Tartalomjegyzék

[Jelölésjegyzék 8](#_Toc99107193)

[1. Fejezet 9](#_Toc99107194)

[1.1. Alfejezet 9](#_Toc99107195)

[1.1.1. Alfejezet 9](#_Toc99107196)

[2. Új fejezet 10](#_Toc99107197)

[2.1. Új alfejezet 10](#_Toc99107198)

[3. Instrukciók 11](#_Toc99107199)

[Irodalomjegyzék 12](#_Toc99107200)

[Mellékletek 13](#_Toc99107201)

[Ábrajegyzék 14](#_Toc99107202)

[Táblázatjegyzék 15](#_Toc99107203)

Bevezetés

A modern számítógépes rendszerek alapvető része a feldolgozott és feldolgozandó adatok tárolása. Az adatok tárolása adatstruktúrákban történik, melyek mind elméleti, mind megvalósításbeli tulajdonságai alapvetően határozzák meg egy szoftver, vagy szoftverek rendszerének teljesítményét. A modern számítástudomány számos eszközt, összetett adatstruktúrát kínál, melyek között szinte minden feladatra találunk alkalmasat. Az adatstruktúrák közvetlen felhasználáson túl, náluk összetettebb adatstruktúrák alkotóelemeként is használhatóak.

Munkám során az adatstruktúrák egy olyan alcsoportjában szeretnék egy új alternatívát bemutatni, amely szinte minden összetett adatszerkezet szerves kihagyhatatlan eleme, és mégis jelenleg csak kevés tagját ismerjük, használjuk. A tömb, vagy általánosabban, az index alapon konstans időben elérést biztosító adatstruktúrák szinte mindenhol jelen vannak, annak ellenére, hogy ezekbe a beszúrás eddig ismert megoldásokkal csak a teljes tömb méretével egyenes arányban növő időben volt lehetséges.

Jelen munkámban, szeretném bemutatni, a gyorsított tömböt, amely a klasszikus tömbnek, illetve annak kényelmesebb, modernebb, de alapvetően azonos előnyökkel és hátrányokkal bíró változatainak egy gyorsabb beszúrási és törlési idejű alternatívája lehet.

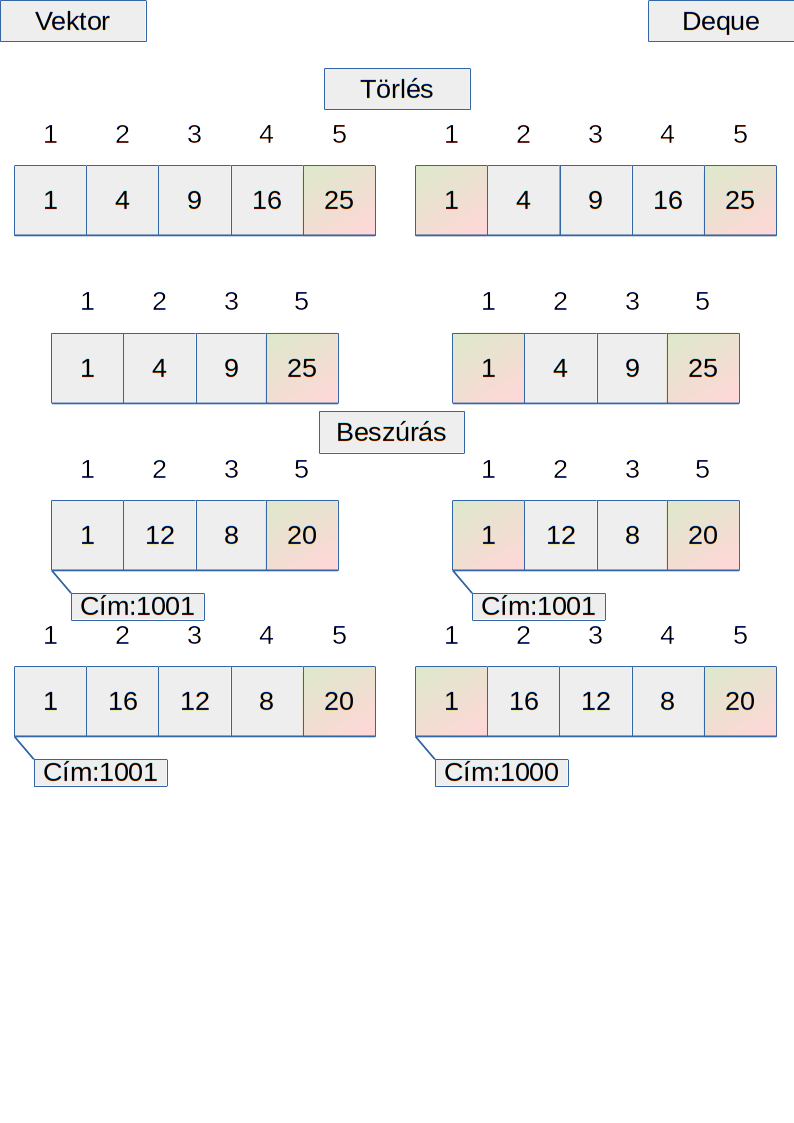
A gyorsított tömb, a hagyományos tömbökhöz hasonlóan konstans idejű index alapú elérést biztosít, de ezen túl képes gyök N időben beszúrást és törlést megvalósítani. Mindehhez gyök N-es többlet memóriahasználat tartozik.

A dolgozatomban szeretném, mindennek a megvalósítását, és a fent említett tulajdonságok bizonyítását bemutatni, összevetve, a jelenleg ismert legjobb alternatívákkal.

1. Jelenlegi állapotok

1.1 Dinamikus tömbök koncepciója

Jelenleg a szekvenciális tárolók között a dinamikus tömbök (vektor és deque) egyeduralkodóknak számítanak, annak ellenére, hogy ezeknél a beszúrás és törlés művelet igen lassú, mivel a konstans idejű index alapú elérés rendkívül nagy előny. A dinamikus tömbök egyszerűen egymás utáni helyeken tárolják az adatokat több helyet lefoglalva, mint ami kell, és középre beszúrás, törlés esetén elmozdítják az összes adattagot a nagyobbik( vektor) vagy közelebbiek) vége felől/felé.

