
Pannon Egyetem
Műszaki Informatikai Kar
Rendszer- és Számítástudományi Tanszék
Programtervező informatikus BSc

SZAKDOLGOZAT

**Korszerű adatstruktúra tervezése és implementálása
gyakori adatszűveletek hatékony megvalósítására**

Márton Attila

Témavezető: Dr. Süle Zoltán, egyetemi docens

2023



PANNON EGYETEM

MŰSZAKI INFORMATIKAI KAR

Programtervező informatikus BSc szak

Veszprém, 2023. október 24.

SZAKDOLGOZAT TÉMAKIÍRÁS

Márton Attila

Programtervező informatikus BSc szakos hallgató részére

Korszerű adatstruktúra tervezése és implementálása gyakori adatműveletek hatékony megvalósítására

Témavezető: Dr. Süle Zoltán, egyetemi docens

A feladat leírása:

A szoftverek működésének egyik legfontosabb feladata, hogy adatokat kezeljenek és tartsanak nyilván. Ezen adatokat általában csoportosítva, speciális adatstruktúrákban tároljuk, hiszen a megvalósításhoz felhasznált alkalmazott adatszerkezetek alapvetően meghatározzák a szoftverek működését, különös tekintettel annak futási sebességére. Általában igaz, hogy a tárolt elemek számának növekedésével a kulcsfontosságú beszűrési és törlési műveletek idejei is nőnek, amelyek nagyban befolyásolják a szoftverek hatékonyságát.

A Jelölt feladata egy olyan újszerű adatszerkezet elkészítése, amely a meglévő hagyományos adatstruktúrákra építve tud gyors beszűrési és törlési műveleteket garantálni. Az adatszerkezet kidolgozásán túl a feladat részét képezi a megtervezett és implementált adatstruktúra tesztelése és az eredmények elemzése is.

Feladatkiírás:

- Végezzen szakirodalmi kutatást és dolgozza fel a témához kapcsolódó hazai és nemzetközi publikációkat, valamint az ismert adatszerkezeteket és technológiákat!
- Mutassa be a megtervezett adatstruktúrát és elemezze annak működését!
- Implementálja a kidolgozott adatstruktúrát, és tesztelje azt funkció és teljesítmény szempontjából!
- Elemezze a tesztelés eredményeit és mutassa be annak potenciális felhasználási területeit!

Dr. Süle Zoltán
egyetemi docens
témavezető, szakfelelős

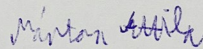
Hallgatói nyilatkozat

Alulírott Márton Attila hallgató kijelentem, hogy a dolgozatot a Pannon Egyetem Rendszer- és Számítástudományi Tanszékén készítettem a Programtervező Informatikus végzettség megszerzése érdekében.

Kijelentem, hogy a dolgozatban lévő érdemi rész saját munkám eredménye, az érdemi részen kívül csak a hivatkozott forrásokat (szakirodalom, eszközök stb.) használtam fel.

Tudomásul veszem, hogy a dolgozatban foglalt eredményeket a Pannon Egyetem, valamint a feladatot kiíró szervezeti egység saját céljaira szabadon felhasználhatja.

Dátum: Veszprém, 2023.10.17.



Márton Attila

Témavezetői nyilatkozat

Alulírott Dr. Süle Zoltán témavezető kijelentem, hogy a dolgozatot Márton Attila a Pannon Egyetem Rendszer- és Számítástudományi Tanszékén készítette programtervező informatikus végzettség megszerzése érdekében.

Kijelentem, hogy a dolgozat védeésre bocsátását engedélyezem.

Veszprém, 2023. december 4.



Dr. Süle Zoltán

Köszönetnyilvánítás

Szeretnék köszönetet mondani mindenkinek, aki a dolgozat elkészülése előtt vagy alatt hozzásegített.

Tartalmi Összefoglaló

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 program és adatstruktúra 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 legrosszabb esetben a teljes tömb méretével egyenes arányban növvő időben volt lehetséges.

Jelen munkámban, szeretném bemutatni, a Gyorsított Tömböt, amely a klasszikus tömbnek, illetve annak modernebb változatainak egy garantáltan gyorsabb mutációs idejű alternatívája lehet.

A Gyorsított Tömb, a hagyományos tömbökhöz hasonlóan konstans idejű index alapú elérést biztosít, de ezen felül képes n elem esetén garantáltan \sqrt{n} időben beszúrást és törlést megvalósítani. Mindehhez \sqrt{n} -es többlet memóriahasználat tartozik.

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

Kulcsszavak: Adatstruktúra, Tömb, Adattárolás, Adatbázis, Adatváltozás

Abstract

The storage of data is an essential part of modern computational systems. The storage of data usually happens in data structures, the speed of which, fundamentally effect the speed of a software or system of softwares. Modern computer science offers several tools, complex datastructures, among which one can find a fitting solution for almost any problem. On top of direct usage, data structures may be used for implementing more complex data structures.

In this work, I would like to show a new alternative, in a subgroup of data structures that is an essential part of almost all complex program and data structure, yet only a small fraction of it's members is used. The arrays, or more generally, data structures with constant time index based access is present in almost all application despite the speed of insertion and deletion in these being proportional to the total size.

In this work, I would like to introduce the Accelerated Array, which similarly to classical Arrays, is capable of index based access of elements in constant time, but also implements, insertion and deletion at a guaranteed speed of the square root of the total size.

In this work, I would like to show the implementation, and the proof of the aforementioned capabilities, and compare it to the currently existing alternatives.

Keywords: Data structure, Array, Data storage, Database, Data change

Tartalomjegyzék

1 Jelenlegi állapotok.....	11
1.1 Statikus és Dinamikus Tömbök.....	12
1.2 Hagyományos és Unrolled Listák.....	14
1.3 Piros-Fekete-Fa.....	15
1.4 B fák.....	17
1.5 Hasító táblák.....	19
1.6 HAT.....	20
1.7 Összehasonlítás (Táblaelemzéssel).....	21
2 Gyorsított Tömb.....	24
2.1 Alapötlet és Szerkezet.....	24
2.2 Leírás és elemzés.....	25
2.2.1 Létrehozás.....	27
2.2.2 Megsemmisítés.....	27
2.2.3 GetRelPos.....	28
2.2.4 BalanceShift:.....	31
2.2.5 További segédfüggvények.....	35
2.2.6 Mutáció.....	37
2.2.7 Elérés.....	40
2.2.8 Keresés.....	41
2.2.9 Forgatás.....	42
3 Implementáció, mérések.....	43
3.1 Implementáció.....	43
3.2 Verifikáció.....	43
3.3 Mérés.....	45
3.4 Elemzés, felhasználhatóság.....	47
3.5 Implementáció felhasználása.....	48
4 Összefoglalás és További lehetőségek.....	49
4.1 megjegyzések.....	49

