Összehasonlítás-alapú rendezések és ládarendezés
Bináris keresőfák

Csima Judit
BME SZIT
csima@cs.bme.hu

2019. október 9.

Összehasonlítás-alapú rendező algoritmusok

Látva, hogy az összefésüléses rendezés gyorsabb, mint a többi, korábban látott eljárás, felmerül a kérés, hogy lehetséges-e esetleg még az összefésüléses rendezésnél is gyorsabban rendezni. A válasz attól függ, hogy mit engedünk meg a rendezőalgoritmusnak, milyen típusú kérdésekkel nyerhet információt a rendezendő tömb elemeiről.

Definíció Egy rendezőalgoritmus akkor összehasonlítás-alapú rendezőalgoritmus, ha az elemek rendezése során az algoritmus az input A tömb elemeihez csak $A[i]$ kisebb-e, mint $A[j]$ típusú kérdésekkel fér hozzá, azaz az algoritmus döntései csak a tömb elemeinek egymással való összehasonlításán alapulnak.

Ha visszagondolunk arra, hogy mi történt az eddig tanult rendező algoritmusokban, akkor láthatjuk, hogy mindegyik eljárás összahsonlításalapú volt: a kiválasztásos rendezés során a minimumok megkereséséhez az elemeket egymással hasonlítottuk, a buborékrendezésnél a cserék szükségessége az összehasonlításoktól függött, a beszúrásos rendezésnél az új elem lefele mozgatása a szomszédos elemek összehasonlításától függően tötént meg, az összefésüléses rendezésben pedig az rendezett tömbök összefésülése során csak összehasonlításokat használtunk. Az összehasonlítás-alapú algoritmusok a lehető legáltalánosabb rendezőalgoritmusok, mert a tömb elemeit mindig össze lehet hasonlítani, hiszen különben magának a rendezési feladatnak sem lenne értelme. A következő tétel azt mondja, hogy ezek között az általános, összehasonlítás-alapú rendező algoritmusok között az összefésüléses rendezés nagyságrendileg az egyik leggyorsabb.

Tétel Ha $\mathcal{A}$ egy olyan összehasonlítás-alapú rendezőalgoritmus, ami helyesen rendez minden inputot, akkor biztos, hogy minden n értékre van olyan n méretű input, amin az $\mathcal{A}$ algoritmus legalább $1/4n \log n$ összahasonlítást kénytelen használni.

Ezt a tételt nem bizonyítjuk.

A fenti tétel azt mutatja, hogy egy olyan rendezőalgoritmus, mint például az összefésüléses rendezés, a lehető leggyorsabb, hiszen ennek lépésszáma $O(n \log n)$ és a tétel értelmében minden más eljárásnak is szüksége van $1/4n \log n$ lépésre.

Ládarendezés

Az előző tétel azonban csak akkor igaz, ha összehasolítás-alapú rendezésekről beszélünk. Vannak olyan rendező eljárások, amik nem ilyenek, ezek speciális inputokon használhatók általában, összehasonlításon kívül más módon (is) nyernek információt a rendezendő tömb elemeiről és így speciális esetekben gyorsabban tudnak rendezni, mint például az összefésüléses rendezés.

Ládarendezés

Egy ilyen rendezési algoritmus a ládarendezés, melynek elve a következő.

A ládarendezés legegyszerűbb esetében az input egy n elemű $A[0 : n-1]$ tömb, mely különböző egész számokat tartalmaz. Tudjuk továbbá azt is, hogy a tömb elemei a $0, 1, 2, \ldots, m-1$ egész számok közül kerülnek ki.
Ekkor a következő módon rendezhetjük a számokat:

1. Létrehozunk egy m méretű $B[0 : m-1]$ tömböt (melynek cellái éppen a lehetséges értékekkel vannak indexelve), kezdetben minden cella értéke 0.
2. Végigmegyünk az A tömbön és ha $A[i]$-ben a j értéket látjuk, akkor $B[j]$-t átállítjuk 1-re.
3. Végigmegyünk a B tömbön és ha $B[j]$-ben 1 áll, akkor kiírunk az outputra j-t.

A ládarendezés helyes, mert a B tömbben pontosan azokon a helyeken jelenik meg 1-es, amely értékek szerepeltek az input tömbben és mivel a B tömb indexein rendezetten megyünk végig, az indexek pedig megegyeznek a kiírt számmal, ezért a számok rendezetten kerülnek ki az outputra.

