**Tartalomjegyzék**

[1 Jelenlegi állapotok 3](#__RefHeading___Toc1678_3221834476)

[1.1 Statikus és Dinamikus Tömbök 4](#__RefHeading___Toc2987_1132801084)

[1.2 Hagyományos és Unrolled Listák 6](#__RefHeading___Toc1587_446001274)

[1.3 Piros-Fekete-Fa 7](#__RefHeading___Toc1531_2065436672)

[1.4 B fák 9](#__RefHeading___Toc2991_1132801084)

[1.5 Hasító táblák 12](#__RefHeading___Toc2993_1132801084)

[1.6 HAT 13](#__RefHeading___Toc2995_1132801084)

[1.7 Összehasonlítás(Tábla elemzéssel) 14](#__RefHeading___Toc2997_1132801084)

[2 Gyorsított Tömb 16](#__RefHeading___Toc2999_1132801084)

[2.1 Alapötlet és Szerkezet 16](#__RefHeading___Toc3001_1132801084)

[2.2 Leírás és elemzés 17](#__RefHeading___Toc3003_1132801084)

[2.2.1 Létreozás 19](#__RefHeading___Toc3005_1132801084)

[2.2.2 Megsemmisítés 20](#__RefHeading___Toc3007_1132801084)

[2.2.3 GetRelPos 20](#__RefHeading___Toc2881_2418417342)

[2.2.4 BalanceShift: 23](#__RefHeading___Toc1580_4263731678)

[2.2.5 További segédfüggvények: 26](#__RefHeading___Toc1582_4263731678)

[2.2.6 Mutáció 29](#__RefHeading___Toc2883_2418417342)

[2.2.7 Elérés 32](#__RefHeading___Toc2885_2418417342)

[3 Implementáció, mérések 34](#__RefHeading___Toc2887_2418417342)

[3.1 Implementáció 34](#__RefHeading___Toc1298_1312389254)

[3.2 Verifikáció 35](#__RefHeading___Toc1300_1312389254)

[3.3 Mérés 36](#__RefHeading___Toc2889_2418417342)

[3.4 elemzés, felhasználhatóság 38](#__RefHeading___Toc2893_2418417342)

[4 Összefoglalás és További lehetőségek 40](#__RefHeading___Toc2895_2418417342)

[4.1 megjegyzések 40](#__RefHeading___Toc1676_3221834476)

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 ezekben 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.

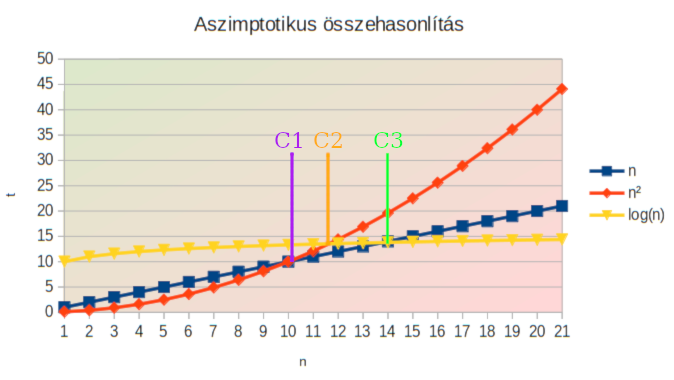
A Gyorsított Tömb, a hagyományos tömbökhöz hasonlóan konstans idejű index alapú elérést biztosít, de ezen túl képes gyök N időben beszúrást és törlést megvalósítani. Mindehhez gyök N-es többlet memóriahasználat tartozik.

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

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. Ahhoz, 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.

1. Ábra: Függvények összemérése

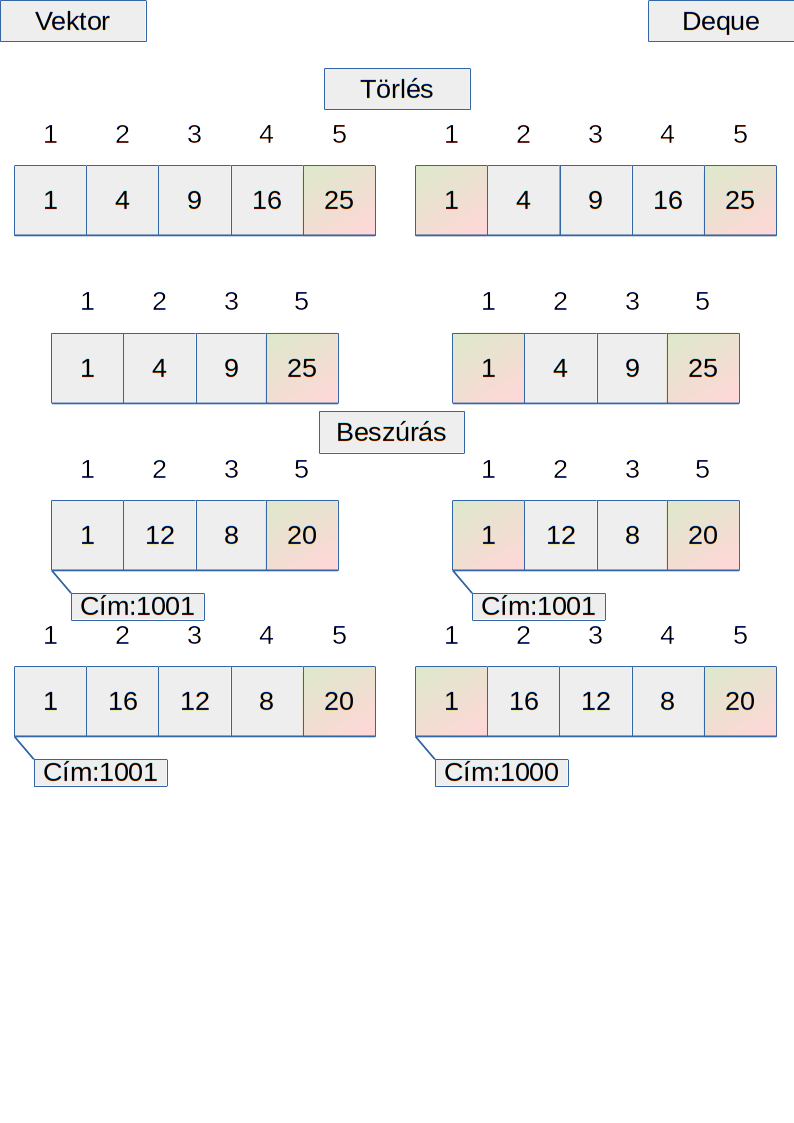
Egy vizsgált f függvényt akkor nevezhetünk nagyobbnak egy másik g függvénynél végtelenben, ha f tetszőleges pozitív c konstans együtthatója mellett is, n változó végtelenbe tartása során létezik n0 küszöbindex amelytől n>n0 esetén c\*f(n)>g(n). Ekkor f felülről becsüli g-t. Ezt úgy jelöljük, hogy g=O(f). Ez a nagy Ordó jelölés. Előfordulhat az is, hogy n változó végtelenbe tartása során létezik n0 küszöbindex amelytől n>n0 esetén c\*f(n)<g(n). Ekkor f alulról becsüli g-t. Ezt g=Omega(f)-el jelöljük. Ha mindkettő fennáll, g=Theta(f).

Ennek megfelelően, az ábrán A kvadratikus függvény felülről becsüli a másik kettőt, a lineárist C1-től, a logaritmikust C2-től. Ennek megfelelően, azok alulról becsülik a kvadratikus függvényt ugyanezektől a küszöbértékektől. A lineáris függvény C3-tól felülről becsüli a logaritmikust. Ennek megfelelően, a logaritmikus függvény C3-tól alulról becsüli a lineárist. Ezen felül minden függvény alulról és felülről is becsüli önmagát, mivel egynél nagyobb konstans szorzóval minden elem nagyobb, egynél kisebb konstans szorzóval pedig minden elem kisebb mint az eredeti érték.

## 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 indexszel. 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. A Dinamikus Tömbök egyszerűen egymás utáni helyeken tárolják az adatokat több helyet lefoglalva, mint ami szükséges és mutáció esetén elmozdítják akár 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.

