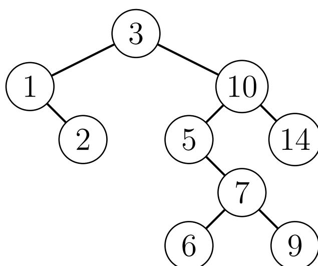
Műveletek bináris keresőfákban, AVL-fák
Hash

Csima Judit
BME SZIT
csima@cs.bme.hu

2019. október 16.

Bináris keresőfák műveletei

Az előző órán megismerkedtünk a bináris fa fogalmával, most pedig azt nézzük meg, hogyan lehet a keresés, beszúrás és törlés műveleteket elvégezni, ha a számokat bináris keresőfában tároljuk. A műveletek illusztrálására az alábbi fát fogjuk használni:



Keresés
Ha a fenti példában keressük a 9-et, akkor a 9-et a gyökérrel összehasonlítva láthatjuk, hogy a 3-as gyökérből a jobb részfa felé kell mennünk (mert a keresőfa tulajdonság miatt arra vannak a 3-nál nagyobb értékek). Hasonlóan folytatva a 10-ből balra, az 5-ből jobbra, a 7-ből ismét jobbra megyünk és megtaláljuk a 9-et. Eközben annyi lépést tettünk, ahány szinten a fában át kellett haladnunk.

Ha a fenti példában a 11-et keressük, akkor a gyökérből megint a jobb részfa felé kell mennünk, onnan megint jobbra, a 14-ből pedig balra, de ott nincs semmi, vagyis ott, ahol a 11-nek lennie kellene nincsen semmi, tehát a keresett érték nincsen a fában.

Általában is úgy keresünk, hogy a keresett s értéket az aktuálisan vizsgált csúccsal összehasonlítjuk (ez a csúcs kezdetben a gyökér), majd vagy jobbra vagy balra lépünk a gyerek mutatók segítségével a következők szerint:

- ha az aktuális csúcsban s van, akkor megtaláltuk a keresett számot,
- ha s kisebb, mint az aktuális csúcs értéke, akkor a bal gyerek felé megyünk tovább és folytatjuk ezt az eljárást.
- ha s nagyobb, mint az aktuális csúcs értéke, akkor pedig a jobb gyerek felé megyünk tovább és folytatjuk ezt az eljárást.

Ez az eljárás úgy ér véget, hogy vagy megtaláljuk a keresett elemet, vagy egy olyan helyen látunk $None$ értéket a gyerekmutatónál, amerre lennie kellene valaminek, ha a keresett elem a fában volna (így vesszük észre, hogy az elem nincs a fában).
Ez az eljárás azért helyes, mert ha egy x csúcs részfájában keresünk, akkor a keresőfa tulajdonság miatt az x-nél kisebb elemek a bal, az x-nél nagyobb elemek pedig a jobb részfában vannak.

A keresésnek a lépésszáma attól függ, hogy hány szinten haladunk át, ez pedig legfeljebb h, a fa magassága lehet. Mivel egy szinten konstans sok lépésst teszünk, a keresés lépésszáma $O(h)$.

Beszúrás
Ha a fenti példában a 11-et akarjuk beszúrni, akkor először megkeressük a helyet, ahol lennie kellene. Ez a keresés ugyanaz, mintha magát az elemet keresném, azaz (a példában) a gyökérből megint a jobb részfa felé kell mennünk, onnan megint jobbra, a 14-ből pedig balra és amikor látjuk, hogy ott nincs semmi, akkor tudjunk, hogy ide kell jönnie a 11-nek, vagyis a 14-es csúcs bal gyerek mutatóját beállítjuk a 11-re, a 11 szülő mutatóját pedig a 14-re.

Általában is úgy szúrunk be, hogy a beszúrandó s értéket az aktuálisan vizsgált csúccsal összehasonlítjuk (ez a csúcs kezdetben a gyökér), majd vagy jobbra vagy balra lépünk a gyerek mutatók segítségével ahogy a keresésnél történik, mindaddig míg a beszúrandó elem helyét meg nem találjuk. Ekkor két mutató beállításával be tudjuk már illeszteni az új elemet. Ez az eljárás azért helyes, mert a (sikertelen) keresés éppen ott fog véget érni, ahol a beszúrandó elemnek lennie kellene.

