
- enableSaveAlgorithm: a menüben található *Algoritmus mentése* elemet engedélyezi, valamint viselkedést rendel hozzá. A metódust külső osztályok hívhatják, ugyanis a viselkedése függ a paraméterként kapott rendező algoritmus típusától.
- disableSaveAlgorithm: letiltja a menüben található *Algoritmus mentése* elemet. Erre akkor van szükség, amikor a felhasználó a szerkesztőfelületen tartózkodik.
- createSaveFileChooser: visszatérési értékként a fájlok mentési helyének kiválasztására hoz létre ablakot.
- createLoadFileChooser: *.sav* kiterjesztésű fájlok betöltését teszi lehetővé. Visszatérési értéke a fájlkiválasztó ablak.
- changeCenterPanelToEditor: a *centerPanel*-t állítja *editorPanel*-re. Ezt úgy teszi, hogy a jelenlegi *centerPanel*-t eltávolítja az ablakról, majd az *editorPanel*-t kapja értékül, amit az ablak validálása után megjelenít.
- changeCenterPanelToAlgo: a *centerPanel*-t állítja *AlgorithmPanel* típusúra. Az *editorPanel* adattag értéke megmarad, így később visszaállítható a régi állapot.
- setComponentWidth, setComponentHeight: osztályszintű metódusok, mely a paraméterként kapott grafikus komponens szélességének / magasságának beállításáért felelősök.
- showErrorMessage: a paraméterként kapott hibaüzenetet jeleníti meg egy új ablakban.

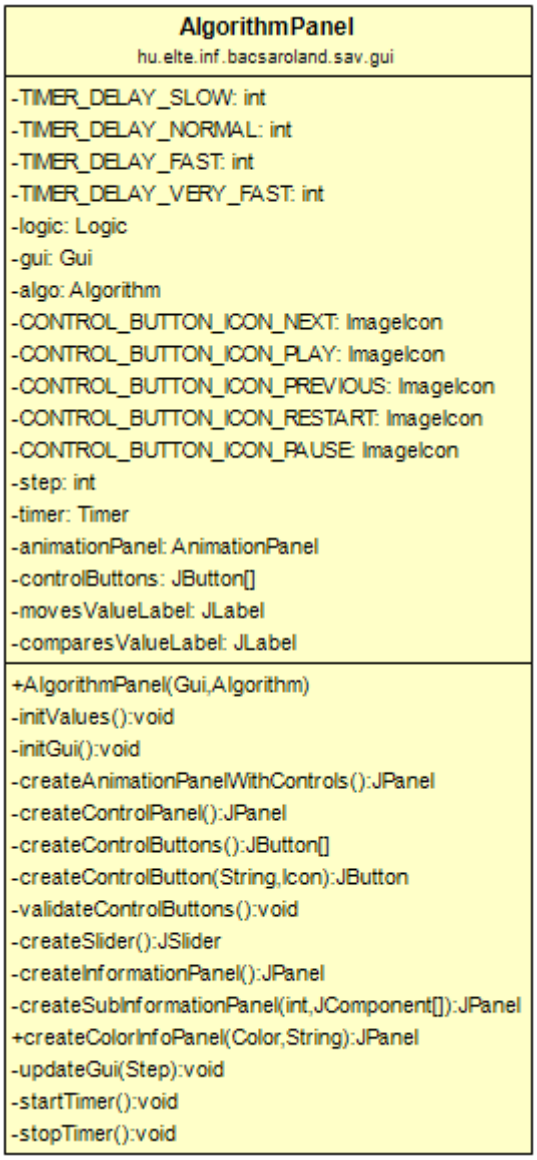
AlgorithmPanel osztály:

A kiválasztott rendező algoritmus **szemléltetését** végző osztály, mely 3 nagyobb részpanelből épül fel:

1. a legnagyobb területet foglaló *animationPanel*-ből,

2. az *animationPanel*-től jobb oldalra található *informationPanel*-ből, melyen a rendező algoritmus adatai, valamint a szemléltetéshez használt színek magyarázata található,
3. legvégül pedig a panel legalján található *controlPanel*-ből, mellyel a rendező algoritmus vezérelhető.

Az osztály a *javax.swing.JPanel* osztályból származik, mely lényegében egy konténer, ami különböző grafikai elemeket tárol, változtatható elrendezéssel és kinézettel.



AlgorithmPanel osztály UML diagramja.

Adattagok:

- TIMER_DELAY_SLOW, TIMER_DELAY_NORMAL, TIMER_DELAY_FAST, TIMER_DELAY_VERY_FAST: előre definiált értékek, melyek a rendező algoritmus folyamatos léptetésének sebességét adják meg.
- logic: a *Logic* osztály egyetlen példánya.
- gui: a *Gui* osztály példánya. Ezen adattagon keresztül van lehetőség visszalépni a szerkesztő felületre.
- algo: maga a rendező algoritmus, melyet szemléltet az osztály.
- CONTROL_BUTTON_ICON_NEXT, CONTROL_BUTTON_ICON_PLAY, CONTROL_BUTTON_ICON_PREVIOUS, CONTROL_BUTTON_ICON_RESTART, CONTROL_BUTTON_ICON_PAUSE: a *controlPanel*-en található gombok ikonjai.
- step: a rendező algoritmus aktuális lépésének sorszámát tárolja.
- timer: a folyamatos léptetést lehetővé tevő *javax.swing.Timer* osztály példánya. Létrehozásakor definiálni kell a viselkedését (paraméterként egy *ActionListener*-t kap), melyet a beállított késleltetéssel (TIMER_DELAY_SLOW / TIMER_DELAY_NORMAL / TIMER_DELAY_FAST / TIMER_DELAY_VERY_FAST) folyamatosan végrehajt.
- animationPanel: a tömbelemeket reprezentáló oszlopokat kirajzoló panel.
- controlButtons: a *controlPanel*-en található vezérlő gombokat tároló tömb.
- movesValueLabel, comparesValueLabel: az algoritmus értékadásainak és összehasonlításainak számát jelenítik meg.

