

## 2-3 paieškos medis

Tai vienas iš subalansuotų (dvejetainių) paieškos medžių. Jį 1970 metais pristatė Džonas Hapkorftas (John Hopcroft).

### 2-3 medžių savybės ir taisyklės

2-3 medžiai pasižymi šiomis savybėmis:

- visi lapai yra viename lygyje;
- kiekviena vidinė viršūnė turi 2 arba 3 vaikus;
- kiekviena viršūnė turi 1 arba 2 reikšmes;
- medžio gylis yra tarp  $\lceil \log_3 n \rceil$  ir  $\lfloor \log_2 n \rfloor$ , kur  $n$  - viršūnių skaičius;

2-3 paieškos medis gali būti naudojamas saugojimui tiek aibės (elementai negali kartotis), tiek rinkinio (elementai gali kartotis).

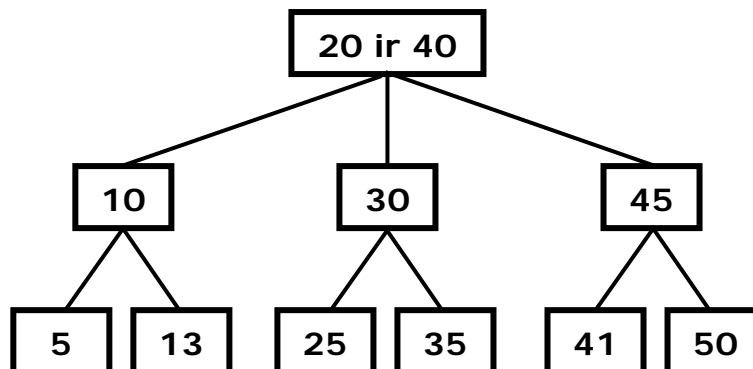
2-3 medžių sudarymo taisyklės aibės atveju:

1. kiekvienas lapas bei vidinė viršūnė saugo 1 arba 2 reikšmes. ( $x$  arba  $x$  ir  $y$ );
2. esant viršūnėje dviems reikšmėms, pirmoji yra mažesnė už antrąją ( $x < y$ );
3. kiekviena vidinė viršūnė su 1 reikšme  $x$  turi 2 pomedžius:
  - kiekviena 1-ojo pomedžio reikšmė  $R$  yra mažesnė už  $x$  ( $R < x$ );
  - kiekviena 2-ojo pomedžio reikšmė  $R$  yra didesnė už  $x$  ( $R > x$ );
4. kiekviena vidinė viršūnė su 2 reikšmėmis  $x$  ir  $y$  turi 3 pomedžius:
  - kiekviena 1-ojo pomedžio reikšmė  $R$  yra mažesnė už  $x$  ( $R < x$ );
  - kiekviena 2-ojo pomedžio reikšmė  $R$  yra tarp  $x$  ir  $y$  ( $x < R < y$ );
  - kiekviena 3-ojo pomedžio reikšmė  $R$  yra didesnė už  $y$  ( $R > y$ );

Įterpiant ar išmetant elementą iš 2-3 medžio, pasitaiko situacijų, kai vidinė viršūnė laikinai turi tris elementus ( $x, y, z$ ) ir keturis pomedžius. Tokia vidinė viršūnė turi tenkinti tokias taisykles, kurios ekvivalenčios esant 2 reikšmėms ir 3 vaikams:

- kiekviena 1-ojo pomedžio reikšmė  $R$  yra mažesnė už  $x$  ( $R < x$ );
- kiekviena 2-ojo pomedžio reikšmė  $R$  yra tarp  $x$  ir  $y$  ( $x < R < y$ );
- kiekviena 3-ojo pomedžio reikšmė  $R$  yra tarp  $y$  ir  $z$  ( $y < R < z$ );
- kiekviena 4-ojo pomedžio reikšmė  $R$  yra didesnė už  $z$  ( $R > z$ );

2-3 medžio pavyzdys:



## Elemento paieška

Elemento paieška 2-3 medžiuose labai panaši kaip ir dvejetainės paieškos medžiuose. Paieška pradedama nuo šaknies. Bendru atveju, tikrinama viršūnės reikšmė (reikšmės). Galimi variantai:

- ieškoma reikšmė yra viršūnėje → paieška baigta;
- reikšmė yra mažesnė už i-tąją viršūnės reikšmę → paieška tęsiama i-tajame pomedyje;
- reikšmė yra didesnė už visas vidinės viršūnės reikšmes → paieška tęsiama i+1 pomedyje;

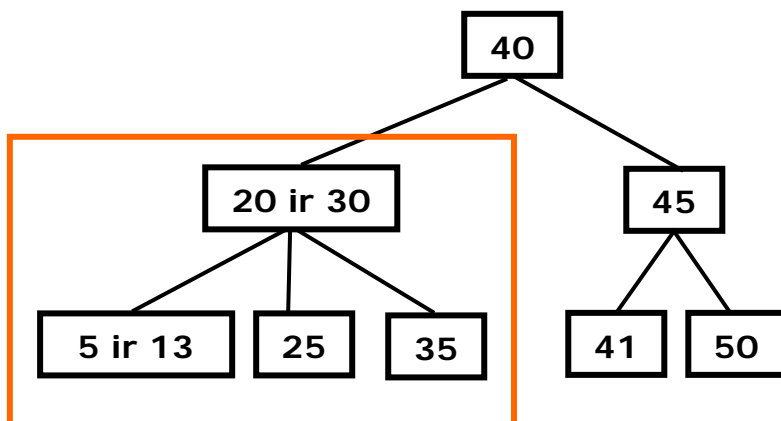
Minėti žingsniai vykdomi, kol randama reikšmė arba kol prieinama prie lapo (elementas nerastas).

## Elemento įterpimas

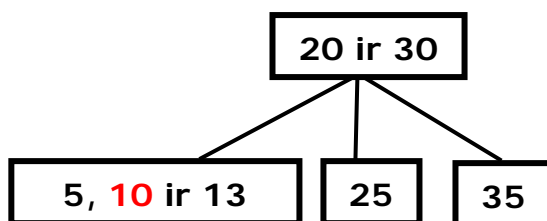
Elementas įterpiamas į medį pagal bendras medžių taisykles. Po to pagal susidariusią situaciją medis yra pertvarkomas, kad atitiktų 2-3 medžio taisykles. Įterpus naują elementą į 2-3 medį gali būti viena iš šių dviejų situacijų:

- lapas turi 2 reikšmes, 2-3 medžių formavimo taisyklės nėra pažeistos;
- lapas turi 3 reikšmes, tokių lapų 2-3 medžiuose negali būti.

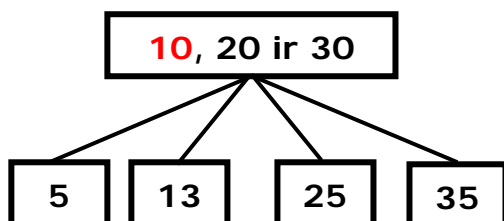
Elemento įterpimą ir pertvarkymą pavaizduosime pavyzdžiu:



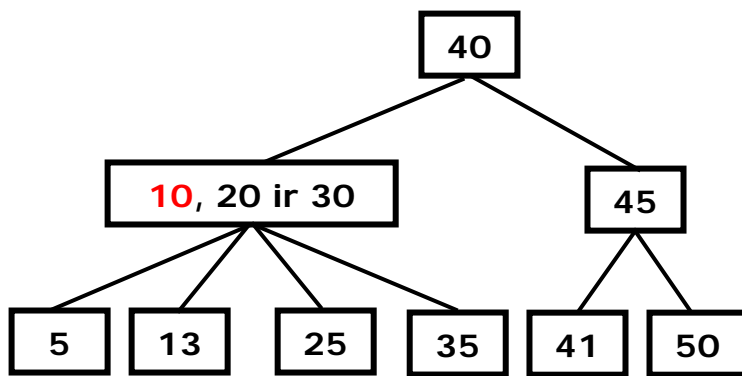
Štai į šį medį įterpsime reikšmę 10. Toliau (2 žingsnius) nagrinėsime tik raudonai apvestą pomedį



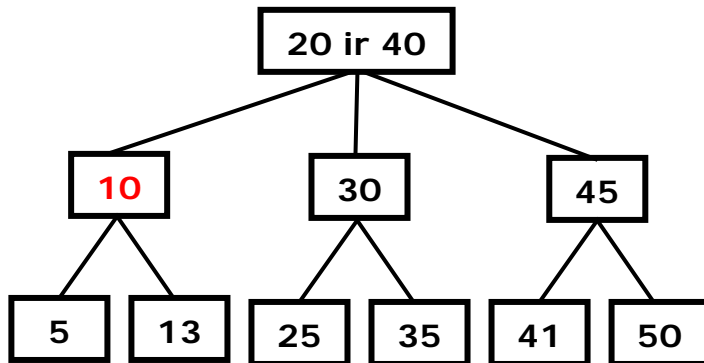
Skaičių 10 įterpėme į jam skirtą vietą, nes jis yra mažesnis už 20. Šis pomedis nėra 2-3 medis, nes jo lape yra trys reikšmės.



Paėmėme probleminio lapo vidurinę reikšmę ir perdavėme ją tėvui. Dabar tėvas turi 3 reikšmes ir 4 vaikus. 3 reikšmės suskirsto į 4 intervalus, į kuriuos įeina kiekvienas pomedis, kaip aprašyta taisyklėse.



Dabar viena vidinių viršūnių turi 3 reikšmes. Tos viršūnės vidurinį elementą (20) perduodame tėvui.



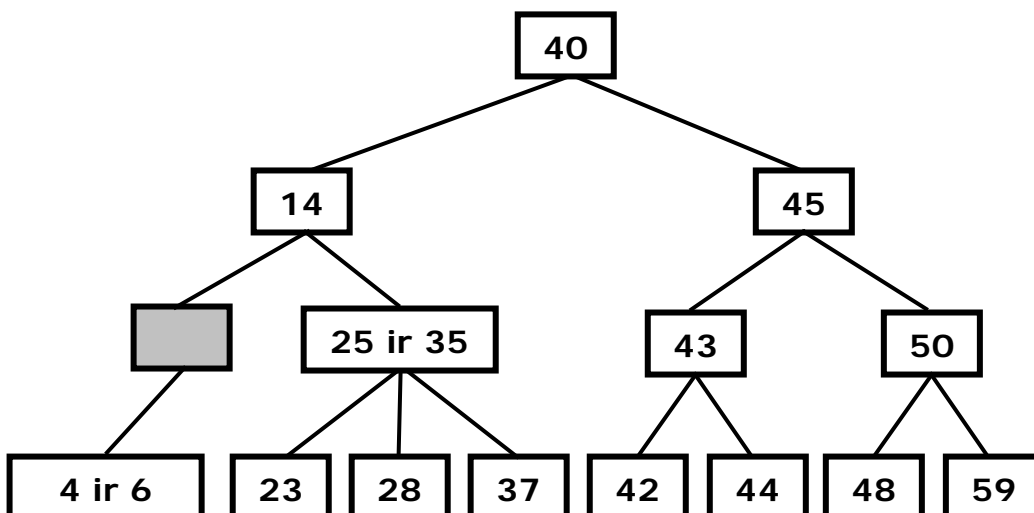
O gretimos viršūnės reikšmės (10 ir 30) tampa naujomis vidinėmis viršūnėmis ir pasidalina vaikus. Rezultatas - 2-3 medis.

### Elemento išmetimas.

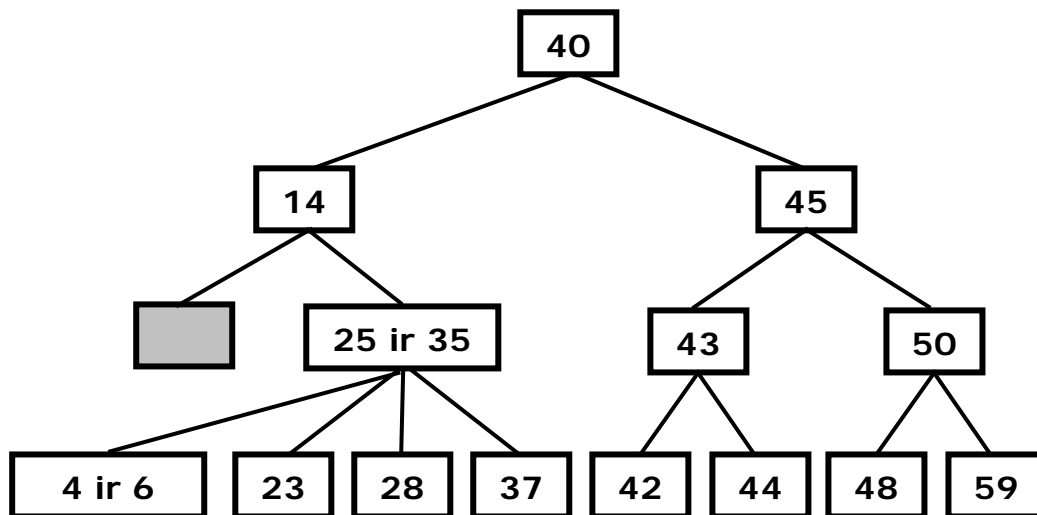
Norint elementą išmesti iš medžio, pirmiausia reikia jį pašalinti, po to sutvarkyti medį taip, kad jis atitiktų 2-3 medžio taisykles. Galimi 2 atvejai:

- elementas yra vidinėje viršūnėje;
- elementas yra lape.

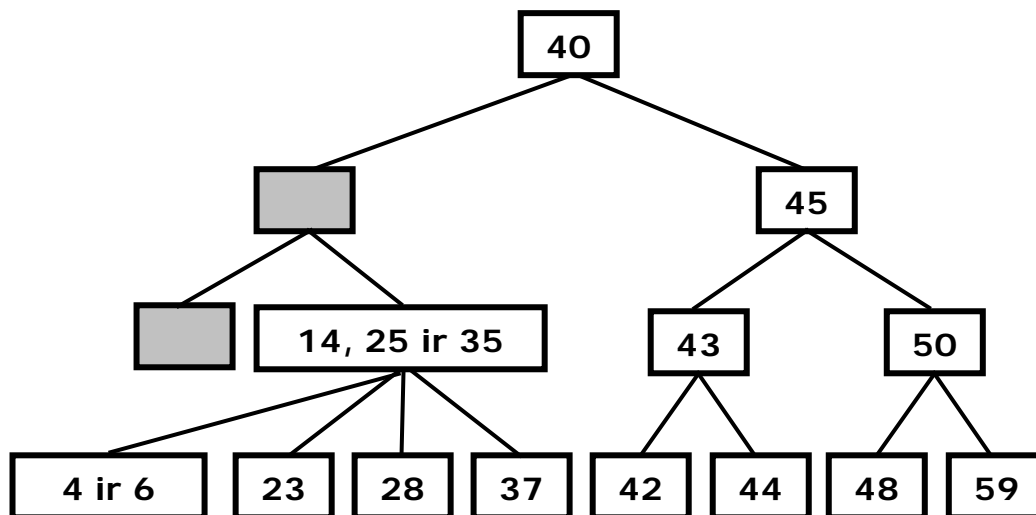
Pirmuoju atveju reikia elementą, esantį vidinėje viršūnėje, sukeisti su didžiausiu elementu iš to vidinės viršūnės pomedžio, kurio visi elementai yra mažesni už šalinamą elementą. Tada, kaip ir antruoju atveju bus galima išmesti lape esančią reikšmę. Jei išmetus iš lapo reikšmę, lape yra dar nors viena reikšmė, išmetimo darbas baigtas. Priešingu atveju turime tuščią lapą, o tai prieštarauja 2-3 medžio taisyklėms. Toliau pristatysime būdą, kaip reikia pataisyti medį, kad po tokio elemento pašalinimo jis vėl būtų 2-3 medis. Šis pataisymo būdas tinka ir tuščiai vidinei viršūnei, su vienu pomedžiu. Medžio sutvarkymas, esant tuščiai vidinei viršūnei.



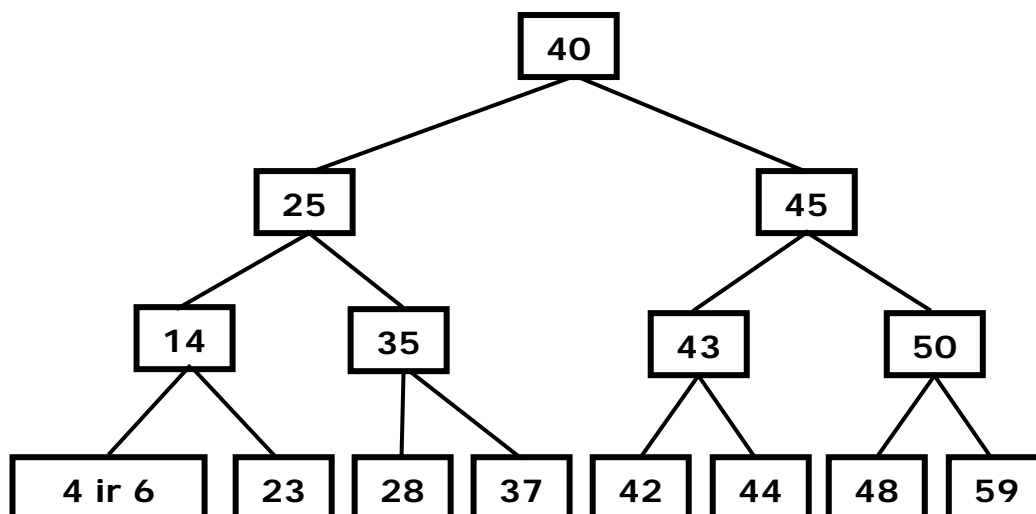
Tarkime, turime štai šitokį 2-3 paieškos medį. Pilkai yra pažymėta tuščia vidinė viršūnė.



Tuščios vidinės viršūnės sūnų perduodame laimingajam broliui. Laimingasis brolis (LB) nustatomas taip: jei tuščia viršūnė yra paskutinis (n-asis) vaikas, tai LB yra (n-1)-asis, priešingu atveju (tėvas turi daugiau negu n vaikų) LB yra (n+1)-asis.



Perdavus LB sūnų, reikalinga papildoma reikšmė LB vidinėje viršūnėje. Ją paimame iš tėvo. Dabar galime pašalinti tuščią viršūnę, o su LB vidine viršūne pasielgsime taip pat, kaip ir įterpimo atveju.



LB vidinę viršūnę skiriame į tris dalis, vidurinę reikmę nusiunčiame į tėvo viršūnę, o kitos dvi reikšmės atitinkamai pasidalina vaikus. Rezultate gauname 2-3 medį.

**Abstraktus duomenų tipas (ADT)** yra duomenų apibrėžimas, **supakuotas (inkapsuliuotas)** su apibrėžimu visų tam duomenų tipui prasmingų operacijų.

**Supakavimas arba inkapsuliavimas:** *Realizacija duomenų tipo kartu su jo operacijomis taip, kad šis duomenų tipas ir jo operacijos galėtų būti naudojamos, nežinant realizacijos detalių.*

Norime apibrėžti naujus duomenų tipus, kurie būtų kiek galima artimesni standartiniams (angl. *built in*) programavimo kalbos duomenų tipams. Kitais žodžiais, norime praplėsti programavimo kalbą, įvesdami naujus duomenų tipus kartu su jų operacijomis. Kad tai pasiekti, turime išsaugoti privalumus, kuriuos turi standartiniai duomenų tipai:

- \* *Patys tipai yra nepermatomi (angl. opaque), t.y. galima juos naudoti, nežinant, kaip jie yra vaizduojami atmintyje.*

- \* *Operacijos turi natūralią sintaksę, todėl galima jas naudoti paprastai ir natūraliai.*

- \* *Operacijos yra nepermatomos (angl. opaque), t.y. galima jas naudoti, nežinant, kaip jos realizuotos. Kitais žodžiais, pakanka žinoti *ką* jos daro, bet neprivalu žinoti *kaip* jos tai daro.*

**Nepermatomas tipas arba operacija:** *Duomenų tipas ar operacija, kurios realizacijos detalės yra paslėptos.*

Pažymėkime skirtumą tarp standartinių duomenų *naudojimo* ir žinojimo kaip juos *padaryti*. Pascal sustiprina šį skirtumą, nepasakydamas kaip standartiniai duomenų tipai vaizduojami atmintyje ir kaip realizuotos jų operacijos. Jis pasako tik tai, ko reikia tam, kad galėtume naudoti standartinius duomenų tipus ir operacijas. Ši slėpimo nuo vartotojo realizacijos detalių, kurių jam nereikia žinoti, strategija vadinama **informacijos slėpimu** (angl. *information hiding*). Tai svarbu aukšto lygio programavimo kalbose, nes tai leidžia mąstyti apie duomenų tipus abstrakčiau.

**Informacijos slėpimas:** *Metodas rodymo klientams (programuotojams) tik to, ką reikia žinoti tam, kad naudoti duomenų tipą ar operacijas, ir neatskleidimo jiems realizacijos detalių.*

Palyginkime tai su būdu kaip konstruojamas automobilis. Vairuotojas (priešingai mechanikui) mąsto apie automobilį kaip apie nepermatomą objektą - tai automobilis ir negalvoja, kaip jis viduje veikia. Vairuotojo požiūriu automobilis yra abstrakcija, daiktas skirtas transportavimui, o ne tam tikras kokių 5000 detalių rinkinys. Vairuotojo abstrakcija taip pat apima automobilio panaudojimo operacijas: susoti, padidinti greitį ir kt. Kaip matome, vairuotojas supranta automobilį **sąsajos** (interfeiso; angl. *interface*) tarp vairuotojo ir automobilio terminais. Vairuotojo supratimas automobilio kaip abstrakcijos galimas gero projekto ir glaudaus automobilio detalių integravimo dėka, kas duoda paprastas ir natūraliai atrodančias operacijas, t.y. sąsają (*interfeisą*) tarp automobilio ir vairuotojo.

Iš kitos pusės, mechanikas (automobilių konstruktorius ar vairuotojas su sudaužyta mašina) privalo mąstyti apie vidinę automobilio konstrukciją. Šie žmonės supranta automobilį visiškai kitaip. Jie privalo galvoti apie automobilį jo sudedamųjų dalių hierarchijos terminais, o ne kaip apie vieną daiktą. Galima sakyti, kad jie privalo nusileisti *žemiau abstrakcijos*, gilindamiesi į **realizaciją**.

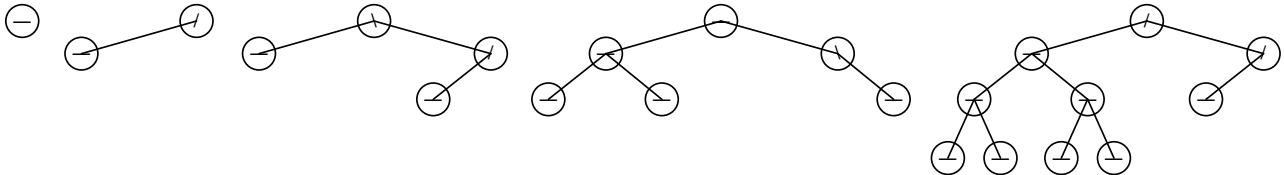
Dabar galvokime apie automobilį kaip abstraktų duomenų tipą. Vartotojo požiūriu automobilis yra koncepcija kartu su aibe savo operacijų, prasmingų naudojant šią koncepciją - vairuojant automobilį. Šis įvaizdis vadinamas abstraktaus duomenų tipo **sąsaja** (*interfeisu*). Iš kitos pusės, abstraktaus duomenų tipo **realizacija** (angl. *implementation*) susideda iš visų dalių, kurios leidžia interfeisui dirbti: procedūrų ir funkcijų kodo. Atskirdami realizaciją nuo interfeiso, padarėme svarbų atskyrimą tarp koncepcijos naudojimo ir jos vidinių mechanizmų. Yra dar vienas svarbus aspektas: *interfeisas gali būti užtikrintas daugiau nei vienu realizacijos būdu*. Pavyzdžiui, konstruktorius gali projektuoti automobilį kaip atliekamų funkcijų hierarchiją. Dalys yra tos pačios kaip ir mąstant automobilio sudedamųjų dalių hierarchijos terminais, bet jos grupuojamos skirtingais principais. Toks požiūris labiau priimtinas inžinieriui negu mechanikui. Vairuotojai dažniausiai nesidomi nei vienu hierarchiniu požiūriu: vairuotojai rūpinasi kaip pilnai mašina atrodo, jaučiama ir valdoma.

Ši analogija turėtų padėti pajauti skirtumą tarp abstrakčios sistemos interfeiso ir jos realizacijos.

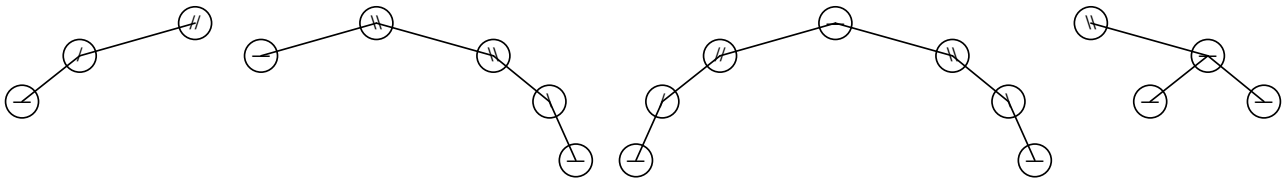
## AVL – medžiai

**Apibrėžimas:** AVL-medis – tai paieškos medis, kurio šaknies dešinysis ir kairysis pomedžiai skiriasi daugiausiai vienetu ir kurio pomedžiai, savo ruožtu, tai pat yra AVL-medžiai.

AVL-medis, gavęs vardą iš savo kūrėjų (Adel'son-Vel'skii ir Landis) – tai subalansuotas dvejetainis paieškos medis. Kadangi kairiojo ir dešiniojo bet kurios medžio viršūnės pomedžio aukščiai neturi skirtis daugiau kaip vienetu, paieška AVL-medyje yra tokia pat efektyvi, kaip ir dvejetainiame paieškos medyje, turinčiame minimalų aukštį. AVL-medis – viena iš seniausių subalansuoto medžio rūšių.

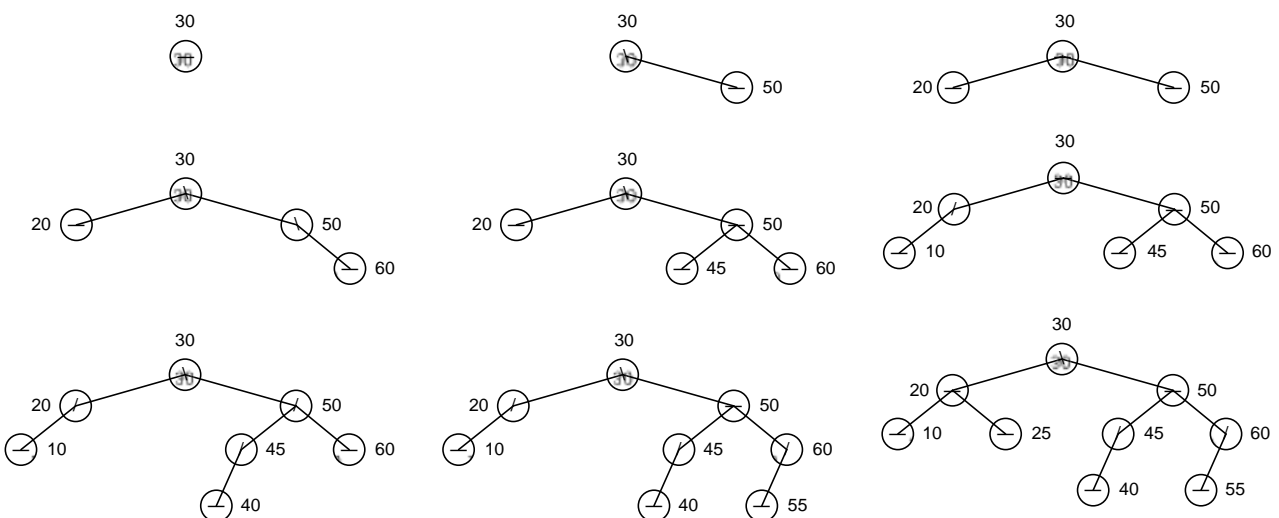


Pav 1. AVL-medžiai



Pav 2. ne AVL-medžiai

Bet kurį dvejetainį paieškos medį, turintį  $n$  viršūnių, galima pertvarkyti taip, kad naujas medis būtų minimalaus aukščio  $\lceil \log_2(n+1) \rceil$ . Bet nuolatinis medžio pertvarkymas gali būti per brangus, todėl AVL-medžiai siūlo alternatyvą. Jis siūlo daryti medį beveik minimalaus aukščio, tuo pačiu metu darant mažiau darbo. Pagrindinė šio metodo mintis yra nuolat stebėti medžio formą. Įterpimo ir pašalinimo operacijos nieko nesiskiria nuo tų pačių dvejetainio paieškos medžio operacijų, bet po kiekvieno įterpimo arba pašalinimo vykdomas tikrinimas, ar medžio savybės išliko. Kitaip tariant, po kiekvienos operacijos (įterpimo ar pašalinimo) tikrinama, ar kiekviena viršūnė teturi pomedžius, kurių aukščiai skiriasi ne daugiau kaip vienetu. Išanalizuokime AVL-medžio augimą, kai keičiasi tik šaknies balansas, o pomedžių aukštis nesiskiria daugiau nei vienetu. Diagramose naudojamas toks žymėjimas: „/“ – kai kairiojo pomedžio aukštis yra didesnis (left high); „\“ – kai dešiniojo pomedžio aukštis yra didesnis (right high); „–“ – kai pomedžių aukščiai lygūs (equal high).



Pavyzdys 3. Paprastas elementų įterpimas į AVL-medį

Bazinė įterpimo algoritmo struktūra bus panaši į standartinio dvejetainio medžio įterpimo algoritmą, išskyrus kelis papildomus veiksmus, skirtus palaikyti AVL-medžio struktūrą. Pirmiausia, kiekvienas įrašas turės papildomą elementą, aprašytą kaip **bf** : **balancefactor**; , kur **bf** apibrėžtas tokiu būdu: **type balancefactor = (LH, EH, RH)**; . Čia LH – left high, EH – equal high ir RH – right high, atitinkamai. Antra, mes turime tikrinti ar naujo elemento įterpimas padidino medžio aukštį ar ne, kad tiksliai reaguoti į pasikeitimus. Tam įvedamas naujas kintamasis **taller** tipo **Boolean**. Balanso atstatymo funkcijai skirtos procedūros **LeftBalance** ir **RightBalance**.

**procedure** Insert(**var** root: **pointer**; newnode: **pointer**; **var** taller: **Boolean**);  
**var**

```

    tallersubtree: Boolean;
begin
    if root = nil then
        begin
            root := newnode;
            root^.left := nil;
            root^.right := nil;
            root^.bf := EH;
            taller := true;
        end
    else with root^ do
        if newnode^.info.key = info.key then
            Error
        else if newnode^.info.key < info.key then
            begin
                Insert(left, newnode, tallersubtree);
                if tallersubtree then
                    case bf of
                        LH: LeftBalance;
                        EH: begin
                                bf := LH;
                                taller := true
                            end;
                        RH: begin
                                bf := EH;
                                taller := false
                            end;
                    end
                else taller := false
            end
        else begin
            Insert(right, newnode, tallersubtree);
            if tallersubtree then
                case bf of
                    LH: begin
                            bf := EH;
                            taller := false
                        end;
                    EH: begin
                            bf := RH;
                            taller := true
                        end;
                    RH: RightBalance;
                end
            else taller := false
        end
    end;

```

**end**;

**Pav 4.** Naujo elemento įterpimas į AVL-medį

Aptarkime pavyzdžius, kai pomedžių aukščių skirtumas po naujo elemento įterpimo tampa didesnis nei vienetas ir gautas medis netenkina AVL-medžio apibrėžimą. Iš viso galimos dvi situacijos: kai pakanka vieną kartą pasukti medį, kad jis taptų subalansuotu, arba kai reikia dvigubo pasukimo balansui atstatyti. Analizuokime kiekvieną iš jų.

