Otsingumeetodid

Otsing *(ingl search)* on mingi andmekogumi ühe või enama andmeelemendi uurimine etteantud omadusega elementide leidmiseks. (IT-Terministandardi sõnastik: EVS-ISO 2382-6:1999)

Otsingumeetodid tegelevad probleemiga, kuidas (mis viisil korrastatuna) salvestada andmed arvutis ja kuidas neid sealt uuesti üles leida. Oluline on organiseerida andmed selliselt, et andmeelement oleks võimalikult kiiresti leitav. Enamasti kasutatakse andmete otsimiseks mingit identifikaatorit, nn **otsinguvõtit** ehk **otsivõtit** K *(*ingl *search key)*. **Otsinguvõti** on andmete välja otsimiseks kasutatav võti.

Andmekogumi kirjete hulgast on vaja leida kirje, mille võti võrdub otsinguvõtmega. Leitud kirje tagastatakse (võimalik, et antakse ka lihtsalt teada, et leiti) või antakse teada, et sellise võtmega kirjet ei leitud.

Kirje koosneb paljudest andmetest. Aga igas kirjes eristatakse võtmevälja, mille väärtuse järgi otsing toimub. Tavaliselt eeldatakse, et igal võtmel on unikaalne väärtus, et selle järgi kirje üheselt identifitseerida.

Otsingut nimetatakse **edukaks**, kui leiti otsinguvõtmele K vastava võtmega kirje, ning **mitteedukaks** (**ebaedukaks**) kui ei leitud.

Otsingumeetodis ega algoritmis ei kirjeldata seda, mis saab leitud kirjest edasi või mida tehakse siis, kui kirjet ei leitud. Otsingumeetodid peatuvad hetkel, kui otsunguvõtmele K vastav kirje leitakse või tuvastatakse, et otsitav kirje puudub. Küll aga võib otsingumeetod tegeleda kirjete paigutamisega (või eeldab teatud paigutust), sest on loogiline, et süsteemselt paigutatud materjali hulgast on alati efektiivsem otsida ning kirjete paigutamine ning nende hulgast otsimine on tihedalt omavahel seotud.

Oluline on otsinguks kuluv aeg, kuid alati ei saa ainult ajast lähtuda. Kui otsinguga kaasneb ka töö kirje paigutamiseks, tuleb valik teha lähtudes erinevate tegevuste (andmestiku muutumine vs sealt otsimine) esinemise sagedusest.

Aeglasemad otsingumeetodid töötavad lineaarse ajaga `O(N)`. Kiiremate algoritmide töö küünib logaritmilise `O(log N)` ja konstantse ajani `O(1)`. Samal ajal nõuavad kiiremad algoritmid lisatööd kirjete paigutamiseks.

Klassifitseerimine

Otsingumeetodeid saab klassifitseerida mitmeti ning meetodi valik sõltub paljus andmete käitumisest ja iseloomust.

- Mälukasutuse järgi:
  - siseotsing – kõik andmed on korraga operatiivmälus;
  - välisotsing – kõik andmed pole korraga mälus, osa on kõvakettal.
- Andmete muutumise sageduse järgi:
  - staatiline – andmestik ei muutu kasutuse käigus (eriti) ja põhiülesanne on otsinguaja vähendamine;
  - dünaamiline – andmestik muutub tihti elementide lisamise ja kustutamise tõttu.
- Otsinguvõtmete iseloomu ja kasutamise järgi:
  - kasutab võtmete võrdlemist;
  - kasutab võtmete arvulisi omadusi;

- kasutab tegelikke võtmeid;
- kasutab muudetud võtmeid.

Järgnevalt kirjeldatakse järjestotsingut, kahendotsingust ning puuotsingust. Teksti lihtsustamise huvides ei korrata pidevalt, et otsitakse kirjet teatud võtme väärtusega, vaid kirjutatakse, et otsitakse võtit. Siiski tasub meeles pidada, et võti on vaid üks osa andmestikust ning temaga on alati kaasas veel mitmeid olulisi andmeid. Enamasti eeldatakse, et võtmeväärtused on unikaalsed, kuid mitmeid otsingumeetodeid on võimalik kohandada ka olukorrale, kus võtmete väärtused korduvad.

Järjestotsing

Järjestotsing (ka jadaotsing, lineaarotsing, ingl *sequaential search, linear search*) on kõige lihtsam otsingumeetod.

Järjestotsing on otsing, milles andme kogum skaneeritakse järjestikku. (IT-Terministandardi sõnastik: EVS-ISO 2382-6:1999)

Idee: andmed on jadas, alustatakse jada algusest ja võrreldakse järjest iga kirje võtit otsinguvõtmega; jätkatakse seni, kuni on leitud otsinguvõtmele vastav kirje. Peale kõigi kirjete läbivaatamist võib ka jõuda arusaamisele, et sellise võtmega kirje puudub.

