Kulcsmanipulációs rendezések

Vannak nem összehasonlítás-alapú rendező algoritmusok is, melyek az összehasonlításokon kívül más információt is kihasználnak a rendezendő tömb elemeiről, például előre tudják, hogy milyen elemek fordulhatnak elő a rendezendő tömb celláiban. Az ilyen algoritmusokat kulcsmanipulációs rendező-algoritmusoknak nevezzük, és ezek bizonyos speciális esetekben akár gyorsabban is tudnak rendezni, mint $O(n \log_2 n)$.

Egy ilyen rendezési algoritmus például a ládarendezés.

Ládarendezés

Az alábbiakban azt az egyszerű esetet fogjuk tekinteni, amikor a bemeneti T tömb elemei a 0, 1, ..., $m-1$ számok közül kerülnek ki.

A ládarendezés szövegesen leírva.
A bemenet egy $n \geq 1$ darab számot tartalmazó T tömb, melynek elemei a 0, 1, ..., $m-1$ számok közül kerülnek ki (ahol m egy adott pozitív egész szám), a kimeneti tömb pedig a bemeneti tömb elemeit tartalmazza növekvő (pontosabban nemcsökkenő) sorrendben.

- Először hozzunk létre egy m-méretű A tömböt, melynek minden eleme legyen egy üres láncolt lista.
- Menjünk végig a T tömb elemein: ha $T[i] == j$, akkor az A tömb j indexű cellájában található láncolt lista végéhez fűzzük hozzá a $T[i]$ elemet (azaz a j számot).
- Ezután menjünk végig az A tömbben található láncolt listákon, és az azokban látott elemeket sorra írjuk át a T tömbbe.
- Végül térjünk vissza a T tömbbel.

Helyesség. Ez az eljárás helyes, hiszen az A tömb j indexű cellájában található láncolt listába éppen a T tömb azon elemeit tesszük bele, amik j-vel egyenlőek, a végén pedig ezeket a láncolt listákat sorban olvassuk ki. Tehát az eljárás végén a T tömb növekvő sorrendben fogja tartalmazni az elemeket. Az algoritmus biztosan véget ér, amit rögtön látni fogunk a lépésszám-elemzésből.

Lépésszám.
- Az A tömb létrehozása $O(m)$ lépés.
- Ezután az n-méretű T tömbön megyünk végig, és minden elemnél $O(1)$ lépést végzünk.

– Majd az A tömbön is végigmegyünk, kiolvassuk az ott található láncolt listákat, és az elemeiket sorban beírjuk a T tömbbe. Ez az m darab láncolt lista összesen n elemet tartalmaz, vagyis a méretük összesen (a láncolt listák végén található `none` értékeket is értelemszerűen beleszámolva) $n + m$, és minden értéknél $O(1)$ lépést végzünk.
– Végül az eredmény-visszaadás $O(1)$ lépés.

Ez összesen

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

lépés.

Vagyis ezzel beláttuk a következő állítást.

5.7. tétel.
Ha az n-méretű bemeneti tömb elemei a $0, 1, \ldots, m - 1$ számok közül kerülnek ki, akkor ládarendezés lépésszáma $O(n + m)$.

Amennyiben m „kicsi”, a ládarendezés lépésszáma jobb is lehet, mint az összefésüléses rendezés $O(n \log_2 n)$-es lépésszáma: ha például $m = 1000n$, akkor a ládarendezés $O(1000n + n) = O(n)$ lépésben végezni fog. Ez természetesen nincs ellentmondásban az 5.6. tétellel, hiszen a ládarendezés nem összehasonlítás-alapú: nem a bemeneti tömb elemeit hasonlítjuk egymáshoz, hanem indirekt címzést használunk annak ismeretében, hogy a tömb milyen számokat tartalmaz.

Ha azonban m „nagy”, akkor a ládarendezés lassabb is lehet az összefésüléses rendezésnél: ha például $m = n^2$, akkor a ládarendezés $O(n^2 + n) = O(n^2)$ lépést fog tenni.

5.8. feladat.
Hogyan kellene módosítani az előző algoritmust, ha azt tudjuk, hogy a bemeneti T tömbben található számok $a_1, a_2, \ldots, a_m$?

Amennyiben a rendezendő elemek valóban csak számok, a ládarendezésben a láncolt listákat egyszerűen kiválthatjuk számlálókkal is: az $A[i]$ cellában található számlálóval azt tartjuk nyilván, hogy a T tömbben az addig megvizsgált elemek között hányszor fordult elő az i szám.