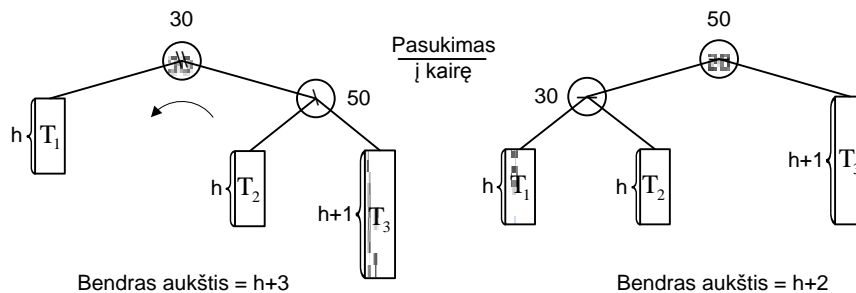
Pavyzdžiui turime medį, pavaizduotą **Pav 7**. Viršūnėse 30 ir 50 nėra balanso. Tam kad atstatyti subalansuotą formą, reikia atlikti vieną pasukimą į kairę. 50 tampa šaknimi, o 30 tuo tarpu tampa dešiniuoju 50 vaiku. Schema rodo, kad prieš pasukimą bendras aukštis buvo  $h+3$ , o po pasukimo į kairę –  $h+2$ .

```

procedure RotateLeft(var p: pointer);
                                {p – šaknis, kurio pomedis yra sukamas}
var
    temp: pointer;
begin
    if p = nil then
        Error
    else if p^.right = nil then
        Error
    else begin
        temp := p^.right;
        p^.right := temp^.left;
        temp^.left := p;
        p := temp;
    end;
end;

```

Pav 5. Procedūra, skirta pasukti medį į kairę



Pav 6. Pirma situacija: balanso atstatymas sukimu į kairę

Kai kuriais atvejais gali prireikti sudėtingesnių veiksmų. Analizuokime sekantį pavyzdį, kai 50 kairysis pomedis skiriasi daugiau negu vienetu nuo dešiniojo. Tam, kad atstatyti balansą, reikia atlikti dvigubą pasukimą, tai yra pasukti 45 iš pradžių į dešinę, o paskui į kairę. Po dvigubo pasukimo 45 tampa šaknimi. Schema rodo, kad prieš pasukimą bendras aukštis buvo  $h+3$ , o po dvigubo pasukimo –  $h+2$ .

```

procedure RightBalance;
var
    x,                                {rodyklė į šaknies dešinįjį pomedį}
    w: pointer;                       {kairysis x^ pomedis}
begin
    x := root^.right;
    case x^.bf of
        RH: begin
            root^.bf := EH;
            x^.bf := EH;
            RotateLeft(root);
            taller := false;
        end;
        EH: Error;
        LH: begin
            w := x^.left;
            case w^.bf of
                EH: begin
                    root^.bf := EH;
                    x^.bf := EH;
                end;
                LH: begin
                    root^.bf := EH;
                    x^.bf := RH;
                end;
                RH: begin
                    root^.bf := LH;
                    x^.bf := EH;
                end;
            end;
            w^.bf := EH;
            RotateRight(x);
            root^.right := x;
            RotateLeft(root);
        end;
    end;

```

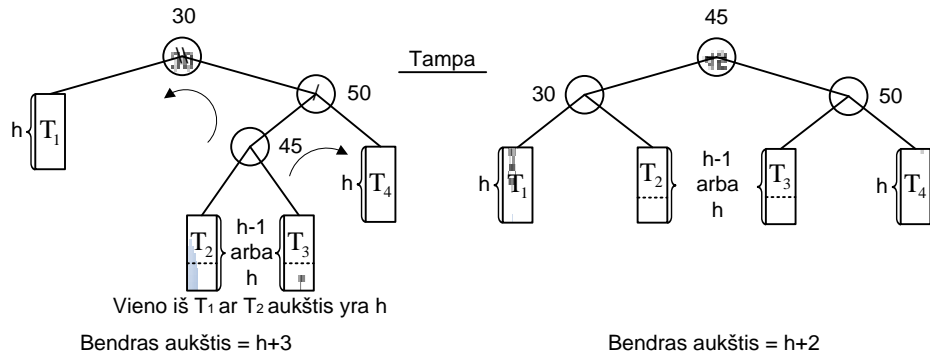


```

taller := false;
end
end
end;

```

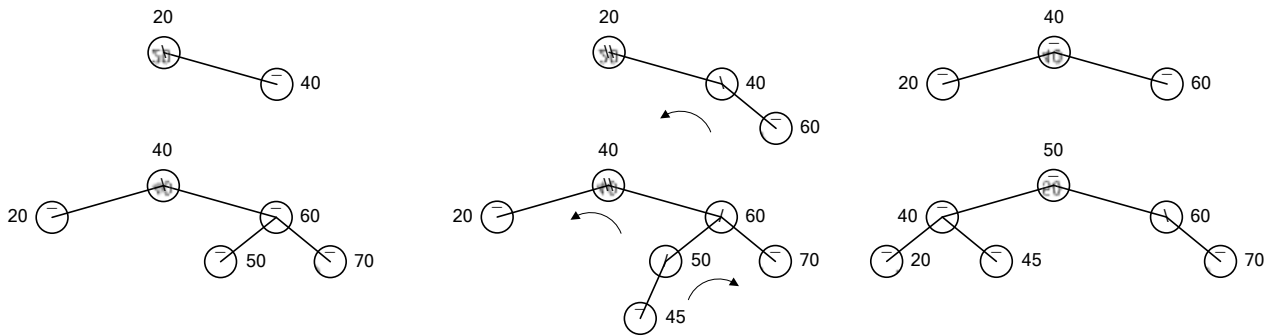
Pav 7. Procedūra, skirta atstatyti balansą



Pav 8. Antra situacija: balanso atstatymas dvigubu pasukimu

Procedūros RotateRight ir LeftBalance yra labai panašios į RotateLeft ir RightBalance atitinkamai. Medžio pasukimus nebūtina atlikti po kiekvienos įterpimo ar pašalinimo operacijos.

**Pav 10** vaizduoja algoritmo veikimą, įterpiant naujus elementus. Kaip galima matyti, medis sukamas tik tuo atveju, jeigu jis praranda AVL-medžio savybes. Šituo atveju, priklausomai nuo situacijos, vyksta arba viengubas, arba dvigubas pasukimas.



Pav 9. Algoritmo veikimo demonstravimas

Didžiausias AVL-medžių privalumas, kad jų aukštis visada labai artimas teoriniam minimumui  $\lceil \log_2(n+1) \rceil$ .

## B-medžiai

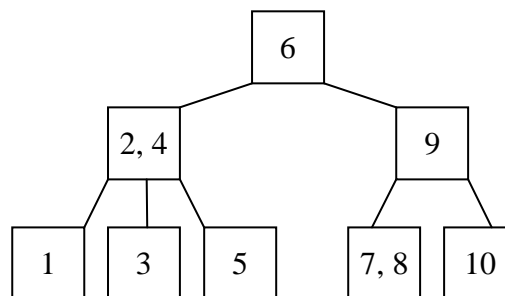
B-medžius pasiūlė R. Bayer ir M.C. McCreight 1972 metais. Jie ir buvo pavadinti autoriaus R. Bayer vardu.

B-medžiai – tai medžiai, kažkiek panašūs į dvejetainius paieškos medžius, nes jų tikslas užtikrinti efektyvią paiešką. Tačiau jie nėra nei dvejetainiai paieškos medžiai, nei iš viso dvejetainiai medžiai, kadangi kiekviena viršūnė gali turėti daugiau negu du vaikus. Taip pat kiekviena viršūnė gali turėti daugiau negu vieną reikšmę (angl. *entry*). B-medžiai gali būti naudojami tiek reikšmių aibei (pasikartojančios reikšmės yra negalimos), tiek reikšmių rinkiniui (pasikartojančios reikšmės yra leistinos) saugoti.

B-medžiai (reikšmių aibės atveju) yra nusakomi tokiomis taisyklėmis: Kiekvienas B-medis priklauso nuo teigiamos konstantos  $\text{MINIMUM} > 0$ , kuri nusako, kiek reikšmių gali turėti viršūnė.

1. Šaknis turi turėti ne mažiau negu 1 reikšmę (0 reikšmių, jei šaknis neturi vaikų). Kiekviena kita viršūnė (ne šaknis) turi turėti **ne mažiau negu  $\text{MINIMUM}$  reikšmių**.
2. Kiekviena viršūnė turi turėti **ne daugiau negu  $\text{MINIMUM} * 2$  reikšmių**.
3. Kiekvienos viršūnės reikšmės gali būti laikomos masyve, **reikšmės turi būti surūšiuotas didėjimo tvarka**, t. y.  $m[0] < m[1]$  ir t. t.
4. Kiekviena viršūnė (išskyrus lapus) **turi turėti (viršūnės\_reikšmių\_skaicius + 1) vaiką**. Pvz: jei viršūnė turi 15 reikšmių, tai ji turi turėti 16 vaikų.
5. Kiekvienai viršūnei (išskyrus lapus) galioja šios taisyklės: **i-toji viršūnės reikšmė yra didesnė už visas i-tojo vaiko reikšmes ir i-toji viršūnės reikšmė yra mažesnė už visas (i + 1)-ojo vaiko reikšmes**.
6. Visi B-medžio lapai yra viename aukštyje.

B-Medžio pavyzdys ( $\text{MINIMUM} = 1$ ):



## Reikšmės paieška B-medyje

Reikšmės paieška B-medyje primena paiešką dvejetainiame paieškos medyje. Paieška yra pradedama nuo šaknies. Paieškos B-medyje algoritmas yra toks (viršūnių reikšmės yra saugomos masyve, kuriame pirmojo elemento indeksas yra 0; viršūnės vaikai irgi numeruojami nuo 0):

1. Jei ieškoma reikšmė yra viršūnėje, tai paieška yra baigta. Reikšmė medyje yra.
2. Jei ieškomos reikšmės nėra viršūnėje ir viršūnė neturi vaikų, tai paieška taip pat yra baigta. Reikšmės medyje nėra.
3. Jei ieškomos reikšmės nėra viršūnėje ir viršūnė turi vaikų, tai:

- a. Reikia tikrinti visas nagrinėjamoje viršūnėje esančias reikšmes iš eilės tol, kol surasime reikšmę, kuri yra didesnė už ieškomą reikšmę. Rastos reikšmės poziciją pažymėkime indeksu I. (Jei ieškoma reikšmė yra didesnė negu visos viršūnėje esančios reikšmės, tai  $I = \text{Reikšmių\_skaičius\_viršūnėje}$ )
- b. Po to reikia keliauti į I-tąją nagrinėjamos viršūnės vaiką ir viską kartoti iš pradžių.

Reikšmės paieškos funkcijos pseudokodas yra toks:

```
I = Get_index(data);
```

```
/*Funkcija Get_index() tikrina visas nagrinėjamoje viršūnėje esančias reikšmes iš eilės tol, kol suranda reikšmę, kuri yra ne mažesnė už ieškomą reikšmę ir grąžina jos indeksą. Jei ieškoma reikšmė yra didesnė negu visos viršūnėje esančios reikšmės, tai funkcija grąžina reikšmių skaičių nagrinėjamoje viršūnėje; masyve data yra saugomos visos viršūnėje esančios reikšmės.*/
```

```
If (ieškoma reikšmė yra nagrinėjamoje viršūnėje)
```

```
Return I;
```

```
Else if (nagrinėjama viršūnė neturi vaikų)
```

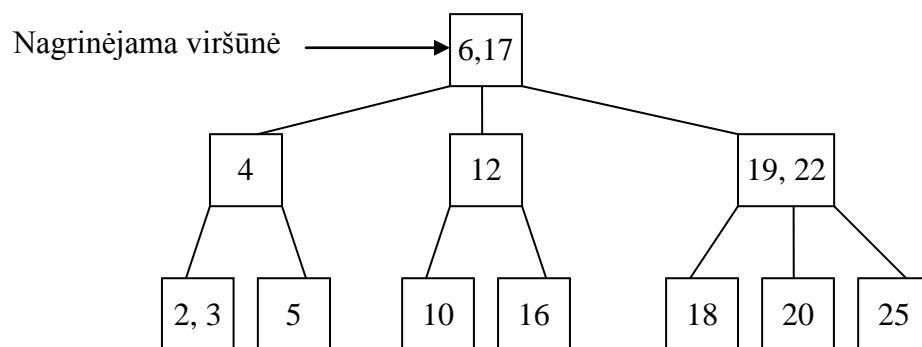
```
Return 0;
```

```
Else
```

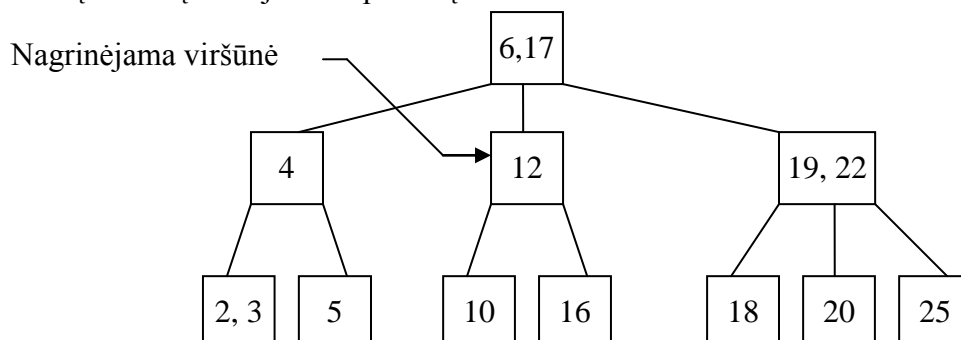
```
Return subset[i]->count(target);
```

```
/*subset – rodyklių masyvas, kur kiekviena rodyklė rodo į nagrinėjamos viršūnės vaikus; funkcija count() grąžina 0 – jei ieškomos reikšmės nėra nagrinėjamoje viršūnėje, 1 – jei ieškoma reikšmė yra nagrinėjamoje viršūnėje. Taigi šis sakinytis rekursiškai iškviečia paieškos funkciją ir paieška yra tęsiama I-tajame nagrinėjamos viršūnės vaike.*/
```

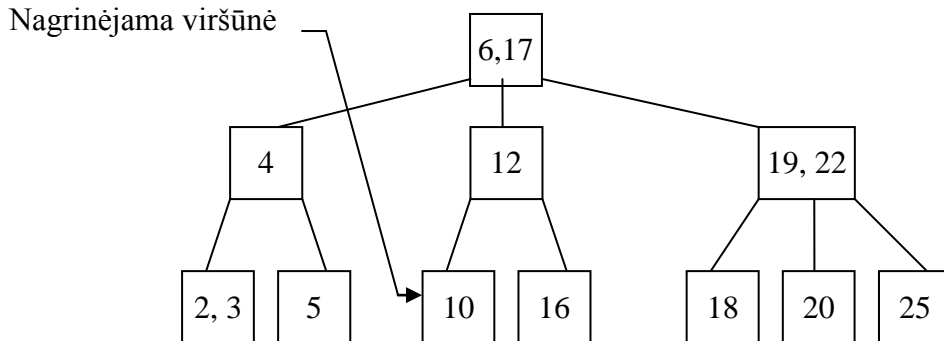
Pvz.: turime tokį B-Medį ir jame norime surasti reikšmę 10



Šaknyje yra reikšmės 6 ir 17. Kadangi reikšmės 10 šaknyje nėra ir šaknis turi vaikų, tai tikriname visas šaknyje esančias reikšmes iš eilės tol, kol surandame reikšmę, kuri yra didesnė už ieškomą reikšmę. Ta reikšmė yra 17. Ji yra 1-je pozicijoje. Taigi  $I = 1$ . Toliau keliaujame į 1-ąjį šaknies vaiką ir viską kartojame iš pradžių.



Dabar nagrinėjamoje viršūnėje yra reikšmė 12. Kadangi ieškomos reikšmės joje nėra ir nagrinėjama viršūnė turi vaikų, tai tikriname visas šaknyje esančias reikšmes iš eilės tol, kol surandame reikšmę, kuri yra didesnė už ieškomą reikšmę. Ta reikšmė yra 12. Ji yra 0-je pozicijoje. Taigi  $I = 0$ . Toliau keliaujame į 0-ąją nagrinėjamos viršūnės vaiką ir viską kartojame iš pradžių.



Dabar nagrinėjamoje viršūnėje yra reikšmė 10. Kadangi tai ir yra mūsų ieškoma reikšmė, tai paieška medyje yra baigta – reikšmė rasta.

### Reikšmės įterpimas į B-medį

Laisvo (angl. *loose*) reikšmės įterpimo į medį algoritmas yra toks (medis pradedamas nagrinėti nuo šaknies; viršūnių reikšmės yra saugomos masyve, kuriame pirmojo elemento indeksas yra 0; viršūnės vaikai irgi numeruojami nuo 0):

1. Jei reikšmė, kurią norime įterpti, yra nagrinėjamoje viršūnėje, tai įterpimas yra nutraukiamas, nes 2 vienodų reikšmių medyje negali būti.
2. Jei nagrinėjama viršūnė neturi vaikų, tai reikšmė, kurią norime įterpti, yra įterpiama į tą viršūnę.
3. Jei nagrinėjama viršūnė turi vaikų, tai:
  - a. Reikia tikrinti visas nagrinėjamoje viršūnėje esančias reikšmes iš eilės tol, kol surasime reikšmę, kuri yra didesnė už reikšmę, kurią norime įterpti. Rastos reikšmės poziciją pažymėkime indeksu  $I$ . (Jei reikšmė, kurią norime įterpti, yra didesnė negu visos viršūnėje esančios reikšmės, tai  $I = \text{Reikšmių\_skaičius\_viršūnėje}$ )
  - b. Po to reikia keliauti į  $I$ -tąją nagrinėjamos viršūnės vaiką ir viską kartoti iš pradžių.

Laisvo reikšmės įterpimo į medį funkcijos *Loose\_insert(Reikšmė\_kurią\_norime\_įterpti)* pseudokodas yra toks:

```
I = Get_index(data);
/*Funkcija Get_index() tikrina visas nagrinėjamoje viršūnėje esančias reikšmes iš eilės tol,
kol suranda reikšmę, kuri yra ne mažesnė už reikšmę, kurią norime įterpti, ir grąžina jos
indeksą. Jei ieškoma reikšmė yra didesnė negu visos viršūnėje esančios reikšmės, tai
funkcija grąžina reikšmių skaičių nagrinėjamoje viršūnėje; masyve data yra saugomos visos
viršūnėje esančios reikšmės.*/
If (reikšmė, kurią norime įterpti, yra nagrinėjamoje viršūnėje)
    Return false;
Else if (nagrinėjama viršūnė neturi vaikų) {
    reikšmę įterpiame data[I], prieš tai visas dešiniau esančias reikšmes pastūmę į dešinę
    Return true; }
Else {
    Subset[I]->Loose_insert(reikšmė_kurią_norime_įterpti);
```

```
/* subset – rodyklių masyvas, kur kiekviena rodyklė rodo į nagrinėjamos viršūnės  
vaikus; šiuo sakiniu rekursiškai iškviečiama laisvo elemento įterpimo į medį funkcija ir  
elementą bandoma įterpti į I-tąją nagrinėjamos viršūnės vaiką*/  
If (viršūnėje yra reikšmių perteklius)  
    Fix_excess(); //funkcija Fix_excess() pašalina reikšmių perteklių viršūnėje  
}
```

Kadangi šis algoritmas nepaiso antrosios B-medžių taisyklės, tai įterpus reikšmę gali atsirasti taip, kad viršūnėje, į kurią buvo įterpta reikšmė, bus ( $\text{MINIMUM} * 2 + 1$ ) reikšmių.

Norint pašalinti reikšmių perteklių, reikia:

1. Vidurinę viršūnės reikšmę perkelti vienu lygiu aukštin.
2. Viršūnę padalinti į dvi lygias dalis, kuriose būtų po  $\text{MINIMUM}$  reikšmių.

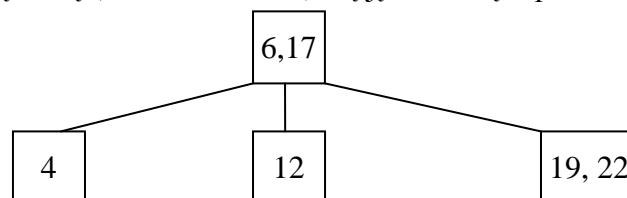
Taigi pilnas reikšmės įterpimo į B-medį algoritmas yra toks:

1. Įterpiame norimą reikšmę į medį, pasinaudoję laisvuju reikšmės įterpimo į medį algoritmu.
2. Jei reikšmės įterpti nepavyko, grąžiname FALSE.
3. Jei kurioje nors viršūnėje yra reikšmių perteklius, tai pašaliname jį.
4. Grąžiname TRUE.

Reikšmės įterpimo į medį pilnos funkcijos pseudokodas yra toks:

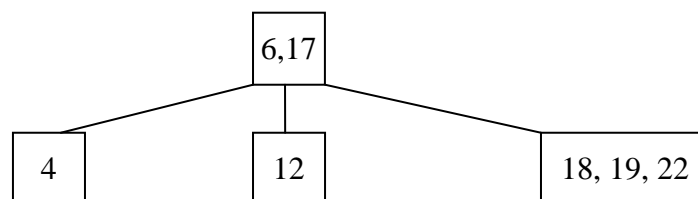
```
If (!Loose_insert(reikšmė_kurią_norime_įterpti))  
    Return false;  
If (data_count > MAXIMUM) {  
    /*data_count – reikšmių skaičius viršūnėje; MAXIMUM = MINIMUM * 2; */  
    Pašalinti reikšmių perteklių;  
    Return true;  
}
```

Pvz.: turime tokį medį ( $\text{MINIMUM} = 1$ ) ir į jį norime įterpti reikšmę 18.



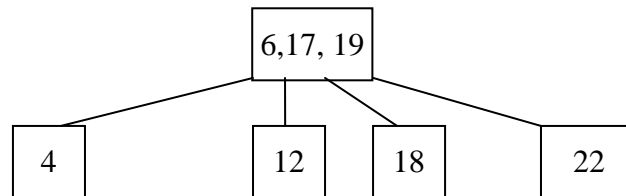
Kadangi šaknis turi vaikų, tai tikriname visas šaknyje esančias reikšmes iš eilės tol, kol surasime reikšmę, kuri yra didesnė už reikšmę, kurią norime įterpti. Kadangi reikšmė, kurią norime įterpti, yra didesnė už visas šaknyje esančias reikšmes, tai  $I = 2$ . Toliau nagrinėjame 2-ąją šaknies vaiką ir viską kartojame iš pradžių.

Kadangi nagrinėjama viršūnė neturi vaikų, tai į ją įterpiame norimą reikšmę ir gauname tokį medį:

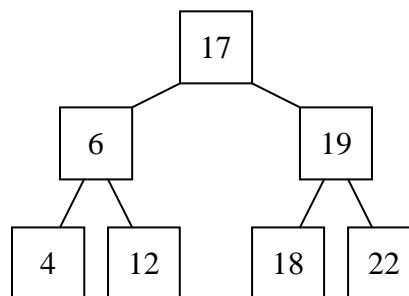


Kadangi viename iš lapų yra reikšmių perteklius, tai vidurinę reikšmę (šiuo atveju ji yra 19) pakeliame vienu lygiu aukštyr ir lapą padaliname į dvi dalis po 1 reikšmę, kadangi  $\text{MINIMUM} = 1$ .

Gauname tokį medį:



Kadangi šaknyje yra reikšmių perteklius, vidurinę reikšmę (šiuo atveju ji yra 17) pakeliame vienu lygiu aukštyr ir šaknį padaliname į dvi dalis po 1 reikšmę. Taip atsiranda nauja šaknis ir gauname tokį medį:



### Reikšmės pašalinimas iš B-Medžio

Reikšmę iš B-Medžio pašalinam tokiu būdu:

*Pred-sąlyga:* Visas B-medis – tvarkingas (*valid*).

*Post-sąlyga:* Jei reikšmės, kurią norime išmesti, nebuvo medyje, tai medį paliekam nepakeistą, antraip reikšmė išmetama, grąžinama reikšmė – true ir visas medis yra tvarkingas. **Išskyrus** tą atvejį, kai reikšmių šaknyje yra viena mažiau nei viena (t.y. 0).

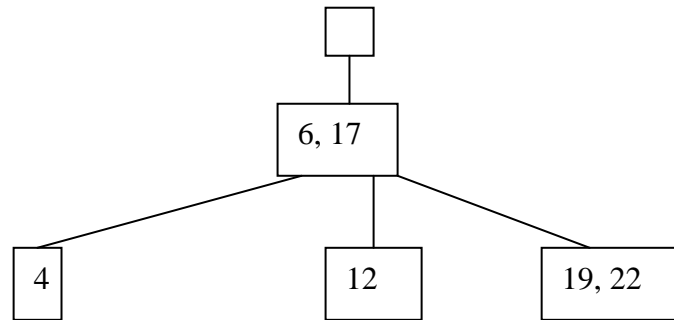
Šią funkciją pavadinkime *loose\_erase*.

Atlikus šiuos veiksmus reikia patikrinti, ar neliko šaknies be reikšmių ir su vienu vaiku, ir sutvarkyti, jei blogai. Taigi šių žingsnelių pseudokodas:

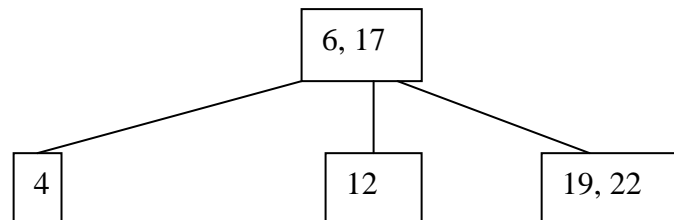
```
if (!loose_erase(target)) // target – tai reikšmė, kurią norime pašalinti
    Grąžinti false, nes neištrynė reikšmės.
if (( reikšmių_kiekis == 0 ) && ( vaikų_kiekis == 1 ))
    Sutvarkyti medį taip, kad jis neturėtų nulinių reikšmių.
    Grąžinti true, kai viena reikšmė pašalinta.
```

Kaip sutvarkyti, kad medis neturėtų tuščių viršūnių?

Tarkime, gavome tokį medį:



Galimas toks sprendimas: sukuriame rodyklę, kuri rodo į tą vienintelį vaiką; tada nukopijuojame viską iš to vienintelio vaiko į šaknį ir pašaliname vaiką. Po šių pertvarkymų, medis bus vienu lygiu žemesnis:



Šis sumažėjimas ties šaknim yra vienintelis atvejis, kai B-Medis praranda savo aukštį.

### Loose\_erase funkcijos realizavimas

*Loose\_erase* funkcija prasideda taip pat, kaip ir paieškos bei įterpimo funkcijos, randant pirmąjį indeksą *I* tokį, kad *data[I]* nebūtų mažesnis už reikšmę, kurią norime pašalinti. Radus indeksą, galimi tokie keturi veiksmų variantai:

1. Kintamąjį *i* padarom lygų tokiam pirmajam indeksui, kad *data[i]* nebūtų mažesnis už reikšmę, kurią norime pašalinti. Jei tokio indekso nėra, tai *i = data\_count*, parodo, kad visos reikšmės yra mažesnės už reikšmę, kurią norime pašalinti.
2. Susiduriame su šiais galimais atvejais:
  - 2a. Šaknis neturi vaikų, ir mes neradome reikšmės, kurią norime pašalinti: šiuo atveju nėra ką daryt, grąžinam *false*.
  - 2b. Šaknis neturi vaikų, ir mes radome reikšmę, kurią norime pašalinti: šiuo atveju pašaliname reikšmę iš duomenų masyvo ir grąžinam *true*.
  - 2c. Šaknis turi vaikų, ir mes neradom reikšmės, kurią norime pašalinti (žr. žemiau).
  - 2d. Šaknis turi vaikų, ir mes radome reikšmę, kurią norime pašalinti (žr. žemiau).

2c ir 2d atvejai nagrinėjami atskirai:

**2c atvejis.** Šiuo atveju šaknyje neradome reikšmės, kurią norime pašalinti, bet ji dar gali būti *subset[i]*. Darom rekursiją:

*subset[i]* -> *loose\_erase(target)*

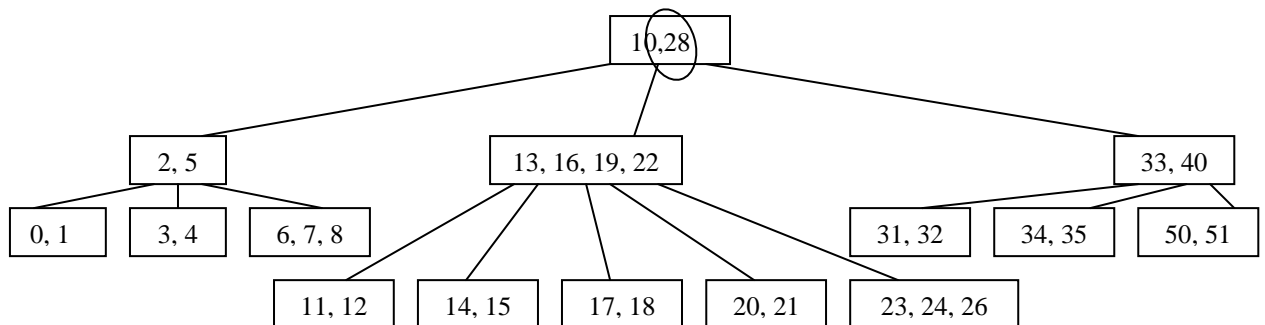
Toks iškvietaimas išmes reikšmę iš *subset[i]*, bet problema kita – *subset[i]* šaknis gali turėti tik vieną reikšmę. Tokiu atveju sutvarkome viską, iškvietsdami tokią funkciją, kurią galime pavadinti *fix\_shortage*:

*Pred-sąlyga:* ( $i < \text{vaikų\_skaičius}$ ) ir visas B-medis yra tvarkingas, išskyrus tą atvejį, kai  $\text{subset}[i]$  turi MINIMUM - 1 reikšmę.

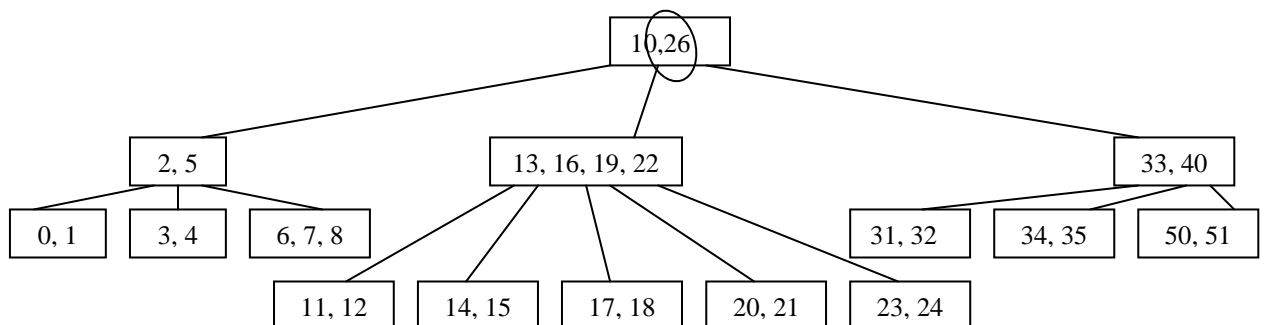
*Post-sąlyga:* medis pertvarkomas taip, kad visas B-medis yra tvarkingas, išskyrus tą atvejį, kai reikšmių kiekis šaknyje yra mažesnis nei MINIMUM.