Jelölések

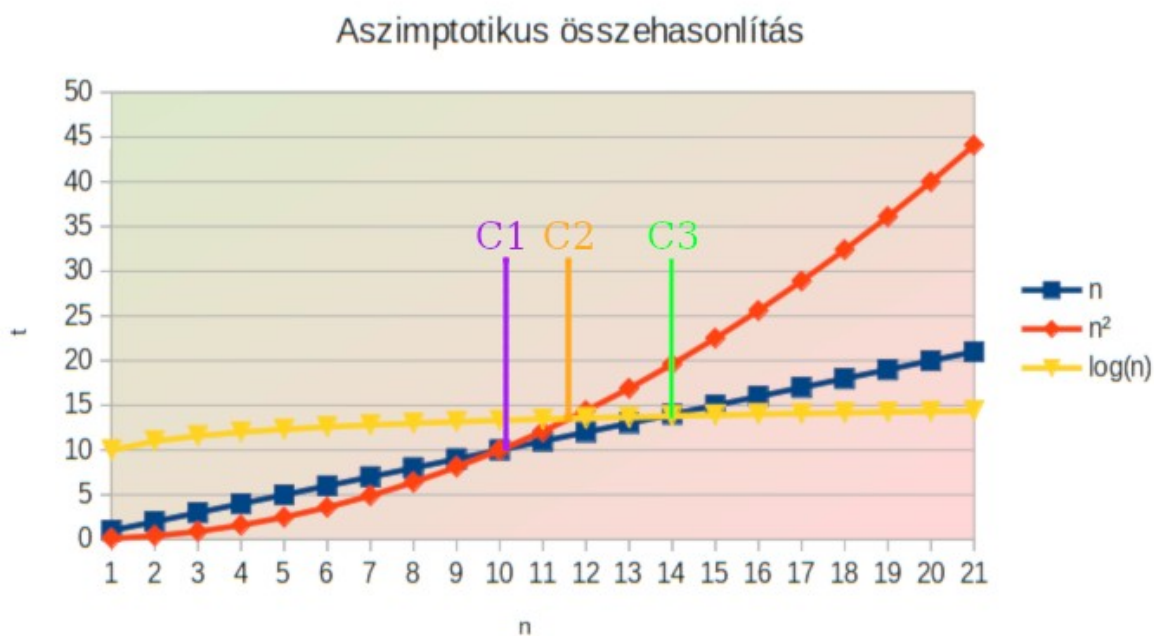
Access / Elérés: Read and write (Írás és olvasás)

Mutation / Mutáció: Insertion and deletion (Beszúrás és törlés)

1 Jelenlegi állapotok

Jelenleg számos Adatstruktúra ismert, melyek különböző előnyöket és hátrányokat hordoznak. A megválasztásukhoz bármilyen feladatra és összehasonlításukhoz, 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 jelölési módszert amely képes a végtelenbe tartó függvények között valamilyen relációt, hierarchiát felállítani.



1. Ábra: Függvények összemérése

Egy vizsgált f függvényt akkor nevezhetünk nagyobbknak egy másik g függvényénél végtelenben, ha f tetszőleges pozitív c konstans együtthatója mellett is, n változó végtelenbe tartása során létezik n_0 küszöbindex amelytől $n > n_0$ esetén $c * f(n) > g(n)$. Ekkor f felülről

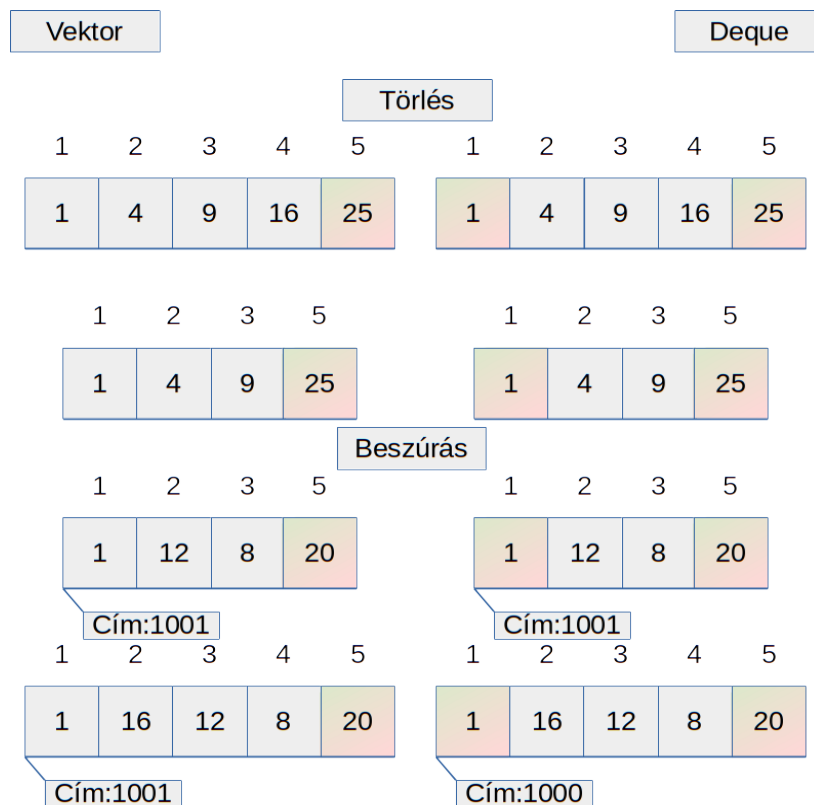
becsüli g -t. Ezt úgy jelöljük, hogy $g = O(f)$. Ez a nagy O jelölés.[1] Előfordulhat az is, hogy n változó végtelenbe tartása során létezik n_0 küszöbindex amelytől $n > n_0$ esetén $c * f(n) < g(n)$. Ekkor f alulról becsüli g -t. Ezt $g = \Omega(f)$ -el jelöljük. Ha mindkettő fennáll, $g = \Theta(f)$.

Ennek megfelelően, az ábrán A kvadratikus függvény felülől becsüli a másik kettőt, a lineárist $C1$ -től, a logaritmikust $C2$ -től. Ennek megfelelően, azok alulról becsülik a kvadratikus függvényt ugyanezeketől a küszöbértékektől. A lineáris függvény $C3$ -tól felülől becsüli a logaritmikust. Ennek megfelelően, a logaritmusos függvény $C3$ -tól alulról becsüli a lineárist. Ezen felül minden függvény alulról és felülől is becsüli önmagát, mivel egynél nagyobb konstans szorzóval minden elem nagyobb, egynél kisebb konstans szorzóval pedig minden elem kisebb mint az eredeti érték. A következő fejezet alapját, amely a már létező technológiákat foglalja össze, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, és Clifford Stein Új Algoritmusok című könyve alapozta meg.[2]

1.1 Statikus és Dinamikus Tömbök

Jelenleg a szekvenciális tárolók között a Statikus Tömb, és annak dinamikus változatai (Vektor és Deque) közel egyeduralkodóknak számítanak. A Statikus Tömb egy rendkívül 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 címezés 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

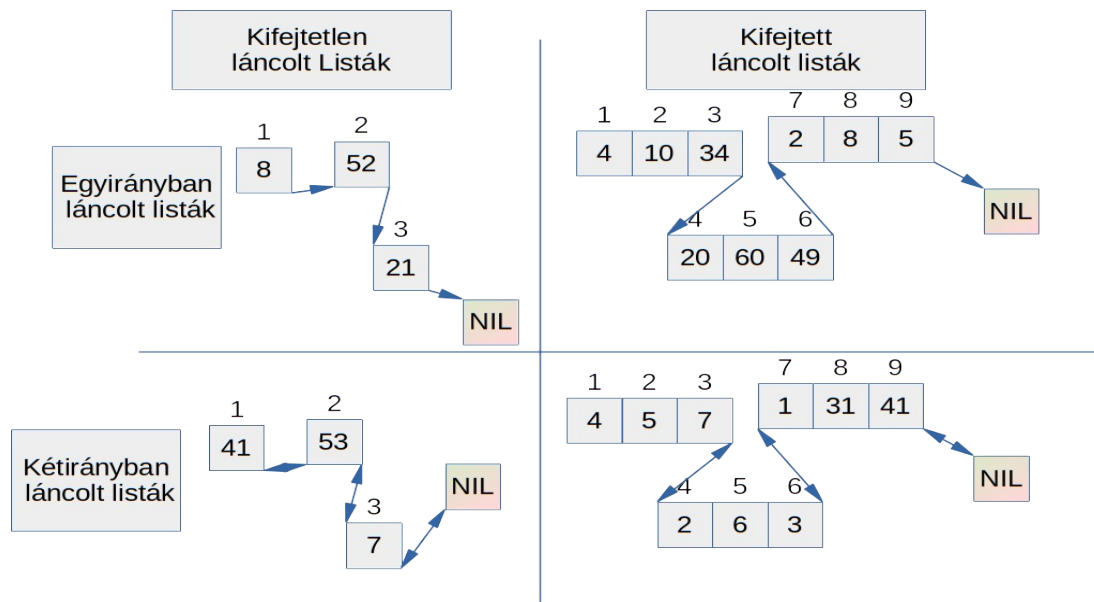
A 2. Ábrán látható a növekedés és csökkenés fizikai helye Vektorok és Dequek esetében. Az index alapú elérés legrosszabb esetben is $O(1)$, míg a mutáció legrosszabb és átlagos esetben $\Theta(n)$. A Deque legrosszabb esetben az elemek felét kell csak, hogy elmozdítsák, míg a vektor legrosszabb esetben (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.[3]

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.

