Otsimine

Otsimine *(searching)* tegeleb probleemiga, kuidas salvestada andmed arvuti mällu ja kuidas neid sealt uuesti üles leida. Oluline on organiseerida materjal selliselt, et ta oleks võimalikult kiiresti kättesaadav. Enamasti kasutatakse andmete otsimiseks mingit identifikaatorit, nn **võtit** K *(key)*.

Otsimisülesanne on järgmine: on N kirjet ja nende hulgast on vaja leida üks konkreetne kirje, mille võti vastab otsitavale võtmele K. Vastav kirje tagastatakse või antakse teada, et sellise võtmega kirjet ei olnud.

Igal kirjel on võtmeväli, mille väärtuse järgi otsimine toimub (vrdl sorteerimine). Eeldatakse, et võti on unikaalne, st on N erinevat võtit, et selle järgi kirje üheselt identifitseerida. Seega on otsimine funktsioon argumendiga (võtmega) K.

Otsimine võib olla:

edukas - leiti kirje võtmeväärtusega K

mitteedukas - kirjet võtmeväärtusega K ei leitud

Otsimisülesande jaoks pole enam oluline leitud kirjes olevate andmete töötlemine. Otsimismeetodid peatuvad hetkel, kui kirje võtmega K leitakse või tuvastatakse, et vastav kirje puudub. Küll aga tegeleb otsimine ka kirjete paigutamisega, sest on loogiline, et süsteemselt paigutatud materjali hulgast on alati kergem (efektiivsem) otsida.

Otsimise juures on oluline otsimiseks kuluv aeg, kuid alati ei saa ainult ajast lähtuda. Kui otsimisega kaasneb ka töö kirje paigutamiseks, tuleb valik teha lähtudes erinevate tööde esinemise sagedusest. Aeglasemad variandid on lineaarse ajaga `O(N)` algoritmid. Kiiremad algoritmid küünivad logaritmilise ajani `O(log N)` ja konstantse ajani `O(1)`. Samal ajal nõuavad kiiremad algoritmid lisatööd kirjete paigutamiseks.

Klassifitseerimine

Otsimisi saab klasifitseerida mitmeti.
- Sisemine otsimine – kogu info on korraga mälus
- Välimine otsimine – kogu info pole korraga mälus.

- Staatiline – tabel informatsiooniga ei muutu otsimise käigus ja põhiülesanne on aja vähendamine ilma, et peaks arvestama tabeli muutmisega
- Dünaamiline – tabel informatsiooniga muutub nii elementide lisamise kui ka kustutamise tõttu

- Meetodid, mis tuginevad võtmete võrdlemisel
- Meetodid, mis põhinevad võtmete arvulistel omadustel.

- Meetodid, mis kasutavad tegelikke võtmeid
- Meetodid, mis kasutavad muudetud (transformeeritud) võtmeid.

Otsimismeetodite hulka kuuluvad:
- Järjestikotsimine
- Kahendotsimine
- Kahendotsimispuu
- Paiskmeetod

Järjestikotsimine

Järjestikotsimine (ka jadaotisimine, *sequaential search*) on kõige lihtsam otsimismeetod.

Idee: alusta algusest ja võrdle iga kirje võtit otsitava võtmega jätka nii kaua, kuni on leitud otsitava võtmega kirje, seejärel peatu. Või jõua peale kõigi kirjete läbivaatamist arusaamisele, et sellise võtmega kirje puudub.

Järjestikotsimine on kõige lihtsam. Ta saab lõppeda nii edukalt kui ka edutult, sõltuvalt sellest, kas võti oli tabelis olemas või mitte. Algoritmi keerukus on lineaarne (O(N)). Keerukus parimal ja halvimal juhul (otsitav on esimene vs otsitavat ei leitud) erinevad üksteisest oluliselt. Mingit eeldust selle kohta, kuidas võtmed tabelis eelnevalt paiknevad, ei ole. Nimetatud meetod töötab hästi massiivil, kuid teda võib sama hästi kasutada ka ahela korral. Suuremate andmehulkade ja korduva

otsimise korral ei ole seda meetodit siiski mõistlik kasutada

Funktsioon on järgmine – parameetriteks on massiiv, kust otsitakse, massiivi suurus ja otsitav võti, väljundiks on otsitava elemendi indeks või -1, kui elementi ei leitud. Lihtsustuse mõttes koosneb massiiv täisarvudest, mis on ka ühtlasi võtmed.

```
int jadaotsing(int arvud[MaxN], int N, int otsitav) {
//    arvud – massiiv, kust otsitakse
//    N – elementide arv massiivis
//    otsitav – väärtus, mida otsitakse
//     Funktsioon tagastab -1, kui otsitavat ei leitud ja indeksi kui leiti
   int i, leitud;
   leitud = -1;
   i = 0;
   while (i < N && leitud == -1) {
       if (otsitav == arvud[i]) {
           leitud = i;
       }
       i++;
   }
  return leitud;
}
```

Kahendotsimine

Kahendotsimise *(binary search)* aluseks on teadmine, et andmed (võtmed) on järjestatud: $k_0<k_1<k_2< \ldots <k_n$.