**2d atvejis.** Šiuo atveju mes radome reikšmę, kurią norime pašalinti, šaknyje, bet mes negalim jos taip paprastai išmesti iš duomenų masyvo, nes ji turi vaikų. Vietoj to mes einame į  $\text{subset}[i]$  ir pašaliname didžiausią reikšmę iš čia. Nukopijuojame šią reikšmę į  $\text{data}[i]$  (kuris saugojo reikšmę, kurią norime pašalinti). Tokiu būdu pašalinome reikšmę.

Tarkime, radome elementą 28  $\text{data}[i]$  masyve, šio B-medžio šaknyje:



Mūsų planas yra toks: reikia nueiti į  $\text{subset}[i]$ , išmesti didžiausią reikšmę (26) ir įrašyti išmestą reikšmę vietoj reikšmės, kurią norime pašalinti. Po šių žingsnių, B-medis daugiau nebeturės reikšmės 28.



Visas darbas (išmesti didžiausią reikšmę iš  $\text{subset}[i]$  ir nukopijuoti ją į  $\text{data}[i]$ ) gali būti padarytas, iškvietus kitą funkciją, kurią galim pavadinti *remove\_biggest*:

*Pred-sąlyga:* ( $\text{duomenų\_kiekis} > 0$ ) ir visas B-Medis yra tvarkingas (*valid*).

*Post-sąlyga:* didžiausia reikšmė išmetama ir nukopijuojama į *removed\_entry*.

visas B-medis yra tvarkingas, išskyrus tą atvejį, kai reikšmių kiekis šaknyje yra mažesnis nei MINIMUM.

Naudojantis *remove\_biggest* funkcija, beveik visas 2d žingsnis padaromas iškvietus vieną funkciją:

*subset[i] -> remove\_biggest(data[i]);*

Įvykdydami šią funkciją mes pašalinome didžiausią reikšmę iš  $\text{subset}[i]$  ir nukopijavome ją į  $\text{data}[i]$  (įrašėme ją vietoje tos reikšmės, kurią norėjome pašalinti).



Be viso šito, subset[i] šaknyje dar gali atsirasti vienas *trūkumas*, nes gali būti (dėka funkcijos subset[i]->remove\_biggest postsąlygos), kad subset[i] šaknis liks su MINIMUM – 1 reikšmių. Todėl turime sutvarkyti šitą trūkumą: galime daryt taip, kaip darėm 2c žingsnyje, panaudodami funkciją fix\_shortage. Taigi 2d žingsnio pseudokodas yra toks:

```
subset[i]->remove_biggest(data[i]);  
if( subset[i]->data_count < MINIMUM )  
    fix_shortage(i);  
return true; //true reiškia, kad sėkmingai pašalinome reikšmę.
```

### Funkcija, kuri ištaiso vaiko trūkumą fix\_shortage

Kai iškviečiama fix\_shortage(i) funkcija, žinome, kad subset[i] turi tik MINIMUM – 1 reikšmių. Bandant taisyti šį trūkumą, susiduriame su tokiomis keturiomis situacijomis:

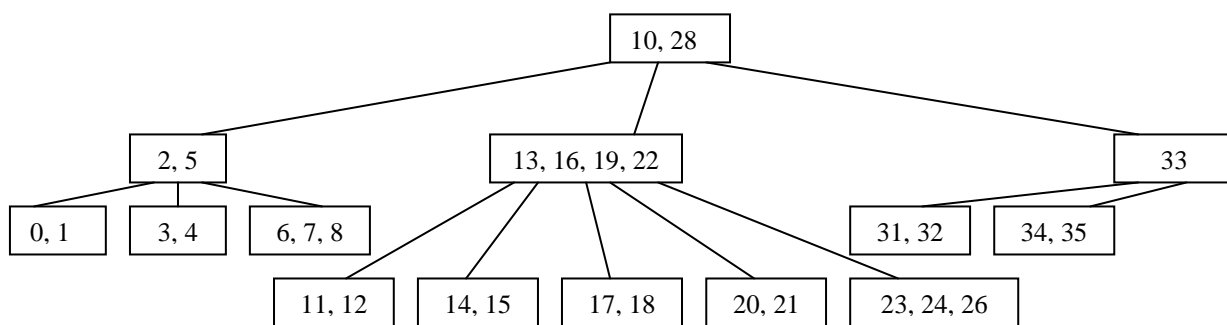
#### 1 atvejis: Persiūsti papildomą reikšmę iš subset[i-1].

Tarkim, kad subset[i-1] turi daugiau negu MINIMUM reikšmių.

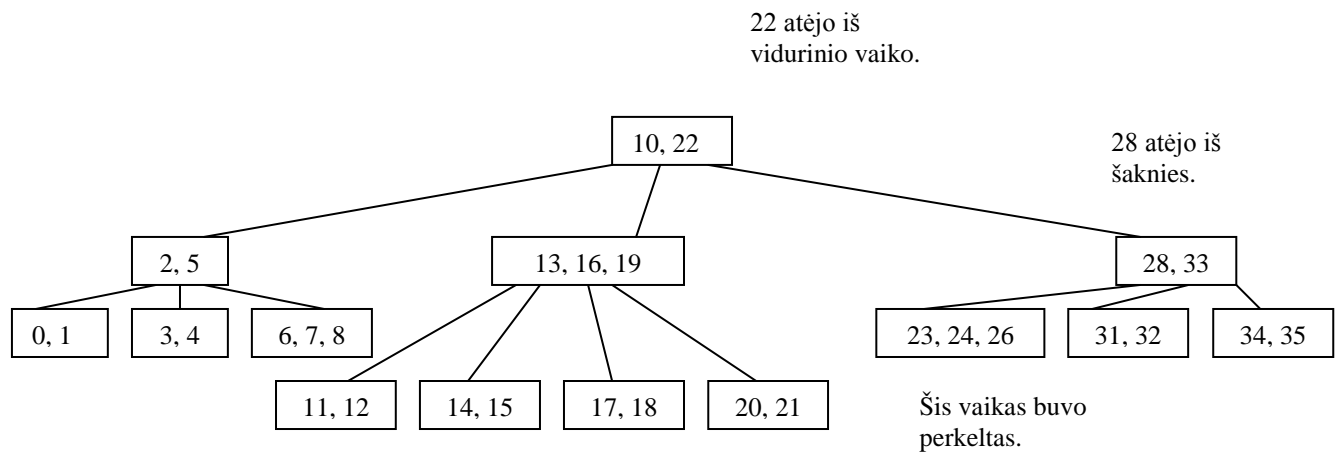
Tada mes galime atlikti tokius persiuntimus:

- Persiūsti data[i-1] į subset[i]->data priekį.  
Reikia perstumti esančias reikšmes tam, kad atlaisvinti vietos ir pridėti vieną prie subset[i]->data\_count.
- Persiūsti galutinę subset[i-1]->data reikšmę į data[i-1] ir atimti vieną iš subset[i-1]->data\_count.
- Jei subset[i-1] turi vaikų, persiūsti paskutinį vaiką į subset[i] priekį.  
Taip atliekamas persistūmimas per egzistuojančią eilę subset[i]->subset tam, kad atlaisvinti vietą naujo vaiko rodyklei subset[i]->subset[0]. Taip pat pridedam vieną prie subset[i]->vaikų\_kiekis ir atimam vieną iš subset[i-1]->vaikų\_kiekis.

Pavyzdžiui, iškviessim funkciją fix\_shortage (2) šiam medžiui (MINIMUM = 2 ):



Šiame pavyzdyje mums reikia sutvarkyt subset[2], nes jis turi tik 1 reikšmę. Pagal aukščiau pateiktą aprašą mes galime persiūsti reikšmę iš subset[1]. Reikšmė 22 persiunčiama į data[1], o 28 (iš data[1]) nukeliamas į pirmąją reikšmę turimoje rodyklėje (ten, kur tik viena reikšmė). Vienas vaikas taip pat persiunčiamas iš subset[1] galo į subset[2] priekį:



## 2 atvejis: Persiųsti papildomą reikšmę iš subset[i+1].

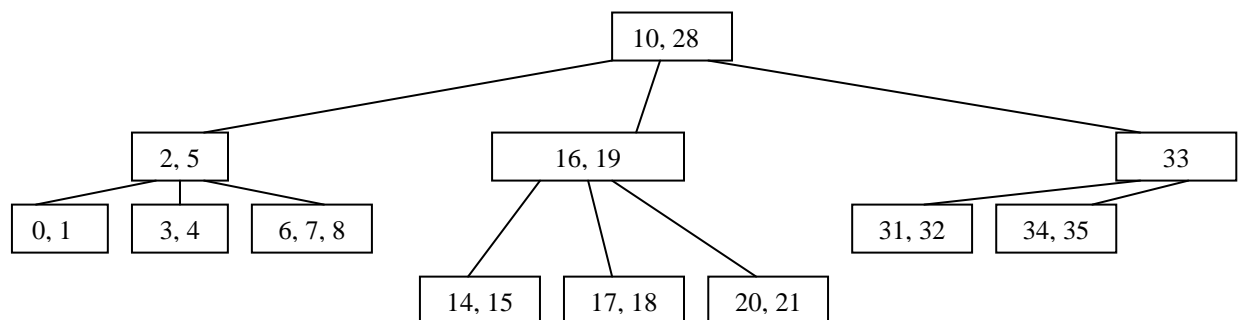
Šis atvejis panašus į reikšmės persiuntimą iš subset[i-1].

## 3 atvejis: Subset[i] jungimas su subset[i-1].

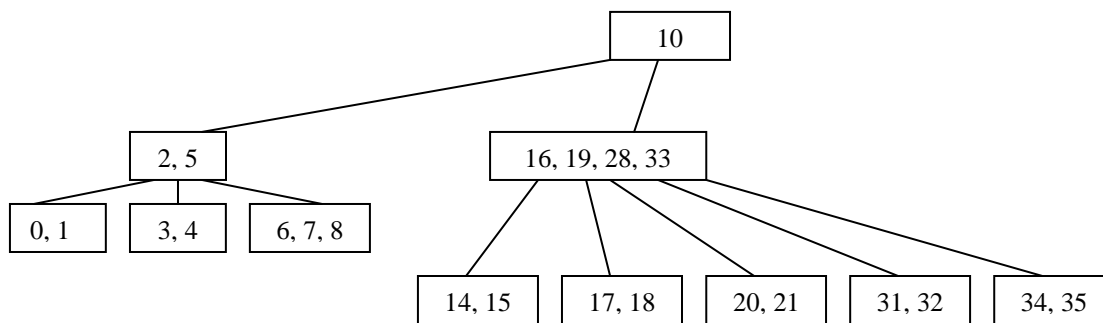
Tarkim yra subset[i-1] (kitais žodžiais  $i > 0$ ), bet turi tik MINIMUM įrašų. Šiuo atveju mes negalim persiųsti įrašo iš subset[i-1], bet galime sujungti subset[i] su subset[i-1]. Jungimas atliekamas trim žingsniais:

- Persiunčiam data[i-1] į subset[i-1]->data galą. Toks veiksmas iš tikro išmeta reikšmę iš šaknies, taigi perstumiam data[i], data[i+1] ir taip toliau į kairę tam, kad užpildytume likusį tarpą. Taip pat atimame vieną iš data\_count ir pridedam vieną prie subset[i-1]->data\_count.
- Persiunčiam visas reikšmes ir vaikus iš subset[i] į subset[i-1] galą. Atnaujinam subset[i-1]->data\_count ir subset[i-1]->vaikų\_kiekis reikšmes. Taip pat subset[i]->data\_count ir subset[i]->vaikų\_kiekis turi būti lygūs nuliui.
- Ištrinam subset[i] rodyklę, perstumiam subset[i+1], subset[i+2] ir taip toliau į kairę tam, kad užpildytume likusį tarpą. Taip pat sumažinam vaikų\_kiekis vienu.

Pavyzdžiui, iškviečiame funkciją *fix\_shortage* (2) šiam medžiui (MINIMUM = 2):



Šiame pavyzdyje subset[2] sujungiamas su jo broliu į kairę. Per šį jungimą 28 nusiunčiamas iš šaknies į sujungtų rodyklių naujos reikšmės vietą. Gautas rezultatas būtų toks:



Kaip matome, šis medis turi per mažai reikšmių savo šaknyje, bet tai nesukelia problemų, nes *fix\_shortage* funkcijos postsąlyga leidžia šakniai turėti viena reikšmę mažiau negu reikia.

#### 4 atvejis: **Subset[i] ir subset[i+1] sujungimas.**

Šis atvejis panašus į prieš tai nagrinėtą.

### Didžiausios reikšmės išmetimas iš B-Medžio

Funkcija, kuri išmetą didžiausią reikšmę *remove\_biggest* atrodytų taip:

*Presąlyga:* (*data\_count* > 0) ir visas B-Medis tvarkingas (*valid*).

*Postsąlyga:* didžiausia reikšmė išmesta ir išmestos reikšmės kopija įdedama į *removed\_entry*. Visas B-medis yra tvarkingas, išskyrus tą atvejį, kai reikšmių kiekis šaknyje yra mažesnis nei MINIMUM.

Jei šaknis neturi vaikų, tai *remove\_biggest* veikimas toks: nukopijuojame didžiausią reikšmę į *removed\_entry*. Atimame vieną iš *data\_count* ir viskas. Gali būti, kad liksime su rodykle, kuri turi mažiau negu MINIMUM reikšmių, bet pagal postsąlygą, tai leidžiama.

Jei šaknis turi vaikų: darome rekursiją tam, kad pašalintume didžiausią reikšmę iš dešiniausio vaiko:

*subset[vaikų\_kiekis - 1] -> remove\_biggest(removed\_entry);*

*removed\_entry* bus lygus tai reikšmei, kurią išmes *remove\_biggest* funkcija.

Liko tik viena bėda – ši rekursija gali palikti *subset[vaikų\_kiekis - 1]* šaknį su viena reikšme per mažai. Vėl naudojame tą pačią trūkumus tvarkančią funkciją *fix\_shortage*.

Taigi po rekursijos reikia patikrinti reikšmių *subset[vaikų\_kiekis - 1]* šaknyje kiekį. Jei reikšmių kiekis mažesnis už MINIMUM, tai iškviečiame funkciją *fix\_shortage(vaikų\_kiekis - 1)*.

## **Išoriniai B-medžiai**

Programuotojai naudoja B-medžius, nes tokie medžiai yra visą laiką subalansuoti (visi lapai viename gylyje). Jeigu subalansuotas medis vienintelė tavo užduotis, tai konstantos MINIMUM pasirinkimas nėra toks svarbus. Kartais MINIMUM yra tiesiog 1, kas reikštų, kad kiekviena viršūnė turi vieną arba dvi reikšmes ir (jei ji ne lapas) 2 arba 3 vaikus. Toks atvejis vadinamas **2-3 medžiu** (*2-3 tree*).

Kartais B-medžiai tampa labai dideli, tokie dideli, kad visas medis niekaip netelpa kompiuterio operatyvioje atmintyje vienu metu. Pavyzdžiui, JAV gyventojų telefonų knygą sudaro 100 milijonų įrašų. Tiek įrašų vis tiek gali būti organizuojami B-medžiu, bet dauguma viršūnių turi būti saugoma lėtesnėje antrinėje atminties laikmenoje, pavyzdžiui, kietuosiuose ar kompaktiniuose diskuose. Paprastai šaknis būna pakrauta į operatyvią (darbinę) atmintį, kur ji pasilieka visą laiką (kol vykdoma programa). Bet kitos viršūnės turi būti pakrautos į operatyvią atmintį tik tada, kai jų prireikia. Toks atvejis vadinamas **išoriniu B-medžiu**.

Kuriant išorinį B-medį, visų pirmą, ką reiktų apgalvoti, tai kuo labiau sumažinti kreipimus į antrinės atminties įrenginius. Dėl to MINIMUM konstanta turi būti parinkta pakankamai didelė. Kai MINIMUM = 1000, mes galime saugoti daugiau nei 1 milijardą įrašų tik su viena šaknimi ir 2 lygiais. Šaknis visą laiką būna operatyvioje atmintyje, todėl jokia paieška nereikalaus daugiau nei 2 kreipinių į antrinės atminties įrenginius.

Mus domina dar vienas faktorius – grąžinimo mechanizmas, naudojamas antrinėje laikmenoje. Kai įvykdoma užklausa kietajam diskui ar CD-ROM'ui, tai disko skaitymo galvutė nueina į reikiama vietą lėtai, palyginus su duomenų skaitymu. Kai galvutė jau nutaikyta reikiamam vieton, skaitymas vyksta pakankamai greitai. Šie duomenys siunčiami fiksuoto dydžio paketais dideliu nepertraukiamo persiuntimo greičiu. Pavyzdžiui, 40x CD-ROM'o galvutė suranda reikiamą vietą per 1/10 sekundės, o jau tada duomenys siunčiami 6 milijonų baitų per sekundę greičiu. Turint tai omenyje, būtina užtikrinti, kad informacija būtų saugoma nuosekliai diske, nes kiekvienai nutolusiai rodyklei surast prireiks 1/10 sekundės.

## Duomenų struktūros

Panagrinėsime keletą žinomų ir įvairiuose taikymuose naudojamų duomenų struktūrų. Priklausomai nuo pasirinktos programavimo kalbos, joje gali būti atitinkamas duomenų tipas (pavyzdžiui, LISP kalboje yra duomenų tipas sąrašas) arba gali tekti konstruoti reikiamą duomenų struktūrą, naudojantis kitais programavimo kalbos pateikiamais duomenų tipais.

Nagrinėdami duomenų struktūras, kartu apibrėšime ir darbui su jomis reikalingas operacijas bei keletą būdų, kaip tos duomenų struktūros gali būti realizuotos Pascal kalboje.

### Sąrašas

**Sąrašas** arba **tiesinis sąrašas** (*angl. list or linear list*) yra sutvarkytas rinkinys (tiesinė seka) elementų, struktūrizuotų taip, kad kiekvienas elementas, išskyrus pirmą, turi vienintelį prieš jį einantį elementą ir kiekvienas elementas, išskyrus paskutinį, turi vienintelį po jo einantį elementą. Kiekvienas sąrašo elementas saugo tam tikrus duomenis. Sąrašo pradžia ir pabaiga dažnai dar vadinamos **galva** (*angl. head*) ir **uodega** (*angl. tail*) atitinkamai.

#### Operacijos:

- Sukurti tuščią sąrašą
- Patikrinti, ar sąrašas tuščias
- Patikrinti, ar sąrašas pilnas (teoriškai sąrašas gali būti bet kokio ilgio, bet praktinėse realizacijose sąrašo ilgis būna daugiau ar mažiau ribotas, priklausomai nuo pasirinkto sąrašo realizavimo būdo)
- Suskaičiuoti sąrašo elementus
- Gauti n-tojo sąrašo elemento duomenis
- Įterpti naujus duomenis (naują elementą) prieš n-tąjį elementą
- Panaikinti n-tąjį sąrašo elementą
- Rasti sąrašo elemento numerį su nurodytais duomenimis.
- Išvesti sąrašo elementus

Tai nėra visos operacijos, kurias galima atlikti su sąrašu, bet šis rinkinys yra pakankamas, kad naudojantis šiomis operacijomis būtų galima atlikti bet kokius veiksmus su sąrašu. Priklausomai nuo konkretaus taikymo gali būti parinktas ir kitas pakankamas operacijų rinkinys arba įvestos papildomos operacijos palengvinančios darbą, pavyzdžiui:

- Įterpti naujus duomenis (naują elementą) po n-tojo elemento
- Įterpti naujus duomenis (naują elementą) sąrašo pradžioje
- Įterpti naujus duomenis (naują elementą) sąrašo pabaigoje
- Panaikinti pirmą sąrašo elementą
- Panaikinti paskutinį sąrašo elementą
- Prijungti kitą sąrašą duoto sąrašo pabaigoje
- Įterpti naujus duomenis (naują elementą) prieš pirmą elementą su nurodytais duomenimis
- Įterpti naujus duomenis (naują elementą) po pirmo elemento su nurodytais duomenimis
- Pereiti prie kito sąrašo elemento
- Įterpti naujus duomenis (naują elementą) prieš einamąjį elementą
- Įterpti naujus duomenis (naują elementą) po einamojo elemento

- Sunaikinti sąrašą (sunaikinti visus sąrašo elementus)

„Nusileidžiant“ arčiau realizacijos galima apibrėžti viapusį ir dvipusį sąrašus.

**Vienpusis sąrašas** – sąrašas, kurio kiekvienas elementas „žino“ tik, koks elementas yra po jo.

**Dvipusis sąrašas** – sąrašas, kurioje kiekvienas elementas „žino“, koks elementas yra po jo ir koks prieš jį.

Vienpusis ir dvipusis sąrašai yra apibendrintos duomenų struktūros sąrašas patikslinimai, jau dalinai nusakantys duomenų struktūros realizaciją: tiek pačią duomenų struktūrą (pavyzdžiui, kiek elementas turi nuorodų), tiek galimas operacijas (pavyzdžiui, operacija "Pereiti prie ankstesnio sąrašo elemento" pakankamai natūrali dvipusio sąrašo atveju; vienpusiam sąrašui tokią operaciją, žinoma, galima realizuoti, bet nėra tikslinga).

### **Realizacija:**

1. Masyvas, kurio elementai tokie patys kaip sąrašo elementai (t.y. elementuose nėra saugoma jokių papildomų nuorodų). Tokiu būdu galima vaizduoti tiek viapusį, tiek dvipusį sąrašą, nes jokios išreikštinės nuorodos nesaugomos, einamasis, kitas ir ankstesnis elementai nustatomi pagal indeksus. Papildomai tereikia žinoti sąrašo elementų skaičių.

Sąrašo pradžia yra pirmas masyvo elementas (jei sąrašas nėra tuščias, elementų skaičius daugiau už 0), sąrašo galas nustatomas pagal elementų skaičių (jei masyvas indeksuojamas nuo 1, tai paskutinis sąrašo elementas yra masyvo elementas su indeksu lygiu elementų skaičiui).

### **Privalumai:**

- maksimaliai paprasta struktūra
- paprasta operacijų realizacija
- laikomi tik patys duomenys (nereikia papildomos atminties nuorodoms saugoti)

### **Trūkumai:**

- masyvo dydis turi būti nusakytas iš anksto, todėl gali būti arba naudojama tik nedidelė jo dalis, arba pritrūkti vietos;
  - elemento įterpimo/naikinimo operacijos yra neefektyvios, nes reikia perstumti kitus elementus.
2. Du masyvai: pirmame saugomi duomenys (t.y. jo elementai tokie patys kaip sąrašo elementai), antro masyvo atitinkamame elemente (su tuo pačiu indeksu) saugoma nuoroda į po jo einantį elementą (jei vaizduojamas vienusis sąrašas) arba nuorodos į po jo ir prieš jį einančius elementus (jei vaizduojamas dvipusis sąrašas). Kadangi Pascal kalboje yra duomenų tipas įrašas, tai galima naudoti ir vieną masyvą, kurio elementai bendru atveju būtų įrašo tipo - sudaryti iš sąrašo duomenų ir nuorodos(ų). (Atskiru atveju, jei sąrašo duomenų tipas sutampa su indeksų tipu, masyvo elementas galėtų būti masyvas arba galima būtų naudoti dvimatį masyvą).

Papildomai būtina žinoti sąrašo pradžios (pirmo jo elemento) indeksą masyve (arba 0, jei sąrašas tuščias). Sąrašo galo nustatymui gali būti arba (1) saugomas sąrašo elementų skaičius, arba (2) paskutinio sąrašo elemento indeksas masyve, arba (3) sąrašo galas gali būti nustatomas pagal tai, kad elementas neturi nuorodos į po jo einantį elementą

(nuorodos nebuvimas turi būti žymimas specialia reikšme, pavyzdžiui, 0, jei masyvas indeksuojamas nuo 1).

Kad prireikus įterpti naujus duomenis, laisvo elemento paieška būtų efektyvesnė, tradiciškai nuorodų masyve saugomas ne tik užimtų elementų sąrašas, bet ir laisvų elementų sąrašas.

Privalumai (lyginant su 1-u būdu):

- elemento įterpimas/naikinimas pakankamai efektyvus, nes nereikia perstumti kitų elementų.

Trūkumai (lyginant su 1-u būdu):

- operacijų realizacija gerokai sudėtingesnė
- reikia papildomos atminties nuorodoms saugoti.

Trūkumai:

- masyvo (tuo pačiu ir sąrašo) dydis turi būti nusakytas iš anksto, todėl gali būti arba naudojama tik nedidelė jo dalis, arba pritrūkti vietos.

3. Dinaminis sąrašas: tokio sąrašo elementai būtų įrašo tipo, sudaryti iš sąrašo duomenų ir vienos (vienpusio sąrašo atveju) arba dviejų (dvipusio sąrašo atveju) rodyklių.

Privalumai:

- atmintis naudojama tik egzistuojantiems sąrašo elementams, t.y. nerezervuojama vieta potencialiems elementams;
- elemento įterpimas/naikinimas pakankamai efektyvus.

Trūkumai

- operacijų realizacija sudėtingesnė (ji palyginama su 2-o būdo operacijų realizacija).
- kiekvienai nuorodai saugoti reikia daugiau atminties (rodyklė užima daugiau atminties nei tipas, kuriuo indeksuojama, pavyzdžiui, sveikų skaičių).

Kai kuriuose taikymuose tikslinga naudoti specifinį sąrašo atvejį ciklinį sąrašą.

**Ciklinis sąrašas** - sąrašas, kuriame po paskutinio elemento „seka“ pirmas sąrašo elementas.

Savo ruožtu ciklinis sąrašas gali būti tiek vienpusis, tiek dvipusis. Jo realizavimui gali būti taikomi anksčiau aptarti būdai.

## Stekas

**Stekas** (*angl. stack*) yra sąrašas, kuriame elementai gali būti įterpiami/naikinami tik jo pradžioje, vadinamoje viršūne (*angl. top*). Taigi stekas yra specifinis sąrašas su apribotomis operacijomis. Tai LIFO (*Last In First Out*) duomenų struktūra, atitinkanti lietuvišką patarlę "kas pirmas į maišą, paskutinis iš maišo". Kiekvienas steko elementas saugo tam tikrus duomenis.

**Operacijos:**

- Sukurti tuščią steką
- Patikrinti, ar stekas tuščias
- Patikrinti, ar stekas pilnas
- Įdėti (*angl. push*) naują elementą į steką
- Išimti (*angl. pop*) elementą iš steko

- Sunaikinti steką

Pastebėjime, kad steko atveju elemento įterpiamo/naikinamo operacijos turi specifinius pavadinimus, labiau atitinkančius jų prasmę. Nagrinėjant apibendrintą steko realizaciją, pirmiausia pažymėjime, kad nėra prasmės jo vaizduoti kaip dvipusio sąrašo.

Kadangi stekas yra atskiras sąrašo atvejis, jo realizacijai galėtų būti panaudoti visi bet kokio sąrašo realizavimo būdai aptarti anksčiau, tačiau naudoti du masyvus (arba masyvą papildytą nuorodomis) nėra prasmės, nes viduriniai steko elementai negali būti „liečiami“. Trumpai aptarsime kitų dviejų būdų privalumus ir trūkumus steko atveju.

**Realizacija:**

1. Masyvas, kurio elementai tokie patys kaip steko elementai. Steko viršūnė būtų paskutinis užpildytas masyvo elementas, t.y. užpildytas masyvo elementas su maksimaliu indeksu.

**Privalumai:**

- maksimaliai paprasta struktūra;
- paprasta operacijų realizacija;
- laikomi tik patys duomenys (nereikia papildomos atminties nuorodoms saugoti).

**Trūkumai:**

- masyvo (tuo pačiu ir steko) dydis turi būti nusakytas iš anksto, todėl gali būti arba naudojama tik nedidelė jo dalis, arba pritrūkti vietos.

Reikia pastebėti, kad elemento įterpimo/naikinimo operacijos steko atveju efektyvios.

2. Dinaminis sąrašas.

**Privalumai:**

- atmintis naudojama tik egzistuojantiems steko elementams, t.y. nerezervuojama vieta potencialiems elementams.

**Trūkumai:**

- operacijų realizacija truputį sudėtingesnė (reikia mokėti dirbti su rodyklėmis);
- reikia daugiau atminties, nes saugomi ne tik duomenys, bet ir nuorodos (rodyklės).

Gyvenime steką atitiktų, pavyzdžiui, automato apkaba (rusiškai ir pati duomenų struktūra taip vadinama - "magazin").

Stekas yra labai populiarus duomenų struktūra informatikoje, daugelyje kompiuterių, tame tarpe ir personaliniuose, stekas yra realizuotas aparatūriniame lygyje ir naudojamas parametrų, perduodamiems į funkcijas/procedūras, ir jų lokaliems kintamiesiems saugoti.

Stekas taip pat gali būti naudojamas, pavyzdžiui, postfixinių (*angl. postfix*) išraiškų skaičiavimui. Mes esame įpratę išraiškas rašyti tokia forma:  $(A + B) * C$ . Ji vadinama infixine (*angl. infix*) forma, nes operacija yra tarp operandų (operandas operacija operandas). Galimos ir kitos išraiškų užrašymo formos: postfixinė (*angl. postfix*), kurioje operacija eina po operandų (operandas operandas operacija), ir prefiksinė (*angl. prefix*), kurioje operacija eina prieš operandus (operacija operandas operandas). Pastarosios formos dar vadinamos lenkiška ir atvirkštine lenkiška forma. Jos ypatingos tuo, kad išraiškose nereikia skliaustų, todėl jas skaičiuoti žymiai paprasčiau. Postfixinių išraiškų pavyzdžiai:

$$A B + C * = (A + B) * C$$

$$5 2 3 * + = 2 * 3 + 5$$



$$5 \ 2 \ * \ 4 \ 2 \ / \ + \quad = \ 5 \ * \ 2 \ + \ 4 \ / \ 2$$

Postfiksinę išraišką galima nesunkiai apskaičiuoti, naudojant steką. Bet kokios išraiškos skaičiavimo algoritmas labai paprastas:

1. Išskirti eilinį elementą iš išraiškos
2. Jei išskirtas elementas yra operandas, padėti jį į steką.
3. Jei išskirtas elementas yra operacija, ištraukti iš steko operandus, atlikti operaciją ir gautą rezultatą padėti į steką. (*Pastaba: pirmuoju iš steko išimamas antras operandas, pavyzdžiui, jei operacija yra "/" ir stekas (2, 4), tai bus skaičiuojama išraiška 4 / 2.*)
4. Jei išraiška baigta nagrinėti, rezultatas yra steke, priešingu atveju kartoti žingsnius 1-3.

Panagrinėkime, kaip veikia algoritmas, skaičiuodamas išraišką: 5 2 3 \* +

Pradžioje stekas tuščias: ().

Išskirtas elementas "5" - operandas, dedam jį į steką: stekas (5). Išraiška nebaigta nagrinėti.

Išskirtas elementas "2" - operandas, dedam jį į steką: stekas (2, 5). Išraiška nebaigta nagrinėti.

Išskirtas elementas "3" - operandas, dedam jį į steką: stekas (3, 2, 5). Išraiška nebaigta nagrinėti.

Išskirtas elementas "\*" - operacija, išimam iš steko operandus "3" ir "2", atliekam operaciją ir jos rezultatą "6" dedam jį į steką: stekas (6, 5). Išraiška nebaigta nagrinėti.