A beszúrás lépésszáma csak konstans sok lépéssel több, mint a keresésé, vagyis ez is $O(h)$.

Maximum- és minimumkeresés
Ha a gyökérnek nincsen jobb gyereke, akkor a gyökér a legnagyobb elem a fában. Ha a gyökérnek van jobb részfája, akkor a maximális elem csak ebben lehet, ezért ilyenkor jobbra lépünk és ezt az eljárást folytatjuk addig, amíg már nem tudunk jobbra lépni (az aktuális csúcsnak nincs jobb gyereke). Ez a csúcs, ahol elakadtunk lesz a fa maximális értéke.
A fenti példában ez azt jelenti, hogy addig megyünk jobbra, amígy a 14-hez nem érünk, itt fogunk elakadni.

Ez az eljárás azért helyes, mert a keresőfa tulajdonság miatt (jobbra vannak a nagyobb elemek) mindig jobbra kell lépnünk, amíg tudunk.

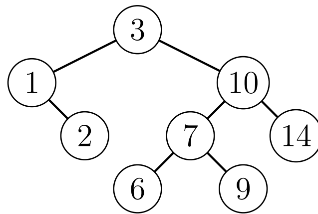
A minimumkeresés hasonlóan zajlik, csak itt addig megyünk balra, amíg tudunk.

Mindkét eljárásra igaz, hogy minden szinten konstans sok lépéss történik, vagyis a lépésszám $O(h)$ ismét.

Törlés
A törlés mindig a törlendő s elem keresésével kezdődik. Ha az s érték nincs a fában, akkor kész vagyunk, nem kell (nem tudjuk) kitörölni. Ha s benne van a fában, akkor három lehetőség van aszerint, hogy a törlendő s elemnek hány gyereke van:
0 gyerek: Ekkor s könnyen törölhető úgy, hogy az s szülőjének eddig s-re mutató gyerekmutatóját $None$-ra állítjuk, ez a keresésen felül konstans sok lépés.
1 gyerek: Ekkor s könnyen törölhető úgy, hogy az s szülőjének eddig s-re mutató gyerekmutatóját s egyetlen gyerekére irányítjuk (és s egyetlen gyerekének szülő-mutatóját pedig s szülőjére), ez a keresésen felül konstans sok lépés.
Ez ezért eredményez bináris keresőfát (azaz nem rontjuk el a keresőfa tulajdonságot), mert így s-et és egész részfáját helyettesítjük s gyerekével és annak részfájával és ha s részfája jól állt s szülőjéhez képest, akkor s gyerekének részfája is jól fog állni (mert ugyanazon az oldalon áll,

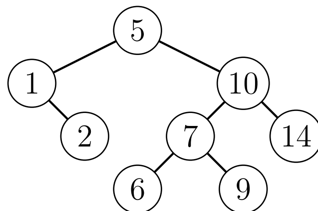
mint eddig).

A fenti példában az 5-t törölve ezt kapjuk:



2 gyerek: Ekkor a következőt tesszük: a keresőfa egy olyan $*$ elemét akarjuk s helyére rakni, amivel továbbra is fennáll a bináris keresőfa tulajdonság, azaz a $*$ elem kisebb, mint a jobb részfájának minden eleme, nagyobb, mint a bal részfájának minden eleme és jól áll a szülőhöz (s szülőjéhez) képest. Egy ilyen tulajdonságú jó választás az s jobb részfájának legkisebb eleme, ezt fogjuk kivenni a jobb részfából és s helyére rakni. Ez, mivel az eddigi jobb részfa legkisebb eleme volt kisebb lesz, mint minden más elem a jobb részfában. Továbbá mivel a jobb részfában volt, ezért s-nél nagyobb, ami nagyobb volt a bal részfa minden eleménél és mivel a $*$ elem s részfájában volt korábban és most s helyére kerül, ezért jól fog állni s szülőjéhez képest is.