Järjestotsing on kõige lihtsakoelisem otsingumeetod. Algoritmi keerukus on lineaarne `(O(N))`. Keerukus parimal ja halvimal juhul (otsitav on esimene vs otsitavat ei leitud) erinevad üksteisest oluliselt. Mingeid eelduseid kirjete paiknemise kohta andmestikus ei ole. Meetod töötab hästi massiivil, kuid teda võib sama hästi kasutada ka ahelloendist otsimisel. Suuremate andmehulkade ja korduva otsimise korral ei ole järjestotsingut siiski mõistlik kasutada

Järjestotsing funktsioonina: parameetriteks on massiiv, massiivi suurus ja otsinguvõti, väljundiks on otsitava elemendi indeks või -1, kui elementi ei leitud. Lihtsustamiseks koosneb massiiv täisarvudest, mida käsitletakse võtmetena.

```
int SequentialSearch(int data[MaxN], int N, int searchKey) {
/*
 *  data – otsungumassiiv
 *  N – elementide arv massiivis
 *  searchKey – otsinguvõti
 *  Tagastab -1 ebaedukal otsingul ja indeksi edukal otsingul.
 */
   int i = 0, found = -1;
   while (i < N && found == -1) {
       if (searchKey == data[i]) {
           found = i;
       }
       i++;
   }
  return found;
}
```

Kahendotsing

Kahendotsing (ingl *binary search*) on dihhotoomotsing, mis töötleb võrdse andmeelementide arvuga kogumeid, paaritu elementide arvuga algkogumi korral aga lubab ühel kogumil sisaldada üht lisaelementi. (IT-Terministandardi sõnastik: EVS-ISO 2382-6:1999)

Dihhotoomotsing (ingl *dichotomizing search*) on otsing, mille puhul andmeelementide järjestatud kogum jagatakse kaheks teineteist välistavaks osaks, jättes ühe neist kõrvale; protsessi korratakse

allesjäänud osaga, kuni otsing on lõppenud. (IT-Terministandardi sõnastik: EVS-ISO 2382-6:1999)

Kahendotsingu aluseks on eeldus, et andmed on järjestatud võtmete väärtuste järgi: $k_0 < k_1 < k_2 < \ldots < k_n$.

Peale otsinguvõtme k ja andmestiku võtme k_i võrdlemist võime väita ühte järgnevast:

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

kui $k = k_i$ – hurraa, otsitav on leitud;

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

Kahendotsing arvestabki eelnevaga. Andmemassiiv jagatakse pooleks, massiivi keskel olevat võtit võrreldakse otsinguvõtmega. Kui otsinguvõti võrdub massiivis oleva võtmega, on kirje leitud. Kui otsinguvõti on väiksem, võib massiivi ülemise poole edasisest vaatlusest kõrvale jätta. Edasi jagatakse pooleks massiivi alumine pool jne, kuni kirje leitakse või vaatlusse jäänud elemente enam pooleks jagada ei saa.

Kahendotsingut sobib kasutada massiivil, kus on kerge indeksi järgi leida keskmist kirjet. Ehk siis lisaks sorteeritusele peab andmestik võimaldama otsepöördust suvalise kirje poole. Seega näiteks võib välistada kahendotsingu kasutamise ahelloendis, ka siis, kui ta on sorteeritud.

Kahendotsing funktsioonina: parameetriteks on massiiv, massiivi suurus ja otsinguvõti, väljundiks on otsitava elemendi indeks või -1, kui elementi ei leitud. Lihtsustamiseks koosneb massiiv täisarvudest, mida käsitletakse võtmetena.

```
int BinarySearch(int data[MaxN], int N, int searchKey) {
/*
 *  data – otsingumassiiv
 *  N – elementide arv massiivis
 *  searchKey – väärtus, mida otsitakse
 *  Tagastab -1 ebaedukal otsingul ja indeksi edukal otsingul.
 */
    int begin, end, middle; /* vahemiku alumine, ülemine ja keskmine indeks */
    begin = 0;
    end = N - 1;
    while (begin <= end) {
        middle = (end + begin)/2;
        if (data[middle] == searchKey) {
            return middle;
        }
        else {
            if (searchKey < data[middle]) {
                end = middle - 1;
            }
            else {
                begin = middle + 1;
            }
        }
    }
     return -1;
}
```

Kahendotsingu keerukusklass on `O(log n)`, seega on tegemist päris hea meetodiga. Kui aga on tarvis vaid paar korda otsida, siis pole mõtet kiiremaks otsinguks kasutada linearitmilise või lausa ruutkeerukusega sorteerimisalgoritmi. Järjestotsing on piisavalt hea.