1.2 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 pointerok 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 szűrni, 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 3. Ábrán a Láncolás iránya és kifejtettség szerinti lehetőségek látszanak, nem körkörös, nullpointer által befejezett példákon keresztül.



3. Ábra: Láncolt lista típusok

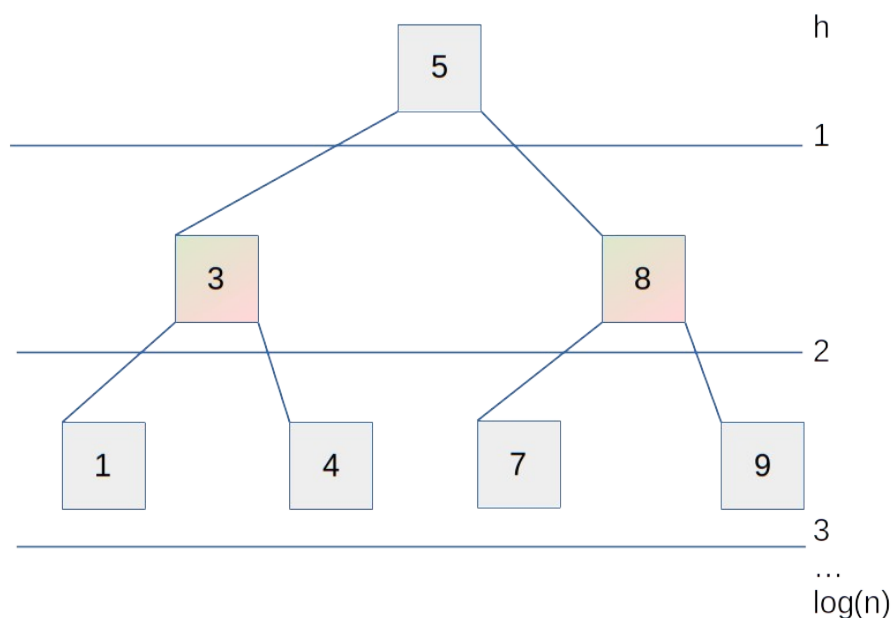
A láncolt listák egy fontos felhasználási módja, hogy fákat lehet velük implementálni. Ez úgy történik, hogy az adattag maga is lehet egy láncolt lista. A láncolt listákat általában közvetlenül olyan feladatok ellátására szánják, ahol a sorrendi, vagy esetleg fordított bejárás valószínű, és gyakori a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

1.3 Piros-Fekete-Fa

A Piros-Fekete fa egy közel kiegyensúlyozott bináris keresőfa. Hogy a Piros-fekete Fát vizsgálhassuk, először ennek a jelentését kell megismernünk. Itt „fa”-ként a gyökeres fákra fogok hivatkozni. 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 fokszámnak. 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 a fokú kiegyensúlyozott keresőfában, egy adott keresett érték megkeresése $O(\log_a(n))$ idejű, mivel $\log_a(n)$ a maximális távolság, a gyökértől. Egy elem keresésénél, a gyökértől indulunk, és megvizsgáljuk, hogy a keresett elem nagyobb-e mint a pillanatnyilag vizsgált. Ha nagyobb, a jobb gyermek felé megyünk tovább, ha kisebb, a bal felé, ha egyenlő, megtaláltuk. Így végighaladunk a teljes fán. Amennyiben levélhez érünk, és ez nem egyezik meg a keresett elemmel, a keresett elem nincs a fában. 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 4. 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 távoli, mint a gyökérhez legközelebbi.

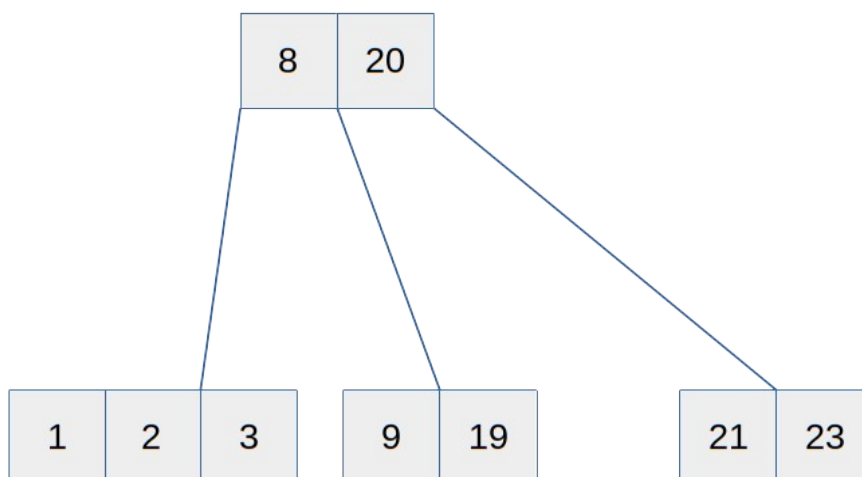
A beszúrás törlés és keresés műveletek a a Piros fekete fa kiegyensúlyozó műveleteivel együtt is $\Theta(\log_2 n)$ idejűek.

1.4 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 balra, 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 helyezkedik el. Lényegében minden művelet innen indul, a bináris és egyéb keresőfákhoz hasonlóan, viszont az egy csúcsban tárolt elemek számának növelésével, laposabbá tehető, és az $O(\log(n))$ 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-ről töltjük be az adatokat, mivel ekkor sokkal nagyobb memóriaegységek

kerülnek betöltésre mindenképpen, amit a B fa képes kihasználni. A B fák csúcsaiban a foksám nem konstans, hanem egy adott tartományban változik. Ez a tartomány általában egy fára jellemző maximális foksám, 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 foksámának határok között tartásával, és ezen határok megsértésének kezelésével oldja meg. Ha egy csúcs a kívánt beszúrás után is a maximális foksám 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 foksám 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 törölünk, amely kellően nagy ahhoz, hogy a belőle való törlés ne okozzon problémát. Ekkor egyszerűen eltávolítjuk az adott értéket és készen vagyunk. Azonban

előfordulhat az is, hogy a gyökérből, vagy egy belső csúcsból kell törölnünk. Ekkor a feladat bonyolultabb, de belátható, hogy a szükséges lépések sebessége logaritmus idejű. A keresés során, a Piros -Fekete Fákhoz hasonlóan járunk el, a gyökérből indulva, megkeresve azt a két kulcsot, amely közé a keresett elem esik, és továbbmegyünk az ezek közötti gyermek felé. Amennyiben megtaláljuk az egyik csúcsban a keresett értéket, készen vagyunk, ha egy levélre jutunk és ott sincs, akkor nincs a keresett elem a fában.