Ebben az esetben a törlés egy keresést jelent (ez $O(h)$), aztán egy minimumkeresést s jobb fájában (ez megint csak $O(h)$), majd néhány mutatót kell állítanunk (s szülőjénél, illetve gyerekeinél, továbbá az s helyére mozgatott elemnél, de ezek együtt is csak konstans lépés), majd pedig végre kell hajtanunk a $*$ elem törlését a régi helyéről, ami viszont egy olyan törlés, amikor a törlendő $*$ elemnek nincs vagy csak 1 gyereke van (mert ha lenne $*$-nak bal gyereke, akkor nem $*$ lenne a részfában a legkisebb). Ezen utolsó lépés is $O(h)$, vagyis a teljes lépésszám $O(h)$ ebben az esetben is.
A fenti példában a 3-at törölve ezt kapjuk:



A bináris keresőfa szintjeinek száma

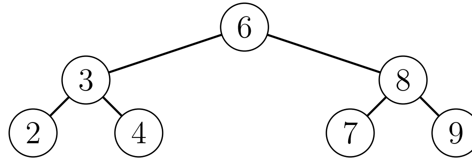
Láttuk, hogy az összes művelet lépésszáma $O(h)$, ahol h a fa szintjeinek száma, ezért most megvizsgáljuk, hogy mekkora lehet h a fában tárolt elemek számának n-nek a függvényében.

A rossz hír az, hogy $h = n$ is lehetséges, amikor a fa egy jobbra vagy balra tartó út, azaz ebben az esetben az összes művelet lépésszáma n-nel arányos.

Vizsgáljuk meg, hogy mennyire lehet kicsi h? Melyik az a fa, amikor a lehető legkevesebb szintre rakunk be lehető legtöbb csúcsot? Ez a fa nem más, mint a teljes bináris fa, azaz egy olyan bináris fa, ahol a leveleket kivéve minden csúcsnak két gyereke van. Ebben a fában az 1. szinten 1, a 2. szinten 2., a 3. szinten $4 = 2^2$, a 4. szinten $8 = 2^3$, általában is a k. szinten 2^{k-1} csúcs van, azaz a h szinten összesen $1 + 2 + 4 + \dots 2^{h-1} = 2^h - 1$ csúcs van. Azaz ekkor $n = 2^h - 1$ vagyis $h = \log(n + 1)$ vagyis $O(h)$ ebben a fában $O(\log n)$-t jelent.

A fentiek alapján az volna az ideális, ha a számokat egy teljes bináris keresőfában tárolnánk (vagy hasonló érveléssel belátható lenne, hogy akkor is megkapnánk h-ra az $O(\log n)$-es értéket,

ha a fa az utolsó szintet kivéve van csak tele, az utolsó szinten hiányozhatnak csúcsok, tehát egy ilyen fa is jó lenne). Az a gond ezzel a megoldással, hogy amikor egy ilyen fába elkezdünk elemeket beszúrni, akkor a fa szép tulajdonsága, a teljessége gyorsan el tud romlani. Ha például ebbe a fába beszúrjuk egymás után a $10, 11, 12, \ldots, 20$ elemeket, akkor a fa már nagyon mesze lesz a teljes bináris fától:



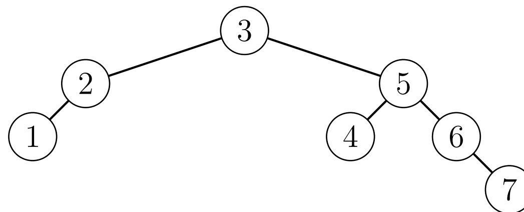
Kiegyensúlyozott keresőfák, AVL-fa

Láttuk, hogy olyan bináris keresőfákat szeretnénk használni, ahol a fa h magassága (szintjeinek száma) $O(\log n)$. Ezt úgy fogjuk elérni, hogy speciális bináris keresőfákat fogunk használni, olyanokat, ahol a bináris keresőfa tulajdonságon felül valami további szép tulajdonsággal is rendelkezik a fa és ez a szép tulajdonság