2. Ábra: Vektor és Deque alapszerkezet

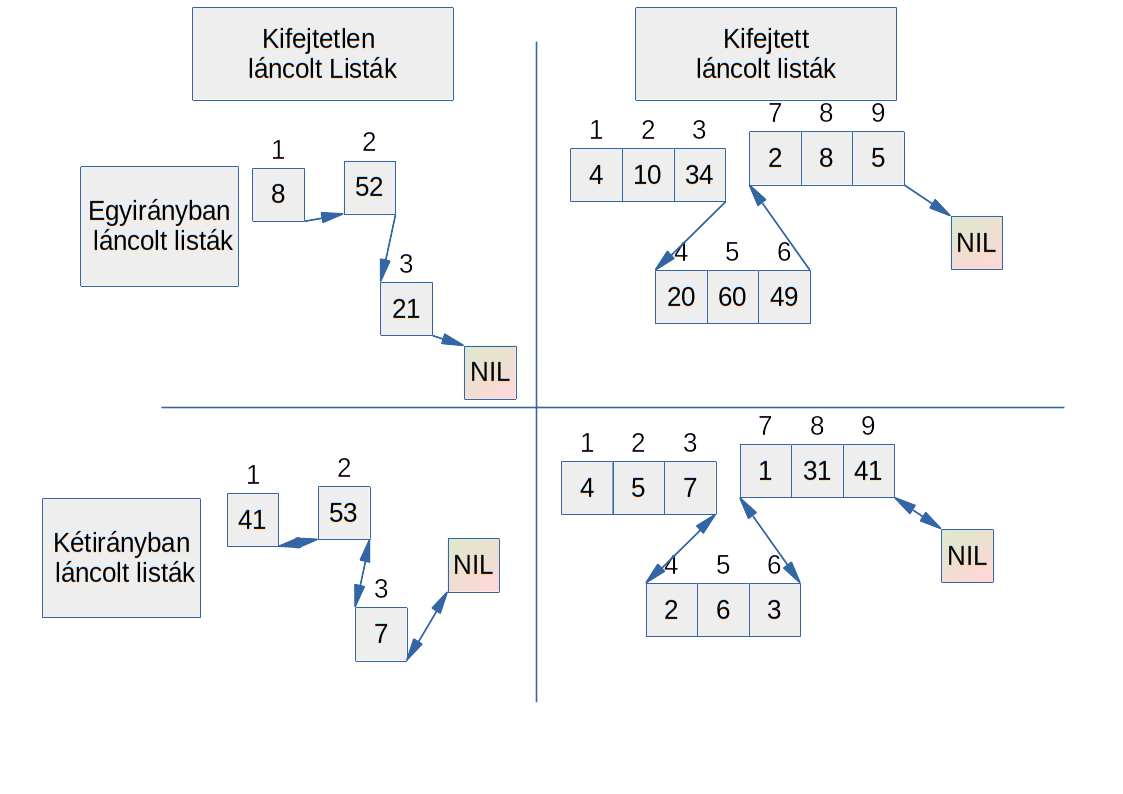
Az 1. Ábrán látható a növekedés és csökkenés fizikai helye Vektorok és Dequek esetében. Az index alapú elérés legrosszabb esetben is O(1), míg a mutáció legrosszabb és átlagos esetben Ordó=Theta(n). A Dequek legrosszabb esetben az elemek felét kell csak, hogy elmozítsák, míg a vektor legrosszabb esetben(első elem elé beszúrás, vagy annak törlése) az összes elemet elmozdítja. A Vektor fő előnye a hogy az első elem fix helye miatt lehet rá azzal hivatkozni. 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 láncszemekből 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 „x” 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, az új láncszem következő elemre mutató pointerét átállítjuk „x” következő elemre mutató pointerre, majd „x” előre mutató pointerét átállítjuk az újonnan létrehozott láncszem címére. Kétszeres láncolt lista esetén a visszafelé mutató pointereket is átállítjuk az előzőhöz hasonló gondolatmenet szerint, úgy, hogy az az új sorrendet tükrözze. A törlés szintén konstans idejű amennyiben megjegyeztük a törlendő előtti elemet, vagy 2-szeresen láncolt listával dolgozunk. Ekkor a mutáció szintén csak annyit igényel, hogy az új állapothoz igazítjuk a pointereket. A láncolt listáknál felmerülhet, hogy egynél több elemet tárolunk egy láncszemben. Ekkor unrolled listáról beszélünk. Ehhez általában Vektort vagy Dequet használnak. Ezen kívül a láncolt listáknak egyéb különleges megoldásai is lehetnek, például a nullpointer (NIL) helyett őrelemek, amik egyedi listavéget jelölnek, vagy „első előtti” elemet jeleznek. Ezen kívül logikailag körkörös adattárolást igénylő problémákat érdemes lehet körkörös láncolt listákkal leírni, ahol az utolsó elemet az első követi. A 2. Ábrán a Láncolás iránya és kifejtettség szerinti lehetőségek látszanak, nemkörkörös, nullpointer által befejezett példákon keresztül.

3. Ábra: A Listák főbb típusai

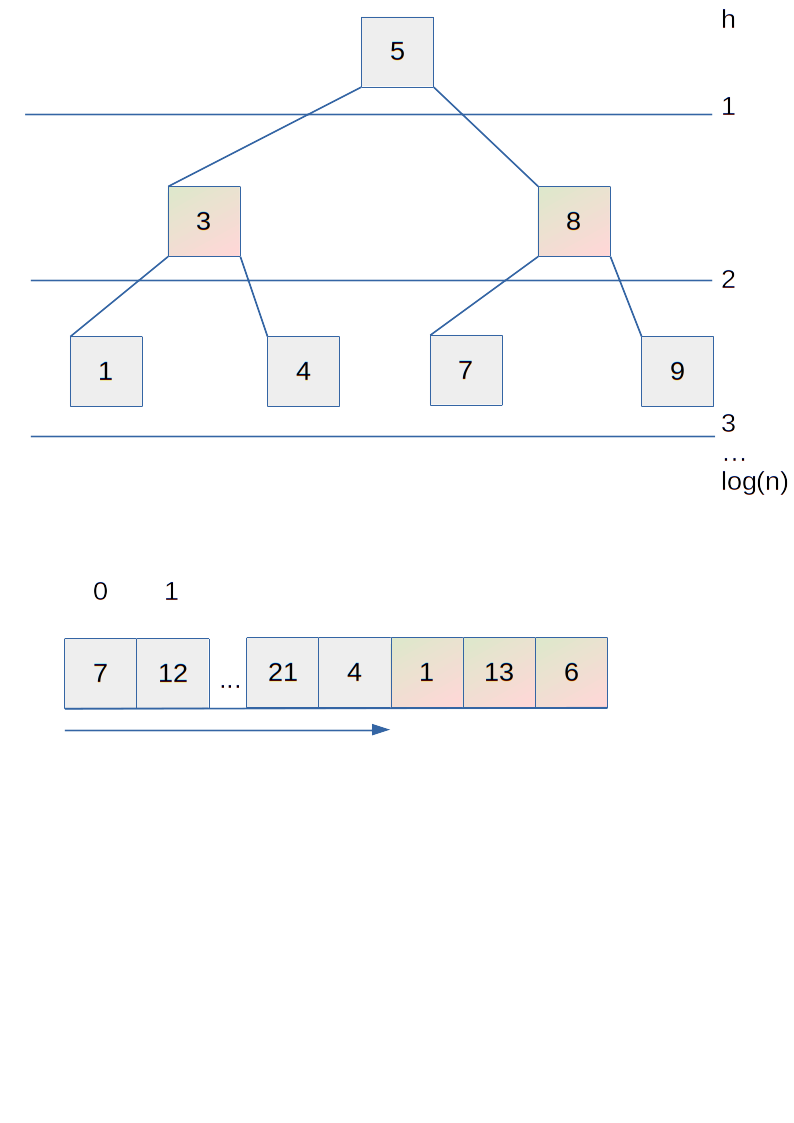
A láncolt listák egy fontos felhasználási módja, hogy fákat lehet velük implementálni. Ez úgy történik, hogy az adattag maga is lehet egy láncolt lista. A láncolt listákat általában közvetlenül olyan feladatok ellátására szánják, ahol a sorrendi, vagy esetleg fordított bejárás valószínű, és gyakori a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

## Piros**-Fekete-**Fa

A Piros-Fekete fa egy közel kiegyensúlyozott bináris keresőfa. Hogy a Piros-fekete Fát vizsgálhassuk, először ennek a jelentését kell megismernünk. 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 kiegyensúlyozott fa magassága lg(n), ahol a logaritmus alapja, a fa fokszáma.

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. Amennyiben a fa nem tökéletesen kiegyensúlyozott, de a kiegyensúlyozatlanság mértéke korlátozott, a fa közel kiegyensúlyozottnak tekinthető.

4. Ábra: Piros Fekete Fa

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

A Piros fekete fák fenntartanak néhány belső tulajdonságot, amelyek segítségével a közel kiegyensúlyozottság biztosított marad. Ezek a következők:

* Minden levél vagy piros vagy fekete
* A fa gyökere, és minden levél fekete.
* Minden piros csúcs gyerekei feketék
* A gyökértől minden levélig ugyanannyi fekete csúcs van.

A 3. Ábrán egy ezeket teljesítő Piros Fekete Fa látható. Ezek fenntartását átszínezésekkel, és ha kell, forgatásokkal éri el. A piros fekete fa képes fenntartani azt a tulajdonságot, hogy a Gyökértől legtávolabbi csúcs legfeljebb kétszer olyan messzi, mint a gyökérhez legközelebbi.

A beszúrás törlés és keresés műveletek a a Piros fekete fa kiegyensúlyozó műveleteivel együtt is Theta és Ordo(lg n) idejűek.

## B fák

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

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 művelet innen indul, a bináris és egyéb keresőfákhoz hasonlóan, viszont az egy csúcsban tárolt elemek számának növelésével, laposabbá tehető, és az O(lg(n)) idejű műveleteknél, a logaritmus alapja a tárolt elemek számának fele. A B fák emellett, azért is gyorsabbak a gyakorlatban, a bináris keresőfáknál, mivel egymáshoz közel tárolnak adatokat, így egy adott RAM elérésnél, amelyeknek a mérete általában 64 bájt, olyan adatokat is betölt a processzor, amelyeket nem kért eredetileg, de nagy eséllyel szüksége lesz rá később. Ez a tulajdonság még nagyobb előnnyé válik, amikor HDD-kről töltjük be az adatokat, mivel ekkor sokkal nagyobb memóriaegységek kerülnek betöltésre mindenképpen, amit a B fa képes kihasználni. A B fák csúcsaiban a fokszám nem konstans, hanem egy adott tartományban változik. Ez a tartomány általában egy fára jellemző maximális fokszám, és annak fele között mozog,kivéve a gyökérben, ahol megengedett, hogy ennél kevesebb legyen. A kulcsok száma egy csúcsban, eggyel kisebb mint a gyermekek száma. A csúcsok belső tartalmát, mind a kulcsok, mind a gyermekek mutatói esetében Dinamikus Tömbök látják el.