1. Ábra Vektor és Deque alapszerkezet

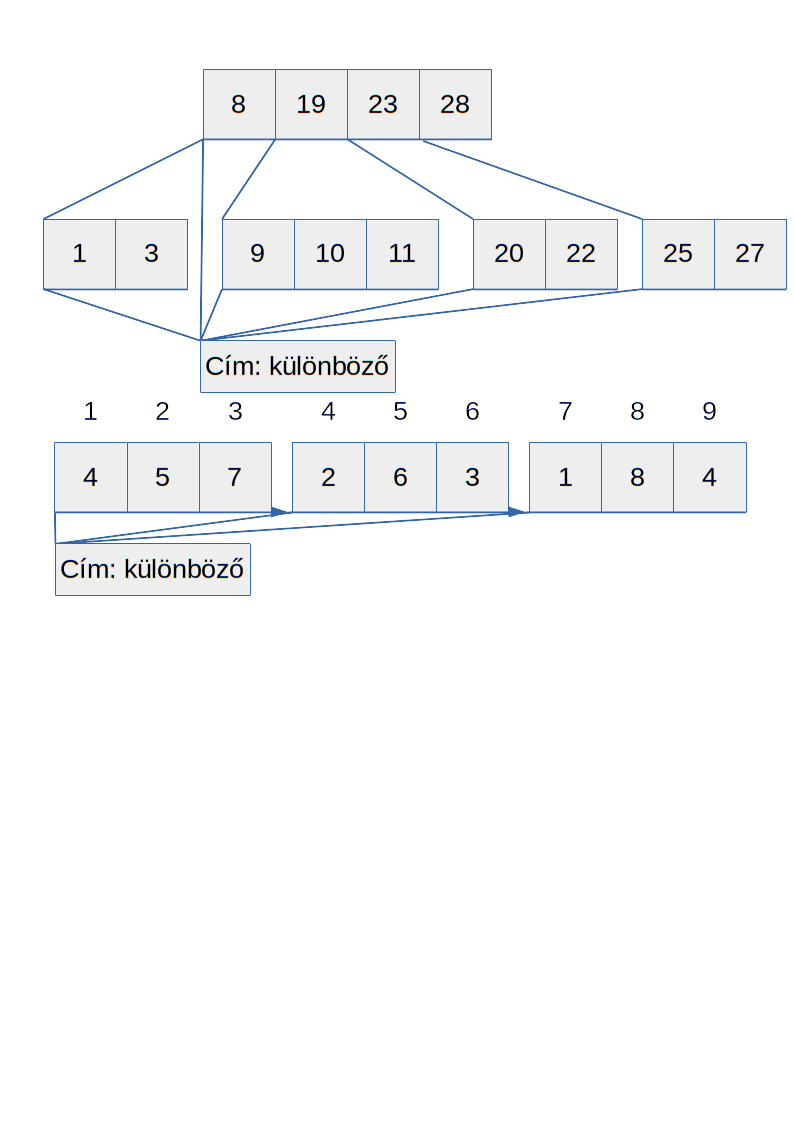
1.2 Dinamikus Tömbök tulajdonságai

Az index alapú elérés legrosszabb esetben is O(1), míg a mutáció legrosszabb és átlagos esetben Ordó=Theta(n). Ugyanakkor fontos tényező, hogy a vektorok egyik, és a deque mindkét, végén konstans időben lehet beszúrni és törölni, ha a memória újrafoglalástól eltekintünk. Ettől azért megengedett eltekinteni, mivel az újrafoglalásnál általában kétszeres méretű új tömböt foglalunk, így ahogy a deque mérete tart a végtelenhez, az egy elemre jutó esély arra, hogy újra kell foglalni a memóriát 1/n, míg a szükséges munka( másolás) n, így a várható munka: (1/n)\*n, ami 1. Ugyanakkor az is egy lehetőség, hogy egy modern rendszerben, a közel betelt dequeket, egy háttérszál újrafoglalja, még az előtt, hogy az átméretezés megakasztaná a programot.

2 Használat

2.1 Közvetett használat

.Ezeket a tároló adatszerkezeteket, a közvetlen felhasználáson túl, a komplexebb adatstruktúrák felépítésére is használják.

2. Ábra Vektor és Deque közvetett felhasználása

Így a modern adatbázisok hiába használnak fákat (többnyire a B fa valamely továbbfejlesztett változatát, mint a B+ Fa), közvetve, még mindig függnek a felhasznált szekvenciális tárolók sebességétől. A valós sebességet általában a háttértárak elérési ideje határozza meg, viszont: az éppen használt adat a RAM-ban van tárolva, így az azokon történő módosítási idő a memóriában végzett műveleteken múlik, amit a tárolók nagyban meghatároznak.

