**Tartalomjegyzék**

[1 Jelenlegi állapotok 3](#__RefHeading___Toc2985_1132801084)

[1.1 Statikus és Dinamikus Tömbök 3](#__RefHeading___Toc2987_1132801084)

[1.2 Hagyományos és Unrolled Listák 5](#__RefHeading___Toc2989_1132801084)

[1.1 Piros-Fekete-Fa 5](#__RefHeading___Toc1531_2065436672)

[1.3 B fák 7](#__RefHeading___Toc2991_1132801084)

[1.4 Hasító táblák 9](#__RefHeading___Toc2993_1132801084)

[1.4 HAT 10](#__RefHeading___Toc2995_1132801084)

[1.5 összehasonlítás(Tábla elemzéssel) 11](#__RefHeading___Toc2997_1132801084)

[1 Gyorsított Tömb 13](#__RefHeading___Toc2999_1132801084)

[1.1 Alapötlet és Szerkezet 13](#__RefHeading___Toc3001_1132801084)

[1.2 Leírás és elemzés 14](#__RefHeading___Toc3003_1132801084)

[1.2.1 Létreozás 16](#__RefHeading___Toc3005_1132801084)

[1.2.2 . Megsemmisítés 16](#__RefHeading___Toc3007_1132801084)

[1.2.3 Segédfüggvények 17](#__RefHeading___Toc2881_2418417342)

[1.2.4 Mutáció 22](#__RefHeading___Toc2883_2418417342)

[1.2.5 Elérés 23](#__RefHeading___Toc2885_2418417342)

[2 Implementáció, mérések 24](#__RefHeading___Toc2887_2418417342)

[1.1 Implementáció 24](#__RefHeading___Toc1298_1312389254)

[1.2 Verifikáció 25](#__RefHeading___Toc1300_1312389254)

[1.3 Mérés 26](#__RefHeading___Toc2889_2418417342)

[1.4 elemzés, felhasználhatóság 28](#__RefHeading___Toc2893_2418417342)

[2 Összefoglalás és További lehetőségek 29](#__RefHeading___Toc2895_2418417342)

Bevezetés

A modern számítógépes rendszerek meghatározó része a feldolgozott és feldolgozandó adatok tárolása. Az adatok tárolása adatstruktúrákban történik, melyek mind elméleti, mind megvalósításbeli tulajdonságai alapjaiban 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 mutáció 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 modernebb változatainak egy gyorsabb mutációs 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.

# Jelenlegi állapotok

Jelenleg rengeteg Adatstruktúra ismert, melyek különböző előnyöket és hátrányokat hordoznak. A megválasztásukhoz, bármilyen feladatra, elengedhetetlen a megfelelő ismeretük.

Ahhoz, hogy a különböző sebességbeli előnyökről beszélhessek, elengedhetetlen az alapvető fogalmak és definíciók bevezetése.

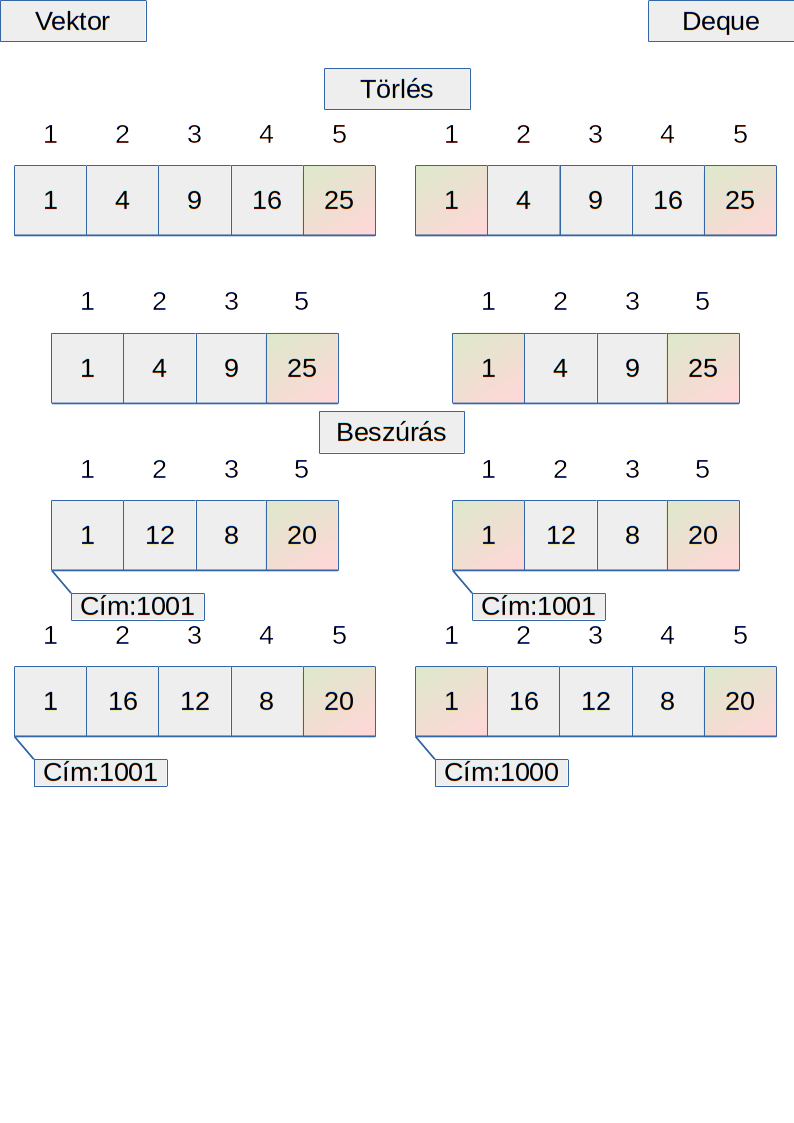
ORDO és a többi definíciója

Mivel a sebességeket az implementáció és a fizikai architektúra nagyban befolyásolja,a naiv implementációk mérése önmagában nem adna teljes képet. Továbbá az is fontos tényező, hogy mik az elméleti korlátai egy algoritmusnak vagy adatstruktúrának. Hogy ezekről matematikailag letisztult módon beszélhessünk, érdemes azt vizsgálni hogyan viselkedik egy adatstruktúra vagy algoritmus, ha a bemenet, vagy tárolt elemek száma minden határon túlnő, azaz a végtelenhez tart.Ez önmagában csak két lehetőséget hordozna,vagy a végtelenbe tart, vagy valamilyen konstanshoz. Hogy a végtelenbe tartó sebességeket össze tudjuk hasonlítani, be kell vezetnünk egy olyan matematikai jelölést módszert amely képes a végtelenbe tartó függvények között valamilyen relációt, hierarchiát felállítani.

## Statikus és Dinamikus Tömbök

Jelenleg a szekvenciális tárolók között a Statikus Tömb, és annak dinamikus változatai (Vektor és Deque) egyeduralkodóknak számítanak. A Statikus Tömb egy rendívül egyszerű adatszerkezet, mely egymás után tárolja az elemeit, melyek mérete egyforma kell, hogy legyen. Az adott indexű elemek memóriacíme megkapható úgy, hogy a Statikus Tömb kezdetéhez (első elem kezdete), hozzáadjuk egy elem méretét, szorozva a keresett indexxel. Ehhez, 0 alapú címzés kell, ami azt jelenti, hogy az első elem indexe 0.

A Dinamikus Tömbök ezt úgy egészítik ki, hogy lehetővé teszik a beszúrás és törlés műveleteket középre.A dinamikus tömbök egyszerűen egymás utáni helyeken tárolják az adatokat több helyet lefoglalva, mint ami szükséges és középen történő mutáció esetén elmozdítják az összes adattagot.Vektor esetén az egyik, általában a nagyobbik, Deque esetén a közelebbi vége felől/felé. A konstans idejű index alapú elérés rendkívül nagy előny. Ez annak ellenére is gyakran megéri, hogy ezeknél a mutáció igen lassú.