Puuotsing

Eespool kirjeldatud meetodid on head, kui andmestik on staatiline (ei muutu tihti) ja on salvestatud massiivina. Kui andmeid on vaja tihti lisada ja kustutada, ei ole massiiv selleks sobiv andmestruktuur. Andmete korrastamise (uue sorteerimise) vajadus igal lisamisel ja eemaldamisel muudab töö ebaefektiivseks. Tihti muutuvate andmete jaoks on vaja sellist struktuuri, mis lubab andmeid efektiivsemalt lisada ja kustutada ning selle juures säilitada andmete järjestatus.

Puuotsing *(*ingl *tree search)* (ka **hargotsing**) toimub puustruktuuris. Otsingu igal sammul saab otsustada, millise osa puust võib edasisest otsingust kõrvale jätta. (IT-Terministandardi sõnastik: EVS-ISO 2382-6:1999)

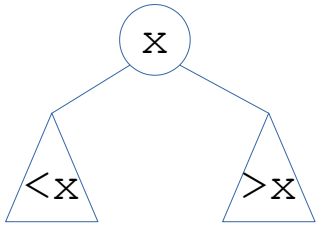
Kahendotsingupuu

Muutuva andmestiku jaoks 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.

Kahendotsingupuus *(*ingl *binary search tree, lühendatult BST)* lisatakse võtmed järgmiste **reeglite** kohaselt:

1. Iga tipu vasakpoolse lapse võti on väiksem tipu võtmest, üldisemalt kõik tipu vasakpoolses alampuus olevad võtmed on väiksemad tipu võtmest.
2. Iga tipu parempoolse lapse võti on suurem tipu võtmest, üldisemalt kõik tipu parempoolses alampuus olevad võtmed on suuremad tipu võtmest.
3. Need kaks reeglit kehtivad iga alampuu kohta.

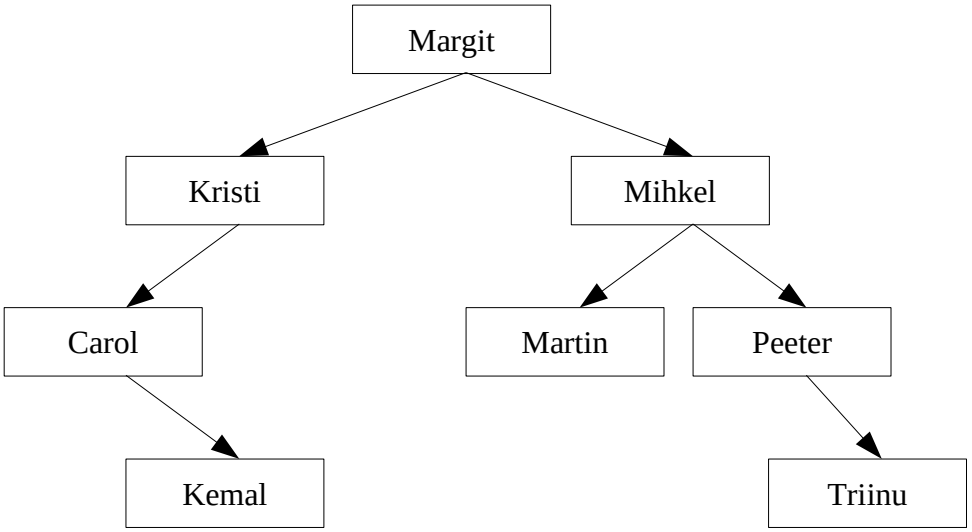
Öeldakse, et kahendotsingupuul on **otsinguomadus** *(*ingl *search property)* ehk **kahendotsingupuu omadus** *(*ingl *BST property)*, mis seisneb eespool toodud reeglites. Väga oluline on sealjuures mõista, et kõik tipu vasakus alampuus olevad võtmed on väiksemad ja kõik paremas alampuus olevad võtmed on suuremad vastava tipu võtmest (vt Joonis 1).



Joonis 1: Kahendotsingupuu reegel

Kahendotsingupuusse paigutavad võtmed võivad olla suvalist tüüpi, oluline, et neid saab järjestada. Näiteks võime käsitleda nimesid võtmetena, kus järjestuse määrab tähestik (vt Joonis 2). Efektiivne on nii otsimine kui ka uute andmete lisamine.