Példa Legyen $A = [1, 8, 3, 7, 2, 6]$ és $m = 9$. Az A tömb végigjárása után a B tömb így néz ki: $[0, 1, 1, 1, 0, 0, 1, 1, 1]$, a B tömböt végigjárva pedig a $C = [1, 2, 3, 6, 7, 8]$ tömböt kapjuk kimenetként.

A ládarendezés pszeudokódja

Az inputot jelölje $A[0 : n-1]$, emellé létrehozunk egy $B[0 : m-1]$ tömböt, csupa nullával és az output $C[0 : n-1]$ tömböt, melynek kezdetben minden értéke $None$. Ezután a következő történik:

```
ciklus i = 0-tól (n-1)-ig:
    B[A[i]] := 1
ciklus vége

l := 0
ciklus j=0-tól (m-1)-ig:
    ha B[j] == 1:
        C[l]: = j
        l := l + 1
ciklus vége
```

A ládarendezés lépésszáma

Az első ciklus n-szer fut le, a ciklusmag konstans sok lépésből áll, így ez a kódrészlet $O(n)$. A második ciklus előtt 1 lépés történik, a ciklusmag m-szer fut le, egy lefutás konstans sok lépésből áll, vagyis ez a rész $O(m)$, a két rész együtt pedig $O(n+m)$.

Vegyük észre, hogy ez a lépésszám jobb lehet, mint az összefésüléses rendezés $O(n \log n)$-es lépésszáma, amennyiben m kicsi. Ha például $m \leq 1000n$, akkor $O(n+m)$ $O(n)$-et jelent.

Vegyük észre azt is, hogy ez nincs ellentmondásban azzal a tétellel, hogy legalább $1/4n \log n$ összehasonlítás kell minden összehasonlítás-alapú rendezésnek, mert a ládarendezés nem összehasonlítás-alapú, nem a tömb elemeit hasonlítjuk egymáshoz, hanem a tömb értékeit használjuk indirekt címzéssel a B tömb átalakítására.

Vegyük még észre azt is, hogy a módszer akkor is használható, ha a rendezendő számok nem 0 és $m-1$ közöttiek, mert ekkor megkeresve a legkisebb és legnagyobb értékeket, az eljárást kicsit átalakítva ($m = legnagyobb - legkisebb$ választással és $B[A[i]] := 1$ helyett $B[A[i] - legkisebb] :=$ 1-et használva, illetve a második ciklust is értelemszerűen módosítva) minden ugyanúgy működik, mint eddig. Viszont ha az $m = legnagyobb - legkisebb$ különbség nagyon nagy (pl. IP címeket akarunk rendezni, ahol m nagyjából 2^{128}), akkor ez a módszer nagyon pazarló és jobban járunk az összefésüléses rendezéssel.

A ládarendezés kis módosítással használható akkor is, ha az input elemei nem különbözőek (ekkor a B tömbben számlálókat használunk $0-1$ értékek helyett), ezt a változatot részletesen megtárgyalták a Programozás alapjai tárgy 3. előadásán. Ez mutatja a ládarendezés egy tipikus alkalmazási módját: konstans sok lehetséges érték fordulhat elő az inputon, ekkor az eljárás lépésszáma $O(n)$ lesz, mivel ekkor m értéke konstans.

Egyszerű adatszerkezetek

Az eddigiekben az adatainkat, a rendezendő számokat mindig tömbben tároltuk. A tömb az egyik legegyszerűbb adataszerkezet, most több más adatszerkezettel is meg fogunk ismerkedni.

Adatszerkezet alatt ebben a tárgyban azt értjük, hogy megmondjuk, hogy hogyan tároljuk az adatainkat. A tárolás módjának ott lesz jelentősége, hogy a használt adatszerkezet jelentősen befolyásolja a használt műveletek gyorsaságát, mi általában a keresés (adott érték megkeresése a tároltak között), beszúrás (új érték beillesztése), illetve törlés (adott érték megkeresése és törlése a tároltak közül) műveleteket fogjuk vizsgálni.

Mielőtt az adatszerkezeteket tárgyalni kezdenénk teszünk egy kis kitérőt, megbeszéljük, hogy milyen formátumú adatokat akarunk tárolni. Az eddigiekben mindig számokról beszéltünk, ezek között kerestünk, ezeket rendeztük. Meggondolható, hogy a legtöbb tanult algoritmus (a ládarendezés kivételével) számok helyett bármi más adattal is működne, feltéve, hogy a tárolt elemeket egymással össze lehet hasonlítani. Ha például nem számokat, hanem Neptun-kódokat akarunk rendezni, az összefésüléses rendezés ugyanúgy működik, hiszen ahhoz csak arra volt szükség, hogy két tárolt elemről meg tudjuk mondani, melyik a kisebb, pl. BATMAN előrébb van a sorrendben, mint JOKER1.