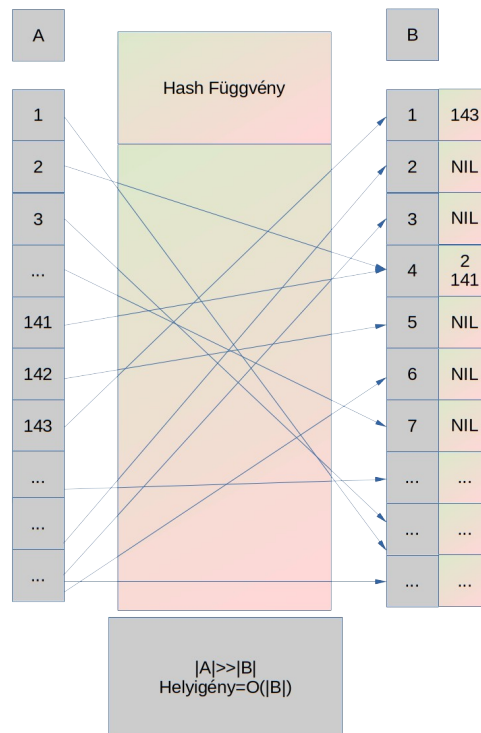
A B Fák alapértelmezett szerkezetén túl, előfordulnak további variánsok. Ilyen például a B+ fa. A B+ Fa abban tér el az alap B Fa szerkezettől, hogy a tárolt elemek kizárólag a levelekben helyezkednek el. Ekkor a magasabb szinteken, a levelekben tárolt adatok másolatai vannak, oly módon, hogy azok a bal legszélső gyermek legnagyobb elemei. Ezen felül a levelek unrolled láncolt listaként is össze vannak kötve, ezzel gyorsítva a szekvenciális elérést. Ezen kívül a B fának további változatai is ismertek, melyek a gyakorlatban valamilyen okból javítanak az eredeti struktúrán, e aszimptotikusan minden B fa variáns azonos sebességű.

1.5 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 szétszórásához 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: Hasító Tábla 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 6. 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.

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

1.6 HAT

A Hasított Tömbfa (HAT) közel \sqrt{n} darab \sqrt{n} hosszú elemet tartalmaz, de mutáció esetén az egész adatszerkezetet újraépíti lineáris időben.[4]

A szerkezete a Későbbiekben bemutatásra kerülő Gyorsított Tömbhöz hasonló. Az előnye a hagyományos tömbökhöz képest az, hogy a többlet memóriahasználata \sqrt{n} -es míg ugyanez a hagyományos Dinamikus Tömböknél lineáris. Egy Dictionaryből áll amely körülbelül \sqrt{n} 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 egészezen 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. Tehát 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 $\log_2(8)$, 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 többi 0-ra állítjuk. Tehát például $m = 3$ esetén a 8-al történő moduló megoldható tetszőleges 2-es számrendszerbeli számra (például 01010110_2) úgy, hogy a 3 legkisebb helyi értékű bitet meghagyjuk a többi pedig nullára állítjuk (a példában ez 00000110_2).

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

1.7 Összehasonlítás (Táblaelemzéssel)

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)	$O(1)$	$O(1)^*$	$O(n)^*$	$O(1)$	$O(n)$
Láncolt Lista	$O(1)$	$O(n)$	$O(n)^*$	$O(n)^*$	$O(n)$
Fa	$O(1)$	$O(\lg n)$	$O(\lg n)$	-	$O(\lg n)$
Hasítótábla	$O(1)$	$O(1)$	$O(1)^*$	-	$O(1)^*$
Gyorsított Tömb	$O(1)$	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(1)$	$O(n)$

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 kívül fontos tényező, hogy ezek a futási idők amortizált futási idők, viszont a hasítótáblánál a legrosszabb eset Mutációra és keresésre, $O(n)$. A HAT megsemmisítési ideje $\Theta(\sqrt{n})$, illetve néhány Dinamikus Tömb (Tiered Vektor) képes az $O(\sqrt{n})$ -es amortizált mutációs időre, még ha ezt nem is garantálja minden esetben.

Felmerülhet az is, hogy a Dinamikus Tömböket kereső adatstruktúráként használjuk, úgy, hogy az elemeket sorba rendezzük. Mivel ekkor adott elem megkeresése $O(\log_2 n)$ 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 művelet lineáris idejű, amíg ez a fáknál logaritmikus idejű volt.

Mindezekon túl, a különböző adatstruktúrák különböző problémák megoldására lettek kifejlesztve. Ennek megfelelően minden összehasonlításnál figyelembe kell venni, hogy milyen probléma megoldására keressük 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 komplexebb 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 különbséget teremtenek. Ilyen a B fák nagyobb gyakorlati sebessége a piros fekete fákhhoz képest, mivel a B fák egy memóriaelérés során több értékes adatot kérnek be, mivel általában a kulcsok egymás után találhatók 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 eléri, a B fák ezen előnye megszűnik.

2 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 Gyorsított 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 szétördelve több kisebb Dinamikus Tömb között, amelyeknél a közel azonos hossz biztosított.

2.1 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 változóknak megtartani a sebesség vagy olvashatóság érdekében.

- Felső Tömb

A Felső Tömb tárolja az Alsó Tömbök címeit.

- Alsó Tömbök

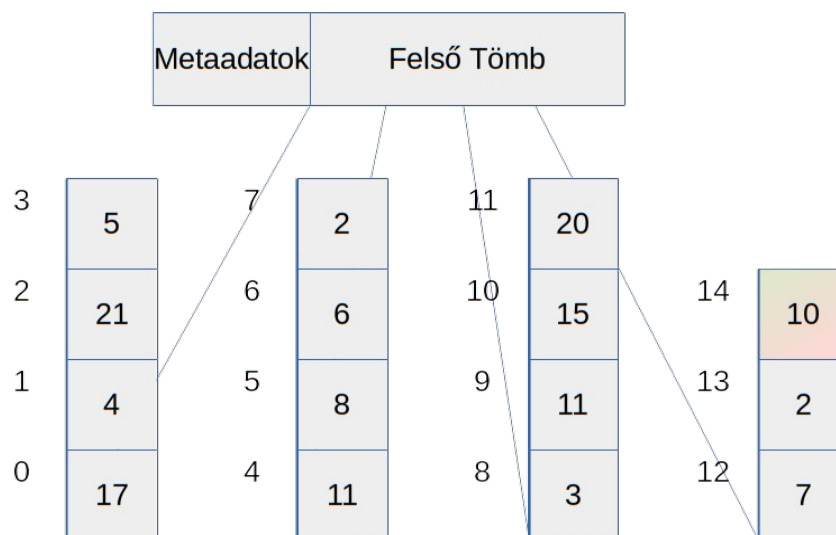
Az Alsó Tömbök tárolják az adatokat, Ezek Deque-k kell hogy legyenek, vagy egy másik olyan lineáris index alapú adatstruktúra, amely konstans időben valósítja meg a végeken a beszúrás és törlés műveleteket.

A Gyorsított Tömb, az adattagok beszúrásnál szükséges nagy mennyiségű adatmozgatást úgy kerüli el, hogy az adatokat nem egy darab nagy méretű vektorban, n elem esetén, \sqrt{n} , vagy $\sqrt{n+1}$ darab kisebb vektorban tárolja, amelyeknek a hossza is \sqrt{n} , vagy $\sqrt{n+1}$.

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 megegyezik az Alsó Tömbök számával.