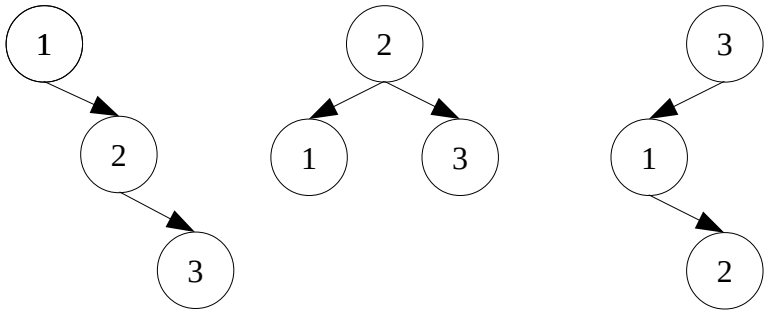
Sisuliselt toimub otsimisel **kahendotsing:** puu tipus olevat võtit võrreldakse otsinguvõtmega ning tehakse otsus vasakusse või paremasse alampuusse liikumiseks. Uue võtme lisamisel otsitakse talle sarnasel viisil sobiv koht ning lisatakse uus võti leheks. Andmed on pidevalt sorteeritud.



Joonis 2: Kahendotsingupuu nimedest (Margit, Kristi, Mihkel, Peeter, Martin, Carol, Triinu, Kemal).

Samadest võtmetest saab sõltuvalt võtmete lisamise järjekorrast tekitada mitu erinevat otsingupuud. Lisades Joonis 2 olevad nimed näiteks vastupidises järjekorras saame hoopis teise kujuga puu, kuid kahendotsingupuu on ta endiselt.

Olgu meil on võtmed 1, 2 ja 3. Sõltuvalt sellest, millises järjekorras nad lisame, tekivad järgmised puud (vt Joonis 3):



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

Tegevused kahendotsingupuus:

- võtme lisamine;
- võtme kustutamine;
- otsinguvõtme järgi otsimine;
- minimaalse ja maksimaalse võtme tagastamine;

- suuruselt eelmise ja järgmise võtme leidmine;
- sorteerimine.

Võtme lisamine

Võtme lisamiseks tuleb leida talle sobiv koht vastavalt võtme väärtusele ja seejärel ta uueks leheks lisada (uus võti saab alati leheks). Lisamise (tegelikult küll õige koha otsimise) keerukus on `O(log n)` vastavalt puus olevale elementide arvule `n`.

Algoritm on järgmine:

- alustades puu juurelemendist võrreldakse iga võtit lisatava võtmega;
- kui lisatav võti on väiksem, jätkatakse liikumist vasakus alampuus;
- kui lisatav võti on suurem, jätkatakse liikumist paremas alampuus;
- kui valitud alampuu on tühi, on uuele võtmele koht leitud.

Lisamise funktsioon saab sisendiks puu juure aadressi `T` ja lisatava võtme 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 raamatule Cormen, Leiserson, Rivest *"Introduction to Algorithms"*, nn *CLR*.

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x != NIL              // liigume leheni, mille küljes õige tühi alampuu
    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, saab z juureks
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.

Otsing

Otsinguvõtme järgi otsimise tulemuseks on kas tipu aadress või teade sellise tipu / võtme puudumisest.

Algoritm on lühike ja lihtne: alustades puu juurest liigutakse vastavalt otsinguvõtme väärtusele vasakusse või paremasse alampuusse kuni võti leitakse või kuni jõutakse leheni. Kui võti leitakse, tagastatakse tipu aadress, kui ei leita, siis `NIL`. Algoritmile on sisendiks otsinguvõti `k` ja puu juure aadress `T` ning väljundiks tipu aadress `x` või `NIL`. 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 ei leita, jõuab muutuja `x` leheni ja omandab väärtuse `NIL` (mille abil ka tsükli jätkamist kontrollitakse). Kui võti leitakse katkeb tsükkel varem ja `x`-i väärtuseks on otsitava tipu aadress.

Sama algoritm rekursiivselt (vastavalt *CLR*-ile):

```
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 (vastavalt *CLR*-ile) 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 kolm võimalust:

1) kui tipul pole lapsi, võib tipu lihtsalt ära kustutada;
2) kui tipul on üks laps, paigutatakse see kustutatud tipu asemele;
3) kui tipul on mõlemad lapsed, 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 (parema alampuu kõige väiksema võtmega tipp).

Kõigi võtmete väljastamine

Kahendpuu läbimiseks oli kirjeldatud kolm võimalikku järgnevust. Kasutades läbimiseks **inorder**-järjekorda (vasak alampuu, juur, parem alampuu) saab võtmed kätte sorteeritult ehk kasvavas järjekorras. Vastav algoritm on järgmine (vastavalt *CLR*-ile):