Išskirtas elementas "+" - operacija, išimam iš steko operandus "6" ir "5", atliekam operaciją ir jos rezultatą "11" dedam jį į steką: stekas (11). Išraiška baigta nagrinėti, taigi jos rezultatas yra steke - "11".

## Eilė

Eilė (*angl. queue*) yra sąrašas, kuriame elementai gali būti įterpiami tik į galą, o naikinami tik pradžioje. Taigi eilė (kaip ir stekas) yra specifinis sąrašas su apribotomis operacijomis. Tai FIFO (*First In First Out*) duomenų struktūra (kartais angliškai ji dar vadinama *FIFO stack*). Kiekvienas eilės elementas saugo tam tikrus duomenis.

### Operacijos:

- Sukurti tuščią eilę
- Patikrinti, ar eilė tuščia
- Patikrinti, ar eilė pilna
- Įdėti (*angl. enqueue*) naują elementą į eilę
- Išimti (*angl. dequeue*) elementą iš eilės
- Gauti pirmo eilės elemento duomenis, neišimant jo iš eilės
- Gauti eilės elementų skaičių
- Sunaikinti eilę

Nagrinėjant apibendrintą eilės realizaciją, pirmiausia pažymėkime, kad nėra prasmės jos vaizduoti kaip dvipusio sąrašo. Kadangi eilė yra atskiras sąrašo atvejis, jos realizacijai galėtų būti panaudoti visi bet kokio sąrašo realizavimo būdai aptarti anksčiau, tačiau naudoti du masyvus (arba masyvą papildytą nuorodomis) nėra prasmės, nes viduriniai eilės elementai negali būti „liečiami“. Trumpai aptarsime kitų dviejų būdų privalumus ir trūkumus eilės atveju.

**Realizacija:**

1. Masyvas, kurio elementai tokie patys kaip eilės elementai.

Privalumai:

- maksimaliai paprasta struktūra
- gana paprasta operacijų realizacija
- laikomi tik patys duomenys (nereikia papildomos atminties nuorodoms saugoti)

Trūkumai:

- masyvo (tuo pačiu ir eilės dydis turi būti nusakytas iš anksto, todėl gali būti arba naudojama tik nedidelė jo dalis, arba pritrūkti vietos.

Reikia pastebėti, kad elemento įterpimo/naikinimo operacijos eilės atveju gali būti realizuotos efektyviai: panaikinus pirmą eilės elementą visai nebūtina visus elementus perstumti, pakanka saugoti pirmo eilės elemento indeksą masyve.

2. Dinaminis sąrašas.

Privalumai:

- atmintis naudojama tik egzistuojantiems eilės elementams, t.y. nerezervuojama vieta potencialiems elementams.

Trūkumai:

- operacijų realizacija truputį sudėtingesnė (reikia mokėti dirbti su rodyklėmis);
- reikia daugiau atminties, nes saugomi ne tik duomenys, bet ir nuorodos (rodyklės).

Gyvenime daug kur dirbama pagal eilės principą, todėl ir daugelyje taikymų prasminga naudoti duomenų struktūrą - eilę.

**Dekas**

Dekas (*angl. deck or dequeue*) yra sąrašas, kuriame elementai gali būti įterpiami/naikinami tik pradžioje ir gale. Taigi dekas (kaip ir eilė bei stekas) yra specifinis sąrašas su apribotomis operacijomis. Dekas dar vadinamas abipusiu steku (arba abipuse eile), nes galima įsivaizduoti, kad dekas yra du slankiai sujungti stekai, kurių viršūnėse galima atlikti elementų įterpimo/naikinimo operacijas. Kiekvienas deko elementas saugo tam tikrus duomenis.

**Operacijos:**

- Sukurti tuščią deką
- Patikrinti, ar dekas tuščias
- Patikrinti, ar dekas pilnas
- Įterpti naujus duomenis (naują elementą) į deko pradžią
- Panaikinti elementą deko pradžioje
- Įterpti naujus duomenis (naują elementą) į deko pabaigą
- Panaikinti elementą deko pabaigoje
- Gauti deko pradžios elemento duomenis, neišimant jo iš deko
- Gauti deko pabaigos elemento duomenis, neišimant jo iš deko
- Gauti deko elementų skaičių
- Sunaikinti deką

Kadangi dekas yra atskiras sąrašo atvejis, jo realizacijai galėtų būti panaudoti visi bet kokio sąrašo realizavimo būdai aptarti anksčiau, tačiau naudoti du masyvus (arba masyvą papildytą

nuorodomis) nelabai prasminga, nes viduriniai deko elementai negali būti „liečiami“. Trumpai aptarsime kitų dviejų būdų privalumus ir trūkumus eilės atveju.

**Realizacija:**

1. Masyvas, kurio elementai tokie patys kaip deko elementai.

Privalumai:

- maksimaliai paprasta struktūra
- gana paprasta operacijų realizacija
- laikomi tik patys duomenys (nereikia papildomos atminties nuorodoms saugoti)

Trūkumai:

- masyvo (tuo pačiu ir deko dydis turi būti nusakytas iš anksto, todėl gali būti arba naudojama tik nedidelė jo dalis, arba pritrūkti vietos.

Reikia pastebėti, kad elemento įterpimo/naikinimo operacijos deko atveju gali būti realizuotos efektyviai: pakanka saugoti pirmo ir paskutinio deko elementų indeksus masyve.

2. Dinaminis sąrašas.

Privalumai:

- atmintis naudojama tik egzistuojantiems deko elementams, t.y. nerezervuojama vieta potencialiems elementams.

Trūkumai:

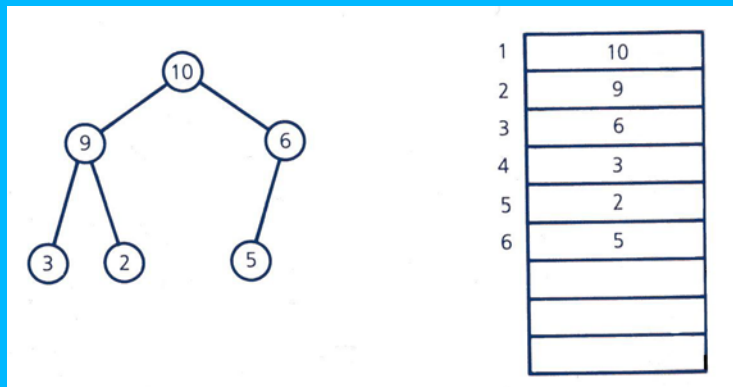
- operacijų realizacija truputį sudėtingesnė (reikia mokėti dirbti su rodyklėmis);
- reikia daugiau atminties, nes saugomi ne tik duomenys, bet ir nuorodos (rodyklės).

## Duomenų struktūra *heap*

*Heap* yra duomenų struktūra, panaši į dvejetainį paieškos medį, bet skiriasi nuo pastarojo dviem esminiais aspektais.

Visų pirma paieškos medžiuose duomenys yra surikiuoti, o *heap'e* – ne. Tačiau tai, kaip duomenys yra organizuoti *heap'e*, leidžia efektyviai realizuoti prioritetinės eilės operacijas, tokias kaip: *sukurti*; *patikrinti*, ar eilė yra tuščia; *įterpti* ir *išmesti*.

Kitas esminis skirtumas yra tas, kad dvejetainiai paieškos medžiai gali būti įvairūs (pvz., AVL medis, Raudonai-juodas medis). *Heap* visada yra užbaigtas (*complete*) dvejetainis medis, todėl jei jo maksimalus dydis yra žinomas iš anksto, tai realizacija masyvu yra labai efektyvi.



**Duomenų struktūra *heap*** - užbaigtas dvejetainis medis, kurio šaknies prioriteto reikšmė yra didesnė arba lygi kiekvieno jos vaiko prioriteto reikšmei ir kuriame abu šaknies pomedžiai yra duomenų struktūros *heap*.

Kitaip nei dvejetainiame paieškos medyje, nėra jokio sąryšio tarp vaikų reikšmių, t.y. nežinoma, kurio iš vaikų reikšmė didesnė.

## Prioritetinės eilės realizacija duomenų struktūra *heap*

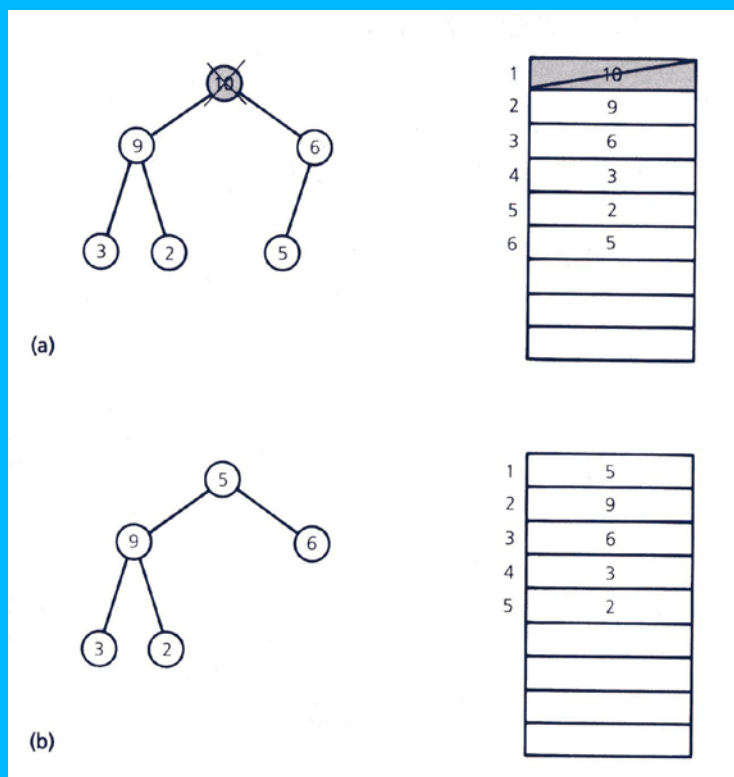
Štai kaip galime panaudoti duomenų struktūrą *heap* prioritetinės eilės operacijoms „*įterpti*“ ir „*išmesti*“ realizuoti. Tegu prioritetinė eilė PQ bus įrašas su dviem laukais:

- **Items:** prioritetinės eilės narių masyvas .
- **Last:** integer tipo laukas, kuris rodo, kiek narių yra prioritetinėje eilėje.

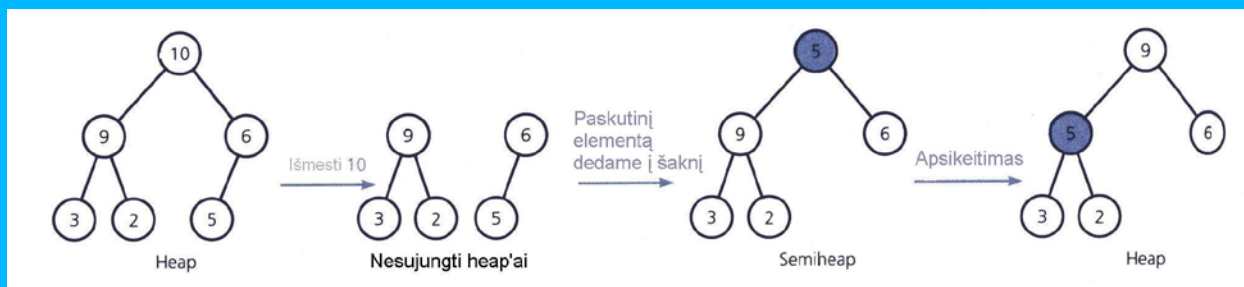
Masyvas *Items* yra užbaigto dvejetainio medžio realizacija masyvu.

## Išmetimas

Pradėsime nuo operacijos „Išmesti“. Kadangi vaikų reikšmės yra arba mažesnės, arba lygios šaknies reikšmei, todėl pati didžiausia reikšmė turi būti medžio šaknyje. Po didžiausios reikšmės išmetimo lieka dvi nesujungtos duomenų struktūros *heap*. Reikia pertvarkyti likusius mazgus taip, kad jie vėl sudarytų duomenų struktūrą *heap*. Pertvarkymas pradedamas nuo to, kad yra paaimama paskutinio mazgo reikšmė ir yra patalpinama į šaknį. Po šio žingsnio gausime užbaigtą medį. Vientintelė problema yra ta, kad šaknies reikšmė (dažniausiai) yra ne vietoje. Tokia struktūra vadinama *semi-heap*. Taigi reikalingas būdas, kuris leistų pertvarkyti ją į duomenų struktūrą *heap*. Vienas iš būdų yra „nuleidinėti“ medžio šaknies reikšmę tol, kol pastaroji neatsidurs savo vietoje, t. y. kol ji nepasieks lapo arba pirmo mazgo, kuriame ji bus didesnė už vaikų reikšmes arba lygi joms



Tai padaryti galima palyginę šaknies reikšmę su jos vaikų reikšmėmis. Jei šaknies reikšmė yra mažesnė už bent vieno iš vaikų reikšmę, tai sukeičiame šaknies reikšmę su didesne iš vaikų reikšmių.



Nors pavyzdyje reikšmė atsiduria savo vietoje jau po pirmo sukeitimo, bendru atveju gali prireikti ir daugiau sukeitimo operacijų. Kai šaknies ir didesnio vaiko C reikšmės yra sukeičiamos, C tampa semi-heap'o šaknimi. (Atkreipkite dėmesį į tai, kad C lieka savo vietoje, keičiasi tik jo reikšmė). Šią idėją galima realizuoti tokiu rekursiniu algoritmu.

*Adjust* (*H* : *heap*, *root* : *reikšmė*)

*begin*

{ Rekursiškai „nuleidžiama“ šaknis, kol heap nebus sutvarkytas.

Šaknis „nuleidžiama“, kai jos reikšmė yra mažesnė už kurio nors iš vaikų. }

*if* ( $2 * \text{root} > H.\text{Last}$ ) *<root'as yra lapas ir jokių veiksmų imtis nereikia>*

*if* ( $2 * \text{root} < H.\text{Last}$ ) *AND* ( $(H.\text{Items}[\text{root}] < H.\text{Items}[2 * \text{root}])$  *OR*  $H.\text{Items}[\text{root}] < H.\text{Items}[2 * \text{root} + 1]$ ))

*then*

*begin*

*<nustatyti kuris vaikas yra didesnis>*

*Swap*( $H.\text{Items}[\text{root}]$ ,  $H.\text{Items}[\text{child}]$ )

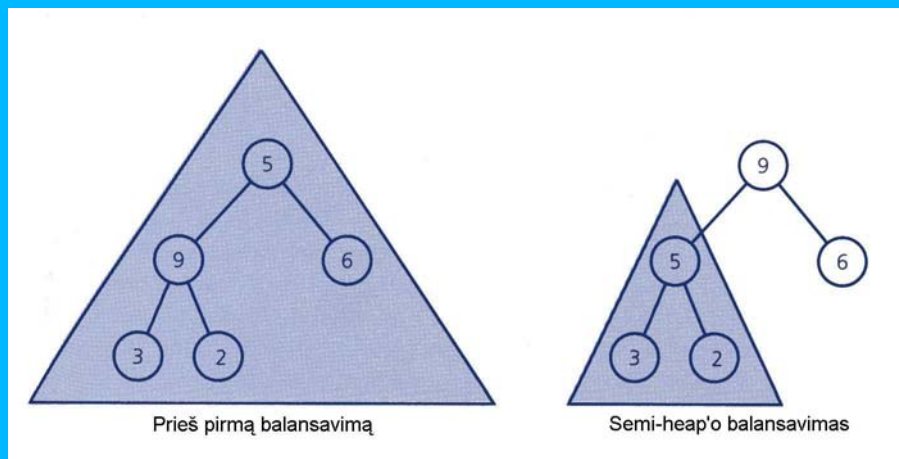
*Adjust*(*H*, *child*) {Rekursinis iškvietimas}

*end*

*else <viskas gerai, heap sutvarkytas.>*

*end*

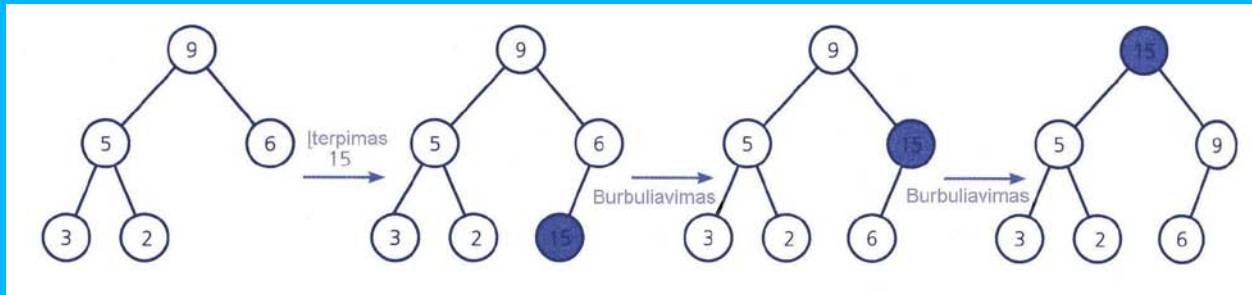
Prieš pradėdami nagrinėti funkcijos „Įterpti“ algoritmą, išstirkime funkcijos „Išmesti“ efektyvumą. Kadangi medis yra realizuotas masyvu, norėdami išmesti vieną mazgą, turime sukeisti vietomis masyvo elementus, o ne pakeisti kelias rodykles. Šitie sukeitimai gali pasirodyti neefektyvūs, tačiau tai nereiškia, kad algoritmas yra neefektyvus. Gali kilti klausimas, kiek daugiausiai reikės sukeisti masyvo elementų. Funkcija kopijuoja paskutinio mazgo reikšmę į šaknį. Funkcija „Derinti“ nuleidinėja reikšmę tol, kol ji nepasieks savo vietos.



Todėl funkcijai „Derinti“ reikės sukeisti ne daugiau nei medžio aukštis reikšmių. Kadangi pilno medžio su  $N$  mazgų aukštis visada yra apytiksliai  $\log_2 N$ , todėl funkcija „Išmesti“ yra pakankamai efektyvi.

## Įterpimas

Funkcijos „Įterpti“ veikimo principas yra priešingas funkcijos „Išmesti“ veikimo principui. Kai nauja reikšmė yra įterpiama, ji patalpinama į medžio apačią, o paskui „pakeliama“ iki savo vietos medyje. Pakelti reikšmę pakankamai lengva, kadangi mazgo  $i$  tėvas laikomas  $PQ.Items[i \div 2]$



Funkcijos „Įterpti“ efektyvumas yra panašus į funkcijos „Išmesti“ efektyvumą. Blogiausiu atveju funkcijai „Įterpti“ teks keisti elementus nuo lapo iki šaknies. Todėl sukeitimų skaičius negali viršyti medžio aukščio. Kadangi pilno medžio aukštis yra visada apytiksliai  $\log_2 N$ , todėl funkcija „Įterpti“ irgi yra pakankamai efektyvi.

Taigi duomenų struktūra *heap* yra efektyvi prioritetinės eilės realizacija.

Dabar palyginsime duomenų struktūros *heap* realizaciją masyvu ir dvejetainio paieškos medžio realizaciją su rodyklėmis. *Heap* realizacija efektyvi tik tada, kai yra žinomas didžiausias narių skaičius. Jei tas skaičius yra žinomas, tada masyvo realizacija turi keletą privalumų. Vienas jų yra tas, kad realizacija nenaudoja rodyklių, todėl reikia mažiau atminties. Tačiau daugeliu atvejų šis privalumas nėra esminis.

Tikras privalumas yra tas, kad *heap* yra visada subalansuotas. (Užbaigtumo sąlyga yra griežtesnė už balansavimo, t. y. jei medis užbaigtas, vadinasi ir subalansuotas, bet ne atvirkščiai). Kaip žinia, dvejetainio paieškos medžio aukštis gali gerokai viršyti  $\log_2 N$ . Operacijos, kurios palaiko medį subalansuotą, yra daug sudėtingesnės nei *heap* operacijos. Tačiau nemanysite, kad duomenų struktūra *heap* gali pakeisti dvejetainį paieškos medį kaip lentelės realizaciją. Šioje situacijoje duomenų struktūra *heap* netinka, kadangi ji neleidžia efektyviai patikrinti, ar yra ieškomas elementas *heap*'e.

Tarkime, turime baigtinį kiekį skirtingų prioritetų reikšmių, pvz., skaičiai nuo 1 iki 10. Tokiais atvejais tikėtina, kad daug reikšmių turės vienodas prioritetų reikšmes ir todėl tikėtina, kad juos reikės surikiuoti.

Eilių *heap* leidžia išspręsti šį uždavinį. Kiekviena prioriteto reikšmė gali būti tik viena. Norint įterpti reikšmę į prioritetinę eilę, reikia įdėti jos prioritetinę reikšmę į duomenų struktūrą *heap* (jei jos ten dar nėra) ir po to įdėti į atitinkamą eilę. Norint išmesti reikšmę, reikia išmesti pirmąją reikšmę iš eilės, kurios prioritetas didžiausias. Jei po šio išmetimo eilė tampa tuščia, tai jos prioriteto reikšmę reikia išmesti iš duomenų struktūros *heap*.

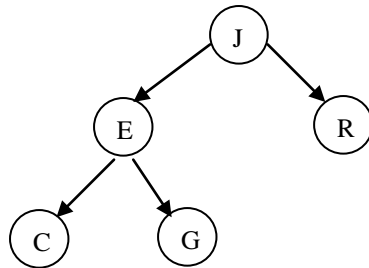
### Dvejetainis paieškos medis

Visos iki šiol nagrinėtos duomenų struktūros buvo tiesinės, vienmatės (po kiekvieno elemento galėjo būti tik vienas kitas elementas). Tokias duomenų struktūras gana paprasta programuoti ir naudoti, tačiau kai kuriems taikymams jos gali būti neefektyvios, todėl reikia nagrinėti sudėtingesnes, pavyzdžiui, dvimates duomenų struktūras. Viena iš tokių duomenų struktūrų yra medis.

Medis yra tokia hierarchinė duomenų struktūra, kurioje į kiekvieną elementą (viršūnę), išskyrus vieną, vadinamą medžio šaknimi, yra tik vienintelė nuoroda iš kito elemento (viršūnės), iš kiekvieno elemento (viršūnės) gali būti viena arba daugiau nuorodų į kitus elementus (viršūnes) ir visas kitas viršūnes galima pasiekti iš šaknies, einant nuorodomis.

Panagrinėjus šį apibrėžimą, matome, kad tiesinis sąrašas irgi galėtų būti laikomas atskiru medžio atveju.

Kaip jau buvo minėta, viena medžio viršūnė (į kurią nėra nuorodų) vadinama medžio šaknimi (*angl. root*). Viršūnė, iš kurios nėra nuorodų į kitas viršūnes, vadinama lapu. Kiekviena nuoroda su viršūnėmis, į kurias galima patekti, pradėjus eiti ta nuoroda, vadinama medžio šaka arba pomedžiu (nes ši struktūra be pradinės nuorodos savo ruožtu yra medis).



Pastebėkime, kad tradiciškai duomenų struktūra medis piešiama iš viršaus žemyn (skirtingai nei auga medžiai gamtoje). Šiame pavyzdyje: šaknis yra viršūnė "J", lapai viršūnės "C", "G" ir "R", iš viršūnių "J" ir "E" išeina po dvi šakos (pomedžius). Galbūt ryšium su genealoginiais medžiais, bet kalbant apie medžius dažnai vartojama tėvų-vaikų terminologija, pavyzdžiui, viršūnė "C" yra viršūnės "E" vaikas (arba dukterinė viršūnė), viršūnė "J" yra viršūnės "E" tėvas. Viršūnė "C" yra ir viršūnės "J" palikuonis, bet tarp jų dar yra viršūnė "E", todėl tikslumo dėlei kartais sakoma, kad "C" yra viršūnės "E" betarpiškas vaikas, o "E" yra viršūnės "C" betarpiškas tėvas. Be to, kalbant apie medžius įvedama lygio sąvoka. Viršūnės lygis yra vienetu didesnis nei reikia žingsniu tam, kad nuo šaknies ateiti iki viršūnės. Pateiktame pavyzdyje 1-ame lygyje yra viena viršūnė "J", 2-ame lygyje - 2 viršūnės "E" ir "R", o 3-iame lygyje - irgi 2 viršūnės "C" ir "G". Medžio aukštis yra skirtingų lygių skaičius (pavyzdžio medžio aukštis yra 3).

Bendru atveju iš kiekvienos viršūnės gali išeiti bet koks šakų skaičius, tačiau praktiniuose taikymuose dažniausiai uždedami papildomi apribojimai šakų skaičiui, pavyzdžiui, ne daugiau kaip dvi. Apribojimai šakų skaičių supaprastina duomenų struktūros realizaciją. Tokioms duomenų struktūroms suteikiami specifiniai vardai:

Dvejetainis medis (*angl. binary tree*) yra medis, iš kurio viršūnės gali išeiti ne daugiau kaip dvi šakos.

Toliau nagrinėsime tik dvejetainius medžius. Kadangi iš dvejetainio medžio viršūnės gali išeiti dvi šakos, todėl apibrėžtumo dėlei viena iš jų vadinama kairiąja šaka (kairiuoju pomedžiu), o kita dešiniąja šaka (dešiniuoju pomedžiu). Pateiktame pavyzdyje viršūnės "J" kairiajame pomedyje yra 3 viršūnės ("E", "C", "G"), o dešiniajame - viena viršūnė ("R").



Dvejetainis medis vadinamas subalansuotu, jei kiekvienos jo viršūnės dešiniajame ir kairiajame pomedžiuose yra vienodai (beveik vienodai) viršūnių.

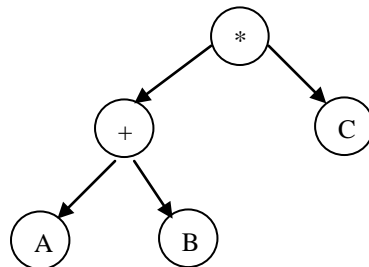
Kiekvienoje medžio viršūnėje saugomi duomenys. Norint visus juos surinkti, reikia apvaikščioti visas medžio viršūnes. Šis procesas vadinamas medžio apėjimu. Dvejetainio medžio apėjimo tvarkos klasifikuojamos pagal tai, kada aplankoma pati viršūnė (nuskaitomi jos duomenys). Šiuo požiūriu galimos trys apėjimo tvarkos:

1. VKD: aplankyti Viršūnę; apeiti Kairinį pomedį; apeiti Dešinįjį pomedį.
2. KVD: apeiti Kairinį pomedį; aplankyti Viršūnę; apeiti Dešinįjį pomedį.
3. KDV: apeiti Kairinį pomedį; apeiti Dešinįjį pomedį; aplankyti Viršūnę.

Pateiktame pavyzdyje, apeinant medį kiekviena tvarka, gautume tokius rezultatus:

- VKD: JECGR
- KVD: CEGJR
- KDV: CGERJ

Duomenų išdėstymas viršūnėse ir medžio apėjimo tvarka parenkama priklausomai nuo taikymo. Kai kuriais atvejais galima naudoti skirtingas medžio apėjimo strategijas tam, kad gauti skirtingus rezultatus. Panagrinėkime pavyzdį, kai dvejetainiame medyje saugoma išraiška:



Apeidami medį pagal VKD strategiją, gausime prefiksinę išraišką  $* + A B C$ .

Apeidami medį pagal KVD strategiją, gausime infiksinę išraišką  $A + B * C$ , bet tam, kad veiksmas būtų atliekamas reikiama tvarka, šią išraišką reikia papildyti skliaustais.

Apeidami medį pagal KDV strategiją, gausime postfiksinę išraišką  $A B + C *$ .

Iki šiol nagrinėjome dvejetainius medžius, kuriuose duomenys talpinami į viršūnes bet kokia tvarka. Gali būti tikslinga apibrėžti konkrečią duomenų išdėstymo viršūnėse tvarką.

Dvejetainis paieškos medis (*angl. binary search tree*) yra dvejetainis medis, kurio kiekvienai viršūnei galioja taisyklė, kad visos reikšmės esančios kairiajame pomedyje yra mažesnės už toje viršūnėje saugomą reikšmę, o visos reikšmės esančios dešiniajame pomedyje yra didesnės už toje viršūnėje saugomą reikšmę (žinoma, galima fiksuoti ir atvirkštinį surūšiavimą: kairėje daugiau, dešinėje mažiau).

Šis medis vadinamas paieškos medžiu, nes jame galima efektyviai ieškoti informacijos. Tam, kad surasti reikiamus duomenis arba įsitikinti, kad jų medyje nėra, reikės atlikti nedaugiau žingsnių nei medžio aukštis (pavyzdyje pateiktam medžiui nedaugiau kaip 3 žingsnius). Paieškos algoritmas labai paprastas:

1. Patikrinti viršūnę
2. Jei ieškoma reikšmė lygi viršūnės reikšmei, baigti darbą "reikšmė rasta".
3. Jei ieškoma reikšmė mažesnė už viršūnės reikšmę, pereiti į kairinį pomedį ir kartoti nuo 1-o žingsnio (jei jis tuščias, baigti darbą "reikšmė nerasta").
4. Jei ieškoma reikšmė didesnė už viršūnės reikšmę, pereiti į dešinįjį pomedį ir kartoti nuo 1-o žingsnio (jei jis tuščias, baigti darbą "reikšmė nerasta").

Jei medis subalansuotas, tai po kiekvieno žingsnio viršūnių - potencialių kandidatų sumažėja dvigubai, taigi jei turime  $N$  duomenų (viršūnių), daugiausia prireiks  $\log_2 N$  žingsnių.

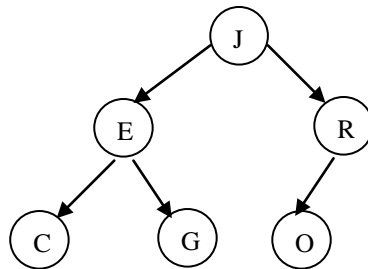
Tai žymiai mažiau nei reiktų atlikti žingsnių, norint patikrinti, ar reikiami duomenys yra tiesiniame sąraše. Žinoma, jei medis nesubalansuotas, gali prireikti gerokai daugiau žingsnių nei  $\log_2 N$ .

Norint atspausdinti dvejetainio paieškos medžio duomenis didėjimo tvarka, reikia naudoti KVD medžio apėjimą.

Panagrinėkime, kaip reiktų įterpti naujus duomenis į dvejetainį paieškos medį. Tai galima padaryti, naudojantis tokiu algoritmu:

1. Patikrinti viršūnę.
2. Jei įterpiama reikšmė lygi viršūnės reikšmei, baigti darbą "tokia reikšmė jau yra".
3. Jei įterpiama reikšmė mažesnė už viršūnės reikšmę,  
jei kairysis pomedis netuščias, pereiti į jį ir kartoti nuo 1-o žingsnio  
jei kairysis pomedis tuščias, sukurti jame naują viršūnę su įterpiamais duomenimis
4. Jei įterpiama didesnė už viršūnės reikšmę,  
jei dešinysis pomedis netuščias, pereiti į jį ir kartoti nuo 1-o žingsnio  
jei dešinysis pomedis tuščias, sukurti jame naują viršūnę su įterpiamais duomenimis

Pritaikykite šį algoritmą pavyzdžio medžiui ir įterpiamai reikšmei "O". Tikriname šaknį: įterpiama reikšmė "O" didesnė už jos reikšmę "J". Dešinysis pomedis netuščias, todėl pereiname į jį (einamoji viršūnė "R") ir kartojame. Tikriname einamąją viršūnę: įterpiama reikšmė "O" mažesnė už jos reikšmę "R". Kairysis pomedis tuščias, todėl sukuriame jame viršūnę "O" ir rezultate gauname tokį medį:

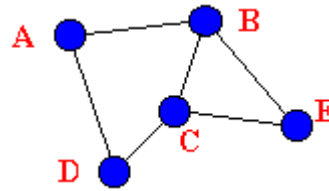


Darbą su dvejetainiais medžiais lengviausia suprogramuoti rekursiškai, bet galima apsieiti ir be rekursijos.

