3 | Egyszerű adatszerkezetek: tömb és láncolt lista

Eddig az adatainkat mindig tömbben tároltuk, ami az egyik legegyszerűbb adatszerkezet. Most megismerkedünk egy másik alapvető adatszerkezettel is, a láncolt listával.

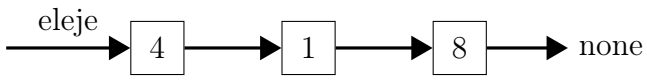
Az adatszerkezet alatt az adatok elrendezésének és tárolásának módját értjük. Az adatszerkezet határozza meg, hogyan kezelhetők és érhetők el az adatok. A leggyakrabban az alábbi műveleteket akarjuk végrehajtani az adatokkal – a választott adatszerkezettől pedig nagyban függ, hogy milyen gyorsan végezhetők el ezek a műveletek:

- keresés (egy adott érték megtalálása a tároltak között),
- beszúrás (új érték beillesztése),
- törlés (egy adott érték megkeresése és eltávolítása a tároltak közül).

Az n-elemű tömb olyan adatszerkezet, amelyben lefoglalunk egymás mellett n darab cellát. A cellák 0-tól $(n-1)$-ig vannak megindexelve, és az index megadásával $O(1)$ lépésben elérjük a megfelelő cella tartalmát.

A láncolt lista esetében nincsenek indexek: az egymás utáni elemekre mutatók (pointerek) mutatnak. A mutató azt mondja meg, hogy hol található a láncolt lista következő eleme.

A 3.1. ábrán egy láncolt listát láthatunk. Az „eleje” mutató a láncolt lista első elemére mutat, a további mutatók pedig rendre a következő elemekre. A legutolsó elem mutatójának értéke „none”. A láncolt listában úgy tudunk mozogni, hogy az elejéről indulunk, majd a mutatók mentén haladva sorban el tudjuk érni az összes elemet.



3.1. ábra. Példa láncolt listára.

Létezik úgynevezett kettős láncolt is, ahol nem csak a következő elemre, hanem az előzőre is mutat egy mutató, de ezekkel itt most nem foglalkozunk részletesebben.

Műveletek tömbben és láncolt listában

A következőkben megvizsgáljuk, hogy hogyan működik a keresés, a beszúrás, és a törlés tömbök és láncolt listák esetén.

Keresés. A láncolt listában úgy tudunk csak keresni, hogy elindulunk a lista elején, és a mutatók mentén haladva sorra végignézzük az összes elemet. A keresés addig tart, amíg meg nem találjuk a keresett értéket, vagy el nem érünk a lista végére. Vagyis egy n-elemű láncolt listában a keresés lépésszáma $O(n)$.

Rendezetlen tömbökben való keresést tulajdonképpen már megvalósítottunk korábban – persze nem csak 7-est lehet keresni, hanem a bemenet részévé tehető, hogy milyen elemet keresünk. Ennek az algoritmusnak is $O(n)$ a lépésszáma.

Azonban ha a tömb rendezett, akkor lehetünk ennél ügyesebbek is; ezt hamarosan látni fogjuk.

Beszúrás. Tömbbe új elemet csak akkor tudunk beszúrni, ha van még szabad hely. Ha a tömb betelt, akkor egy nagyobb tömböt kell lefoglalnunk, amibe át kell másolni az összes eddigi elemet, majd beletenni az új elemet. Emiatt a gyakorlatban sokszor eleve valamivel nagyobb méretű tömböt foglalunk le, mint amire látszólag szükségünk van. Ennek a tömbnek csak az elejét töltjük ki, a vége üresen marad – lehetővé téve, hogy egy darabig könnyedén szúrjunk be új elemeket a tömbbe. De még ha vannak is üres cellák a tömb végén, a beszúrás abban az esetben sem feltétlenül megy azonnal, ha nem a tömb végére szeretnénk elhelyezni az új elemet.

Tegyük fel, adott egy $T[0 : N-1]$ tömb, amiben $n < N$ elemet tárolunk, és valamelyik $i \in \{0, 1, \ldots, n-1\}$ indexű cellába akarunk beszúrni egy új elemet (tehát ahol már van egy elem).

