PÉCSI TUDOMÁNYEGYETEM
TERMÉSZETTUDOMÁNYI KAR
MATEMATIKAI ÉS INFORMATIKAI INTÉZET

ALGOLYZE

Témavezető: Kiss-Vincze Tamás
tanársegéd

Készítette: Fenyvesi János Adrián
(LVYBOL)
Programtervező informatikus

PÉCS, 2024

Tartalomjegyzék

Absztrakt

A szakdolgozatom központi témája a rendező algoritmusok és azok megvalósítása az MVVM alapú architektúra alapján. A kutatásom hipotézise két helyben rendező algoritmus vizuális megvalósítása. A célom eléréséhez a Windows Presentation Foundation – WPF – grafikus felhasználói felületet alkalmaztam C# nyelven. A kutatásom célcsoportja olyan diákok, hallgatók, akik jobban szeretnék elmélyíteni a tudásukat az algoritmusok és rendezések világában. A megfigyelést középiskolai diákok körében végeztem. A felmérés során az egyes rendezések vizuális megjelenítése kellőképpen segített megértetni a diákokkal az algoritmusok működését. A fókuszban leginkább olyan diákok álltak, akik az informatikában szeretnék a tudásukat jobban elmélyíteni. A dolgozatom során a jól ismert buborék és beszúró rendezést tárgyalom, mind elméleti és gyakorlati szinten. A dolgozatom kitér az MVVM alapú architektúra fejlesztésének módszertanára és, hogy hogyan kell modulárisan fejleszteni egy kisebb-nagyobb alkalmazást. A kutatásom során arra az eredményre jutottam, hogy egyes algoritmusok megértése érdekében elengedhetetlen pont, hogy vizuálisan szemléltessük a működésüket. Továbbá a moduláris fejlesztés lehetővé teszi a kód átláthatóságát, egyszerűbb bővíthetőséget és a hibák javítását.

Bevezetés

Az informatikai és információs eszközök használata már általános, mindennapi egy ember életében. Az információszerzéshez felhasználjuk a rendelkezésünkre álló legközelebbi (okos)eszközt, amellyel rácsatlakozunk az elérhető hálózatra, majd rákeresünk a kívánt kulcsszóra. A keresett információkat rendezés szerint előállíthatjuk. A keresés során nem valószínű, hogy belegondolunk, hogy milyen eljárások, utasítássorozatok zajlanak le a művelet végrehajtása során. A szakdolgozatomban az olvasó megismerkedik az algoritmusokkal és egyes adatszerkezetekkel, amelyek hozzájárulnak ehhez a folyamathoz. Továbbá megismertetem, hogyan kell absztrakt módon megfogalmazni a hétköznapi problémákat és azokat algoritmizálni.

A kutatásommal vizuálisan szeretném bemutatni, hogy az általam kiválasztott algoritmusok hogyan működnek. Ezzel segíteni a tanulási, illetve vizualizációs képességeket, és az absztrakt gondolkodási módot a további tanulmányok során. A szakirodalom áttekintése során kiemelkedik néhány meghatározó munka, mint például *Lovász László és Gács Péter: Algoritmusok* vagy *Hemant Jain: Problem Solving in Data Structures & Algorithms Using C* című könyvek. Mindkettő remekül felvázolja és elmagyarázza az algoritmusok működését, mind elméleti és gyakorlati szinten. Ugyanakkor, bár ezek az innovációs szakirodalmak stabil alapokat adnak, nehezen feldolgozható a vizualizálás hiánya miatt.

Célkitűzés

A szakdolgozatommal egy tanuló programot valósítok meg, amely működése felhasználó barát, ezáltal egy laikus is kezelni tudja. A program kettő ismertebb rendező algoritmust valósít meg, amely a buborékrendezés és a beszúró rendezés. A felhasználó képes megadni bemenetet – számsorozat – és a kapott inputon lefuttatni a kettő említett rendezést. A vizualizálás során a felhasználónak lehetősége van a rendezés menetét lassítani, megállítani/elindítani, valamint gyorsítani. A kutatásom arra törekszik, hogy egy szemléletesebb és sokkal vizuálisabb módon megismertessem az olvasót az algoritmusok és adatszerkezetek témakörével. A szakdolgozatom mellé készítettem egy egyéni programot, amely a fent említett problémára ad megoldási lehetőséget, hogy pár rendező algoritmust illusztráljon és elemezzen. A program egy MVVM tervezési minta alapján készült, amely modulárisan felosztja, hogy egyes régetek milyen szerepet vállalnak a futtatás során.

I. Elméleti áttekintés

Az elméleti áttekintés során érinteni fogunk pár matematika és logikai alapismereteket, témaköröket is. Célja, hogy bemutassa, hogyan épül fel egy algoritmus, és hogy milyen gondolatmenet szükséges ahhoz, hogy mi is meg tudjuk írni és értelmezni a saját – vagy már megírt – algoritmusunkat.

II. 1. Az algoritmus

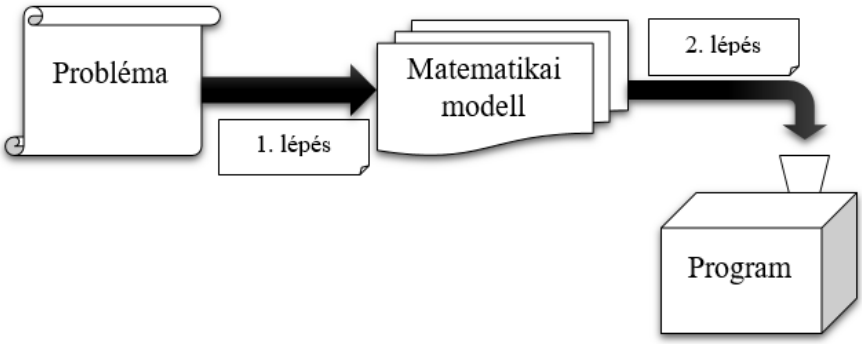
A számításelmélet és számítástechnika szakirodalmakból hallhatunk és olvashatunk leginkább az algoritmusokról, amit általában elvont számítógépekkel és valamiféle absztrakt matematikai modellezéssel kapcsolják össze [1]. A számítógépek implementációkat modelleznek, ami az algoritmusok speciális fajtája. Az algoritmusok egy meghatározott röviden leíró rekurzív eljárások [1].

Az algoritmust pontosan megfogalmazni nem lehet, mert a szakmán belül több értelemben használják. Ahány szakirodalom annyi megfogalmazása van. Sok szinonimája van, mint például recept, eljárás [2]. Egy számítási lépéssorozat, amely átalakítja a kapott bemenetet kimenetté [3]. Az algoritmus feladata, hogy véges számú lépésben megtalálja a megoldást a kapott problémára. Felfoghatjuk egy tervnek is. A helyes algoritmus a kapott bemenetre megoldja az adott problémát és végállapotba kerül [3]. A helytelen algoritmus ennek az ellentéte. A kapott bemenetre egyes esetekben egyáltalán nem áll le, lefagy vagy leállhat, de helytelen eredményt fogunk kapni [3]. Fontos megjegyezni, hogy az algoritmus csak akkor működő képes, ha pontos utasítássorozat leírást kap [3].

II. 1. 1. Algoritmus problémák

Az algoritmikus probléma megoldásához szükségesek absztrakt tervek, gondolatmenetek és példák, esetleg ellenpéldák. Az algoritmus problémára nincsen pontos recept, hogy minden problémát egy füst alatt lezavarjunk. A probléma megoldását egy absztrakt, elvont módon kell elképzelni majd modellt készíteni belőle (1. ábra problémától

a programig). A modellezés során egy hétköznapi problémát fogunk lefordítani a matematika nyelvére [2]. A megoldást szokás kétlépcsős folyamatként elképzelni.



1. ábra problémától a programig [2]

Az **1. lépés** és **2. lépés egy-egy leképzés**. Az **1. lépés leképzésben** a problémát átfogalmazzuk egy többé-kevésbé formalizált modellbe [2]. Ez a lépés pontosítást és egyszerűsítést jelent. A következő leképzés a **2. lépés,** ami alatt a hatékony algoritmus tervezését, kidolgozását értjük. Többnyire a modell által adott keretekből indul. Itt már pontos algoritmikus problémával van dolgunk [2]. Miután megvan a kellően absztrakt, jól megfogalmazott matematikai modell egy problémára, akkor megfogalmazhatunk egy algoritmust a modell alapján [4].

Az algoritmusok kezdeti változatában szerepelhetnek általános meghatározások, amelyeket a későbbiekben finomítani kell. A leírásukhoz többféle mód áll rendelkezésünkre [4]:

- ***Pszeudokód***: egy jól áttekinthető beszédes leíró szerkezet.
- ***Folyamatábra***: különféle alakzatokkal való ábrázolás, egyes folyamatokat írja le.
- ***Stuktogram***: egy strukturált, egységbezárt leíró eszköz.

A kapott algoritmust, amelyet a modellből képzetünk le koránt sem biztos, hogy tökéletes, vagy optimális. Előfordulhat, hogy egy korábbi állapotból kell új utat keresni [2]. Foglalkoznunk kell az algoritmussal, elemeznünk kell, megvizsgálni, hogy a módszer mennyire hatékony és hogy mennyire optimalizálható, javítható [2]. Ha egy algoritmus optimális, akkor arra nincs jobb megoldás. A **szuboptimális** algoritmus nem sokkal rosszabb, mint az **optimális**. Például tegyük fel, hogy a K egy természetes egész szám és *K-optimális* az algoritmusunk. Ez annyit jelent, hogy legfeljebb K-szor rosszabb a teljesítménye, mint az optimális megoldásnak. Ez egy megközelítő megoldás, de a hatékonysága javítható.

II. 1. 2. Hatékonyság

Ahhoz, hogy megtudjuk, hogy egy algoritmus mennyire bonyolult, meg kell ismernünk annak **lépésszámát** és a **futási idejét** [5]. Ha megtudjuk határozni az algoritmusunk lépésszámát, akkor a kapott adatokból eldönthető, hogy egy-egy algoritmus mennyire **bonyolult**. A bonyolultságból következik, hogy egy algoritmus mennyire hatékony. Az algoritmus bonyolultságát lényegében felfoghatjuk egy **függvényként** is. Mi is a *függvény*? A függvényt röviden leképzésnek hívjuk, a matematikában egy absztrakt fogalom. Mindig van egy értelmezési tartományunk és egy értékkészletünk [6]. Minden egyes *x* elemhez **egyetlen** *y* kimeneti értéket rendelünk [6]. Általános jelölése az $f(x)$.

Az **algoritmus gyorsaságát** meghatározhatjuk elemi utasításokkal, amiből becsülhető a futási idő *n* (ahol az *n* egy természetes szám) értékekre. Ez a futási idő függhet még más komponensektől (összetevőktől) is. Például a számítógép teljesítménye függ a hardverektől, processzor gyorsaságától, miszerint milyen gyorsan tudja végrehajtani az utasításokat. Az algoritmusok futási ideje általában nem megegyező.

Az algoritmusok összehasonlításánál fontos, hogy a különböző megközelítések mennyire hatékonyak és a teljesítményt kiértékeljük. Az összehasonlításnak két fő dimenziója a **futási idő** és a **memóriahasználat** – vagy tár igény - [7].

- A **futási idő** összehasonlításhoz általában az **O**-jelölést használjuk, amely a legrosszabb várható futási időt adja meg egy adott bemenetre [7]. Például, ha egy algoritmus futási ideje ***O(n)*** egy *n* elemű bemenetre, akkor lineárisan növekszik. A konstans szorzókat és kisebb rendű tagokat figyelmen kívül hagyjuk az összehasonlítás során.
- A **memóriahasználat** összehasonlítása ugyanúgy fontos, mint az előbb említett futási idő különösen, ha az erőforrás korlátozott [7]. Az összehasonlítás ugyanúgy zajlik, mint a futási időnél. Például az ***O(n)*** az algoritmus memóriahasználata lineárisan növekszik a bemeneti méret növekedésével [3].

Az algoritmusok futási idejét számos, az imént említett tényező befolyásolhatja. Az algoritmusok időkomplexitása függhet, hogy a kódsorok hányszor iterálnak [8]. Az idő pontos értékét bár nem kapjuk meg, de egy becslést tudunk róla szerezni.

A **lépésszám** az algoritmus folyamata során végrehajtandó műveletek száma. Miért is fontos nekünk, hogy tudjuk egy algoritmus lépésszámát?

- Az ***átlagos*** lépésszám fontos, ha az algoritmust gyakran kell lefuttatni. Példa: Az adatbázisban, egy webáruházban termékek keresése. Az algoritmus átlagos lépésszámának ismerete segít megérteni, mennyi időbe telik átlagosan egy lekérdezés feldolgozása.
- A ***maximális*** lépésszám fontos, ha egy adott időn belül szeretnénk az eredményt kapni, valamiféle határidőre (például időjárás jelentés). Másik példa: Egy RTS (real-time strategy) játékban az AI-nak gyorsan kell döntenie, hogy hogyan reagáljon a játékos cselekvéseire, még ha az nem is optimális megoldás.
- A ***minimum*** lépésszám ahol, mi az abszolút legalacsonyabb erőforrás-igény, amire szükség van az eredmény eléréséhez. Példa: Egy egyszerű rendezési algoritmus, mint a buborékrendezés. Ebben az esetben a minimum lépésszám megmutatja, hogy ha az adatsor már rendezett, akkor az algoritmusnak csak egyszer kell iterálnia az elemeken, hogy ellenőrizze, tényleg rendezettek-e.