Konstruktor: beállítja az adattagokat az *initValues* és az *initGui* metódusok segítségével, legenerálja a rendező algoritmus lépéseit, engedélyezi a menüben az *Algoritmus mentése* menüelemet.

Függvények és metódusok:

- initValues: inicializálja a *step* és a *timer* adattagot.
- initGui: az *AlgorithmPanel* kinézetét definiálja a *create** függvények segítségével.
- createAnimationPanelWithControls: létrehozza az *animationPanel*-t és a *controlPanel*-t.

- createControlPanel: létrehozza a *controlPanelt* a rajta található vezérlőelemekkel.
- createControlButtons: a *controlPanel*-en található gombokat hozza létre.
- createControlButton: a *controlPanel*-en található gombok egy példányának létrehozásáért felelős függvény, mely beállítja a gomb kinézetét a paraméterként kapott *String* és *Icon* változók alapján.
- validateControlButtons: a rendező algoritmus aktuális állapota alapján letiltja / kattinthatóvá teszi a *controlPanel*-en található gombokat.
- createSlider: a *controlPanel*-en található „csúszka” (*javax.swing.JSlider*) létrehozásáért és beállításáért felelős függvény. Ennek a segítségével lehet a *timer* adattag késleltetését átállítani a 4 előre definiált érték egyikére.
- createInformationPanel: a *informationPanel* létrehozásáért és beállításáért felelős függvény. Az *informationPanel*-en találhatóak a rendező algoritmus összehasonlításainak- és értékadásainak száma, valamint a szemléltetéshez használt színek magyarázata.
- *createSubInformationPanel*: az *informationPanel*-en található komponensek elrendezését segítő függvény. Paraméterként a kívánt rendezést definiáló *java.awt.FlowLayout* osztály egy osztályszintű adattagját, valamint tetszőleges számú grafikus komponenst kap. Visszatérési értéke egy panel, melyen a komponensek a megadott elrendezésben helyezkednek el.
- createColorInfoPanel: az *informationPanel*-en található színek és hozzá tartozó rövid magyarázat létrehozásáért felelős függvény.
- updateGui: a grafikus felület frissítését végző metódus, mely beállítja az *animationPanel* oszlopait, a rendező algoritmus adatait és a vezérlő gombokat a paraméterként kapott *Step* változó alapján.
- startTimer, stopTimer: a *timer* adattag elindításáért / leállításáért felelős metódusok.

EditorPanel osztály:

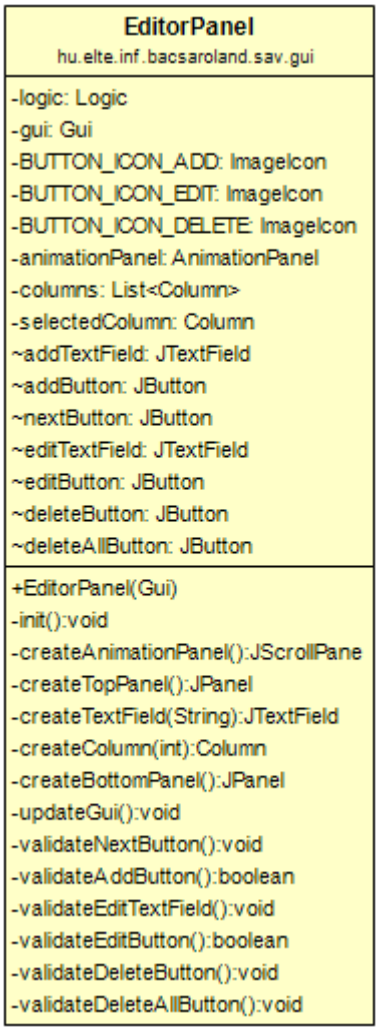
A felhasználó ennek az osztálynak a segítségével hozhat létre **új tömböt**, melyen kiválaszthatja, hogy milyen rendező algoritmust szeretne futtatni. Három nagyobb panelből épül fel:

1. egy felső *topPanel*-ből, melyen a szerkesztő gombok, illetve a hozzájuk tartozó input mezők helyezkednek el,

2. egy középen elhelyezkedő *animationPanel*-ből, melyen a tömb elemeit reprezentáló oszlopok helyezkednek el,
3. egy alsó *bottomPanel*-ből, melyen kiválasztható a rendező algoritmus, és véglegesíthető a szerkesztés.

A szerkesztés befejezésekor megnyomott *Mehet* gomb után a létrehozott tömböt az *AlgorithmPanel* osztály példánya fogja kezelni.

Az osztály a *javax.swing.JPanel* osztályból származik.