1. Ábra Vektor és Deque alapszerkezet

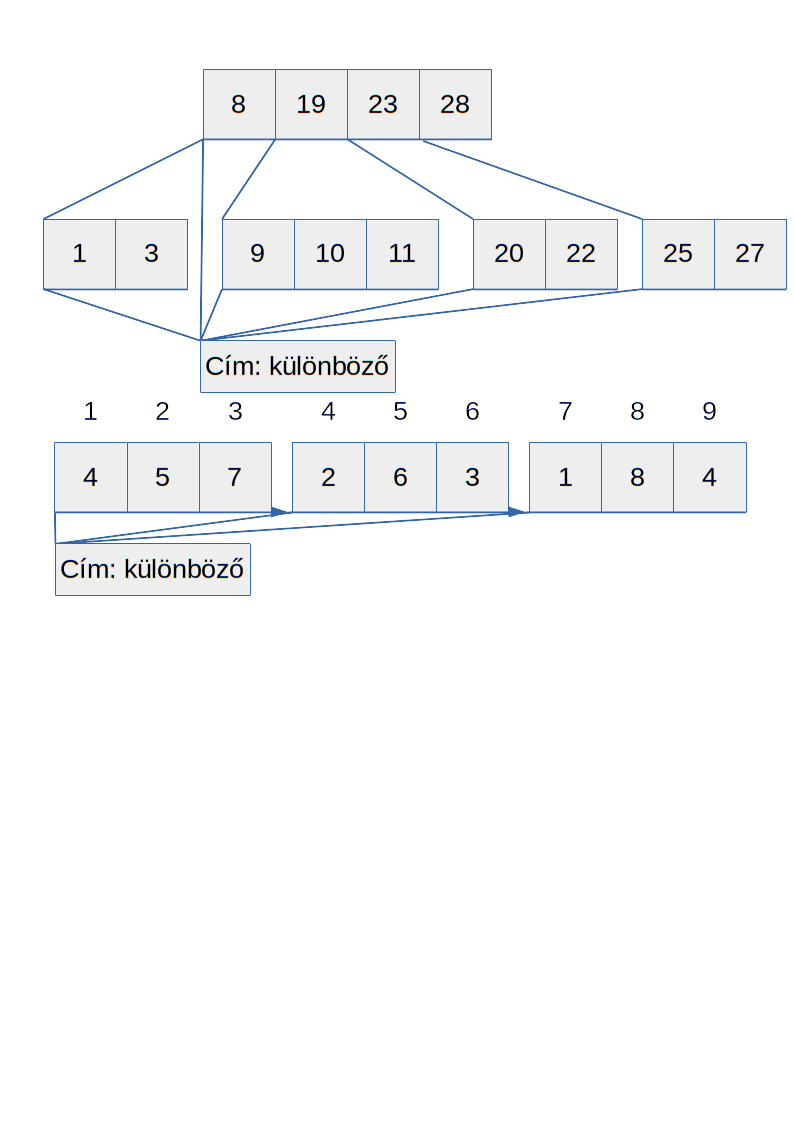
Az index alapú elérés legrosszabb esetben is O(1), míg a mutáció legrosszabb és átlagos esetben Ordó=Theta(n). Ugyanakkor fontos tényező, hogy a Vektor egyik és a Deque mindkét végén konstans időben lehetséges beszúrni és törölni, ha a memória újrafoglalástól eltekintünk.

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,. A szükséges munka( másolás) n, így a várható munka: (1/n)\*n, ami Theta(1). Ugyanakkor az is egy lehetőség, hogy egy modern rendszerben a közel betelt dequeket egy háttérszál újrafoglalja még az előtt, hogy az átméretezés lelassíthatná a programot.

A Dinamikus Tömböket a közvetlen felhasználáson túl, a komplexebb adatstruktúrák felépítésére is használják. 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.

## Hagyományos és Unrolled Listák

A Láncolt listák olyan index alapú adatszerkezetek, amelyek lácszemekből épülnek fel. Ezek a láncszemek egy adattagból, és legalább egy következő elemre mutató pointerből állnak. A láncszemek ezen kívül tartalmazhatnak egy előző elemre mutató pointert is. Ebben az esetben kétszeresen láncolt listáról beszélünk. Ha csak előre mutató pointerek vannak, egyszeresen láncolt listáról. A lista végét egy nullpointer jelzi, amely az érvénytelen 0 címre mutat. A láncolt listáknál adott index eléréséhez, be kell járni a listát, az adott indexig. Ez O(n) idejű művelet. Ha az adott elemet ami után be szeretnénk szúrni, már elértük egy másik művelet részeként, akkor a beszúrás konstans idejű. Ehhez lefoglalunk egy új láncszemet a tárolni kívánt adattaggal, A törlés szintén konstans idejű amennyiben megjegyeztük az előző elötti elemet, vagy 2-szeresen láncolt listával dolgozunk. A láncolt listáknál felmerülhet, hogy egynél több elemet tárolunk egy láncszemben. Ekkor unrolled listáról beszélünk. Ehhez általában Vektort vagy Dequet használnak.

2. Ábra Vektor és Deque közvetett felhasználása, unrolled listához

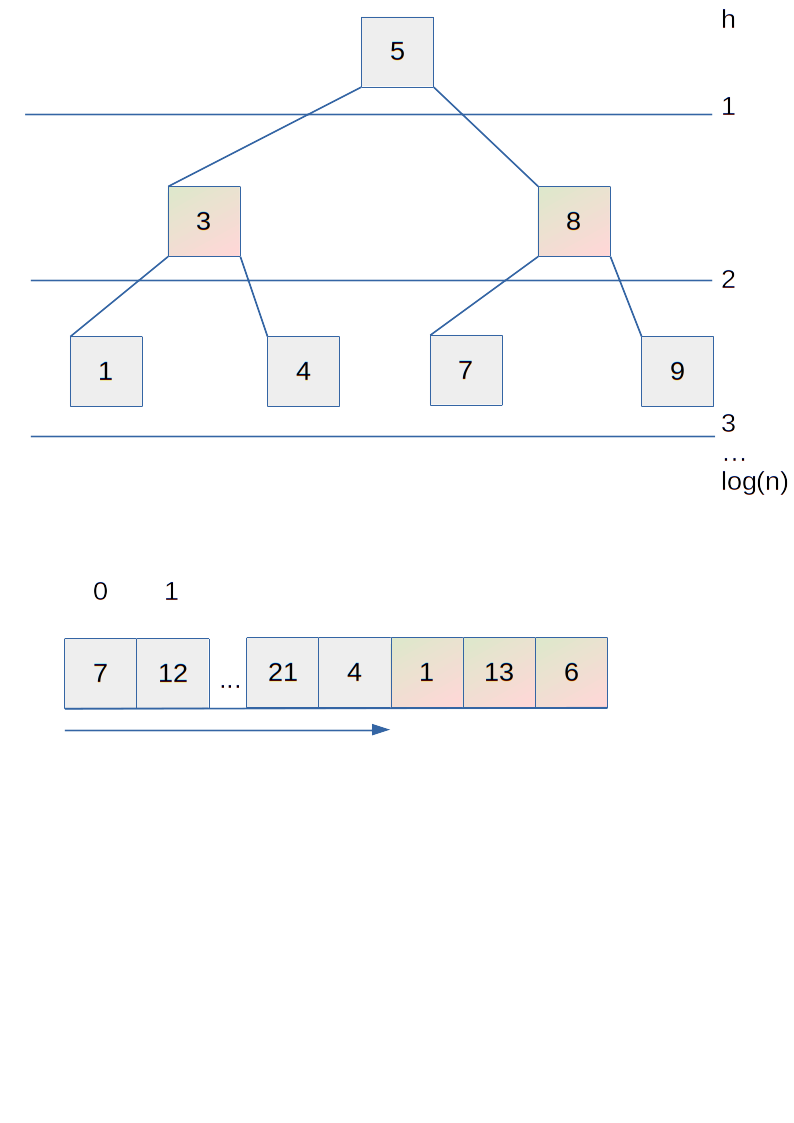
A láncolt listák egy fontos felhasználási módja, hogy fákat lehet velük implementálni. Ez úgy történik, hogy az adattag maga is lehet egy láncolt lista. A lánolt listákat általában közvetlenül, olyan feladatok ellátására szánják, ahol a sorrendi, vagy esetleg fordított bejárás valószínű, és gyakori a beszúrás és törlés.

