S. 64.
Adrián Patrik, 12. évf.
Baross Gábor Középiskola, Szakiskola és Kollégium, Debrecen

Könnyű dolgunk lenne, ha a feladat megoldásához egyszerűen szimulálhatnánk a tömbből a leírt módon történő törlést. Habár maga a tömbelem eltávolítása $O(1)$ időben megoldható, az azt követők balra csúsztatása már $O(N)$ időt igényel. Mivel a törlési műveletek száma K, ezért a program futása $K \cdot O(N)$ ideig tart, ami $K = \Theta(N)$-nel számolva $O(N^2)$-et ad; ez $N = 1\ 000\ 000$ esetén rengeteg műveletet jelent. A legrosszabb esettel akkor van dolgunk, ha mindig az első elemet kell törölnünk, összesen N-szer. Ez N darab törlést és $\frac{N(N-1)}{2}$ mozgatást jelent, ami a maximális esetben $5 \cdot 10^{11}$ nagyságrendű. Ennyi időnk nincs.

Vagy egy tömbökön működő ügyes algoritmust kell találnunk, amely elő tudja állítani a törlendő pozícióból az adott elem eredeti pozícióját, így elkerülve a rengeteg mozgatást, vagy elvetjük a tömbök használatának ötletét és más, bonyolultabb adatszerkezetek után kutatunk, melyekből a sorrend megtartásával elvégezhető az elemek törlése lényegesen kevesebb, mint $O(n)$ idő alatt. Az utóbbi utat választjuk.

A tömbökkel szemben másik végletet alkotó láncolt listák sem jelentenek megoldást, mert igaz, hogy az elemek eltávolítása konstans idejű, megkeresésük azonban hosszadalmas. Úgy kell választanunk, hogy mind a helymeghatározás (keresés), mind a törlés gyorsan – ha nem is konstans időben – menjenek.

Itt eszünkbe juthatnak a bináris keresőfák, melyek magassága szerencsés esetben az elemek számának logaritmikus függvénye és a keresés és a törlés is megvalósítható bennük a fa magasságával arányos, azaz $O(\log n)$ időben. Használhatunk AVL vagy piros-fekete fákat is, de mint azt mindjárt látni fogjuk, egyikre sincs szükség.

A fa csúcsainak kulcsaként az eredeti tömb indexeit használjuk, de mivel ezek explicit tárolására sem a fa megépítéséhez, sem az elemek törléséhez nincs szükség, ezért tulajdonképpen az adatszerkezet már csak mélyen az elméletében lesz keresőfa. (Sem elemek beszúrására, sem érték szerinti keresésre nincs szükségünk, ezért megengedhetjük a kulcsok elhagyását.) Végül is egy olyan fát építünk, melynek minden csúcsára igaz, hogy bal (jobb) részfájában csak olyan elemek vannak, amelyek az eredeti tömbben előbb (később) szerepeltek.

A fa építése rekurzívan történik, egy tökéletesen kiegyensúlyozott bináris fát létrehozva (minden csúcs bal és jobb részfájában szereplő elemek számának különbsége legfeljebb 1). A teljes tömbbel indulva kiválasztjuk a középső elemet (páros elemszám esetén a bal oldalit), mint gyökeret, majd a két részfát rekurzívan megépítjük a két oldali résztömbből. Az így kapott fa magassága, mivel tökéletesen kiegyensúlyozott, $\lfloor \log_2 N \rfloor + 1$.

Ha ezt a tulajdonságot meg szeretnénk tartani, akkor önkiegyensúlyozó (self-balancing) bináris fát kell építenünk. Viszont mivel beszúrási műveletet nem végzünk, ezért a gyökértől bármely levélig vezető út nem lehet hosszabb, mint a fa eredeti magassága, ami még a legnagyobb N esetén is csak 20, arról nem is szólva, hogy még ilyen esetben is csak az elemek felének törlése után csökkenne a fa magassága 1-gyel. Ez gyakorlatilag csak az utolsó 1000–10 000 elem esetén okozna jelentős időmegtakarítást. Az állandó egyensúlyban tartás azonban több időt vehet el, mint amennyit így összesen megspórolhatunk, ezért célszerű (és nem utolsósorban egyszerűbb) egy törlés után úgy hagyni a fát, ahogy van.