EditorPanel osztály UML diagramja.

Adattagok:

- logic: a *Logic* osztály egyetlen példánya.
- gui: a *Gui* osztály példánya. Ezen adattagon keresztül van lehetőség továbblépni az *AlgorithmPanel*-re, mely a rendező algoritmus működését szemlélteti.
- BUTTON_ICON_ADD, BUTTON_ICON_EDIT, BUTTON_ICON_DELETE: a szerkesztő gombok ikonjai.

- columns: egy lista, amely a létrehozott oszlopokat tárolja. Ennek a listának az elemeit rajzolja ki az osztály az *animationPanel*-re.
- selectedColumn: az aktuálisan kijelölés alatt lévő oszlop. Az adattag értékét az oszlopokhoz rendelt *ActionListener*-ek változtathatják, melyek az oszlopra való kattintáskor futnak le.
- addTextField: input mező melyben a hozzáadni kívánt oszlop (tömb elem) értékét adhatjuk meg. A mező helyességét a *validateAddButton* nevű függvény ellenőrzi, mely helytelen érték esetén letiltja a hozzá tartozó *addButton* gombot.
- addButton: az *addTextField*-ben megadott értékű oszlopot helyez el az *animationPanel*-en. A gomb csak akkor kattintható, ha a *validateAddButton* nevű függvény (amely ellenőrzi az *addTextField* helyességét) *true* értékkel tért vissza.
- nextButton: a *bottomPanel*-en található *Mehet* gomb, amely a létrehozott tömb és a kiválasztott algoritmus alapján betölti az *AlgorithmPanel*-t. A gomb kattinthatóságáról a *validateNextButton* nevű metódus gondoskodik.
- editTextField: input mező, mely a kijelölt oszlop értékét kapja meg. Itt lehet megadni neki új értéket, melyet a hozzátartozó *editButton* segítségével lehet menteni. A mező szerkeszthetőségéről a *validateEditTextField* metódus gondoskodik.
- editButton: az *editTextField*-ben megadott értéket adja a kijelölés alatt lévő, *selectedColumn* adattagban tárolt oszlopnak. Kattinthatóságáról a *validateEditButton* nevű függvény gondoskodik.
- deleteButton: eltávolítja a kijelölés alatt lévő, *selectedColumn* adattagban tárolt oszlopot. Kattinthatóságáról a *validateDeleteButton* nevű metódus gondoskodik.
- deleteAllButton: eltávolítja az összes oszlopot az *animationPanel*-ről. Kattinthatóságáról a *validateDeleteAllButton* nevű metódus gondoskodik.

Konstruktor: a paramétere alapján inicializálja a *gui* adattagot, és meghívja az *init* metódust.

Függvények és metódusok:

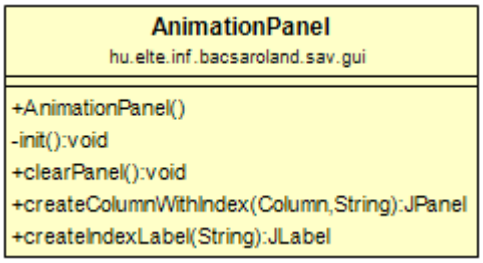
- init: inicializálja az osztály adattagjait a *create** nevű függvények segítségével, valamint definiálja a kinézetet. Ezen kívül generál egy 20 elemű random tömböt, amiből oszlopokat hoz létre, majd meghívja az *updateGui* nevű metódust.
- createAnimationPanel: létrehoz egy példányt az *AnimationPanel* osztályból. Visszatérési értéke: a létrehozott panel, amely görgethető.

- createTopPanel: létrehozza a *topPanel*-t, valamint az ezen található különböző szerkesztő input mezőket és gombokat, amiknek definiálja a viselkedésüket is.
- createTextField: egy input mezőt hoz létre, melybe alapértelmezetten a paraméterként kapott *String*-ben tárolt szöveg kerül megjelenítésre. Ezen kívül beállítja a mező magasságát.
- createColumn: egy oszlopot hoz létre, a paraméterként megadott *String* érték szerint, ami az oszlop méretét és rajta olvasható szám értéket adja meg. Ezen kívül egy *ActionListener*-t rendel az oszlophoz, ami segítségével kattintásra ki lehet jelölni.
- createBottomPanel: létrehozza az ablak alján látható *bottomPanel*-t. Ez a panel egy baloldali és egy jobboldali részpanelből áll. A baloldalira elhelyezi a random tömböt generáló *randomButton*-t. A jobboldalira az algoritmus kiválasztásához szükséges lenyíló dobozt (*javax.swing.JComboBox*), valamint a *Mehet* gombot, amely segítségével szemléltető felületre (*AlgorithmPanel*) léphet a felhasználó.
- updateGui: a grafikus felület frissítését végző metódus, mely kirajzolja az *animationPanel* oszlopait, majd validálja az input mezőket és a gombokat a *validate** nevű metódusok (és függvények) segítségével.
- validateNextButton: a *Mehet* gomb kattinthatóságát állító metódus. Ha 2-nél több oszlopot hozott létre a felhasználó, akkor a gombot engedélyezi.
- validateAddButton: az új oszlop hozzáadását végző gomb kattinthatóságát állítja. Ha az *addTextField* input mezőbe beírt érték kisebb, mint a maximális érték, nem negatív és az oszlopok száma kevesebb mint 50, akkor a visszatérési érték *true*, ellenben *false*.
- validateEditTextField: az oszlopok értékeinek szerkesztéséhez szükséges input mezőt engedélyezi, illetve tiltja le. Akkor engedélyezi, ha a felhasználó már kiválasztott egy oszlopot.
- validateEditButton: az oszlopok szerkesztését véglegesítő gomb engedélyezéséért és letiltásáért felelős. A gombot akkor engedélyezi, ha a felhasználó kiválasztott egy oszlopot, az *editTextField* mezőbe beírt érték kisebb, mint a maximális érték, nem negatív, valamint nem egyezik meg a korábbi értékével az oszlopnak. Ha teljesül minden feltétel a visszatérési értéke a függvénynek *true*, ellenben *false*.