Peale otsitava võtme `k` ja `ki` võrdlemist võime väita ühte järgnevast:

$k < k_i$ – võtmed $k_{i+1} \ldots k_n$ langevad vaatluse alt välja

$k = k_i$ – hurraa, otsitav on leitud!

$k > k_i$ – võtmete $k_1 \ldots k_{i-1}$ hulgast pole vaja otsida.

Kahendotsimine arvestabki eelnevaga. Massiiv jagatakse pooleks, keskel olevat võtit võrreldakse otsitava võtmega. Kui otsitav võti on sellest väiksem, võib massiivi ülemise poole kõrvale jätta. Edasi jagatakse pooleks massiivi alumine pool jne.

Kahendotsimist sobib kasutada massiivi jaoks, kus on kerge indeksi järgi leida nö keskmist kirjet.

Funktsioon on järgmine:

```
int kahendotsing(int arvud[MaxN], int N, int otsitav) {
// arvud – massiiv, kust otsitakse
// N – elementide arv massiivis
// otsitav – väärtus, mida otsitakse
// Tagastab -1, kui otsitavat ei leitud ja vastava indeksi kui leiti
    int al,yl;       // vaadeldava vahemiku alumine ja ülemine indeks
    int kesk;
    al = 0;
    yl = N-1;
    while (al <= yl) {
        kesk = (yl + al)/2;
        if (arvud[kesk] == otsitav) {
            return kesk;
        }
        else {
            if (otsitav < arvud[kesk]) yl = kesk - 1;
            else al = kesk + 1;
        }
    }
     return -1;
}
```

Selle meetodi keerukus on `O(log n)`, seega on tegemist päris hea meetodiga. Kui aga on tarvis massiivist vaid ühte võtit üks kord otsida, siis pole selle jaoks mõtet kasutada ruutkeerukusega sorteerimisalgoritmi, et hiljem kiiremini otsida saaks ja järjestikotsimine on sel juhul piisavalt hea.

Kahendotsimispuu

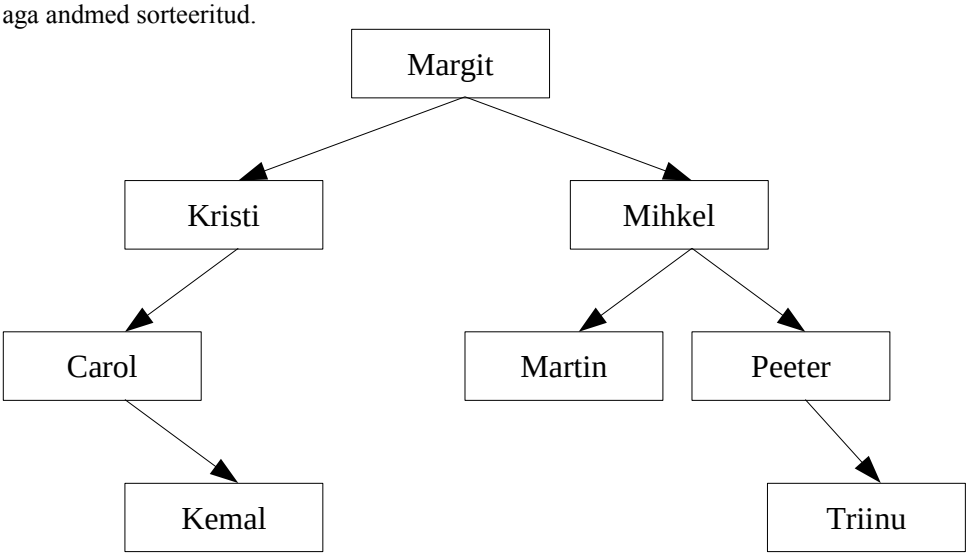
Eespool kirjeldatud meetodid on head, kui andmestik, kust otsitakse, on staatiline ja esitatud massiivina. Kui aga informatsiooni on vaja tihti lisada ja kustutada, ei ole massiiv selleks sobiv andmestruktuur. Andmete nihutamise vajadus igal lisamisel ja eemaldamisel (et säilitada andmete sorteeritus) muudab töö aeglaseks. Dünaamiliste andmete korral sobib paremini viitade abil ehitatav kahendpuu, kui dünaamiline struktuur. Kahendpuus saab elemente kiiresti lisada ja kustutada ning saab ka efektiivselt otsida. Et kahendpuu muutuks sobivaks struktuuriks otsimisel, on vaja elemendid puusse paigutada teatud reeglite järgi.

Kahendotsimispuu *(binary search tree)* jaoks pannakse võtmed puusse järgmiste **reeglite** kohaselt.