## Grafai

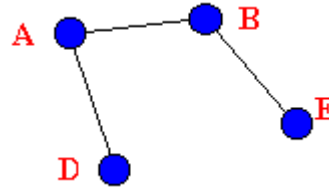
**Grafas** – aibių pora  $(V, L)$  [ $V$  – viršūnių aibė,  $L$  – briaunų aibė]

**Briauna** – atkarpa, jungianti dvi grafo viršūnes.



1 pav. Grafas

**Pografinis** (*subgraph*) – poaibis grafo briaunų (*edge*) bei jų viršūnių (*vertex*).



2 pav. Pografis

**Dvi viršūnės yra gretimos** (*adjacent*), jei jos sujungtos briauna. Viršūnės  $V_i$  ir  $V_j$  yra kaimyninės, jeigu egzistuoja  $B_k=(V_i, V_j)$ . Pvz.: **A** kaimyninės viršūnės yra **B** ir **D**.

**Plokščias grafas** – grafas, kuri galima nupiešti plokštumoje (bent vienu būdu) taip, kad nė viena pora briaunų nesikirstų.

**Kelias** (*path*) tarp viršūnių – briaunų seka, prasidedanti vienoje viršūnėje ir besibaigianti kitoje viršūnėje.

**Paprastas kelias** (*simple path*) – kelias, per kiekvieną jam priklausančią viršūnę einantis tik po vieną kartą. Pvz.: Kelias – **ADCBE** nėra paprastas kelias, nes per viršūnę C eina du kartus.

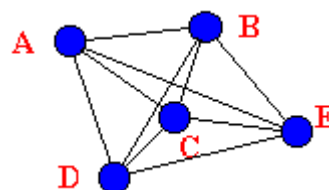
**Ciklas** (*cycle*) – paprastas kelias, kuris prasideda ir baigiasi toje pačioje viršūnėje. Pvz.: **ABCD A**.

**Jungus grafas** (*connected*) – jei egzistuoja kelias tarp bet kurių viršūnių porų.

$\forall V_i, V_j: \exists \text{kelias } V_i \rightarrow V_j$ . (pvz. 1 pav.)

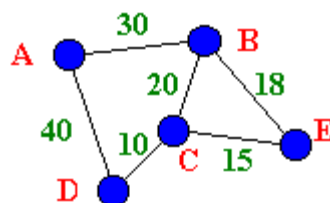
**Pilnas grafas** (*complete*) – jei yra briauna tarp kiekvienos viršūnių poros. Aišku, kad pilnas grafas taip pat yra ir jungus, tačiau jungus grafas nebūtinai yra pilnas.

$\forall V_i, V_j: \exists B = (V_i, V_j)$



3 pav. Pilnas grafas

**Grafas su svoriais** (*weighted*) – grafas, kurio bet kokia briauna turi skaitinę reikšmę.



4 pav. Grafas su svoriais

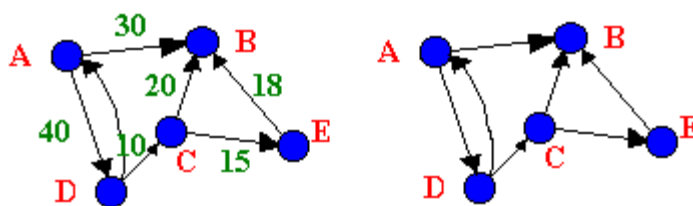
Iki šiol buvo kalbama apie grafus, kurių briaunos neturi krypties, t.y. briauna galima keliauti bet kuria kryptimi. Tai reiškia, kad tarp dviejų viršūnių gali būti tik viena briauna, jei grafas yra be svorių.

**Orientuotas** (kryptinis) **grafas** (*directed*) – grafas su lankais (visos briaunos turi kryptį).

**Lankas** – orientuoto grafo briauna, turinti kryptį.

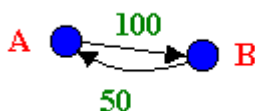
Orientuotame grafe gali būti ne daugiau dviejų briaunų, jungiančių dvi viršūnes, jei grafas yra be svorių. Visi apibūdinimai, kurie buvo taikomi neorientuotiems grafams, taip pat tinka ir orientuotam grafui.

Pvz.: Orientuotas kelias yra seka kryptingų briaunų tarp dviejų viršūnių. **A** kaimynas yra **B**, bet **B** – nėra **A** kaimynas.



5 pav. Orientuoti grafai

Pvz.: Knygų skolinimasis: **A** iš **B** pasiskolino 100 knygų, o **B** iš **A** pasiskolino 50 knygų.



6 pav.

### Grafai kaip abstraktūs duomenų tipai (ADT)

Grafas gali būti traktuojamas kaip abstraktus duomenų tipas. Įterpimo ir ištrynimo operacijos šiek tiek skiriasi nuo kitų ADT. ADT grafus galima apibrėžti tiek su reikšmėmis, tiek ir be jų. Pagrindinės operacijos su grafais kaip ADT:

- Sukurti tuščią grafą.
- Įdėti/išmesti viršūnę.
- Įdėti/išmesti briauną tarp viršūnių  $V_1$  ir  $V_2$ .
- Sužinoti (rasti), ar yra kelias tarp viršūnių  $V_1$  ir  $V_2$ .
- Sužinoti ( $V_1$ ,  $V_2$ ) svorį.
- Pakeisti ( $V_1$ ,  $V_2$ ) svorį.

## Grafų realizavimas

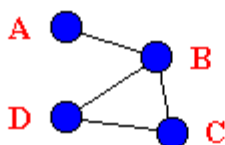
Yra du svarbiausi grafų realizavimo būdai:

- kaimynystės matrica
- kaimynystės sąrašai

Abiem atvejais patogiausia įsivaizduoti, kad viršūnės numeruojamos 1, 2 ir taip toliau iki N.

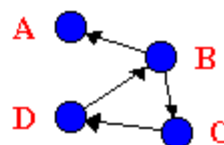
**Kaimynystės matrica** grafiui be svorių su N viršūnių yra N iš N loginis masyvas A toks, kad  $A[i, j]$  yra teisingas tada ir tik tada, kai egzistuoja briauna iš viršūnės 'i' į viršūnę 'j'. Pagal susitarimą  $A[i, i]$  yra klaidingas. Įsidėmėtina, kad kaimynystės matrica neorientuotam grafiui yra simetriška, tai yra  $A[i, j] = A[j, i]$ .

	A	B	C	D
A	F	T	F	F
B	T	F	T	T
C	F	T	F	T
D	F	T	T	F



Neorientuotas grafas  
(simetriška matrica)

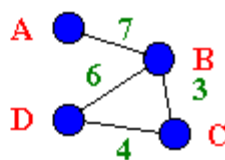
	A	B	C	D
A	F	F	F	F
B	T	F	T	F
C	F	F	F	T
D	F	T	F	F



Orientuotas grafas  
(dažniausiai nesimetriška matrica)

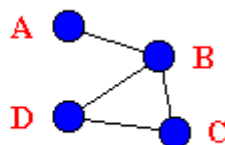
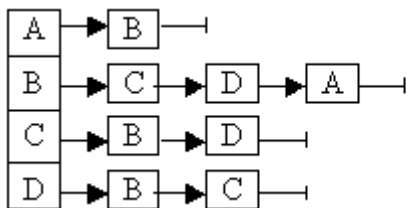
Kai turim grafą su svoriais, yra patogu, kad  $A[i, j]$  būtų briaunos iš viršūnės 'i' į viršūnę 'j' svoris. Tada  $A[i, j]$  žymėtų, kai nėra briaunos iš viršūnės 'i' į viršūnę 'j'. Be to, įstrižainės  $A[i, i]$  reikšmės lygios 0.

	A	B	C	D
A	0	7	$\infty$	$\infty$
B	7	0	3	6
C	$\infty$	3	0	4
D	$\infty$	6	4	0



8 pav. Grafo su svoriais matrica

**Kaimynystės sąrašas** grafo iš N viršūnių, kurios numeruojamos 1, 2, ..., N, susideda iš N sujungtų sąrašų.



9 pav. Grafo kaimynystės sąrašas

Jei grafas yra su svoriais, tai jie saugomi kartu su viršūnę. Pvz.: 

A	→	B	7
---	---	---	---

Dvi dažniausiai naudojamos grafų operacijos yra:

1. Duotos dvi viršūnės 'i' ir 'j'; rasti, ar yra briauna iš 'i' į 'j'.
2. Rasti visas viršūnes, kurios yra kaimynės duotajai viršūnei  $V_i$

Jei grafas yra netoli pilno grafo, tai masyvas yra efektyvesnis už sąrašą. Jei briaunų mažai, tai lieka daug nepanaudotos vietos matricoje, kas yra minusas taupant, tuomet geriau sąrašas.

## Grafo apėjimas (traversal)

Egzistuoja du apėjimo algoritmai: į gylį (DFS – *Depth First Search*) ir į plotį (BFS – *Breadth First Search*).

**DFS (paieška į gylį).** Iš duotos viršūnės 'v' einama gilyn ir gilyn, kol pasiekama viršūnė, iš kurios giliau eiti jau nebeįmanoma. Tai yra, aplankius viršūnę 'v', DFS algoritmas aplanko dar neaplankytą kaimyninę viršūnę. Jis tęsiasi iš 'u' tiek toli, kiek galima iki grįžimo į 'v', kad aplankytų kitą neaplankytą viršūnę kaimyninę 'v'. Galima ir rekursyvi DFS realizacija.

### DFS pseudo – kodas:

```
{ S – stekas (stack) }
Push(S, v)
MarkV(v) {pažymi aplankytą viršūnę}

While (not IsEmpty(S)) do
  If (neegzistuoja neaplankyta viršūnė kaimynė steko viršuje esančiai viršūnei) then Pop(S)
  else
    begin
      imame 'u' – S viršūnės kaimynė ('u' ∈ V)
      Push(S, u)
      MarkV(u)
    end
  end
end
```

Grafo apėjimo DFS būdu pavyzdys:

	<p>Grafas bus pradedamas apeidinėti nuo viršūnės 'V'.</p> <p>Į steką įdedame viršūnę ,nuo kurios pradėsime apėjimą į gylį:</p> <p>Viršūnė: 'V'</p> <p>Stekas: 'V'</p>																
	<p>Maksimalus ėjimas į gylį nuo viršūnės V:</p> <table border="1"> <thead> <tr> <th>Viršūnės</th> <th>Stekas</th> </tr> </thead> <tbody> <tr><td>1</td><td>V 1</td></tr> <tr><td>2</td><td>V 1 2</td></tr> <tr><td>3</td><td>V 1 2 3</td></tr> <tr><td>4</td><td>V 1 2 3 4</td></tr> <tr><td>5</td><td>V 1 2 3 4 5</td></tr> <tr><td>6</td><td>V 1 2 3 4 5 6</td></tr> <tr><td>7</td><td>V 1 2 3 4 5 6 7</td></tr> </tbody> </table> <p>Dabar reikia grįžti atgal, kol vėl atsiras neaplankyta viršūnė.</p>	Viršūnės	Stekas	1	V 1	2	V 1 2	3	V 1 2 3	4	V 1 2 3 4	5	V 1 2 3 4 5	6	V 1 2 3 4 5 6	7	V 1 2 3 4 5 6 7
Viršūnės	Stekas																
1	V 1																
2	V 1 2																
3	V 1 2 3																
4	V 1 2 3 4																
5	V 1 2 3 4 5																
6	V 1 2 3 4 5 6																
7	V 1 2 3 4 5 6 7																
	<table border="1"> <thead> <tr> <th>Viršūnės</th> <th>Stekas</th> </tr> </thead> <tbody> <tr><td>Atgal</td><td>V 1 2 3 4 5 6</td></tr> <tr><td>8</td><td>V 1 2 3 4 5 6 8</td></tr> </tbody> </table> <p>Vėl reikia grįžti atgal, kol atsiras neaplankyta viršūnė.</p>	Viršūnės	Stekas	Atgal	V 1 2 3 4 5 6	8	V 1 2 3 4 5 6 8										
Viršūnės	Stekas																
Atgal	V 1 2 3 4 5 6																
8	V 1 2 3 4 5 6 8																

	<table> <tr><td>Viršūnė</td><td>Stekas</td></tr> <tr><td>Atgal</td><td>V 1 2 3 4 5 6</td></tr> <tr><td>Atgal</td><td>V 1 2 3 4 5</td></tr> <tr><td>Atgal</td><td>V 1 2 3 4</td></tr> <tr><td>Atgal</td><td>V 1 2 3</td></tr> <tr><td>9</td><td>V 1 2 3 9</td></tr> <tr><td>10</td><td>V 1 2 3 9 10</td></tr> <tr><td>11</td><td>V 1 2 3 9 10 11</td></tr> </table> <p>Vėl reikia grįžti atgal, kol atsiras neaplankyta viršūnė.</p>	Viršūnė	Stekas	Atgal	V 1 2 3 4 5 6	Atgal	V 1 2 3 4 5	Atgal	V 1 2 3 4	Atgal	V 1 2 3	9	V 1 2 3 9	10	V 1 2 3 9 10	11	V 1 2 3 9 10 11
Viršūnė	Stekas																
Atgal	V 1 2 3 4 5 6																
Atgal	V 1 2 3 4 5																
Atgal	V 1 2 3 4																
Atgal	V 1 2 3																
9	V 1 2 3 9																
10	V 1 2 3 9 10																
11	V 1 2 3 9 10 11																
	<table> <tr><td>Viršūnė</td><td>Stekas</td></tr> <tr><td>Atgal</td><td>V 1 2 3 9 10</td></tr> <tr><td>12</td><td>V 1 2 3 9 10 12</td></tr> </table> <p>Vėl reikia grįžti atgal, kol atsiras neaplankyta viršūnė, o jei neatsiras, vadinasi grafas jau apeitas.</p>	Viršūnė	Stekas	Atgal	V 1 2 3 9 10	12	V 1 2 3 9 10 12										
Viršūnė	Stekas																
Atgal	V 1 2 3 9 10																
12	V 1 2 3 9 10 12																
	<table> <tr><td>Viršūnė</td><td>Stekas</td></tr> <tr><td>Atgal</td><td>V 1 2 3 9 10</td></tr> <tr><td>Atgal</td><td>V 1 2 3 9</td></tr> <tr><td>Atgal</td><td>V 1 2 3</td></tr> <tr><td>Atgal</td><td>V 1 2</td></tr> <tr><td>Atgal</td><td>V 1</td></tr> <tr><td>Atgal</td><td>V</td></tr> </table> <p>Grįžome į pradinę viršūnę ir iš jos jau nėra briaunų į neaplankytas viršūnes, vadinasi grafas jau apeitas.</p>	Viršūnė	Stekas	Atgal	V 1 2 3 9 10	Atgal	V 1 2 3 9	Atgal	V 1 2 3	Atgal	V 1 2	Atgal	V 1	Atgal	V		
Viršūnė	Stekas																
Atgal	V 1 2 3 9 10																
Atgal	V 1 2 3 9																
Atgal	V 1 2 3																
Atgal	V 1 2																
Atgal	V 1																
Atgal	V																

DFS paieškos algoritmas tiksliai nenusako tvarkos, kuria turi būti aplankomos kaimyninės viršūnės. Viena galimybė yra aplankyti viršūnes, kaimynines 'v', abėcėlės arba numerių didėjimo tvarka.

**BFS (paieška į plotį).** Aplankius viršūnę v, BFS aplanko visas viršūnes kaimynines 'v'. Tokiu būdu apėjimas neprasisdės iš jokios kitos viršūnės kaimyninės 'v', kol nėra aplankyti visi galimi kaimyninės viršūnės. Pastebėkime, kad BFS atveju nėra rekursyvios realizacijos.

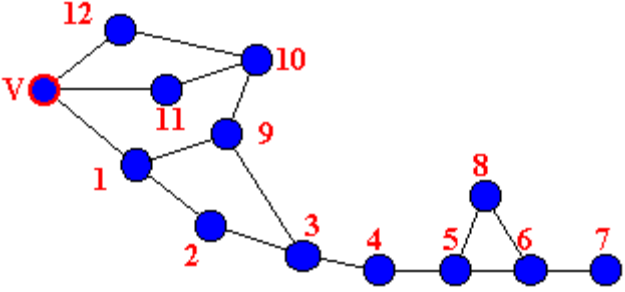
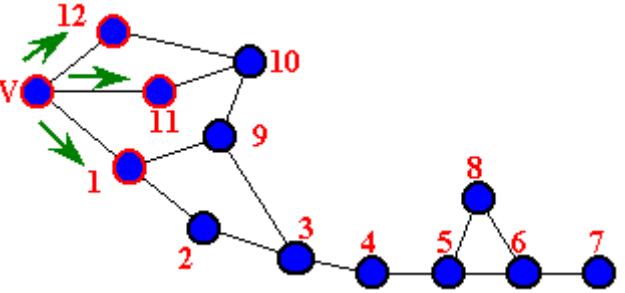
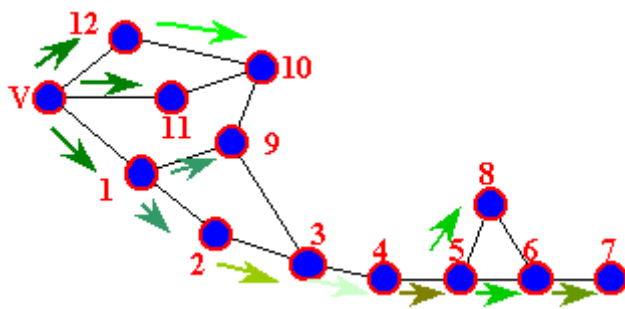
### BFS pseudo – kodas:

```

{Q – eilė}
Add(Q, V) {įdedame viršūnę V į eilę}
MarkV(V) {pažymi aplankytą viršūnę}
While (not IsEmpty(Q)) do
begin
    w := Get(Q) (Nuskaitomas ir išmetamas pirmas eilės elementas)
    for ∀ w neaplankytai kaimynei u do
    begin
        MarkV(u)
        Add(Q, u)
    End
end

```

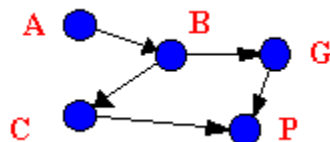
### Grafo apėjimo BFS būdu pavyzdys:

	<table> <tr> <th>Viršūnė V</th><th>Eilė V</th></tr> </table>	Viršūnė V	Eilė V																																										
Viršūnė V	Eilė V																																												
	<table> <tr> <th>Viršūnė</th><th>Eilė</th></tr> <tr> <td>-</td><td>-</td></tr> <tr> <td>1</td><td>1</td></tr> <tr> <td>11</td><td>1 11</td></tr> <tr> <td>12</td><td>1 11 12</td></tr> </table> <p>Aplankėme visas V kaimynines viršūnes, dabar iš eilės išimsime viršūnes ir dėsime jų kaimynes iki tol, kol eilė bus tuščia ir nebeliks kaimyninių viršūnių.</p>	Viršūnė	Eilė	-	-	1	1	11	1 11	12	1 11 12																																		
Viršūnė	Eilė																																												
-	-																																												
1	1																																												
11	1 11																																												
12	1 11 12																																												
	<table> <tr> <th>Viršūnė</th><th>Eilė</th></tr> <tr> <td>-</td><td>11 12</td></tr> <tr> <td>2</td><td>11 12 2</td></tr> <tr> <td>9</td><td>11 12 2 9</td></tr> <tr> <td>10</td><td>12 2 9</td></tr> <tr> <td></td><td>12 2 9 10</td></tr> <tr> <td></td><td>2 9 10</td></tr> <tr> <td></td><td>9 10</td></tr> <tr> <td>3</td><td>9 10 3</td></tr> <tr> <td></td><td>10 3</td></tr> <tr> <td></td><td>3</td></tr> <tr> <td></td><td>-</td></tr> <tr> <td>4</td><td>4</td></tr> <tr> <td></td><td>-</td></tr> <tr> <td>5</td><td>5</td></tr> <tr> <td></td><td>-</td></tr> <tr> <td>6</td><td>6</td></tr> <tr> <td>8</td><td>6 8</td></tr> <tr> <td></td><td>8</td></tr> <tr> <td>7</td><td>8 7</td></tr> <tr> <td></td><td>7</td></tr> <tr> <td></td><td>-</td></tr> </table> <p>Kaimyninių viršūnių nebeliko ir eilė tuščia, vadinasi grafas yra visas apeitas.</p>	Viršūnė	Eilė	-	11 12	2	11 12 2	9	11 12 2 9	10	12 2 9		12 2 9 10		2 9 10		9 10	3	9 10 3		10 3		3		-	4	4		-	5	5		-	6	6	8	6 8		8	7	8 7		7		-
Viršūnė	Eilė																																												
-	11 12																																												
2	11 12 2																																												
9	11 12 2 9																																												
10	12 2 9																																												
	12 2 9 10																																												
	2 9 10																																												
	9 10																																												
3	9 10 3																																												
	10 3																																												
	3																																												
	-																																												
4	4																																												
	-																																												
5	5																																												
	-																																												
6	6																																												
8	6 8																																												
	8																																												
7	8 7																																												
	7																																												
	-																																												



## Topologinis rikiavimas (*Topological Sorting*)

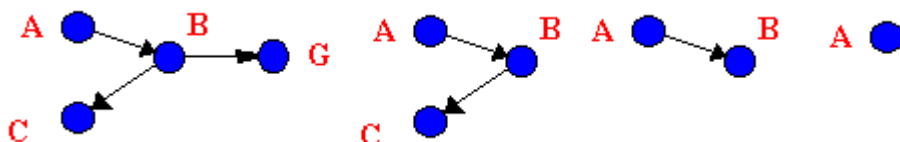
Taikomas orientuotam grafui be ciklų. Pvz.: briaunos išreiškia, koks dalykas prieš kokį dalyką turi būti išklaustyas. Prieš dalyką **G** reikia išklausti **B**. Du variantai: **ABCGP** **ABGCP**.



10 pav. Kryptinis be ciklų grafas

Paprastas algoritmas (*pa*):

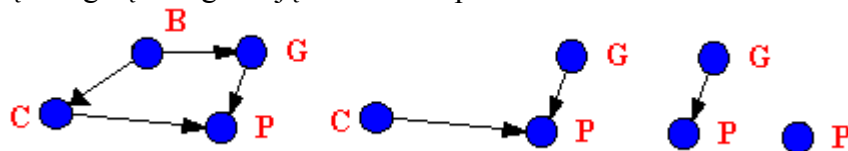
1. Imti viršūnę iš kurios nėra išeinančių briaunų.
2. Įdėti ją į sąrašo pradžią ir iš grafo ją išmesti.



11 pav. Topologinis rikiavimas (*pa*)

Modifikuotas paprastas algoritmas (*pam*):

1. Imti viršūnę į kurią nėra įeinančių briaunų.
2. Įdėti ją į sąrašo galą ir iš grafo ją išmesti. 12 pav.



12 pav. Topologinis rūšiavimas (*pam*)

Šis būdas nėra labai efektyvus, jei bus naudojami kaimynystės sąrašai.

## Aprėpties medžiai (*Spanning Trees*)

**Medis** – jungus neorientuotas grafas be ciklų.

Grafo  $G$  **aprėpties medis** – grafo  $G$  pografis  $G'$ , turintis visas grafo  $G$  viršūnes ir pakankamai briaunų medžiui. (Grafiui gali egzistuoti *keli* aprėpties medžiai.)

Jei jungiame neorientuotame grafe *išnaikinsime ciklus*, gausime aprėpties medį.

Neorientuoto grafo *savybės*:

1. **Jungus neorientuotas grafas su  $N$  viršūnių turi ne mažiau kaip  $N-1$  briauną.**

Reikia prisiminti, kad jungiame grafe egzistuoja kelias tarp bet kurių dviejų viršūnių. Tarkime, pasirenkame viršūnę ir sujungiame ją briauna su kita viršūne: 2 viršūnės ir 1 briauna; jei pridėdama dar vieną viršūnę, būtina ir papildoma briauna jai prijungti prie bet kurios kitos grafo viršūnės.

2. **Jungus neorientuotas grafas su  $N$  viršūnių ir  $N-1$  briauna negali turėti ciklų.**

Pagal ankstesnę savybę orientuotas grafas su  $N$  viršūnių privalo turėti ne mažiau kaip  $N-1$  briauną, kad būtų jungus. Jei jungus grafas turi ciklą, galima pašalinti bet kurią briauną iš ciklo ir grafas liks jungus. Taigi, jei grafas su  $N$  viršūnių ir  $N-1$  briauna turės ciklą, galima bus pašalinti vieną briauną ir gautas grafas su  $N-2$  briauna turi likti jungus, kas prieštarauja ankstesnei savybei.

3. **Jungus neorientuotas grafas su  $N$  viršūnių ir daugiau kaip  $N-1$  briauna turi bent vieną ciklą.**

Paimkime jungų neorientuotą grafą su  $N$  viršūnių ir  $N-1$  briauna. Pasirinkime bet kokią porą viršūnių  $X$  ir  $Y$  ir pridėkime naują briauną  $B$ , jungiančią viršūnes  $X$  ir  $Y$ . Kadangi pradinis grafas buvo jungus, tai jame egzistavo kelias iš viršūnės  $X$  į viršūnę  $Y$ . Pridėjus prie šio kelio naują briauną  $B$ , gaunamas ciklas, t.y. kelią iš viršūnės  $X$  į ją pačią.

Taigi patikrinti, ar jungus neorientuotas grafas turi ciklą, galima paprasčiausiai suskaičiuojant jo viršūnes ir briaunas. Iš to išplaukia, kad grafo  $G$  su  $N$  viršūnių aprėpties medis turi  $N-1$  briauną.

Du aprėpties medžio konstravimo algoritmai, remiasi anksčiau nagrinėtomis medžio apėjimo strategijomis: paieškos į gylį (DFS) ir paieškos į plotį (BFS). Bendru atveju šie algoritmai sukonstruos skirtingus aprėpties medžius, kuriuos toliau sąlyginai vadinsime **DFS** ir **BFS aprėpties medžiais** atitinkamai.

### DFS aprėpties medis

Vienas iš būdų sukonstruoti aprėpties medį jungiam neorientuotam grafiui yra apeiti grafo viršūnes, naudojant paieškos į gylį algoritmą. Apėjimo metu žymimos briaunos, kuriomis einama. Baigus apėjimą, pažymėtos briaunos sudarys **paieškos į gylį aprėpties medį** (*DFS spanning tree*). Tereikia nežymiai modifikuoti paieškos į gylį algoritmą (iteracinį ar rekursinį), kad apėjimo metu žymėtų briaunas. Rekursinis algoritmas galėtų būti toks:

DFSmedis( $V$ )

{ Suformuoja aprėpties medį, pradėdamas nuo viršūnės  $V$  ir naudodamas paieškos į gylį metodą }

*Pažymėti viršūnę  $V$  kaip aplankytą*

*for kiekvienai neaplankytai viršūnei  $U$  kaimyninei su  $V$  do*

*begin*

*Pažymėti briauną iš  $V$  į  $U$  kaip aplankytą*

*DFSmedis( $U$ )*

*end*

### BFS aprėpties medis

Kitas būdas sukonstruoti aprėpties medį jungiam neorientuotam grafiui yra apeiti grafo viršūnes, naudojant paieškos į plotį algoritmą. Apėjimo metu žymimos briaunos, kuriomis einama. Baigus apėjimą, pažymėtos briaunos sudarys **paieškos į plotį aprėpties medį** (*BFS spanning tree*).

Tam reikia modifikuoti paieškos į plotį algoritmą taip, kad jis pažymėtų briauną iš viršūnės  $W$  į viršūnę  $U$ , prieš pridėdamas viršūnę  $U$  į eilę.

```
{Q – eilė}
Add(Q, 'V') {įdedame viršūnę V į eilę}
Save('V', 'u') {Išsaugome briauną (V, 'u')}
MarkV('V') {pažymi aplankytą viršūnę}
While (not IsEmpty(Q)) do
    begin
        w := Get(Q) (Nuskaitomas ir išmetamas pirmas eilės elementas)
        for  $\forall$  'u', kuri yra neaplangyta kaimynė 'w' do
            begin
                Save(w, 'u')
                MarkV('u')
                Add(Q, 'u')
            end
        end
    end
```

### Minimalūs aprėpties medžiai (MAM) (*Minimum Spanning Trees*)

Tarkime, reikia suprojektuoti telefonų linijų sistemą, leidžiančią visiems miestams kalbėtis tarpusavyje. Akivaizdus sprendimas nutiesti telefonų linijas tarp bet kurių dviejų miestų. Tačiau sujungti kai kurias poras miestų gali būti neįmanoma, pavyzdžiui, dėl kalnų. Ištyrus galimybes, rezultate buvo gautas jungus neorientuotas grafas su svoriais, kuriame briauna reiškia, kad nutiesti telefono liniją galima, bei briaunos svoris reiškia linijos nutiesimo kainą. Akivaizdu, kad norima nutiesti linijų sistemą kuo pigiau. Jei briaunos būtų be svorių (arba visų briaunų svoriai būtų vienodi), tereiktų surasti gautam grafiui aprėpties medį. Bendra linijų nutiesimo kaina yra *aprėpties medžio kaina*. Kadangi gali egzistuoti ne vienas aprėpties medis ir jų kaina gali būti skirtinga, uždavinio sprendimas yra pasirinkti medį su mažiausia kaina. Toks medis vadinamas *minimaliu aprėpties medžiu*. Toks medis gali būti ne vienintelis, bet visų jų kainos yra lygios.

#### *Kruskal'io Algoritmas.*

$E_{(1)}$  – aibė briaunų priklausančių MAM.

$E_{(2)}$  – aibė likusių briaunų.

$V(x)$  –  $x$  yra viršūnė.

#### Pseudokodas

$E_{(1)} = 0, E_{(2)} = E$

While ( $E_{(1)}$  turi mažiau  $n-1$  briaunų AND  $E_{(2)} \neq \emptyset$ )

{

  Iš  $E_{(2)}$  išrinkti  $e(ij)$  su mažiausiu svoriu.

$E_{(2)} = E_{(2)} - [e(ij)]$

  If  $V(i), V(j) \notin$  tam pačiam medžiui {Apjungti medžius su  $V(i)$  ir  $V(j)$  į vieną. }

}

## Trumpiausi keliai (*Shortest Paths*)

Tarkime, kad orientuotas grafas su svoriais vaizduoja lėktuvų maršrutus: viršūnės – miestai, briaunos – egzistuojantys skrydžiai, o svoriai – atstumai (neneigiami).

Dažnai orientuotam grafiui su svoriais reikia sužinoti trumpiausią kelią tarp kažkurių dviejų viršūnių. Trumpiausias kelias yra kelias, kurio svoris (jį sudarančių briaunų svorių suma) yra minimalus. Nors tradiciškai vartojama sąvoka trumpiausias kelias, bet svoriai nebūtinai turi būti atstumai, jie gali išreikšti ir, pavyzdžiui, kainą, skrydžio laiką.

Trumpiausio kelio grafe radimo algoritmas priskiriamas E.Dijkstra. Patogumui pažymėkime pradinę viršūnę, iš kurios ieškome kelio, '1', o visas likusias grafo viršūnes - nuo '2' iki N. Pastebėkime, kad algoritmas randa trumpiausius kelius tarp pradinės viršūnės ir visų kitų grafo viršūnių.