- validateDeleteButton: a kijelölt oszlop törlését végző gomb engedélyezéséért és letiltásáért felelős metódus. Ha van a felhasználó által kijelölt oszlop, a gombot engedélyezi, ellenben letiltja.
- validateDeleteAllButton: az összes oszlop törlését végző gomb engedélyezéséért és letiltásáért felelős metódus. Ha az oszlopok száma nagyobb, mint 0, a gombot engedélyezi, ellenben letiltja.

AnimationPanel osztály:

Az *EditorPanel* és az *AlgorithmPanel* legnagyobb területét lefoglaló panel, melyen a tömb elemeit képviselő oszlopok, illetve az oszlopok indexei találhatóak. Ezt a panelt első sorban a kód duplikáció elkerülése érdekében kellett osztály szintre emelni. Az osztály a *javax.swing.JPanel* osztályból származik.



Az AnimationPanel osztály UML diagramja.

Konstruktor: meghívja a szülő osztály konstruktorát, majd az *init* metódust.

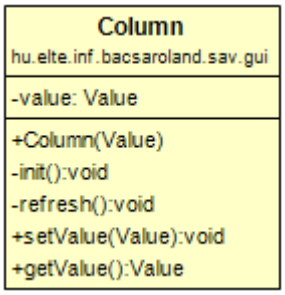
Függvények és metódusok:

- init: a panel elrendezését és hátterét állítja be.
- clearPanel: külső osztályok által hívható metódus, mely eltávolít minden elemet a panelről.
- createColumnWithIndex: a paraméterként kapott oszlop és index alapján egy panelt hoz létre, melyet el tud tárolni az osztály. Külső osztályok által hívható függvény.
- createIndexLabel: a paraméterként kapott index alapján egy előre meghatározott kinézetű feliratot hoz létre. A *createColumnWithIndex* függvény segédfüggvénye.

Column osztály:

A rendező algoritmusok szemléltetéséhez szükséges osztály. A *javax.swing.JButton* osztályból származik, mely egy **nyomógombot** definiál, minden szükséges műveletével és grafikus elemével. A *Column* osztály segítségével jeleníthetőek meg szemléletesen a

rendező algoritmusok által használt tömb elemei **oszlop** formában. Ez úgy érhető el, hogy a gomb fix szélességet, valamint a tárolt értéktől függő magasságot kap.



A Column osztály UML diagramja.

Adattagok:

- value: a *Value* szám és szín adattagokat tartalmazó osztály példánya. A számértékétől függ a gomb magassága.

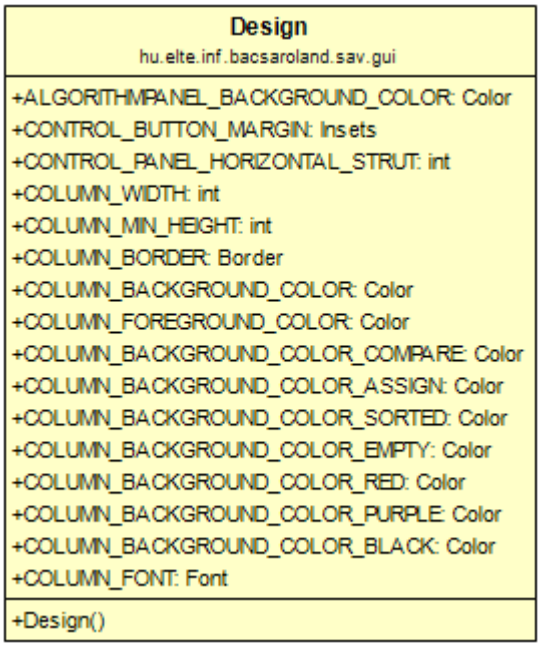
Konstruktor: a gomb kinézetét definiálja a *refresh* metódus segítségével.

Függvények és metódusok:

- refresh: az osztály által tárolt *value* adattag megváltozásakor a gomb megjelenésének is változnia kell. A metódus segítségével a gomb kinézete módosítás után is a tárolt értékét tükrözi.
- setValue: a paraméterként kapott változóra cseréli az osztályban tárolt *value* adattagot, majd meghívja a *refresh* metódust.
- getValue: visszatérési értéke a *private* láthatósággal rendelkező *value* adattag.