## **Piros-Fekete-**Fa

A fa adatstruktúrák olyan adatstruktúrák amelyek rekurzív módon elágazva tárolják az elemeiket. A legelső elemet a fa gyökerének nevezzük, magukat az elemeket pedig csúcsoknak. Általában olyan láncolt listák által kerülnek implementálásra, ahol a láncszemek adattagja maga is lehet egy láncolt lista. Ennek megfelelően egy adott csúcsból legalább 2 irányba mehetünk tovább, tehát n lépés után legalább 2^n csúcsba juthatunk el. Ez exponenciálisan jobb, mint az elágazásmentes láncolt listák n lineáris ideje. Azokat a csúcsokat ahová adott x csúcsból juthatunk, x csúcs gyermekeinek nevezzük, a gyermekek számát a csúcs fokszámának. Ha minden csúcsnak ugyanannyi gyermeke van, ez a fa fokszáma is. A 2 fokszámú fákat bináris fáknak nevezzük.

Ez a tulajdonság azonban, csak akkor áll fenn, ha a fa kellően kiegyensúlyozott. Egy adott fa magasságán, azt értjük, hány lépésre van a legtávolabbi csúcs a gyökértől. Levélnek azt a csúcsot nevezzük, amelynek nincsenek gyermekei, hanem adatokat tárol. Egy fát akkor nevezünk kiegyensúlyozottnak, ha minden levél vagy ugyanolyan messze van a gyökértől mint a legtávolabbi levél, vagy legfeljebb 1-el közelebb. Egy fa triviálisan kiegyensúlyozott, amikor üresen létrehozásra kerül. Ezután mutáció során a kiegyensúlyozottság megtartására több módszer ismert. Egy n elemű kiegyenbsúlyozott fa magassága lg(n), ahol a logaritmus alapja, az egy csúcs gyermekeinek száma.

A keresőfák olyan fák amelyek valamilyen egyértelműen sorba rendezhető elemeket tárolnak. Az elemek sorba vannak rendezve úgy, hogy 2 fokszám esetén egy csúcs bal gyermeke mindig kisebb, a jobb nagyobb, vagy egyenlő. Egy kiegyensúlyozott keresőfában, egy adott keresett érték megkeresése O(lg(n)) idejű, mivel lg(n) a maximális távolság, a gyökértől. A magasabb fokszámú keresőfák az 1.3. fejezet alatt, a B-fák által kerülnek bemutatásra.

Ábra

A Piros-Fekete Fa olyan kiegyensúlyozott bináris keresőfa, amely a kiegyensúlyozottságot logaritmikus idejü többlet munkával, és csúcsonként 1 bit többlet helyhasználattal valósítja meg, amely azt jelöli, hogy az adott csúcs piros-e vagy fekete.

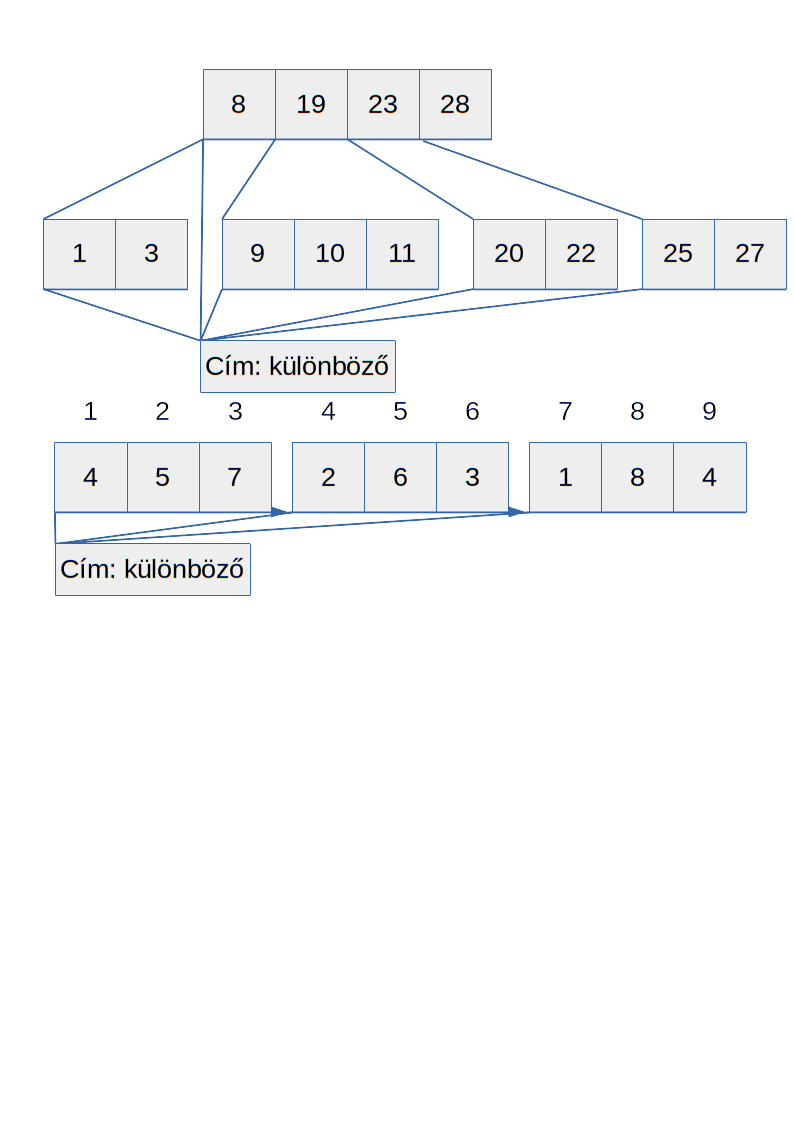
Alap szerkezet( black box cél és tulajdonságok, mi a megoldandó probléma és megoldás, implementáció szerkezet, ebből tulajdonságok levezetése,disclaimerek)

## B fák

A B fák a leggyakoribb összetett adatszerkezetek, amiket olyan adatgyüjtemények megvalósítására használunk, ahol gyakran történik beszúrás és törlés. A B fák olyan kiegyensúlyozott keresőfák, amelyek képesek csúcsonként több mint 2 gyermek tárolására.

Alap szerkezet( black box cél/probléma és tulajdonságok, implementáció szerkezet, ebből tulajdonságok levezetése,disclaimerek)

A csúcsok tartalmazzák a szülő azonosítóját, a gyermekek azonosítóját, és a gyermekek közötti elválasztó értékeket, úgynevezett kulcsokat. A gyermekek sorba vannak rendezve, és a kulcs értékek közöttük helyezkednek el. A kulcs értékek felülről korlátozzák a tőlük bajra, és alulról a tőlük jobbra eső gyermek értékeit. A B fák legalsó szintjén levelek vannak, amelyek adatokat tárolnak (B+ fa esetén kizárólag itt történik adattárolás), legfelső szintjén a fa gyökere van. Lényegében minden müvelet innen indul, a bináris és egyéb keresőfákhoz hasonlóan, viszont az elemek számának növelésével, laposabbá tehető, az O(lg(n)) idejű műveleteknél, a logaritmus alapja egy nagyobb konstans. A B fák emellett, azért is gyorsabbak a gyakorlatban, a bináris keresőfáknál, mivel egymáshoz közel tárolnak adatokat, így egy adott memóriaelérésnél, amelyeknek a mérete általában 64 bájt, olyan adatokat is betölt a processzor, amelyeket nem kért eredetileg, de nagy eséllyel szüksége lesz rá később. A B fák csúcsaiban a fokszám nem konstans, hanem egy adott tartományban változik.

3. Ábra B fa

State of the art

Altpusok

Tömbhasználat kihangsúlyozása

## 1.4 Hasító táblák

A hasító tábla egy olyan kereső tároló amely tömbben szétszórtan tárolja az elemeit, melyek általában kulcs érték párok. Az alapja az, hogy az összes lehetséges kulcs közül csak nagyon kis arányban fogunk kulcsokat felhasználni. Emiatt megengedhetőnek tekintjük, hogy több kulcs ugyanoda kerüljön. A kulcsok szétosztásához a tároló tömbben, egy olyan függvényre van szükség, amely:

* konstans időben fut( feltételezve, hogy a kulcsok mérete korlátos)
* determinisztikus, tehát ugyanarra a bemenetre ugyanazt a kimenetet adja
* a tároló tömb minden címét felhasználja, közel egyenletesen
* hasonló bemenetekre nem ad azonos kimenetet.