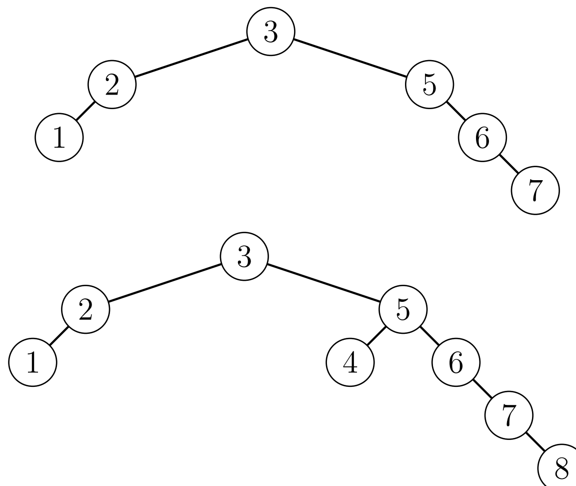
- garantálja, hogy h $O(\log n)$-es és
- a szép tulajdonság vagy nem romlik el a beszúrás és törlés közben vagy ha mégis, akkor $O(\log n)$ munkával helyreállítható.

Egy ilyen lehetőség a következő, ahol a bináris keresőfa tulajdonságon felül azt követeljük meg, hogy minden csúcsra legyen az igaz, hogy a csúcs jobb és bal részfájának magassága legfeljebb eggyel tér el (ezt AVL-tulajdonságnak nevezzük). Az ilyen bináris keresőfákat, ahol ez az AVL-tulajdonság is teljesül, AVL-fának nevezzük.

Az alábbi fa például egy AVL-fa, mert minden csúcsra teljesül, hogy a jobb és bal részfájának magassága legfeljebb eggyel tér el (például a 3 bal fájában 2 szint van, jobb fájában pedig 3, stb.):



Az alábbi fák azonban nem AVL-fák, mert az 5 jobb fájában kettővel több szint van, mint a bal fájában:



Be lehet látni (de ez túlmutat a tárgy keretein), hogy egy n csúcsú AVL-fában $O(\log n)$ szint van és azt is meg lehet mutatni, hogy ha egy beszúrás vagy törlés után megsérül az AVL-tulajdonság, akkor $O(\log n)$ lépésssel a fa átalakítható úgy, hogy újra AVL-fa legyen.

A bináris keresőfák és az AVL-fák műveleteinek szemléltetésére érdemes megnézni a https://visualgo.net/en/bst?slide=1 linken található animációkat, ebből egy kicsit látszik, hogy hogyan lehet visszaállítani az AVL tulajdonságot, amennyiben elromlana törlés vagy beszúrás során.

Az AVL-fa csak egy lehetőség az úgynevezett kiegyensúlyozott keresőfa megvalósítására. Akkor beszélünk kiegyensúlyozott bináris keresőfáról, ha olyan extra tulajdonságot írunk elő a keresőfa tulajdonságon felül, ami garantálja az $O(\log n)$-es szintszámot és amely tulajdonság $O(\log n)$ munkával karbantartható törlés és beszúrás után. Az AVL-fa esetén ez az extra előírás az AVL-tulajdonság volt, ami garantálta az $O(\log n)$-es magasságot és így az $O(\log n)$-es lépésszámot a keresésre, beszúrásra, törlésre.

Hash

Az előbb látott AVL-s megoldással el tudjuk érni, hogy a keresés, beszúrás és törlés lépésszáma n tárolt elem esetén $O(\log n)$ legyen. Most egy olyan adatszerkezetet fogunk tanulni, amiben a legrosszabb esetben a műveletek lépésszáma n, de átlagos esetben (jól megválasztott paraméterek esetén) konstans lépéssben lehet keresni, beszúrni és törölni.
Az új adatszerkezetre motivációul az alábbi példa szolgálhat: Tegyük fel, hogy a BME hallgatóiról szeretnénk adatok tárolni (pl. TAJ-szám, adószám, név, születési idő, születési hely, stb.) úgy, hogy létrehozok egy óriási tömböt, megindexelve 000 000 000 = 0-tól kezdve 999 999 999-ig és minden hallgató adatát a TAJ-számának megfelelő cellában tárolom. Így nagyon könnyen tudok keresni, mert a tömbben egy lépésben bármely cellát ki tudok olvasni az index, azaz a TAJ-szám ismeretében és beszúrni, törölni is hasonlóan könnyű.