1. Iga tipu vasakpoolse lapse võti on alati selle tipu võtmest väiksem ehk üldisemalt kõik ühe tipu vasakpoolses alampuus olevad võtmed on sellest tipu võtmest väiksemad
2. Iga tipu parempoolse lapse võti on selle tipu võtmest suurem ehk üldisemalt kõik tipu parempoolses alampuus olevad võtmed on suuremad tipus olevast võtmest.
3. Need kaks reeglit kehtivad iga alampuu kohta.

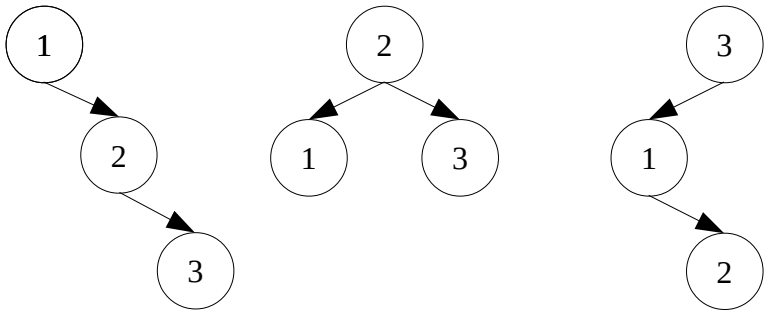
Kahendotsmispuusse võib paigutada suvalisi andmeid, mida on võimalik järjestada. Näiteks võib panna kahendpuusse nimed tähestikulises järjekorras (vt Joonis 1). Pärast on kerge otsida või vajaduse korral elemente (nimesid) lisada.

Sisuliselt toimub otsimisel ja lisamisel **kahendotsimine**. Uute väärtuste lisamine on tunduvalt mugavam kui massiivi kasutades, sest elemente ei pea sorteerimise käigus ümber paigutama (lisatavad elemendid saavad lehtedeks). Sisuliselt on aga andmed sorteeritud.



Joonis 1: Kahendotsimispuu nimedest (Margit, Kristi, Mihkel, Peeter, Martin, Carol, Triinu, Kemal).

Samadest väärtustest saab tekitada mitu erinevat otsimispuud, sõltuvalt väärtuste lisamise järjekorrast. Lisades Joonis 1 olevad nimed näiteks vastupidises järjekorras saame hoopis teise kujuga puu, kuid kahendotsimispuu on ta endiselt. Oletame, et meil on väärtused 1, 2 ja 3. Sõltuvalt sellest, millises järjekorras nad lisatakse, saab tekitada järgmised puud (vt Joonis 2):



Joonis 2: Samadest väärtustest koostatud kahendotsimispuud. Lisamise järjekord: a) 1-2-3, b) 2-1-3 või 2-3-1, c) 3-1-2

Tegevused kahendotsimispuus:

- uue elemendi lisamine
- elemendi kustutamine
- võtme järgi otsimine
- minimaalse ja maksimaalse võtme tagastamine
- puu tasakaalustamine

Uue elemendi lisamine

Uue elemendi lisamiseks tuleb leida talle sobiv koht vastavalt võtmeväärtusele ja seejärel ta uueks leheks lisada (iga uus element saab leheks). Lisamise keerukus on `O(log n)` vastavalt puu kõrgusele `n`.

Algoritm on järgmine:

- Alustades puu juurest võrreldakse iga elemendi võtmeväärtust lisatava elemendi võtmeväärtusega.
- Kui uue elemendi võti on väiksem, jätkatakse liikumist vasakus alampuus
- Kui uue elemendi võti on suurem, jätkatakse liikumist paremas alampuus
- Selliselt toimitakse iga elemendi juures.
- Kui valitud alampuusse enam edasi minna ei saa, sest see puu on tühi, ongi uuele elemendile sobiv koht leitud.

Lisamise funktsioon saab sisendiks puu juure aadressi `T` ja lisatava elemendi aadressi `z`. Igas puu tipus on väljad `key` (võti), `left` ja `right` (vastavalt vasaku ja parema alampuu aadress). Uue elemendi võtmeväljas on võti, väljad `left` ja `right` on väärtustatud `Nil`-idega. Lisaks võib igas sõlmes olla veel üks viidaväli, kuhu kirjutatakse elemendi vanema aadress (nimetatud `p`, *parent*). Algoritm on esitatud pseudokeeles vastavalt CLR-ile.