A valóságban azonban még ennél is bonyolultabb a helyzet, mert itt nem számokat vagy szavakat tárolok, hanem több részből álló objektumokat pl. (Neptun-kód, név, születési dátum, születési hely) négyeseket és ilyen négyesek halmazában akarunk gyorsan keresni pl. Neptun-kód szerint. Ebben az általános helyzetben a tárolandó objektumoknak mindig van egy kitüntetett komponense, ami alapján a tárolás és a keresés zajlik, ezt kulcsnak nevezzük (a fenti példában pl. a Neptun kód lehet ilyen). Ekkor, noha több komponensű objektumokat tárolok, a tárolás/keresés

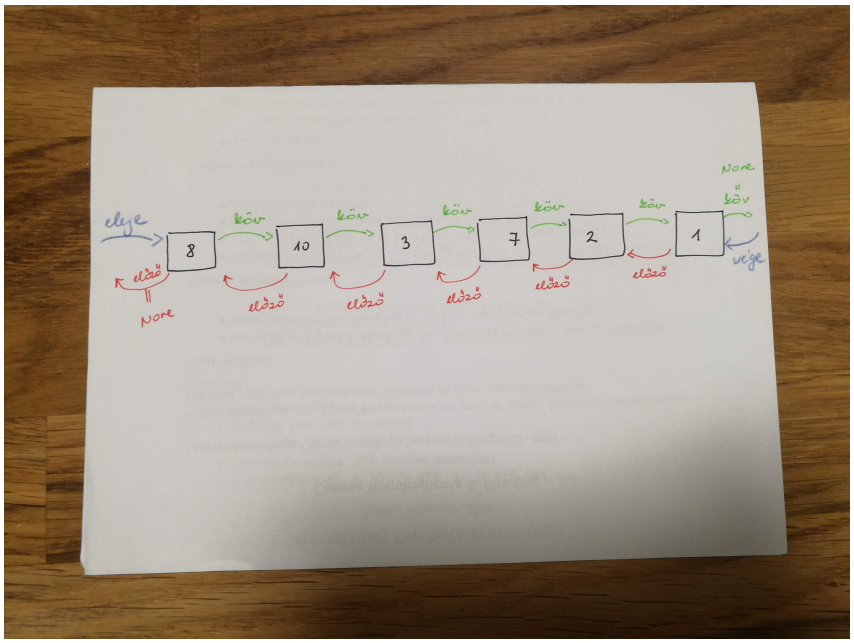
során csak a kulcs mező értéke játszik szerepet, ezért a továbbiakban úgy tekintjük, mintha csak ezt a kulcs mezőt tárolnánk és feltesszük, hogy a tárolandó adatok számok (vagy olyan elemek, melyeket egymással tudunk hasonlítani, de az egyszerűség kedvéért mindig számokat fogunk mondani.)

Tömb és lista

Ahogy már eddig is használtuk, az n elemű tömb olyan adatszerkezet, amiben lefoglalunk n darab helyett (cellának nevezzük ezeket), a cellák meg vannak indexelve 0-tól $n-1$-ig és az index megadásával egy lépésben el tudjuk érni az adott indexű cella tartalmát.

Egy másik egyszerű adatszerkezet a lista. Itt nincsenek indexek, az egymást követő cellákat mutatók kötik össze egymással, a listán belül ezek segítségével tudunk mozogni. A mutató az az információ, hogy hol van a következő elem. A lista fogalmát az alábbi példával lehet szemléltetni: képzeljük el, hogy kincskeresős játékot szervezünk, amiben egy kirakós játék darabjait kell összegyűjteni. A játék elején megmondjuk, hogy hol van az első darab, pl. a lábtörlő alatt, majd a lábtörlő alatt megtaláljuk az első darabot, mellette egy újabb cetlivel, hogy a második darab az előszobában, a cipők között van, ahol megtaláljuk a második darabot és egy cetlit, higy a 3. darab a konyhában, a kések mellett van, ahol megtaláljuk a harmadik darabot és mellette egy cetlit, hogy ez volt az utolsó darab.