- Ha a tömb elemeinek sorrendjére nem vagyunk tekintettel, akkor ez egyetlen mozgatással megoldható: a $T[i]$ elemet áttesszük az n indexű cellába, így az i indexű cellába már beírhatjuk az új elemet. Ebben az esetben tehát $O(1)$ lépésben tudunk beszúrni a tömbbe.
- Ha a tömb elemeinek sorrendjét meg akarjuk őrizni, akkor az $i, i+1, \ldots, n-1$ indexű cellák tartalmát is eggyel arrébb kell mozgatnunk: a $T[n-1]$ elemet át kell tennünk az n indexű cellába, az $T[n-2]$ elemet az $n-1$ indexű cellába, …, a $T[i]$ elemet az $i+1$ indexű cellába, és ezután írhatjuk be i indexű cellába az új elemet. Ebben az esetben akár n darab mozgatásra is szükség lehet (ha a tömb elejére akarunk beszúrni). Vagyis a rendezett tömbbe való beszúrás lépésszáma $O(n)$.

Ezzel szemben a láncolt lista könnyedén bővíthető bármelyik részén. Ha például az A és B elemek közé szeretnénk beszúrni egy X elemet, akkor elegendő A mutatóját X-re, X mutatóját pedig B-re állítani. Hasonlóan könnyen megy a beszúrás a láncolt lista elejére vagy a végére is. Feltéve, hogy adott egy mutató arra az elemre, amelyik mögé egy új elemet akarunk beszúrni, a láncolt listába beszúrás lépésszáma $O(1)$.

Törlés. A tömbből való törlés hasonlóan működik, mint a beszúrás. Tegyük fel, hogy adott egy $T[0 : N-1]$ tömb, amiben $n < N$ elemet tárolunk, és aminek most valamelyik $i \in \{0, 1, \ldots, n-1\}$ indexű celláját akarjuk törölni.

- Ha a tömb elemeinek sorrendjére nem vagyunk tekintettel, akkor ez egyetlen mozgatással megoldható: a $T[n-1]$ elemet áttesszük az i indexű cellába. Ebben az esetben tehát $O(1)$ lépésben tudunk törölni a tömbből.
- Ha a tömb elemeinek sorrendjét meg akarjuk őrizni, akkor az $i+1, \ldots, n-1$ indexű cellák tartalmát eggyel arrébb kell mozgatnunk: a $T[n-1]$ elemet át kell tennünk az $n-2$ indexű cellába, az $T[n-2]$ elemet az $n-3$ indexű cellába, $\ldots$, a $T[i+1]$ elemet az i indexű cellába. Ebben az esetben akár $n-1$ darab mozgatásra is szükség lehet (ha a tömb első celláját akarjuk törölni). Vagyis a rendezett tömbből való törlés lépésszáma $O(n)$.

Láncolt listában a törlés hasonlóan egyszerű a beszúráshoz. Ha például az A és a B elem között álló X elemet akarjuk törölni, akkor elegendő az A elem mutatóját a B-re átállítani (és az X elemet a mutatójával együtt törölni). Hasonlóan könnyű a láncolt lista első vagy utolsó elemének a törlése is. Feltéve, hogy adott egy mutató arra az elemre, amelyik mögötti elemet ki akarjuk törölni, a láncolt listából törlés lépésszáma $O(1)$.

A tömb és a láncolt lista közötti különbséget jól szemlélteti a következő hasonlat.

A tömb olyan, mint egy rögzített lapú, számozott füzet, amibe folytonosan jegyzetelünk. Az oldalszámok alapján gyorsan kereshetünk, de utólag nem tudunk betoldani oldalakat a füzet belsejébe, csak a végéhez tudunk hozzáírni, feltéve, hogy a füzetünk még nem telt be. Továbbá törölni sem lehet belőle kulturáltan.

A láncolt lista ezzel szemben olyan, mint egy gyűrűs lefűző. Itt nincsenek oldalszámok, vagyis lassabban találjuk meg, amit keresünk, viszont bármikor tetszőleges helyen beszúrhatunk vagy kivehetünk lapokat.