5. Ábra: Egyszerű B fa

A B Fák esetében a kiegyensúlyozottságot a csúcsok fokszámának határok között tartásával, és ezen határok megsértésének kezelésével oldja meg. Ha egy csúcs a kívánt beszúrás után is a maximális fokszám alatt marad, a megfelelő elem beszúrása elvégezhető, a megfelelő levélbe, és ez után nincs további teendő. Amennyiben egy levél beszúrás során a megengedettnél több elemet tartalmazna, kettévágásra kerül, és a középső elem a szülőbe kerül beszúrásra. Ekkor ez bekerül a megfelelő helyre, a sorba rendezett kulcsok közé, és ez válik az elválasztó kulccsá, a 2 új levél között. Amennyiben ez a szülőbe beszúrás a szülőt a megengedett fokszám fölé emeli, itt is szükség van egy kettévágásra, hasonló szülőbe kerülő elválasztó elemmel. Ez rekurzívan folytatódik, addig, amíg erre szükség van. Legvégső esetben ez egy új gyökércsúcs létrehozását jelenti. Mivel a fa magassága logaritmikus, és a csúcsok legnagyobb megengedett mérete konstans, ezért a teljes beszúrás legrosszabb esetben logaritmikus idejű. Törlés esetén több lehetőség is van. A legegyszerűbb eset, hogy egy olyan levélből törlünk, amely kellően nagy ahhoz, hogy a belőle való törlés ne okozzon problémát. Ekkor egyszerűen eltávolítjuk az adott értéket és készen vagyunk.

Törlés egyéb esetei

keresés

Altípusok

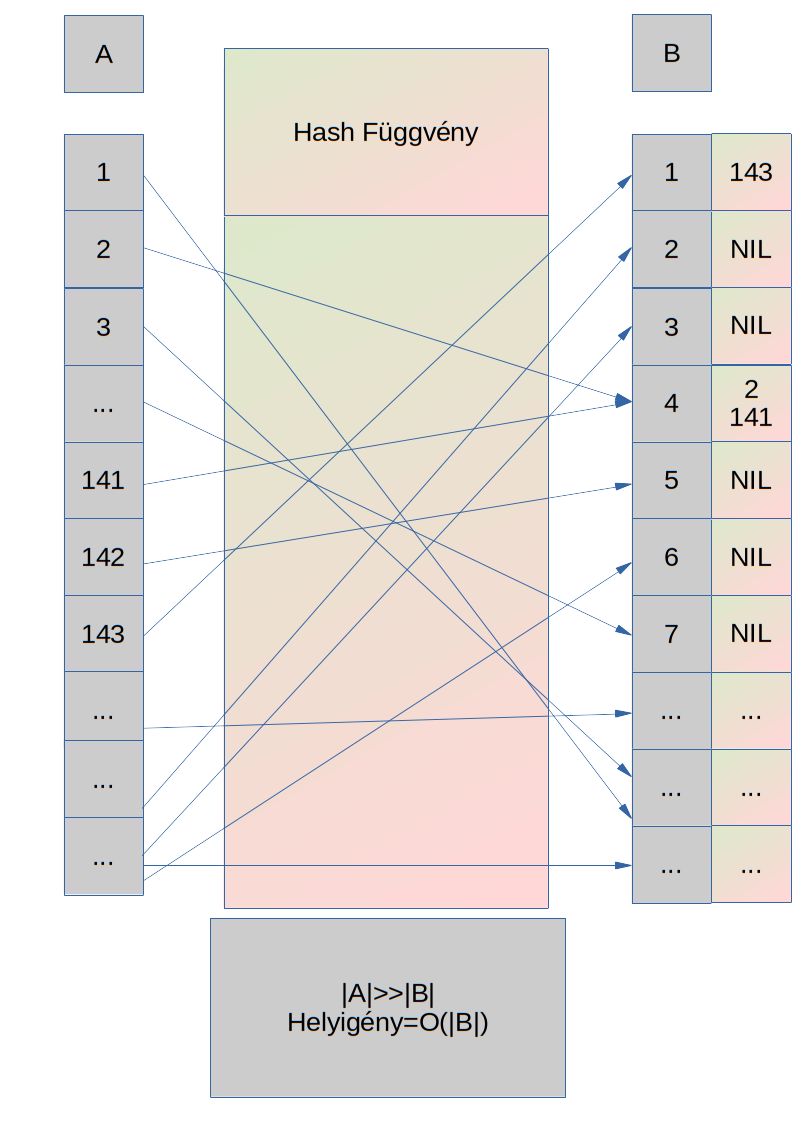
A B Fák alapértelmezett szerkezetén túl, előfordulnak további variánsok. Ilyen például a B+ fa. A B+ Fa abban tér el az alap B Fa szerkezettől, hogy a tárolt elemek kizárólag a levelekben helyezkednek el. Ekkor a magasabb szinteken, a levelekben tárolt adatok másolatai vannak, oly módon, hogy azok a bal legszélső gyermek legnagyobb elemei. Ezen felül a levelek unrolled láncolt listaként is össze vannak kötve, ezzel gyorsítva a szekvenciális elérést.

## 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 hasonló 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 az egyetlen ott található értékre. Ez az átlagos és egyben a legjobb eset.

6. Ábra: Hasítótábla Szerkezete

Sajnos mivel több kulcsot kell, hogy rendeljünk egy indexhez, ezért elkerületlen, hogy különböző kulcsok időnként ne használják ugyanazt a helyet. Ezt ütközésnek nevezzük. Ilyen például az 5. Ábrán látható 4. hely, a tároló B tömbben, amelyhez a 2 és a 141 kulcsok is hozzárendelésre kerültek. 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.

## 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 Későbbiekben 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, és a méretük megegyezik a Dictionary méretével. A tényleges címeket egy i indexre úgy kapjuk meg, hogy i-t maradékosan elosztjuk a tároló tömbök méretével. A hányados adja a keresett indexű elem tömbjének az indexét, a maradék pedig az ezen belüli indexet. Mivel a maradékképzés és az egészeken végzett osztás rendkívül lassú, érdemes a Dictionary méretet annak a kettő hatványnak megválasztani amelyik a kettő hatványok között az első, amely négyzete a tárolt elemek számánál nagyobb. Thehát például 37 elem esetén ez 8, mivel 64 a legkisebb olyan 2 hatvány amely 37-nél nagyobb.

Ekkor az osztás megvalósítható eltolással, úgy, hogy annyiszor toljuk az osztandó értéket jobbra, mint ahányadik kettő hatvánnyal osztunk (m), ez 8 esetén log2(8), vagyis m=3. A moduló szintén sokkal gyorsabb, mivel ez elvégezhető úgy, hogy csak a legkisebb ugyanezt a 2 hatvány kitevőt vesszük (m) vesszük, és ennyi bitet meghagyunk jobboldalt, majd a többit 0-ra állítjuk. Tehát például m=3 esetén a 8-al történő moduló megoldható tetszőleges 2-es számrendszerbeli számra (például 010101102) úgy, hogy a 3 legkisebb helyi értékű bitet meghagyjuk a többit pedig nullára állítjuk( a példában ez 000001102).

Mutáció során az egész adatstruktúra újraépítésre kerül, a Dictionary méretének megállapítása után,azzal megegyező méretű Vektorokba szervezve kerülnek tárolásra az elemek.

## Összehasonlítás(Tábla elemzéssel)

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 aszimptotikus tulajdonságok csak az elemszám növekedésével válnak jelentőssé. Kis elemszámnál, a konstans időigényű részműveletek válhatnak hangsúlyossá, míg a végtelenben vett határértékeknél ezek teljesen eltűnnek. Nagyon kis méretnél általában a Dinamikus Tömbök a leggyorsabbak minden feladatra.

1. Táblázat: Adatstruktúra Sebességek

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Létrehozás | Megsemmisítés | 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)*** |

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 időt igényelnek. Ezen felül, ha 2 irányba láncolt 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). A HAT megsemmisítési ideje Theta(sqrt(n)).

Felmerülhet az is, hogy A Dinamikus Tömböket kereső adatstruktúraként használjuk, úgy, hogy az elemeket sorba rendezzük. Mivel ekkor adott elem megkeresése O(lg n) időt vesz igénybe, bináris keresés során, ez jelentősen jobb, mint rendezetlen Dinamikus Tömbökben keresni, viszont csak ugyanannyira jó, mint a keresőfák. A probléma ezen kívül az, hogy a beszúrás és törlés müvelet lineáris idejű, amíg ez a fáknál logaritmikus idejű volt.