```
INORDER-TREE-WALK(x)
if x != NIL
```

```
INORDER-TREE-WALK(x.left)
print x.key
INORDER-TREE-WALK(x.right)
```

Korduvad võtmed

Üldiselt eeldatakse, et otsimine toimub unikaalsete võtmete järgi. Kahendotsingupuusse saab lisada ka korduvaid võtmeid. Sõltuvalt ülesande iseloomust on kolm võimalust:

1. Igasse tippu lisatakse loendur, mis näitab mitu sellist võtit on.
2. Uute tippude lisamisel kasutatakse "väiksem"-suhte asemel "väiksem või võrdne"-suhet, seega sama võtmega element pannakse vasakusse alampuusse.
3. Ei lasta ennast häirida ja korduvat võtit ei lisata.

Tasakaalustatud puu

Üldiselt loetakse kahendotsingupuust otsimist, nagu ka teisi operatsioone, logaritmilise keerukusega tegevuseks. Millest sõltub otsinguks kuluv sammude arv? Puu kõrgusest. Maksimaalne sammude arv on puu kõrgus ehk otsingul läbitakse tee juurelemendist leheni, kui otsitavat varem ei leita. Puu kõrgus on aga omakorda seotud puus olevate elementide arvuga logaritmilise seose läbi. Ehk kahendpuu (täpsemalt küll täieliku kahendpuu - mis see oli?) kõrgus on `log(N)`, kus `N` on tippude arv puus. Seega logaritmiline keerukus on tagatud vaid juhul, kui puu tasemed on maksimaalselt täidetud ehk on tekkinud täielik kahendpuu. Puu tegelik headus sõltub sellest, millises järjekorras võtmed lisatakse, sest selle järgi saab puu endale kuju. Milline on halvim variant? Kindlasti see, kus lisatavad võtmed on eelnevalt järjestatud. Sel juhul saadakse sisuliselt lineaarne ahelloend ja otsimise keerukus on lineaarne.

Kui puu sõlmed on võimalikult ühtlaselt jaotunud, kõik tasemed on enam-vähem täis, siis on teed puu juurelemendist lehtedeni (enam-vähem) võrdse pikkusega ja aeg tõepoolest logaritmiline.

Kuidas hoida otsingupuud heas seisundis?

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

Et saavutada otsingupuus paremaid tulemusi, tasakaalustatakse neid. Kasutatakse mõistet **tasakaalus otsingupuu** *(*ingl *balanced search tree)*. Tasakaalus kahendotsingupuu tagab logaritmilise aja nii otsimisel, lisamisel kui ka kustutamisel. Ehkki sammude arv `log(N)` võib olla korrutatud mõne konstandiga, on tegemist siiski keerukusklassiga `O(log N)`. Ehk me ei eelda täpset sammude arvu `log N` vaid tegemist on suurusjärguga `log N`.

Järgnevalt käsitletakse kahte tasakaalus otsingupuud: AVL puud ja puna-musta puud.

AVL puu

Hea lahenduse puu optimaalses seisundis hoidmiseks pakkusid 1962. aastal välja teadlased *G. M. Adelson-Velskii* ja *E. M. Landis*. (Adelson-Velskii, G. M., & Landis, E. M. (1962). An algorithm for the organization of information. Soviet Math. Doclady, 3, 1259–1263, *https://zhjwpku.com/assets/pdf/AED2-10-avl-paper.pdf*) Nende meetod vajab küll iga tipu jaoks veidi täiendavat mälu, kuid garanteerib see-eest alati logaritmilise otsimisaja. Lähtudes puu reeglitest hinnatakse otsingu sammude arvuks halvimal juhul *(*ingl *worst case)* `1.440 * log N`. Oma leiutajate auks kannab see puu **AVL-puu** *(*ingl *AVL-tree)* nime.

AVL puu peamine reegel on, et iga tema tipu vasaku ja parema alampuu kõrguste vahe (erinevus) ei