Algoritmas naudoja pasirinktų viršūnių aibę S ir masyvą W, kur  $W[v]$  yra svoris trumpiausio (pigiausio) kelio iš viršūnės 1 į viršūnę v, einančio per aibės S viršūnes. Jei  $v \in S$ , tai kelias eina tik per aibės S viršūnes. Jei  $v \notin S$ , tai yra vienintelė viršūnė, priklausanti keliui, bet nepriklausanti S, t.y. kelias baigiasi briauna, jungiančia viršūnę iš S ir viršūnę v. Pradžioje aibėje S yra tik viršūnė 1 ir masyve W yra tik svoriai kelių, sudarytų iš vienos briaunos tarp viršūnės 1 ir kitų viršūnių, kitaip sakant masyvas W yra pirma kaimynystės matricos A eilutė.

Po inicializavimo žingsnio pasirenkamos viršūnės, nepriklausančios aibei S, ir atitinkamai koreguojamas masyvas W. Pasirenkama viršūnė v, nepriklausanti aibei S, tokia, kad  $W[v]$  yra minimalus. Pridėjus viršūnę v į aibę S, reikia patikrinti reikšmes  $W[u]$  visoms viršūnėms u, nepriklausančioms S, tam, kad užtikrinti, jog šios reikšmės iš tikrųjų yra minimalios. Kitais žodžiais, reikia patikrinti, ar galima sumažinti  $W[u]$  – kelią  $1 \Rightarrow u$ , einantį per naujai pasirinktą viršūnę v. Tam kelią nuo viršūnės 1 iki viršūnės u galima suskaidyti į 2 dalis ir rasti jų svorius:

```
W[v] = svoris trumpiausio kelio nuo 1 iki v
A[v, u] = svoris briaunos iš v į u
Tada reikia palyginti dabartinį W[u] su W[v] + A[v, u] ir pakeisti W[u], jei nauja reikšmė
mažesnė. Trumpiausio kelio radimo algoritmo pseudokodas:
ShortestPath(G, W)
{ Randa trumpiausius kelius tarp viršūnės 1 ir visų kitų viršūnių orientuotame grafe G su N
  viršūnių }
{ žingsnis 1: Inicializavimas }
S := [1];
for v := 1 to N do
  W[v] := A[1, v]
  { žingsniai 2 ... N }
  { Ciklo invariantas:
    v ∈ S, W[v] yra trumpiausias kelias iš 1 į v, einantis tik per viršūnes iš S iki v;
    v ∉ S, W[v] yra trumpiausias iš visų kelių iš 1 į v, einantis tik per S viršūnes
  }

  for Step := 2 to N do
    begin
      Rasti mažiausią W[v], kur v nepriklauso S
      S := S + [v];

      { Patikrinti W[u] visoms u, nepriklausančioms S }
      for visoms u nepriklausančioms S do
        if W[u] > W[v] + A[v, u]
          then W[u] := W[v] + A[v, u]
      end { for }
    end { for }
```

Ciklo invariantas teigia, kad jei  $v$  yra įtraukta į  $S$ ,  $W[v]$  yra absoliučiai trumpiausias kelias ir nesikeis. Algoritmas tikrai randa trumpiausią kelią, jei ciklo invariantas yra teisingas. Tai galima įrodyti indukcijos pagal žingsnius būdu.

### Kelios sudėtingos problemos

**Prekybos agento uždavinys. Grandinė** (*circuit*) yra kelias, kuris prasideda viršūnėje  $v$ , apeina visas grafo viršūnes ir baigiasi viršūnėje  $v$ . Akivaizdu, kad nejungiamo grafe negali būti grandinės. Tačiau nustatyti ar duotame grafe egzistuoja grandinė gali būti sudėtinga. Gerai žinomas šio uždavinio variantas – prekybos agento uždavinys: grafas su svoriais vaizduoja kelių žemėlapi, briaunų svoriai išreiškia atstumus tarp miestų (arba laiką); prekybos agentas turi pradėti kelionę duotame mieste, aplankyti kiekvieną miestą vienintelį kartą ir grįžti į pradinį miestą, tačiau grandinė, kuria keliauja prekybos agentas, turi būti pigiausia. Žinoma, šio prekybos agento uždavinio sprendimas yra ne paprastesnis nei nustatymas, ar grandinė egzistuoja.

**Trijų komunalinių paslaugų uždavinys.** Tarkime, yra 3 namai  $A$ ,  $B$  ir  $C$  bei 3 komunalinės paslaugos  $X$ ,  $Y$  ir  $Z$  (pavyzdžiui, telefonas, vanduo ir elektra). Jei namai ir komunalinės paslaugos yra grafo viršūnės, ar galima sujungti kiekvieną namą su kiekviena komunaline paslauga taip, kad briaunos nepersikirstų. Atsakymas – ne, kadangi grafas yra izomorfinis  $K_{3,3}$ , kuris nėra plokščias. Trijų komunalinių paslaugų uždavinio apibendrinimas – nustatyti, ar duotas grafas yra planarinis. Ši grafo savybė yra svarbi daugelyje taikymų. Pavyzdžiui, grafas gali vaizduoti elektrinę grandinę, kur viršūnės atitinka elementus, o briaunos ryšius tarp elementų. Ar galima suprojektuoti elektrinę grandinę taip, kad ryšiai nesikirstų?

**Keturių spalvų uždavinys.** Ar duotą planarinį grafą galima nuspalvinti ne daugiau kaip 4 spalvomis taip, kad nebūtų vienos spalvos kaimyninių viršūnių? Atsakymas į šį klausimą – taip, bet įrodyti tai yra sudėtinga. Iš tikrųjų šis uždavinys buvo suformuluotas daugiau kaip prieš 100 metų, tačiau įrodytas tik 1970-aisiais panaudojus kompiuterius.

## Dėstymo lentelės (*hash tables*)

Dvejetainiai medžiai suteikia labai efektyvius darbo metodus su lentelėmis. Pavyzdžiui, tam, kad rastume reikiamą elementą lentelėje, kurioje yra apie 10.000 elementų reikia daugiausiai  $\log_2 10.000 \approx 13$  žingsnių. Tačiau būna situacijų, kai ir toks žingsnių skaičius yra per didelis. Tokiais atvejais reikalinga visiškai skirtinga lentelių apdorojimo strategija.

Įsivaizduokime, kad turime metodą, kuris leidžia rasti ar įterpti elementą praktiškai momentaliai.

Tarkime turime masyvą  $T[0..N-1]$ , bei “magišką juodą dėžutę” - “adresų skaičiuoklę”, kuri kiekvieną kartą, kai norėsime įterpti naują elementą, iškart pasakys, kur turi būti įrašytas duotasis elementas. O jeigu prireiks ištraukti elementą, tai “adresų skaičiuoklė” momentaliai nurodys, kur gali būti rastas ieškomas elementas. Tokiu būdu nereikia ieškoti elementų, tereikia tik nustatyti jų adresus lentelėje. Sugaištas laikas priklausys tik nuo to, kaip greitai “adresų skaičiuoklė” apskaičiuos adresą.

Tokia “adresų skaičiuoklė” vadinama **maišos funkcija** (*hash function*), metodas – **hash'avimu** (*hashing*), o lentelė  $T$  – **dėstymo lentelė** (*hash table*).

Pateiksime vieną pavyzdį. Sakykime, turime masyvą  $T[0..100]$ , mūsų raktai yra sveiki teigiami skaičiai. Tada mūsų *hash* funkcija priima kaip argumentą bet kokią sveiką teigiamą skaičių ir pagal jį vienareikšmiškai nustato vietą duotam raktui.

$$h(x) = n, n \in [0...100]$$

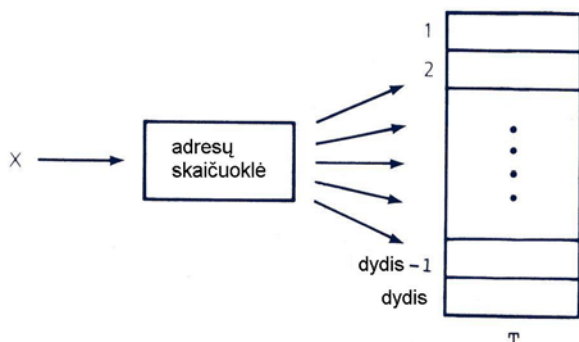
Atrodo, jog visos operacijos yra beveik momentiškos, tačiau ar iš tikrųjų hash'avimas yra tiek geras, kaip atrodo? Jei būtų taip, tai nebūtų prasmės vystyti kitas duomenų struktūras. Būtų idealu jei būtų galima visada vienareikšmiškai nustatyti, koks elementas kur turi būti saugomas. Tokia funkcija vadinama **idealia maišos funkcija**, tačiau tam būtina iš anksto žinoti visas galimas rakto reikšmes, kas ne visada yra įmanoma. Dažniausiai atsitinka tokia situacija, kai į vieną vietą pretenduoja du arba daugiau raktų. Tokia situacija vadinama **kolizija**. Netgi, kai elementų yra gerokai mažiau negu vietos lentelėje, tokie atvejai pasitaiko.

Pvz.: jei turime du raktus 123445678 ir 1234456779 ir  $h(123445678) = h(1234456779) = 44$ , tai abu elementai pretenduoja užimti tą pačią vietą – susidaro kolizija.

Vienas iš būdų yra paruošti iš anksto vietą visiems galimiems raktams, tačiau tai yra labai neefektyvu, todėl toks sprendimas praktiškai yra netinkamas. Taigi tokie atvejai buvo sukurti specialūs kolizijų sprendimo metodai.

Apibendrinant, gera maišos funkcija pasižymi šiomis savybėmis:

1. Lengvai ir greitai apskaičiuojama
2. Tolygiai skirsto raktus po visą lentelę.



## Kolizijų sprendimas

Apžvelkime problemas, susijusias su kolizijomis. Tarkime, norime įterpti į lentelę elementą, kurio raktas yra 123445678. Jei skaičiui 123445678 pritaikysime maišos funkciją  $h(x) = x \bmod 101$ , tai gausime, kad elementas turės būti padėtas į  $T[44]$ . Tačiau, tarkime, kad  $T[44]$  jau yra narys, kurio paieškos raktas yra 123445779, t.y. anksčiau jau padėjome elementą su raktu 123445779 į  $T[44]$ , nes  $123445779 \bmod 101 = 44$ . Išskyla problema: kur dėti naują elementą? Akivaizdu, kad negalime nepadėti šio elemento į lentelę, pagrįsdami savo sprendimą tuo, kad lentelė užpildyta, nes kolizija įmanoma ir tada, kai lentelėje yra tik vienas elementas.

Yra du iš esmės skirtingi kolizijų problemos sprendimo būdai. Vienas iš būdų yra tiesiog rasti naujam elementui kitą vietą. Šis būdas vadinamas atviru adresavimu (*open addressing*). Kitas būdas yra pakeisti lentelės struktūrą taip, kad kiekvienoje vietoje  $T[i]$  galėtų tilpti daugiau nei vienas elementas. Keturi žemiau pateikti kolizijų problemos sprendimo metodai iliustruoja šiuos du būdus.

**Tiesinis dėstymas** (*linear probing*). Tai atviro adresavimo metodas: jei bandoma įterpti naują elementą, o pagal maišos funkciją gauta jo vieta jau užimta, tai ieškoma kokios nors kitos tuščios (*atviros*) vietos tam elementui padėti. Aišku, kad nauja vieta turėtų būti parinkta taip, kad po nario įterpimo jį galima būtų efektyviai rasti, t.y. algoritmas, kuris randa naują vietą funkcijoje „įterpti“ turėtų būti lengvai atkartojamas tokiose funkcijose, kaip „išmesti“ ir „gauti“.

Skirtumas tarp įvairių atviro adresavimo realizacijų yra funkcijos, naudojamos naujai vietai lentelėje rasti. Vienas dažniausiai naudojamų būdų yra tiesinis dėstymas. Remiantis šiuo metodu bandoma narį padėti į vietą, rastą pagal maišos funkciją, pvz.,  $T[h(\text{key})]$ . Jei ta vieta yra užimta, bandoma narį dėti į sekančią vietą, t.y.  $T[h(\text{key})+1]$ . Jei ir ta vieta užimta, tai bandoma dėti į  $T[h(\text{key})+2]$  ir t.t. tol, kol nebus rasta tuščia vieta. Dažniausiai, jei pasiekiamas lentelės galas, tai peršokama į lentelės pradžią, t.y. jei lentelėje yra  $n$  vietų, tai po  $T[n]$  vietos bandoma dėti į  $T[1]$  vietą.

Jei nėra operacijos „išmesti“, tai funkciją „gauti“ realizuoti pakankamai lengva. Naudojamas tas pats būdas kaip ir funkcijoje „įterpti“. Jei ieškomo nario nėra vietoje, į kurią rodo maišos funkcija, tai peržiūrimi kiti elementai tol, kol jis nebus galiausiai rastas, nebus pasiekta tuščia lentelės eilutė arba nebus peržiūrėti visi lentelės nariai.

Jei operacija „išmesti“ visgi yra, tai realizacija tampa sudėtingesnė. Pačią funkciją „išmesti“ realizuoti nėra sunku: naudojamas „gauti“ funkcijoje aprašytas būdas. Paskui nario užimta lentelės eilutė yra atlaisvinama. Tačiau po tokios operacijos „gauti“ funkcija pradeda veikti nekorektiškai: ji gali rasti funkcijos „išmesti“ ištuštintas vietas ir klaidingai pranešti, kad ieškomo nario nėra. To išvengiama, jei lentelės vietos gali būti vienoje iš trijų būsenų: užimta (šioje vietoje yra narys), tuščia (šioje vietoje nėra ir nebuvo nario) ir ištrinta (šioje vietoje buvo narys, tačiau jis buvo ištrintas). Po to yra patobulinamas funkcijos „gauti“ algoritmas taip, kad funkcija ieškotų nario nepaisant vietų, kurių būseną yra „ištrinta“. Panašiai tobulinama ir funkcija „įterpti“: jai leidžiama rašyti ne tik į tuščias, bet ir į ištrintas vietas.

**Efektyvumas.** Panagrinėkime šio metodo efektyvumą. Kuo didesnė tampa lentelė, tuo didesnė tikimybė, kad įvyks kolizija. Kadangi auga kolizijų skaičius, paieškos laikas ilgėja. Taigi nesėkminga paieška užtrunka ilgiau nei sėkminga. Pvz., lentelėje, kuri yra  $2/3$  pilna, nesėkmingai paieškai prireiks vidutiniškai 5 palyginimo operacijų, kai sėkmingai – daugiausiai dviejų. Norint palaikyti lentelės efektyvumą, reikia neleisti jai prisipildyti.

Viena iš tiesinio dėstymo problemų yra ta, kad po kurio laiko vienoje lentelės dalyje narių bus daug, o kitoje, palyginti mažai. Dėl to žymiai pailgėja paieškos laikas ir sumažėja efektyvumas.

**Dvigubas dėstymas** (*double hashing*). Dvigubas dėstymas padeda išspręsti problemą, aprašytą anksčiau – elementų susigrūdimą kai kuriose lentelės dalyse. Esminis dvigubo dėstymo skirtumas nuo tiesinio dėstymo yra tas, kad lentelės vietos yra peržiūrimos ne iš eilės (viena po kitos), o su tam tikru žingsniu. Tam žingsniui apskaičiuoti naudojama kita maišos funkcija. Pvz., tegu  $h_1$  ir  $h_2$  bus atitinkamai pirma ir antra maišos funkcijos, kurios apibrėžtos taip:

$$h_1(\text{key}) = \text{key} \bmod 11$$

$$h_2(\text{key}) = \text{key} \bmod 7$$

Jei  $key = 25$ , tai  $h1$  rodo, kad elementas turės būti padėtas į 3 vietą ( $25 \bmod 11$ ), o  $h2$  rodo, kad perrinkimų žingsnis yra 4 ( $25 \bmod 7$ ), t.y. vieta elementui bus ieškoma taip: visų pirmą bus pažiūrėta 3 vieta ( $h1$  funkcijos reikšmė), po to 7 ( $3+4$ ), 0 (kadangi po paskutinės eilutės seka pirma), 4, 8, 1 (nes vėl yra peršokama į lentelės pradžią), 5, 9, 2, 6, 10, 3.

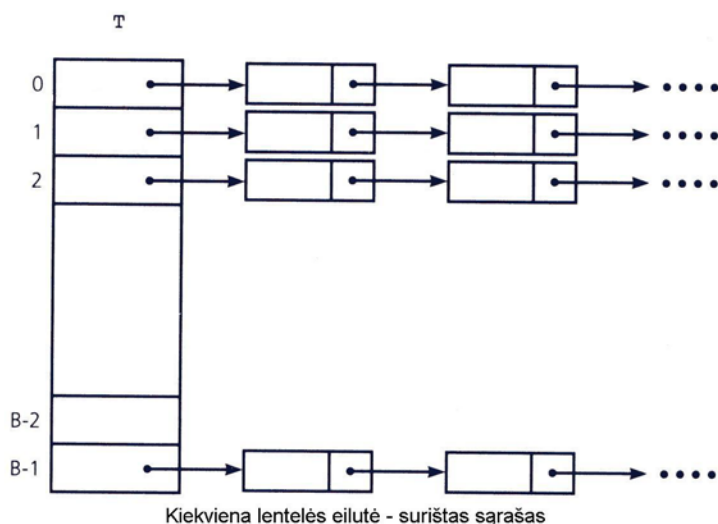
Jei  $key$  yra lygus 9, tai pirmiausiai bus pažiūrėta 9 vieta ( $h1$  funkcijos reikšmė), o perrinkimų žingsnis bus 2 ( $h2$  funkcijos reikšmė). Taigi paieškos bus peržiūrėtos šios vietos: 9, 0, 2, 4, 6, 8, 10, 1, 3, 5, 7, 9.

Atkreipkite dėmesį į tai, kad peržiūrimos visos lentelės eilutės. Taip atsitinka, kai lentelės dydis ir perrinkimų žingsnio dydis yra tarpusavyje pirminiai skaičiai, t. y. jų didžiausias bendras daliklis yra 1. Kadangi lentelės dydis yra dažniausiai pirminis skaičius, todėl jis ir bet koks perrinkimų žingsnio dydis bus tarpusavyje pirminiai skaičiai.

Dažniausiai dvigubam dėstymui reikia mažiau palyginimų nei tiesiniam dėstymui. Taigi, naudojant šį metodą galima apsieiti mažesne lentele, nei naudojant tiesinę paiešką. Tačiau kadangi abu šie metodai remiasi atviros adresacijos idėja, abiejų jų trūkumas iškyla, kai nežinome, kiek bus įdėjimo ir išmetimo operacijų. Jei lentelė bus per maža, ji greitai prisipildys, ir paieškos efektyvumas sumažės.

**Bucket'ai.** Kitas metodas, naudojamas kolizijų problemai spręsti, yra toks: yra keičiama lentelės struktūra taip, kad vienoje vietoje galėtų tilpti daugiau nei vienas elementas. Galima, pavyzdžiui, pakeisti masyvą  $T$  taip, kad kiekvienas jo narys  $T[i]$  irgi būtų masyvas, vadinamas *Bucket* (kibiras), kuriame telpa  $B$  narių. Esminė šio metodo problema yra ta, kad pakankamai sunku rasti tinkamą skaičių  $B$ . Jei  $B$  yra mažas, tai kolizijų problema bus išspręsta tik tol, kol visuose masyvo  $T$  nariuose dar yra vietos naujiems elementams. Tačiau, jei  $B$  bus didelis skaičius, tai bus veltui naudojama atmintis.

**Atskiros eilės** (*separate chaining*). Geresnis už anksčiau aprašytą būdą gali būti toks: lentelės vietos yra ne masyvai, o sąrašai. T.y. pagal šį metodą kiekvienas  $T$  masyvo elementas yra rodyklė į sąrašo pradžią. Sąrašuose laikomi nariai, kurių vietą nurodė hash funkcija.



Kai į lentelę įterpiamas naujas elementas, jis patalpinamas į sąrašo, kurį nurodė maišos funkcija, pradžią.

Jeigu norime gauti elementą, tai ieškome jo sąrašo, kurį nurodė maišos funkcija.

Funkcijos „išmesti“ algoritmas labai panašus į funkcijos „gauti“, todėl nėra nagrinėjamas.

Taigi atskirų eilių metodas yra pakankamai sėkmingas būdas kolizijoms išvengti. Atkreipkite dėmesį į tai, kad bendras lentelės dydis dabar yra dinaminis, nes kiekvienas sąrašas gali būti kiek norima ilgas. Tačiau kyla klausimas: kiek efektyvios lentelės, realizuotos atskiromis eilėmis, operacijos. Funkcija „įterpti“ liko tokia pat efektyvi, kadangi norimas sąrašas yra randamas pasinaudojus maišos funkcija. Tačiau funkcijoms „gauti“ ir „ištrinti“ reikės ne tik rasti norimą sąrašą, bet ir jame rasti konkretų elementą. Paanalizuokime šių funkcijų efektyvumą įdėmiau. Jei masyvas  $T$  turi  $TableSize$  narių, ir lentelėje yra  $N$  narių, tai vidutinis kiekvieno sąrašo ilgis yra  $N/TableSize$ . Kai randame masyvo  $T$  ilgį,



turime atsižvelgti į skaičių  $N$ . Skaičius  $\text{TableSize}$  turi būti parinktas taip, kad skaičius  $N/\text{TableSize}$  būtų mažas. Pvz., galima parinkti skaičių  $\text{TableSize}$  artimą  $N$  (didžiausias lentelės elementų skaičius). Jei pasiseks randant skaičių  $\text{TableSize}$ , tai tikėtina, kad sąrašai, kuriuose funkcijos „įterpti“ ir „išmesti“ ieško narių, bus pakankamai trumpi.

Dabar panagrinėkime blogiausią atvejį. Tarkime, kad smarkiai nuvertinome  $\text{TableSize}$  arba kad didesnė reikšmių dalis (arba visos reikšmės) pateko į tą patį sąrašą.

Kaip matome funkcijų „gauti“ ir „ištrinti“ veikimo laikas gali smarkiai kisti: nuo labai mažo (jei sąrašas yra pora reikšmių) iki labai didelio (jei į sąrašą pateko visos reikšmės). Taigi nepaisant to, kad realizacija dėstymo lentelės dažnai gali būti greitesnė nei realizacija paieškos medžiu, blogiausiu atveju ji gali būti daug lėtesnė. Jei taip atsitinka, reikia pakeisti maišos funkciją arba padidinti lentelės dydį. Nereikėtų naudoti sudėtingų kolizijų problemos sprendimo algoritmų.

### ADT Dėstymo lentelė (Hash table)

Nagrinėjama dėstymo lentelės realizacija turi du parametrus, kurie įtakoja jos efektyvumą: *lentelės dydį* bei *apkrovimo koeficientą*. Apkrovimo koeficientas gali būti tarp  $0.0$  ir  $1.0$ . Kai įrašų skaičius dėstymo lentelėje viršija apkrovimo koeficientą einamajai lentelės talpai, dėstymo lentelės talpa yra padidinama iškviečiant atnaujinimo (*rehash*) metodą. Lentelė su didesniu apkrovimo koeficientu efektyviau naudoja atmintį, bet informacijos paieška gali trukti ilgiau. Jei dėstymo lentelėje bus sukurta daug įrašų, pakankamai didelės pradinės talpos dėstymo lentelės sukūrimas leis įterpti įrašus daug efektyviau negu atveju, kai įterpian įrašus reiks didinti lentelę, atliekant automatinį atnaujinimą (*rehash*).

- **create** ( [int initialCapacity, float loadFactor] )

Sukuria naują (tuščią) dėstymo lentelę. Nebūtini parametrai: pradinė talpa bei duotas apkrovimo koeficientas.

- **put**(key, object)

Įdeda naują elementą į lentelę. Key – raktas, pagal kurį skaičiuojama hash reikšmė. Object – duomenų rinkinys.

- **get**(key)

Gražina reikšmę, kuri yra atitinka raktą dėstymo lentelėje, arba *Null* reikšmę, jei nerasta ieškomo rakto.

- **remove**(key)

Išmeta raktą atitinkančią reikšmę iš dėstymo lentelės. Šis metodas nieko nedaro, jei rakto nėra dėstymo lentelėje.

- **isEmpty**()

Patikrina, ar dėstymo lentelė yra tuščia.

- **size**()

Gražina raktų skaičių dėstymo lentelėje.

- **clear**()

Išvalo dėstymo lentelę.

- **search**(key)

Patikrina, ar duotas raktas yra dėstymo lentelėje.

- **rehash**()

Perkelia dėstymo lentelės turinį į didesnės talpos dėstymo lentelę. Šis metodas yra iškviečiamas automatiškai, kai raktų skaičius dėstymo lentelėje viršija tos lentelės apkrovimo koeficientą einamajai lentelės talpai.

Pagal knygą: *Michael Main, Walter Savitch, Data Structures and Other Objects, A Second Course in Computer Science 9Turbo Pascal Edition*, The Benjamin/Cummings Publishing Company, 1995

### Heapsort algoritmas

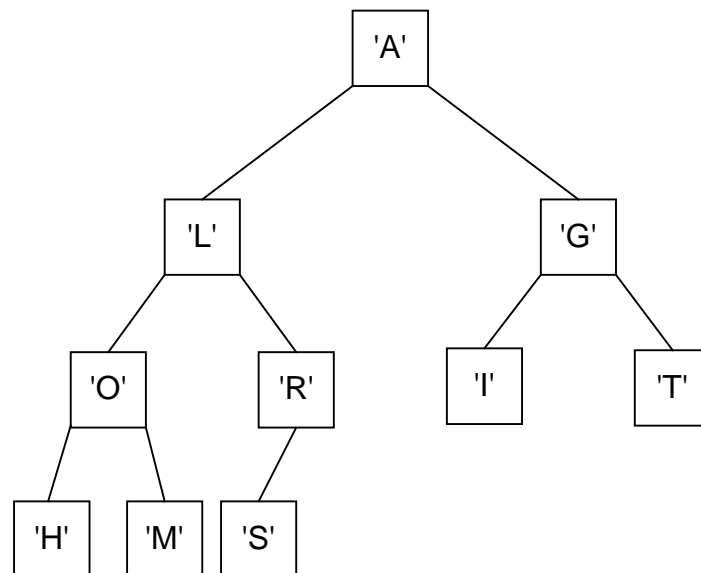
*Heapsort* algoritmo idėja yra panaši į rūšiavimo paieška (*selection sort*) algoritmo. Prisiminkime, kaip atliekamas rūšiavimas paieška: surandamas mažiausias elementas ir jis padedamas į pirmą vietą, tada surandamas antras pagal dydį elementas (arba mažiausias iš likusių) ir jis padedamas į antrą vietą ir t.t. *Heapsort* algoritmo strategija panaši: surandama didžiausia reikšmė ir ji padedama į paskutinę poziciją ir t.t. Tačiau *heapsort* algoritmas naudoja žymiai efektyvesnį būdą reikiamos reikšmės suradimui. *Heapsort* algoritmas transformuoja rūšiuojamą masyvą į duomenų struktūrą *heap*.

Prisiminę, kad duomenų struktūra *heap* yra užbaigtas (*complete*) dvejetainis medis ir kad pilną dvejetainį medį efektyviai atvaizduojamas masyve, galime aptarti *heapsort* algoritmo idėją.

*Heapsort* algoritmas traktuoja rūšiuojamą masyvą kaip užbaigtą dvejetainį medį, pavyzdžiui, masyvas

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
'A'	'L'	'G'	'O'	'R'	'I'	'T'	'H'	'M'	'S'
Šaknis	1 lygis		2 lygis				3 lygis		

gali būti traktuojamas kaip užbaigtas dvejetainis medis:



Kadangi masyve reikšmės išsidėsčiusios bet kokia (atsitiktine) tvarka, todėl ir medyje reikšmės bus išsidėsčiusios bet kokia (atsitiktine) tvarka. *Heapsort* algoritmas siekia, kad reikšmės medyje (masyve) būtų išsidėsčiusios taip, kad sudarytų duomenų struktūrą *heap* (tenkintų jos reikalavimus).

Pirmas *heapsort* algoritmo žingsnis yra pertvarkyti masyvo reikšmes taip, kad atitinkamas pilnas dvejetainis medis būtų duomenų struktūra *heap*, t.y. kiekvienos viršūnės reikšmė būtų nemažesnė už visų jos vaikų reikšmes.

Kai duomenys tenkins duomenų struktūros *heap* reikalavimus, didžiausias elementas bus medžio šaknyje – pirmame masyvo elemente. Tačiau mūsų tikslas yra surūšiuoti elementus nuo mažiausio iki didžiausio, todėl tam, kad didžiausias elementas atsikurtų savo pozicijoje, paprasčiausiai sukeičiame jį su paskutiniu elementu.

Po šio sukeitimo didžiausias elementas yra paskutinėje pozicijoje, bet likusi masyvo dalis (be paskutinio elemento) nebėra duomenų struktūra *heap*. Tačiau likusi masyvo dalis yra *beveik* duomenų struktūra *heap*: joje yra vienintelė reikšmė (ką tik padėta į šaknį), kuri gali pažeisti duomenų struktūros *heap* reikalavimus. Todėl sekantis algoritmo žingsnis yra atstatyti duomenų

struktūrą *heap*, „nuleidžiant“ šaknies reikšmę (šis algoritmas jau buvo aptartas, nagrinėjant duomenų struktūrą *heap*): šaknies reikšmę lyginama su dviejų jos vaikų reikšmėmis ir, jei ji yra mažesnė už bent vieno vaiko reikšmę, šaknies reikšmė sukeičiama su didžiausio vaiko reikšme – t.y. reikšmė „nuleidžiama“ vienu lygiu; šis procesas kartojamas tol, kol „probleminė“ reikšmė pasiekia lapą arba ji tampa nemažesnė už savo vaikų reikšmes.

Liko neaptarta, kaip bet kokią pilną dvejetainį medį pertvarkyti į duomenų struktūrą *heap*. Tam gali būti naudojama reikšmių „pakėlimas“, darant vis didesnę duomenų struktūrą *heap* nuo masyvo pradžios. Reikšmės „pakėlimas“ yra procesas, kai reikšmė yra sukeičiama su jos tėvo reikšme, jei ji didesnė už tėvo reikšmę. Procesas baigiamas, kai reikšmė atsiduria šaknyje arba ji yra nedidesnė už tėvo reikšmę.

Pradinė duomenų struktūra *heap* sudaryta iš pirmo masyvo elemento  $A[1]$ . Akivaizdu, kad vienintelė reikšmė tenkina duomenų struktūros *heap* reikalavimus. Tada pridėdame elementą  $A[2]$  ir, jei reikia, atliekame reikšmės „pakėlimą“. Tai kartojama visiems masyvo elementams.

*Heapsort* algoritmo schema:

1. Sukurti pradinę duomenų struktūrą *heap*, plečiant ją nuo pirmojo elemento iki visų masyvo elementų ir naudojant reikšmės „pakėlimo“ operaciją (rezultate  $A[1]$  yra didžiausias elementas).
2. Riba = elementų skaičius
3. Kol  $Riba > 1$  kartoti 3 žingsnius:
  - 3.a. Sukeisti elementus  $A[1]$  ir  $A[Riba]$ .
  - 3.b.  $Riba = Riba - 1$
  - 3.c. „Nuleisti“ reikšmę  $A[1]$  duomenų struktūroje *heap*, iš  $Riba$  elementų.