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x != NIL                  // liigume puus õige leheni
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y                         // väli p on vanema aadress
if y == NIL
    T.root = z                  // puu T oli tühi
elseif z.key < y.key
    y.left = z
```

```
else y.right = z
```

Õige koha leidmiseks läbitakse puus alati üks tee alustades juurest ja lõpetades tühja alampuuga.

Otsimine

Võtme järgi otsimise lahenduseks peab olema vastava tipu aadress või teade sellise tipu puudumisest.

Algoritm on lühike ja lihtne: alustades puu juurest liigutakse vastavalt otsitava võtme väärtusele vasakusse või paremasse alampuusse kuni võti leitakse või kuni jõutakse leheni. Kui tipp leitakse, tagastatakse tipu aadress, kui ei leita, siis `Nil`. Algoritmile on sisendiks võtmeväärtus `k` ja puu juure aadress `T` ning väljundiks tipu aadress `x`. Algoritm on esitatud pseudokeeles vastavalt CLR-ile.

```
Tree-Search(k,T)
    x = T
    while x != NIL and k != x.key do
        if k < x.key
            x = x.left
        else
            x = x.right
    return x
```

Kui võtit `k` ei leita, jõuab `x` leheni ja omandab väärtuse `Nil` (mille abil ka tsükli jätkamist kontrollitakse). Kui võti `k` leitakse katkeb tsükkel varem ja `x`-i sees ongi otsitava tipu aadress.

Sama algoritm rekursiivselt:

```
Tree-Search(x,k)
    if x == NIL or k == x.key
        return x
    if k < x.key
        return Tree-Search(x.left,k)
    else
        return Tree-Search(x.right,k)
```

Väikseim ja suurim võti

Väikseima võtme leidmiseks tuleb liikuda juurest alates mööda puud seni vasakule kui saab. Viimases kättesaadavas tipus ongi väikseim võti.

Suurima võtme leidmiseks tuleb aga liikuda mööda puud paremale kuni `Nil` ette tuleb ja siis on suurim võti käes.

Kumbki tegevus on keerukusega `O(log n)`.

Järgnevad funktsioonid kutsutakse välja nii, et argumendiks antakse puu juure aadress.

```
TREE-MINIMUM(x)
while x.left != NIL
    x = x.left
return x

TREE-MAXIMUM(x)
while x.right != NIL
    x = x.right
return x
```

Tipu kustutamine

Tipu kustutamine on sarnaselt tipu lisamisega oluline kahendpuus tehtav töö. Kustutamine algab otsimisest. Kui vastav tipp on leitud, siis on tema kustutamisel 3 võimalust:

1) kui tipul pole lapsi, võime tipu lihtsalt ära kustutada

2) kui tipul on üks laps, paigutatakse see kustutatava tipu asemele

3) kui tipul on mõlemad lapsed olemas, otsitakse tema asemele talle suuruselt järgnev tipp x, mille vasak alampuu on tühi; füüsiliselt eemaldatakse tipp x (eelmise algoritmi kohaselt) ja tipus x olnud andmed kirjutatakse kustutatava tipu asemele. Suuruselt järgmine tipp peaks paiknema kustutatava tipu paremas alampuus kõige vasakul.

Kõigi võtmete väljastamine

Kahendpuu läbimiseks oli kirjeldatud kolm võimalikku järgnevust. Kasutades läbimiseks **inorder**-järjekorda (vasakpoolne AP, juur, parempoolne AP) saame võtmed kätte sorteeritult ehk kasvavas järjekorras. Vastav algoritm on järgmine:

```
INORDER-TREE-WALK.x/
if x != NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```

Korduvad võtmed

Üldisel eeldatakse, et otsimine toimub unikaalsete võtmete järgi. Mõnikord võib kahendotsimispuu olla siiski vajalikuks struktuuriks korduvate võtmeväärtuste korral. Lahendus sõltub sellest, milleks me puud kasutame. Sõltuvalt ülesande iseloomust on kolm võimalust:

1. Igasse sõlme lisatakse loendur, mis näitab mitu sellist väärtust on olnud.
2. Uute tippude lisamisel kasutatakse "väiksem"-suhte asemel "väiksem või võrdne"-suhet, seega sama väärtusega element pannakse vasakpoolsesse alampuusse.
3. Ei lasta ennast häirida ja korduva võtme tekkimisel ei lisata uut tippu.

Tasakaalustatud puu

Üldiselt on kahendotsimispuust otsimine logaritmilise keerukusega tegevus. Logaritmiline keerukus on tagatud vaid juhul, kui puu tasemed on maksimaalselt täidetud. Sammude arv sõltub väga otseselt puu kõrgusest. Tegelik headus sõltub sellest, millises järjekorras võtmed saabuvad, sest selle järgi ka puu tekib. Milline on halvim variant? Kindlasti see, kus võtmed „saabuvad" järjestatult. Sel juhul saadakse sisuliselt lineaarne loend (ahel) ja otsimise keerukus on lineaarne, kuid tegelikult aeglasemgi, sest tehakse palju ülearust võrdlemistööd võrreldes tavalise järjestikotsimisega.

Kui puu sõlmed on võimalikult ühtlaselt jaotunud, kõik tasemed on enam-vähem täis, siis on teed puu lehtedeni võrdse pikkusega ja aeg tõepoolest logaritmiline. Kuidas hoida otsimispuud heas seisundis?