7. Ábra: Gyorsított Tömb felépítése

2.2 Leírás és elemzés

Amikor az Alsó Tömbök a közepébe beszúrás történik, az Alsó Tömb hosszával arányos adatmozgatás, csak \sqrt{n} darab adat mozgatását teszi szükségessé. A teljes tároló növekedése úgy történik, hogy először egy \sqrt{n} -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.

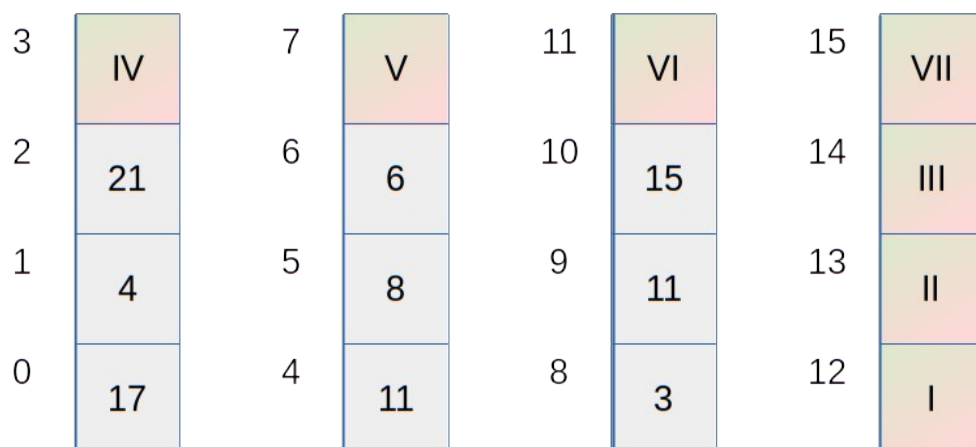
Ebből következik, hogy egy Alsó Tömb végéről levett, és az azt követő Alsó Tömb elejére helyezett elem indexe nem változik. Ugyanez az ellenkező irányban is működik.

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 \sqrt{n} méretűek maradjanak, szükséges, ezeket kiegyensúlyozni. A kiegyensúlyozásnál, a többletet tartalmazó Alsó Tömb irányából a következő növekedés helye felé történik.



8. Ábra: Gyorsított Tömb növekedése

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 8. 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 8. 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.

2.2.1 Létrehozás

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

A létrehozás üresen:

- I A Felső Tömb létrehozása
- II a legelső Alsó Tömb lefoglalása
- III 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 memóriefoglalás(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.

$\Theta(1)$

2.2.2 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, $\sqrt{n} + 2$ törlés összesen, legrosszabb esetben. Leggyorsabb esetben csak 1 Alsó Tömb van, így ekkor 2 törlésre van szükség.

$$\Theta = \sqrt{n}$$

2.2.3 GetRelPos

Ahhoz, hogy bármely indexnél műveleteket 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 indexszel 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 a keresett i indexnek a hányadosa megadja a keresett Alsó Tömböt, a moduló pedig a belső indexet:

$$(i / \sqrt{n}, \sqrt{n} \mid i).$$

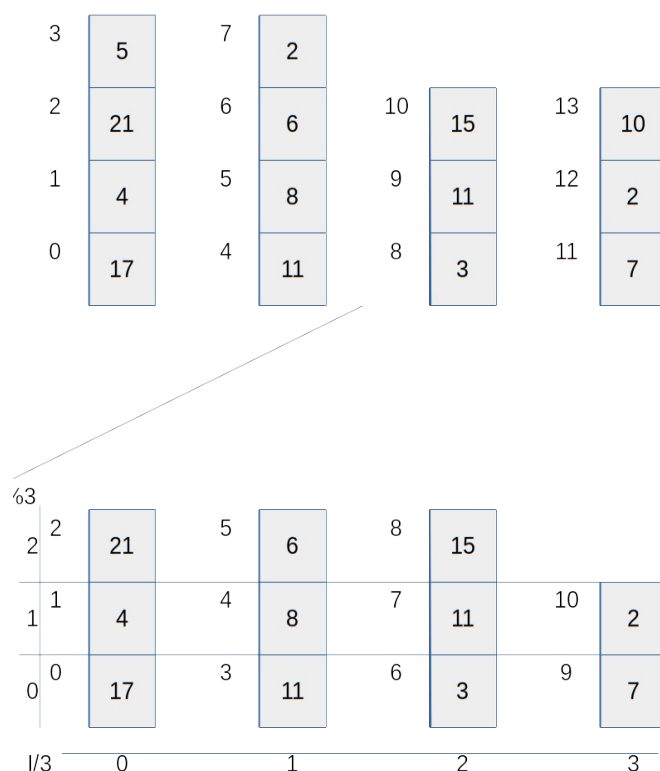
- Ha betelt az Újtömb, akkor az Alsó Tömbök mérete sorban 1-el növelésre kerül. Először meg kell állapítani hogy a keresett index az így növelt tartományba esik-e.

Ha igen, akkor ezen belül a II. Pont szerint járunk el, $\sqrt{n+1}$ 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.

A 9. 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 kezelhetők, 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 $\sqrt{n+1}$ -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 $\sqrt{n+1}$ -edik tagjával, a következő négyzet alakú mag elérésre kerül.



9. Ábra: Elemek indexei

GetRelPos(index):

```

1    if popsqr = 0
2        then return pair(0, 0)
3    if popextra <= popsqr
4        then return divmod(index, popsqr)
5    else
6        if (popextra - popsqr) > index / (popsqr + 1)
7            then return divmod(index, popsqr + 1)
8        else
9            uninced  $\leftarrow$  index - (popextra - popsqr) * (popsqr + 1)
10           uc  $\leftarrow$  divmod(uninced, popsqr)
11           return list(popextra - popsqr + uc.first, uc.second)