Mindezeken túl, a különnböző adatstruktúrák különböző problémák megoldására lettek kifejlesztve. Ennek megfelelően minden összehasonlításnál figyelembe kell venni, hogy milyen probléma megoldására kerresük a megfelelőt. Az eltérő felhasználási módok miatt a Gyorsított Tömböt leginkább a Dinamikus Tömbökkel érdemes összehasonlítani. Ezen kívül, alkalmanként, a fizikai eszköz tulajdonságai is beleszólhatnak abba, hogy melyik a legjobb megoldás. Esetleges új technológiák esetén érdemes figyelembe venni, hogy a legtöbb komplex adatstruktúra egyszerűbb adatstruktúrákból építkezik, így az egyszerűbb adatstruktúrák javítása, vagy lecserélése egy alternatívával, a komplexebbb adatstruktúrák sebességére is kihathat. Alapjában véve az adatstruktúrák között nincsen objektíven legjobb, még az aszimptotikus idők alapján sem. Viszont előfordulhat, hogy egyes hardware tulajdonságok az azonosnak tűnő tulajdonságok között külömbséget teremtenek. Ilyen a B fák nagyobb gyakorlati sebessége a piros fekete fákhoz képest, mivel a B fák egy memóriaelérés során több értékes adatot kérnek be, mivel általában a kulcsok egymás után találhatóak a memóriában. Ugyanakkor, a tényleges implementáció során a kérdés még összetettebbé válik, és adott javulás lehetősége még nem garantál jobb sebességet. Ha például rendkívül nagy kulcsokat használunk, amelyek egy cache line méretét is elérik, a B fák ezen előnye megszűnik.

# Gyorsított Tömb

A Gyorsított Tömb egy adatstruktúra, amely a Statikus Tömbhöz hasonlóan indexelve tárolja az elemeit. A Vektor és a Deque a Statikus 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 Dinamikus 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 alapú 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.

7. Ábra: A Gyorsított Tömb felépítése

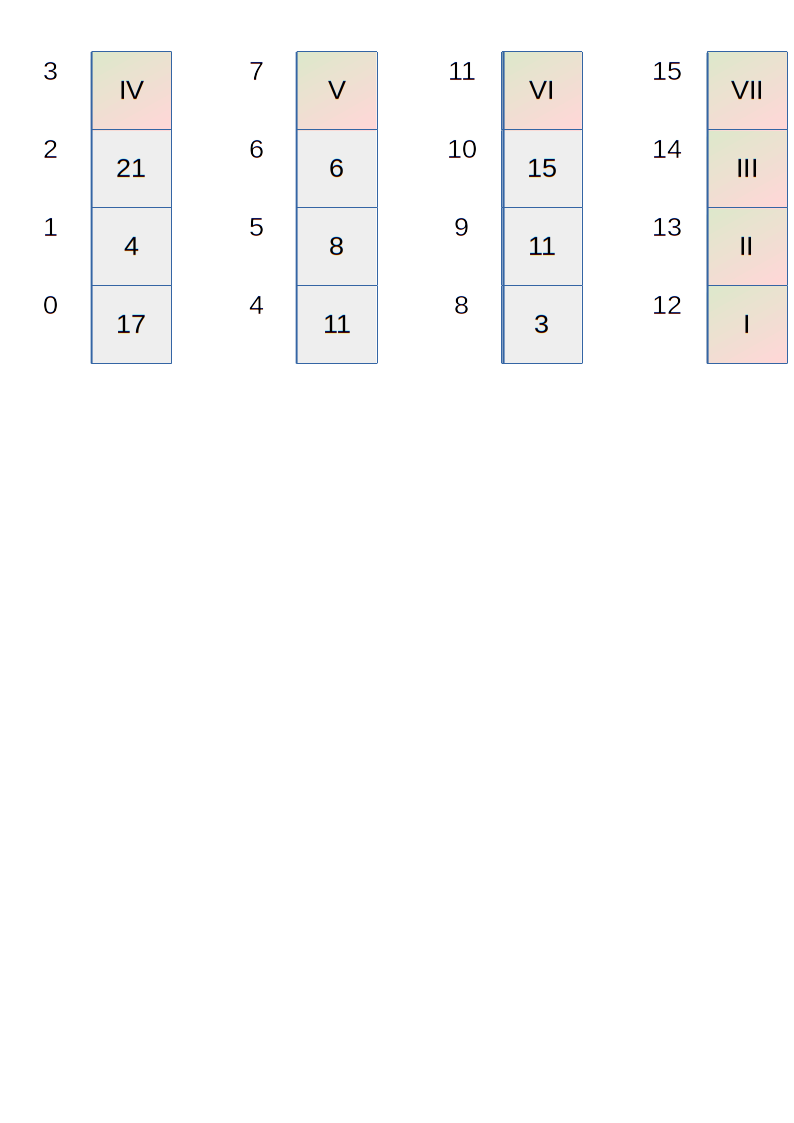
## Leírás és elemzés

.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.

8. Ábra: A Gyorsított Tömb Növekedése

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.

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 leírható egy négyzet alakú magra történő építkezésként. A 7. 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 a 7. Ábrán a 4. oszlopként van jelölve. Ezt Újtömbnek nevezzük. 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 mind egyszer növelve, egyszer pedig csökkentve lettek méretben, így ezek mérete nem változik.

### Létreozás

Ahhoz, hogy egy Adatstruktúrát használni tudjunk, ehhez először létre kell tudnunk hozni. Ez jelen esetben egy üres adatstruktúrát jelent, amelyben néhány inicializációt hajutnk végre. Ha egy másik Dinamikus Tömb másolatát akarjuk így tárolni, ez után beszúrhatjuk annak elemeit sorban, a később leírt módon.

A létrehozás üresen:

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 a magon felül, mennyi extra elem van.

Create():

1 felső\_tömb←vector()

2 insert(felső\_tömb,deque())

3 population←0

4 popsqrt←0

5 popextra←0

accarr() : metaData(0){

content = new deque<SQ<T> \*>();

content->push\_back(new SQ<T>());

}

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

Szükség esetén meg is kell tudnunk semmisíteni a már nem használt adatstruktúrákat, hogy ne fogyjunk ki memóriából. Az összes Alsó majd a Felső Tömb megsemmisítése.

Delete():

1 for minden i eleme felső\_tömb -re

2 do delete(i)

3 delete(felső\_tömb)

~accarr(){

for (auto i : \*content)

delete i;

delete content;

}

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

### GetRelPos

Ahhoz, hogy bármely indexnél műveleket hajthassunk végre, tudnunk kell, hol van ez valójában, fizikai index szerint. Ezt a GetRelPos függvénnyel tudjuk lekérni, mely egy érvényes indexet kap és egy fizikai indexxel tér vissza.

A Működése a következő:

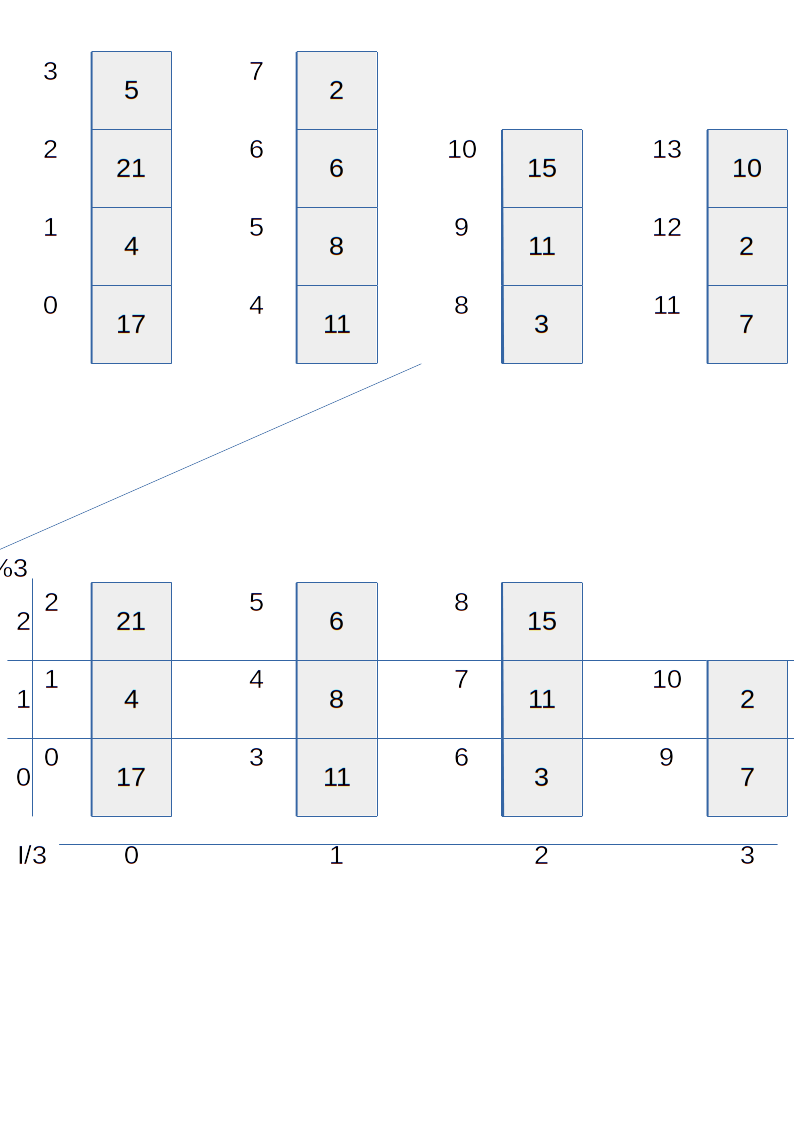
* 0-nál (0,0) a visszatérési érték.
* Ha az Újtömb még nem telt be, az Alsó Tömbök hosszának és az indexnek a hányadosa megadja a keresett Alsó Tömböt, a moduló pedig a belső Indexet:

(I / √n, √n | I).