Az alábbi ábra azt mutatja, hogy hogyan képzeljük el a listát. Ez a lista egy úgynevezett kettős láncolt lista, amiben a kék "eleje" mutató mutatja a lista első tagját, a kék "vége" mutató pedig az utolsót (hogy a listát mindkét oldalról indulva be lehessen járni). A z előre mutató nyilak mutatják a következő, a hátra mutató piros nyilak pedig az előző elemet (a legutolsó elemnél a "következő" mutató értéke "None", az első elemnél az "előző" mutató értéke "None"). A listában úgy tudunk mozogni, hogy vagy az elején vagy a végén elindulunk és a mutatók mentén haladva fel tudjuk keresni az összes tárolt elemet (de pl. nem tudunk a lista közepére ugrani egy lépésben).



Műveletek tömbben és listában

Keresés
A tömbben akkor tudunk gyorsan keresni, ha a tömb rendezett, ekkor a bináris kereséssel $O(\log n)$ lépésben a keresés megvalósítható. Listában nem tudunk máshogy keresni, mint úgy, hogy vagy az elején vagy a végén elindulunk és a mutatók mentén haladva végignézzük az összes elemet, amíg vagy meg nem találjuk a keresett értéket vagy a lista végére nem érünk. Ennek lépésszáma $O(n)$.

Beszúrás
A tömbbe új elemet beszúrni nem tudunk, ha a tömb tele van. Ekkor az egyetlen lehetőség az, hogy egy újabb, hosszabb tömböt veszünk fel, abba átmásoljuk az összes eddigi elemet és belerakjuk az új értéket is. A gyakorlatban a tömbök megvalósítása úgy történik, hogy egy nagyobb tömböt részlegesen töltünk csak fel először elemekkel úgy, hogy az elemek a tömb elején vannak, a tömb végén levő cellák üresek, így a fenti probléma (nem fér bele az új elem a tömbbe) nem merül fel (egy darabig). Azonban még abban az esetben, ha van hely a tömb végén, akkor is nehézkes a beszúrás, amennyiben meg akarjuk őrizni a rendezettséget (márpedig ezt általában meg akarjuk őrizni, mert ettől lesz gyors a keresés). Előfordulhat, hogy a beszúrandó új elem kisebb, mint bármelyik korábbi érték, akkor az új elemnek az első, a 0-s indexű cellába kell kerülnie, minden más elemet hátra kell mozgatnunk eggyel, ez $O(n)$ lépés.

Listában ez sokkal egyszerűbb: egyrészt a lista nem fix méretű, bővíthető, nem merül fel az a probléma, hogy az új elem nem fér be; másrészt tetszőleges helyre be tudjuk szúrni az új elemet, mert ha például A és B közé akarok X-et berakni, akkor csak az A "következő" mutatóját kell X-re, B "előző" mutatóját X-re állítanom és X két mutatóját A-ra ás B-re irányítanom. (A lista elejére vagy végére hasonlóan egyszerű beszúrni.) Ez a művelet $O(1)$, azaz konstans lépés, ha meg van adva (pl. egy mutatóval), hogy mely elem elé vagy mögé kell az újat beszúrni.
Törlés
A tömbből azért nehéz törölni, mert ekkor a törölt elem utáni összes cella tartalmát balra kell mozgatni eggyel, ami az első elem törlésénél sok mozgatást jelent, a törlés lépésszáma $O(n)$.

Listában a törlés is sokkal egyszerűbb ha meg van adva (pl. egy mutatóval), hogy mely elem elől vagy mögül kell törölni: ha például A és B közül akarjuk X-et kitörölni, akkor csak az A "következő" mutatóját kell X helyett B-re, B "előző" mutatóját X helyett A-ra állítanom, ennek költsége $O(1)$, azaz konstans.

Megjegyzések a tömb és a lista közötti különbségekről

A tömb és a lista közötti különbséget az alábbi szemléletes példa mutatja: ha jegyzeteket akarunk készíteni, akkor választhatunk egy rögzített lapú füzetet vagy könyvet, aminek lapjai számozva vannak vagy használhatunk kivehető lapos, kapcsos jegyzettömböt is. A rögzített lapú, számozott oldalú füzetben az oldalszámok alapján tartalomjegyzésket tudunk csinálni és gyorsan tudunk keresni, de nem tudunk utólag betoldani extra oldalakat a jegyzet belsejébe, csak a jegyzet végét tudjuk bővíteni (és törölni de tudunk kulturáltan). A kapcsos jegyzettömbös megoldásban nincsenek ugyan oldalszámok, ami segítene gyorsan megkeresni valamit, de bármikor tudunk betoldani és kivenni lapokat.