Kui puu pannakse kokku juhuslike väärtustega võtmetest on alust küll loota, et halvimat varianti (lineaarset loendit) ei teki, kuid optimaalset olekut ei pruugi samuti tekkida.

AVL-puu

Hea lahenduse puu optimaalses seisundis hoidmiseks pakkusid 1962. aastal välja teadlased *G. M. Adelson-Velskii* ja *E. M. Landis*. Nende meetod vajab küll iga tipu jaoks veidi täiendavat mälu, kuid garanteerib see-eest alati logaritmilise otsimisaja. Oma leiutajate auks kannab see puu **AVL-puu** *(AVL-tree)* nime.

Puu on tasakaalustatud juhul, kui tema iga alampuu jaoks vasaku ja parema alampuu kõrguste vahe pole suurem kui üks.

Tasakaalustatud kahendpuu tagab logaritmilise aja nii võtme järgi otsimiseks, elemendi lisamiseks kui ka elemendi eemaldamiseks. Siiski on tema paremus tuntav siis, kui tippe on puus palju (vähemalt 256) ja puus on vaja tihti võtme järgi otsida, lisada ja kustutada.

Tasakaalustamine

AVL-puu saamiseks on peale iga sõlme lisamist või kustutamist vaja kontrollida tema tasakaalustatust ja vajaduse korral puu tasakaalu viia sõlmi ümber paigutades.

Puu tasakaalustamiseks seotakse iga sõlmega tasakaalu faktor (balancing factor), mille abil iseloomustatakse iga tipu vasaku ja parema alampuu kõrguste vahet. Faktoril saab olla kolm erinevat väärtust, mis kirjeldavad sõlme olukorda:

- sõlme vasak alampuu on 1 võrra kõrgem (-1)

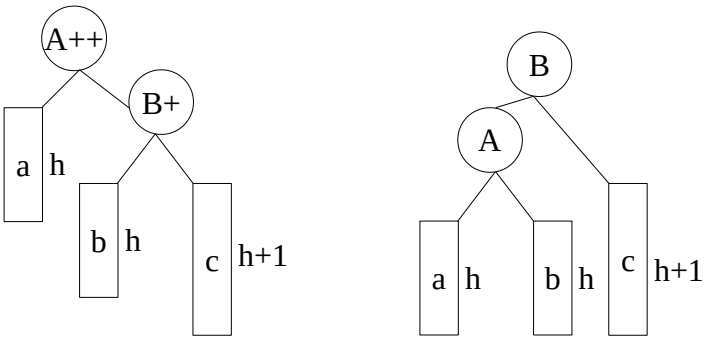
+ sõlme parem alampuu on 1 võrra kõrgem (+1)

* sõlme mõlema alampuu kõrgused on võrdsed (0)

Puu jaoks kriitilised on sõlmed faktoritega -1 ja +1 ga. Kui esimesel juhul lisada sõlm vasakusse alampuuse ja teisel juhul paremasse alampuusse, läheb puu vastava sõlme kohalt tasakaalust välja.

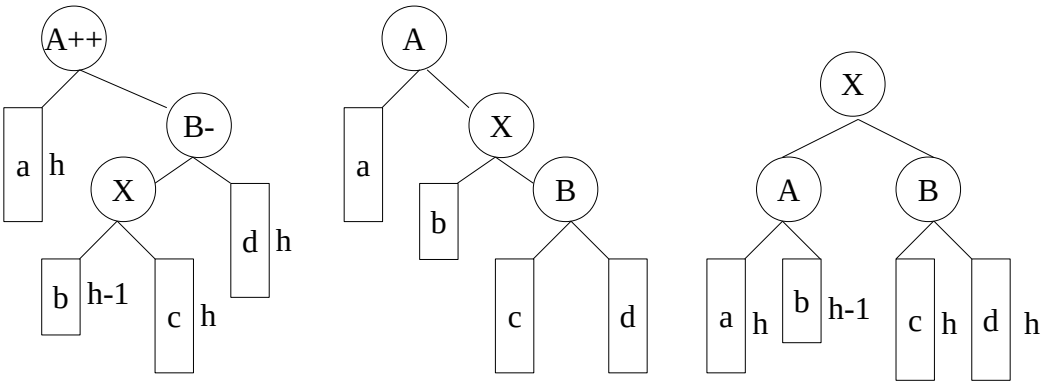
Eristatakse kahte erinevat varianti tasakaalust välja minekul (ja vastavaid tasakaalustamise algoritme) + peegeldused:

a) sõlmel A on juba ühe võrra kõrgem parem alampuu (faktor +1), parema alampuu (millel juureks B) paremasse alampuusse lisatakse sõlm, sellega muutub ka sõlme B faktor +1 ja sõlme A kohalt on puu tasakaalust väljas. Sarnane juhus tekib juhul, kui eelnevalt on sõlme A faktor -1 ja tema vasakusse alampuusse tipuga B lisatakse vasakule juurde tipp. Selle tagajärjel saab tipu B faktoriks -1 ja tipu A kohalt on puu tasakaalust väljas (vt Joonis 3).
b) sõlmel A on ühe võrra kõrgem parem alampuu (faktor +1), tema paremasse alampuuse tipuga B lisatakse vasakule poole element, nii et tipu B jaoks tasakaalufaktor muutub -1. Sarnaselt: tipul A on eelnevalt faktor -1 ja tema vasakusse alampuusse tipuga B lisatakse paremle juurde tipp, mille tulemusena tipu B faktoriks saab +1 ning puu on A kohal tasakaalust väljas (vt Joonis 4).



Joonis 3: Puu tasakaalustamise esimene variant

Selgitus esimesele tasakaalustamise variandile (Joonis 3). Puu on tasakaalust väljas kaks korda ühele poole. Enne tipu lisamist on tipu A tasakaalufaktor +. Tipu B tasakaalufaktor saab peale tipu lisamist +. Puu on tipu A kohalt tasakaalust väljas. Tasakaalustamiseks pööratakse alampuud tipuga A vasakule. B saab uueks alampuu juureks ning tipu B vasak alampuu saab A paremaks alampuuks.



Joonis 4: Puu tasakaalustamise teine variant

Selgitus teisele tasakaalustamise variandile (Joonis 4). Puu on tasakaalust väljas A juures paremale ja B juures vasakule. Enne tipu lisamist oli tipu A tasakaalufaktor +. Peale tipu lisamist saab tipu B tasakaalufaktoriks - ja puu on A kohalt tasakaalust väljas. Tasakaalustamiseks tehakse 2 pööret: kõigepealt tipu B kohal paremale (B asemel saab alampuu juureks tema vasak järglane X ja tekib 1. variant) ja seejärel tipu A juures vasakule (uueks juureks saab X).

Puna-must puu

Puna-must puu *(red-black tree)* on puu, kus kasutatakse ühte lisabitti määramaks tipu värvi: punane või must. Arvesatdes värve ja puna-musta puu reegleid, ei saa ükski tee puu juurest leheni olla üla kahe korra pikem võrreldes ükskõik millise teise samas puus oleva teega. Võib väita, et puu on ligikaudu tasakaalus *(balanced)*. Tehnika pakkus esimest korda välja R. Bayer. Pikemalt töötasid temaatika läbi ja andsid puule nime L. J. Guibas ja R. Sedgewik.

Puna-must puu on kahendotsimispuu, mis täidab **puna-musti omadusi**:

1. Iga sõlm on kas punane või must.
2. Puu juur on must.
3. Lehtede `NIL`-viidad (ehk tühjad alampuud) on mustad.
4. Kui sõlm on punane, on tema mõlemad lapsed mustad..
5. Iga tipu jaoks kõik temaga algavad teed kuni lehtedeni sisaldavad ühepalju musti sõlmi.

Kolmas ja neljas reegel tagavad selle, et puu on otsimiseks mõistlikus seisus. Tänu nendele reeglitele pole ükski tee juurest leheni üle kahe korra pikem kui mistahes teine tee. Kui mustal tipul võib olla ka must järglane või vanem, siis punasel tipul tohivad olla vaid mustad lapsed ja must vanem, seega liikudes puu juurest leheni ei saa kohata järjest kahte punast tippu.

Tänu kirjeldatud omadustele ei ületa n tipuga PM-puus **otsimise aeg** `2*log(n+1)`**.**

3. reegliga peab arvestama puu reegleid rikkuvate tippude uurimisel ja puu korrastamisel.

Lisaks tavalisele puu kõrgusele räägitakse ka PM-puu **mustast kõrgusest** (*black hight)* - see on mustade tippude arv puu juurest leheni (mustad tühjad alampuud kannavad numbrit 0). Puu must kõrgus on iga lehe suhtes ühesugune (5. reegel).

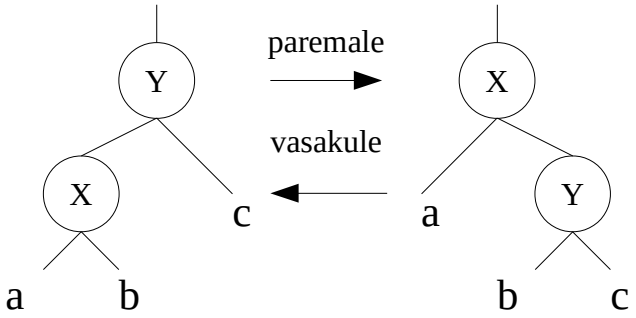
Punamusta puu korral toimub puust teatud **võtmega tipu otsimine**, samuti **vähima** ja **suurima elemendi leidmine** sarnaselt tavalisele kahendotsimispuule (keerukus kõigi tööde jaoks `O(log n)`). Uue elemendi lisamisel on erinevused.