Egy algoritmus átlagos lépésszámát a **valószínűségi változó várható értéke** határozza meg [2]. A matematikában és azon belül a statisztikában használják leginkább. Egy közelítő értéket ad. Az algoritmusoknál ez annyit jelent, hogy ha ismerjük a különböző lépésszámok valószínűségét, akkor kiszámítható az átlagos lépésszám, amit igényelni fog az elvégzendő művelethez [2].

III. Adatszerkezetek és adattípusok

Az algoritmusok és az adatszerkezetek a számítógépes program építőkövei. Az előző fejezetben részletesen olvashattunk arról, hogy az algoritmusokkal a problémákat fogalmazzuk meg egy kellően absztrakt módon [4]. Meghatározzuk a tervet, majd elemi lépésekben végrehajtjuk azt. Az **adatszerkezetek** vagy adatstruktúrák a mi általunk megfogalmazott terv megvalósítására nyújtanak segítséget. A fejezet célja, hogy az olvasó megismerkedjen és megértse az egyes adatszerkezetek működését és, hogy azokat milyen algoritmusok vezérlik [9].

Az **adattípusokkal** az egyes adatokat kategorizáljuk. Például egy számot érdemes numerikus értékként kezelni, míg a karaktereket, karakterláncokat karakter vagy szöveg

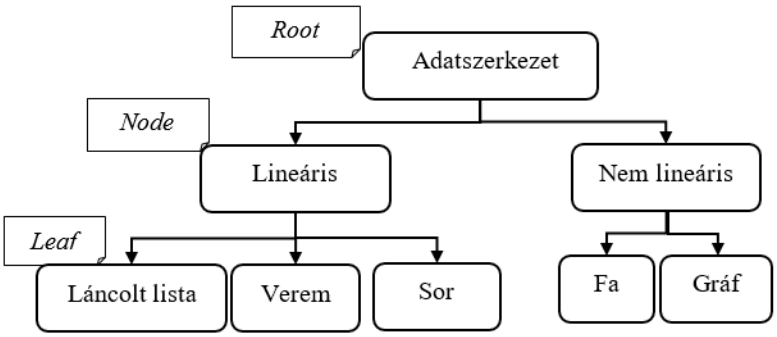
típusban tárolni. Ezeknek a típusoknak megvannak az előnyei és egyben hátrányai is [10]. Mindig a problémához legközelebb álló adatszerkezet és adattípus választására törekedünk.

Az adatszerkezeteknek két nagy csoportja van [10]:

- primitív adatszerkezetek,
- nem primitív adatszerkezetek (összetett).

III. 1. Összetett adatszerkezet

Az **adatszerkezetek** (lásd: 2. ábra adatszerkezet) az adatok konkrét reprezentációi. Ezek a programozó nézőpontjából vannak meghatározva [10]. Reprezentálják, hogy az adatok a memóriában hogyan lesznek eltárolva. Ahogy már fent említésre került, mindig a problémához legközelebb álló adatszerkezetet választjuk [10].



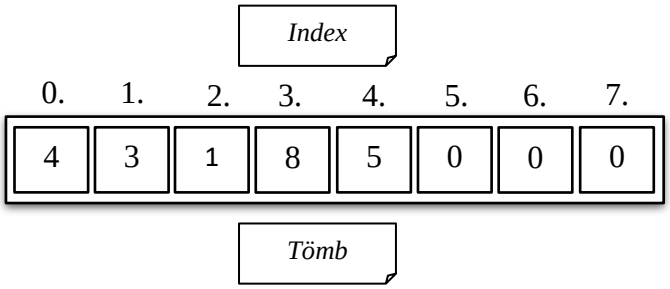
2. ábra adatszerkezet

A **lineáris adatszerkezeteket** általában egy vektorként kell elképzelni. Az elemek indexelése nullától kezdődik. A mi témakörünkben a lineáris adatszerkezetekkel fogunk foglalkozni, mivel a kettő rendező algoritmust ezekkel az adatszerkezetekkel implementáljuk.

III. 1. 1. Tömb

A **tömb** egy szekvenciális gyűjtemény és mégis primitív, amely azonos típusú elemeket tárol. A legáltalánosabb és – sok esetben – a leghasználtabb szerkezet is. **Azonos típusnak kell lennie minden elemnek** [9]. Bizonyos nyelvek megengedik az asszociativitást, mint például a PHP, ahol a tömb képes egyszerre akár karaktereket és számokat is tárolni. A tömbben a lineáris sorrendet a tömb indexei határozzák meg [3].

A tömbök **mérete statikus**. Ez azt jelenti, hogy előre meg kell határoznunk egy konstans értéket ami a tömb elemszámát fogja képviselni, hogy mennyi elemet legyen képes tárolni [10]. Ha esetleg a tömbünk mérete nagyobb, mint az adatok elemszáma, akkor az üres értékeket nullával tölti fel.

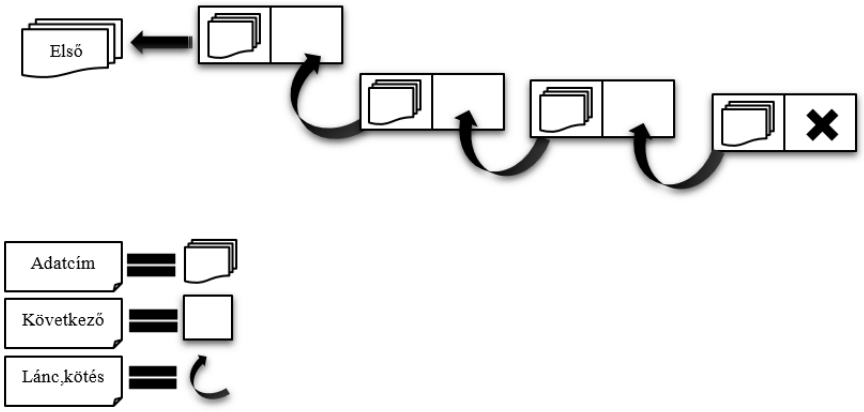


3. ábra Array - statikus tömb

Fontos megjegyezni, hogy a programfutás ideje alatt a **tömb mérete nem változtatható** [10]. Az ábrán (3. ábra Array - statikus tömb) látható egy példa, hogy a tömb mérete nagyobb, mint az elemek száma, amelyeket eltároltunk. Mi is ezzel a gond? A válasz implikálja magát, hogy a nem feltöltött helyeket – melyeket általában nullákkal tölt fel ha üres – ugyanúgy lefoglalja a memóriában [10].

III. 1. 2. Láncolt Lista

A tömb egy flexibilisebb verziója a **láncolt lista** (4. ábra láncolt lista). A láncolt lista egy **dinamikus** adatszerkezet ahol, az objektumok lineáris sorrendben vannak elrendezve [3]. Ezzel szemben a tömbbel, amelyben a lineáris sorrendet a tömb indexei határozzák meg. A láncolt listában a sorrendet minden objektumban található **mutató határozza meg** [3]. Itt a koncepció nem az, hogy az elemeket folyamatosan tároljuk, mint a tömb esetében, hanem **hivatkozásokat** használunk, úgynevezett mutatókat *(pointerek)* [10]. A **mutatók az értékek memória címeire mutatnak**. A láncolt listák a tömbbel szemben **lassabbak**, mivel nincs közvetlen hozzáférésük az elemekhez [10]. Előnye viszont, hogy nem kell előre megadnunk a lista méretét.



4. ábra láncolt lista

Az adatok helyéről az operációs rendszer dönt a memóriában és az adat kezdőcímeit átadja a programnak. Fontos megjegyezni, hogy a **lista első elemének helyét explicit módon meg kell határozni**. Ha tudjuk, hol van az első elem, akkor megmondhatja nekünk, hol van a második, és így tovább. A lista fejét (*L fej* vagy *L head*) egyes irodalmakban gyakran „külső hivatkozásnak” nevezik. Hasonlóképpen, az utolsó elemnek tudnia kell, hogy nincs következő elem [11]. Azaz, amíg a következő referencia nem üres, nem NULL érték (ábrán a nagy X jelölése) [9].

III. 2. Adattípusok

Az adattípusok a programozásban az úgynevezett primitív adatszerkezetek. Ez egyfajta **osztályozás**, amely kategorizálja és meghatározza egyes változó értékét [12]. A primitív adatszerkezetek általában egyetlen értéket tárolnak, míg az összetett adatszerkezeteket egy meghatározott struktúrának tekintjük. A felsorolt alap típusok lehetnek:

- **numerikus** értékek: egész és nem egész számok,
- **logikai** értékek: igaz vagy hamis,
- **karakterek**: *a,A,b,B*, stb.

Vegyük észre, hogy a szöveg adattípus egy speciális eset. Karakterek halmaza, mondhatni egy karakter tömb. Példa vegyük az *Alma* szót a következőképp: {’A’, ’l’, ’m’, ’a’}. A karaktereket az *ASCII* kódtáblázat alapján tudjuk ábrázolni, minden karakterhez tartozik egy kód [13]. *C#* nyelven minden típus a következő kategóriákba esik [14]:

- **Érték**: egy konstans, állandó tartalma egyszerű érték például 0. [14].
- **Referencia**: összetettebb, mint az értéktípus. Két része van, objektum és az objektum referenciája. Olyan objektumra mutat, amely tartalmazza az értéket [14].

- **Generikus**: típusparamétereket deklarál. Mondhatni helykitöltő típusokat. *Stack<T>*, amely arra lett tervezve, hogy *T* típusú példányokat helyezzen egymásra [14].
- **Mutató**: minden értéktípus vagy referenciatípus – legyen ez *P* – esetében létezik egy megfelelő pointer (mutató) típus, *P**. Egy pointer a példány egy változó címét tartalmazza [14].

A következő táblázat C# nyelven előre definiált és leggyakrabban használt adattípusokat tartalmazza [14]:

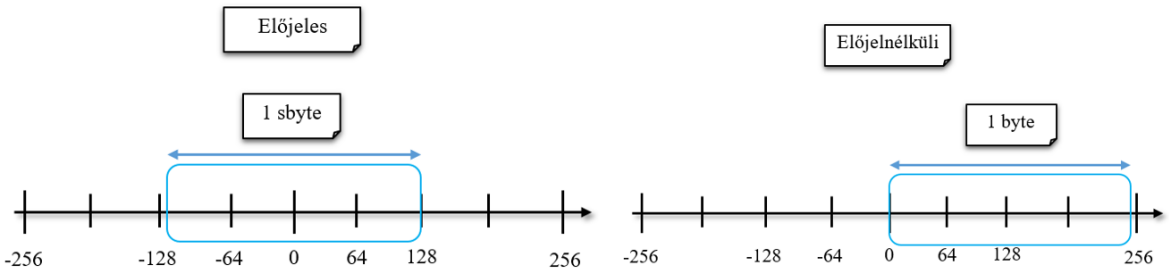
1. táblázat Adattípusok *[14]*

Adattípusok				
Érték szerint			**Referencia szerint**	
Numerikus	**Logikai**	**Karakter**	**Szöveg**	**Objektum**
Előjeles integer: sbyte, short, int, long	bool	char	string	object
Előjelnélküli integer: byte, ushort, uint, ulong:				
Valósszámok: float, double, decimal				

Az egyes értékekhez társítható helyigény, hogy hány bitet igényel [15]:

- byte, sbyte: 8 bit,
- short, ushort: 16 bit,
- int, uint: 32 bit,
- long, ulong: 64 bit,
- float: 32 bit,
- double: 64 bit,
- decimal: 128 bit,
- bool: 8 bit (true, false értékeket vesz fel),
- char: 16 bit,
- string: karakterlánc, minden karakter 16 bit,
- object: 32 vagy 64 bit, bármilyen típust képes tárolni.

A numerikus értékeknél fontos még megjegyezni, hogy elkülönítjük az előjeles és előjelnélküli értékeket *(5. ábra előjeles és előjelnélküli byte).* Nézzünk egy példát byte és sbyte esetén:



5. ábra előjeles és előjelnélküli byte

Vegyük észre, hogy az *sbyte* esetén -127-től tart egészen 128-ig az intervallum. A -128 már nem esik bele. A *byte* esetén intervallum 0-tól egészen 255-ig tart. Mind a *signed* és *unsigned* esetén a helyigény megegyező.

A fenti táblázat C# nyelv alapján készült, tehát más nyelven eltérő lehet egyes értéktípusok megnevezése. A primitív adatszerkezeteket azért hívják így, mert **közvetlenül támogatottak** a fordított kódban található utasításokon keresztül [14]. Ez általában közvetlen támogatást jelent az alapul szolgáló processzoron [14].

IV. Rendezés

A rendezés – vagy szortírozás – a számítástechnika egyik legalapvetőbb algoritmusa, és gyakran használt művelet a programozásban, illetve az adatbázisokban is [16]. A rendezés az egy olyan folyamat, amikor egy gyűjtemény elemeit **valamilyen sorrendbe** helyezzük **bizonyos feltétel alapján**. Például, egy szavakból álló listát ábécé szerint vagy hossz szerint rendezhetünk [11]. Egy városokból álló listát lakosság, terület vagy irányítószám alapján is lehet rendezni. A rendezés egyszerűbbé teszi a keresési algoritmusok megvalósítását, hogy hatékonyabban találjuk meg az adathalmazból a kívánt, keresett elemet. A rendezés elvégzése lehet **számszerűen** (sorszámok alapján) vagy akár **lexikálisan** (nevek alapján), vagy **bizonyos értékek alapján**, pontszámok alapján (értékelések szerint).