ole suurem kui üks. See reegel kehtib absoluutselt igas tipus.

Tasakaalustamine

AVL-puu saamiseks on vaja peale tipu lisamist või kustutamist kontrollida puu tasakaalustatust ja vajadusel viia puu tasakaalu tippe ümber paigutades.

Puu tasakaalustamiseks seotakse iga tipuga **tasakaalufaktor** *(*ingl *balancing factor)*, mille abil iseloomustatakse iga tipu vasaku ja parema alampuu kõrguste vahet. Faktoril saab olla kolm erinevat väärtust, mis kirjeldavad tipu olukorda:

-1 tipu vasak alampuu on 1 võrra kõrgem

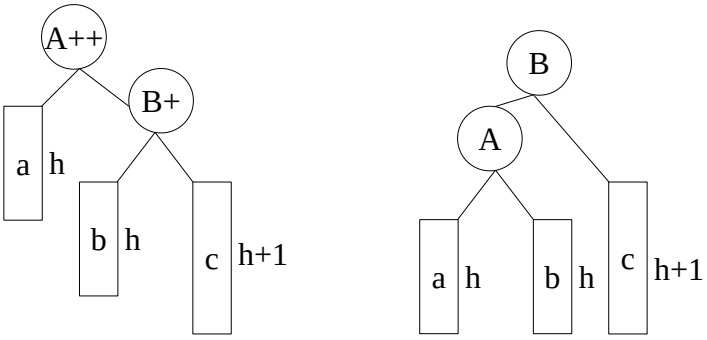
+1 tipu parem alampuu on 1 võrra kõrgem

0 tipu mõlema alampuu kõrgused on võrdsed

Puu jaoks kriitilised on tipud faktoritega `-1` ja `+1`. Kui esimesel juhul lisada tipp vasakusse alampuuse ja teisel juhul paremasse alampuusse, läheb puu vastava tipu kohal tasakaalust välja.

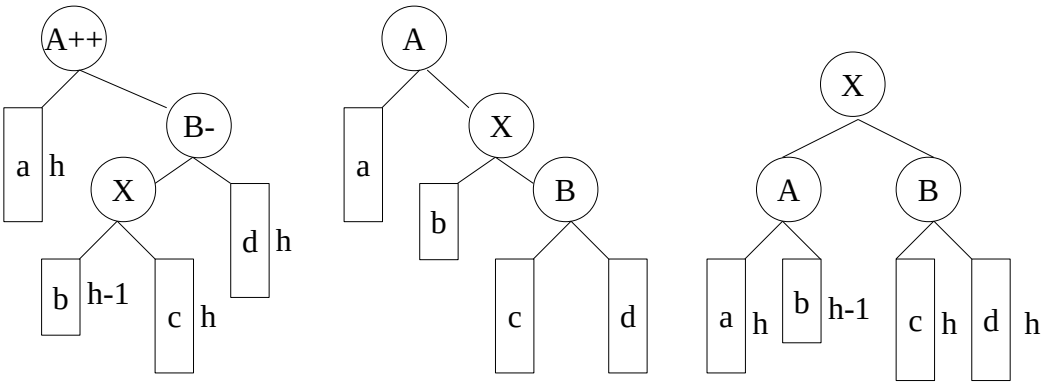
Eristatakse kahte erinevat varianti tasakaalust välja minekul (ja vastavaid tasakaalustamise algoritme) pluss peegeldused:

1. Tipu A parem alampuu on ühe võrra kõrgem (faktor +1), parema alampuu (millel juureks B) paremasse alampuusse lisatakse tipp, sellega muutub ka tipu B faktor +1 ja tipus A läheb puu tasakaalust välja. Peegeldatud olukord tekib siis, kui eelnevalt on tipu A faktor -1 ja tema vasakusse alampuusse tipuga B lisatakse vasakule juurde tipp. Selle tagajärjel saab tipu B faktoriks -1 ja puu on tipus A tasakaalust väljas (vt Joonis 4).
2. Tipul A on ühe võrra kõrgem parem alampuu (faktor +1), tema paremasse alampuuse tipuga B lisatakse vasakule poole tipp, nii et tipu B jaoks tasakaalufaktor muutub -1. Peegelduses: tipul A on eelnevalt faktor -1 ja tema vasakusse alampuusse tipuga B lisatakse paremale juurde tipp, mille tulemusena tipu B faktoriks saab +1 ning puu on tipus A tasakaalust väljas (vt Joonis 5).



Joonis 4: Puu tasakaalustamise esimene variant

Selgitus esimesele tasakaalustamise variandile (Joonis 4). Puu on tasakaalust väljas kaks korda ühele poole. Enne tipu lisamist on tipu A tasakaalufaktor +. Tipu B tasakaalufaktor saab peale tipu lisamist +. Puu on tipus A tasakaalust väljas. Tasakaalustamiseks pööratakse alampuud tipuga A vasakule. B saab uueks alampuu juureks ning tipu B vasak alampuu saab A paremaks alampuuks.



Joonis 5: Puu tasakaalustamise teine variant

Selgitus teisele tasakaalustamise variandile (Joonis 5). Puu on tasakaalust väljas tipus A paremale ja tipus B vasakule. Enne tipu lisamist oli tipu A tasakaalufaktor +. Peale tipu lisamist saab tipu B tasakaalufaktoriks - ja puu on tipus A 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).

Lisaks tasub tähele panna, et uue tipu lisamisel võib puu tasakaalust välja minna korraga mitme tipu juures. Peale tipu lisamist muudetakse alt üles (lehest juureni) liikudes kõik tasakaalufaktorid ning koos sellega kontrollitakse puu seisundit, alustatakse kontrolli ja tippude pööramist alt poolt. Kontrolli käigus liigutakse mööda puud kuni juureni ja pööramisi võib olla vaja teha rohkem kui ühes kohas. Et kontrollida saaks mugavalt, peab lisaks tasakaalufaktorile igas puu tipus ka meeles pidama selle tipu kõrgust, sest see annab ühtlasi teada alampuu kõrguse. Lehe kõrgus on 0, järgmistel tippudel on kõrgus pikim tee leheni. Juure kõrgus on kogu puu kõrgus ehk sama moodi pikim tee leheni.