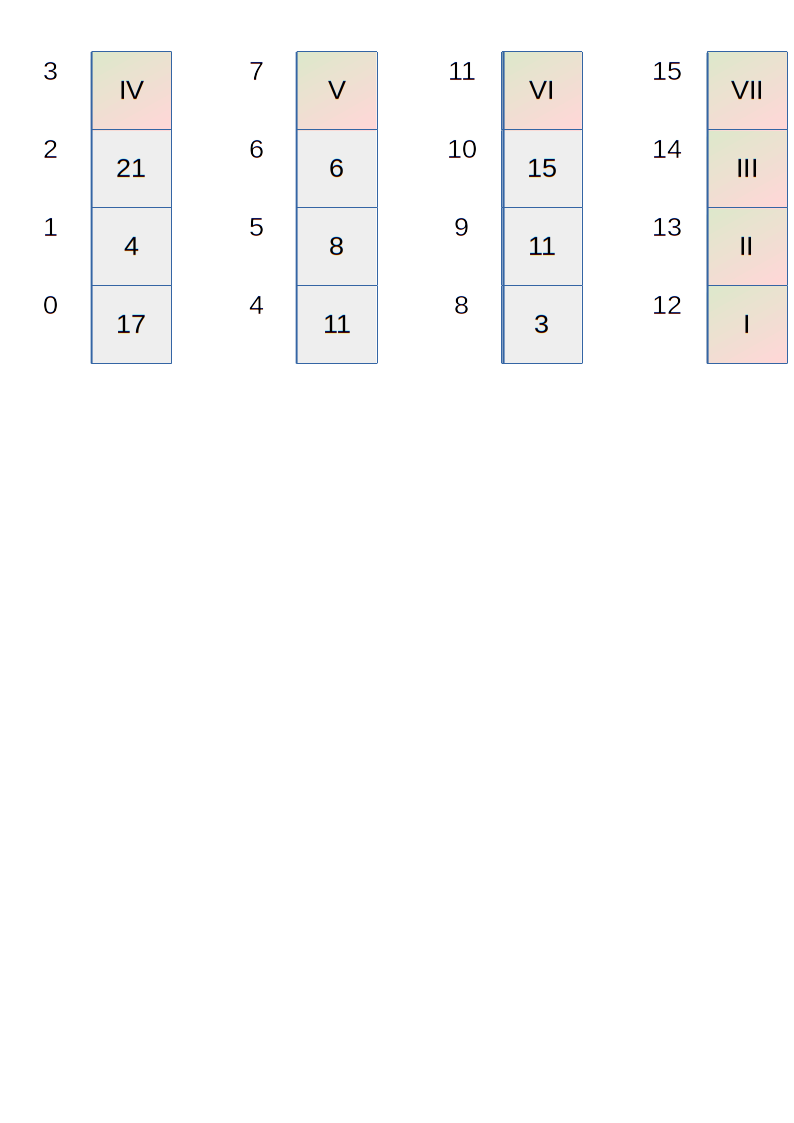
2.2 Közvetlen használat

Amennyiben az elérések aránya nagyon magas a mutációkhoz képest, előfordul az is, hogy a tömbbe beszúrás ideje egy elfogadható szükséges rossznak van tekintve.

2.3 B fák

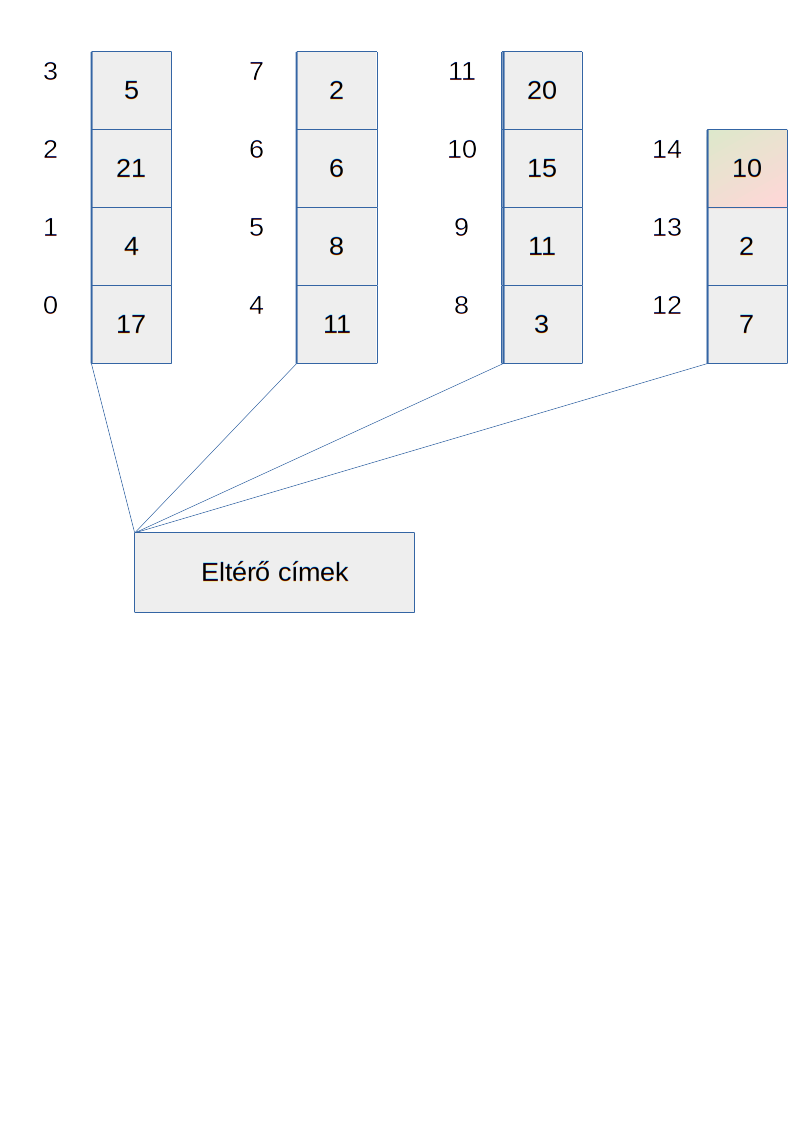
3. Gyorsított Tömb

3.1. Alapötlet

A gyorsított tömb, a beszúrásnál szükséges nagy mennyiségű adatmozgatást úgy kerüli el, hogy az adatokat nem egy darab nagy méretű vektorban, hanem gyök N, vagy gyök N+1 darab kisebb vektorban tárolja, amelyeknek a hossza is gyök N, vagy gyök N+1.