4 | Keresés

Ebben a fejezetben azt a feladatot fogjuk vizsgálni, amikor adott egy n-elemű T tömb és egy s elem, és azt akarjuk eldönteni, hogy az s elem szerepel-e a T tömbben.

Lineáris keresés

Az említett feladatot $s = 7$ esetén már megoldottuk korábban, és nyilván általános s elem esetén is pont ugyanezt kell csinálni. Ezt az eljárást, amikor sorra megvizsgáljuk az elemeket, hogy megegyeznek-e a keresett elemmel, lineáris keresésnek nevezzük.

1. algoritmus: Lineáris keresés rendezetlen tömbben

Input: egy $n \geq 1$ darab különböző számot tartalmazó T tömb és egy s szám
Output: ha T tartalmazza az s számot, akkor az s szám T-beli indexe, ellenkező esetben `nincs`

ciklus: i **fut** 0**-től** $(n-1)$**-ig**
  ha $T[i] == s$**:**
    return i
ciklus vége
return `"nincs"`

Ahhoz az algoritmushoz hasonlóan, amelyikkel eldöntöttük, hogy van-e 7-es egy tömbben, az 1. algoritmus is nyilván helyes és a lépésszáma $O(n)$.

Amennyiben tudjuk, hogy egy növekvően rendezett T tömbben keresünk, úgy korábban is észrevehetjük, ha a lineárisan keresett s elemet nem tartalmazza a T tömb: ha a tömb elejéről indulva még nem láttuk s-et, és az éppen vizsgált elem már nagyobb, mint s, akkor már biztosan állíthatjuk, hogy s nincs benne a T tömbben.

2. algoritmus: Lineáris keresés rendezett tömbben

Input: egy $n \geq 1$ darab különböző számot tartalmazó, növekvően rendezett T tömb, valamint egy s szám

Output: ha T tartalmazza az s számot, akkor az s szám T-beli indexe, ellenkező esetben `nincs`

ciklus: i **fut** 0**-től** $(n-1)$**-ig**
  ha $T[i] == s$**:**
    return i
  különben ha $T[i] > s$**:**
    return `"nincs"`
ciklus vége
return `"nincs"`

Helyesség. A tömb rendezettsége miatt minden olyan cellát megnézünk, amiben a tárolt elem legfeljebb s, vagyis ha s benne van a tömbben, akkor azt biztosan észre fogjuk venni, és ekkor jó kimenetet adunk. Ha azonban s nincs a tömbben, akkor vagy még megnézzük az s-nél nagyobb elemek közül a legkisebbet, vagy a tömb végére érünk, és ebből helyesen következtetünk arra, hogy s nincs a tömbben, azaz ekkor is jó kimenetet adunk. Az algoritmus biztosan véget ér, legkésőbb a tömb végén.

Lépésszám. Tekintsük lépésnek az összehasonlításokat, valamint az eredmény visszaadását. A ciklus n-szer fut le, és minden lefutáskor $O(1)$ lépést végzünk (nevezetesen legfeljebb két összehasonlítást), végül az algoritmus még $O(1)$ lépést tesz (az eredmény visszaadásával). Ez összesen $n \cdot O(1) = O(n)$ lépés.

Könnyű megmutatni azt is, hogy amennyiben a tömb legutolsó elemét vagy egy tömbben nem tárolt elemet keresünk, úgy ez az algoritmus is $n + 1$ darab lépést tesz, vagyis az $O(n)$-es lépésszám-becslésünk éles.

Bináris keresés

Rendezett tömbben azonban még hatékonyabban is tudunk keresni. A módszer lényege a következő ötleten alapul: ha egy nyomtatott szótárban (amiben a szavak ábécé szerint vannak rendezve) keressük az `algoritmus` szót, akkor ezt nem úgy keressük meg, hogy a könyv elejéről egyesével végigolvassuk a szavakat, hanem felcsapjuk valahol a szótárt – ha például az `adatszerkezet` szó az első azon az oldalon, ahol a szótár kinyílt, akkor tudjuk, hogy az `algoritmus` szó hátrébb lesz, míg ha a `tömb` szónál nyílt ki a szótár, akkor tudjuk, hogy előrébb kell keresnünk.