Mi a baj ezzel a megoldással? Az, hogy 10^{10} cellát használ, pedig csak 25 000 hallgatóról akarok adatokat tárolni, vagyis nagyon pazarló. Ez a gondolat, amit itt láttunk ugyanaz, mint amit a ládarendezésnél hsználtunk és már ott is megbeszéltük, hogy ezt akkor érdemes csinálni, ha az előforduló értékek tartománya nem túlságosan nagy a tárolandó elemhalmaz méretéhez képest.

Módosítjuk hát a fenti ötletet a következőképpen: a számokat, amiket tárolni akarunk egy függvény segítségével (úgy hívjuk ezt, hogy hash függvény) leképezzük a $\{0, 1, 2, \ldots, M-1\}$ tartományba, ahol M sokkal kisebb, mint annak a tartománynak a mérete, ahonnan a számok (kulcsoknak is nevezzük őket ebben a szövegkörnyezetben) jönnek. Ez a h hash függvény minden K kulcshoz egy $h(K)$ értéket rendel és azt az elvet fogjuk követni, hogy a K kulcsot egy 0-tól $M-1$-ig indexelt tömb $h(K)$-adik cellájába fogunk rakni.

Tipikus hash függvény (mi ezt fogjuk mindig használni), a $h(K) = K$ maradéka M-mel osztva (itt feltesszük, hogy a K kulcsok pozitív egészek). Például a $h(K) = K$ maradéka 11-gyel osztva hash függvény esetén a $K = 3$-as kulcsot a 3-as, a $K = 21$-es kulcsot a 10-es, $K = 33$-as kulcsot pedig a 0-s cellába rakjuk a 11 méretű, 0-tól 10-ig indexelt tömbben.

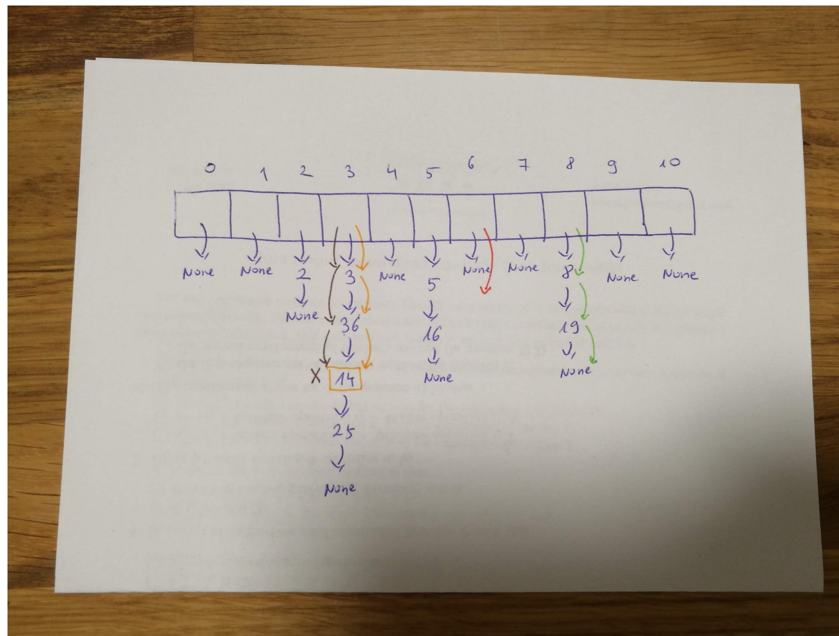
Vödrös hash

A fenti stratégia problémákat vet fel: mivel a tartomány mérete, ahonnan a számok (kulcsok) jönnek sokkal nagyobb, mint M, ezért előfordul úgynevezett ütközés, amikor $h(k_1) = h(K_2)$

két különböző K_1 és K_2 szám esetén, azaz két különböző kulcs szeretne ugyanabba a cellába kerülni. Ezen probléma egyik lehetséges megoldása az ütközések kezelésére a vödrös hash.