Design osztály:

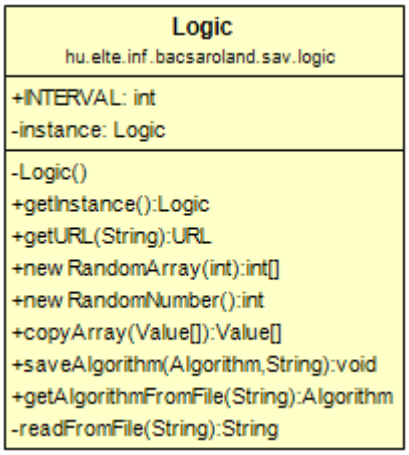
Csak osztályszintű adattagokat tartalmaz. Egyes design elemeket (színeket, betűtípusokat, méreteket) tárol, melyet a többi osztály használhat.



Design osztály UML diagramja.

Logic osztály:

A program **logika számításokat** végző függvényeit, metódusait tárolja. Ezen kívül a programban használt képfájlok elérését biztosítja, valamint **fájlműveleteket** is a *Logic* osztály segítségével lehet végrehajtani.



Logic osztály UML diagramja.

Adattagok:

- INTERVAL: egy előre definiált érték, amely tömbök által tárolható legnagyobb értéket definiálja. Ennél nagyobb számot a random szám generáló függvények sem fognak előállítani.
- instance: az osztály egyetlen példányát tárolja.

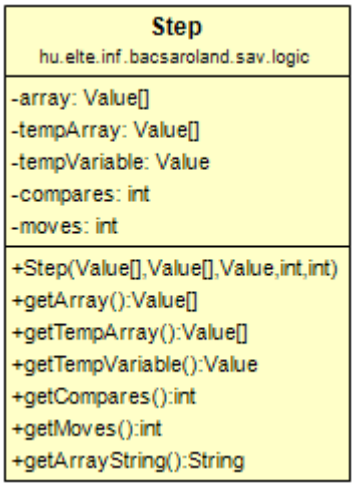
Konstruktor: a „Singleton” programtervezési mintának megfelelően az osztály privát, más osztályok számára elérhetetlen konstruktorral rendelkezik. Példányosítani csak a *getInstance* függvénnyel lehetséges. Az osztályból a program futása során ennek köszönhetően csakis egy példány fog létezni.

Függvények és metódusok:

- getInstance: visszaadja az *instance* adattag által tárolt példányt, amennyiben létezik. Ha nem létezik, előbb létrehozza azt.
- getURL: visszatérési értékként a paraméterként megadott elérési út alapján egy *URL*-t ad vissza. Az *URL*-eknek *.jar* futtatható állomány generálása után van szerepük. Egyértelműen leírják, hogy az egyes forrásfájlok (a program esetében képek) milyen mappában találhatóak, ezáltal ezekre hivatkozva nem képződik *NullPointerException*.
- newRandomArray: paraméterként megadott nagyságú random értékekkel feltöltött tömböt hoz létre.
- newRandomNumber: egy random számot generál az osztályban tárolt *INTERVAL* adattag alapján, melyet visszatérési értékként ad vissza.
- copyArray: a *Value* példányokat tároló tömbök másolását segítő függvény. Egy új tömböt hoz létre, majd a paraméterként megadott tömb elemein végigiterálva lemásolja azt.
- saveAlgorithm: az *Algoritmus mentése* funkció fájlkezelését végzi. A paraméterként megadott algoritmus és elérési út alapján egy fájlt hoz létre, melyben az algoritmus neve és a kezdeti tömb értékei tárolódnak el.
- getAlgorithmFromFile: a paraméterként megadott helyen található fájlt olvassa be, majd egy *Algorithmből* leszármazó példányt (rendezést) hoz létre.

Step osztály:

Az algoritmusok egy-egy lépését leíró osztály. Segítségével az *AlgorithmPanel* minden információt megkap, amivel egy lépést meg tud jeleníteni.



Step osztály UML diagramja.

Adattagok:

- array: az algoritmusok által használt tömbelemek, a hozzájuk tartozó színnel.
- tempArray: az algoritmusok által használt ideiglenes tömb. Ha az adott lépés nem használ ideiglenes tömböt a *tempArray* 0 méretű lesz.
- tempVariable: az algoritmusok által használt ideiglenes változó. Alapértelmezett értéke -1, ami azt jelenti, hogy nincs az adott lépésben használva.
- compares: az adott lépésig megtörtént összehasonlítások számát tárolja.
- moves: az adott lépésig megtörtént értékadások számát tárolja.

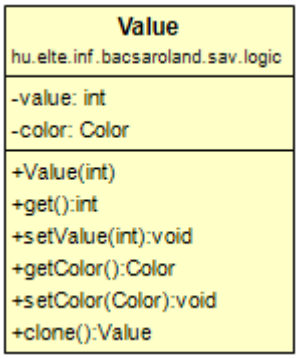
Konstruktor: paraméterként megkapja az összes értéket, amivel az adattagjait inicializálni tudja.

Függvények és metódusok:

- getArray, getTempArray, getTempVariable, getCompares, getMoves: az objektumorientált programozás elvei szerinti getter függvények a privát adattagokhoz.
- getArrayString: az osztályban tárolt array adattag elemeiből készít egy String változót. Ez a függvény segít az algoritmusok fájlba mentésénél.

Value osztály:

A szemléltetés színeinek nyomonkövethetőbb, könnyebb módosítása érdekében jött létre ez az osztály. Egy oszlop megjelenítéséhez szükséges összes logikai adatot tartalmazza.



Value osztály UML diagramja.