A rendezendő elemek azonban nem csak számok lehetnek: elképzelhető például, hogy olyan azonosítók alapján szeretnénk rekordokat rendezni, amelyek többször is előfordulhatnak, miközben a rekordok többi része eltérő – például egy orvosi rendelőben a páciensek neve szerint szeretnénk rendezni az aktákat. Ilyenkor nem elég azt nyilvántartanunk, hogy az egyes azonosítók hányszor fordultak elő,

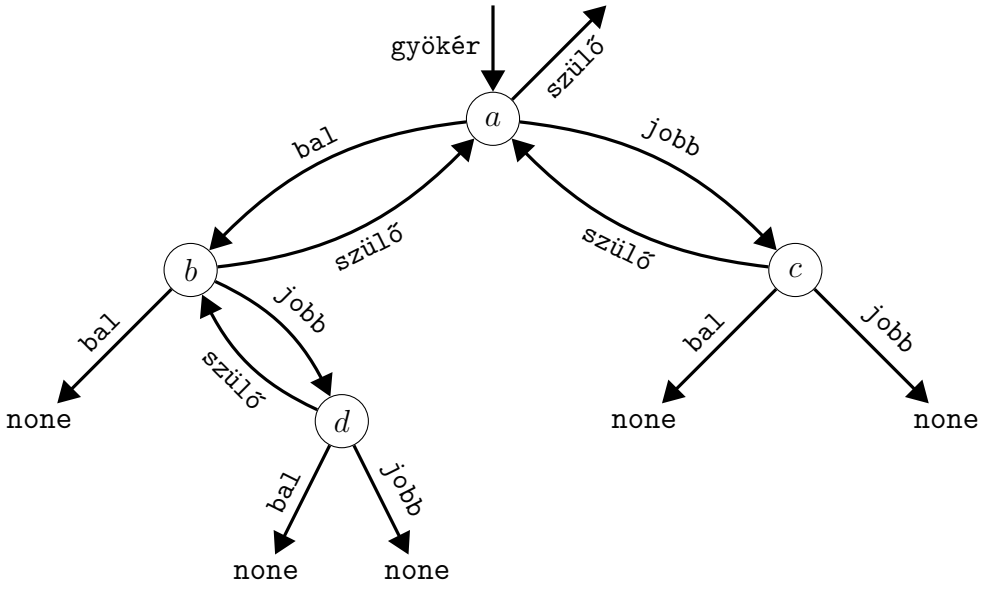
hiszen az azonosítókhoz tartozó rekordokat is meg kell őriznünk – ehhez jön jól a láncolt lista.

Egy másik kulcsmanipulációs rendezés a radixrendezés, ami ennek a tárgynak ugyan nem képezi részét, de érdekességképpen egy rövid megjegyzést teszünk róla. Itt a rendezendő elemeknek több komponense is van: lehetnek ezek például dátumok, amik évszámból, hónapból és napból állnak. A radixrendezés során ezeket az elemeket az utolsó komponensüktől kezdve visszafelé rendezzük a ládarendezésnek a fent látott, láncolt listás megvalósításával. A rendezés helyességéhez ráadásul az is elengedhetetlen, hogy a láncolt listáknak mindig a végére szúrjuk be az elemeket, a kiolvasásukat pedig mindig az elejüknél kezdjük.

6 | Bináris fák

Korábban már láttuk, hogy egy rendezett tömbben gyorsan tudunk keresni, de a beszúrás vagy a törlés már körülményes tud lenni, míg egy láncolt listában lassú a keresés, de ha már tudjuk, hogy hová akarunk beszúrni, vagy honnan akarunk törölni, akkor az már nagyon gyorsan megy. A következőkben egy olyan olyan adatszerkezetet fogunk megismerni, ami kombinálja a tömbök és a láncolt listák jó tulajdonságait: gyors lesz benne a keresés, a beszúrás és a törlés is. Ezt az adatszerkezetet bináris keresőfának fogják hívni, de előbb még a bináris fa fogalmával fogunk megismerkedni.

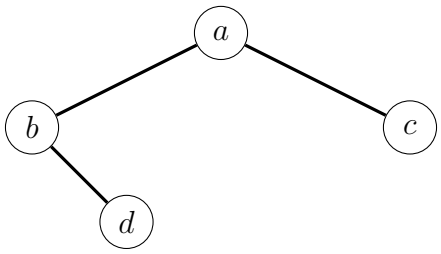
A bináris fák nagyon hasonlítanak a kettős láncolt listához, azzal a különbséggel, hogy minden elemnek kettő helyett három mutatója is lesz. A bináris fákat egy gyökeres, szintezett fának képzeljük el, aminek az elemeit úgynevezett csúcsokban tároljuk. Az „első" elem a fa 0. szintjén elhelyezkedő gyökérnek hívott csúcsban található, és erre mutat a bináris fa `gyökér` nevű mutatója. A fában tárolt minden x elemből két mutató indul ki, egy `bal` és egy `jobb` nevű. Ezek a mutatók vagy a fának újabb csúcsaira mutatnak, amikben további elemeket tárolunk, és egy ilyen csúcsot az x gyerekének nevezünk, vagy `none` értékűek lehetnek. Az x csúcs gyerekei közvetlenül az x alatti szinten helyezkednek el a fában, és a gyerekekből egy-egy `szülő` mutató mutat vissza az x elemre. A gyökércsúcs szülője értelemszerűen `none`. A 6.1. ábrán egy bináris fát láthatunk.