* Ha betelt az Újtömb, akkor az Alsó Tömbök mérete sorban 1-el növelésre kerül. Először meg kell állapítani hogy a keresett index az így növelt tartományba esik-e.

Ha igen, akkor ezen belül a II. Pont szerint járunk el,√n+1 hosszú Alsó Tömbökkel számolva.

Ha nem, akkor kivonjuk az így tárolt elemek számát i-ből, ezzel kiszámoljuk a keresett indexet a II. Pont szerint, majd hozzáadjuk, a II. pont módjára kiszámolt felső indexhez, a növelt Alsó Tömbök számát.

9. Ábra: Elemek indexei

A 8. Ábrán alul, az elemek számát a tőlük balra lévő szám mutatja. Itt egy olyan Gyorsított Tömb látható amely jelenleg a végén növekszik. Látható, hogy az Alsó Tömböket sorban bejárva, először a maradék érték feltöltődik, majd amikor ez eléri a teljes hosszt, a hányados eggyel nő, és a maradék visszaesik 0-ra. Ennek megfelelően az egész adatstruktúra végig van indexelve. Érvénytelen fizikai címet az utolsó Alsó Tömb végén csak érvénytelen logikai címmel lehet elérni.

A felül növekvő Gyorsított Tömb visszavezethető 2 különböző helyzetre, melyek mind kezelhetőek, a végén növekvő Gyorsított Tömbre visszavezetve. Ekkor a keresett index vagy beleesik a növelt hosszú Alsó Tömbökbe, vagy azokon túl van. Ezt úgy állapítjuk meg, hogy a keresett indexet gyökN+1-el elosztjuk. Ha a hányados nagyobb mint a megnövelt hosszú Alsó Tömbök száma, akkor kívül esik, ha legfeljebb akkora akkor beleesik a megnövelt Alsó Tömbökbe. Mivel Minden Négyzet alakú magon kívül tárolt elemet vagy az Újtömb, vagy a többi tömb extra tagja tárolja, így a megnövelt Alsó Tömbök száma megegyezik a négyzet alakú magon túl tárolt elemek számának, és az Újtömb tárolt elemszámának különbségével. Az Újtömb GyökN+1-edik tagjával, a következő négyzet alakú mag elérésre kerül.

GetRelPos(index):

1 if popsqrt = 0

2 then return pair(0, 0)

3 if popextra<=popsqrt

4 then return divmod(index, popsqrt)

5 else

6 if (popextra - popsqrt) > index / (popsqrt + 1)

7 then return divmod(index, popsqrt + 1)

8 else

9 uninced ← index - (popextra - popsqrt) \* (popsqrt + 1)

10 uc←divmod(uninced,popsqrt)

11 return list(popextra - popsqrt + uc.first, uc.second)

pair<int, int> getRelPos(int index) const {

if (popsqrt == 0){

return pair<int, int>(0, 0);

}

if (popextra < popsqrt){

return divmod(index, popsqrt);

}

else{

if ((popextra - popsqrt) > index / (popsqrt + 1)) /{

return divmod(index, popsqrt + 1);

}

else{

int uninced = index - (popextra - popsqrt) \* (popsqrt + 1);

auto uc = divmod(uninced, popsqrt);

return pair<int, int>(popextra - popsqrt + uc.first, uc.second);

}

}

}

Az index megállapítása során, először az az eset van kezelve, amikor a populáció 0. Ekkor, feltéve hogy érvényes index került lekérésre, ami csak beszúrásnál lehet, ez az érték 0. Ennek az esetnek a külön kezelésére azért van szükség, mert a fizikai index megállapítása során szükség van osztásra, és modulóra, a populáció négyzetgyökével.

Ez után megvizsgáljuk, hogy az Újtömb betelt-e. Ezt könnyen megkaphatjuk a tárolt metaadatokból, anélkül, hogy az Újtömböt betötenénk. Ehhez megvizsgáljuk, hogy a Négyzet alakú magon kívül tárolt elemek száma(Popextra) nagyobb-e, mint az Újtömbben tárolható elemek száma.

Ha nem, akkor az Újtömb még nem telt be, és emiatt egyszerű osztás és moduló művelettel megkaphatjuk a kívánt fizikai indexet, ahogy az az 5. sorban látható. A keresett index és a popsqrt hányadosa adja a az Alsó Tömb indexét, a moduló ugyanezekkel pedig az ezen belűli indexet.

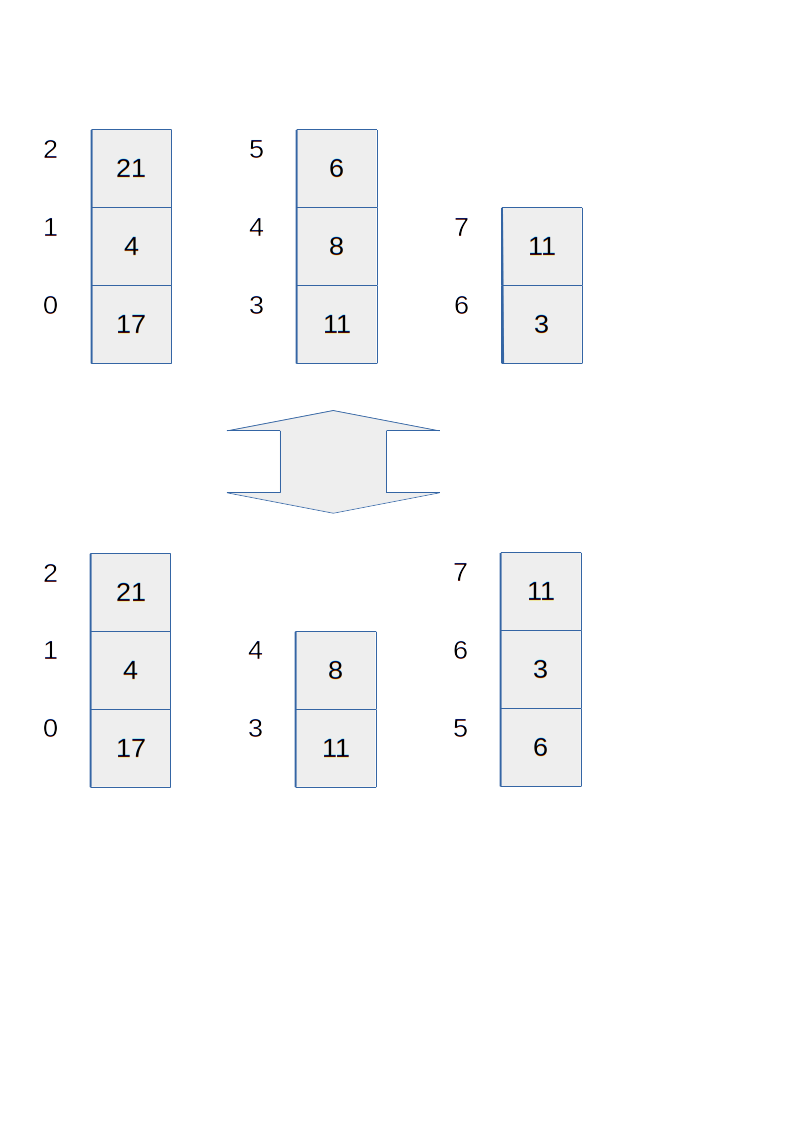
Ha igen, akkor több extra elem van tárolva, mint ami az Újtömbben elfér, ezek az Alsó Tömbök extra tagjaiként vannak tárolva és ennek megfelelően korrigálni kell az extra mérettel ami az első néhány Alsó Tömböt jellemzi. Ehhez szükségünk van az inkrementálatlan részbe eső, index által elért elemek számára. Ezt a 10. sorban az uninced nevű változóba nyerjük ki. Ehhez ki kell vonnunk, a növelt méretű Alsó Tömbökbe eső elemek számát, a keresett indexből. A növelt részbe eső elemek száma, annyi, mint az lévő Alsó Tömbök hossza( popsqrt+1 ), szorozva a növelt Alsó Tömbök számával(popextra - popsqrt). Ezután az inkrementálatlan részbe eső indexelt értéknek, kiszámoljuk az indexét, mintha ez az 5. sorbeli eset lenne. Ezzel megkapjuk az inkrementált rész után eső Alsó Tömbök számát és az azon belüli indexet, a 11. sorban. Végül ezt korrigáljuk úgy, hogy a visszatérendő Alsó Tömb indexhez hozzáadjuk az inkrementált Tömbök számát.

Mivel minden esetben Konstans idejű műveletekre van szükség, rekurzió vagy iteráció nélkül, maga a teljes függvény is Theat=Omega=Ordó(1).