3. Ábra négyzet alakú eset

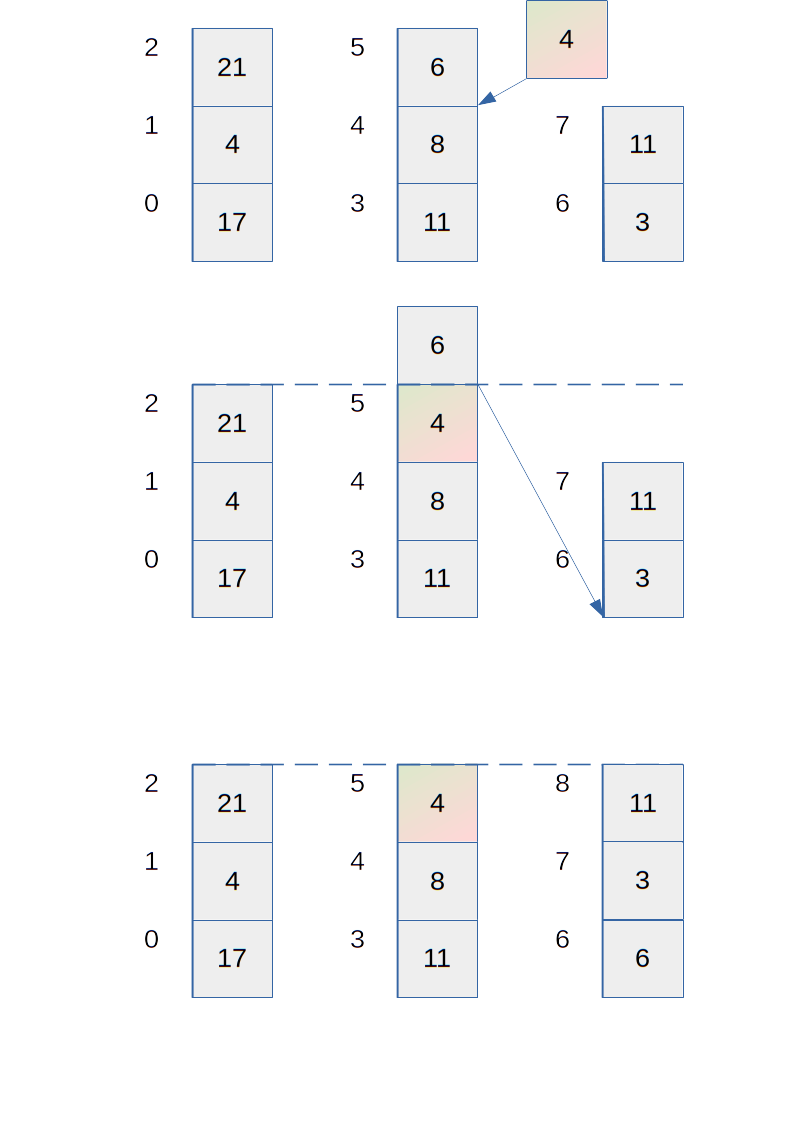
Az adatok sorrendje, a elsősorban a tároló tömb helye sorrend szerint, majd a tárolón belüli pozíció. Az alsó tömbök, cím szerint egy felső tömbben vannak tárolva.

4. Ábra tárolási és logikai sorrend összekapcsolása

3.2 Leírás

.Amikor ezeknek a közepébe beszúrás történik, az alsó vektor hosszával arányos adatmozgatás, csak gyök N darab adat mozgatását teszi szükségessé. A teljes tároló növekedése úgy történik, hogy először egy gyök N+1-edik vektor van feltöltve alulról fölfelé, majd minden meglévő vektor végére egy új elem kerül. Mivel az elemek sörrendje elsősorban a tartalmazó vektortól függ, így ugyanannak az elemnek ugyanott történő tárolását jelenti logikailag, ha az x-edik vektor végén, vagy az x+1-edik vektor elején van.

Ahhoz hogy a kis, alsó vektorok közel gyök N méretűek maradjanak, szükséges, ezeket kiegyensúlyozni. A kiegyensúlyozásnál, a többletet tartalmazó alsó tömb irányából, a következő elem helye felé történik. A popFront vagy popBack műveletekkel, a többlet eltávolításra kerül, és bekerül a következő altömbbe.

5. Ábra Kiegyensúlyozás

innen ugyanígy van továbbvíve amíg el nem ér a megfelelő helyre. Mivel a Popfront, PopBack, PushFront és PushBack mind konstans idejű műveletek, és legfeljebb gyök N alsó tömbön kell átmenni, maga az egyensúlyozás is gyök N-es.

3.3. miért négyzet:

Mivel a kiegyensúlyozásnál az alsó tömbök hosszával arányos a beszúrás/törlés időigényének egyik része és a felső tömb hosszával arányos a másik része, ez akkor minimális ha a kettő megegyezik, mivel azonos területű téglalapok szomszédos oldalainak összege akkor minimális, ha egy négyzetről van szó.

3.4. kiegyensúlyozás

A beillesztés majd kiegyensúlyozás az alapötlet a piros fekete fák és sok más kiegyensúlyozott adatszerkezet mögött. A itt kiegyensúlyozás kihasználja a dequek azon tulajdonságát, hogy a doublestack műveletek mind konstans amortizált időben futnak le ( nagyobb tároló újrafoglalásától eltekintünk).

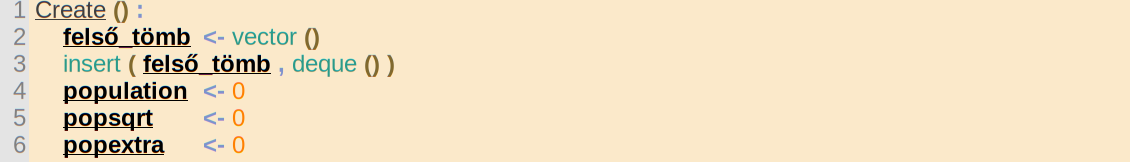
3.5 hasonló koncepció

A hasított tömbfa (HAT) közel gyök N darab gyök N hosszú elemet tartalmaz, de beszúrás/törlés esetén az egész adatszerkezetet újraépíti lineáris időben.

4. Bemutatás alapműveleteken keresztül

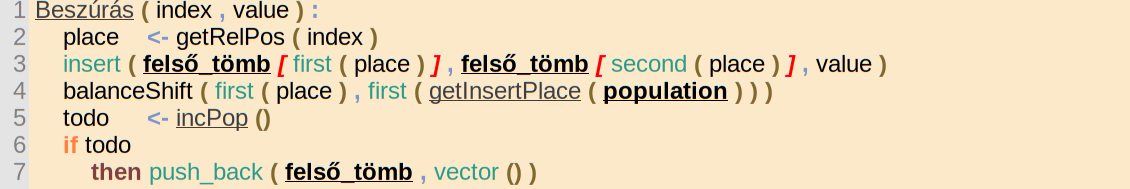
4.1 Létrehozás

A felső, és a legelső alsó 2 tömb lefoglalása, belső adatokat tároló konstans méretű struktúra létrehozása és adatokkal feltöltése.

 4.2. megsemmisítés

Az összes alsó majd a felső tömb megsemmisítése.

4.3. Beszúrás

Adott indexre, ami alatt itt a logikai index értendő. Először megkeressük a tényleges adatszerkezet-beli helyet, ami a megfelelő felső tömböt, és az azon belüli indexet jelöli. Ide megtörténik a beszúrás, majd a Balansceshift művelet eltolja a többlet elemet a növekedési pont felé. Ezek után a gyorsított tömb belső adatai konstans időben újraszámításra kerülnek.