A nagy elemszámú gyűjtemény rendezése jelentős számítási erőforrásokat vehet igénybe, míg a kisebb gyűjtemények esetén egy bonyolult rendezési módszer több problémát okozhat, mint amennyit ér [11]. A rendezések **csoportokra** oszthatók:

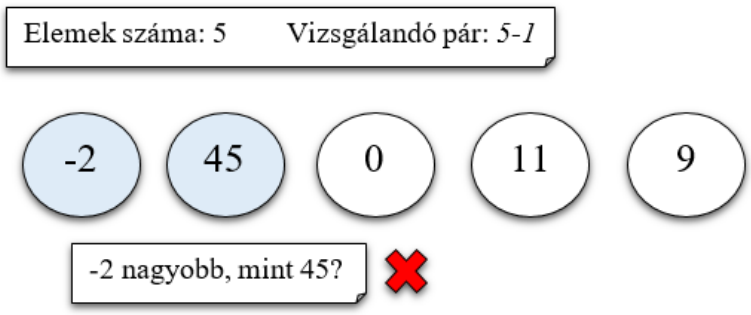
- a **helyben rendező algoritmusok** (in-place sorting), amelyek a buborékrendezés, kiválasztás, beszúrásos rendezés,
- másik csoport a **verem rendező algoritmusok**, melyek az összefésülő és a Ford-Johnson rendezés.

A **helyben rendező** algoritmusok a rendezésre szánt elemeket az eredeti adatszerkezetben hajtják végre a hozzátartozó utasításokat [17]. Ebből következik, hogy nem igényelnek jelentős mennyiségű tárolóhelyet. A másik csoportot, amelyek eltérnek ettől, hívjuk **verem rendező** algoritmusoknak. Nagyobb mennyiségű memóriát használnak a helyben rendezővel szemben, mivel egy másolatot hoznak létre vagy egy újat az eredeti adatszerkezetből [17].

IV. 1. Buborékrendezés

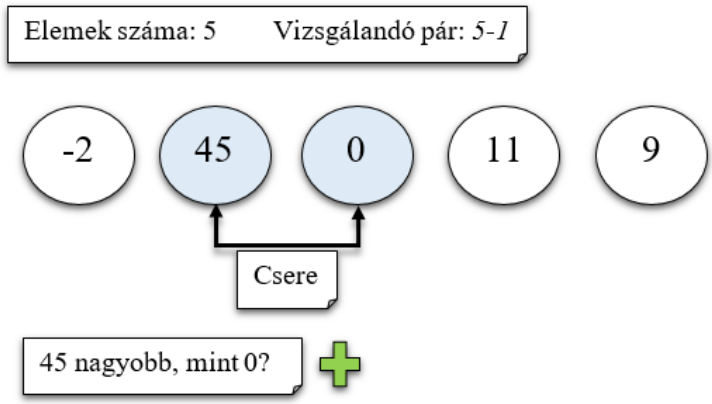
A buborékrendezés – másnéven Bubblesort – egy népszerű, de nem hatásos rendezési algoritmus. A működése lényegében az, hogy ismételten **felcseréli az egymás mellett lévő elemeket**, amelyek a sorozatban nem megfelelő sorrendben helyezkednek el [3]. Mondhatni minden elem „*felfelé buborékozik*". Tegyük fel, hogy van egy *n* elemű listánk, amelyekben számok vannak bizonyos sorrendben. Az első iterálás során *n* elem van és *n-1* elem párt vizsgálunk. A szomszédok összehasonlítás után **mindig a legnagyobb elemet választjuk ki és mozgatjuk tovább**, majd hasonlítjuk össze a soron következővel [11]. Az első iteráció során megtaláljuk a legnagyobb elemet, így már a második alkalommal *n-1* elem marad rendezetlenül. Ez azt jelenti, hogy *n-2* pár lesz, amit vizsgálnunk kell. Ismét egy elem a helyére kerül, itt megtaláljuk a második legnagyobb elemet [11]. Az algoritmus addig folytatódik, amíg a legkisebb elem nincs a helyén. A buborékrendezés egy **lassú** rendező algoritmus, viszont gyakran használják, mivel könnyű implementálni [10]. A komplexitását megtekinthetjük a *táblázat buborékrendezés elemzés* táblázatban.

Nézzünk egy példát a buborékrendezésre. Tegyük fel, hogy van egy rendezetlen halmazunk, listánk a következő elemekkel: {-2, 45, 0, 11, -9} (*6. ábra buborék hasonlítás*).



6. ábra buborék hasonlítás

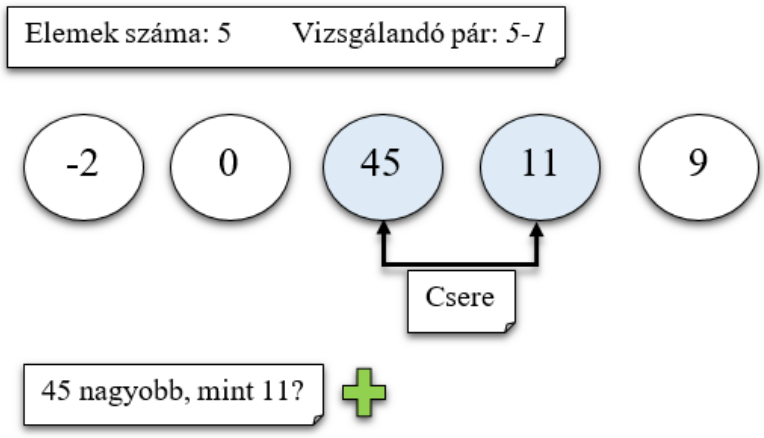
A első elemmel indulunk és hasonlítjuk össze a szomszédjával. A -2 nem nagyobb, mint a 45, tehát nem történik meg csere. Az algoritmus folytatódik, és a következő elempárt vizsgálja (*7. ábra buborék hasonlítás és csere*).



7. ábra buborék hasonlítás és csere

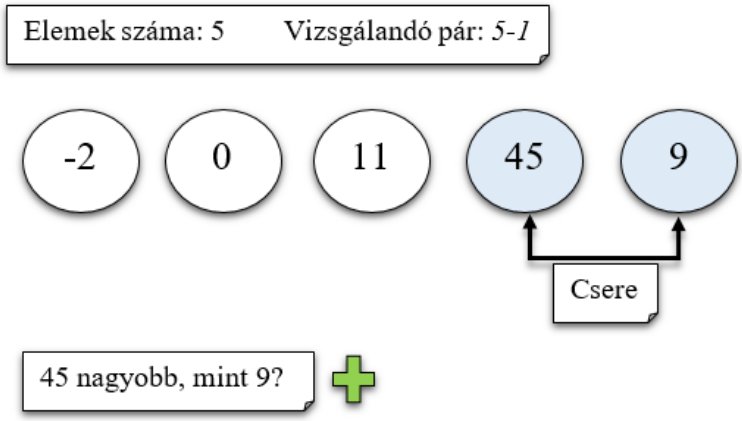
A következő lépésben a 45 és a 0-át hasonlítjuk össze, ahol a feltétel teljesül, így egy csere fog megvalósulni. A 0 és a 45-ös buborék pozíciót váltanak. Majd folytatjuk a vizsgálatot a következő szomszéddal. Vegyük észre, hogy addig vizsgáljuk az elemet, amíg nagyobb, mint a szomszédja. Ha a következő szomszédja nagyobb lenne, mint a vizsgálandó elem, akkor azzal az elemmel folytatnánk tovább a pozíció mozgatást.

Nézzük tovább a példát:



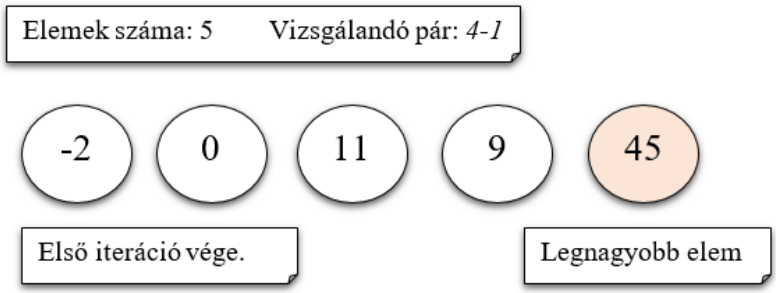
8. ábra buborék hasonlítás és csere

A *8. ábra buborék hasonlítás és csere* jól szemlélteti, hogy a 45 nagyobb, mint a 11-es buborék, így ismételten egy pozíció váltás kerül végrehajtásra.



9. ábra buborék hasonlítás és csere

Elérkeztünk az első iterációnak utolsó összehasonlításához (*9. ábra buborék hasonlítás és csere*). Látható, hogy az elem igen is nagyobb, mint a 9-es buborék, így a csere megtörténik.



10. ábra első iteráció vége

Az első iteráció során megkaptuk a **legnagyobb elemet** a listánkban, ami a 45. Az algoritmus itt nem ér véget, ezzel azt sikerült elérnünk, hogy egy elem pozíciója már fix. Így a vizsgálandó elempárok száma is csökken. Az algoritmus addig folytatódik, amíg nem kerülnek az elemek a helyükre. Ezt hívjuk egy **iterációnak** (*10. ábra első iteráció vége*).

A működése egyszerű, de ahogy korábban említve volt, ez a módszer nem optimális nagy adatkészletek esetén [10]. A rendezés során a sorozat összes elemén ismételten végig kell haladni és egyszerre csak két egymás melletti elemet vizsgál az algoritmus. Átalában kisebb adatkészleteken használják, vagy olyan adathalmazon, ahol az elemek már részben rendezettek.

2. táblázat buborékrendezés elemzés *[10]*

Buborékrendezés	
Legrosszabb eset	**O(n^2)**
Legjobb eset	**O(n)**
Átlagos eset	**O(n^2)**
Helyigény	**O(1),** szükség van egy *csere* változóra

Mit is jelent ez? A legjobb eset az, ha a lista már rendezett és ebből következik, hogy cserék nem lesznek. Azonban ha a legrosszabb esetet nézzük, akkor minden összehasonlítás egy cserét fog eredményezni [11]. Átlagosan fele annyi időt cserélünk.

A rendezés során betápláljuk az alap adatokat. A buborékrendezés általában két egybeágyazott ciklust igényel a tömb vagy lista bejárásához. A második ciklus magjában történik meg az összehasonlítás és a csere. A külső ciklus ***n***-szer fut le, míg a belső ciklus ***n-1*** alkalommal, mivel az elemek fokozatosan a helyükre kerülnek a listában az iteráció során.

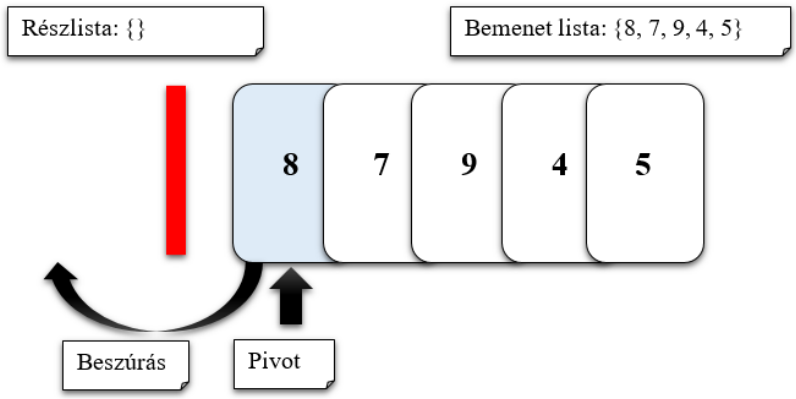
IV. 2. Beszúró rendezés

A beszúrásos rendezés – másnéven Insertion sort – egy hasonló architektúrájú és működő rendező algoritmus, mint a buborékrendezés [10]. Ehhez a rendezési módszerhez gyakori példa a kártyák sorba rendezése (például franciakártyák). A rendezés során mindig fent tartunk egy már **részben rendezett listát/sorozatot** (subarray) [11]. Ezt követően ellenőrizzük a már részben rendezett listánkat, és összevetjük a jelenleg vizsgált elemmel. A **pivot**[1] elemet megpróbáljuk beilleszteni a már részben rendezett listába a megfelelő helyre [5]. Ha egy kisebb elemhez, mint az aktuális, vagy a részlista végére érünk, akkor beszúrjuk az pivot elemet a megfelelő pozícióba [11].

Hogyan is működik? A résztömb elemeit mindig eltoljuk a pivot elemnek kedvezően [2]. Nézzünk egy példát. Tegyük fel, hogy a kezünkben öt darab franciakártyát tartunk figurától és szimbólumtól függetlenül a következő sorrendben: {8, 7, 9, 4, 5}.

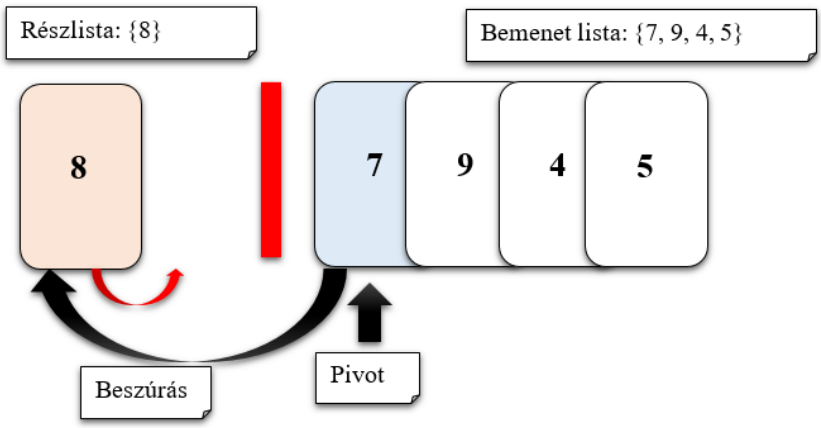
A direkt feltételezés az, hogy az **egy elemű lista már egy rendezett lista**, azaz a 0. pozícióban lévő elem [11].