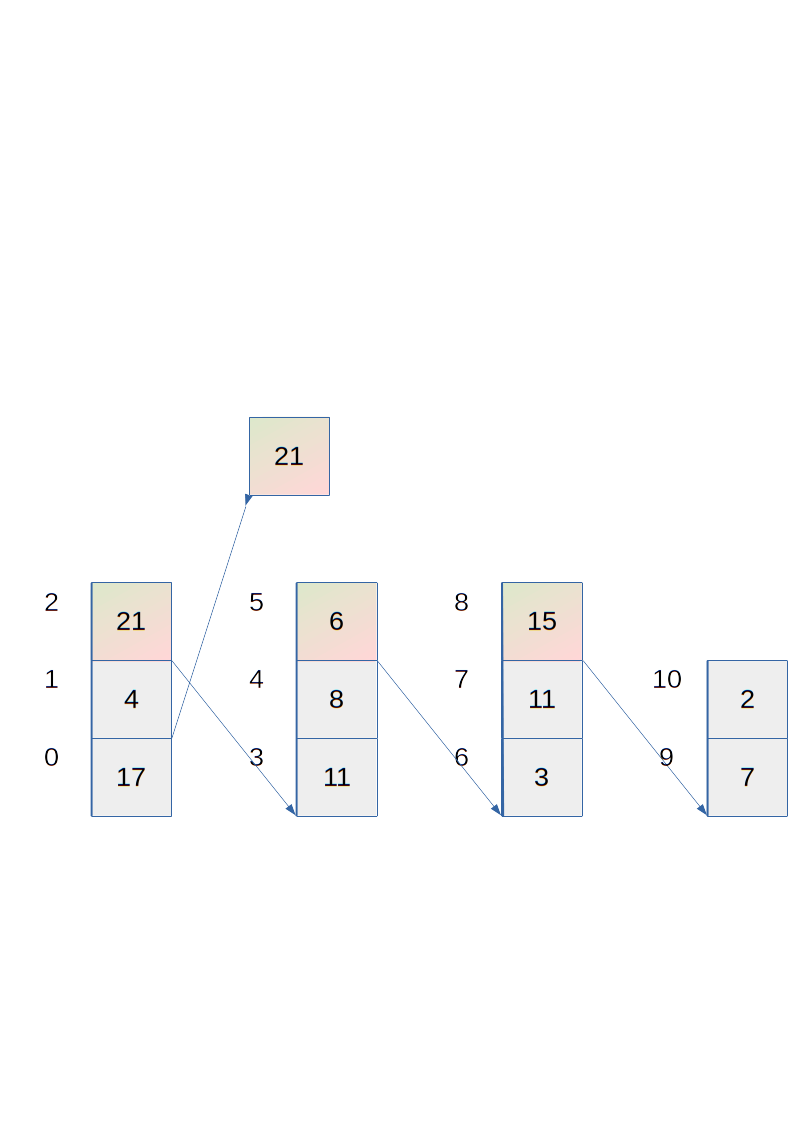
### BalanceShift:

A kiegyensúlyozás és annak futási ideje elemi fontosságú a Gyorsított Tömbben. A mutáció majd kiegyensúlyozás az alapötlet, a piros fekete fák és sok egyéb kiegyensúlyozott adatszerkezet mögött. A Gyorsított Tömb esetében a kiegyensúlyozás kihasználja a dequek azon tulajdonságát, mely szerint a doublestack műveletek mind konstans amortizált időben futnak le ( nagyobb tároló újrafoglalásától eltekintünk).

Egy adott Alsó Tömb végéről az azt követő elejére átvitt elem logikai indexe nem változik. Ugyanilyen gondolatmenet szerint, egy adott Alsó Tömb elejéről, az azt megelőző Alsó Tömb végére helyezett elem logikai indexe szintén nem változik. Ha egy Alsó Tömbbe be is illesztünk egy elemet, illetve ki is veszünk egyet, a hossz nem változik. A kiegyensúlyozást az teszi lehetővé, hogy a fente említett tények miatt, a logikai index és a fizikai index egymástól elválasztva kezelhető. Ennek köszönhetően, mutáció után a Gyorsított Tömb helyreállítható.

10. Ábra: Balance eltolás szemléltetése

A 9. Ábrán látható két azonos logikai felépítésű de eltérő fizikai felépítésű Gyorsított Tömb. Az alsó egy olyan állapotot mutat, ami a középső Alsó Tömbből való törlés után jelentkezik, a felső pedig ennek egy korrigált állapotát. A törlés után a logikai sorrend nem sérül, csupán az elemek eloszlása. Az egyensúly felborulása esetén, tetszőleges mozgatásához elég végigmennünk a forrás Alsó Tömbtől a Cél alsó Tömbig, amiknek a Felső Tömbbeli indexét, a BalanceShift paraméterben kapja meg és a megfelelő végeken egy elemet levenni, és a következő Alsó Tömbök megfelelő vég helyezni. A balanceShift megfelelő paramétereinek megválasztása, az azt meghívó függvény feladata.

11. Ábra: Kiegyensúlyozás beszúrás után

Ehhez megnézzük melyik irányba szeretnénk elmozdulást elérni. Ha kisebb indexű Alsó Tömb felől nagyobb indexű Alsó Tömb felé, akkor az Alsó Tömbök végéről vesszük el az elemeket, és a következő Alsó Tömb elejére helyezzük azokat. Ha nagyobb index felől kisebb felé, akkor az Alsó Tömbök elejéről vesszük el az elemeket és a következő Alsó Tömb végére kerülnek. Ha a kiindulási és érkezési Alsó Tömb azonos, nincs szükség semmilyen műveletre, mivel az egyensúly nem borult fel, a Mutáció a változás tervezett helyén történt. A 10. Ábrán a „21” érték beszúrása látható az 0-s index után. Ekkor az első Alsó Tömbben beszúrás megy végbe, ami miatt itt egy extra elem van. Ennek orvosolására, a Beszúrás, a Törléshez hasonlóan a BalanceShiftet hívja meg. Jelen esetben ez egy olyan függvényhívást jelent, ahol a többlet eltolása az elsőtől az utolsó elemig történik. Ekkor a 21 értékű blokk ( amelyik eredetileg a 2-es logikai indexen volt ) levételre kerül az 1. Alsó Tömbről és a 2. Alsó Tömb elejére kerül. Ekkor a logikai helye nem változik, viszont a fizikai többlet a kívánt helyhez közelebb került. Ez után a 2. Alsó Tömb utolsó eleme, a „6”-os értékű kerül levételre, és a 3. Alsó tömb elejére kerül. Végül a 3. Alsó Tömb végén található „15”-ös átkerül a negyedik Alsó Tömb Elejére. Ezzel a szerkezet korrigálva lett.

BalanceShift(from,towards):

1 if from < towards

2 then temp ← pop\_back(felső\_tömb[from])

3 for i ← from + 1 to towards

4 do push\_front(felső\_tömb[i])

5 temp ← pop\_back(felső\_tömb[i])

6 temp ← push\_front(felső\_tömb[from])

7 else if from > towards

8 then temp ← pop\_front(felső\_tömb[from])

9 for i ← from - 1 back\_to towards

10 do push\_back(felső\_tömb[i])

11 temp ← pop\_front(felső\_tömb[i])

12 temp ← push\_back(felső\_tömb[from])

void balanceShift(int from, int to)

{

if (from < to){

T temp = ((\*content)[from])->pop\_back();

for (int i = from + 1; i < to; i++){

temp = ((\*content)[i])->revplace(temp);

}

((\*content)[to])->push\_front(temp);

}

else if (from > to){

T temp = ((\*content)[from])->pop\_front();

for (int i = from - 1; i > to; i--){

temp = ((\*content)[i])->placing(temp);

}

((\*content)[to])->push\_back(temp);

}

}

A tényleges kód során megvizsgáljuk a relációt az indulási és érkezési index között. Ha az indulási index kisebb mint a cél index, a végekről vesszük le, az elemeket, és a következő elejére helyezzük. Ehhez először a kiindulási Tömb végéről veszünk le egy elemet, és utána a kiinudlási és a cél Tömb közötti összes Alsó Tömbön végigmegyünk, úgy, hogy mindegyikbe egyszer beszúrunk az eljén és leveszünk egy elemet a végén. Ez után végül a cél Tömb elejére beszúrjuk az utoljára levett elemet. Ha a Cél nagyobb mint a Kiindulási Index, ugyanígy cselekszünk, azzal a küönbséggel, hogy az Alsó Tömbök elején vesszük el az elemeket, és az Alsó Tömbök végére szúrunk be.

Mindezekből látszik, hogy a BalanceShift során konstans lépést végzünk az iterációkon kívül illetve legfeljebb egy iterációt végzünk, a 3. és a 10. sorban látható lépés szerint. Ez a Kiindulási index+1 és a Cél index távolsága, azaz különbségük abszolút értéke. A kiindulási és Cél index is, a Felső Tömb indexei. Tehát a különbségük legfeljebb a Felső Tömb hossza. Ez O(GyökN), amin a gyengébb constans +1 tényező nem változtat.A ciklusmag pontosan 2 konstans idejű lépést végez. Mindezek alapján a sebesség O(GyökN\*1)=O(GyökN). A mean line segment alapján a várható sebesség Theta(GyökN/3)=Theta(GyökN). Legjobb esetben nem léünk be az iterációba, így ekkor Omega(1).

### További segédfüggvények:

A **divmod** belső felhasználású függvény. Feladata az, hogy egyszerűen elvégezze, az osztás és moduló műveleteket ugyanazokkal az értékekkel.