```

```

pair<int, int> getRelPos(int index) const {
    if (popsqr == 0){
        return pair<int, int>(0, 0);
    }
    if (popextra < popsqr){
        return divmod(index, popsqr);
    }
    else{
        if ((popextra - popsqr) > index / (popsqr + 1)) /{
            return divmod(index, popsqr + 1);
        }
        else{
            int uninced = index - (popextra - popsqr) * (popsqr + 1);
            auto uc = divmod(uninced, popsqr);
            return pair<int, int>(popextra - popsqr + 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 belüli indexet.

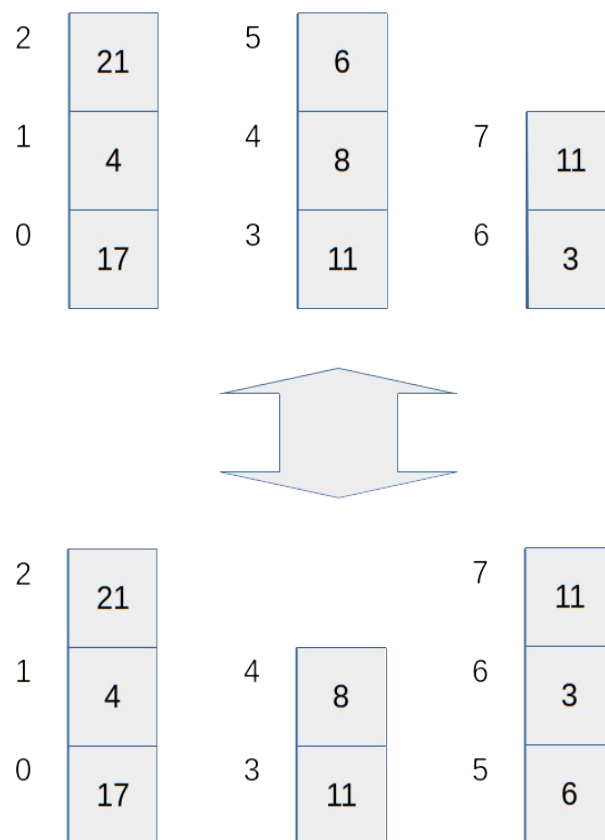
Ha igen, akkor több extra elem van tárolva, mint ami az Újtömbben elfér, ezek az Alsó Tömbök extra tagjaiként vannak tárolva és ennek megfelelően korrigálni kell az extra mérettel ami az első néhány Alsó Tömböt jellemzi. Ehhez szükségünk van az inkrementálatlan részbe eső, index által elért elemek számára. 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 ($\text{popsqrt} + 1$), szorozva a növelt Alsó Tömbök számával ($\text{popextra} - \text{popsqrt}$). 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 **$\Theta(1)$** .

2.2.4BalanceShift:

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 Deque 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.

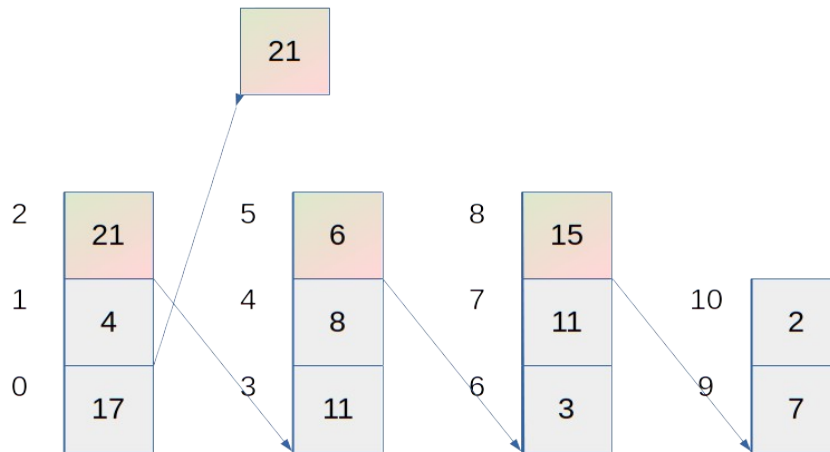


10. Ábra: Balance eltolás szemléltetése

A 10. 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 mozgathatáshoz 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.

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.



11. Ábra: Többszörös tárolás eltolása

A 11. 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 többszörös 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 Tömbből és a 2. Alsó Tömb elejére kerül. Ekkor a logikai helye nem változik, viszont a fizikai többszörös 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 kiindulási és a cél Tömb közötti összes Alsó Tömbön végigmegyünk, úgy, hogy mindegyikbe egyszer beszúrunk az elején 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 különbséggel, 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 indexszel.. Tehát a különbségük legfeljebb a Felső Tömb hossza. Ez $O(\sqrt{n})$, amin a gyengébb konstans + 1 tényező nem változtat. A ciklusmag pontosan 2 konstans idejű lépést végez. Mindezek alapján a sebesség $O(\sqrt{n} * 1) = O(\sqrt{n})$. A mean line segment alapján a várható sebesség $\Theta(\sqrt{n} / 3) = \Theta(\sqrt{n})$. Legjobb esetben nem lépünk be az iterációba, ekkor a futási idő konstans.

2.2.5 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 modulból 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);
```

```
}
```

Mivel minden lépés konstans idejű, a teljes függvény $\Theta(1)$.

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      popsqr ← floor(sqrt(population))
```

```
4      popextra ← population – popsqr * popsqr
```

```
5      return oldSQRT = popSQRT
```

```
bool incPop(){
```

```
population++;
```

```
auto oldsqr = popsqr;
```

```
popsqr = SQRT(population);
```

```
popextra = population - popsqr * popsqr;
```

```
return oldsqr != popsqr;
```

```
}
```

A **populációcsökkentés** frissíti a metaadatokat, törlés után. Ezen felül visszatér 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;
}
```

Mivel minden lépés konstans idejű, a teljes függvény $\Theta(1)$.

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);
    }
}
```

Mivel minden lépés konstans idejű, a teljes függvény $\Theta(1)$.

A calcDeletePlace a törlés során, a helyes méretcsökkenés helyét adja vissza. Ezzel a függvénnyel kérhető le, hogy egy törölt 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);
    }
}

```

Mivel minden lépés konstans idejű, a teljes függvény $\Theta(1)$.

2.2.6 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 beszúráss az Alsó Tömbben, majd a Balanscshift 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() )

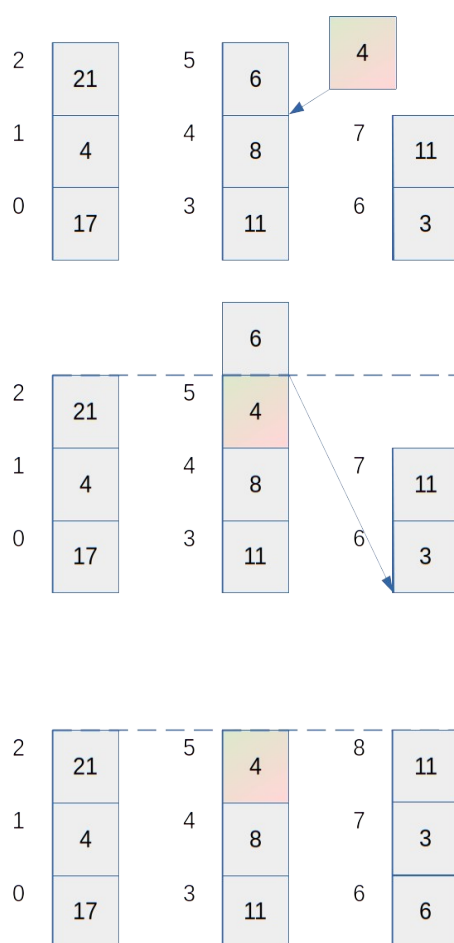
```

```

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 12. 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 jelentkezik 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. Mindezek 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 döntenek 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 \sqrt{n} időben fut le, mivel a Deque közel \sqrt{n} hosszú (vagy pontosan annyi, vagy $\sqrt{n} + 1$). Legjobb esetben a végére szúrunk be aminek köszönhetően konstans lehet a beszúrási idő. A BalanceShift legjobb esetben szükségtelen, de erre csak $1:\sqrt{n}$ -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 a hossz / 3[5], ami itt $\sqrt{n} / 3$, másik esetben átlagos távolság végponttól, ami hossz / 2, jelen esetben $\sqrt{n} / 2$, így átlagosan 5 / 12-ed \sqrt{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.[6]

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

$\Theta(\sqrt{n})$.