[1] Kiválasztott, kitüntetett



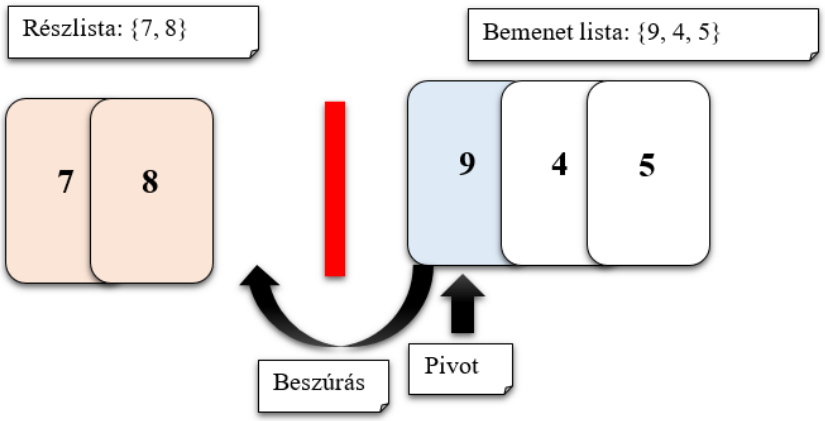
11. ábra első elem beszúrása

Első lépésként vegyük ki az első kártyát a bemenetként kapott listából. Így a **részlistánk** egy elemű (*11. ábra első elem beszúrása*) lesz: {8}. A részlistánk rendezett, így lépünk a következő elemre.



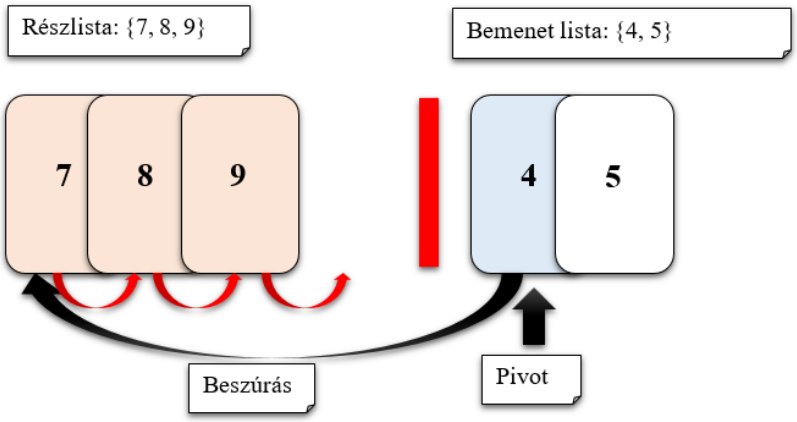
12. ábra második elem beszúrása és eltolás

A nyolcas és a hetes kártya összehasonlításkor észrevesszük, hogy a nyolcas nagyobb, mint a hetes (*12. ábra második elem beszúrása és eltolás*), így a nagyobb értéknek **eggyel jobbra kell tolódnia** [11], hogy helyet adjon a kisebb elemnek, jelen esetben a pivotnak. A beszúró rendező algoritmus minden iterációban eltávolít egy elemet a bemeneti adatkészletből majd, megtalálja a helyét a rendezett részlistában és beilleszti a megfelelő helyre.



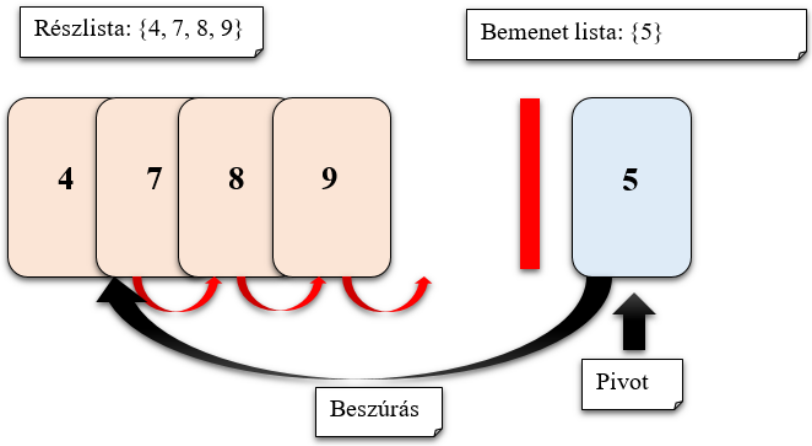
13. ábra harmadik elem beszúrása

A következő lépésben (*13. ábra harmadik elem beszúrása*) a lista elemei nem tolódnak el jobbra, hisz a kilences kártya nagyobb, mint a többi érték, az algoritmus csak beszúrja a pivot elemet a megfelelő helyre [10]. A következő lépésben viszont, el kell végezni az eltolást, hisz a négyes kártya kisebb, mint a részlistában található kártyák értékei (*14. ábra negyedik elem beszúrása és eltolás*).



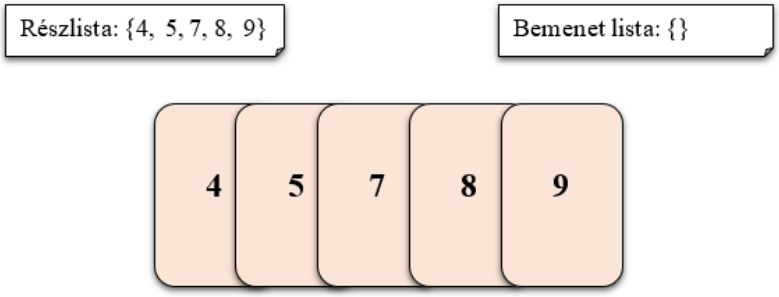
14. ábra negyedik elem beszúrása és eltolás

Vegyük észre, hogy mindig a részlistánkban tároljuk a helyes sorrendet. A beszúrásos rendezés során is egy segédváltozót használunk, mint a buborékrendezésnél. A különbség a buborékrendezéshez képest, hogy a *csere* változó egy ***eltolást*** képvisel, ami indexek variációját jelenti [8].



15. ábra ötödik elem beszúrása és eltolás

Az ötös kártya az kisebb, mint a többi érték, de nagyobb mint a négyes kártya. Így a beszúrás során úgy tolja el a lista elemeit az algoritmus (*15. ábra ötödik elem beszúrása és eltolás*), hogy a négyes és a hetes közé kerüljön a pivot elem. Az algoritmus véget ér, amint elfogynak az elemek a bemeneti listából (*16. ábra rendezett - részlista*).



16. ábra rendezett - részlista

A beszúrásos rendezés *n-1* iterációból áll *n* elem esetén [3]. Az egyes iterációkban a pivot elemet addig mozgatjuk balra, amíg nem találjuk a megfelelő helyét a sorozatban [10]. Az eltolás művelet kevesebb feldolgozást igényel, mint maga a csere, mivel csak egy hozzárendelés történik [3].

3. táblázat beszúrásos rendezés *[10]*

Beszúrásos rendezés	
Legrosszabb eset	**$O(n^2)$**
Legjobb eset	**O(n)**
Átlagos eset	**$O(n^2)$**
Helyigény	**O(1),** szükség van egy *eltoló* változóra az eltolásokhoz

A beszúrásos rendezés időkomplexitása azonos, mint a buborékrendezés algoritmusnak, de egy fokkal mégis jobban teljesít nála [3].

Az algoritmus két egybeágyazott ciklust használ. A külső ciklust használjuk arra, hogy a beszúrandó pivot elemet a bal oldali részben a már rendezett listába helyezzük. Utána a kiválasztott elemet egy ideiglenes *aktuális* – fent említve eltoló – változóba tároljuk. Ezt követően megvalósulnak a belső ciklusban az összehasonlítások és a pivot elemet addig mozgatjuk, amíg meg nem találjuk a megfelelő pozícióját [10]. A külső ciklus minden iterációval növeli a rendezett részlistánk elemszámát [10]. Amikor kilépünk a külső ciklusból, akkor ér véget az algoritmus és egyúttal listánk rendezett is lesz. A beszúró rendezést általában akkor használjuk, amikor:

- a tömbben vagy listában kevés elem található,
- nem az egész tömböt kell rendezni, csak néhány elem rendezetlen.

V. Fejlesztői dokumentáció

A fejezet célja, hogy segítse a program logikájának és a továbbfejlesztésének lehetőségeit. Ez a fejezet egy fokkal mélyebbre nyúlik már a programozásban. Itt az olvasó betekintést nyer arról, hogy a program milyen eszközökkel készült és milyen fejlesztői környezetben. Nagyobb hangsúlyt fektet a program architektúrájára és tervezésére.

V. 1. Alkalmazott fejlesztői eszközök és környezet

A program a **Visual Studio** integrált fejlesztői környezet (IDE[2]) segítségével valósult meg. A Visual Studio a Microsoft által fejlesztett IDE [18]. Az évek során folyamatosan bővül és egyre több programozási nyelvet támogat, mint például C++, C#, sőt még MASM (Microsoft Macro Assembler) is a része. Szükségesnek tartom szót ejteni a programozási nyelvekről. A programozási nyelveket az alábbi csoportra tudjuk bontani [19]:

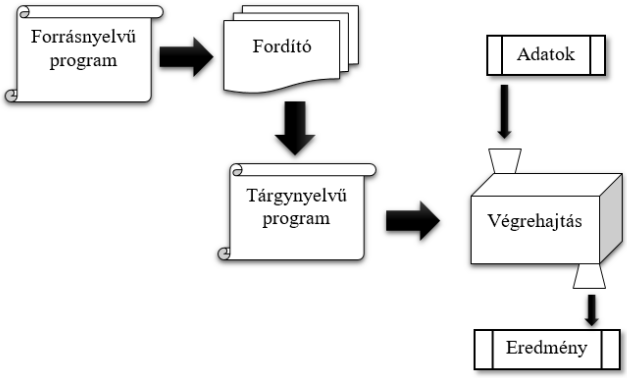
- gépi kód,
- assembly nyelv,
- magasszintű nyelvek: imperatív, deklaratív (logikai és funkcionális), objektumelvű.

A **fordítóprogram** egy forrásnyelven írt programot célnyelven írt programmá alakít át. A legismertebb típusú fordítóprogram az, amely egy magasszintű nyelvet – például C vagy C++ – gép natív assembly nyelvére fordít, hogy azt végrehajthassa [14] [19]. A **C#** is ilyen

[2] Integrated Development Environment

magasszintű nyelvekhez sorolható [14]. Egy általános célú, erősen típusos, objektumorientált programozási nyelv [14]. A C# előnye, hogy **egyszerű** és **kifejezőképes**. A nyelv **platformfüggetlen** (crossplatform) és számos platformspecifikus futtatókörnyezettel működik együtt. A Python nyelv például egy interpreter (értelmező), amely közvetlenül olvassa a forráskódot.

A **fordítóprogram** alapvető **működése**:



17. ábra compiler szerkezete *[19]*

A **forrásnyelv**, amelyet a programozó ír, de a számítógép nem tudja közvetlenül értelmezni. A fordító, azaz a **compiler** megkapja a forrásnyelvű programot és egy alacsonyabb szintű gépi kódra vagy köztes nyelvre fordítja le [19]. A **tárgynyelvű** program lesz a fordítás eredménye, amit a számítógép már tud értelmezni és **végrehajtani**.

A programom vizuális megvalósításához a **WPF** (Windows Presentation Foundation) nyílt forráskódú grafikus felületet alkalmaztam. A WPF a .NET keretrendszer 3.0 verziójában jelent meg [20]. A Windowsform és WPF mindkettő tökéletesen alkalmas a felhasználói felületek kezelésére. A WPF egy átfogó alkalmazásfejlesztési funkciókészletet biztosít, amely magában foglalja az **XAML** (Extensible Application Markup Language), vezérlőket, **adatkötést**, elrendezést, 2D és 3D grafikát, animációt, stílusokat, sablonokat, dokumentumokat, médiaelemeket, szöveget és tipográfiát [20]. Ebből következik, hogy a WPF alkalmas az **MVC** és az **MVVM** architektúrára. Általában a WPF és az MVVM egy egységet alkot.

A felhasználói élmény fokozásához különböző **ikonokat** alkalmaztam az icons8[3] segítségével. Ezek az ikonok képként ingyenesen letölthetők. A letöltött ikonokat megtekinthetők a forrásállomány mappájában.

Az ábrák – nagy része és persze maga a dokumentáció –, táblázatok **Word** segítségével valósultak meg, míg az egyes diagramokat (UML diagram) a **Creately**[4] weboldal segítségével hoztam létre.

V. 2. Architektúra

Első lépésként beszéljük át mi is, azaz architektúra és mik a részei. **Az architektúra** a rendszert felépítő **szoftver komponensek halmaza** – másnéven alrendszerek – amelyek, szükségesek a rendszerhez és magukba foglalják a szoftverelemeket és a közöttük lévő kapcsolatokat [21]. Röviden és tömören felfogható egy magasszintű tervnek, ami a rendszer átfogó struktúrája. Több nézetből definiált komponensek és azok kapcsolatainak összesége.