Ehhez egyszerűen visszatér azzal a listával, ami a hányadosból és a modulóból áll.

divmod(a,b):

1 return list(floor(a/b),a % b)

pair<int, int> divmod(int a, int b){

return pair<int, int>(a / b, a % b);

}

A **Populációnövelés** valósítja meg, a metaadatok naprakészen tartását beszúrás után. Ezen felül megállapítja, hogy szükséges-e új Alsó Tömböt foglalni.

Ehhez megjegyzi a populáció régi négyzetgyökét, majd frissíti a populáció értékét, annak négyzetgyökét, és az ez fölött tárolt elemek számát. Végül visszatér azzal, hogy az új négyzetgyök megegyezik-e a régivel( ami akkor és csak akkor történik, ha új négyzet lett elérve).

incPop():

1 population←population+1

2 oldSQRT←popSQRT

3 popsqrt←floor(sqrt(population))

4 popextra←population-popsqrt\*popsqrt

5 return oldSQRT=popSQRT

bool incPop(){

population++;

auto oldsqrt = popsqrt;

popsqrt = SQRT(population);

popextra = population - popsqrt \* popsqrt;

return oldsqrt != popsqrt;

}

A **populációcsökkentés** frissíti a metaadatokat, törlés után. Ezen felül viszatér azzal, hogy törölni kell-e a legutolsó Alsó Tömböt.

Ehhez, a populáció növeléshez hasonlóan, megjegyzi a populáció régi négyzetgyökét, majd frissíti a populáció értékét, annak négyzetgyökét, és az ez fölött tárolt elemek számát. Végül visszatér azzal, hogy az új négyzetgyök megegyezik-e a régivel( ami akkor és csak akkor történik, ha új négyzet lett elérve).

decPop():

1 population←population-1

2 oldSQRT←popSQRT

3 popsqrt←floor(sqrt(population))

4 popextra←population-popsqrt\*popsqrt

5 return oldSQRT=popSQRT and population>0

bool decPop(){

population--;

auto oldsqrt = popsqrt;

popsqrt = SQRT(population);

popextra = population - popsqrt \* popsqrt;

return oldsqrt != popsqrt;

}

A calcInsertPlace a 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.

calcInsertPlace():

1 if popextra < popsqrt

2 then return list(popsqrt, popextra)

3 else

4 then return list(popextra - popsqrt, popsqrt)

pair<int, int> calcInsertPlace() const{

if (popextra < popsqrt){

return pair<int, int>(popsqrt, popextra);

}

else{

return pair<int, int>(popextra - popsqrt, popsqrt);

}

}

Keresés, subrotate és egyéb műveletek, amikre nem lett szánva

A calcDeletePlace a törlés során, a helyes méretcsökkenés hellét adja vissza. Ezzel a függvénnyel kérhető le, hogy egy tölrölt elem esetén, honnan kell annak helyére hozni a kiegyensúlyozó új elemet.Bizonyos értelemben a calcInsertPlace ellentéte.

calcDeletePlace():

1 if popextra ==0

2 then return list(popsqrt-1, popextra-1)

3 else if popextra<=popsqrt

4 then return list(popsqrt, popextra)

5 else

6 then return list(popextra-popsqrt-1,popsqrt)

pair<int, int> calcDeletePlace() const{

if (popextra == 0){

return pair<int, int>(popsqrt - 1, popsqrt - 1);

}

else if (popextra <= popsqrt){

return pair<int, int>(popsqrt, popextra);

}

else{

return pair<int, int>(popextra - popsqrt - 1, popsqrt);

}

}

### 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.

Beszúrás( index, value):

1 place←getRelPos(index)

2 insert(felső\_tömb[first(place)], felső\_tömb[second(place)], value)

3 balanceShift(first(place), first( getInsertPlace( population)))

4 todo ← incPop()

5 if todo

6 then push\_back(felső\_tömb, vector() )

void insert(int index, const T &value){

auto to = metaData.getRelPos(index);

((\*content)[to.first])->addTo(to.second, value);

balanceShift(to.first, metaData.calcInsertPlace().first);

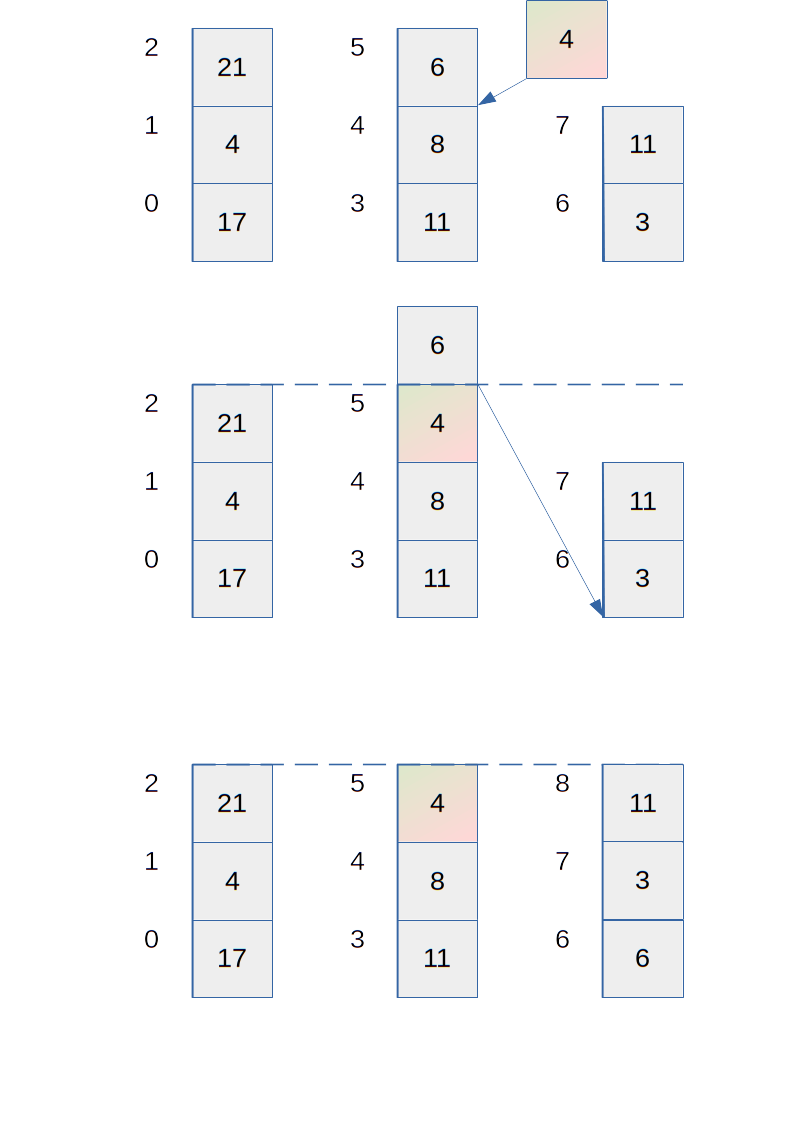
auto todo = metaData.incPop();

if (todo)

content->push\_back(new SQ<T>());

}

A 11. Ábrán látható egy példa a Beszúrás teljes menetére. Miután a GetRelPos megkeresi a kívánt fizikai helyet, ami itt a 4. index után van, elvégzésre kerül a dequebe beszúrás. Ekkor a beszúrt „4” érték jó logikai helyen van viszont többlet jelentkezett a második Alsó Tömbben ahova a beszúrás történt. Ez után a CalcInsertPlace által megadott növekedési helyre, a balanceShift eltolja az elem többletet, ezzel megszüntetve a helytelen eloszlását az elemeknek. Amennyiben a beszúrás olyan helyre került volna, logikailag, ahova a fizikai növekedés is esik, a balanceshift nem végez semmilyen műveletet. Midezek után a Gyorsított Tömb Metaadatai frissítésre kerülnek és a frissítési függvény visszatér azzal, hogy szükséges-e új Alsó Tömbbel bővíteni a Felső Tömböt. Ez jelen esetben így van, mivel új négyzet lett elérve az elemek szerkezete által( 3×3 ).

12. Ábra: Beszúrás Menete

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 annyi, 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 adat szerkezet 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.

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.[link]

Legjobb esetben a növekedési helyre esik a kívánt logikai index szerinti beszúrás, ezért ekkor nincs további nem konstans idejű teendő, mivel ekkor a Balanceshift nem tesz semmit. Ezen felül, a növekedési pont mindig egy Alsó Tömb végére esik.

Omega=1

Theta=Ordo=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.

Törlés(index):

1 place←getRelPos(index)

2 erase(felső\_tömb[first(place)], felső\_tömb[second(place)])

3 balanceShift( first( getDeletePlace()),first(place))

4 todo ← decPop()

5 if todo

6 then pop\_back(felső\_tömb)

void erase(int index){

auto to = metaData.getRelPos(index);

((\*content)[to.first])->deleteFrom(to.second);

balanceShift(metaData.calcDeletePlace().first, to.first);

auto todo = metaData.decPop();

if (todo && (content->size()) > 1)

content->pop\_back();

}

Mivel nincs se iteráció, se rekurzió, a hívott függvények döntik el itt is, a futási időt. Ezek között a Dequeből törlés, és a BalanceShift a nem konstans idejű. A dequeből törlés a Dequebe beszúráshoz hasonlóan viselkedik idő szempontjából, tehát legjobb esetben konstans, legrosszabb és átlagos esetben gyökN-es. Mindezek alapján a Törlés során a futási idő itt is, Legjobb esetben Konstans, legrosszabb és átlagos esetben GyökN-es.