Az ezeket teljesítő függvényt, hash függvénynek nevezzük, melyre számtalan lehetséges jelölt van. A Hasító Tábla használata során a tárolandó, törlendő, vagy elérendő kulcsnak kiszámítjuk a hash függvénnyel vett értékét. Majd az ez által indexként mutatott helyen elvégezzük a kívánt műveletet. Ez az ideális és egyben a legjobb eset. Sajnos mivel több kulcsot kell, hogy rendeljünk egy indexhez, ezért elkerülhetlen, hogy küönböző kulcsok időnként ne használják ugyanazt a helyet. Ezt ütközésnek nevezzük. Ezeknek a feloldására több megoldás is létezik, például egy láncolt lista használata az elemek tárolására, az adott indexre helyezve, közvetlen tárolás helyett. Ekkor belátható, hogy a fent említett műveletek(beszúrás, törlés, keresés) konstans amortizált időben futnak, ha kétirányban láncolt listát használunk, és a tömb mérete legfeljebb konstans arányú.[1]link

A Hasító Tábla általában Statikus Tömböket használ, ezért a Gyorsabb mutációból nem származna gyorsulás általános esetben.

## **HAT**

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

A szerkezete a Késöbbiekben bemutatásra kerülő Gyorsított Tömbhöz hasonló. Az előnye a hagyományos tömbökhöz képest az, hogy a többlet memóriahasználata gyökN-es míg ugyanez a hagyományos Dinamikus Tömböknél lineáris. Egy Dictionariből áll amely körülbelül gyökN méretű, és Dinamikus tömbök címeit tárolja. Ezek a dinamikus tömbök tárolják magukat az adatokat. A tömbökön belűli címeket egy i indexre úgy kapjuk meg, hogy

Alap szerkezet( black box cél és tulajdonságok, mi a megoldandó probléma és megoldás, implementáció szerkezet, ebből tulajdonságok levezetése);pseudo, értelmezéssel;elemzés;step by step

Bemutatás alapműveleteken keresztül

## 1.5 összehasonlítás(Tábla elemzéssel)

szöveg

Táblaleírás

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Létrehozás | Törlés | Mutáció | Elérés | Keresés |
| Dinamikus Tömb (Vektor, Deque, HAT) | **Theta(1)** | **Theta(1)** | **Theta(n)\*** | **Theta(1)** | **Theta(n)** |
| Láncolt Lista | **Theta(1)** | **Theta(n)** | **Theta(n)\*** | **Theta(n)\*** | **Theta(n)** |
| Fa | **Theta(1)** | **Theta(lg n)** | **Theta(lg n)** | **-** | **Theta(lg n)** |
| Hasítótábla | **Theta(1)** | **Theta(1)** | **Theta(1)\*** | **-** | **Theta(1)\*** |
| *Gyorsított tömb* | ***Theta(1)*** | ***Theta(sqrt n)*** | ***Theta(sqrt n)*** | ***Theta(1)*** | ***Theta(n)*** |

Az aszimptotikus sebességek a végtelenben vett elméleti sebességet írják le. Éppen ezért, a másik véglet, vagyis a nagyon kis elemszám teljesen más tulajdonságokat mutathat, és az aszimtotikus tulajdonságok csak az elemszám növekedésével válnak jelentőssé. Kis elemszámnál, a konstans időigényű részmeletek válhatnak hangsúlyossá, míg a végtelenben vett határértékeknél ezek teljesen eltünnek. Nagyon kis méretnél általában a dinamikus tömbök a leggyorsabbak minden eladatra.

Fontos megjegyezni, hogy a Láncolt Lista csak akkor igényel lineáris időt, ha az adott indexhez el is kell jutni. Amennyiben egy másik művelet során a listát már egyébként is bejártuk, a Beszúrás és az elérés konstans id igényelnek. Ezen felül, ha 2 irányba láéncolt a lista vagy megjegyeztük az előző indexű elemet, a törlés is konstans idejű. Ezen kívülk fontos tényező, hogy ezek a futási idők átlagos futási idők, viszont a hasítótáblánál a legrosszabb eset Mutációra és keresésre, O(n).

Alapjában véve a különnböző adatstruktúrák különböző problémák megoldására lettek kitalálva. Ennek megfelelően minden összehasonlításnál figyelembe kell venni, hogy milyen probléma megoldására kerresük a megfelelőt. Ezen kívül, alkalmanként, a fizikai eszköz tulajdonságai is beleszólhatnak abba, hogy melyik a legjobb megoldás.

gráfok

# Gyorsított Tömb

A Gyorsított Tömb egy adatstruktúra, amely a hagyományos Tömbhöz hasonlóan indexelve tárolja az elemeit. A Vektor és a Deque a hagyományos Tömb továbbfejlesztett változatai, melyek megvalósítják a beszúrás és törlés műveleteket, lineáris időben.

A Gyorsítot Tömb célja, hogy a konstans index elérési időt megtartva egy gyorsabb beszúrási és törlési időt tegyen lehetővé. Ezt úgy éri el, hogy nem egy tömbben tárolja az elemeket folytonosan, hanem széttördelve több kisebb tömb között, amelyeknél a közel azonos hossz biztosított.

## Alapötlet és Szerkezet

A Gyorsított Tömb 3 alkotóeleme:

* Metaadatok

A Metaadatok tárolják a főbb adatokat, mint a jelenlegi populáció, amelynek ismerete elengedhetetlen a működéshez. Emellett minden olyan további tulajdonságot, amelyet szeretnénk ideiglenes változókban megtartani a sebesség vagy olvashatóság érdekében.

* Felső Tömb

A Felső Tömb tárolja az Alsó Tömbök címeit.

* Alsó Tömbök

Az Alsó Tömbök tárolják az adatokat, Ezek Dequek kell hogy legyenek, vagy egy másik olyan lineáris index alaú adatstruktúra, amely constans időben valósítja meg a végeken a beszúrás és törlés műveleteket.

A gyorsított tömb, az adattagok 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.

Az adatok sorrendje, elsősorban a tároló tömb helye sorrend szerint, majd a tárolón belüli pozíció. Az alsó tömbök, cím szerint a Felső Tömbben kerülnek tárolásra. Mivel a Gyorsított Tömb logikailag index alapján éri el az elemeit, beszélhetünk logikai indexről, amely az adott elem pozíciója a tárolt elemek sorában. Ezen kívül beszélhetünk fizikai címről amely alatt itt, egy egész számpárt értünk amely megadja először, hogy melyik tömbben van egy adott elem, majd azt is, hogy azon belül hányadik. Minden logikai címhez tartozik egy fizikai index, és fordítva.

Négyzet alakúnak azt a Gyorsított Tömböt nevezzük, amelyben minden Alsó Tömb mérete megyezik az alsó tömbök számával.