4.4 Törlés

A beszúráshoz hasonlóan, megkeressük a tényleges helyet, GetRelPos-sal, megtörténik a törlés majd a csökkenési hely felől megtörténik a kiegyensúlyozás BalanceShift-tel. Ezek után a belső adatok újraszámításra kerülnek.

4.5 olvasás adott helyen

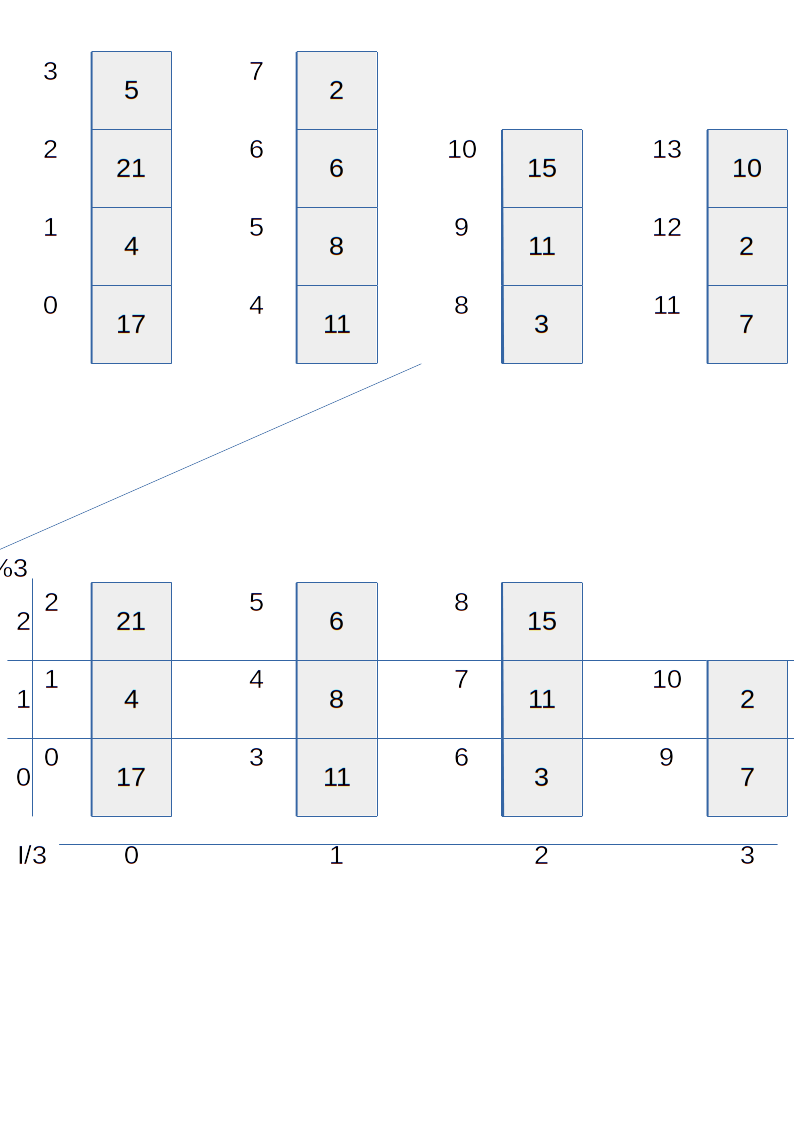
Tényleges hely megkeresése, majd elérése olvasásra.

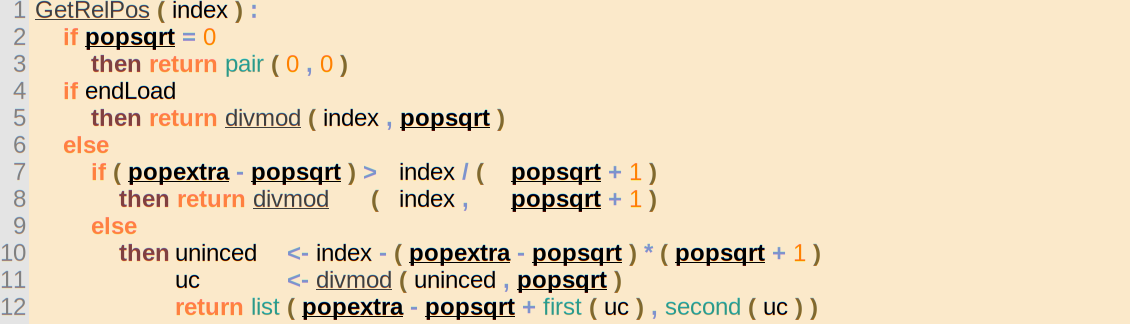
4.6  Írás adott helyen

 Tényleges hely megkeresése, majd elérése írásara.

4.7 GetRelPos:megkeresi az alsó vektort ahol a keresett elem van, és azon belül az indexet, i-edik elemre:

* 0-nál 0,0
* ha a gyök N+1-edik vektor még nem telt be: i DIV gyökn,i MOD gyök N
* Ha betelt: kiszámoljuk i DIV (gyök N+1)-et, ami, ha kisebb mint a vektorok végein tárolt többlet akkor visszatér:i DIV (gyökn+1),i MOD (gyök N+1) ha nem, akkor kivonjuk az így tárolt elemek számát, és a különbségből legmaradt értékkel, a második esethez hasonlóan számolunk, azzal, a különbséggel, hogy a hányadoshoz hozzáadjuk i DIV (gyök N+1)-et.