Elemendi lisamine

Et tegemist on kahendotsmispuuga, siis uue elemendi lisamine puusse algab samamoodi nagu juba varem kirjeldatud. Edasi tuleb aga kontrollida, kas puna-mustad omadused on jätkuvalt tääidetud ja kui ei ole, tuleb hakata puud korrastama, ehk sõlmede asukohti muutma. Tippude suhtes tehakse AVL-puuga sarnaseid pöördeid. Esimese kordategemise tagajärjel võivad omadused mitte kehtida järgmises kohas ja nii “parandatakse” olukorda mööda puud ülespoole liikudes, kuni jälle kehtivad kõik puna-mustad omadused.

Puu korrastamiseks kasutatakse kolme operatsiooni:

- osade tippude värvimine - punane tipp värvitakse mustakse ja must punaseks;
- pööre vasakule - tipu X parem laps saab uueks (alam)puu juureks ning X ise satub tema vasakuks lapseks (vt Joonis 5 ja ka pööramine AVL-puus);
- pööre paremale - tipu X vasak laps saab uueks (alam)puu juureks ning X ise satub tema paremaks lapseks (vt Joonis 5 ja ka pööramine AVL-puus).



Joonis 5: Puu pööramine tippude X ja Y kohalt paremale ja vasakule.

Uus tipp lisatakse leheks nagu tavalise kahendotsimispuu puhul ja värvitakse esialgu punaseks. Kui tipp lisatakse punase tipu külge, siis lisatud tipp rikub puu reegleid. Järgneb PM-puu taastamiseks tsükkel mööda puud üles, kuni enam reegleid ei rikuta. On kolm erinevat juhust (+ kolm peegeldatud juhust), millele vastavalt tegutseda tuleb.

1. Lisatud (reegleid rikkuva) tipu isa ja onu on punased:

Lisatud tipu isal, onul ja vanaisal muudetakse värv (vanaisa punaseks, isa ja onu mustaks).

2. Reegleid rikkuva tipu isa on punane ja onu on must. Lisatud tipp on oma isale paremaks lapseks, isa aga vasakuks lapseks. Reegleid rikkuva tipu ja tema punase vanema juures tehakse pööre vasakule. Tavaliselt viib see 3. juhuse tekkimisele.

3. Reegleid rikkuva tipu isa on punane ja onu on must. Nii lisatud tipp kui ka isa on vasakud lapsed.

Kõigepealt tehakse pööre paremale punase isa ja musta vanaisa suhtes. Seejärel värvitakse ringi punane isa ja must vanaisa ning puna-mustad omadused on taastatud.

Kui pööramiste tulemusena puu juur läheb punaseks, siis juur värvitakse mustaks.

Siinkohal soovitan pöörduda veebiaadressile, kus on mängimiseks animatsioon:

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

B-puu

B-puu *(B-tree)* on teatud tüüpi tasakaalustatud puu, mis on sobiv info hoidmiseks kettal, kus toimub otsepöördumine. Hoides infot kettal pole nii oluline arvutusteks kuluv aeg, kui aeg, mis kulub kettaga suhtlemiseks (lugemiseks ja kirjutamiseks). Suurema osa pöördumisajast *(access time)* võtab lugemispea nihutamine õigele rajale ja sektorile, sektor loetakse tavaliselt korraga. Hilisem töötlemine on reeglina kiirem. Seega vajab optimeerimist pöördumiste arv. Mida vähem tuleb järgmise sektori poole pöörduda õige sektori leidmiseks, seda parem.

B-puu puhul on s/v operatsioonide arv proportsionaalne puu kõrgusega, seepärast tuleb puud hoida madalana. Ja ühe tipu järglaste arvu ei piirata 2-ga, vaid praktiliselt võib see ulatuda ka 1000-sse. Seega on puu kõrgus hinnaguliselt palju väiksem kui suvalise tasakaalustatatud kahendotsimispuu puhul.

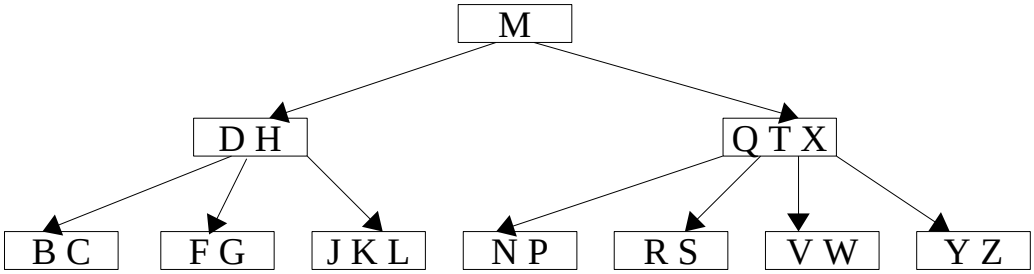
Reeglid B-puu jaoks, mille järk on t:

1. Igas tipus x on väljad, milles hoitakse:
   a) tipu võtmete arvu n{x],
   b) võtmeid endeid mittekahanevas järjekorras k1[x]<=k2[x]<= ...<=kn[x],
   c) tähist, kas tipp on leht (loogikaväärtus)
2. Kui `x` on sisemine tipp, on temas `n[x]` elementi ja `n[x]+1` viita lastele.
3. Kuna lehtedel lapsi pole, pole nende jaoks ka viidaväljasid ettenähtud

4. Tipus olevad võtmed (n[x] tükki) on piirideks, mille järgi jaotatakse võtmed alampuude vahel.
5. Kõik lehed on samal tasemel
6. Ühes tipus säilitatavate võtmete arv on piiratud nii ülalt kui ka alt ja see on ühesugune kogu `t>=2` järku puu tippude jaoks. T `(t>=2)` on B-puu **minimaalne järk** *(minimum degree)*:
   a) igas tipus (va juures) on vähemalt `t-1` võtit, st sisemistel tippudel on vähemalt `t` last. Kui puu pole tühi, peab juurelemendis olema vähemalt üks võti.
   b) igas tipus ei ole rohkem kui `2t-1` võtit, st sisemisel tipul ei ole rohkme kui `2t` last. Tipp on **täis** *(full)* kui temas on `2t-1` võtit.

Lihtsaimal juhul on t=2-ga, Siis saadakse teatud puu tüüp – **2-3-4-puu** *(2-3-4 tree)*. Praktikas soovitatakse t võtta suurem kui 2.

Tipus `X` säilitatakse `n[X]` võtit. Nende võtmete järgi jagatakse lapsed `n[X]+1` gruppi. Seega on tipul `X n[X]+1` järglast. Otsimisel võrreldakse otsitavat võtit tipus `X` olevate võtmetega ja valitakse jätkamiseks üks võimalikest teedest, vastavalt sellele, millisesse võtmete abil määratud vahemikku otsitav võti kuulub.



Joonis 6: B-puu, kuhu on paigutatud kaashäälikud.

Tipp võtmetega D ja H jaotab kaashäälikud kolme gruppi: BC (kuni D-ni), FG (D kuni H), ja JKL (H kuni M, sest M jaotab võtmed puu juures) (vt Joonis 6).

Algoritmid B-puude jaoks hoiavad põhimälus vaid osa informatsiooni ja seepärast ei ole puu suurus piiratud põhimälu suurusega. Kuna lugemine on seotud sektori suurusega, on hea, kui B-puu tipp katab täpselt ühe ketta sektori. Mida suurem on hargnemine igas tipus (suurem on puu järk), seda madalam puu tekib, vähem on vaja sõlmi kettalt lugeda ja seda kiiremini otsimine toimub.

Otsimine B-puus

Otsimine B-puus on sarnane otsimisele kahendpuus. Vahe on selles, et valida tuleb tohkem kui kahe `(n[x]+1)` lapse vahel. Otsimisel vaadatakse juurest alustades igas tipus `x` läbi võtmed. Kui võti `k` leitakse, tagastatakse viide tipule ja võtme järjekorranumber antud tipus. Igas tipus vaadatakse võtmed järjest läbi ja peatutakse vähima `i` juures, mille jaoks $k<=key_i[x]$. Kui sellist ei leita, siis saab `i` väärtuseks `n[x]+1`. Kui otsitav võti `k` leiti, siis töö lõppeb, vastasel juhul programm peatub (kui uuritav tipp on leht) või kutsub enda rekursiivselt välja, olles eelnevalt kettalt mällu lugenud sektori järgmise tipuga, millest otsimist jätkata.

Võtmete lisamine B-puusse

Võtmete lisamine B-puuse on keerulisem operatsioon, kui võtme lisamine kahendotsimispuusse. Võtmete lisamise käigus võib juhtuda, et mõnda tippu saab rohkem võtmeid kui lubatud (rohkem kui `2t-1` võtit). Sel juhul tuleb tipp poolitada. Saadakse 2 tippu, millest kummaski on `t-1` võtit. Seoses sellega tekib võti, mis on poolitajaks (median key), See võti tuleb omakorda lisada poolitatud tipu vanema võtmete hulka. Nüüd on võimalus, et vanema võtmete arv läheb "üle serva". Sel juhul tuleb poolitamisprotseduuri korrata vanema jaoks jne kuni juureni välja.

Võtmete kustutamine B-puust

Võtmete kustutamisel võib tipus olevate võtmete arv langeda alla minimaalse järgu t ja sel juhul tuleb kaks tippu ühendada.

Seoses sellega kaob ka üks võti ühendatud tippude vanem-tipust, kus võib omakorda tekkida “alatäitumine” jne.