Adattagok:

value: egy szám érték, mely egy tömbelemet reprezentál.

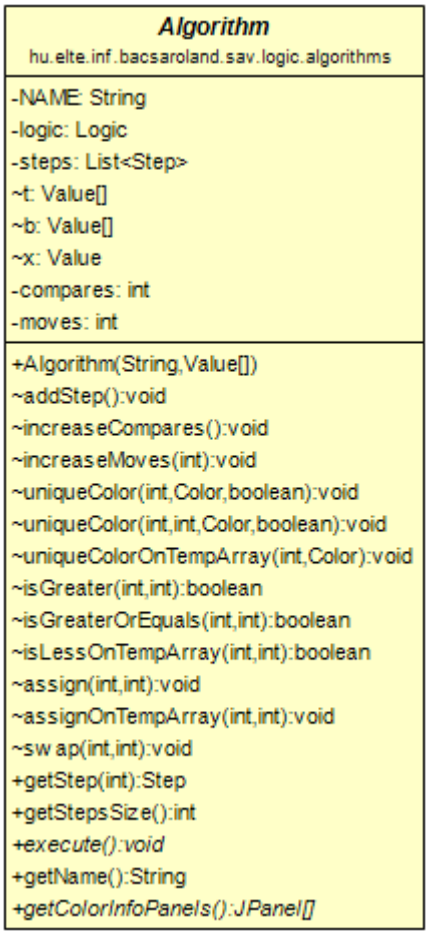
color: az oszlop színét adja meg a későbbi megjelenítéshez.

Konstruktor: a paraméterként kapott szám változó alapján inicializálja a *value* adattagot, a *color* adattagnak pedig egy alapértelmezett színt ad értékül.

- get, getColor: az adattagok lekérdezéséhez szükséges getter függvények.
- setValue, setColor: az adattagok módosításáért felelős setter függvények.
- clone: mivel az osztály implementálja a *java.lang.Cloneable* interface-t, ezért a *clone* függvényt felül kell definiálni. Segítségével másolható lesz az osztály.

Algorithm osztály:

A rendező algoritmusok számára szükséges minden adatot tartalmazó absztrakt osztály. Az osztály package-private metódusai és függvényei segítségével az összehasonlítások és értékadások a műveletek elvégzése mellett színezési lépéseket is létrehoznak. Ezáltal kevesebb színező metódust kell meghívni a rendező algoritmusok osztályaiban, így átlátható marad a kód.



Algorithm osztály UML diagramja.

Adattagok:

- logic: a *Logic* osztály egyetlen példánya.
- steps: egy lista, melyben az algoritmus lépései tárolódnak. A leszármaztatott osztályok töltik fel az *execute* metódusuk segítségével.
- t: az algoritmus által használt tömb.
- b: az algoritmus által használt ideiglenes tömb. Ha nem használja értéke egy 0 hosszú tömb.
- x: az algoritmus által használt ideiglenes változó. Ha nem használja értéke -1.
- compares: összehasonlítások száma.
- moves: értékadások száma.

Konstruktor: inicializálja az adattagokat a paraméterül kapott algoritmus név és kezdőtömb alapján.

Függvények és metódusok:

- addStep: az osztály pillanatnyi állapotát reprezentáló *Step* példányt hoz létre, melyet a *steps* lista végéhez fűz.

- increaseCompares: 1-el növeli az összehasonlítások számát.
- incraseMoves: a paraméterként megadott értékkel növeli az értékadások számát.
- uniqueColor: a paraméterben megadott indexű elem vagy elemek színét változtatja meg. Az utolsó logikai paraméter megmondja, hogy a szín megváltoztatása után hozzáadódjon-e a lépés a listához.
- uniqueColorOnTempArray: az ideiglenes tömb adott indexű elemének színezésére használható metódus.
- isGreater: a paraméterként megadott indexű tömbelemeket hasonlítja össze. Visszatérési értéke *true*, ha az 1. paraméterként kapott index helyén lévő tömbelem nagyobb, mint a 2. Visszatérés előtt a függvény egy színezést is végrehajt, melyet eltárol a listában lépésként.
- isGreaterOrEquals: az előző függvény, azzal a különbséggel, hogy itt az ekvivalencia is megengedett.
- isLessOrEqualsOnTempArray: a paraméterként megadott indexű tömbelemeket hasonlítja össze az ideiglenes tömbön. Visszatérési értéke *true*, ha az 1. paraméterként kapott index helyén lévő tömbelem kisebb vagy egyenlő, mint a 2. Visszatérés előtt a függvény egy színezést is végrehajt, melyet eltárol a listában lépésként.
- assign: a paraméterként megadott indexű tömbelemnek a *value* paramétert adja értékül. Ezen kívül egy színezést is végrehajt, amit lépésként el is tárol.
- assignOnTempArray: az előző metódus ideiglenes tömbön.
- swap: a megadott indexű elemeket cseréli ki egy segédváltozó segítségével a tömbben. A lépés tárolásra kerül az érintett elemek színezése után.
- getStep: a paraméterben megadott sorszámú lépést adja vissza a *steps* listából.
- getStepsSize: a *steps* lista méretét adja vissza.
- execute: A rendező algoritmus működését megadó absztrakt metódus, melyet minden leszármaztatott osztálynak definiálnia kell.
- getName: az algoritmus nevét adja vissza.
- getColorInfoPanels: absztrakt függvény, amelyet a leszármaztatott rendező algoritmusoknak definiálnia kell. Az algoritmus megértését segítő színmagyarázatokat adja vissza visszatérési értékként.