Ennél a megoldásnál a cellákban listákat fogunk tárolni úgy, hogy a cellában vagy egy $None$ értékű mutató van (ha egyetlen tárolt elem hash értéke sem egyezik meg a cella indexével) vagy a cellában levő mutató annak a listának az első elemére mutat, ami azokból a tárolt elemekből áll, melyek hash függvény általi értéke a cella indexével egyezik meg.

Az alábbi ábra azt mutatja, hogy hogyan néz ki egy ilyen vödrös hash tábla, ha a $8, 3, 5, 36, 14, 25, 16, 19, 2$ elemek vannak benne és a $h(K) = K$ maradéka 11-gyel osztva hash függvényt használjuk. (A színes nyilakat egyelőre hagyják figyelmen kívül, az majd csak a későbbi példában fog számítani.)



Ha a vödrös hash során az s elemet keressük, akkor először kiszámoljuk $h(s)$-t, majd a $h(s)$ indexű cellában levő listát végigjárva megkeressük az s elemet (hiszen csak itt lehet), ha pedig itt nem találjuk, akkor tudjuk, hogy nem volt a tárolt elemek között. Ezen műveletnek a lépésszáma 1 lépés (feltéve, hogy $h(s)$ kiszámolását 1 lépéssnek tekintjük) és még 1 + annyi lépés, amilyen hosszú a $h(s)$-edik cella listája.

Beszúráskor megint csak úgy kezdjük, hogy kiszámoljuk $h(s)$-t, majd a $h(s)$ indexű cellában levő listát végigjárva megkeressük az s elemet. Ha megtaláljuk, akkor már nem kell beszúrni, mert már szerepel, ha pedig nem szerepel, akkor a lista végére rakjuk, aminek extra költsége a kereséshez képest konstans lépés.

Törléskor megint csak úgy kezdjük, hogy kiszámoljuk $h(s)$-t, majd a $h(s)$ indexű cellában levő listát végigjárva megkeressük az s elemet. Ha nem találjuk, akkor nincs tennivaló, ha pedig megtaláljuk, akkor konstans sok mutató állításával kitöröljük a listából (ahogy múlt órán a listából való törlést csináltuk), az extra költség a kereséshez képest megint csak konstans lépés.

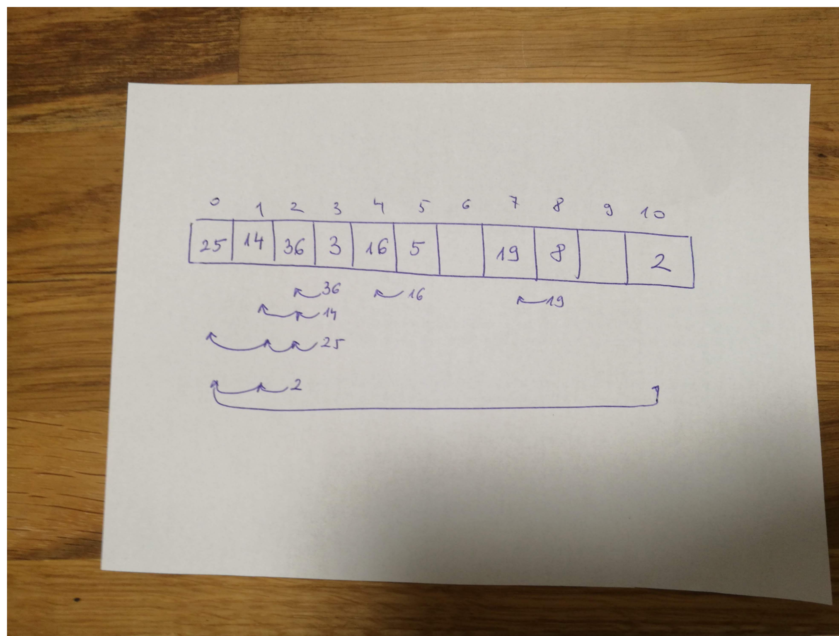
Az ábrán a narancssárga út mutatja, hogy mit járunk végig Keres(14) során, a barna út a Töröl(14)-hez tartozik, a piros út a Keres(6)-hoz, a zöld pedig a Keres(30)-hoz.