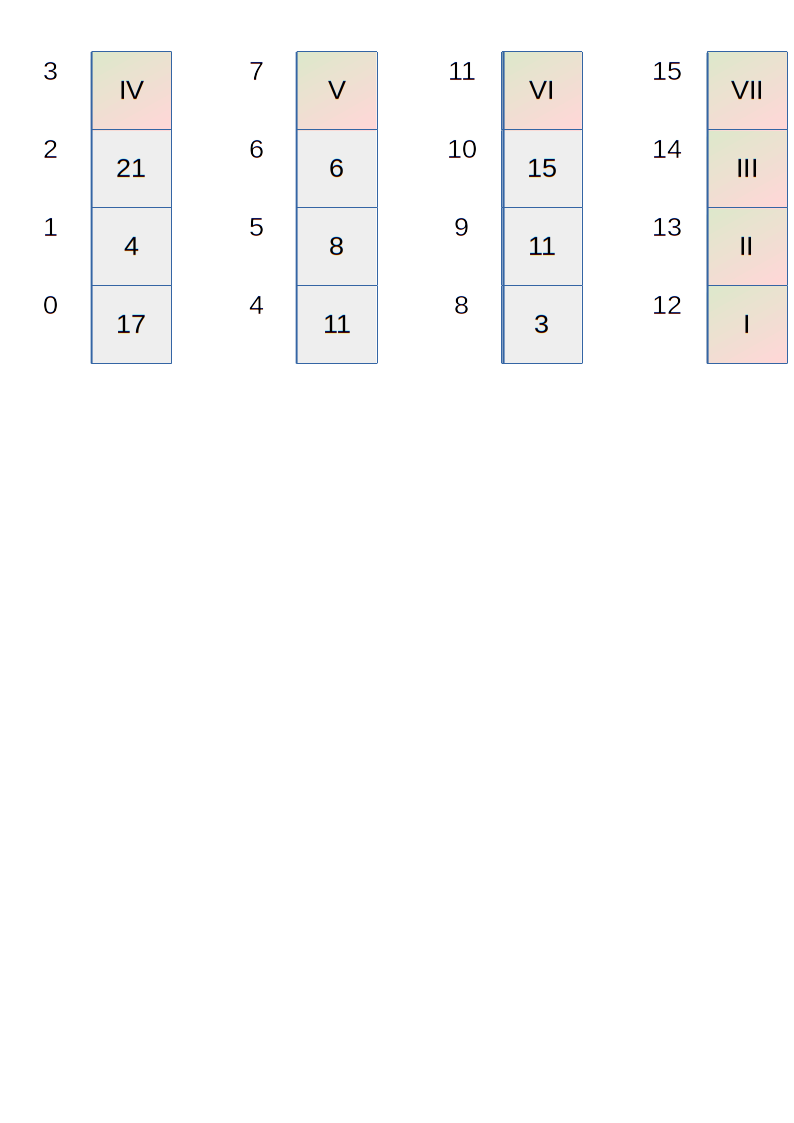
4. Ábra

## Leírás és elemzés

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

Ebből következik, hogy egy Alsó Tömb végéről levett, és az azt követő Alsó Tömb elejére helyezett elem indexe nem változik. Ugyanez az ellenkező irányban is működik.

Ha egy Alsó Tömb elejére helyezünk egy új elemet, és utána a végéről levesszük az utolsót, a hossza nem változik.

Ezt a két tényt kihasználva, az Alsó Tömbök elemeinek eloszlása megváltoztatható, a logikai sorrend megváltoztatása nélkül. Ez azért lehetséges, mert egy tetszőleges Alsó Tömbbe beszúrás esetén a többlet elvihető a kívánt Alsó Tömbbe anélkül, hogy a logikai sorrend megváltozna.Mindez törlés során szintén működik.

Ahhoz hogy az Alsó Tömbök közel gyök N méretűek maradjanak, szükséges, ezeket kiegyensúlyozni. A kiegyensúlyozásnál, a többletet tartalmazó Alsó Tömb irányából a következő növekedés helye felé történik.

5. ábra

A Gyorsított Tömb a létrehozáskor üres.

Ebben az állapotban a beszúrás triviálisan egy új alsó Tömbbe történik. Az új alsó Tömb címe a Felső Tömbbe kerül.

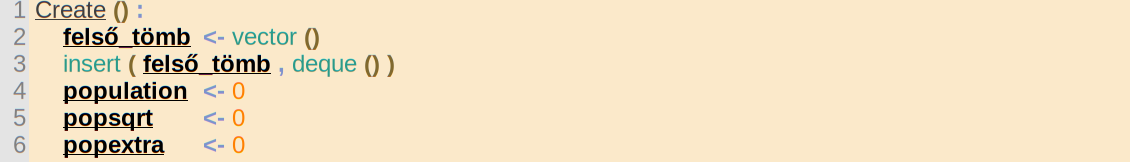
Mivel az 1 négyzetszám, innentől a Gyorsított Tömb bővítése innentől leírható egy négyzet alakú magra történő építkezésként. A 4. Ábra egy 9 és egy 16 elemű gyorsított Tömb közötti átmenetet mutat. A négyzet alakú mag bővítése során, először egy új Alsó Tömb beszúrása történik, sorban a többi után, amely az 5. Ábrán a 4. oszlopként van jelölve. Ez feltöltésre kerül ugyanaddig mint a meglévő alsó tömbök. Ez sorban a Római 1,2,3 elemek.

Ezt követően minden tömb mérete 1-el kerül növelésre, sorban, ahogy az a Római 4.5.6 számnál látható. Ezáltal egy eggyel nagyobb négyzet alakú struktúrához jutunk.

A Deque műveletekkel a többlet eltávolításra kerül és bekerül a következő altömbbe, iterálva, amíg a kívánt helyre kerül. Mivel a beszúrás és a növekedés Alsó Tömbje közötti tömbök minde egyszer növelve, egyszer pedig csökkentve lettek méretben, így ezek mérete nem változik.

### Létreozás

1. A Felső Tömb létrehozása
2. a legelső Alsó Tömb lefoglalása
3. belső adatokat tároló konstans méretű struktúra létrehozása és adatokkal feltöltése, ahol a populáció tárolása szükséges, de érdemes további adatokat is eltárolni, mint a populáció lefelé kerekített négyzetgyöke, ami a négyzet alakú mag oldalhosszát adja meg, illetve azt, hogy ezen túl, mennyi extra elem van.



Konstans darab memóriafoglalás( legalább 2,az első alsó és a felső tömbnek, de érdemes lehet több Alsó Tömböt előre lefoglalni) pontosan 4 változó értékadással deklarálása, majd az egyik 1 lépésben módosítása. Mivel a Létrehozás konstans számú műveletet végez, melyek külön-külön konstans idejűek, a Létrehozás művelete is konstans idejű.

Omega=Theta=Ordo(1)

### . Megsemmisítés

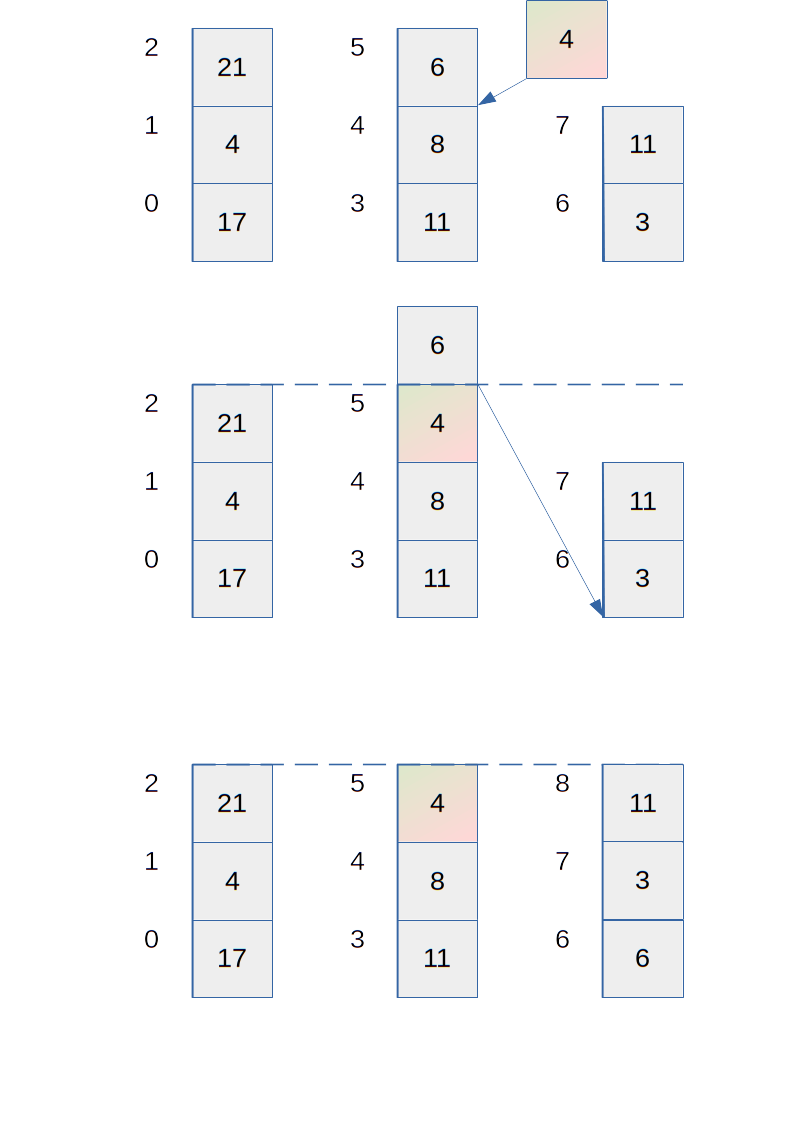
Az összes alsó majd a felső tömb megsemmisítése.

Felső tömb minden elemét (Alsó Tömbök) felszabadítani, majd magát a Felső Tömböt is, gyök N+2 törlés összesen, legrosszabb esetben. Leggyorsabb esetben csak 1 Alsó Tömb van, így ekkor 2 törlésre van szükség.