A tömb és a lista adatszerkezet a legtöbb programozási nyelvben elkülönül, de a Pythonban nem. A Pythonban tanult list nevű dolog (a neve ellenére) leginkább egy tömbnek tekinthető, mert vannak indexek, bár a list mérete nem fix. Ebben a list-ben (noha tömb-szerű) látszólag tudunk törölni és beszúrni (legalábbis vannak ilyen műveletek a nyelvben), de ez csalóka: mivel a list mögött egy tömb van igazából, a list elejéről törölni sokkal lassabb, mint a végéről. Ha egy

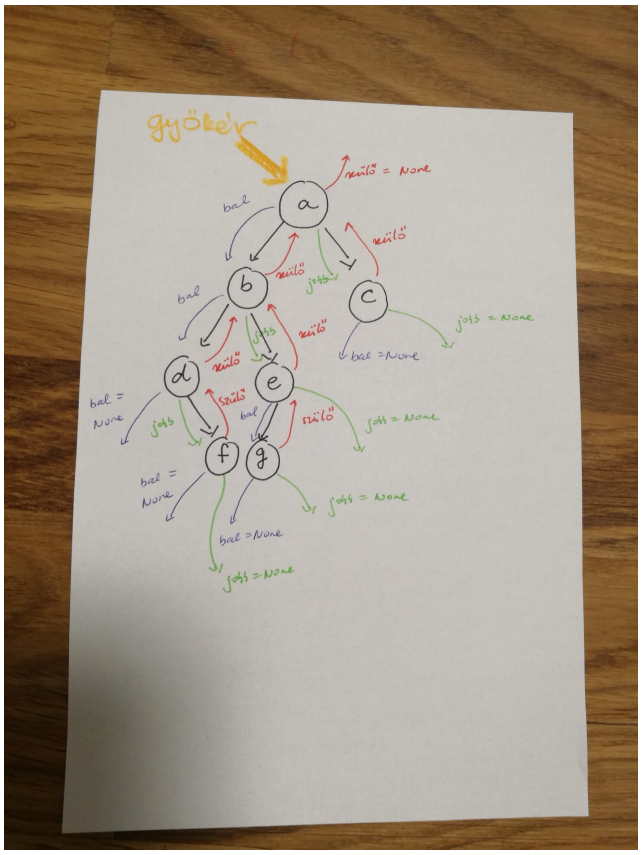
300 000 hosszú list elejéről kitörlöm az elemek tizedét, akkor az 6 másodpercig tart nagyjából, ha azonbana 300 000 hosszú lista végéről törlök, akkor az csak 0.004 másodperc. Ez azért van így, mert az első elemek törlésekor a háttárben a list módosítása sokkal több munkát igényel, mint ha a végén törlök.

Bináris fák

Láttuk az előző részben, hogy a rendezett tömb remek, ha keresni akarunk (a keresés lépésszáma $O(\log n)$), de rossz, ha beszúrás vagy törlés történik, a lista pedig ugyan jó, ha törölni vagy beszúrni akarunk, de lassú benne a keresés. A továbbiakban egy olyan adatszerkezetet fogunk megismerni, ami kombinálja a két egyszerű adatszerkezet jó tulajdonságait: gyors lesz benne mindhárom művelet (és még más műveletek is).

Ehhez először a bináris fa fogalmával kell megismerkednünk. A bináris fában csúcsokban értékeket tárolunk, a csúcsokat pedig olyan mutatók kötik össze, amilyeneket a listánál már megismertünk. Azért hívják a fát binárisnak, mert minden csúcsnak legfeljebb két gyereke lehet. Azt a csúcsot, ahol a fa "kezdődik" gyökérnek hívjuk, azokat a csúcsokat pedig, akiknek nincs gyereke leveleknek.

Az alábbi ábra egy bináris fát mutat, ahol a tárolt elemek a, b, c, d, e, f, g, ezeket pedig a nyilakkal jelölt mutatók kötik össze: a nagy narancssárga "gyökér" mutató mutatja a fa gyökércsúcsát, a kék "bal" mutatók adják meg minden csúcshoz a baloldali gyeket (ha nincs baloldali gyerek, akkor ennek a mutatónak az értéke "None"), a zöld "jobb" mutatók adják meg minden csúcshoz a jobboldali gyeket (ha nincs ilyen gyerek, akkor ennek a mutatónak az értéke "None"), a piros "szülő" mutatók pedig a szülő csúcsot adják meg (a gyökérnél ennek értéke "None").