Andmestruktuuri, kus hädapärastele andmetele lisatakse operatsioonide efektiivsuse tõstmiseks veel mingeid väärtuseid (puu kõrgused jms) nimetatakse **laiendatud andmestruktuuriks** *(*ingl *augmented data structure)*.

Puna-must puu

Puna-must puu *(*ingl *red-black tree)* on kahendotsingupuu, kus määratakse igale tipule värv: punane või must. Värvide abil kirjeldatakse puna-musta puu reeglid. Arvestades puna-musta puu reegleid, ei saa ükski tee puu juurest leheni olla üle kahe korra pikem võrreldes ükskõik millise teise samas puus oleva teega. Võib väita, et puu on ligikaudu tasakaalus. Sellele puule omase tasakaalustamise tehnika pakkus esimest korda välja R. Bayer. Pikemalt töötasid puu kallal ja andsid talle nime L. J. Guibas ja R. Sedgewik.

Puna-must puu (lühendan edaspidi PM-puu) on kahendotsingupuu, kus on täidetud **puna-mustad omadused** *(*ingl *red-black properties)*:

a) iga tipp on kas punane või must;

b) puu juur on must;

c) lehtede `NIL`-viidad (ehk tühjad alampuud) on mustad;

d) kui tipp on punane, on tema mõlemad lapsed mustad;

e) iga tipu jaoks kõik temast algavad teed kuni leheni 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.

`n` tipuga PM-puu maksimaalne kõrgus on `2*log(n+1)`. Sellest järeldub, et üldisemalt igasugused operatsioonid jäävad logaritmilisse suurusjärku `O(log n)`.

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

Lisaks tavalisele puu kõrgusele räägitakse PM-puu **mustast kõrgusest** *(*ingl *black hight)* - see on mustade tippude arv puu juurest leheni (väljaarvatud juur ise). Puu must kõrgus on iga lehe suhtes ühesugune (5. reegel). Sarnaselt määratakse must kõrgus igale tipule - mustade tippude arv, välja arvatud tipp ise.

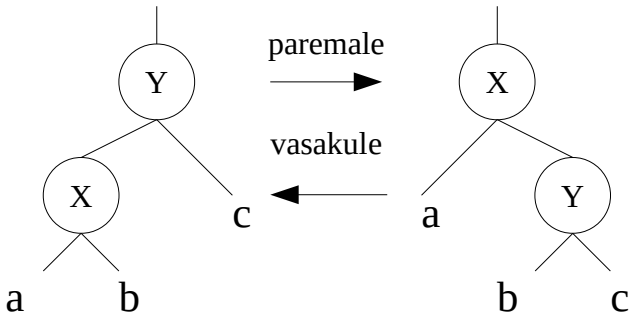
PM-puus toimub **otsinguvõtmega tipu otsimine**, samuti **vähima** ja **suurima elemendi leidmine** sarnaselt tavalisele kahendotsingupuule (keerukus kõigi operatsioonide jaoks `O(log n)`). Uue elemendi lisamisel on erinevused, mis seostuvad reeglite täidetuse kontrollimise ja puu korrastamisega.

Elemendi lisamine

Et tegemist on kahendotsingupuuga, algab uue elemendi lisamine puusse sarnaselt varem kirjeldatuga. Edasi tuleb aga kontrollida, kas puna-mustad omadused on jätkuvalt täidetud ja kui ei ole, tuleb hakata puud korrastama, ehk tippude asukohti muutma. Vajadusel tehakse AVL-puule sarnaseid pöördeid. Esimese korrastamise järel võivad omadused mitte kehtida järgmises kohas ja nii korrastatakse puud ülespoole liikudes, kuni jälle kehtivad kõik puna-mustad omadused.

Puu korrastamiseks kasutatakse kolme operatsiooni:

- tippude värvimine - punane tipp värvitakse mustaks ja must punaseks;
- pööre vasakule - tipu X parem laps saab uueks (alam)puu juureks ning X ise satub tema vasakuks lapseks (vt Joonis 6 ja ka pööramine AVL-puus Joonis 4);
- pööre paremale - tipu X vasak laps saab uueks (alam)puu juureks ning X ise satub tema paremaks lapseks (vt Joonis 6 ja ka pööramine AVL-puus Joonis 4).