Az architektúrának három fő típusa van:

- **Modul szerkezetek**: a rendszer szervezése. Modulokra, adategységekre szervezzük a funkciókat [21].
- **Komponens és összekötő szerkezetek**: a futásbéli elemeket és kölcsönhatásaik szerkezetét és egyes komponensek interakcióit tárja fel [21].
- **Allokáció**s – elhelyezkedés – **szerkezetek**: a szoftver és a szoftveren kívüli struktúrához való kapcsolódás bemutatása. Például tárolásuk vagy mely processzoron futnak az egyes szoftverelemek [21].

Tehát a programunk egyes funkcióit elkülönítjük, rétegekre bontjuk a kódolás során. Ez egy fajta **modularitás**.

V. 3. MVC és MVVM

A **Model-View-Controller** – röviden MVC – a szoftvertervezésében használt programtervezési minta. Az MVC (magyarul Model-Nézet-Vezérlő) három fő rétegre bontja a szoftvert [22]. A szerkezete megtekinthető a *18. ábra MVC minta.*

[3] https://icons8.com
[4] https://creately.com/lp/uml-diagram-tool/

Model

Az adatok, amivel maga az alkalmazás dolgozik. Például adatbázis kapcsolatot itt implementáljuk. **Nincsen hozzáférése a nézethez**, csak nyers adatokkal dolgozik [22]. Ezek az adatok egy fajta struktúrák, objektumok, amelyek képesek tárolni entitásokat. Az adatok módosítása, manipulációja a model feladata.

Nézet

A nézet **felelős a kimenetért**, amit a felhasználó lát. Úgy ahogy a model nem „ért" a felhasználói felülethez, úgy a nézet sem „ért" a nyers adatokhoz, amelyet a model kezel [22]. A nézet nem tud számolásokat és bonyolult műveleteket végezni. A vezérlő értesítése a feladata. A kapott adatot, információt a modelltől jeleníti meg a felhasználó számára.

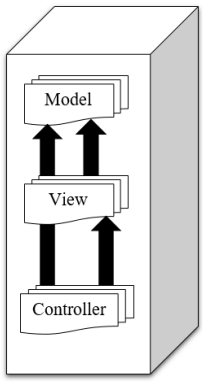
Vezérlő

A vezérlő – ahogy a neve is árulkodik – felelős mindenféle kapcsolatért. Irányítja a program menetét. Az alkalmazás logikáért felel és semmiféle adatot nem tárol. Úgymond egy hidat teremt a model és a nézet között. Közvetlen kapcsolata van a főprogrammal. A vezérlő **a model és a nézet közötti kapcsolatot biztosítja** [22]. Adatokat kér el a modelltől, amelyet továbbít a nézet számára.

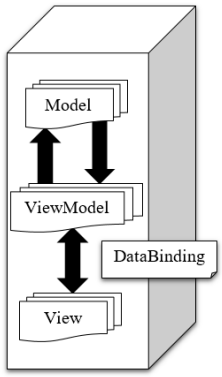
Az MVC előnyei:

- Könnyen átlátható.
- Frissítést egyszerűvé teszi a modularitás miatt (ez előző pontból következik).
- Az alacsony szintű kódrészletek redundanciája megszűni.
- Modularitás lehetővé teszi, hogy a logikáért felelős fejlesztői csoport és a felhasználói felületért fejlesztői csoport egyszerre dolgozzanak [22].
- Többféle adatstruktúra is megjeleníthető ugyanabban a nézetben.

A tervezés során először a programom funkcióit és azok működését kellett megszervezni.



18. ábra MVC minta



19. ábra MVVM minta

A **Model-View-ViewModel –** röviden MVVM – szintúgy egy programtervezési minta. Az elnevezését eredetileg a Microsoft határozta meg a WPF és Silverlight számára [22]. Az MVVM az MVC és az MVP minták alapján készült. Célja, hogy jobban szeparálja a felhasználói felület (UI[5]) fejlesztését az alkalmazás üzleti logikájától és viselkedésétől [22]. Ennek érdekében az MVVM minta megvalósítása során gyakran használnak **adatkötéseket** (binding). Az adatkötések lehetővé teszik, hogy a felhasználói felület és a fejlesztés munka menete egyidőben történjen ugyanabban a kódban.

Model

Az MVVM-nél a Model réteg ugyanazt képviseli, mint az MVC mintánál. Az információkat tárolja, de viselkedést nem kezel. Nem formáznak, és nem befolyásolják a kimenet megjelenítését. A Model réteggel a ViewModel kommunikál, itt valósul meg az üzleti logika [22].

View

A View – nézet – az alkalmazás azon része, ahol a felhasználók interakciókat végeznek. A ViewModel állapotát képviselik. A különbség az MVC-hez képest, hogy itt a View úgymond egy „aktív” nézet. Ez annyit jelent, hogy tartalmazza az adatkötéseket, eseményeket és viselkedéseket, amelyek a Model és a ViewModel megértését igénylik [22]. Feladata, hogy megjelenítse a ViewModel információit.

ViewModel

A ViewModel felfogható egy speciális vezérlőnek – Controller – ami adat konverterként funkcionál. A UI mögött helyezkedik el és kinyeri a szükséges adatokat, amelyekre a View-

[5] User-Interface

nak szüksége van a Modelből. Egy forrásnak tekinthető, ahova View adatokért, műveletekért fordul segítségül [22].

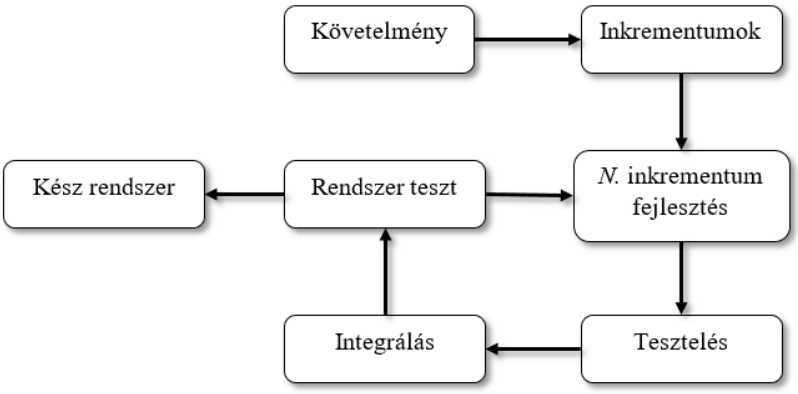
VI. Tervezés

A szoftverfejlesztés egyik kulcsfontosságú szakasza a tervezés és tesztelése. Hasonlóan kell elképzelni, mint az algoritmusok leképzését. Egy absztrakt módon meghatározzuk, hogy egy adott szoftver milyen funkciókkal rendelkezzen és milyen követelményeknek feleljen meg.

VI. 1. Módszertan

A programom fejlesztése során az **inkrementális módszertan** elveit követtem. A folyamat hasonló a vízesésmodellhez [23]. Az inkrementális módszertan lehetővé teszi a fokozatos fejlesztést, így az eredmények gyorsabbak, de nem feltétlen strukturáltak. A módszertan során több „inkrementumra", iterációra bontjuk fel egy-egy funkció fejlesztését [23].

Először egy vázlatos követelményrendszert állítunk fel. Ezt követően az egyes szolgáltatásokat osztályozzuk a fontosságuk szerint. Miután a sorrend megvan, megkezdjük a megvalósítást.



20. ábra inkrementális folyamat *[24]*

Feldolgozzuk a követelményeket, majd inkrementumokra bontjuk. **Prioritás szerint** elkezdjük a megvalósítást, és amikor a rendszer tesztelés fázisba érünk, akkor a következő inkrementumra megyünk. Hasonló egy számláló ciklusra, a kezdő érték maga a követelmény, a kilépési feltétel az inkrementumok száma, majd addig iterálunk, amíg nem érjük el a követelmény, az inkrementumok „tömb hosszát".

A program inkrementum prioritása:

1. **Adatok generálása**: A programunk adatokkal, számsorokkal dolgozik, amelyeket szükséges generálni. Ez az első lépés amit a felhasználónak meg kell tennie, hogy a program teljes funkcióját kihasználja.
2. **Generált adatok betöltése**: Az előző pont szükséges feltétele, hogy a betöltés implementálható legyen. Ahhoz, hogy egy állomány betöltésre kerüljön, szükség van már meglévő adatra.
3. **Rendezések**: Az adatok betöltését követően, választhatunk két fajta rendezés közül. Feldolgozza a betöltött adatokat és megjeleníti a rendezéshez méltó vizualizációval. Ez a program fő funkciója.

VI. 2. UML diagrammok

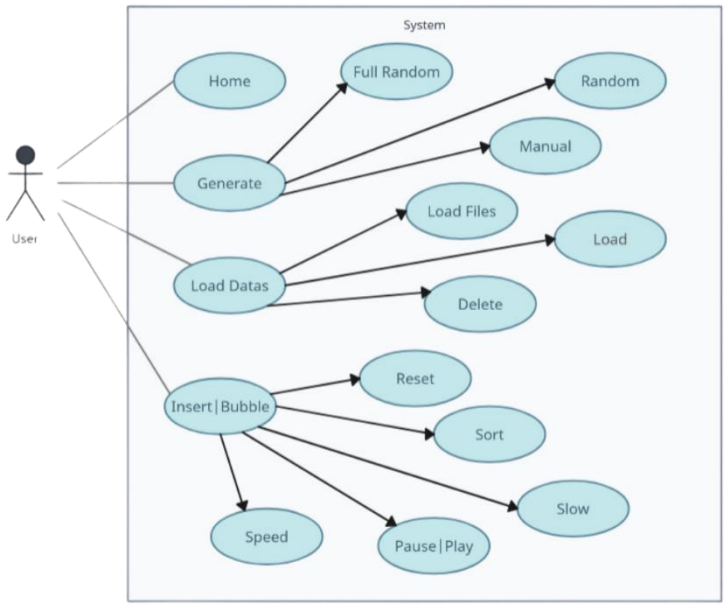
Az UML – **U**nified **M**odeling **L**anguage – a szoftverek alkotóelemeinek meghatározására, modellezésre és dokumentálására szolgáló nyelv [25]. A fejlesztés során segít a rendszer szerkezetének és viselkedésének szemléltetésében. Fontos megjegyezni, hogy a diagramok magyarázata során, nem térünk ki minden aspektusára, csak azokra, amelyek szükségesek a program felépítésének megértéséhez.

VI. 2. 1. Használati eset diagram

A használati eset diagram – másnéven *Use-Case Model* – a **felhasználók interakcióit**, kapcsolatát írja le a **rendszerrel**, ezzel szemléltetve, hogy a különböző esetek milyen kapcsolatban állnak a felhasználóval [25]. A diagramnak három fő pillére van:

1. Az **esetek** (Use-Case): Ellipszis formájú szimbólumok, melyek tartalma a tevékenység vagy folyamat, amelyet a rendszer végrehajt.
2. A **szereplők** (Actors): Akik kapcsolatba lépnek a rendszerrel, maguk a felhasználók. A kapcsolat halmaz nem feltétlen ilyen szűk, lehetnek még más rendszerek vagy entitások.

3. A **kapcsolatok** (Relations): A szereplők és az esetek közötti viselkedést, interakciót jelöli. Szemlélteti a szereplők részvételét a programban.



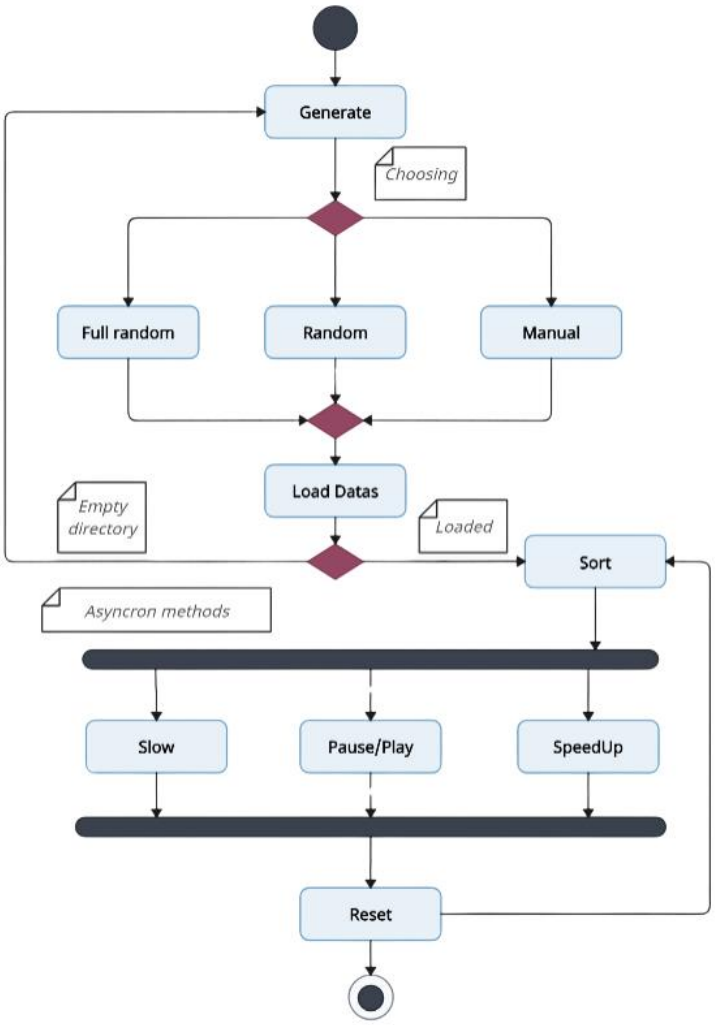
21. ábra Eset diagram

Az *Eset diagram* jól szemlélteti, hogy a felhasználó milyen interakciókat képes végrehajtani és, hogy azok a funkciók milyen további ellenőrzéseket, viselkedéseket tartalmaz.