Láttuk, hogy a műveletek lépésszáma a hash tábla celláiban levő listák hosszától függ. Ez rossz esetben lehet n (ha minden elem ugyanabban a listában van, ugyanabban a cellában), de ha jól választjuk a hash függvényt (egyenletesen szórja szét a kulcsokat a cellákba) és jól válaaztjuk

meg M értékét, akkor egy listában csak n/M elem lesz várhatóan, ami pl. $M = 1.2n$ esetén várhatóan legfeljebb 1, azaz ekkor a műveletek lépésszáma konstans.

Nyílt címzés, lineáris próba

Az ütközések feloldására a másik lehetőség az úgy nevezett nyílt címzés, aminek egy speciális esetét, a lineáris próbát fogjuk nézni. A nyílt címzés esetén egy cellában csak egy elem állhat, nem használunk listákat, ha pedig egy elem cellája (ahova a h hash függvény alapján kerülne) már foglalt, akkor a táblán belül próbálunk neki újabb helyet keresni. A lineáris próba az a stratégia az új hely keresésére, hogy ha a K kulcs beszúrásakor a $h(K)$-adik cella már foglalt, akkor balra kezdünk lépkedni, amíg alkalmas helyet nem találunk a K kulcsnak.
Az alábbi ábra azt mutatja, hogy hogyan néz ki egy ilyen hash tábla, ha a $8, 3, 5, 36, 14, 25, 16, 19, 2$ elemek vannak benne és a $h(K) = K$ maradéka 11-gyel osztva hash függvényt használjuk nyílt címzéssel, lineáris próbával.



A tábla alatt levő nyilacskák azt mutatják, hogy pl. a 25 beszúrása úgy zajlott, hogy először a 3-as, aztán a 2-es, 1-es cellába akartuk rakni, de ezek foglaltak voltak, így végül a 0-s cellába került. A 2-es elem a 2-es, 1-es és 0-s cellába se fért be, ekkor a próbálkozást a tömb végén folytattuk.

Kicsit pontosabban ez történik: egy cella állapota háromféle lehet:

- foglalt: van benne elem
- üres: nincs benne elem és nem is volt soha korábban
- törölt: nincs benne elem, de volt korábban

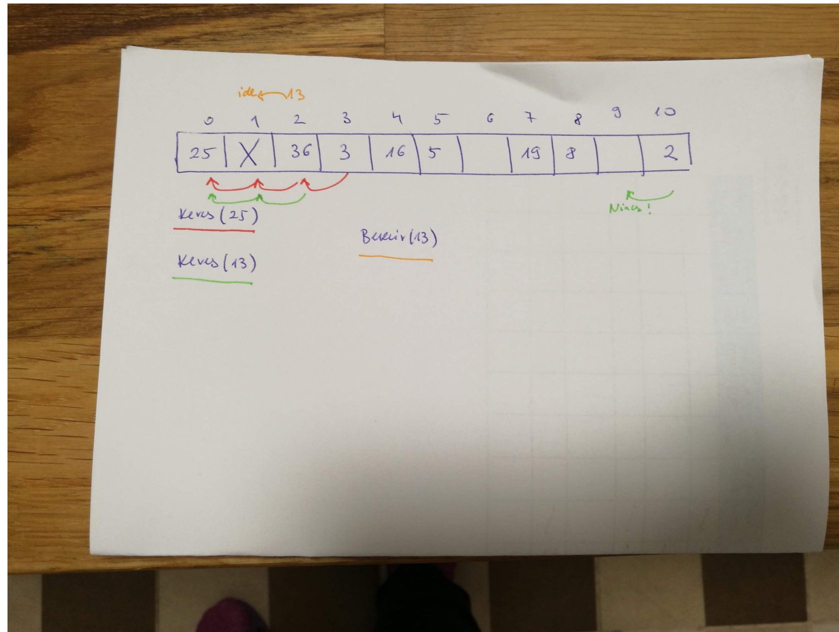
A K kulcs beszúrása úgy zajlik, hogy kiszámoljuk először $h(K)$-t, aztán megnézzük a $h(K)$ indexű cellát és ha az üres vagy törölt, akkor oda berakjuk K-t, ha pedig foglalt, akkor balra indulunk, amíg üres vagy törölt cellát nem találunk, ahol K elfér. Ha körbeérünk, de nem volt ilyen cella (minden hely foglalt), akkor a beszúrás nem lehetséges.