Omega=1

Theta=Ordo=gyök N

### Segédfüggvények

 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 a többletet átvinni, maga az egyensúlyozás is gyök N-es.

A négyzet oka

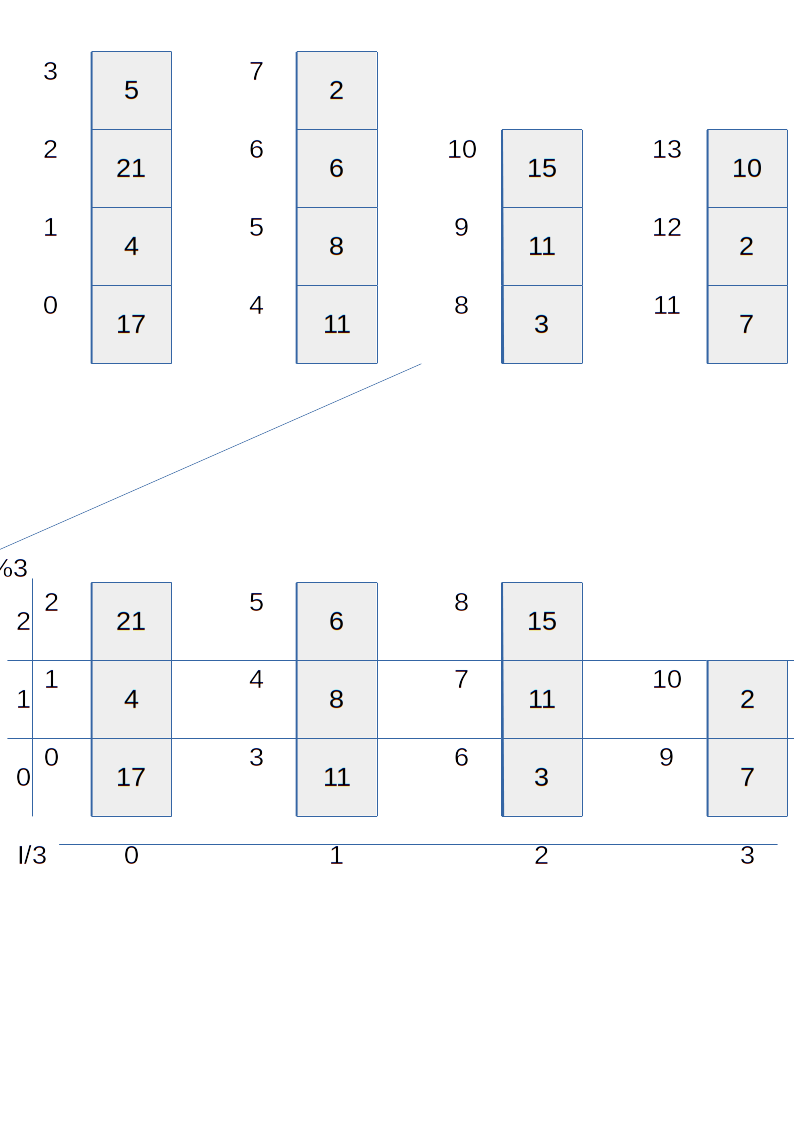
Mivel a kiegyensúlyozásnál az alsó tömbök hosszával arányos a beszúrás/törlés időigényének egyik része és a felső tömb hosszával arányos a másik része, ez akkor minimális ha a kettő megegyezik, mivel azonos területű téglalapok szomszédos oldalainak összege akkor minimális, ha egy négyzetről van szó.

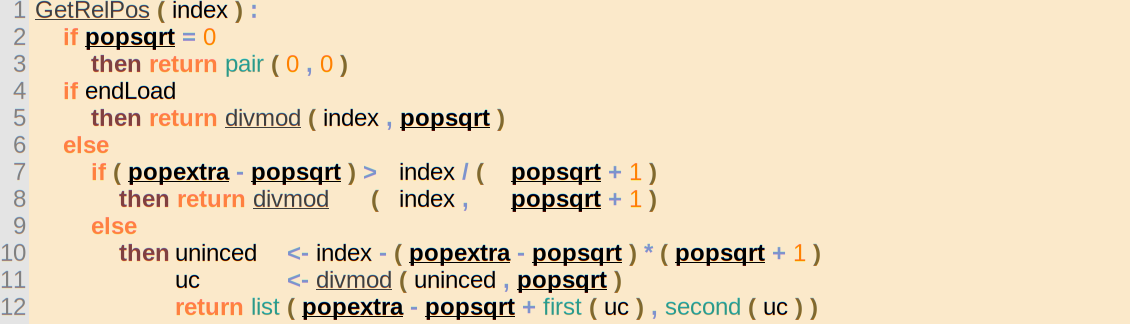
Kiegyensúlyozás

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

GetRelPos: kap egy i indexet, ami az elérendő indexet jelöli, és visszatér, az azt tároló felső tömb indexével, illetve, az azon belüli indexxel, ami az i-edik elemre mutat.

* 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.

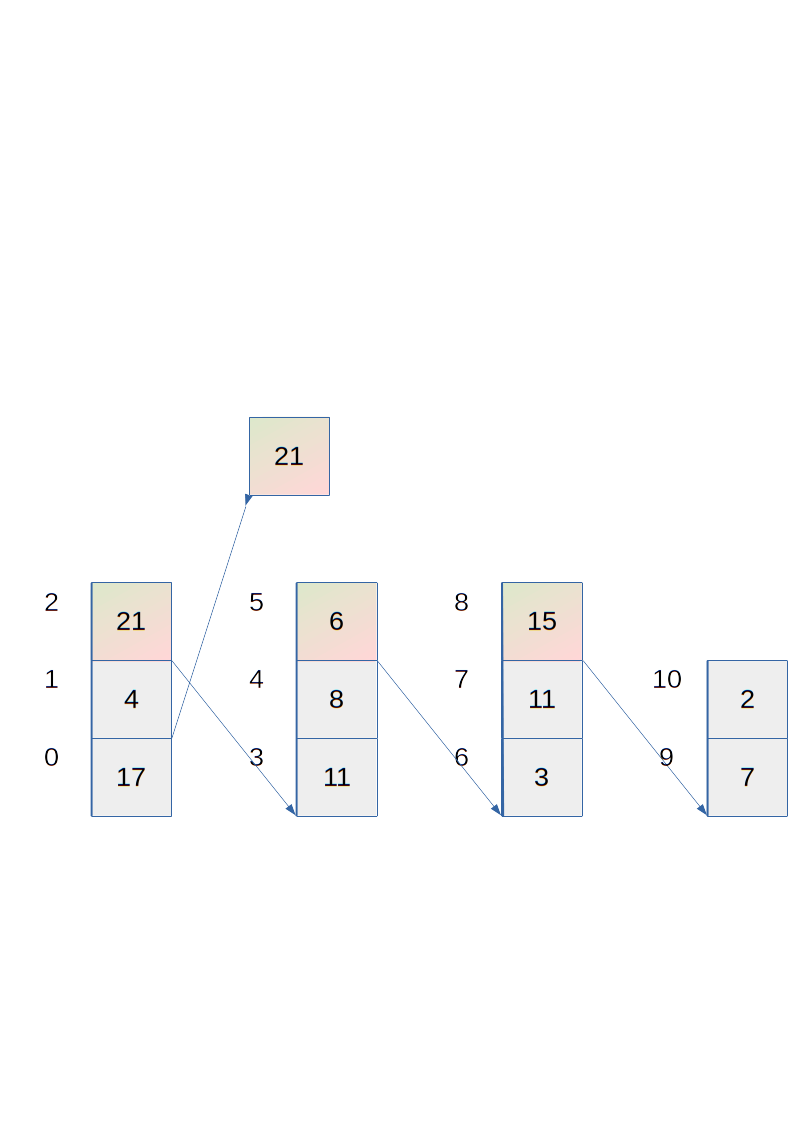
6. Ábra elemek indexe

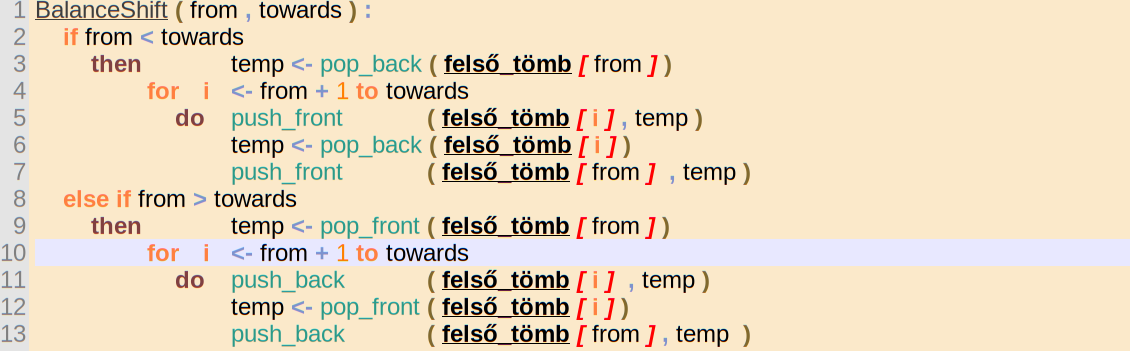


Keresés és egyéb műveletek, amikre nem lett szánva

BalanceShift:

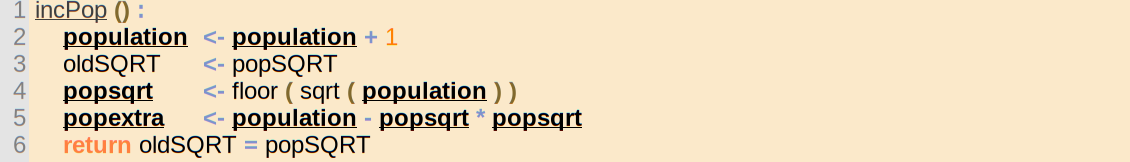
A mutáció helyét megkapja és onnan, beillesztésnél a newPlace, törlésnél a deletePlace felé ellentétes push és pop műveletekkel végighordja a többletet az alsó tömbök között.

7. Ábra eltolás beszúrás után



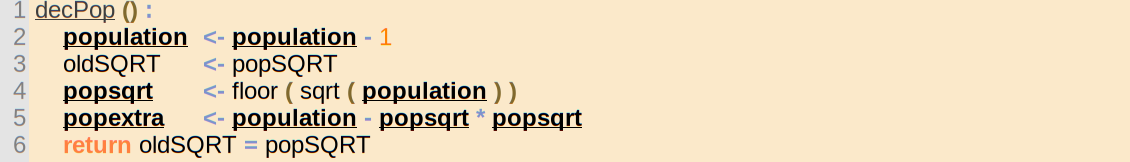
incPop():

Populáció eggyel növelése.

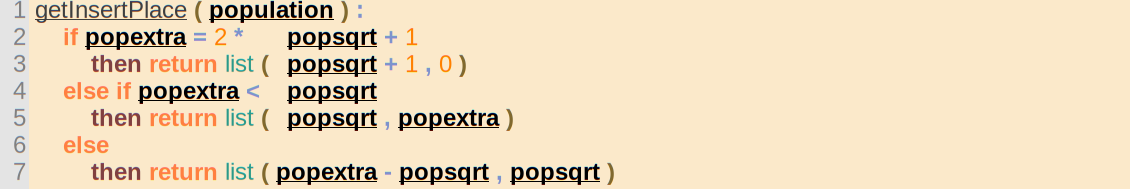


decPop():

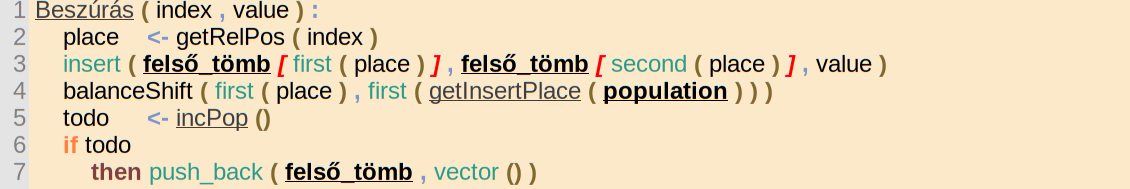
Populáció eggyel csökkentése.

getInsertPlace:

Növekedés helyét adja vissza. Tökéletes négyzet alakú tároló esetén egy új alsó tömböt kezd, és ezt feltölti olyan magasra mint a többi, majd utána minden vektort az elejétől kezdve megnövel egy elemmel.



### Mutáció

A beszúráshoz adott logikai indexre. Először megkeressük a tényleges adatszerkezetbeli fizikai helyet, ami a megfelelő felső tömböt és az azon belüli indexet jelöli. Erre a helyre megtörténik a beszúrás az Alsó Tömbben, majd a Balansceshift művelet eltolja a többlet elemet a növekedési pont felé. Ezek után a gyorsított tömb belső adatai konstans időben újraszámításra kerülnek.

A beszúrás nem tartalmaz önmagában iterációt vagy rekurziót, így a hívott függvények döntik el a sebességét. Kettő meghívott függvény van, ami nem konstans, az Alsó Tömbbe (Dequebe) tetszőleges helyen beillesztés és a BalanceShift. Várhatóan és legrosszabb esetben is a Dequebe beszúrás 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.

A Törlés során, 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.

N, vagyis théta gyök N-es. Legrosszabb esetben, mindkét helyzetben, gyök N mozgatást igényel.

Omega=1

Theta=Ordo=gyök N

### Elérés

4.5 olvasás adott helyen

Adott indexelérése, is visszatérés az ott tárolt értékkel.

4.6  Írás adott helyen

 Adott index elérése, és az ott lévő érték felülírása.

helyben elérés:

omega=theta=ordo=1

incPop():

pontosan 5 szekvenciális lépés minden esetben

omega=theta=ordo=1

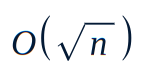
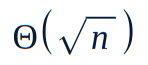
getRelPos(index):

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

omega=theta=ordo=1







# Implementáció, mérések

A gyorsított Tömbhöz készítettem egy adastruktúrát. Ennek több oka volt, Elsősorban szerettem volna megbizonyosodni, hogy tényleges a feladatát végzi, és nincs semmilyen észre nem vett elvi hiba, különös tekintettel, az edge-casekre. Az implementációt ezután felhasználtam a sebesség összehasonlítására, a C++ beépített std::Vectorával szemben. A méréseket ezután elemztem. Fontos kiemelni, hogy az olvashatóság kedvéért, semmilyen komoly optimalizációt nem végeztem, a pseudokódhoz próbáltam a lehető legközelebb maradni.

## Implementáció

Az implementációt C++-ban végeztem el. Ennek több oka volt. Mint az egyik legismertebb programozási nyelv, az ebben megírt kész implementáció rendkívül széles közönséggel bír. Kellően alacsony szintű ahhoz, hogy szinte minden szükséges lépés teret kapjon. Támogatja az objektum orientáltságot és a sablonokat, ami megkönnyíti a tesztelést, és használatot.

A Metaadatoknak, és az Alsó Tömböknek egy-egy saját osztályt definiáltam. A Felső Tömböt viszont, mivel önálló feladatköre nincs, egyszerűen egy Alsó Tömböket tároló tömbként implementáltam. Az implementáció során egy template class-t hoztam létre, amely belső adattagként tárolja a Metaadatokat és a Felső Tömböt. Azért tagoltam fel a teljes projektet így, hogy elemenként tudjam fejleszteni a rendszert. Mivel az elvárásokat már az implementáció megkezdése előtt ismertem, tudtam komponensenként tesztelni, amint elkészültem egy részelemmel. A komponensek összessége után az egész rendszert teszteltem.