A fenti példa annyiban sántít, hogy vannak előzetes ismereteink a betűk eloszlásáról a magyar nyelvben és érezzük, hogy az `algoritmus` szónak valahol a szótár közepén illene lennie, és nem a szótár végén keresgéljük (ahol nagy valószínűséggel a `tömb` szót találtuk helyette). Ha viszont nincs ilyen előzetes tudásunk a tárolt elemekről, akkor a legjobb hely, ahol felcsaphatjuk a szótárat, az éppen a közepe: ekkor a legrosszabb esetben is csak a szótár felében kell tovább keresgélnünk. Hogyan keresünk a szótár felében tovább? Pontosan ugyanígy: a megmaradt felének megint a közepét nézzük meg, ami után a legrosszabb esetben is már csak a teljes szótár negyedében kell folytatnunk a keresést, stb.

Bináris keresés szövegesen leírva.

A bemenet egy $n \geq 0$ darab különböző számot tartalmazó T tömb és egy s szám. A kimenet az s szám T-beli indexe, ha T tartalmazza s-et, ellenkező esetben `nincs`.

Ha $n = 0$, akkor a válasz `nincs`.

Ha $n \geq 1$, akkor vegyük a tömb középső elemét – ha két középső elem van, akkor ezek közül az előrébb állót vegyük – és hasonlítsuk ezt össze s-sel.

- Ha a középső elem s, akkor a válasz a középső elem indexe.
- Ha a középső elem nagyobb, mint s, akkor a keresést ugyanezzel a módszerrel a tömb első felében folytatjuk.
- Ha a középső elem kisebb, mint s, akkor a keresést ugyanezzel a módszerrel a tömb második felében folytatjuk.

Helyesség. Ha $n = 0$, akkor semmilyen elem sincs a tömbben, tehát jól válaszoljuk azt, hogy s nincs a tömbben. Ha $n \geq 1$, akkor három eset van.

- Az első esetben megtaláltuk s-et a középső cellában, tehát jó indexet adtunk vissza kimenetként.
- A második esetben – tehát amikor a középső elem nagyobb s-nél – a tömb rendezettsége miatt a tömb második felében lévő összes elem nagyobb s-nél; vagyis ekkor ha s szerepel a tömbben, akkor csak a tömb középső eleme előtti részben lehet.
- A harmadik esetben – tehát amikor a középső elem kisebb s-nél – a tömb rendezettsége miatt a tömb első felében lévő összes elem kisebb s-nél; vagyis ekkor ha s szerepel a tömbben, akkor csak a tömb középső eleme utáni részben lehet.

Az eljárás biztosan véget ér, hiszen minden cellában lévő elemet legfeljebb egyszer hasonlítunk össze s-sel.

Bináris keresés pszeudokóddal.

A pszeudokód megírásához szükségünk lesz olyan indexekre, amik segítségével a még megmaradt résztömbünket nyilvántarthatjuk. Ezek tárolására egy `eleje` és egy `vege` változót fogunk használni. Ekkor a $T[\texttt{eleje} : \texttt{vege}]$ tömb középső elemének indexe

- $\frac{\texttt{eleje}+\texttt{vege}}{2}$, ha $\texttt{eleje} + \texttt{vege}$ páros;
- $\frac{\texttt{eleje}+\texttt{vege}-1}{2}$, ha $\texttt{eleje} + \texttt{vege}$ páratlan;

amit összefoglalva úgy is felírhatunk, hogy $\left\lfloor \frac{\texttt{eleje}+\texttt{vege}}{2} \right\rfloor$, azaz az $\frac{\texttt{eleje}+\texttt{vege}}{2}$ szám alsó egészrésze.

Ha a középső elem éppen a keresett elem, akkor visszatérünk a `kozepso` változó értékével. Ha a középső elem kisebb, mint s, akkor a vizsgálandó tömb végének az indexét – azaz a `vege` változót – kell módosítanunk, ellenkező esetben pedig az `eleje` változót.