#### *Heapsort* algoritmo analizė

Įvertinkime *heapsort* algoritmo sudėtingumą blogiausiu atveju. Algoritmas naudoja tik priskyrimo, palyginimo ir aritmetinės (sudėties, atimties) operacijas. Tarkime, kad elementų yra  $N$ . Panagrinėję algoritmą galime pastebėti, kad blogiausiu atveju jis atlieka:

fiksuotą skaičių operacijų

+ operacijas pradinės duomenų struktūros *heap* sukūrimui (1 žingsnis)

+ operacijas, atliekamas cikle (3 žingsnyje).

3 žingsnio ciklas kartojamas  $N-1$  kartą, bet algoritmo sudėtingumo vertinime (naudojant  $O$ -notaciją) galime vertinti kaip  $N$ . Kiekvieną kartą cikle atliekamas fiksuotas operacijų skaičius (žingsniai 3.a ir 3.b) ir operacijos skaičius (kurį dar reikia įvertinti) reikšmės „nuleidimui“.

Taigi bendras operacijų skaičius yra nedidesnis nei:

(fiksuotas operacijų skaičius) +

(operacijos pradinės duomenų struktūros *heap* sukūrimui) +

$N * (\text{fiksuotas operacijų skaičius} + \text{žingsnio 3.c operacijų skaičius})$

Kad nustatyti šios formulės reikšmę, reikia įvertinti „nuleidimo“ ir „pakėlimo“ operacijų sudėtingumą. Jau žinome, kad reikšmės pridėjimo į duomenų struktūrą *heap*, naudojant „pakėlimo“ operaciją, sudėtingumas yra  $O(\log m)$ , kur  $m$  yra elementų skaičius. Analogiškai reikšmės „nuleidimo“ operacijos sudėtingumas yra  $O(\log m)$ . Kadangi  $m$  yra nedidesnis nei  $N$ , tai vertindami blogiausią atvejį, galime laikyti, kad operacijų sudėtingumas yra  $O(\log N)$ . Taigi bendras operacijų skaičius yra nedidesnis nei:

(fiksuotas operacijų skaičius) +

(operacijos pradinės duomenų struktūros *heap* sukūrimui) +

$N * (\text{fiksuotas operacijų skaičius} + O(\log N))$

Kadangi vertiname operacijų skaičiaus priklausomybę nuo duomenų kiekio, fiksuotus operacijų skaičius galime praleisti. Tokiu būdu gauname formulę:

(operacijos pradinės duomenų struktūros *heap* sukūrimui) +  $N * O(\log N) =$

(operacijos pradinės duomenų struktūros *heap* sukūrimui) +  $O(N \log N)$

Pradinės duomenų struktūros *heap* sukūrimui  $N-1$  kartą atliekama reikšmės pakėlimo operacija, todėl bendras operacijų skaičius yra nedidesnis nei  $(N - 1) * O(\log N)$ . Taigi duomenų struktūros *heap* sukūrimo algoritmo sudėtingumas blogiausiu atveju yra:  $O(N \log N)$

O viso *heapsort* algoritmo sudėtingumas blogiausiu atveju yra:

$$O(N \log N) + O(N \log N) = O(2N \log N)$$

Kadangi vertinant sudėtingumą daugybą iš konstantos galima ignoruoti, viso algoritmo sudėtingumas blogiausiu atveju gali būti įvertintas kaip  $O(N \log N)$ .

Griežtas tikėtino (vidutinio) sudėtingumo įvertinimas yra pakankamai komplikotas, bet jis irgi yra  $O(N \log N)$ .

## Maksimalaus AVL medžio aukščio įvertinimas

Koks gali būti maksimalus AVL medžio, turinčio  $n$  viršūnių, aukštis? Norėdami atsakyti į šį klausimą, ieškosime AVL medžio, kurio aukštis  $h$ , minimalaus viršūnių skaičiaus.

Tarkime, jog  $F_h$  yra nagrinėjamas AVL medis, kurio aukštis  $h$ . Tegu  $F_k$  ir  $F_d$  yra jo kairysis ir dešinysis pomedžiai. Vieno iš šių pomedžių, tarkime  $F_k$ , aukštis bus  $(h - 1)$ , o kito ( $F_d$ ) –  $(h - 1)$  arba  $(h - 2)$ . Kadangi medis  $F_h$ , iš visų nagrinėjamų AVL medžių, kurių aukštis  $h$ , turi mažiausią skaičių viršūnių, tai  $F_d$  aukštis turi būti  $(h - 2)$ . Iš čia gauname, kad:

$$|F_h| = |F_{h-1}| + |F_{h-2}| + 1$$

Čia  $|F_h|$  yra medžio, kurio aukštis  $h$ , viršūnių skaičius. Laikoma, kad  $|F_0| = 1$  ir  $|F_1| = 2$ . Tokie AVL medžiai dar yra vadinami *Fibonačio medžiais* (žr. 1 pav.).

Prie gautosios lygybės pridėję po 1, gauname:

$$|F_h| + 1 = (|F_{h-1}| + 1) + (|F_{h-2}| + 1)$$

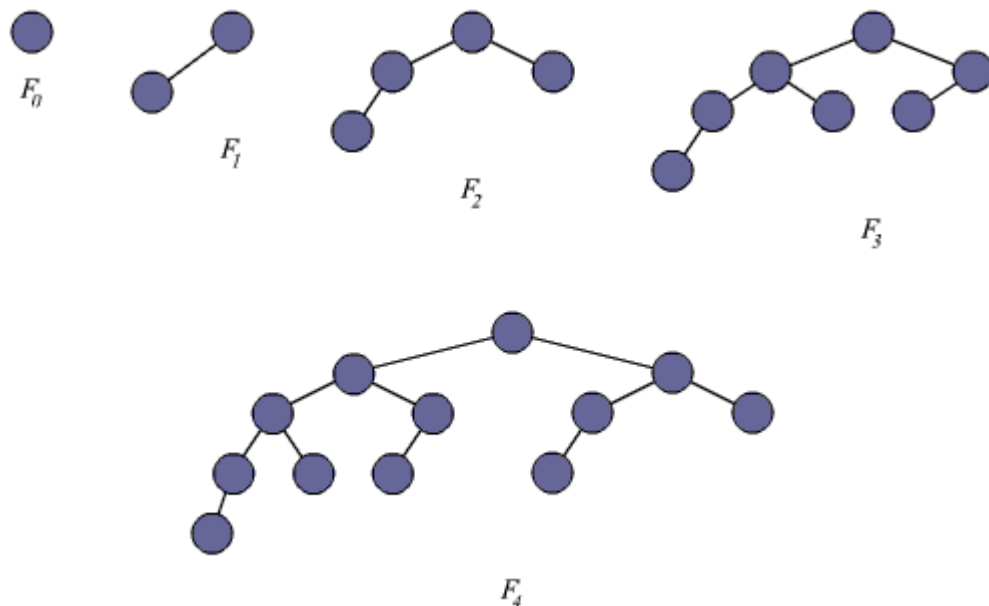
$|F_h| + 1$  yra Fibonačio sekos narys (indeksai skiriasi per 3), todėl jį galima apytiksliai apskaičiuoti:

$$|F_h| + 1 \approx \frac{1}{\sqrt{5}} \left[ \frac{1 + \sqrt{5}}{2} \right]^{h+3}$$

Išreiškus iš šios lygybės  $h \approx 1,44 \lg |F_h|$ . Tai reiškia, kad pačiu blogiausiu atveju AVL medžio su  $n$  viršūnių apytikslis aukštis yra  $1,44 \lg n$ .

Idealiai subalansuoto medžio su  $n$  viršūnių apytikslis aukštis yra  $\lg n$ , o „išsigimusio“ –  $n$ . Taigi algoritmai darbui su AVL medžiais bus visada vykdomi neilgiau, nei  $\approx 44\%$  daugiau laiko už optimalų variantą. Kaip rodo praktika, netgi pats blogiausias AVL atvejis – Fibonačio medis, paieskai sugaišta tik  $\approx 4\%$  daugiau laiko. Be to, dauguma AVL medžių nėra Fibonačio medžiai.

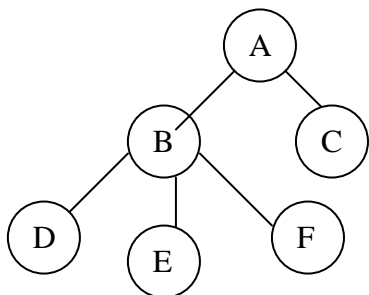
Yra nustatyta, kad AVL medžiai, palyginimo operacijų atžvilgiu, yra labai artimi idealiai subalansuotiems medžiams. AVL medžio vidutinis palyginimo operacijų skaičius labai dideliems  $n$  lygus  $\lg n + 0,25$ .



1 pav. Fibonačio medžių pavyzdžiai

## Medis. Dvejtainis medis

1. **Medžiai** yra hierarchinės struktūros, t.y. tarp medžio viršūnių yra "tėvo-vaiko" santykiai. Paprastas medis atrodo taip:



1 pav.

A, B, C, D, E, F – medžio **viršūnės** (*nodes*). A yra B ir C **tėvas** (*parent*), analogiškai, B yra D, E ir F tėvas, o B ir C yra tėvo A **vaikai** (*children*). B ir C turi tą patį tėvą, todėl jie vadinami **broliais** (*siblings*). A neturi tėvo ir vadinama medžio **šaknimi** (*root*). Jei viršūnė neturi vaikų, ji vadinama **lapu** (*leaf*). 1 pav. lapai yra D, E, F ir C.

Medis, kuris neturi viršūnių yra **tuščias** (*empty*).

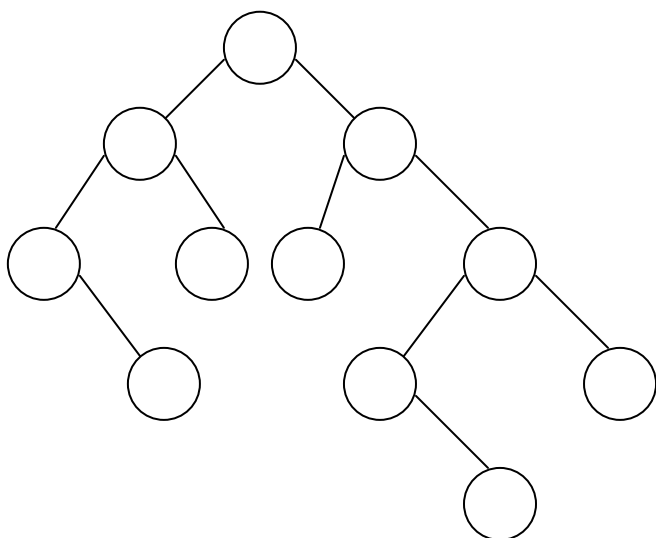
2. **Dvejtainis medis** (*binary tree*) – tai toks medis, kurio kiekviena viršūnė turi ne daugiau kaip 2 vaikus.

Medis yra dvejtainis, jeigu:

- tuščias,
- arba yra pavidalo,

$$\begin{array}{c} r \\ T_k \quad T_d \end{array}$$

kur  $r$  – viršūnė,  $T_k$  – **kairysis pomedis** (*left subtree*) ir  $T_d$  – **dešinysis pomedis** (*right subtree*) D taip pat yra dvejtainiai medžiai. Dvejtainio medžio pavyzdys:



2 pav.

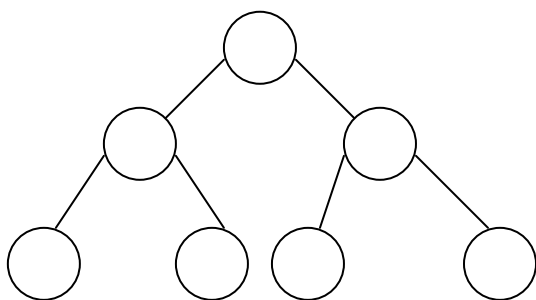
3. **Medžio aukštis ir viršūnės lygis.** Medžio aukštis (*height*) – tai viršūnių skaičius nuo šaknies iki toliausio lapo. Pavyzdžiui, 2 pav. medžio aukštis yra 5 (ieškajome atstumo nuo A iki L). Aukštį galima rasti naudojant viršūnės lygio (*level*) apibrėžimą:

- Jei  $n$  medžio šaknis, tai jos lygis 1.
- Jei  $n$  nėra medžio šaknis, jos lygis vienetu didesnis už tėvo lygį.

Tada aukštį galima apibrėžti taip:

- Jei medis tuščias, jo aukštis 0.
- Jei medis nėra tuščias, jo aukštis lygus maksimaliam viršūnių lygiui

4. **Pilnas (full) dvejetainis medis** tai medis, kurio aukštis  $h$ , turi visus lapus  $h$  lygyje ir visos viršūnės, kurių aukštis mažesnis už  $h$ , turi 2 vaikus. Pilno dvejetainio medžio, kurio aukštis 3, pavyzdys:



3 pav.

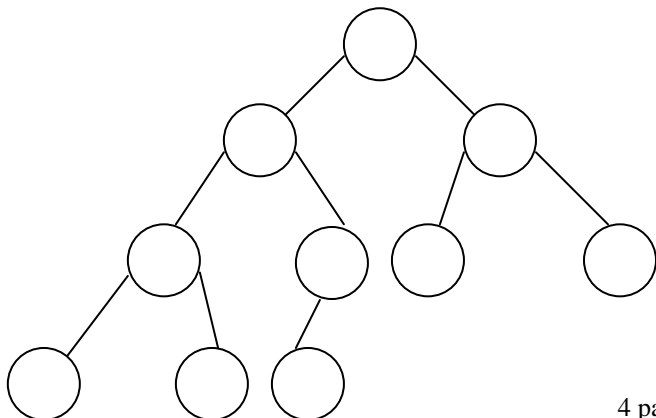
Labai patogiu naudotis šiais rekursyviais apibrėžimais :

- Jei  $T$  yra tuščias, tai  $T$  yra pilnas dvejetainis medis, kurio aukštis 0;
- Jei  $T$  nėra tuščias ir aukštis  $h > 0$ , tai  $T$  yra pilnas dvejetainis medis, jeigu jo šaknies pomedžiai irgi yra pilni dvejetainiai medžiai

Šie apibrėžimai yra panašūs į dvejetainių medžių rekursinius apibrėžimus.

5. **Užbaigtas (complete) dvejetainis medis**, kurio aukštis  $h$ , tai dvejetainis medis, kuris yra pilnas iki aukščio  $h-1$ , o pradedant nuo lygio  $h$  pildomas iš kairės į dešinę. Kitaip tariant, dvejetainis medis  $T$ , kurio aukštis  $h$  yra iki galo užbaigtas, kai:

- Jei viršūnė turi dešinįjį pavaldėtoją, kurio aukštis  $n$ , tai visi lapai jos kairiajame pomedyje yra aukštyje  $n$ , t.y. pomedyje turime visas  $n$  lygio viršūnes;
- Visos viršūnės, kurių aukštis  $h-2$ , turi po 2 vaikus.



4 pav.

**6. Subalansuotas pagal aukštį** (*height balanced*) **medis** tai medis, kurio kiekvienos viršūnės kairiojo ir dešiniojo pomedžio aukščiai skiriasi ne daugiau kaip vienu lygiu. Dvejetainis medis yra visiškai subalansuotas (*completely balanced*), jei kairieji ir dešinieji kiekvienos viršūnės pomedžiai yra to paties aukščio..

**7. Dvejetainis paieškos medis** (*binary search tree*) – tai dvejetainis medis, kuris surūšiuotas pagal reikšmes jo viršūnėse. Kiekvienai viršūnei jis tenkina tokias tris savybes:

1.  $n$ -osios viršūnės reikšmė yra didesnė už visas reikšmes jos kairiajame pomedyje.
2.  $n$ -osios viršūnės reikšmė yra mažesnė už visas reikšmes jos dešiniajame pomedyje.
3. Kairysis ir dešinysis pomedžiai yra dvejetainiai paieškos medžiai.

Dabar pateiksime kelias svarbias teoremas:

**Teorema 1:** Pilnas  $h$  ( $h \geq 0$ ) lygio medis turi  $2^h - 1$  viršūnių.

**Teorema 2:**  $H$  lygio medis turi maksimaliai  $2^h - 1$  viršūnę.

**Teorema 3:**  $N$  viršūnių medžių minimalus aukštis yra  $\lceil \log_2(N+1) \rceil$ .



Pagal knygą: Michael T. Goodrich, Roberto Tamassia, *Algorithm Design: Foundations, Analysis, and Internet Examples*, John Wiley & Sons, 2002

## NP-pilnumas

Yra sudėtingų uždavinių, kuriems vargstame, ieškodami efektyvaus algoritmo. Būtų gerai, jei pavyktų įrodyti, kad efektyvus algoritmas jiems neegzistuoja. Toks įrodymas remiasi sąvoka **NP-pilnumas** (*NP-completeness*), leidžiančia parodyti, kad efektyvaus algoritmo radimas nagrinėjamam uždaviniui yra ne lengvesnis negu radimas efektyvaus algoritmo **visiems** uždaviniams, priklausančioms klasei, vadinamai **NP**. Efektyvus algoritmas čia suprantamas kaip polinominio sudėtingumo algoritmas, t.y. algoritmas, kurio sudėtingumas yra  $O(n^k)$ , kažkokiai konstantai  $k > 0$ , arba, kitaip sakant, algoritmo vykdymo laikas yra  $O(n^k)$ , kur  $n$  – pradinių duomenų dydis.

Iki šiol pradinių duomenų dydį matuodavome elementų skaičiumi. Nagrinėdami NP-pilnumą, **pradinių duomenų dydį** apibrėžiame kaip skaičių bitų, reikalingų šiems duomenims užkoduoti. Remiamės prielaida, kad naudojamas tinkamas kodavimas: simboliams užkoduoti naudojamas fiksuotas bitų skaičius, o sveikam skaičiui  $M > 0$  užkoduoti naudojame ne daugiau kaip  $c \log M$  bitų, kur  $c > 0$  yra konstanta. Tokiu būdu yra uždraudžiamas elementarus kodavimas, kai skaičiaus  $M$  vaizdavimui naudojama  $M$  bitų su reikšme 1. Tarkime, kad pradinių duomenų elementų kiekis yra  $N$ , o jiems užkoduoti reikalingų bitų skaičius yra  $n$ . Tada, jei  $M$  yra didžiausias sveikas skaičius pradinuose duomenyse, tai  $N + \log M \leq n \leq c N \log M$ .

Formaliai algoritmo **A vykdymo laiką** blogiausiu atveju apibrėžiame kaip  $\max(t(n))$  visoms galimoms  $n$  bitų kombinacijoms) kur  $t(n)$  yra algoritmo **A** vykdymo laikas su pradiniais duomenimis  $n$ . Kaip bus parodyta vėliau, dauguma algoritmų su polinominiu vykdymo laiku  $N$  atžvilgiu išlaiko polinominį vykdymo laiką  $n$  atžvilgiu.

Apibrėžkime, kad algoritmas yra **c-augantis** (*c-incremental*), jei visų jo primitivių operacijų su vienu ar dviem objektais, vaizduojamais  $b$  bitų, rezultatas yra objektas, kurio pavaizdavimui reikia ne daugiau kaip  $b+c$  bitų, kur  $c \geq 0$  yra konstanta. Pavyzdžiui, algoritmas, naudojantis daugybą kaip primitivią operaciją, gali nebūti  $c$ -augantis jokiai konstantai  $c$ . Žinoma, mes galime naudoti daugybą  $c$ -augančiame algoritme, bet neturime jos traktuoti kaip primitivios operacijos.

*Lema. Jei c-augančio A algoritmo vykdymo laikas blogiausiu atveju yra  $t(N)$  atžvilgiu pradinių duomenų elementų skaičiaus  $N$ , tai algoritmo A vykdymo laikas yra  $O(n^2 t(n))$  atžvilgiu pradinių duomenų neelementariam kodavimui reikalingų bitų skaičiaus  $n$ .*

Kadangi kiekvienas „tinkamas“ algoritmas, kurio veikimo laikas yra polinominis pradinių duomenų elementų atžvilgiu, turės polinominį veikimo laiką pradinių duomenų bitų skaičiaus atžvilgiu, tai galima grįžti prie pradinių duomenų dydžio vertinimo elementų skaičiumi.

## P ir NP sudėtingumo klasių apibrėžimas

Remdamiesi lema, žinome, kad nagrinėti polinominio sudėtingumo elementų atžvilgiu algoritmai yra polinominio sudėtingumo algoritmai ir bitų skaičiaus atžvilgiu. Be to, polinomų klasė yra uždara sudėties, daugybos ir kompozicijos atžvilgiu: jei  $p(n)$  ir  $q(n)$  yra polinamai, tai  $p(n)+q(n)$ ,  $p(n)*q(n)$  ir  $p(q(n))$  irgi yra polinamai. Todėl kombinuodami polinominio sudėtingumo algoritmus, galime gauti naujus polinominio sudėtingumo algoritmus.

### Sprendimo priėmimo (klasifikavimo) uždaviniai

Tolimesnio nagrinėjimo supaprastinimui apsiribokime *sprendimo priėmimo (klasifikavimo) uždaviniais*, t.y. uždaviniais, kurių atsakymas yra „taip“ arba „ne“. Kitais žodžiais, šių uždavinių rezultatas yra 1 bitas su reikšme 0 arba 1. Tokių uždavinių pavyzdžiai:

- Ar fragmentas  $P$  yra simbolių eilutėje  $T$ ?
- Ar dviejų aibių  $S$  ir  $T$  sankirta yra tuščia?

- Ar duotame grafe su svoriais  $G$  iš viršūnės  $A$  į viršūnę  $B$  egzistuoja kelias, kurio svoris nedidesnis nei  $k$ ?

Paskutinis pavyzdys rodo, kad dažnai *optimizavimo uždavinį* galima reformuluoti į sprendimo priėmimo (klasifikavimo) uždavinį: įvesdami parametą  $k$  ir klausdami, ar optimali reikšmė yra nedidesnė/nemažesnė už  $k$ . Pastebėjome, kad jei parodysime, jog sprendimo priėmimo (klasifikavimo) uždavinys yra „sudėtingas“, tai reikš, kad atitinkamas optimizavimo uždavinys irgi yra „sudėtingas“.

### *Uždaviniai ir kalbos*

Sakysime, kad algoritmas  $A$  *pripažįsta* (*accept*) duomenų eilutę  $x$ , jei jo rezultatas su pradiniais duomenimis  $x$  yra „taip“. Taigi į sprendimo priėmimo (klasifikavimo) uždavinį galime žiūrėti tiesiog kaip į visų tokių duomenų eilučių aibę  $L$ . Iš tikrųjų, raide  $L$  pažymėjome sprendimo priėmimo (klasifikavimo) uždavinį todėl, kad eilučių aibė dažnai vadinama *kalba* (*language*). Galime sakyti, kad algoritmas  $A$  *pripažįsta* kalbą  $L$ , jei kiekvienam  $x \in L$  algoritmas  $A$  duoda atsakymą „taip“ ir duoda atsakymą „ne“ kitais atvejais. Laikysime, kad su neteisingais pradiniais duomenimis algoritmas  $A$  duoda atsakymą „ne“. (Pastaba: galima papildomai numatyti, kad algoritmas  $A$  kai kuriems pradiniais duomenims „užsiciklins“ ir niekada neduos atsakymo, bet mes apsiribosime nagrinėjimu algoritmų, kurie baigia darbą po baigtinio žingsnių skaičiaus.)

### *Sudėtingumo klasė $P$*

*Sudėtingumo klasė  $P$*  (polinominė) yra aibė visų sprendimo priėmimo (klasifikavimo) uždavinių arba kalbų  $L$ , kurių pripažinimui blogiausiu atveju reikia polinominio vykdymo laiko. T.y. egzistuoja algoritmas  $A$  toks, kad jei  $x \in L$ , tai  $A$  duoda rezultatą „taip“ per laiką  $p(n)$ , kur  $n$  yra  $x$  dydis, o  $p(n)$  yra polinomas.

Pastebėjome, kad  $P$  apibrėžimas nieko nesako apie vykdymo laiką duomenims, kuriems algoritmas duoda rezultatą „ne“. Tokios simbolių eilutės sudaro kalbos  $L$  *papildinį* (*complement*) – aibę visų dvejetainių eilučių, nepriklausančių  $L$ . Pastebėjome, kad jei turime polinominį algoritmą  $A$ , pripažįstantį kalbą  $L$ , tai nesunkiai galime sukonstruoti polinominį algoritmą  $B$ , pripažįstantį  $L$  papildinį: pradiniais duomenimis  $x$  algoritmas  $B$  vykdo algoritmą  $A$   $p(n)$  žingsnių, kur  $n$  yra  $x$  dydis, jei algoritmas  $A$  duoda atsakymą „taip“, algoritmo  $B$  atsakymas „ne“, jei algoritmas  $A$  duoda atsakymą „ne“ arba neduoda jokio atsakymo, algoritmo  $B$  atsakymas „taip“. Taigi, jei kalba  $L$ , atitinkanti kažkokį sprendimo priėmimo uždavinį, priklauso  $P$ , tai ir  $L$  *papildinys* priklauso  $P$ .

### *Sudėtingumo klasė $NP$*

*Sudėtingumo klasė  $NP$*  (nedeterministiškai polinominė) apima sudėtingumo klasę  $P$ , bet papildomai įtraukia ir kalbas, galinčias nepriklausyti  $P$ . Konkrečiau  $NP$  uždavinių algoritams leidžiama atlikti papildomą operaciją:

- *pasirinkti*( $b$ ): ši operacija nedeterminuotu būdu pasirenka bitą (reikšmę 0 arba 1) ir priskiria ją  $b$ .

Kai algoritmas  $A$  naudoja primityvią operaciją *pasirinkti*,  $A$  vadinamas *nedeterministiniu algoritmu*. Sakysime, kad algoritmas  $A$  *nedeterministiškai pripažįsta* duomenų eilutę  $x$ , jei egzistuoja *pasirinkti* kreipinių, kuriuos  $A$  gali atlikti su pradiniais duomenimis  $x$ , rezultatų aibė tokia, kad  $A$  duos rezultatą „taip“. Kitais žodžiais, jei nagrinėsime visus galimus *pasirinkti* kreipinių rezultatus ir pasirinksime tik tuos, pagal kuriuos yra pripažįstama, jei tokių egzistuoja. Pastebėjome, kad tai nėra tas pats kaip atsitiktinis pasirinkimas. Galime įsivaizduoti, kad turime *orakulą*, kuris mums nežinomu (nedeterminuotu) būdu atsako į klausimą.

*Sudėtingumo klasė  $NP$*  yra aibė sprendimo priėmimo uždavinių arba kalbų  $L$ , kurios gali būti *nedeterministiškai pripažįstamos* per polinominį laiką. T.y. egzistuoja nedeterministinis algoritmas  $A$  toks, kad jei  $x \in L$ , tai algoritme  $A$  yra *pasirinkti* kreipinių rezultatų aibė, kad  $A$  duoda atsakymą „taip“ per laiką  $p(n)$ , kur  $n$  yra  $x$  dydis, o  $p(n)$  yra polinomas. Kitais žodžiais sakant, jei polinominį

skaičių kartų kreipsimės į *orakulą* ir gausime tinkamus atsakymus, tai algoritmas A duos atsakymą „taip“ per polinominį laiką.

Pastebėjime, kad **NP** apibrėžimas nieko nesako apie vykdymo laiką duomenims, kuriems algoritmas duoda rezultatą „ne“. Iš tikrųjų, rezultato „ne“ atveju algoritmas A gali atlikti gerokai daugiau nei  $p(n)$  žingsnių. Dar daugiau, kadangi nedeterministinis pripažinimas gali turėti polinominį skaičių *pasirinkti* kreipinių, tai, jei kalba  $L$  priklauso **NP**,  $L$  *papildinys* nebūtinai priklauso **NP**. Iš tikrųjų, yra sudėtingumo klasė, vadinama **co-NP**, sudaryta iš visų kalbų, kurių papildiniai priklauso klasei **NP**, ir daugelis mokslininkų tiki, kad **co-NP**  $\neq$  **NP** (bet formaliai tai nėra įrodyta).

### Alternatyvus sudėtingumo klasės NP apibrėžimas

Yra kitas **NP** sudėtingumo klasės apibrėžimo būdas, besiremiantis deterministiniu *patikrinimu* (*verification*) vietoje nedeterministinio pripažinimo. Sakoma, kad kalba  $L$  gali būti *patikrinta* algoritmu A, jei bet kokiai duomenų eilutei  $x \in L$  yra kita duomenų eilutė  $y$  tokia, kad pradiniais duomenimis  $z = x + y$  algoritmas A duos atsakymą „taip“. Eilutė  $y$  vadinama buvimo  $L$  *sertifikatu*, nes ji padeda patikrinti (sertifikuoti), kad  $x$  tikrai priklauso  $L$ .

Naudojantis patikrinimo sąvoka galima suformuluoti alternatyvų klasės **NP** apibrėžimą. *Sudėtingumo klasė NP* yra aibė  $L$  apibrėžiančių sprendimo priėmimo uždavinius kalbų, kurios gali būti *patikrintos* per polinominį laiką. T.y. egzistuoja (deterministinis) algoritmas A toks, kad kiekvienam  $x \in L$ , naudodamas kažkokį sertifikatą  $y$ , jis patikrina, kad iš tikrųjų  $x \in L$  per polinominį laiką  $p(n)$ , įskaitant laiką, sugaištamą  $z = x + y$  nuskaitymui, kur  $n$  yra  $x$  dydis. Kitais žodžiais, sprendinio tinkamumo patikrinimui užtenka polinominio laiko.

Galima įrodyti abiejų apibrėžimų ekvivalentumą.

*Teorema. Kalba L gali būti (deterministiškai) patikrinta per polinominį laiką tada ir tik tada, kai L gali būti nedeterministiškai pripažinta per polinominį laiką.*

Įrodymo idėja pakankamai paprasta: jei turime tinkamą sertifikatą, tai egzistuoja atitinkama kreipinių į orakulą aibė, duodanti to sertifikato reikšmes; jei egzistuoja tinkamų kreipinių į orakulą rezultatų aibė, tai ją galima naudoti kaip sertifikatą.

### $P = NP$ problema

Nėra formaliai įrodyta, ar  $P = NP$ , ar ne. Netgi nėra žinoma, ar  $P = NP \cap \text{co-NP}$ , ar ne. Yra manoma, kad  $P$  skiriasi tiek nuo **NP**, tiek nuo **co-NP**, tiek nuo jų sankirtos. Iš tikrųjų, toliau aptariami **NP** uždaviniai, kurie manoma nepriklauso  $P$ , t.y. nėra žinomas polinominis jų sprendimo algoritmas.

### NP uždavinių pavyzdžiai

Hamiltono ciklo uždavinys yra nustatyti, ar duotame grafe  $G$  egzistuoja paprastas ciklas, apeinantis visas viršūnes tik vieną kartą ir grįžtantis į pradinę viršūnę. Toks ciklas vadinamas grafo  $G$  *Hamiltono ciklu*.