4.3. Tesztelés

A rendező algoritmusok összetettsége miatt a teszteléseket **manuálisan** végeztem el, mivel automatikus teszteket ezekre az algoritmusokra – úgy, hogy azok lefedjenek minden hibalehetőséget – közel lehetetlen írni.

A teszteléseket 3 részben végeztem. Először a **grafikus felület** elemeit vizsgáltam, nagy hangsúlyt fektetve a felhasználó által interakcióba kerülő nyomógombokra.

Másodszor az **algoritmusok működését** ellenőriztem. Ezeket a random generált adatokon kívül tesztfájlok segítségével is megvizsgáltam, így a rendezések ellenőrzésével egyúttal a fájlműveleteket végző komponensek is tesztelve lettek.

Végül a **használhatósági teszt** keretein belül a rendszer egészét, az elemek együttműködését vizsgáltam. Ellenőriztem, hogy a program megállja-e a helyét a felhasználói esetekre, és megfelelően gyorsan, használhatóan működik-e.

4.3.1. Grafikus felület tesztesetei

***Hozzáad* gomb és a hozzá tartozó input mező:**

Esemény	Elvárt eredmény	Kapott eredmény
A „333” helyes érték megadása, és hozzáadása a szerkesztő területhez.	A 333 értékű oszlop az eddigi oszlopok végére kerül, ezután az input mezőben új helyes érték generálódik.	Megegyezik az elvárt eredménnyel.
A „421” megengedettnél nagyobb érték megadása, és hozzáadása a szerkesztő területhez.	A gombot nem lehet megnyomni, így az érték nem kerülhet hozzáadása.	Megegyezik az elvárt eredménnyel.
Az „egy” szó megadása, és hozzáadása a szerkesztő területhez.	A gombot nem lehet megnyomni, így az érték nem kerülhet hozzáadása.	Megegyezik az elvárt eredménnyel.
Input mező üresen hagyása, majd nyomógomb megnyomása.	A gombot nem lehet megnyomni.	Megegyezik az elvárt eredménnyel.

***Szerkeszt* gomb és a hozzá tartozó input mező:**

Esemény	Elvárt eredmény	Kapott eredmény
A kijelölt oszlopnak „10” helyes érték megadása és szerkesztése.	Az oszlop értéke megváltozik, amely a magasságában is megnyilvánul. A kijelölést leveszi a szerkesztett oszlopról.	Megegyezik az elvárt eredménnyel.
A kijelölt oszlopnak „-1” a megengedettnél kisebb érték megadása és szerkesztése.	A gombot nem lehet megnyomni, így az oszlop nem kerülhet szerkesztésre.	Megegyezik az elvárt eredménnyel.
Az „egy” szó megadása a kijelölt oszlopnak és szerkesztése.	A gombot nem lehet megnyomni, így az oszlop nem kerülhet szerkesztésre.	Megegyezik az elvárt eredménnyel.
Input mező üresen hagyása, majd nyomógomb megnyomása.	A gombot nem lehet megnyomni.	Megegyezik az elvárt eredménnyel.
Érték megadása, majd nyomógomb megnyomása úgy, hogy nincs kijelölt oszlop.	Az input mező nem aktív, ezért nem lehet megadni értéket.	Megegyezik az elvárt eredménnyel.

***Torol* gomb:**

Esemény	Elvárt eredmény	Kapott eredmény
A kijelölt oszlop törlése a gomb segítségével.	A kijelölt oszlop eltűnik a szerkesztő területről.	Megegyezik az elvárt eredménnyel.
Gomb megnyomása úgy, hogy nincs kijelölt oszlop.	A gombot nem lehet megnyomni.	Megegyezik az elvárt eredménnyel.

***Mindent torol* gomb:**

Esemény	Elvárt eredmény	Kapott eredmény
Minden oszlop törlése a gomb segítségével.	A szerkesztő területről eltűnnek az oszlopok, a gomb inaktívvá válik.	Megegyezik az elvárt eredménnyel.
Gomb megnyomása úgy, hogy nincs egy oszlop sem a szerkesztő területen.	A gombot nem lehet megnyomni.	Megegyezik az elvárt eredménnyel.

Random tömböt generáló gomb:

Esemény	Elvárt eredmény	Kapott eredmény
A gomb többszöri megnyomása egymás után.	Minden gombnyomásra egy új 20 elemű tömb generálódik.	Megegyezik az elvárt eredménnyel.

***Mehet* gomb:**

Esemény	Elvárt eredmény	Kapott eredmény
Gomb megnyomása 50 elemű tömb létrehozása után.	A gomb továbbít a szemléltető felületre a másodperc töredék része alatt.	Megegyezik az elvárt eredménnyel.
Gomb megnyomása úgy, hogy üres a szerkesztő terület	A gombot nem lehet megnyomni.	Megegyezik az elvárt eredménnyel.

***Vissza* gomb:**

Esemény	Elvárt eredmény	Kapott eredmény
Gomb megnyomása miközben a folyamatos léptetés aktív.	Visszaugrunk a szerkesztő felületre probléma nélkül.	Megegyezik az elvárt eredménnyel.