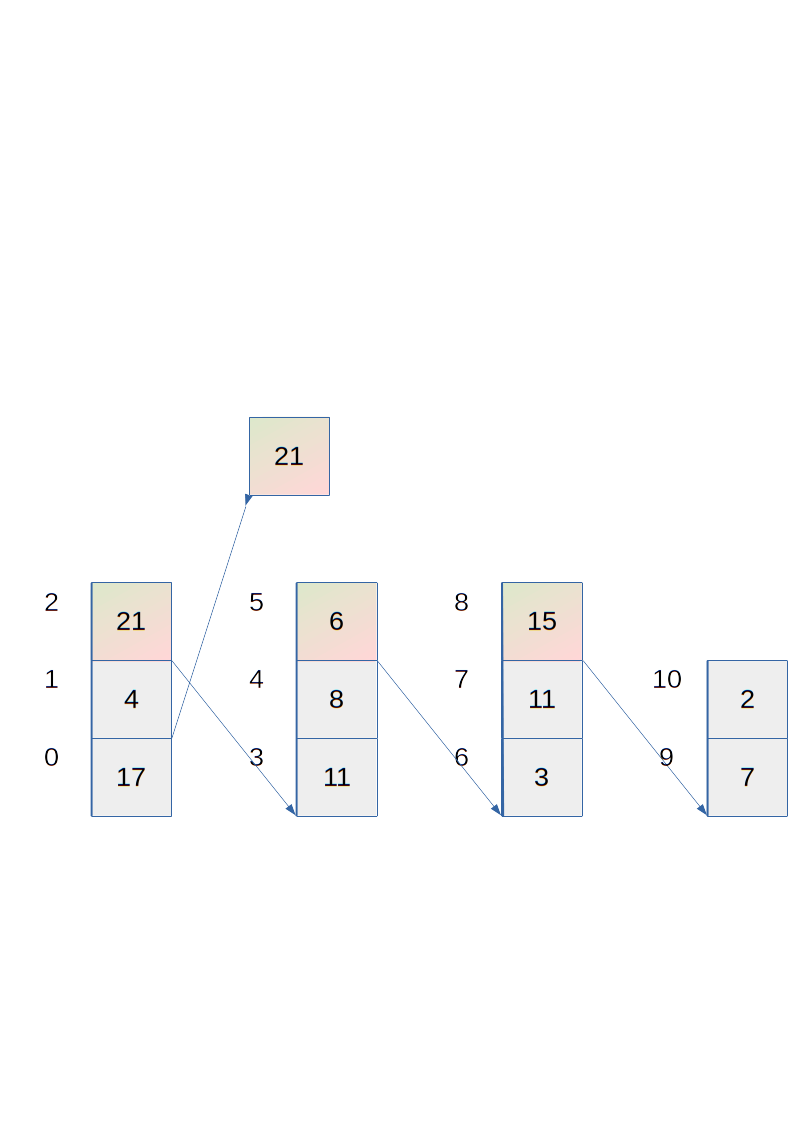


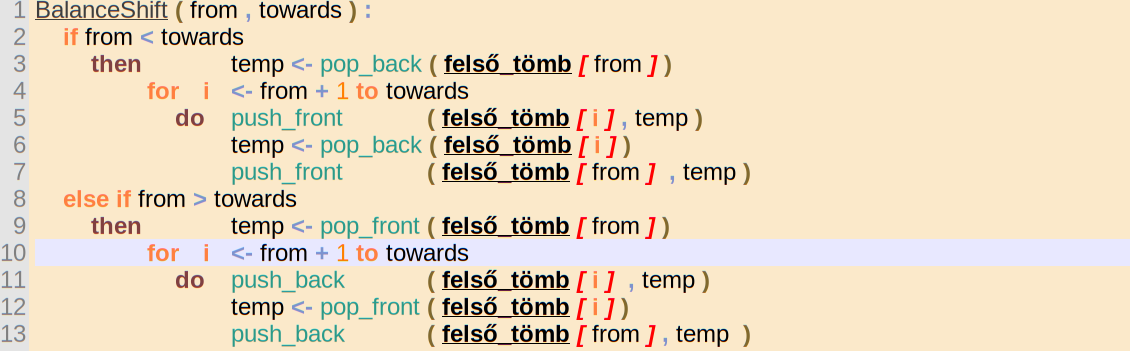


4.8 Keresés és egyéb műveletek, amikre nem lett szánva

4.9 BalanceShift:

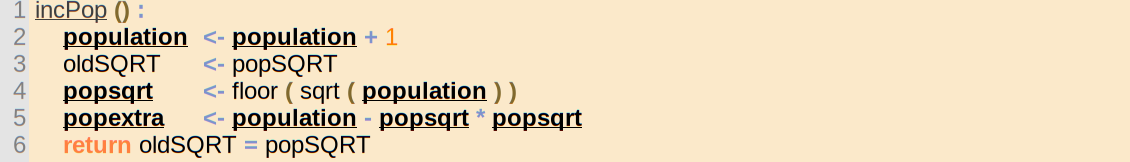
A mutáció helyét megkapja, és onnan, beillesztésnél a newPlace, törlésnél a deletePlace felé ellentétes push és pop műveletekkel végighordja a többletet az alsó tömbök között.





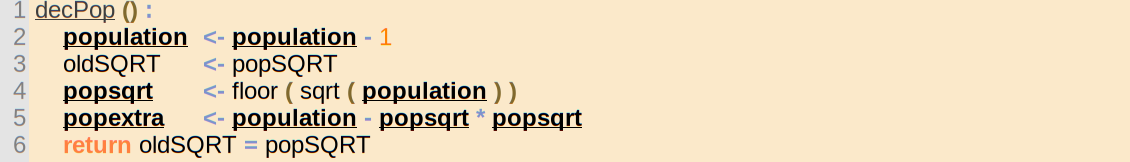
4.10. incPop():

Populáció eggyel növelése.

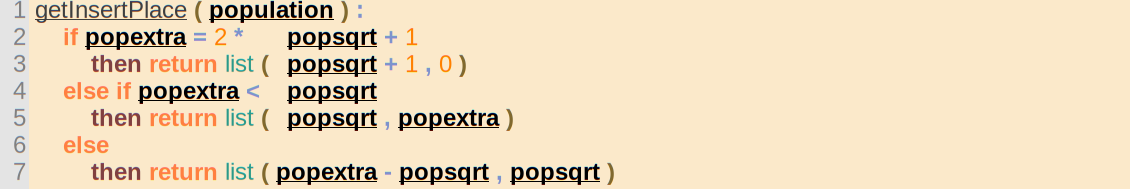


4.11. decPop():

Populáció eggyel csökkentése.