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->pop_back();
}
```

Mivel nincs se iteráció, se rekurzió, a hívott függvények döntenek 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 \sqrt{n} -es. Mindezek alapján a Törlés során a futási idő itt is, Legjobb esetben Konstans, legrosszabb és átlagos esetben $\Theta(\sqrt{n})$.

2.2.7 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 indexszel. A visszatérési érték első eleme a keresett alsó tömb indexe, a második érték az ezen belüli index. Ez után ezt az elemet elérjük, és ott elvégezzük az írás műveletét.

Írás(*index*, *érték*):

- 1 *tarvector* \leftarrow first(getRelPos(*index*))
- 2 *hely* \leftarrow 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 konstans idejű. $\Theta(1)$.

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* \leftarrow first(getRelPos(*index*))
- 2 *hely* \leftarrow 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 $\Theta(1)$.

Az elérés műveleteket implementáló kódok az alábbi referencialekérő függvényre hivatkoznak, amely konstans idejű részműveletek szekvenciájaként, szintén $\Theta(1)$:

```
T &refAt(int index){  
    auto pos = metaData.getRelPos(index);  
    return ((*content)[pos.first])->access(pos.second);  
}
```

2.2.8 Keresés

Annak ellenére, hogy a Gyorsított Tömb nem Keresőadatstruktúrának lett szánva, mégis felmerülhet, hogy elemeket akarunk benne keresni. Ez például akkor fordulhat elő, ha B Fák Csúcsainak tartalmát tároljuk benne. Ebben az esetben, a jelenleg erre a célra használt Dinamikus Tömbökhöz hasonlóan, a tartalmazott elemeket érdemes sorba rendezni. Itt fontos megemlíteni, hogy főleg kis elemszámnál a Beszűrő Rendezés ígéretes lehet, mivel ez azon kevés rendezőalgoritmusok egyike, amely alkalmazza a beszűrítés és törlés műveletet, ez által a Vektorokon és Dequeken tapasztaltnál gyorsabb. Ez után a sorba rendezett elemeken bináris keresést lehet végrehajtani.[7] Ez $O(\log_2 n)$ futási idővel bír. Az itt bemutatott pszeudokódok pusztán az értékkel, vagy logikai hamissal térnek vissza.

Keres_rendezett(érték):

```
1    alsó_korlát ← 0  
2    felső_korlát ← population - 1  
3    while alsó_korlát < felső_korlát  
4        do közép ← floor((alsó_korlát + felső_korlát) / 2)  
5        if érték = Olvas(közép)  
6            then return közép  
7        else if érték < Olvas(közép)  
8            then felső_korlát ← közép - 1  
9        else  
10           then alsó_korlát ← közép + 1
```

Felmerülhet az is, hogy sorba rendezetlen elemeken akarunk keresést végrehajtani. Ekkor egyszerűen nekiállunk végigmenni az összes elemen, és a keresést befejezzük, amikor a keresett elemet megtaláljuk. Mivel legrosszabb esetben minden elemet meg kell vizsgálnunk, ahhoz, hogy lássuk, hogy a keresett elem nincs az adatstruktúrában, ezért ennek a futási ideje $O(n)$.

Keres_rendezetlen(*érték*):

```
1      for  $i \leftarrow 0$  to  $population - 1$ 
2          if Olvasás( $i$ ) = érték
3              then return érték
4          else
5              then return false
```

2.2.9 Forgatás

A Beszúró rendezés egy beszúrás és egy törlés egymás utáni végrehajtásából áll. Ez megfelel, egy Tömb adott indexek közötti résztömbjének a forgatásának. Ahhoz, hogy hatékony Beszúró rendezést végezhessünk, érdemes, a Törlés és Beszúrás egymás utáni naiv elvégzése helyett, egy összevont algoritmust alkalmazni, amely a kiegyensúlyozást a beszúrás és a törlés Alsó Tömbje között hajtja végre. Ekkor a műveletnek nincs szüksége másra, mint a 2 két indexre, ahonnan a kívánt érték törlésre kerül, és utána, ahova beszúrásra kerül. Ez a két index lényegében a legelső és a legutolsó eleme a forgatott résztömbnek. Az itt implementált algoritmus egyszeres forgatást hajt végre, és a második változóként megadott indexről töröl, és az első elé szúr be.

Forgatás(*első*, *utolsó*):

```
1      beszúráshely  $\leftarrow$  getRelPos(első)
2      törléshely  $\leftarrow$  getRelPos(utolsó)
3      érték  $\leftarrow$  Olvasás(utolsó)
4      erase(felső_tömb[first(törléshely)], felső_tömb[second(törléshely)])
5
6      insert(felső_tömb[first(beszúráshely)], felső_tömb[second(beszúráshely)], temp)
7      balanceShift(first(beszúráshely), first(törléshely))
```

Mivel ez a függvény konstans és $\Theta(\sqrt{n})$ idejű függvények szekvenciája, ezért maga is $\Theta(\sqrt{n})$.

3 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::vector`ával szemben. A méréseket ezután elemeztem. Fontos kiemelni, hogy az olvashatóság kedvéért, semmilyen komoly optimalizációt nem végeztem, a pszeudokódhoz 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.

3.1 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.

3.2 Verifikáció

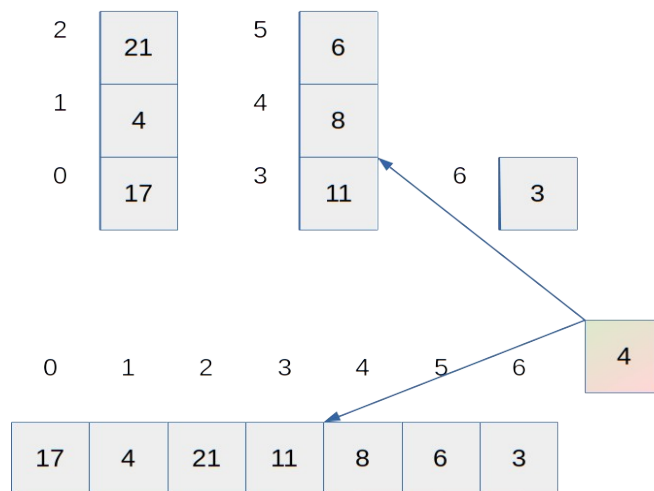
A tesztelés elsődleges célja az volt, hogy meggyőződjek róla, hogy az elkészített implementáció ténylegesen a feladatát látja el. Ezt verifikációnak nevezzük. A verifikációra

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 Dinamikus 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 indexű elemek mind egy indexszel 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 ellenőrzés során, fontos tulajdonság volt több függvénnnyel 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 ellenőrzésének kidolgozása során, előfeltételeztem, hogy a C++ beépített `std::vektora` 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” ellenőrzésére.



13. Ábra: 4 beszúrása a 3. index után

A műveletsorok tartalmaztak beszúrást és törlést, érvényes indexekkel, valamint írás műveletet. A feladatok végeztével, a két adatstruktúra sorban kiolvasásra került és elemenként össze lett hasonlítva. A 13. 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.

3.3 Mérés

A mérések elvégzéséhez a C++ programozási nyelvben 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: 32bit-es 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 műveletekhez képest.

3.4 Elemzés, felhasználhatóság

A Gyorsított Tömb még további optimalizáció nélkül is felhasználható közvetlenül ott, ahol eddig a Vektorok és Deque 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. A Gyorsított Tömb nem tud sebességelőnyt jelenteni a Statikus Tömbhöz képest ott, ahol nem történik mutáció.

Közvetve felhasználható B fák belső tárolóiként, különösen a levelekben, ahol a mutációk aránya a lehető legmagasabb az elérésekhez képest. Amennyiben a csúcsok maximális fokszáma c , úgy a csúcsban történő mutáció Dinamikus Tömbök esetén $\Theta(c)$, míg ugyanez $\Theta(\sqrt{c})$. Ez $c / \sqrt{c} = \sqrt{c}$ időigényű, vagyis \sqrt{c} mértékű gyorsulást eredményez.

Ezekén túl, algoritmusokat is képes gyorsítani, amennyiben azok eddig a Vektorba, Dequebe szűrés által kerültek lassításra. Ilyen a Beszúró Rendezés, amely n darab átlagosan $n/2$ méretű mutációt hajt végre, így $O(n^2)$ -es futási idővel bír, ezeket az adatstruktúrákat használva. Ez javítható $O(\sqrt{n} * n)$ -re Gyorsított Tömb használatával. A meglévő 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 $O(n^2)$ -es rendező algoritmusokra, például Beszúró Rendezésre kis elemszámú rekurzív hívásnál, ezért ezeknél szintén életszerű, hogy előfordulhat gyorsulást tapasztaljunk a Gyorsított Tömb használata miatt.

3.5 Implementáció felhasználása

Annak ellenére, hogy az implementáció főként a verifikáció és a validáció miatt született, előfordulhat, hogy mégis valaki fel kívánja használni. Ebben az esetben szükséges a forráskódot tartalmazó fájlok importálása a projektbe. Ez után kitöltött sablonként lehet változók deklarálásánál hivatkozni rá. A C++ Standardban található Dinamikus Tömbökhöz (`std::vector`, `std::deque`) közel álló szintaxis miatt viszonylag egyszerű ezek lecserélése. Ehhez egyrészt a konkrét típus megnevezéseket kell lecserélni, másrészt az egymással analóg, vagy közel analóg függvényeket. Ilyen például az iterátorok általi elem elérés lecserélése, közvetlen index alapú elérésre. A szögletes zárójel általi indexelés a Gyorsított Tömbnél implementálásra került.