Vezérlő gombok:

Esemény	Elvárt eredmény	Kapott eredmény
Folyamatos léptetés, *Szünet*, *Következő lépés* gombok megnyomása az algoritmus működésének befejezése után.	A gombokat nem lehet megnyomni.	Megegyezik az elvárt eredménnyel.
Szünet, *Előző lépés*, *Vissza az elejére* gombok megnyomása az algoritmus legelején.	A gombokat nem lehet megnyomni.	Megegyezik az elvárt eredménnyel.
A csúszka állítása miközben a folyamatos léptetés aktív.	A léptetés sebessége megváltozik, a szemléltetés nem áll meg.	Megegyezik az elvárt eredménnyel.
Folyamatos léptetés, *Előző lépés*, *Következő lépés*, *Vissza az elejére* gombok megnyomása miközben a folyamatos léptetés aktív.	A gombokat nem lehet megnyomni.	Megegyezik az elvárt eredménnyel.

Fájlműveletek:

Esemény	Elvárt eredmény	Kapott eredmény
Algoritmus betöltése a menüben található menüelem segítségével.	A fájlolvasás sikerességének eredményeként az algoritmus betöltődik, megjelenik a szemléltető felület.	Megegyezik az elvárt eredménnyel.
Algoritmus mentése a menüben található menüelem segítségével.	Az algoritmus későbbi betöltéséhez szükséges adatok tárolásra kerülnek, a megadott fájlban.	Megegyezik az elvárt eredménnyel.
Algoritmus betöltésekor nem létező fájl megadása.	A program hibaüzenet formájában értesít, hogy nem létezik a megadott fájl.	Megegyezik az elvárt eredménnyel.

Megjegyzés: mivel a program generálja a betölthető fájlokat, ezért ezek helyessége feltehető, ha mentéskor az elvárt tartalom kerül a fájlba. A helytelen fájlok kezelése ezen okból kifolyólag nem került implementálásra, azonban egy későbbi verzióban ez megvalósítható.

4.3.2. Algoritmusok működésének tesztelése

A rendező algoritmusok működésének tesztelése **két fázisban** zajlott. Először **random** generált adatokkal végeztem el a tesztelést. Minden rendezést 5 különböző, a grafikus felület által generált random tömbbel ellenőriztem.

Ezután az algoritmusok működését tesztfájlok segítségével is megvizsgáltam. Minden rendezéshez 6 tesztfájl készült, melyek különböző **problémás eseteket** tartalmaznak a rendezések szempontjából. Ezek a következő tömböket tartalmazzák:

- 1.1 tesztfájl: minimális 2 elemű tömb, egyező értékekkel.
- 1.2 tesztfájl: maximális 50 elemű tömb, egyező értékekkel.
- 2.1 tesztfájl: csökkenő sorrendű, rendezett tömb.
- 2.2 tesztfájl: csökkenő sorrendű, közel rendezett tömb.
- 3.1 tesztfájl: rendezett tömb.
- 3.2 tesztfájl: közel rendezett tömb.

A tesztfájlok megtalálhatóak a következő könyvtárban:

Source/hu/elte/inf/bacsaroland/sav/test

A tesztelés során az algoritmusok az elvártnak **megfelelően** működtek a random generált inputokra, és a tesztfájlokra egyaránt.

Egyedül a gyors rendezésnél kellett a színezésen javítani. Ha a *pivot* elem a *cut* függvény lefutását követően a helyén maradt, a *pivoton* kívüli, még nem rendezett részek nem színeződtek vissza kékre.

Ezen kívül a gyors rendezés rendelkezik egy a szemléltetés szempontjából irreleváns esettel, amikor a *quickSort* metódus 0 hosszú intervallumot kap meg paraméterként. Ennek a jelölése eltávolításra került.

4.3.3. Használhatósági teszt

Az elkészült program használhatóságának tesztelésére a legmegfelelőbb alanyoknak a végfelhasználókat találtam, ugyanis ők nem vettek részt a rendszer fejlesztésében, számukra még ismeretlen a program használata.

A tesztalanyokat a program rövid ismertetése után megkértem, hogy:

- hozzanak létre egy tetszőleges elemszámú tömböt,
- a tömbön hajtsák végre a számukra szimpatikus rendező algoritmusokat,
- egy algoritmust mentsenek el fájlba, majd töltsék vissza.

Egybehangzó vélemények alapján a program átment a tesztelésen. A működését megfelelően gyorsnak ítélték, a grafikus felületet áttekinthetőnek, a felhasználót jól informálónak találták. Hibajelenség, probléma nem merült fel a tesztelés során. Levonhatjuk tehát a következtetést, hogy a program megfelelően **használható**.

5. IRODALOMJEGYZÉK

[1] Ásványi Tibor: Algoritmusok és adatszerkezetek I. előadásjegyzet, 2016

[2] Fekete István: Algoritmusok és adatszerkezetek I-II. Elektronikus jegyzet, 2014

[3] Oracle: Java Platform, Standard Edition 7 API Specification
https://docs.oracle.com/javase/7/docs/api/, 2016 október 15.

[4] Wikipedia: Kupacrendezés
https://hu.wikipedia.org/wiki/Kupacrendezés, 2016 december 13.

[5] Wikipedia: Sorting algorithm
https://en.wikipedia.org/wiki/Sorting_algorithm, 2016 december 09.