4.12 getInsertPlace:

növekedés helyét adja vissza, az implementációban mutációnál ki van számolva egy változóba, és le van kérdezve



5. Elemzés

5.1 Esetek

3×3(legjobb, átlagos, legrosszabb/omega, theta, ordo)

Létrehozás:

Konstans darab memóriafoglalás( legalább 2,az első alsó és a felső tömbnek, de érdemes lehet többet) pontosan 4 változó értékadással deklarálása, majd az egyik 1 lépésben módosítása

Theta=Omega=Ordo=1

Felszámolás:

Felső tömb minden elemét felszabadítani, majd magát a felső tömböt is, gyök N+2 törlés összesen Leggyorsabb esetben csak 1 alsó tömb van

Omega=1

Theta=Ordo=gyök N

Beszúrás:

A beszúrás nem tartalmaz önmagában iterációt vagy rekurziót, így a hívott tagok döntik el a sebességét. Kettő meghívott függvény van, ami nem konstans, a deguebe tetszőleges helyre illesztés és a BalanceShift. Várhatóan, és legrosszabb esetben is a deguebe szúrás gyök N időben fut le, mivel a deque közel gyök N hosszú (vagy pontosan, vagy gyök N+1). Legjobb esetben a végére szúrunk be aminek köszönhetően konstans lehet a beszúrási idő. A BalanceShift legjobb esetben szükségtelen, de erre csak 1:gyök N-hez az esély (annak a valószínűsége, hogy az összes alsó tömb közül, pont a növelendőbe esik). Egyébként 2 lehetőség van, az hogy az adatszerkezet a végén növekszik, vagy, az, hogy egyenletesen elosztva, valamely tetszőleges alsó tömbben, egyenlő eséllyel. Mindegyikre 50% esély van. Az első esetben "mean line segment length" alapján hossz/3, ami itt gyök N/3, másik esetben átlagos távolság végponttól, ami hossz/2, jelen esetben gyök N/2, így átlagosan 5/12-ed gyök N, vagyis théta gyök N-es. Legrosszabb esetben, mindkét helyzetben, gyök N mozgatást igényel.

Omega=1

Theta=Ordo=gyök N

Törlés:

helyben elérés:

omega=theta=ordo=1

incPop():

pontosan 5 szekvenciális lépés minden esetben

omega=theta=ordo=1

getRelPos(index):

egy fix méretű döntési fát jár be, konstans mennyiségű feladattal minden ágon, így konstans minden esetben

omega=theta=ordo=1

6. mérések

6.1. Mérés menete

A mérések elvégzéséhez, egy C++-ban megírt implementációt használtam. Az implementáció a pszeudokódok alapján történt, komoly optimalizálások nélkül, mivel elsősorban a validálás volt a célja. A később említett további optimalizálások egyikét sem használtam.

A mérések során ugyanazokat a lépéseket hajtottam végre, egy C++ beépített könyvtári std::vektorra, és a Gyorsított Tömbre. A mérések 20-5120 elemre történtek, többszöri ismétléssel. A mérések között várakozások voltak beiktatva, hogy a mérések a lehető legkisebb hatással legyenek egymásra, viszont, minden eset (elem és műveleti arány kombinációja adott méretű elemekkel) mérése egyben lett elvégezve, így nagy elemszámnál ez torzíthatta a végeredményt. A mérések során a mutációk és az elérések aránya is változott, 1:1-től 1:256-ig, a mutációk javára. A mért értékek, a futási idők hányadosát mutatják, a Gyorsított Tömb idejével, a számlálóban.

6.2. Mérés eredménye

1. Tábla 32bites floatok tárolása, és annak sebessége

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| elem/arány | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 |
| 20 | *15.77* | *5.38* | *12.95* | *14.41* | *11.89* | *15.83* | *12.22* | *15.83* | *11.82* |
| 40 | *11.37* | *8.27* | *12.52* | *12.64* | *12.60* | *10.79* | *9.91* | *8.00* | *11.84* |
| 80 | *9.36* | *5.10* | *8.92* | *5.83* | *7.41* | *8.05* | *7.30* | *7.31* | *6.70* |
| 160 | *6.09* | *4.43* | *4.83* | *4.80* | *5.37* | *4.01* | *5.47* | *4.04* | *5.39* |
| 320 | *4.61* | *3.99* | *4.49* | *3.59* | *4.28* | *4.12* | *2.45* | *2.93* | *2.88* |
| 640 | *3.10* | *2.18* | *1.74* | *1.23* | *1.86* | *1.81* | *1.63* | *1.60* | *1.88* |
| 1280 | *1.20* | *1.31* | *1.55* | *1.53* | *1.32* | *1.70* | *0.92* | *1.31* | *1.42* |
| 2560 | *1.30* | *1.17* | *1.16* | *1.13* | *1.02* | *1.24* | *1.02* | *0.99* | *1.15* |
| 5120 | *0.94* | *0.85* | *0.85* | *0.78* | *0.84* | *0.83* | *0.58* | *0.78* | *0.75* |

2. Tábla 8192byte-os elemek tárolása, és annak sebessége

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 |
| 20 | *1.066* | *0.684* | *1.063* | *1.397* | *1.728* | *1.878* | *1.396* | *1.644* | *1.222* |
| 40 | *1.040* | *0.720* | *1.155* | *1.091* | *1.024* | *0.852* | *0.653* | *0.487* | *0.718* |
| 80 | *0.580* | *0.359* | *0.554* | *0.328* | *0.443* | *0.486* | *0.482* | *0.510* | *0.425* |
| 160 | *0.381* | *0.292* | *0.318* | *0.353* | *0.365* | *0.244* | *0.324* | *0.215* | *0.235* |
| 320 | *0.176* | *0.167* | *0.145* | *0.100* | *0.117* | *0.111* | *0.077* | *0.077* | *0.072* |
| 640 | *0.101* | *0.074* | *0.061* | *0.048* | *0.056* | *0.053* | *0.050* | *0.047* | *0.048* |
| 1280 | *0.057* | *0.044* | *0.038* | *0.034* | *0.031* | *0.035* | *0.027* | *0.030* | *0.031* |
| 2560 | *0.037* | *0.029* | *0.025* | *0.024* | *0.023* | *0.026* | *0.024* | *0.023* | *0.025* |
| 5120 | *0.027* | *0.021* | *0.019* | *0.017* | *0.017* | *0.017* | *0.015* | *0.017* | *0.017* |