VI. 2. 2. Aktivitás diagram

Az aktivitás diagram – másnéven Activity diagram – a **tevékenységek sorozatának megjelenítésére szolgál** [25]. Majdnem minden célra alkalmazható, lehet ez akár algoritmusok lépéseinek szemléltetése. Az alapelemei [26]:

1. maguk a tevékenységek (jelölése egy lekerekített téglalap),
2. átmenetek (nyíl),
3. elágazások (rombusz),
4. kezdő és végállapot (kezdő a telített karikát, végállapot a dupla körös karika).



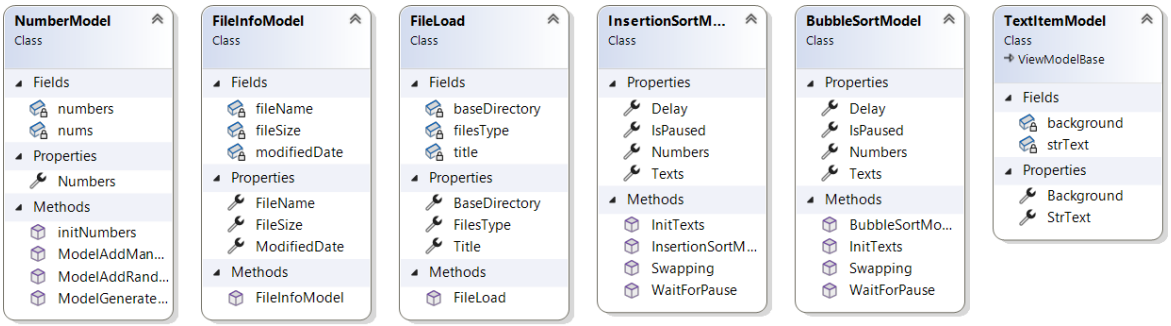
22. ábra aktivitás diagram

Az *aktivitás diagram* ábrázolásához alkalmaztam még a konkurens tevékenységeket is. Előfordul, hogy egyes tevékenységek egymástól függetlenül vagy **párhuzamosan** végezhetők [26]. A diagramban a jelölése egy vastag vonal, amelyből kiágazik több nyíl, azaz több átmenet. Ezek az átmenetek párhuzamosan zajlanak. A programomban a párhuzamosság jelen van, de erről az implementáció részben bővebben lesz szó.

VI. 2. 3. Osztály diagram

Az osztály diagrammok a leghasználtabb és legelterjedtebb ábrázolási módok az objektumorientált programozásban. Általában a tevékenységek meghatározása után kerül sor az osztály diagram kialakítására. Az **osztályok** – entitások – **közötti kommunikációt vázoljuk fel**. A diagrammal az entitások tulajdonságait és láthatóságait is ábrázoljuk. A következő diagrammok a program egyes moduljainak az osztály diagram képei.

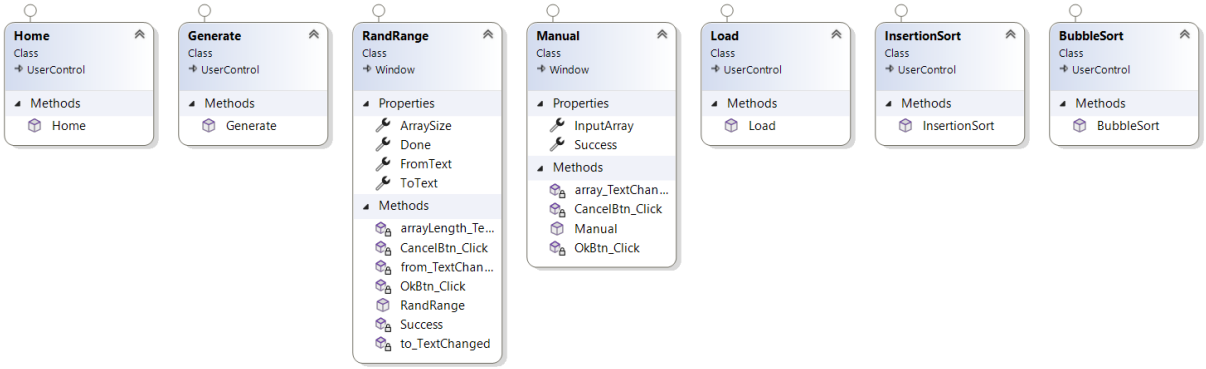
Model osztályok felépítése



23. ábra Model osztályok

Ahogy már korábbi fejezetben említve volt, az MVVM mintát alkalmaztam a programom megvalósítására. A program több csomagból, modulból áll. A *Model osztályok* felelősek minden számításért, logikai műveletért.

View osztályok felépítése

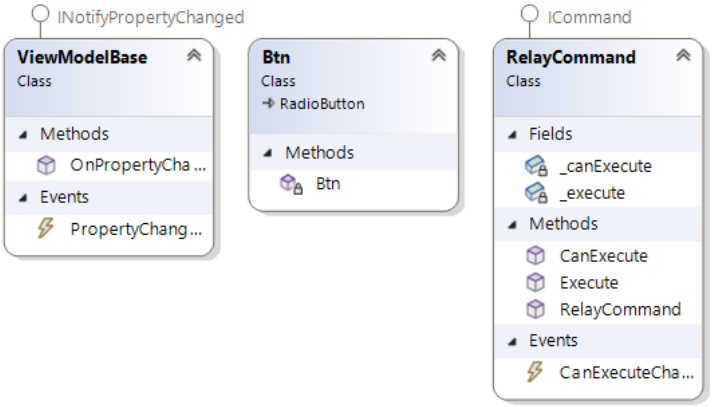


24. ábra View osztályok

A *View osztályok*nak lényegesebben kevesebb feladatuk van. A feladatuk annyi, hogy megjelenítsenek objektumokat vagy valamiféle vizuális módosítást végezzenek. A ***RandRange*** és ***Manual*** osztálynak feltűnően több metódusa van, mint a többi megjelenítőnek. Az oka az, hogy az a két osztály egy „speciális” azaz, egyéni ablak, amelyet én építettem be a programomba az adatok generálásához. A két egyéni ablak bemenetet vár a felhasználótól. A View osztályok implementálják a ***UserControl*** osztályt, Ők csak arra szolgálnak, hogy valamiféle felhasználói vezérlést hajtsanak végre.

Mielőtt beszélnénk a ViewModel modul felépítéséről, szeretném megjegyezni, hogy készítettem specifikus osztályokat. Ezek az osztályok a csak „support” azaz támogatóként vannak jelen.

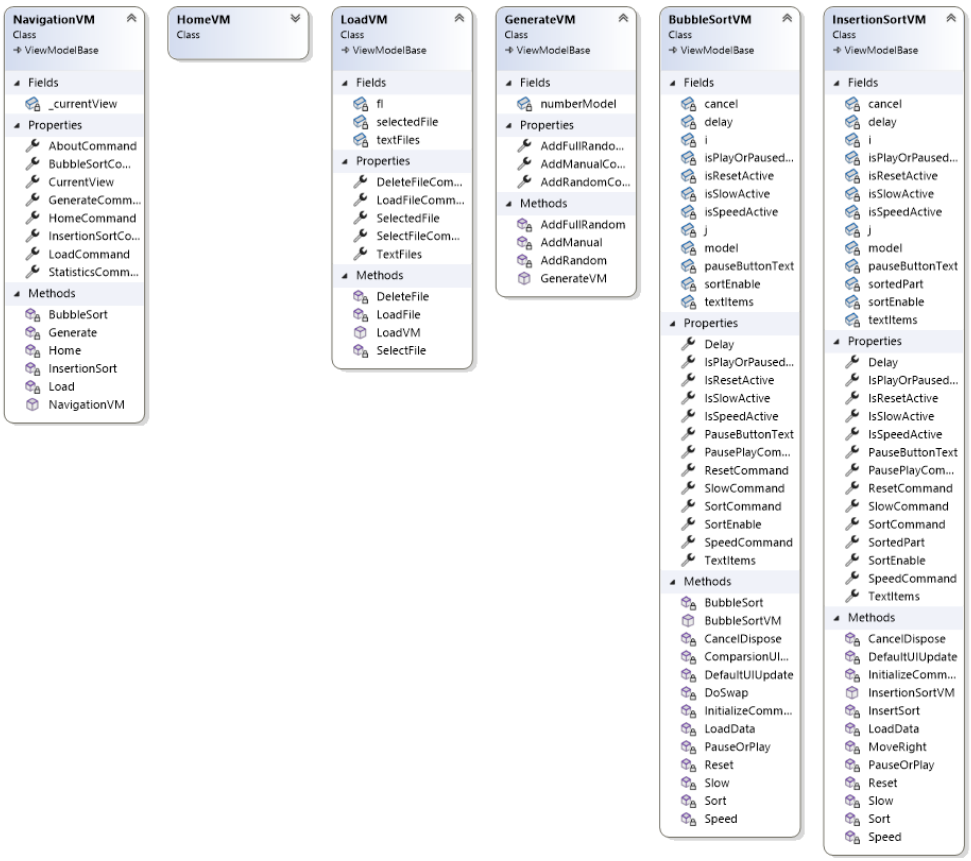
Támogató osztályok felépítése



25. ábra Támogató osztályok

A *Támogató osztályok* célja, hogy egyes ismétlődéseket, redundáns kódrészleteket megszüntessenek. A ***ViewModelBase*** osztály egy beépített interfészt implementál a WPF-ből. Az interfész megvalósítása során egyes elemek tulajdonságainak változását figyeli. Például ha változik egy szövegdoboz értéke, akkor történjen valami. A ***RelayCommand*** osztály szintúgy egy beépített interfészt implementál. Megvalósítja, hogy egyes metódusok **köthetőek legyenek** eseményekhez. Például egy gomb kattintásához hozzákötünk – adatkötéssel – egy parancsot. A ***Btn*** osztály a menüben lévő gombokhoz nyújt segítséget, egyéni vezérlőként funkcionál.

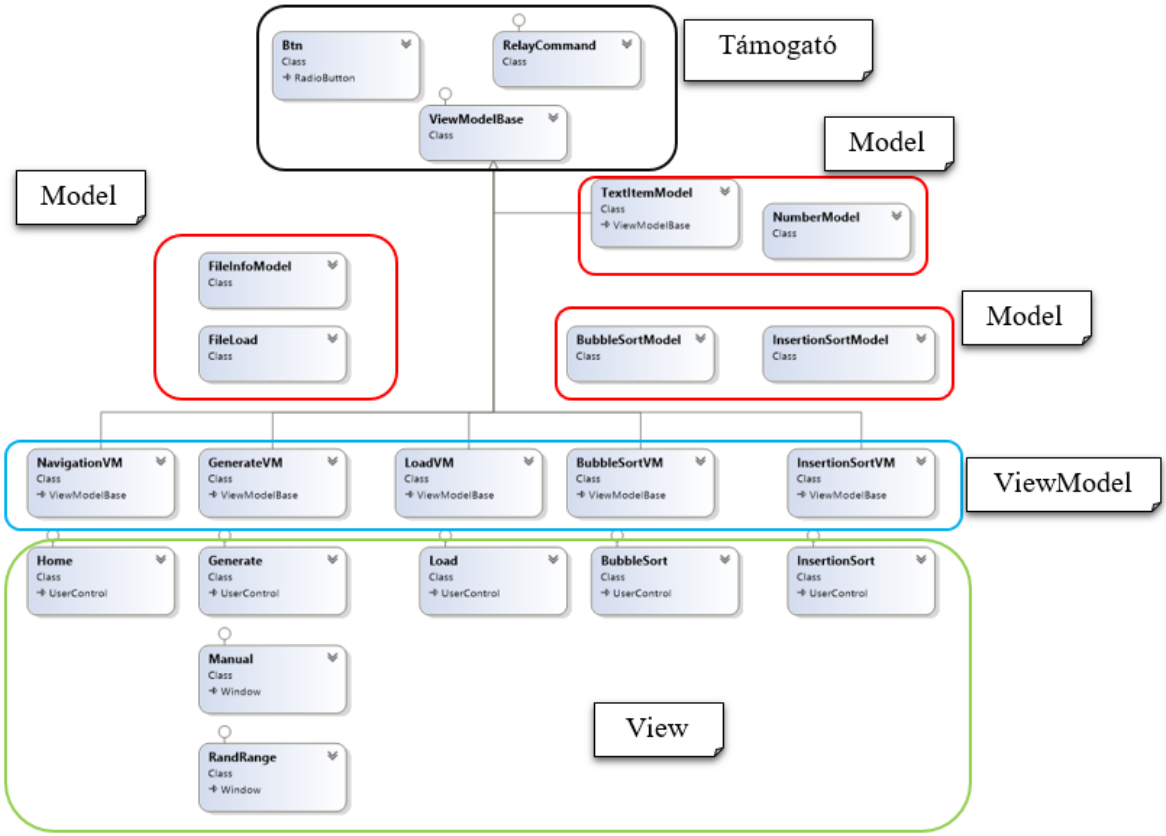
ViewModel osztályok felépítése



26. ábra ViewModel osztályok

A *ViewModel osztályok* közül egyesek jóval több feladatot látnak el, mint a többi. Triviális, hiszen a programom a két rendező algoritmus vizuális bemutatásáért felel, a többi funkció inkább csak egy-egy pillér szerepét képviseli. A ***NavigationVM*** felelős a különböző nézetek meghívásáért. A ***HomeVM*** a belépési pont a programban, itt a logó jelenik meg. A ***LoadVM*** felelős a fájlok betöltéséért, míg a ***GenerateVM*** a fájlok létrehozásáért. A főelemek a rendezések, ***BubbleSortVM*** és az ***InsertionSortVM***. Itt nagyon sok tulajdonság és eljárás szerepel, ezek mind azért felelnek, hogy az algoritmus helyesen működjön és, hogy szemléltesse az algoritmus működését.