```
#include <vector>

vector<int> v;           // üres
v.insert(v.begin(), 0);  // 0
v.insert(v.begin() + 1, 3); // 0 3
v[1] = 2;               // 0 2
v.erase(v.begin() + 0);  // 2
cout << v.at(0) << endl;

#include "accarr.h"

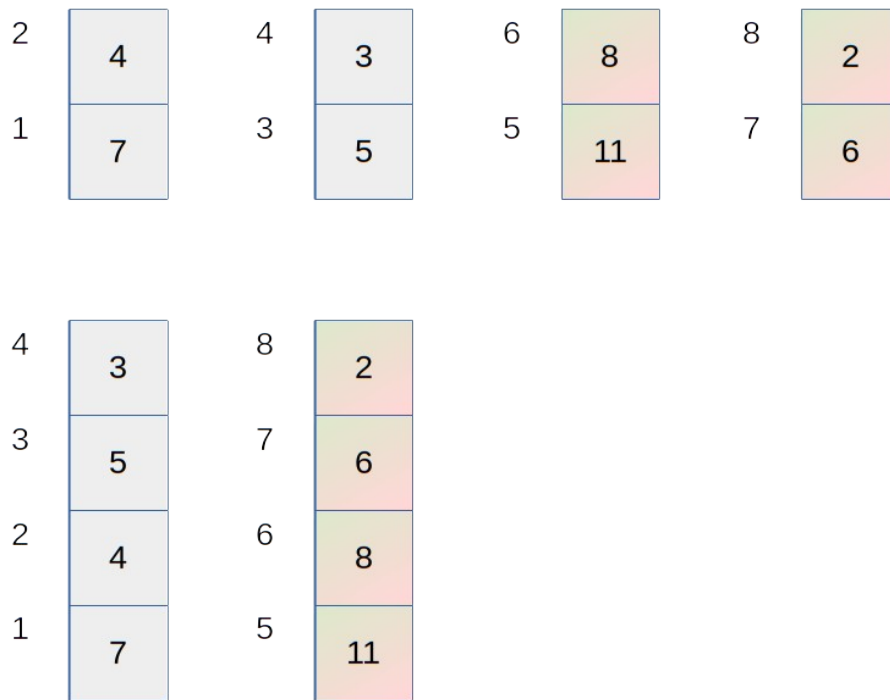
accarr<int> a;           // üres
a.insert(0, 0);          // 0
a.insert(1, 3);          // 0 3
a[1] = 2;               // 0 2
a.erase(0);             // 2
cout << a.refAt(0) << endl;
```


4 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 képen 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 továbbfejlesztési mód, illetve néhány további észrevétel.

4.1 megjegyzések

- A modern processzorok prefetch és cache képességeik miatt, az 1-szeres indirekció nem okoz számottevő lassulást.
- Mivel a belső deque végén a mutáció, konstans idejű, viszont a Gyorsított Tömbnél ez az esetek felében nem áll fenn, így az önmagába ágyazás, nem jár gyorsulással.
- A naiv implementáción túl, felmerülhet 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 helyettesíthető bitstift és bitmaszk műveletekkel, ezzel jelentősen csökkentve az index elérés konstans együtthatóját, a HAT-hez hasonlóan. Sajnos ebben az esetben a mutáció legrosszabb esete $O(n)$, mivel legalább az elemek felét mozgatni kell, a 14. Ábrán látható módon legeslegrosszabb esetben az összeset, nagyobb újrafoglalt Dequebe. Ez nagy méretű adathalmazok esetén nagy pillanatnyi többletterhelést jelentene. Legrosszabb esetben több ilyen többletterhelés egybeesése megakaszthatja egy adatbázis működését. Az amortizált eset továbbra is $O(\sqrt{n})$, az elérési műveletek konstans ideje, pedig a gyakorlatban jelentősen javul, a lassú moduló és egész osztás műveletek bitműveletekkel való leváltásának köszönhetően. Az itt leírt adatstruktúra, a Tiered Vektor.



14. Ábra: Lineáris idejű átméretezés

- Érdekes lehet az új, növekvő vektort középre helyezni a felső vektorban, ezzel megfeleezni 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 $\sqrt{n} / 2$, ha a közepe felé,akkor $\sqrt{n} / 4$. 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, a dequeknél szükséges 2 helyett. Ez tovább lassítaná az elérésnél, a helyes index kiszámítását.
- GetRelPos nagyban gyorsítható polimorfizmussal vagy függvény pointerekkel, amiket méretváltoztatáskor változtatunk, egyébként csak meghívunk, ezzel konstans elérési idő együttthatóját tovább lehet csökkenteni.

Irodalomjegyzék

- [1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, és Clifford Stein, Új Algoritmusok, Sclar Kiadó, 1999.
- [2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, és Clifford Stein, Új Algoritmusok, Sclar Kiadó, 1990.
- [3] Michael T. Goodrich, Roberto Tamassia, Algorithm Design, Wiley, 1999.
- [4] Edward Sitarz, <https://www.drdoobs.com/database/algorithm-alley/184409965?pgno=5> [2023.12.04.], 1996
- [5] forrás: <https://www.cambridge.org/core/journals/bulletin-of-the-australian-mathematical-society/article/average-distance-between-two-points/F182A617B5EC6DB5AD31042A4BDF83AE> [2023.12.04.], 2009
- [6] forrás: <https://owlcation.com/stem/Which-rectangle-gives-the-biggest-area> [2023.12.04.], 2022
- [7] Michael T. Goodrich, Roberto Tamassia, Algorithm Design, 1999.

Mellékletek

accarr.h

accarr_innerData.h

comparee.h

main.cpp

speedcomparison.h

SQ.h

tests.h

mérés.xls

Ábrajegyzék

1. Ábra: Függvények összemérése.....	11
2. Ábra: Vektor és Deque.....	13
3. Ábra: Láncolt lista típusok.....	15
4. Ábra: Piros fekete fa.....	16
5. Ábra: Egyszerű B fa.....	18
6. Ábra: Hasító Tábla szerkezete.....	20
7. Ábra: Gyorsított Tömb felépítése.....	25
8. Ábra: Gyorsított Tömb növekedése.....	26
9. Ábra: Elemek indexei.....	29
10. Ábra: Balance eltolás szemléltetése.....	32
11. Ábra: Többlet távolra eltolása.....	33
12. Ábra: Beszúrás menete.....	38
13. Ábra: 4 beszúrása a 3. index után.....	45
14. Ábra: Lineáris idejű átméretezés.....	50

Táblázatjegyzék

1. Táblázat: Adatstruktúra Sebességek.....	22
2. Táblázat: 32bites floatok tárolása és annak sebessége.....	46
3. Táblázat: 8192byte-os elemek tárolása és annak sebessége.....	46