A törlési algoritmus a következő:

1. Ha a törlendő csúcs levél, akkor töröljük.
2. Ha a törlendő csúcsnak egy gyereke (részfája) van, akkor írjuk vele felül.
3. Ha a törlendő csúcsnak két gyereke (részfája) van, akkor írjuk felül a jobb részfájának legelső elemével (aminek biztosan nincs bal gyereke, mert ha lenne, az még előrébb lenne, mint a legelső elem, ez pedig ellentmondás).

Ez az algoritmus nem változtatja meg az elemek inorder bejárási sorrendjét, azaz megtartja a fa csúcsaira és annak részfáira adott megelőzési tulajdonságot.

S. 64.
Adrián Patrik, 12. évf.
Baross Gábor Középiskola, Szakiskola és Kollégium, Debrecen

A fa tárolását egy tömbbel valósítjuk meg, mert így gyorsabb. Az építésnél az ismert módszert használjuk:

1. A gyökér indexe 1.
2. Bármely i indexű csúcs bal gyerekének indexe $2i$.
3. Bármely i indexű csúcs jobb gyerekének indexe $2i + 1$.

Meg kell ugyanakkor jegyeznünk, hogy a törlések után ez az indexelési séma nem tartható, ezért a bal és jobb gyerekeket minden csúcsra tárolni kell.

A konkrét megvalósítás részletei előtt ejtsünk néhány szót az algoritmus futásidejéről és memóriaigényéről. A fa építésében a rekurzió minden résztömbre a fa egyetlen csúcsát állítja elő; mivel összesen N csúcs van és egy csúcs előállítása $O(1)$ időt igényel, ezért az egész megvan $O(N)$ idő alatt. A törlési műveletek a fa magasságával arányos ideig tartanak ($O(\log N)$), azaz $K \leq N$ miatt $K \cdot O(\log N) = O(N \log N)$ idő alatt készen van. Eszerint az egész program $O(N \log N)$ idő alatt végez a feladattal. A fa minden csúcsának tárolásához $O(1)$ memóriára van szükség és elhanyagolható mennyiségű ($O(\log N)$) veremre, ezért a teljes memóriaigény $N \cdot O(1) + O(\log N) = O(N)$.

A forráskódot csak nagyvonalakban tekintjük itt át, mert benne elhelyezett kommentek részletezik az egyes részek működését.

A teljes fát – mint azt már megjegyeztük – egy statikus foglalású tömbben tároljuk. Ez ugyan nem adaptál a tényleges igényekhez, viszont jóval gyorsabb, mintha az egyes csúcsokat egyesével foglalnánk le és szabadítanánk fel. Az alkalmazott ábrázolás miatt a tömböt a maximális N maximális értékénél nagyobbra, $2^{\lfloor \log_2 N \rfloor + 1} + 1$ eleműre kell minimálisan méretezni. Ennek a maximális értékét egy kicsit felfelé kerekítve kapjuk meg a `nodes` tömb lefoglalt méretét.

A fa `NODE` típusú csúcsai a tömbelem értékét, a bal részfájának és a belőle, mint gyökérből induló fa teljes elemszámát tartalmazzák, továbbá egy mutatót a bal és a jobb gyerekekre. (Ezek tárolásának szükségességére már kitértünk.) Indexek helyett azért inkább mutatókat használunk, hogy a tömbindexek feloldása ne kerüljön extra processzoridőbe. (Még ha csak keveset spórolunk is.) Az elemszámok nyilvántartása azért szükséges, hogy meg tudjuk mondani, hogy a fa i-edik eleme a bal vagy a jobb részfában van, esetleg maga a vizsgált csúcs. Általában `this -> left_count == this -> left -> count`, de így egy feltétellel kevesebb, mert nem kell a `this -> left`-et (`this -> left == NULL`) tesztelni.

A `last_removed_value` értéket akkor használjuk, ha egy kétgyermekes csúcs helyett fizikailag annak soronkövetkezőjét töröljük: ekkor ugyanis rossz értéket távolítottunk el a fából, és ezt helyre kell állítani.

A program csak a standard C elemeire épül, ezért elvileg minden szabványos C fordítóval lefordítható. GCC-vel a `"gcc s64.c -o s64.exe -O3 -std=c99"`, Microsoft Visual C++ alatt a program a `"cl s64.c"` paranccsal fordítható. (A `-std=c99` nélkül a GCC warningot ad.)