### Elérés

Adott helyre **írás** esetén szükségünk van a cél index fizikai helyére amely alapján tudjuk, hogy melyik Alsó Tömböt kell elérni, és azon belül melyik elemet. A fizikai címhez egyszerűen meghívjuk a getRelPos függvényt, a cél logikai indexxel. A visszatérési érték első eleme a keresett alsó töömb indexe, a második érték az ezen belűli index. Ez után ezt az elemet elérjük, és ott elvégezzük az írás műveletét.

Írás(index,érték):

1 tarvector←first(getRelPos(index))

2 hely←second(getRelPos(index))

3 tarvector[hely]=érték

void setAt(int index, const T &value) const{

refAt(index) = value;

}

Mivel mindegyik lépés konstans idejű, és fix darab van belőlük, így a teljes művelet is konsttnas idejű.Omega=Theta=Ordo(1).

Adott indexről történő **kiolvasásnál**, az íráshoz hasonlóan járunk el. Megkeressük a kapott indexhez tartozó Fizikai címet, majd a fizikai cím első tagja által mutatott Alsó Tömbből kiolvassuk a fizikai cím második eleme által mutatott értéket.

Olvasás(index):

1 tarvector←first(getRelPos(index))

2 hely←second(getRelPos(index))

3 return tarvector[hely]

T getAt(int index) const{

return refAt(index);

}

Mivel mindez konstans darab konstans idejű részműveletet igényel, a futási idő is Omega=Theta=Ordó(1).

Az implementációs kódok az alábbi referencialekérő függvényre hivatkoznak:

T &refAt(int index){

auto pos = metaData.getRelPos(index);

return ((\*content)[pos.first])->access(pos.second);

}

# Implementáció, mérések

A Gyorsított Tömbhöz készítettem egy implementációt. 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. Ugyanakkor felhasználtam több helyi változót amelyek index elérésnél le vannak kérdezve, de csak mutáció során változnak. Ilyen például az endLoad nevű boolean változó, amely azt tárolja, hogy a Gyorsított Tömb, az új vektorban növekszik(true), vagy a vektorok végén(false). Ezek lényegében csak kézzel megírt output cache-ek voltak, néhány belső beépített függvény számára.

## 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 Dinamikus 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.

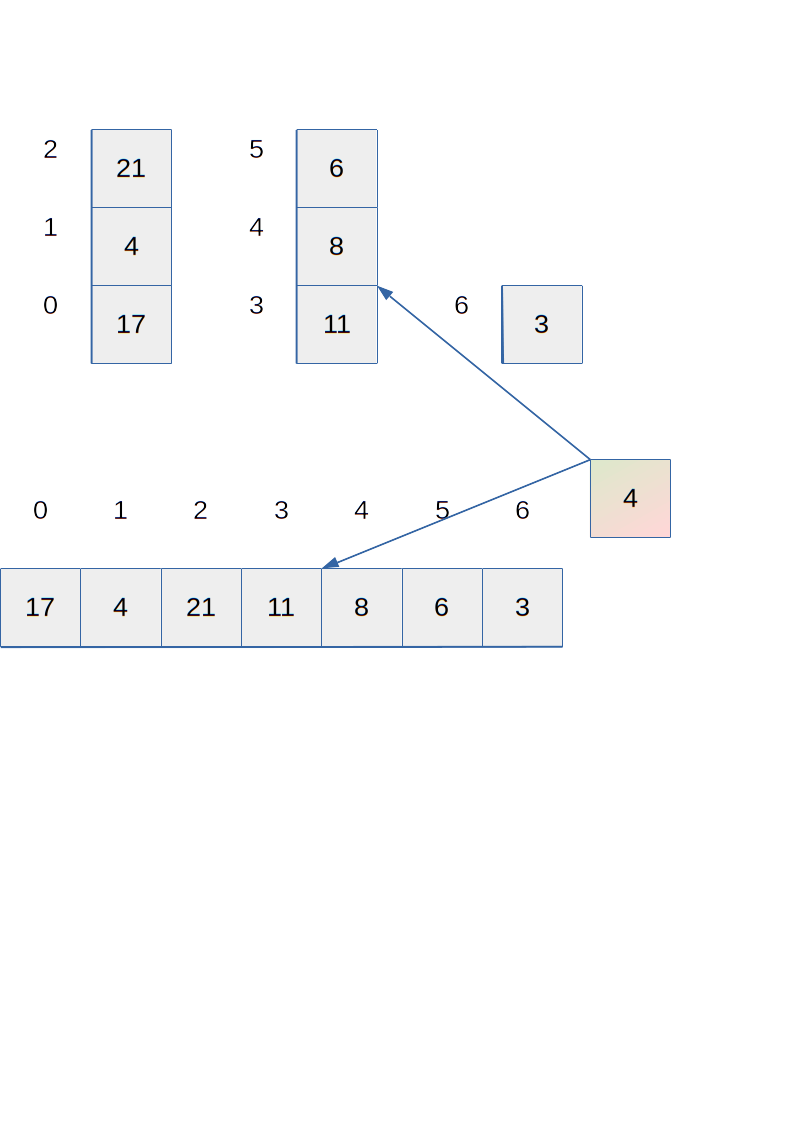
## Verifikáció

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, nagyobb indexérték felé. 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.

13. Ábra: ”4” érték beszúrása a 3. index után

A műveletsorok tartalmaztak beszúrást és törlést, érvényes indexekkel, valamint írás műveletet. A feladatok végesztével, a két adatstruktúra sorban kiolvasásra került és elemenként össze lett hasonlítva. A 12. Ábrán látható a vizsgált Gyorsított Tömb Példány, és a kontroll Vektor. Az Ábrán mindkettő ugyanazokat az értékeket tárolja.

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.

2. Táblázat: 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* |

3. Táblázat: 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, az Alsó Tömbökben, így nem érdemes önmagába ágyazni). Ilyen feladatkör 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, vagy, 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. Mivel a modern hibrid rendező algoritmusok gyakran váltanak át O(n²)-es rendező algoritmusokra, például Beszúró Rendezésre kis elemszámú rekurzív hívásnál, ezért ezeknél szintén életszerű, hogy előfordulhat gyorsulást tapasztaljunk a Gyorsított Tömb használata miatt.

State of the art

# Összefoglalás és További lehetőségek

Az eddigiekből látszik, hogy a Gyorsított Tömb képes lehet gyorsabb alternatívákat nyújtani bizonyos feladatokra, az ismert adatstruktúrákhoz képest, illetve kiinduló pontja lehet esetleges további adatstruktúráknak. Zárás képpen szeretnék megemlíteni néhány lehetséges, vagy éppen lehetetlen továbbfejlesztési módot, azok lehetőségeire, és problémáira kitérve. A Gyorsított Tömb fejlesztése során nem találtam további minden szempontból jobb fejlesztési lehetőségeket, viszont, felmerült néhány bizonyos szempontokból jobb továbfejlesztési mód, illetve néhány néhány további észrevétel.

## 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.
* 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, a HAT-hez hasonlóan. 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( óra ) Oldal

State of the arttal összehasonlítani

B-Fa(helyigény és lehetőséghez mérten) 0.2-1 0.2-1

egyéb segédfüggvények(search, subrotate) 2 2

továbbfejlesztés kifejtése ott ahol tradeoff van 2 2

felhasználhatóság után how to #include 1 1

elméleti jelentőség 1 1

források!

rb fa best case?

formalítások( szövegformázás, markup,stb.) 6 0!

ordo; ábra nevek!

edgecasetesztek visszacsinálása 3 0!

forrás:https://owlcation.com/stem/Which-rectangle-gives-the-biggest-area

<https://en.wikipedia.org/wiki/Mean_line_segment_length>/sources

Ábrajegyzék

Ábrajegyzék

[1. Ábra: Függvények összemérése 3](#__RefHeading___Toc1678_3221834476)

[2. Ábra: Vektor és Deque alapszerkezet 5](#__RefHeading___Toc1678_3221834476)

[3. Ábra: A Listák főbb típusai 7](#__RefHeading___Toc2987_1132801084)

[4. Ábra: Piros Fekete Fa 8](#__RefHeading___Toc2987_1132801084)

[5. Ábra: Egyszerű B fa 10](#__RefHeading___Toc1587_446001274)

[6. Ábra: Hasítótábla Szerkezete 12](#__RefHeading___Toc1587_446001274)

[7. Ábra: A Gyorsított Tömb felépítése 17](#__RefHeading___Toc1531_2065436672)

[8. Ábra: A Gyorsított Tömb Növekedése 18](#__RefHeading___Toc1531_2065436672)

[9. Ábra: Elemek indexei 21](#__RefHeading___Toc2991_1132801084)

[10. Ábra: Balance eltolás szemléltetése 24](#__RefHeading___Toc2991_1132801084)

[11. Ábra: Kiegyensúlyozás beszúrás után 25](#__RefHeading___Toc2993_1132801084)

[12. Ábra: Beszúrás Menete 31](#__RefHeading___Toc2993_1132801084)

[13. Ábra: ”4” érték beszúrása a 3. index után 36](#__RefHeading___Toc3001_1132801084)

Táblázatjegyzék

1. Táblázat: Adatstruktúra Sebességek 14

2. Táblázat: 32bites floatok tárolása és annak sebessége 37

3. Táblázat: 8192byte-os elemek tárolása és annak sebessége 37