Joonis 6: Puu pööramine tippude X ja Y kohalt paremale ja vasakule.

Uus tipp lisatakse leheks nagu tavalises kahendotsingupuus ja värvitakse esialgu punaseks. Kui tipp lisatakse punase tipu külge, siis lisatud tipp rikub puna-musti 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 (kus on 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 juureks saab punane tipp, siis see värvitakse mustaks.

B-puu

B-puu *(*ingl *B-tree)* on teatud tüüpi otsingupuu, mis on sobiv info hoidmiseks kõvakettal, kus toimub otsepöördumine. Hoides infot kettal pole nii oluline arvutusteks kuluv aeg, kui aeg, mis kulub kettaga suhtlemiseks (lugemiseks ja kirjutamiseks). Suurema osa **pöördumisajast** *(*ingl *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 õige sektori leidmiseks järgmise sektori poole pöörduda, seda parem.

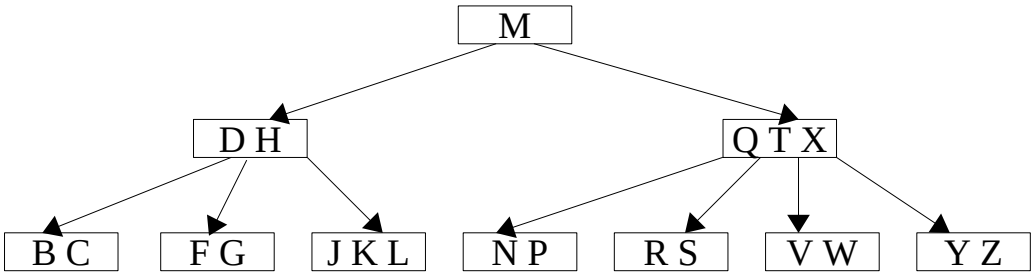
B-puus on sisend-väljundoperatsioonide arv proportsionaalne puu kõrgusega, seepärast tuleb puud hoida madalana. Ühe tipu laste arvu ei piirata 2-ga, vaid praktiliselt võib see ulatuda ka 1000-sse. Seega on puu kõrgus palju väiksem kui tasakaalustatud kahendotsingupuul.

Reeglid B-puu jaoks, mille järk (maksimaalne tipu laste arv) on `t`:

1. Igas tipus `x` on väljad, milles hoitakse:
   a) tipu võtmete arvu `x.n`;
   b) võtmeid mittekahanevas järjekorras $x.k_1 <= x.k_2 <= \ldots <= x.k_n$;
   c) tähist, kas tipp on leht (loogikaväärtus).
2. Kui `x` on sisemine tipp, on temas `x.n` elementi ja `x.n+1` viita lastele.
3. Kuna lehtedel lapsi pole, pole nende jaoks ka viidaväljasid ettenähtud.
4. Tipus olevad võtmed (`x.n` 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** *(*ingl *minimum degree)*:
   a) igas tipus (va juurtipus) 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 rohkem kui `2t` last; tipp on täis *(*ingl *full),* kui temas on `2t-1` võtit.

Lihtsamal juhul on `t=2`, siis saadakse **2-3-4-puu** *(*ingl *2-3-4 tree)*. Praktikas soovitatakse `t` võtta suurem kui 2.

Tipus `x` säilitatakse `x.n` võtit. Nende võtmete järgi jagatakse lapsed `x.n+1` gruppi. Seega on tipul `x` `x.n+1` last. Otsimisel võrreldakse otsinguvõtit tipus `x` olevate võtmetega ja valitakse jätkamiseks üks võimalikest teedest, vastavalt sellele, millisesse võtmete abil määratud vahemikku otsinguvõti kuulub.



Joonis 7: 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 juurelemendis) (vt Joonis 7).

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 rohkem kui kahe `(x.n+1)` lapse vahel. Otsimisel vaadatakse juurest alustades igas tipus `x` läbi võtmed. Kui otsinguvõti `k` leitakse, tagastatakse viide tipule ja võtme järjekorranumber antud tipus. Igas tipus vaadatakse võtmed järjest läbi ja peatutakse vähima `i` juures, mille jaoks `k <= x.key`. Kui sellist ei leita, siis saab `i` väärtuseks `x.n+1`. Kui otsinguvõti `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 kahendotsingupuusse. 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 **poolitav võti** *(*ingl *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.