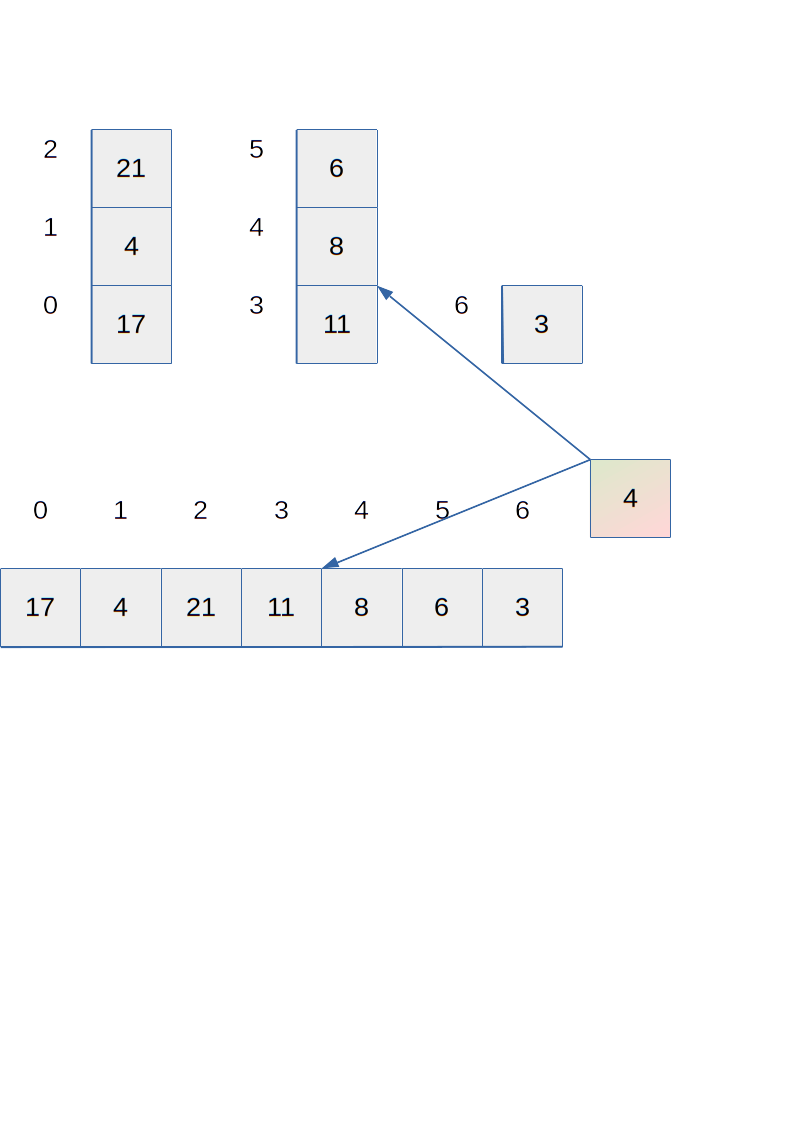
## Verifikáció

A tesztelés elsődleges célja az volt, hogy meggyőződjek róla, hogy az elkészített implementáció ténylegesen a feladatát látja el. Ezt verifikációnak nevezzük. A verifikációra azért is szükség volt, mert enélkül értelmetlen egy program teljesítményét mérni, hiszen nem számít, hogy egy hibás program milyen gyorsan fut. A verifikációnál lényegében arról kellett meggyőződnöm, hogy a Gyorsított Tömb implementációm megfelelően implementálja a Dinmaikus Tömbtől elvárt műveleteket, és viselkedést.

A Dinamikus Tömbnek lehetővé kell tenni, a beszúrás és törlés műveletet, adott indexre, illetve képesnek kell lennie adott érték index alapú lekérésére, is módosítására konstans időben. Ezen felül az adott i indexre történő, x érték beszúrása után, i lekérésére x-el kell, hogy visszatérjen az adatstruktúra. Ezen felül az ennél nagyobb indexü elemek mind egy indexxel eltolásra kell, hogy kerüljenek. Törlésnél, ehhez hasonlóan az i törölt indexnél nagyobb indexű elemek eggyel kisebb indexre kerülnek. Ezen felül adott érték felülírása után i indexen, az i index az új értéket kell hogy tárolja, amíg valami nem indokolja ennek a változását.

Az elemenkénti ellenörzés során, fontos tulajdonság volt több függvénnyel kapcsolatban, hogy előre, formálisan meghatározható volt, adott esetekben, mi a helyes visszatérési érték. Ennek megfelelően, meghívás után könnyen le tudtam ellenőrizni, hogy helyes értékkel tér-e vissza. Miután az elemek külön-külön rendben voltak, a teljes adatstruktúrát kellett tesztelnem. Az összes elem önmagában helyes működése azért nem elegendő, mivel az esetleg elvi hibák csak a teljes rendszer működése során kerülnek felszínre, illetve a részegységek összekapcsolásának hibái, a kapcsolat egyik felében sem jelentkeznek külön vizsgálat esetén.

A Teljes Adatstruktúra ellenörzésének kidolgozása során, előfeltételeztem, hogy a C++ beépített std::vector-a a Dinamikus Tömb tulajdonságoknak megfelelően működik. Erre alapozva, végrehajtottam ugyanazokat a műveleteket, egy std::vector, és a Gyorsított Tömb egy példányára is. A műveleteket brute force alapján generáltam, illetve használtam kézzel előállított utasítássorozatot is, az „edge-casek”ellenörzésére.

Ábra ,”3” érték beszúrása a 3 index utánra

A műveletsorok tartalmaztak beszúrást és törlést, érvényes indexekkel, valamint írási és olvasási műveleteket. Az olvasások eredményei össze lettek hasonlítva, illetve a teszt végén, a két adatstruktúra sorban kiolvasásra került, és elemenként össze lett hasonlítva.

A tesztek több olyan apróbb implementációbeli hibát is felfedtek, amik a komponensenkénti tesztelésnél nem jelentkeztek. Ezek javításával a sikeres tesztek után, az implementációt késznek tekintettem.

## Mérés

A mérések elvégzéséhez a C++ programozási nyelv-ben megírt, verifikációhoz is használt implementációt használtam. Az implementáció a pszeudokódok alapján történt, komoly optimalizálások nélkül, mivel elsősorban a validálás volt a célja. Az Összefoglalásban 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 kisebb várakozások kerültek beiktatásra, hogy a mérések a lehető legkisebb hatással legyenek egymásra, azonban, 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.

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 látható, 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. A mért adatokból megfigyelhető, 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. 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.

## elemzés, 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 tárolásra kerülnek, valamilyen sorrend könnyebb betartásával, így javítva az adatbázisok sebességét és helyigényét, mivel egy sorba rendezett mező indexrendje implicit tárolható. Ezen felül potenciálisan felhasználható közvetve unrolled listákban, B fákban, egyéb komplexebb adatstruktúrák implementálásához. A Gyorsított Tömb nem tud sebességelőnyt jelenteni a Statikus Tömbhöz képest ott, ahol nem történik mutáció.

Ezeken túl, algoritmusokat is képes gyorsítani, amennyiben azok eddig a Vektorba, Dequebe szúrás által kerültek lassításra. 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. A meglkévő algoritmusokon túl felmerülhetnek olyan új algoritmusok, amelyek eddig, a Dinamikus Tömbökben történő mutáció lassú, lineáris ideje miatt voltak csak túl lassúak.

# Összefoglalás és További lehetőségek

szöveg szöveg svöeg

7.0 konkrét összefoglalása az adatstruktúrának;többi(„7.1,7.2”)

7.1. megjegyzések

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

7.2. Ötletek

* A natív implementáción túl, érdemes lehet a felső vektor méretét annak a kettő 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. Sajnos ebben az esetben a mutáció legrosszab esete O(n), de az amortizált eset továbbra is Theta(sqrt n).
* 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. Az 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. Ez tovább lassítaná az elérésnél, a helyes index kiszámítását.
* körkörös queue-stack az alsó tömbökhöz, amellyel elérhető, hogy egy memóriaeléréssel végrehajtható a push és pop művelet. Ez tovább lassítaná az elérésnél, a helyes index kiszámítását.
* 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

TODO Idó Oldal

ORDO és a többi definíciója befejezése, ábrával 1 0.5

B-Fa műveletei, kevés magyarázattal, akár ábrával 3 1

PF-Fa műveletei, kevés magyarázattal, akár ábrával 2 1

HashTábla Ábra 0.5 0.5

HAT műveletei, kevés magyarázattal, akár ábrával 1 1

Összehasonlítás befejezése 0.5 0.5

kis segédfüggvények 1.5 0

getrelpos;shift 2 0

elérési függvények 1 0

Összefoglaló ábrák és továbbiak 1 1

Ábrajegyzék

kisebb dolgok(kétszeresen láncolt lista ábra sttb) 0.5 0.5

kódszerkezet, kódkép 1 0.5

felhasználás elemzése 2 4

elméleti jelentőség 1 1

továbbfejlesztés kifejtése ott ahol tradeoff van 2 1

Táblázatjegyzék