A tömb egymás utáni felezgetését egy ciklussal fogjuk megvalósítani. A tömböt addig felezgetjük, amíg vagy meg nem találjuk a keresett elemet (és ekkor visszatérünk az elem tömbbeli indexével), vagy amíg marad még vizsgálandó elem, azaz amíg $\texttt{eleje} \leq \texttt{vege}$ teljesül.

---

3. algoritmus: Bináris keresés

Input: egy $n \geq 1$ darab különböző számot tartalmazó, növekvően rendezett T tömb, valamint egy s szám
Output: ha T tartalmazza az s számot, akkor az s szám T-beli indexe, ellenkező esetben `nincs`

$\texttt{eleje} \coloneqq 0$
$\texttt{vege} \coloneqq n - 1$
ciklus: amíg $\texttt{eleje} \leq \texttt{vege}$
  $\texttt{kozepso} \coloneqq \left\lfloor \frac{\texttt{eleje} + \texttt{vege}}{2} \right\rfloor$
  ha $T[\texttt{kozepso}] == s$**:**
    return `kozepso`
  különben ha $T[\texttt{kozepso}] > s$**:**
    $\texttt{vege} \coloneqq \texttt{kozepso} - 1$
  különben:
    $\texttt{eleje} \coloneqq \texttt{kozepso} + 1$
ciklus vége
return "`nincs`"

Ez a pszeudokód pont azt valósítja meg, amit korábban szövegesen néztünk. Tehát az algoritmus helyes.

Lépésszám. Annak meghatározásához, hogy legfeljebb hányszor fut le a ciklusmag, tekintsük a még

vizsgálandó tömb méretét. Jelölje k azt, hogy legfeljebb hányszor fut le a ciklusmag.

- Kezdetben a tömb mérete n.
- A ciklusmag első lefutása után a tömb mérete már legfeljebb csak $n/2$,
- a második lefutás után legfeljebb $n/4$,
- a harmadik lefutás után legfeljebb $n/8$,
- $\vdots$
- a k-adik lefutás után legfeljebb $n/2^k$.

A ciklusmag azért futott még le k-adszorra, mert előtte még a vizsgálandó tömb mérete legalább 1 volt, azaz

$$\frac{n}{2^{k-1}} \geq 1$$

teljesült. Ezt átrendezve

$$n \geq 2^{k-1}$$

adódik. Mivel a 2-es alapú logaritmus-függvény szigorú monoton növekvő, ezért

$$\log_2 n \geq k - 1,$$

azaz

$$k \leq \log_2 n + 1.$$

Összességében tehát az algoritmus elején $O(1)$ lépést végzünk (nevezetesen két értékadást), majd $O(\log_2 n)$-szer fut le a ciklusmag, és minden lefutáskor $O(1)$ lépést végzünk (nevezetesen legfeljebb egy értékadást és legfeljebb egy összehasonlítást), majd az algoritmus egy eredmény-visszaadással ér véget, ami $O(1)$ lépés. Ez összesen

$$O(1) + O(\log_2 n) \cdot O(1) + O(1) = O(\log_2 n)$$

lépés.

Érdemes megfigyelni azt is, hogy egy $\log_2 n$ lépésszámú algoritmus hogyan viselkedik az input méretének növelésekor: a korábbi számolásokhoz hasonlóan egy $2k$-méretű tömbön egy $\log_2 n$ lépésszámú algoritmus $\log_2(2k) = \log_2 2 + \log_2 k = 1 + \log_2 k$ darab lépést tesz, azaz mindössze eggyel több lépésre van szüksége, mint egy k-méretű tömb esetén. A bináris keresés esetén ez szépen látszik is: a $(2k)$-méretű tömbből egy lépésben egy k-méretűt csinál, és azzal dolgozik tovább.

Be lehetne bizonyítani, hogy a bináris keresés olyan értelemben a legjobb, hogy nincs olyan összehasonlítás alapú kereső algoritmus, ami minden esetben kevesebb mint $\log_2 n + 1$ összehasonlítást használ.