Asszociáció, kapcsolatok felépítése



27. ábra Osztályok közötti kapcsolat és felbontás

A program megvalósítása során a model osztályok egy részéhez nem tartozik más elem. Ezek az osztályok szintúgy úgymond egyfajta támogatást nyújtanak, de mégis modelként hivatkozom rájuk, mert valamiféle logikai, számítási feladatot végeznek el.

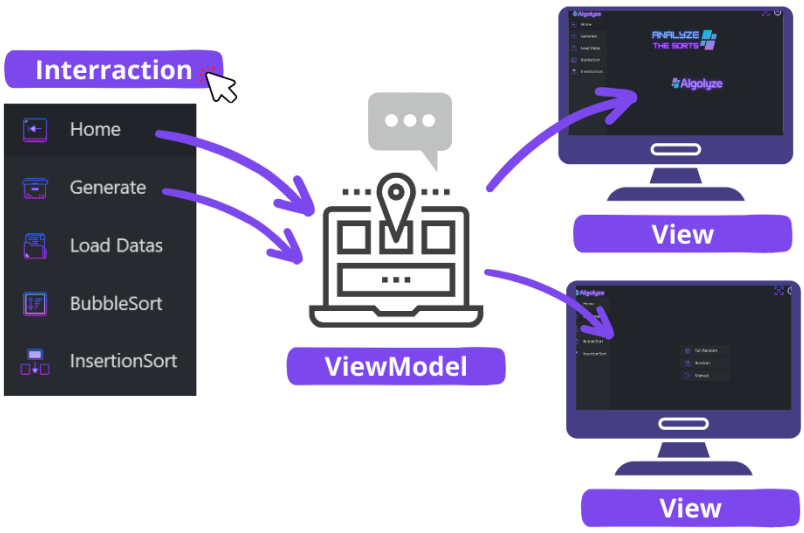
VII. MVVM alapú megvalósítás

Ebben a fejezetben a két – említett – rendezés megvalósításáról lesz szó és, annak megvalósítva az **MVVM** programtervezési minta szerint.

A feladat megoldása során **aszinkron** eljárásokat alkalmaztam. Mit is jelent ez? Mindenki futott már abba a jelenségbe, hogy a program ideiglenesen lefagy, miközben dolgozik [27]. Az aszinkron programok képesek ennek kiküszöbölésére, **párhuzamosan** dolgozni, azaz képes válaszképes maradni a felhasználói interakcióra miközben dolgozik [27]. Például fut a buborékrendezés vizuális megjelenése, ilyenkor minden más utasítás blokkolva lesz, amíg nem fejezi be. Az aszinkron programozásnak köszönhetően ezt a fajta lefagyást megkerültem. Így a program, miközben hajtja végre a rendezést, lehetővé teszi a

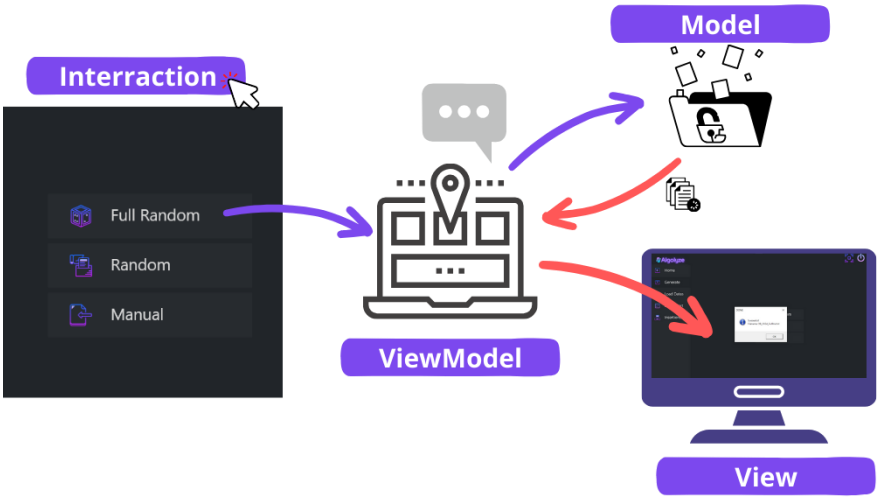
felhasználónak, hogy közben interakciókat hajtson végre. Egy aszinkron metódus típusa *Task<T>* [27].

Nézzük meg az alkalmazás működését. A oldalmenü interakciói nagyon egyszerűek is a működésük is triviális. A *Navigáció működés* példában a ***Home*** és ***Generate*** menüpontok kattintásának példáját láthatjuk.



28. ábra Navigáció működés

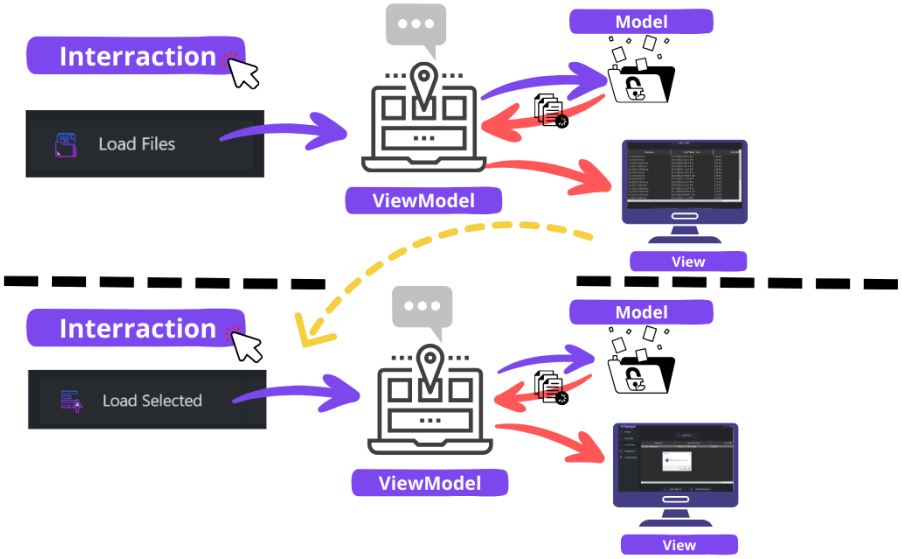
A kiválasztott funkció kattintásakor a ***NavgitonVM*** megkapja a gombhoz rendelt parancsot – adatkötés általál – így jelezve, hogy új nézetet szeretnénk megjeleníteni. A példa során a program helyes működésének sorrendjében fogok eljárni. A helyes sorrend, hogy először létrehozunk egy állományt a ***Generate*** menüponttal, amit már megjelenítettünk.



29. ábra Adat generálás folyamata

Ezt követően kiválasztjuk a tetszőleges almenüt – jelenlegi *Adat generálás folyamata* példában a ***Full Random*** opciót –, amellyel létrehozzuk az állományt.

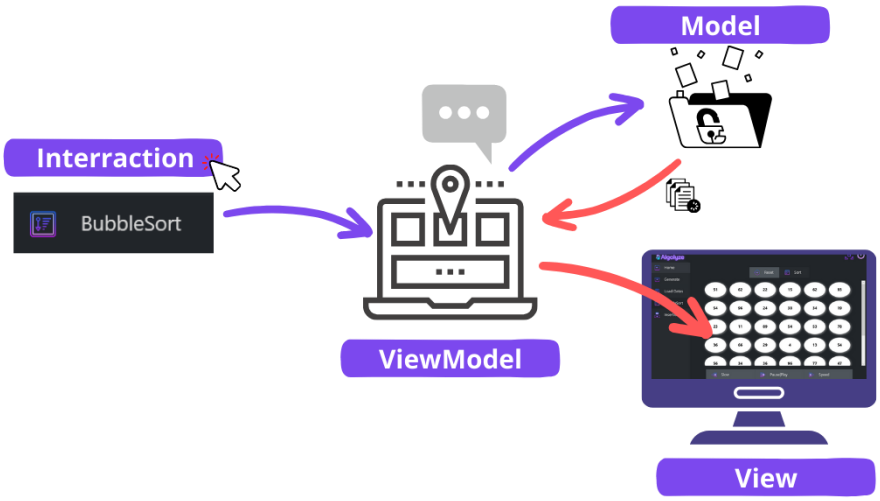
Az adat generálása után, töltsük be a létrehozott állományt.



30. ábra Betöltési folyamat

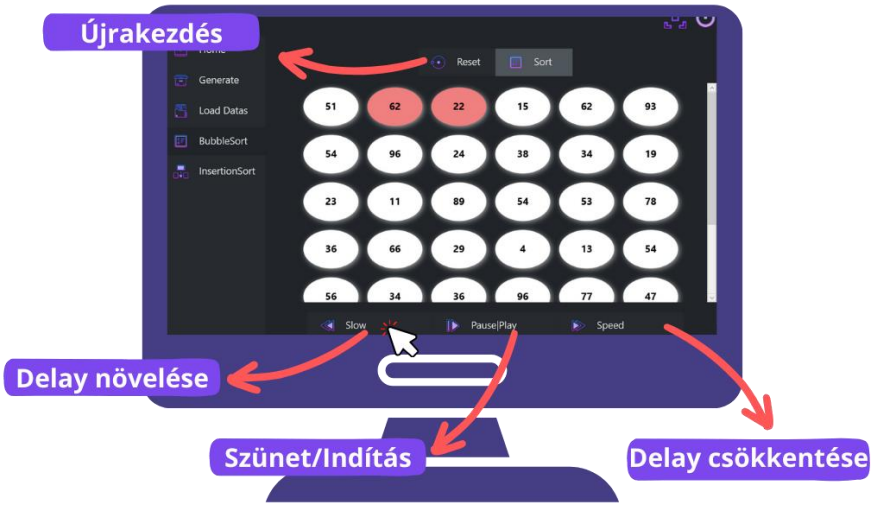
A *Betöltési folyamat* kicsit összetettebb, mint az előzőek. Szaggatott vonallal van elválasztva a két folyamat egymástól. Az egyik maga a ***fájlok betöltése*** az alkalmazás könyvtárából. A másik folyamat a **kiválasztott fájl betöltése**. A kiválasztott **állomány törlése** ugyanilyen metódussal zajlik.

A betöltött állományon hajtsuk vére a buborékrendezés funkcióját.



31. ábra Rendezés megjelenítése

Az eljárás hasonló, mint az előzőekben. A rendezés során az ovális alakzatok vizualizálják a buborékokat, amelyek a fájl számsorozatát tartalmazzák. Az éppen vizsgált elemeket kékkel, míg a rossz sorrendben lévő elempárokat *lightcoral* színnel jelöli, majd cseréli meg.



32. ábra Rendezés működése

A ***Sort*** gomb inaktív miközben a rendezés zajlik. A fent leírtak alapján, a rendezés aszinkron módon zajlik, tehát a ***Delay*** (késleltetés) csökkentése, növelés és a megállítás, indítás funkció elérhető, sőt még a ***Reset*** (újrakezdés) is. Az újrakezdés során a buborékokat visszaállítja a rendezés előtti fázisba.

A beszúró rendezés megvalósítás szinte ugyanaz a buborékrendezéssel, a különbség kettőjük között az animáció és az algoritmus logikája.

VIII. Alkalmazás használata

A fejezet az alkalmazás tájékoztatásáról szól. Említést teszünk a specifikációkról és, hogy a programfuttatásához, milyen rendszerkövetelmények szükségesek, hogy hiba nélkül futtatható legyen a felhasználók számítógépén.

VIII. 1. Általános specifikáció

Az alkalmazás az ***Algolyze*** nevet viseli, ami az algoritmus és az analizálás – az elemzés – szavakból konkatenálódik össze. A célközönségre nincs korlát, bárki, aki elég affinitást érez ahhoz, hogy megismerje az informatika eme területét, annak ajánlott. Az alkalmazás nyelve angol.

A főcélja, hogy a felhasználó betekintést nyerjen az algoritmusok működésébe és azon belül is a rendező algoritmusokba. A program vizuálisan mutatja be a felhasználónak, hogy az előre beépített rendező algoritmusokat, hogyan is kell elképzelni. A vizualizáció különféle alakzatokkal, elemekkel és színezésekkel illusztrálva mutatja be, hogy milyen logikával épül fel és hogyan végzi el a rendezést az algoritmus a kapott bemenetre.

VIII. 2. Rendszerkövetelmények

Minden szoftverhez tartozik egy minimális és ajánlott hardver-szoftver követelmény rendszer. A minimális specifikáció esetén a programfutása megvalósul, bár van rá esély, hogy nem a teljes felhasználói élményt nyújtja. Az alkalmazás futtatása előtt érdemes ellenőrizni, hogy az Ön **számítógépe alkalmas a programfuttatására**.