A K kulcs keresése úgy zalik, hogy kiszámoljuk először $h(K)$-t, aztán megnézzük a $h(K)$ indexű cellát, hogy ott van-e a K kulcs. Ha igen, akkor megtaláltuk. Ha ez a cella üres, akkor biztosan

nincs a táblában K (hiszen itt lennie kéne valaminek vagy a cellának töröltnek kellene lennie, ha esetleg korábban volt itt elem). Ha a cella foglalt vagy törölt, akkor tovább indulunk balra, amíg vagy megtaláljuk K-t, vagy üres cellát találunk (ekkor K nincs a táblában). Amíg a cella foglalt vagy törölt, addig megyünk tovább balra. Ha körbeérünk, de K-t nem találtuk meg, akkor K nincs a táblában.

A K kulcs törlése úgy zajlik, mint a keresés, csak a végén a megtalált cellát töröltre állítjuk.

Az alábbi ábra azt mutatja, hogy a korábbi táblából a 14 törlése után kapott helyzetben hogyan zajlik a 25 keresése, a 13 keresése és 13 beszúrása.



Legrosszabb esetben egy művelet lépésszáma lehet itt is n, mint a vödrös hash esetén, ha például a tábla tele van, de meg lehet mutatni (de ez túlmutat a tárgy keretein), hogy rafináltabb nyílt címzési stratégiák esetén (azaz amikor nem lineáris próbával, hanem más módszerrel keresünk új helyet az elemnek), jól megválasztott M érték és hash függvény esetén a műveletek várható lépésszáma legfeljebb 4, amennyiben a tábla legfeljebb 80%-ban van kitöltve. Ez azt jelenti, hogy ekkor ugyan előfordulhatnak hosszabb műveletek, sok lépéssel, de átlagosan legfeljebb 4 lépésben vége lesz egy keresésnek, beszúrásnak vagy törlésnek.

Az AVL-fák és a hash összehasonlítása

Két különböző megoldást láttunk arra, hogy gyorsan keressünk, beszúrjunk és töröljünk elemeket, az AVL-fát és a hash-t. Most összehasonlítjuk a két módszert több szempontból:

1. **A legrosszabb eset lépésszáma** n tárolt elem esetén: az AVL-fánál minden művelet $O(\log n)$ lépésszámú, a hash azonban lehet n lépés is, bármelyik változatát is tekintjük.

2. **Az átlagos lépésszáma** n tárolt elem esetén: az AVL-fánál az átlagos lépésszám is $O(\log n)$ (ezt nem láttuk, de igaz), a hash azonban jól megválasztott paraméterek esetén, legfeljebb 80%-ban kitöltött tábla esetén átlagosan konstans lépés.

3. **Támogatott műveletek**: az AVL-fában (ahogy minden bináris keresőfában) tudunk maximumot és minimumot keresni, de ez a hash esetén nem lehetséges.

4. **A tárolt elemek számának növekedése**: ha az elején rosszul lőjük be, hogy várhatóan mennyi kulcsot akarunk tárolni majd és emiatt rosszul választjuk meg a hash tábla méretét, akkor a nyílt címzés esetén betelik a tábla és nem tudunk tovább dolgozni vele, a vödrös hash esetén pedig nagyon hosszú listák alakulnak ki a tömbben. Az AVL-fánál nem kell előre tudni a tárolandó elemek számát, nem baj, ha egyre több elem lesz, a fa igazodni tud ehhez.

5. **Rendezés**: az AVL-fa (mint minden bináris keresőfa) segítségével gyorsan tudunk rendezni: a fa elemeit inorder bejárással kiolvasva az elemeket rendezetten kapjuk meg. A hash nem segít abban, hogy az elemeket rendezetten kiolvassuk.