6.1. ábra. Példa bináris fára.

A bináris fában csak a `gyökér` mutatóból indulva a `bal`, `jobb` és `szülő` mutató mentén tudunk

mozogni. A továbbiakban a mutatókat nem fogjuk így kiírni, hanem csak egyszerű vonalakkal fogjuk őket ábrázolni, valamint a `none` értékeket sem fogjuk kiírni, ahogyan a 6.2. ábrán is tesszük.



6.2. ábra. A 6.1. ábrán látható bináris fa egyszerűbben felrajzolva.

Egy x csúcsból `bal` és `jobb` mutatók mentén elérhető csúcsokat az x leszármazottainak nevezzük, míg a `szülő` mutatók mentén elérhető csúcsokat x őseinek nevezzük. Így például a 6.1. ábrán látható bináris fában az a csúcs leszármazottai a b, c és d csúcsok, míg a d csúcs ősei az a és b csúcsok.

Az x csúcs és a leszármazottai által alkotott részt a fában az x gyökerű részfának nevezzük. Az x csúcs bal gyerekéből gyökerező részfát az x bal részfájának, a jobb gyerekéből gyökerezőt pedig az x jobb részfájának nevezzük.

Bináris fák bejárásai

Egy tömb vagy egy láncolt lista esetén nagyon könnyen ki tudjuk írni az összes bennük tárolt elemet: a tömb esetén egyszerűen a cellákon végig tudunk indexelni, míg a láncolt lista esetén az `eleje` mutatótól indulva a `következő` mutatókkal végig tudunk lépkedni az elemeken. Mindkét adatstruktúra esetén, ha n elemet tárolunk bennük, akkor ez $O(n)$ lépést fog igényelni.

A bináris fák esetén nem teljesen egyértelmű, hogy hogyan lehetne gyorsan kiírni a bennük tárolt elemeket. A következőkben mutatunk erre három eljárást is, melyek mindegyike $O(n)$-es lépésszámú lesz. Mindhárom eljárás rekurzív lesz: úgy fog futni egy x gyökerű részfán, hogy meghívja saját magát az x bal és jobb részfájára is.

Preorder bejárás

A preorder bejárás a gyökérből indulva, először kiírja a látott csúcsot értékét, majd rekurzívan meghívódik előbb a csúcs bal, majd pedig a jobb részfájára.

A pszeudokódban az x csúcsban tárolt értéket x`.érték` jelöli, az x bal és jobb részfáját pedig `bal`(x) és `jobb`(x).

20. algoritmus: Egy x gyökerű részfa preorder bejárása (**preorder**)

Input: egy bináris fa x gyökerű részfája
Output: a részfában tárolt elemek felsorolva

print x.`érték`
preorder(`bal`(x))
preorder(`jobb`(x))

Ezek után a teljes bináris fát értelemszerűen a **preorder**(`gyökér`) módon járhatjuk be.

6.1. példa.
A 6.2. ábrán látható bináris fa esetén az a, b, d, c sorrendben fogja kiírni a preorder a fában tárolt elemeket.

Inorder bejárás

Az inorder bejárás a gyökérből indulva, először rekurzívan meghívódik a csúcs bal részfájára, majd kiírja magát a csúcsot, végül pedig a csúcs jobb részfájára hívódik meg.

21. algoritmus: Egy x gyökerű részfa inorder bejárása (**inorder**)

Input: egy bináris fa x gyökerű részfája
Output: a részfában tárolt elemek felsorolva

inorder(`bal`(x))
print x.`érték`
inorder(`jobb`(x))

Ezek után a teljes bináris fát értelemszerűen az **inorder**(`gyökér`) módon járhatjuk be.

6.2. példa.
A 6.2. ábrán látható bináris fa esetén a b, d, a, c sorrendben fogja kiírni az inorder a fában tárolt elemeket.

Posztorder bejárás

A posztorder bejárás a gyökérből indulva, először rekurzívan meghívódik a csúcs bal, majd jobb részfájára, végül pedig kiírja magát a csúcsot.

22. algoritmus: Egy x gyökerű részfa posztorder bejárása (**posztorder**)

Input: egy bináris fa x gyökerű részfája
Output: a részfában tárolt elemek felsorolva

posztorder(`bal`(x))
posztorder(`jobb`(x))
print x`.érték`

Ezek után a teljes bináris fát értelemszerűen a **posztorder**(`gyökér`) módon járhatjuk be.

6.3. példa.
A 6.2. ábrán látható bináris fa esetén a d, b, c, a sorrendben fogja kiírni a posztorder a fában tárolt elemeket.

Nyilván mindhárom bejárás során minden csúcsot pontosan egyszer látogatunk meg, aminek során $O(1)$ lépést végzünk, így a bejárások lépésszáma $O(n)$. Vagyis ezzel beláttuk a következő állítást.

6.4. tétel.
A preorder, inorder és posztorder bejárások lépésszáma $O(n)$.