A futtatáshoz szükséges hardver és szoftver követelmények:

4. táblázat minimum követelmény

Operációs rendszer	Windows 10 , 32 bit
Processzor	2 gigahertz (GHz) vagy gyorsabb processzor
RAM	4 GB memória (DDR3 vagy feletti)
Szabad terület	5 GB
Videókártya	DirectX 9 kompatibilis
Microsoft bővítmény	Microsoft .NET FRAMEWORK

5. táblázat ajánlott követelmény

Operációs rendszer	Windows 10 PRO , 64 bit
Processzor	3 gigahertz (GHz) vagy gyorsabb processzor
RAM	8 GB memória (DDR3 vagy feletti)
Szabad terület	5 GB
Videókártya	DirectX 9 kompatibilis
Microsoft bővítmény	Microsoft .NET FRAMEWORK

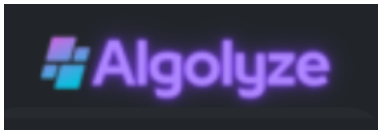
Érdemes megjegyezni, hogy a fent említett követelmények az alkalmazás fejlesztésének környezetéhez viszonyulnak ezért mindenképp ajánlott a Windows operációs rendszer. Az alkalmazás használatához ajánlott egy minimális angolnyelv tudás a nyelvezet miatt.

VIII. 3. Rövid használati útmutató

Az alkalmazás elindítása utána – ha sikeresen teljesültek a korábbi feltételek – az alábbi felület lesz az alapértelmezett indításkor:

33. ábra kezdőlap

A felület egy sötét, modernizált, *Cyberpunk* stílusú megjelenés. A bal sarokban látható az alkalmazás egyedi logója. A logóval egy sávban találhatóak az alkalmazás minimalizálására és nagyítására szolgáló gombok és mellette az alkalmazás bezárása, kikapcsolása funkció.



34. ábra logó

A logó (*34. ábra logó*) elkészítése **Canva**[6] segítségével készült. A Canva egy ingyenes kép, videószerkesztő weboldal, amelyhez tartozik előfizetési lehetőség is. Előfizetés esetén a felhasználó több opcióhoz, stílushoz és szerkesztői felülethez fér hozzá.

35. ábra ablak nagyítása, kicsinyítése

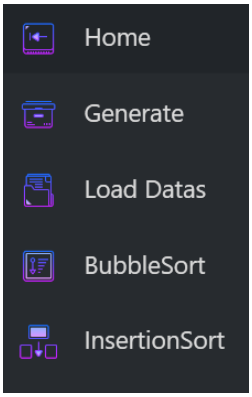
Az ablak méretezésére szolgáló gomb (*35. ábra ablak nagyítása,*) képes teljes képernyőre módosítani az ablak méretét. Ha az ablak már teljes képernyő módban van, akkor a gomb ismételt megnyomásakor visszatér az eredeti állapotába.

[6] https://www.canva.com/

Az alkalmazás bezárásához alkalmazzuk a jobb sarokban található kikapcsoló gombot.

36. ábra bezárás gomb

A baloldalt egy dashboardot láthatunk (*37. ábra dashboard, menüsáv*), ami a főbb funkciók menüjét prezentálja.



37. ábra dashboard, menüsáv

Az alkalmazás főbb funkciói:

- **Home**: a kezdőoldal, ahol az alkalmazás nevét és logóját láthatjuk.
- **Generate**: adatok (számok, sorozat) generálása a rendezésekhez. Ezeket az adatokat az alkalmazás mappájába menti le *txt* fájl kiterjesztéssel. Az adatok létrehozásában három opció van.
    - **Full random**, ahol a program két konstans intervallum között generál számokat és menti le a *txt* fájlba.
    - **Random**: Itt a program a felhasználói inputról beolvassa a tömb méretét és a véletlenszerű szám alsó és felső határát.
    - **Manual**: A program a felhasználótól egy konkrét listát, sorozatot vár, amelyet vesszővel elválasztva adhat meg.
- **Load Datas**: a generált fájlok közül tudunk választani, hogy melyiket töltse be a program.
    - **Load Files**: Először a programot utasítanunk kell, hogy a könyvtárából töltse be a generált állományokat.

    - **Load Selected**: A betöltött majd kiválasztott fájl – ha nem üres a könyvtár – betöltése.
    - **Delete Selected**: A kiválasztott fájl eltávolítása a gyökérkönyvtárból.
- **BubbleSort**: A buborékrendezés funkció csak akkor érhető el, ha a felhasználó már előzőleg sikeresen betöltött a **Load Datas** funkcióval egy állományt. Ellenkező esetben a program figyelmeztet minket a hiányosságra. Az állományban lévő számokat megjeleníti kis ovális szövegdobozokban egymás mellé. További funkciók elérhetőek:
    - **Reset**: A számokat az eredeti sorrendbe rakja, ahogy a fájlban is voltak. Tehát újra kezdhetjük a már rendezett készletet.
    - **Sort**: Maga a rendező algoritmust hajtja végre elempáronként vizuálisan. Mindig két szomszédos elemet hasonlítja össze egymással. Ha a sorrend nem megfelelő, helyet cserélnek.
    - **Sort**: Azonnal rendezi az elemeket.
    - **Slow**: Az algoritmus lassítása – késleltetése. A késleltetésnek van egy maximum ideje, azaz nem mehetünk egy bizonyos pont alá.
    - **Play/Pause**: A gomb kattintásakor leállítjuk az algoritmust, szüneteltetjük. Ha még egyszer megnyomjuk a gombot, akkor újra indul az algoritmus, ott ahol előzőleg megállítottuk.
    - **Speed**: A Slow funkció ellentéte, a gyorsítás. Hasonlóan működik, van egy felsőhatára, annál feljebb nem tud gyorsulni az algoritmus.
- **InsertionSort**: A beszúró rendezés funkcióra ugyanaz felépítés és kritérium érvényes, mint a buborékrendezésre, de a rendezés logikája változik.

IX. Továbbfejlesztési lehetőség

A program nagyon sok lehetőséget és egyben akadályt rejt magában. A már megírt funkciókat a következőképpen fejleszteném tovább:

A **fájl létrehozása** egyelőre csak egész számokat tud kezelni, más típust nem. Ez módosítható lenne, hogy dinamikusan minden adattípust kezeljen. Olyan összetett adatszerkezet szükséges a megvalósításához, amely képes kezelni több típust is egyszerre. Ez megvalósítható lenne olyan listával, ami szövegeket kezel. A szöveg – string – minden típust képes kezelni, és az összehasonlítás után történne meg a konverzió.

A **fájlok betöltése** egyelőre csak egy kritériumnak felelnek meg, csak *txt* kiterjesztésű állományokat képes beolvasni, amely a saját gyökérkönyvtárában helyezkedik el. Ezek a fájlok elválasztása soronként van. Ez szintúgy bővíthető lehetne, hogy képes akár *csv*, *xls* vagy egyéb kiterjesztés is kezelni, és az alapján elválasztani az adatokat egymástól. Például a *csv* kiterjesztéseknél pontosvessző lenne az elválasztás az adatok közt és nem a sortörés.

A program egyelőre csak kettő ismertebb helyben rendező algoritmus végrehajtására képes. A **továbbiakban** ez **bővíthető** lenne akár **rekurzív rendező algoritmusokra** is, például összefésülő rendezés. Ezt követően a **gráfalgoritmusokra** is ki lehetne térni. például Krusal, Prim, Dijkstra algoritmusokra. A Prim és Kruskal algoritmus megvalósításában egy szomszédsági mátrixot tudunk létrehozni, ami a programozásban egy kétdimenziós tömb. Itt eltárolhatók az egyes élek és közöttük lévő kapcsolatokat, hogy egyes pontok között fut-e él vagy sem.

A **vizuális megjelenítésben** még több potenciál van. Például a beszúró rendezés során az adott „kártya” kivételét és beszúrását illusztrálni a megfelelő helyre, amely jobban szemléltetné magát a rendezést.

X. Összegzés

A feladat megoldása során több akadályt is megkellett ugrani. Első és legfontosabb lépésként, hogy a saját komfort zónámból lépjek ki és a kutatásban mélyedjek el. A források, szakirodalmak keresése és feldolgozása. Fontos, hogy több szakirodalmat is megjelöltem az erős alapok bizonyítására. A második lépésben maga az elméleti alapokat átültetni a gyakorlatba.

A programom megvalósítása során törekedtem minden tudásomat felhasználni, amit az elmúlt évek alatt elsajátítottam. A kutatásom fő témája az algoritmusok és az MVC, MVVM programtervezési minta tanulmányozása C# nyelven.

A program megvalósítása előtt több kutatást is kellett végeznem. Elsősorban megismerni a C# nyelvet és a WPF grafikus felületet. A WPF első lépésében az XAML előnyeit kellett megismernem és a legfontosabb részt, az adatkötéseket, ami elengedhetetlen az MVVM szabvány implementálásra. Miután az eszközöket megismertem a következő megismerendő témakör az algoritmus elmélet, hogy egyes algoritmusok, hogyan is működnek és hogyan lehetne őket egyszerűen és mélybenyomást keltve szemléltetni őket.

Egy olyan segítő programot valósítottam meg, amely azok számára hasznos, akik esetleg nem tudják teljes mértékben elképzelni, hogy egy rendezés során milyen lépéseket tesz meg az algoritmus. A program lehetőséget biztosít pár alapvető fájlkezelés funkcióra és a két rendezés végrehajtására a betöltött állományokból. Továbbá a rendezés szimulálja az adott algoritmus folyamatát. Vegyük például a buborékrendezést. A program a betöltött állomány adatait egy ovális alakzatú, buborék mintára hasonlító objektumba helyezi el a tartalmat. Ezeket a buborékokat az implementált logikával végrehajtja az összehasonlításokat és vizualizálva különböző színekkel. Az algoritmus működésének gyorsaságát lehetősége van a felhasználónak módosítani.

XI. Irodalomjegyzék

[1] W. S. Björn Engquist, Mathematics Unlimited: 2001 and beyond, Springer, 2001.

[2] I. G. S. R. Rónyai Lajos, Algoritmusok, 1998.

[3] C. E. L. R. L. R. C. S. Thomas H. Cormen, Introduction to Algorithms (Fourth Edition), MIT Press, 2022.

[4] J. E. H. J. D. U. Alfred V. Aho, Data Structures and Algorithms, 1983.

[5] G. P. Lovász László, Algoritmusok, Műszaki könyv kiadó, 1978.

[6] B. Norbert, Függvények, Gödöllő, 2022.

[7] É. T. Jon Kleinberg, Algorithm Design, 2005.

[8] M. A. Weiss, Data Structures and Algorithm Analysis in C++(Fourth Edition), Florida International University, 2013.

[9] K. Balázs, Hogyan válj jól fizetett C# programozóvá?, Budapest: BBS-INFO, 2019.

[10] H. Jain, Problem Solving in Data Structures & Algorithms Using C: First Edition, Bhopal, India, 2017.

[11] D. R. Brad Miller, Problem Solving with Algorithms and, 2013.

[12] R. Vystavěl, „C# Programming for Absolute Beginners," Apress, 2017.

[13] G. László, „Intel processzorok programozása assembly nyelven," Pécs, 2015.

[14] J. Albahari, C# 9.0 in a Nutshell, O'Reilly, 2021.

[15] „C# Primitive Datatypes," [Online]. Available: https://condor.depaul.edu/sjost/nwdp/notes/cs1/CSDatatypes.htm.

[16] K. V. Ç. S. M. T. K. Ani Kristo, The Case for a Learned Sorting Algorithm, Portland, USA, 2020.

[17] R. Sedgewick, Algorithms Fourth Edition, Pearson Education, 2011.

[18] Microsoft, „What is Visual Studio?," Microsoft, [Online]. Available: https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2022.

[19] C. Zoltán, Fordítóprogramok, Budapest, 2007.

[20] Microsoft, „Desktop Guide (WPF .NET)," Microsoft, [Online]. Available: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/overview/?view=netdesktop-8.0.

[21] P. C. R. Len Bass, Software Architecture in Practice Third Edition, Addison-Wesley, 2012.

[22] A. Osmani, Learning JavaScript Design Patterns, O'Reilly, 2012.

[23] M. Péter, Szoftverfejlesztés, Miskolci Egyetem Általános Informatikai Tanszék, 2017.

[24] S. András, „Fejlesztési modellek és módszertanok,” [Online]. Available: https://szit.hu/doku.php?id=oktatas:programozas:fejlesztesi_modellek_es_modszertanok.

[25] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Unified Process, Pearson , 2004.

[26] G. Béla, „Aktivitás diagram,” [Online]. Available: https://gyires.inf.unideb.hu/KMITT/c02/ch10s04.html.

[27] S. Cleary, Concurrency in C# Cookbook, O'Reilly Media, 2014.

XII. Ábrajegyzék

NYILATKOZAT
az írásmű eredetiségéről

(PTE SZMSZ 5. sz. mellékletének 14/1. számú melléklete alapján)

Alulírott FENYVESI JÁNOS ADRIÁN
LVYBOL (NEPTUN kód), büntetőjogi felelősségem tudatában kijelentem,
hogy ALGOLYZE
című írásomban foglaltak saját, önálló munkám eredményei, ennek elkészítéséhez kizárólag a hivatkozott forrásokat (szakirodalom, eszközök stb.) használtam fel, írásomat a Pécsi Tudományegyetem vonatkozó szabályzatainak betartásával készítettem. Tudomásul veszem, hogy a szerzői jogi szabályok betartását a Pécsi Tudományegyetem plágiumkereső rendszeren keresztül ellenőrizheti.

Pécs, 2024. év MÁJUS hó 05. nap

hallgató aláírása