A mérésekből az látszik, hogy az elemek számának, vagy méretének növelésével, illetve a mutációk arányának növelésével egyre jobb eredményt ér el a Gyorsított Tömb. A gyorsulás legjobb esetben több mint 65-szörös. Ugyanakkor, a mért adatokból az is látszik, hogy az elemek száma sokkal fontosabb mint a művelettípusok aránya, mivel soronként vagy oszloponként lépegetve, ugyanolyan szorzók mellett sokkal drasztikusabb gyorsulás látható. Az elemek mérete szintén meghatározó, nagyobb elemek mellett a Gyorsított tömb előnyei jobban kihangsúlyozódnak. Ugyanakkor a másik irányból megközelítve, nagyon kis elemméret, és szám esetén, a Gyorsított tömb nagyobb konstans együtthatói válnak dominánssá. Ezek a nagyobb együtthatók a GetRelPos belső függvény osztás és moduló műveletére vezethetők vissza, mivel az egész osztás és moduló rendkívül lassú a modern x64-es processzorokon a szorzás és összeadás műveltekhez képest.

6.3. felhasználhatóság

A Gyorsított Tömb, még további optimalizáció nélkül is felhasználható közvetlenül ott, ahol eddig a Vektorok és Dequek voltak a leggyorsabb alternatívák még a beszúrási időkkel együtt is feltéve, hogy ezek nem használták fel a dequek és vektorok gyors végponti mutációjának lehetőségét( sajnos maga a Gyorsított tömb is pont ilyen, így nem érdemes önmagába ágyazni). Ilyen feladatkör például egy olyan adattároló, amelynél az elemek száma nem túl magas, és a mutációk aránya, az elérésekhez képest elenyésző, de a nagy elemméretek miatt, mégis gyorsulást jelent a használata.

Közvetve felhasználható B fák belső tárolóiként, különösen, a levelekben, ahol a mutációk aránya a lehető legmagasabb az elérésekhez képest. Adatbázisokban, felhasználható, az adatok nyers tárolásánál, ahol ténylegesen tárolva vannak, valamilyen sorrend könnyebb betartásával, így javítva az adatbázisok sebességét, és helyigényét, mivel egy sorba rendezett mező indexrendje implicit tárolható. Ezen kívül potenciálisan felhasználható közvetve unrolled listákban, hash táblákban, a komplexebb adatstruktúrák implementálásához. A Gyorsított tömb nem tud sebességelőnyt jelenteni ott, ahol nem történik mutáció.

Ezeken túl, algoritmusokat is képes gyorsítani, ha azok eddig a Vektorba, Dequebe szúrás által voltak lassítva. Ilyen a beszúrásos rendezés, amely n darab átlagosan n/2 méretű beszúrást hajt végre, így O(n²) es futási idővel bír ezeket az adatstruktúrákat használva. Ez javítható O(n^gyök n)-re Gyorsított tömb használatával.

7. Összefoglalás és További lehetőségek

7.1. megjegyzések

* A modern processzorok prefetch és cache képességeik miatt, az 1-szeres in-direkció nem okoz számottevő lassulást.
* Mivel a belső dequek végére és elejére való beszúrás, vagy törlés, konstans idejű, viszont a gyorsított tömbnél ez az esetek felében nem áll fenn, így az önmagába ágyazás, natív módszerekkel nem jár gyorsulással.

7.2. Ötletek:

* A natív implementáción túl, érdemes lehet a felső vektor méretét annak a 2 hatványnak megválasztani, amely a méret gyökét alulról vagy felülről becsüli. Ez azért lenne előnyös, mert az osztás és moduló művelet helyettesíthetők bitstift, és bitmaszk műveletekkel, ezzel jelentősen csökkentve az index elérés konstans együtthatóját.
* Érdemes lehet az új, növekvő vektort középre helyezni a felső vektorban, ezzel megfelezni a várható balanszolási idő felét, az esetek felében. -A beillesztési idő fele az alsó vektorba illesztés, másik fele a balanszolás-Azz esetek felében kapja az új vektor az elemeket, Ha az egyik vége felé balanszolunk akkor az átlagos távolság gyökN/2, ha a közepe felé,akkor gyökN/4.
* körkörös queue-stack az alsó tömbökhöz, amellyel elérhető, hogy egy memóriaeléréssel végrehajtható a push és pop művelet.
* GetRelPos nagyban gyorsítható polimorfizmussal, vagy függvény printerekkel, amiket méretváltoztatáskor változtatunk, egyébként csak meghívunk, ezzel konstans elérési idő együtthatóját tovább lehet csökkenteni.

Irodalomjegyzék

|  |  |
| --- | --- |
| [1] | T. De Smedt and W. Daelemans, “Pattern for python.,” *The Journal of Machine Learning Research,* vol. 13, no. 1, pp. 2063-2067, 2012. |
| [2] | „Záróvizsga információk,” [Online]. Available: https://mik.uni-pannon.hu/index.php/hu/oktatas/zarovizsga.html. [Hozzáférés dátuma: 04 03 2022]. |
| [3] | D. J. Wetherall és A. S. Tanenbaum, Computer networks, Pearson Education, 2013. |

Mellékletek

Mappaszerkezet

+chatbot

| backen.bat

| backend.py

| files.doc

| fixedlinks.json

| ipcheck.py

| linkek.json

| linkfix.py

| log.txt

| sqlwriter.py

| textprocessor.py

|

+---backend

| pyvenv.cfg

+---lara

| | entities.py

| | nlp.py

| | parser.py

| | stemmer.py

+---static

| +---css

| | chat.css

| |

| \---js

| chat.js

|

+---templates

| index.html

|

[PÉLDA!!! Megjegyzés: A Python csomagkezelője által telepített fájlok, illetve a különböző cache fájlok a fenti listából kimaradtak, mivel ezekkel indokolatlanul és aránytalanul hosszú lenne a fenti felsorolás. A beadott fájlok között azonban a teljesség kedvéért szerepelnek ezek a fájlok is.]

Ábrajegyzék

[1. ábra: ábrafelirat (ha szükséges, akkor a forrás megjelölésével) [1] 1](#_Toc97890941)

Táblázatjegyzék

[1. táblázat (forrás megjelölésével) [2] 1](#_Toc97890942)