Fontos észben tartani, hogy a fában csak a mutatók mentén tudunk mozogni, a gyökértől

kiindulva (és pl. nem tudunk (külön előkészületek nélkül) szintenként haladni vagy a levelekre ugrani egy lépésben). A későbbiekben nem fogom a mutatókat berajzolni a fába, csak egy-egy (balra vagy jobbra menő) vonallal jelzem, hogy ki kinek a gyereke.

A bináris fákról beszélve az alábbi szóhasználatot fogjuk követni: egy x csúcs leszárazottjai azon y csúcsok a fában, akik x-ből gyerekmutatókon keresztül (többet is felhasználva esetleg) elérhetők. Ekkor az x csúcsot az ilyen y csúcsok ősének is nevezzük.
Ha x a bináris fa egy csúcsa, akkor az x gyökerű részfa az x-ből, mint gyökérből és x összes leszármazottjából álló fát jelenti.

Bináris fa bejárásai

Sokszor hasznos lesz az, ha egy bináris fa elemeit (valamilyen) sorban ki tudjuk írni. Mivel a fában csak a mutatók mentén tudunk mozogni, ezért erre külön eljárásokat használunk, rögtön három ilyet is tanulunk. Mindhárom eljárás rekurzív lesz, azaz úgy fog futni egy x gyökerű részfán, hogy meghívja saját magát az x gyerekeihez tartozó részfákra.

Preorder bejárás
A preorder bejárást egy x gyökerű részfára hívjuk meg (a fa teljes bejárásakor x az egész fa gyökere lesz).

A preorder(x) eljárás pszeudokódja ez:

```
x meglátogatása
preorder(bal(x))
preorder(jobb(x))
```

Tehát először meglátogatjuk x-et, majd meghívjuk az eljárást először a bal, majd a jobb részfára. A fenti képen látott példa esetén a preorder bejárás a csúcsokat a, b, d, f, e, g, c sorrendben keresi fel.

A preorder bejárás lépésszáma $O(n)$, ha a fának n csúcsa van, mert minden csúcsnál egyszer látogatunk és további két hívás történik a két gyerekre, azaz összesen $3n$ lépésből áll az eljárás.

Inorder bejárás
Az inorder bejárásban először a bal részfára hívjuk meg az eljárást (bejárjuk a bal részfát), aztán meglátogatjuk x-et, majd meghívjuk az eljárást a jobb részfára.

Az inorder(x) eljárás pszeudokódja ez:

```
inorder(bal(x))
x meglátogatása
inorder(jobb(x))
```

A fenti képen látott példa esetén az inorder bejárás a csúcsokat d, f, b, g, e, a, c sorrendben keresi fel.

Az inorder bejárás lépésszáma is $O(n)$, ha a fának n csúcsa van, mert minden csúcsnál egyszer látogatunk és itt is további két hívás történik a két gyerekre, azaz összesen $3n$ lépésből áll az eljárás.

Posztorder bejárás
A posztorder bejárásban először a bal részfára, majd a jobb részfára hívjuk meg az eljárást (bejárjuk a két részfát), aztán látogatjuk meg x-et.

A posztorder(x) eljárás pszeudokódja ez:

```
posztorder(bal(x))
posztorder(jobb(x))
x meglátogatása
```

A fenti képen látott példa esetén a posztorder bejárás a csúcsokat f, d, g, e, b, c, a sorrendeben keresi fel.

A posztorder bejárás lépésszáma is $O(n)$, ha a fának n csúcsa van, mert minden csúcsnál egyszer látogatunk és itt is további két hívás történik a két gyerekre, azaz összesen $3n$ lépésből áll az eljárás.

Mindhárom bejárás alkalmas arra, hogy a fában tárolt elemeket gyorsan kiírja, további hasznos alkalmazási lehetőségekről a gyakorlaton illetve a jövő órán fogunk beszélni.

Bináris keresőfák

A bináris keresőfák olyan bináris fák, melyekben a tárolt adatokra teljesül a bináris keresőfa tulajdonság: tetszőleges x csúcsára a fának igaz az, hogy x bal részfájában minden elem kisebb, mint x, x jobb részfájában pedig midnen elem nagyobb, mint x.
Az alábbi kép bal oldalán egy bináris keresőfa látható, a jobb oldalán pedig egy olyan bináris fa, amiben nem teljesül a bináris keresőfa tulajdonság (mert a 8 jobb részfájában van 6).