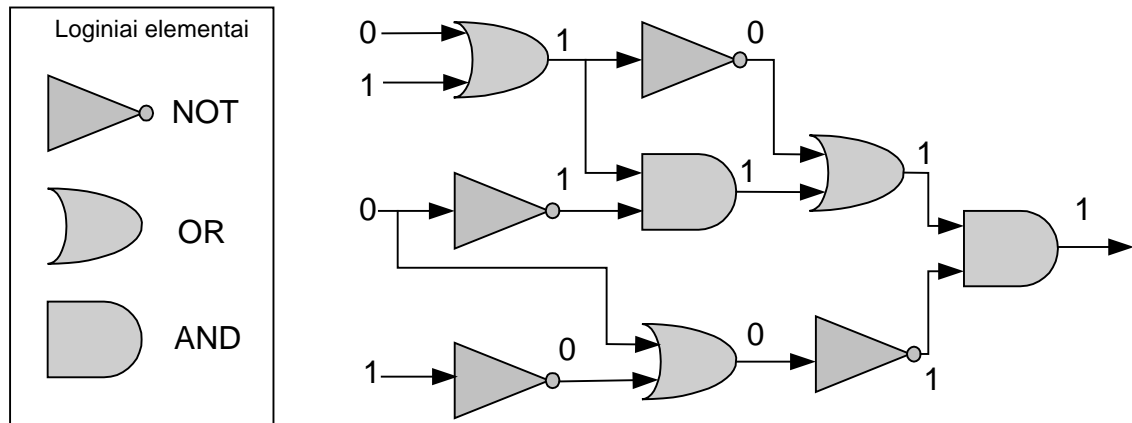
*Lema. Hamiltono ciklo uždavinys yra NP.*

*Įrodymas:* Sunumeruokime grafo  $G$  viršūnes nuo 1 iki  $N$ . Algoritmas A pirmiausia kreipsis į *pasirinkti*, kad sugeneruotų seką  $S$ , sudarytą iš  $N+1$  skaičiaus iš intervalo  $[1, N]$  (tam pakanka polinominio laiko). Dabar algoritmas A patikrins, ar seka  $S$  atitinka grafo  $G$  Hamiltono ciklą:

- 1) 1-a ir paskutinė  $S$  reikšmės turi sutapti
- 2) visos  $G$  viršūnės, skaičiai nuo 1 iki  $N$ , turi būti sekoje  $S$
- 3) visoms poroms  $(s_i, s_{i+1})$  iš  $S$  turi būti briauna grafe  $G$

Visus šiuos patikrinimus galima atlikti per polinominį laiką.

*Loginė grandinė* yra orientuotas grafas, kurio kiekviena viršūnė yra *loginis elementas*, atitinkantis paprastą loginę funkciją AND, OR, NOT. Įeinančios briaunos atitinka pradinis duomenis, o išeinančios – rezultatus.



Loginės grandinės pavyzdys

*Loginės grandinės tenkinimo* uždavinys duotai loginei grandinei su vienu rezultatu turi nustatyti, ar egzistuoja pradiniai duomenys, su kuriais rezultatas yra 1. Tokie pradiniai duomenys vadinami *tenkinančiais*.

*Lema. Loginės grandinės tenkinimo uždavinys yra NP.*

*Įrodymas:* Algoritmas A pirmiausia kreipsis į *pasirinkti*, kad sugeneruotų pradinis duomenis ir kiekvieno loginio elemento rezultatą. Dabar algoritmas A kiekvienam loginiam elementui patikrina, ar sugeneruotas rezultatas yra teisingas pagal sugeneruotus pradinis duomenis. Šį patikrinimą galima atlikti polinominiu laiku. Jei visų loginių elementų rezultatai atitinka jų pradinis duomenis ir visos grandinės rezultatas yra 1, tai algoritmas duoda atsakymą „taip“ (sugeneruoti grandinės pradinis duomenys yra tenkinantys).

Panagrinėkime optimizavimo uždavinio pavyzdį ir parodykime, kad jis yra *NP*.

Duotam grafiui  $G$  *viršūnių denginys* (*vertex cover*) yra poaibis jo viršūnių  $C$  toks, kad kiekvienai grafo  $G$  briaunai  $(v_i, v_j)$   $v_i \in C$  arba  $v_j \in C$  (galbūt abi). Optimizavimo tikslas surasti, kiek galima mažesnį viršūnių denginį.

*Viršūnių denginio* sprendimo priėmimo uždavinys duotam grafiui  $G$  ir sveikam skaičiui  $k$  turi atsakyti, ar egzistuoja viršūnių denginys, sudarytas iš ne daugiau kaip  $k$  viršūnių.

*Lema. Viršūnių denginio uždavinys yra NP.*

## NP-pilnumas

Nedeterministinio pripažinimo sąvoka yra pakankamai keista. Kompiuteris negali efektyviai įvykdyti nedeterministinio algoritmo su dideliu *pasirinkti* kreipinių skaičiumi. Žinoma, galima sumodeliuoti nedeterministinį algoritmą išbandant visus variantus, kuriuos gali duoti jame esantys *pasirinkti* kreipiniai. Bet šis modeliavimas reikalaus eksponentinio laiko bet kuriam nedeterministiniam algoritmui, turinčiam  $n^\epsilon$  *pasirinkti* kreipinių, bet kokiai konstantai  $\epsilon > 0$ . Iš tikrųjų, yra šimtai *NP* sudėtingumo klasės uždavinių, kuriems polinominis algoritmas yra nežinomas ir tikima, kad toks neegzistuoja.

### Polinominis redukavimas

Sakoma, kad kalba  $L$ , apibrėžianti sprendimo priėmimo problemą, yra *polinomiškai redukuojama* į kalbą  $M$ , jei egzistuoja polinominio sudėtingumo funkcija  $f$ , kuri  $L$  pradinis duomenis  $x$  transformuoja į  $M$  pradinis duomenis  $f(x)$  taip, kad  $x \in L$  tada ir tik tada, kai  $f(x) \in M$ . Galima žymėti  $L \rightarrow^{\text{poly}} M$ .

$M$  yra **NP-sunki** (*NP-hard*), jei kiekvienai  $L \in \mathbf{NP}$ ,  $L \rightarrow^{\text{poly}} M$ . Jei be to pati  $M \in \mathbf{NP}$ , tai  $M$  yra **NP-pilna**.

Jei kas nors parodytų, kad *NP-pilnai* problemai egzistuoja polinominis algoritmas, tai automatiškai reikštų kad visa **NP** klasė išsprendžiama per polinominį laiką, t.y.  $\mathbf{P} = \mathbf{NP}$ .

#### *Cook-Levin teorema*

Gali pasirodyti, kad *NP-pilnumo* apibrėžimas yra per daug griežtas, bet pateikiama teorema parodo, kad egzistuoja bent jau viena *NP-pilna* problema.

*Teorema (Cook-Levin teorema). Loginės grandinės tenkinimo uždavinys yra NP-pilnas.*

#### *Kitų NP-pilnų uždavinių pavyzdžiai*

Yra įrodyta, kad anksčiau nagrinėti grafo viršūnių denginio ir Hamiltono ciklo uždaviniai yra *NP-pilni*.

Duotam grafui  $G$  uždara grupė (*clique*) yra poaibis jo viršūnių  $C$  toks, kad kiekvienai porai  $v_i \in C$ ,  $v_j \in C$ ,  $v_i \neq v_j$  egzistuoja grafo  $G$  briauna  $(v_i, v_j)$ , t.y. kiekviena skirtingų viršūnių iš  $C$  pora yra sujungta briauna. Optimizavimo tikslas surasti, kiek galima didesnę uždara grupę.

Įrodyta, kad uždaros grupės uždavinys yra *NP-pilnas*.

*Kuprinės* uždavinys: duota aibė  $S$  daiktų, sunumeruotų nuo 1 iki  $n$ . Kiekvienas daiktas  $i$  turi dydį (svorį)  $s_i$  ir kainą  $k_i$ ; duotai kuprinės talpai  $T$  reikia rasti aibę  $i$  ją telpančių daiktų su bendra didžiausia kaina, t.y. aibę  $S' \subset S$  tokią, kad

$$\text{SUM}(s_i, i \in S') \leq T \text{ ir } \text{SUM}(k_i, i \in S') \text{ maksimali.}$$

Įrodyta, kad *kuprinės* uždavinys yra *NP-pilnas*.

Sakoma, kad daugiau kaip ketvirtadalį realių skaičiavimų laiko procesorius atlieka duomenų paiešką ir rūšiavimą. Todėl svarbu kokius algoritmus naudojame šiems uždaviniams spręsti, kadangi vieni algoritmai yra greitesni, kiti - lėtesni. Mes panagrinėsime keletą iš gerai žinomų paieškos ir rūšiavimo algoritmų ir paanalizuosime, kaip jų vykdymo laikas priklauso nuo duomenų skaičiaus ir tvarkos.

Tarkime, kad mes ieškome duotos reikšmės įrašų apie knygas masyve. Vieną kartą mes galime ieškoti tam tikro autoriaus, kitą kartą - pavadinimo. Bet kuriuo atveju laukiai, kuriuose mes ieškome, vadinami masyvo **paieškos raktais** (*search keys*).

**Raktai** (*keys*): *Laukai, kuriais operuojame dirbdami su duomenų rinkiniu.*

Gali būti, kad masyve nėra ieškomo rakto. Taip pat gali būti, kad raktai masyve kartojasi. Šioje situacijoje paieškos algoritmas gali būti skirtas rasti *bet kurį vieną* raktą, *pirmą* raktą, *paskutinį* raktą ar *visus* raktus. Mes nagrinėsime paiešką bet kurio vieno rakto.

**Rūšiavimas** yra procesas raktų sąrašo sutvarkymo norima tvarka. Yra 4 tvarkos: **didėjimo** (pritaikoma tik tada, kai visi raktai yra skirtingi), **nemažėjimo**, **nedidėjimo** ir **mažėjimo** (pritaikoma tik tada, kai visi raktai yra skirtingi). Mes nagrinėsime rūšiavimą nemažėjimo tvarka.

Kaip ir paieškos atveju, pats natūraliausias (akivaizdžiausias) rūšiavimo algoritmas yra retai geriausias. Buvo padėta labai daug pastangų efektyvių rūšiavimo algoritmų suradimui ir kai kurie standartiniai rūšiavimo algoritmai tikrai puikūs. Be rūšiavimo greičio kartais mums reikia tokių rūšiavimo algoritmų, kurie galėtų rūšiuoti masyvus, netelpančius operatyvioje atmintyje. Šiuo atveju mums reikia **išorinio rūšiavimo algoritmų**, kurie rūšiuoja išorinį failą, kurdami naują surūšiuotą išorinį failą, nenusisaktydami viso failo vienu metu į atmintį. Šie algoritmai pakankamai sudėtingi, ir mes daugiau dėmesio skirsime **vidinio rūšiavimo algoritmams**, kurie rūšiuoja masyvą operatyvioje atmintyje. Pamatysime, kad geriausi algoritmai *atsitiktinių* duomenų rūšiavimui nėra geriausi *beveik surūšiuotų* duomenų rūšiavimui.

### Nuosekli paieška

Pats akivaizdžiausias ir primityviausias reikiamo rakto paieškos masyve būdas yra pradėti nuo masyvo pradžios ir peržiūrėti iš eilės kiekvieną elementą. Toks metodas vadinamas **nuoseklia** arba **tiesine paieška** (*sequential or linear search*).

**Nuoseklios paieškos algoritmas:**

**Tikslas:** Rasti pirmą masyvo elementą, turintį ieškomą raktą *SearchKey* duotoje masyvo dalyje, pradedant *Key[First]* ir baigiant *Key[Last]* arba trumpiau *Key[First..Last]*.

**Pradinės sąlygos** (*preconditions*): *Key[First..Last]* yra inicializuotas.

**Galinės sąlygos** (*postconditions*): *Grąžina, ar ieškomas elementas rastas, ir, jei rastas, indeksą masyvo elemento, turinčio ieškomą raktą. Jei ieškomas raktas nerastas, grąžina poziciją Last+1.*

**Žingsniai** (*steps*):

1. Nustatyti loginio kintamojo *Rasta* reikšmę *False* ir inicializuoti *Pozicija* - *First*.

2. Kol *Pozicija* ne daugiau už *Last* ir *Rasta* yra *False* daryti

*Jei Key[Pozicija] lygus ieškomam raktui,*

*tada (darbą baigėme) įsiminti Pozicija ir nustatyti Rasta reikšmę True*

*priešingu atveju padidinti Pozicija.*

3. Grąžinti reikšmes *Rasta* ir *Pozicija*.

Nuoseklios paieškos algoritmas baigia darbą, kai tik randa ieškomą raktą masyve. Jei mums labai pasiseks, jau pirmas patikrintas raktas bus lygus ieškomam, šiuo atveju algoritmas baigs darbą, atlikęs tik vieną **rakto palyginimo operaciją** (*key comparison*). Tai geriausias atvejis.

Blogiausiu atveju algoritmas gali peržiūrėti visą masyvą, prieš raskdamas ieškomą raktą paskutiniame elemente arba iš viso neraskdamas. Bet kuriuo atveju bus atlikta tiek rakto palyginimo operacijų, kiek elementų yra masyve.

Bendru atveju vykdymo laikas bus tarp šių dviejų ekstremalių reikšmių. Jei visos galimos reikšmės vienodai pasiskirsčiusios masyve, tai vidutinė paieška apims pusę masyvo elementų, t.y. vidutinis nuoseklios paieškos vykdymo laikas bus proporcingas  $N/2$ , kur  $N$  yra raktų (elementų) skaičius masyvo paieškos dalyje.

Galima padidinti nuoseklios paieškos algoritmo efektyvumą, jeigu mes turime papildomos informacijos apie duomenis. Nuoseklios paieškos algoritmas galėtų veikti efektyviau, jei masyvas

surūšiuotas: galima neperžiūrėti visų elementų, o baigti darbą, vos radus elementą didesnę už ieškomą; žinant pirmo ir paskutinio elemento reikšmes galbūt galima pradėti ieškoti ne nuo pradžios bet nuo galo. Algoritmą galima patobulinti, kad jei ieškoma reikšmė mažesnė už pirmą arba didesnę už paskutinę, tai galima neieškoti. Galimas ir kitas rūšiavimo principas.

**Tikimybiniis rūšiavimas.** Paprastai ne visų elementų ieškoma vienodai dažnai, todėl būtų prasminga dažniau ieškomus elementus laikyti masyvo pradžioje. Yra trys būdai tai realizuoti:

*pirmas:* surūšiuoti elementus pagal jų paieškos dažnumą nedidėjimo tvarka; tam reikia iš anksto žinoti elementų paieškos dažnumą ir jis turi nesikeisti;

*antras:* kiekvieną ieškomą elementą suradus padėti į masyvo pradžią; yra keletas problemų susijusių su šiuo metodu: pirma, nėra garantijos, kad padėto į priekį masyvo elemento bus dažnai ieškoma (žinoma, vėliau tas elementas slinks vis toliau), antra, jei realizacijai naudojami masyvai (duomenų struktūra), tai perkelti elementą į pradžią, reikalauja daug veiksmų, nes reikia perstumti likusius elementus;

*trečia:* kiekvieną ieškomą elementą suradus sukeisti vietomis su einančiu prieš jį, po daug paieškos operacijų dažniausiai ieškomi elementai atsidurs masyvo pradžioje.

Tikimybiniis rūšiavimas remiasi prielaida, kad kai kurių elementų ieškoma žymiai dažniau negu likusių, o tai ne visada teisinga, todėl dažniau naudojamas rūšiavimas pagal raktų reikšmes, o šiuo atveju galima pritaikyti kitus efektyvesnius paieškos metodus.

### Dvejtainė paieška

Tarkime, kad ieškome Petro Jonaičio telefono Vilniaus telefonų knygoje. Žinoma, kad mes nenaudosime nuoseklios paieškos, nes tai užims per daug laiko ir telefonų knygoje *pavardės pateiktos surūšiuotos*, todėl galima pritaikyti efektyvesnius metodus. Vietoj nuoseklios paieškos jūs atversite puslapį, kuriame galima tikėtis, kad bus “Jonaitis” (maždaug telefonų knygos viduryje). Tada nustatysite, ar “Jonaitis” yra prieš atverstą puslapį, ar toliau. Pagal tai, ką jūs matysite atverstame puslapyje, jūs praignoruosite knygos dalį ir pakartosite tą patį procesą su ta dalimi, kurioje yra ieškoma pavardė. Ši dalis vadinama **paieškos sritimi** (*search area*). Kiekvieną kartą, kai jūs kartojate procesą, knygos paieškos sritis mažėja, kol randamas reikiamas puslapis ir tada reikiama vieta jame. Iš esmės procesas atlieka paieškos srities dalinimą, kol randamas “Jonaitis” arba nustatoma, kad jo knygoje nėra.

**Dvejtainė paieška** yra paieškos sutvarkytame sąraše procesas, labai panašus į pavyzdį paieškos telefonų knygoje. Jis dalina sutvarkytą sąrašą tol, kol atsiduria vietoje, kurioje yra ieškomas elementas, jei jis iš viso sąraše yra.

**Sutvarkytas sąrašas** (*ordered list*): *Surūšiuotas sąrašas, t.y. sąrašas, kuriame raktai sutvarkyti (surūšiuoti) tam tikra tvarka.*

Pavyzdžiui, mes ieškome skaičiaus 17 (bet kurio) sutvarkytame masyve

Key[1], ... Key[10]: 4, 6, 7, 10, 13, 17, 21, 28, 35, 40.

“Viduriniausias” masyvo elementas yra su indeksu  $(1+10) \div 2$  arba 5. Mes lyginame Key[5]=13 su ieškumu raktu 17, kadangi Key[5] yra mažiau už 17, tai tolimesnėje paieškoje mes galime ignoruoti elementus nuo 1 iki 5. Dabar paieškos sritis gali būti sumažinta iki

Key[6], ... Key[10]: 17, 21, 28, 35, 40.

Kartojame procesą likusiai masyvo daliai. Vidurinis elementas yra su indeksu  $(6+10) \div 2$  arba 8. Lyginame Key[8]=28 su 17 ir kadangi jis yra didesnis, tai ignoruojame elementus nuo 8 iki 10. Dabar mūsų paieškos sritis susideda tik iš Key[6], Key[7]: 17, 21. Vidurinis elementas yra su indeksu  $(6+7) \div 2$  arba 6. Lyginame Key[6]=17 su 17 ir randame, kad tai ieškomas elementas. Gaunamas rezultatas 6 ir algoritmas baigia darbą.

**Dvejtainės paieškos algoritmas:**

**Tikslas:** Rasti bet kurį masyvo elementą, turintį ieškomą raktą SearchKey duotoje masyvo dalyje, pradedant Key[First] ir baigiant Key[Last] arba trumpiau Key[First..Last].

**Pradinės sąlygos** (*preconditions*): Key[First] <= Key[First+1] <= ... <= Key[Last]

**Galinės sąlygos** (*postconditions*): Grąžina, ar ieškomas elementas rastas, ir, jei rastas, indeksą masyvo elemento, turinčio ieškomą raktą. Jei ieškomas raktas nerastas, Pozicija neapibrėžta.

**Žingsniai** (*steps*):

1. Jei First <= Last atlikti:

a. Suskaičiuoti vidurinį paieškos srities indeksą:

Location = (First + Last) div 2

b. Jei SearchKey=Key[Location], tai paieška baigta, grąžinti, kad ieškomas raktas rastas pozicijoje Location ir baigti darbą.

Priešingu atveju, jei Key[Location] < SearchKey, todėl SearchKey, jei yra bus tarp elementų Key[Location+1..Last], taigi praignoruosime pirmuosius elementus, perapibrėždami First=Location+1. Ir kartosime žingsnį 1 su nauja First, bet sena Last reikšmė. Tai mažina paieškos sritį.

Priešingu atveju (jei Key[Location] > SearchKey), todėl SearchKey, jei yra bus tarp elementų Key[First..Location-1], taigi praignoruosime likusius elementus, perapibrėždami Last=Location-1. Ir kartosime žingsnį 1 su nauja Last, bet sena First reikšmė. Tai mažina paieškos sritį.

2. Jei First > Last, tai paieškos sritis tuščia ir joje negali būti SearchKey, todėl grąžinti, kad ieškomas raktas nerastas.

**Algoritmo pagerinimas.** Algoritmas yra aiškus ir paprastas, bet jis palieka poziciją (Location) neinicializuotą, jei paieškos sritis yra tuščia, t.y. jei First>Last, kai procedūra pradeda darbą. (Pataisymas elementarus: galima Location priskirti reikšmę First-1 darbo pradžioje arba visais atvejais, kai ieškomas raktas nerastas.) Be to, kai paieškos sritis netuščia, bet ieškomas raktas (SearchKey) nerandamas, procedūra sustoja pozicijoje, kuri yra tik daugmaž ta vieta, kurioje galėtų būti ieškomas raktas. Pavyzdžiui, ieškome masyve

Key[1..8]: 4, 6, 7, 10, 13, 17, 21, 28, 35, 40.

Jei ieškome 8, tai procedūra grąžina Found=False ir Location=3, t.y. elementas Key[3]=7 yra tas, už kurio turėtų būti ieškomas elementas. Jei ieškome 15, tai procedūra grąžina Found=False ir Location=6, t.y. elementas Key[6]=17 yra tas prieš kurį turėtų būti ieškomas elementas. Taigi, kai nerandame Found=False ir Location yra  $\pm 1$  pozicija, kurioje turėtų būti ieškomas elementas. Jei ieškomas raktas nerandamas, norėtusi, kad algoritmas grąžintų poziciją, į kurią reiktų įterpti ieškomą elementą, išlaikant nemažėjimo tvarką.

**Realizacija.** Dvejetainės paieškos algoritmą galima realizuoti tiek iteratyviai, tiek rekursyviai. Rekursyvi realizacija reikalauja daugiau atminties ir veikia lėčiau, bet šis algoritmas yra pakankamai greitas, todėl priimtina ir tokia realizacija.

**Vykdyto laikas.** Dvejetainės paieškos algoritmas kiekvieną kartą dvigubai sumažina paieškos sritį, kol joje lieka tik vienas elementas. Šis elementas yra ieškomas arba ne, bet visais atvejais algoritmas baigia darbą. Vykdyto laikas matuojamas kartojimų skaičiumi, t.y. paieškos srities dalinimo kartų, kol lieka tik 1 elementas, skaičiumi. Jei N yra elementų skaičius, o C yra dalinimų pusiau skaičius tai:  $N / 2^C \geq 1$  arba C tai didžiausias sveikas skaičius, kad  $N \geq 2^C$ , taigi  $C = \text{trunc}(\log_2 N)$ . Blogiausiu atveju dvejetainė paieška bus atliekama  $\text{trunc}(\log_2 N) + 1$  kartą. Kadangi kiekvieną kartą atliekamos dvi rakto palyginimo operacijos, tai šių operacijų skaičius dvigubai didesnis. Geriausiu atveju gali pakakti vienos rakto palyginimo operacijos.

Palyginimas blogiausių paieškos atvejų (kai ieškoma rakto didesnio negu yra masyve):

N = elementų skaičius	Rakto palyginimo operacijų	
	Dvejetainė paieška ( $2 * \text{trunc}(\log_2 N)$ )	Nuosekli paieška (N+1)
100	12	101
1.000	18	1.001
10.000	26	10.001
100.000	32	100.001

Panaudojus "O" notaciją. Geriausiu atveju abiejų algoritmų vykdymo laikas yra 1 arba O(1), o blogiausiu atveju - O(lgN) ir O(N) atitinkamai. (Praktiškai nėra labai svarbus logaritmo pagrindas.)

Taigi nuoseklios paieškos **algoritmo sudėtingumas** yra tiesinis O(N), o dvejetainės paieškos - **logaritminis**. Dar galimi polinominis, eksponentinis sudėtingumai.

Yra ir kitų paieškos algoritmų. Jei turime surūšiuotą masyvą ir jo elementų raktų reikšmės pasiskirstę daugmaž tolygiai (tiesiškai), tai paieškos sritį galima dalinti ne į dvi lygias dalis, bet priklausomai nuo ieškomo rakto reikšmės: pavyzdžiui, ieškome masyve Key[1..100]: 0, ... 999 - reikšmės 5, tai labai tikėtina, kad ji bus tarp pirmųjų elementų.

Dvejetainė paieška yra efektyvesnė, bet ji gali būti pritaikyta tik surūšiuotiems duomenims, o rūšiavimas irgi užima laiko, todėl kiekvienu konkrečiu atveju turėtų būti įvertinta, ar apsimoka iš



pradžių masyvą surūšiuoti (jeigu duomenys retai keičiasi, o reikia daug kartų atlikti paiešką), ar geriau naudoti nuoseklios paieškos algoritmą.

## R ū š i a v i m a s

**Vidinis rūšiavimas:** rūšiavimo procesas, reikalaujantis, kad visi rūšiuojami elementai būtų operatyvioje (pagrindinėje - RAM) atmintyje. Šių metodų privalumas rūšiavimo greitis, o trūkumas didelis atminties poreikis (todėl gali rūšiuoti tik riboto dydžio masyvus).

**Išorinis rūšiavimas:** rūšiavimo procesas, skaitantis dalimis informaciją į operatyvią atmintį iš išorinio failo, rūšiuojantis nuskaitytus elementus ir rašantis surūšiuotus fragmentus į išorinį failą. Šių metodų privalumas nedidelis atminties poreikis (gali rūšiuoti neriboto dydžio masyvus), o pagrindinis trūkumas algoritmų sudėtingumas.

Visus rūšiavimo algoritmus galima suskirstyti į tris klases:

- \* **rūšiavimas paieška** (*selection sort*) pradeda nuo pirmo masyvo elemento, tada randa elementą, kuris turi būti pirmas (mažiausią), ir sukeičia šiuos elementus vietomis; po to pereina prie sekančio masyvo elemento ir atlieka tuos pačius veiksmus likusioje masyvo dalyje tol, kol bus surūšiuotas visas masyvas;
- \* **rūšiavimas įterpimu** (*insertion sort*) dalina rūšiuojamą masyvą į dvi dalis: pirmoji - "surūšiuota", o antroji - "nesurūšiuota"; iš pradžių surūšiuota dalis sudaryta tik iš vieno elemento; palaipsniui surūšiuota dalis auginama, imant po vieną elementą iš nesurūšiuotos dalies ir įterpian jį į reikiamą vietą, tol, kol surūšiuota dalis yra visas masyvas;
- \* **rūšiavimas keitimu** (*exchange sort*) lygina tam tikrus elementus, priklausomai nuo konkretaus rūšiavimo metodo, ir, jeigu reikia, keičia juos vietomis tol, kol surūšiuojamas visas masyvas.

### Rūšiavimas paieška (*selection sort*)

Vienas iš būdų surūšiuoti masyvą nemažėjimo tvarka yra toks: mažiausią reikšmę įrašyti į pirmą vietą, antrą pagal dydį į antrą ir t.t. Tai rūšiavimo paieška metodas (*selection sort*), kartais dar vadinamas mažiausio elemento metodu.

**Rūšiavimo paieška algoritmas:**

**Tikslas:**

Surūšiuoti masyvą  $Key[First..Last]$  nedidėjimo tvarka.

**Pradinės sąlygos** (*preconditions*):

First, Last ir  $Key[First..Last]$  yra inicializuoti.

**Galinės sąlygos** (*postconditions*):

$Key[First] \leq Key[First+1] \leq \dots \leq Key[Last]$

**Žingsniai** (*steps*):

1. Priskirti First reikšmę FirstUnsorted.

2. Kol  $FirstUnsorted < Last$ , atlikti:

a. Rasti mažiausią reikšmę tarp elementų  $Key[FirstUnsorted..Last]$

b. Sukeisti mažiausią reikšmę su elementu  $Key[FirstUnsorted]$ .

c. Padidinti FirstUnsorted.

(Žingsnio 2a realizavimui irgi reikia ciklo.)

Rūšiavimo paieška algoritmas kiekvienos iteracijos pabaigoje keičia du elementus vietomis, taigi iš viso bus atlikta  $(N-1)$  keitimo operacijų. Galima tikrinti, ar nebandoma keisti elementą vietomis su juo pačiu, tada keitimų skaičius sumažėtų, priklausomai nuo pradinio masyvo surūšiavimo, bet kiekvieną kartą būtų atliekamas indeksų palyginimas (konkretus sprendimas priklauso nuo to, kiek laiko užima keitimas vietomis). Bet raktų palyginimo operacijų visada atliekama vienodai: pirmą kartą  $N-1$ , antrą kartą  $N-2$  ir t.t. iš viso

$$(N-1) + (N-2) + \dots + 1 = N*(N-1) / 2.$$

Šiam algoritmui neegzistuoja geriausias ir blogiausias atvejai.

Taigi rūšiavimo paieška algoritmas yra tiesinio sudėtingumo  $O(N)$  keitimų atžvilgiu ir kvadratinio sudėtingumo  $O(N^2)$  raktų palyginimo operacijų atžvilgiu, todėl kai masyvas yra didelis raktų palyginimo operacijos dominuoja ir algoritmo sudėtingumas yra kvadratinis.

*Apibendrinimas.* Rūšiavimo paieška algoritmas yra paprastas ir intuityvus, lengvai programuojamas, todėl dažnai naudojamas. Bet yra trys priežastys, kodėl šis metodas nenaudojamas komercinėse aplikacijose:

1. Metodo sudėtingumas yra kvadratinis.
2. Tai vidinio rūšiavimo metodas, komerciniuose uždaviniuose (duomenų bazėse) dažnai tenka rūšiuoti didelius informacijos kiekius.
3. Daug komercinių aplikacijų naudoja duomenų bazines, kuriose surūšiuoti duomenų sąrašai yra modifikuojami ir tada perrūšiuojami, tradiciškai tenka rūšiuoti beveik surūšiuotus masyvus ir rūšiavimo algoritmas turėtų į tai atsižvelgti, o šis metodas to nedaro.

#### Rūšiavimas įterpimu (*insertion sort*)

Rūšiavimas įterpimu (*insertion sort*) yra natūralus rūšiavimo metodas naudojamas daugelio kortų žaidėjų. Masyvas yra dalinamas į dvi dalis: kairiąją surūšiuotą (pradžioje ją sudaro pirmas elementas) ir dešiniąją nesurūšiuotą. Pagrindinė idėja yra paimti kairiausią elementą iš nesurūšiuotos dalies ir įterpti surūšiuotoje dalyje į reikiamą vietą.

##### Rūšiavimo įterpimu algoritmas (pseudokodas)

**FOR** elementams nuo antro iki paskutinio atlikti **DO**

Laikina išsaugoti pirmojo iš nesurūšiuotos dalies elemento (*ciklo indeksas*) reikšmę

(Nustatyti, ar išsaugotą reikšmę reiks palikti savo vietoje, ar įterpti į surūšiuotą dalį:)

Nustatyti Indeksas=paskutiniam surūšiuotos dalies elementui

**WHILE** elementas pozicijoje Indeksas yra didesnis už išsaugotą **DO**

Perstumti elementą, į kurį rodo Indeksas, per vieną poziciją į dešinę

Sumažinti Indeksas vienetu

Jei Indeksas yra 0, nutraukti ciklą

Įterpti išsaugotą reikšmę pozicijoje Indeksas+1

Pastebėjime, kad rūšiavimas geriausiu atveju (kai masyvas yra surūšiuotas) kiekvienam elementui atlieka tik po vieną rakto palyginimo operaciją, blogiausiu atveju (kai masyvas yra surūšiuotas atvirkščia tvarka) I-tajam elementui atlieka (I-1) palyginimo operaciją, tikėtinu atveju (vidutiniškai) - (I-1) / 2 palyginimo operacijų.

Taigi algoritmo sudėtingumas yra kvadratinis  $O(N^2)$ , o geriausiu atveju tiesinis. Pastebėjime, kad rūšiavimo įterpimu algoritmas atlieka tiek pat elemento kopijavimo operacijų kiek ir rakto palyginimo operacijų.

#### Burbuliuko metodas

Burbuliuko metodas yra vienas iš rūšiavimo keitimu (*exchange sort*) metodų. Kiekvienos iteracijos metu ne tik didžiausias iš nesurūšiuotų elementų padedamas į savo vietą, bet ir atliekamas kitų elementų patvarkymas. Pagrindinė algoritmo idėja yra lyginti gretimus elementus ir, jeigu reikia, keisti juos vietomis.

##### Burbuliuko algoritmas (pseudokodas)

**REPEAT**

**FOR** elementams nuo pirmo iki priešpaskutinio atlikti **DO**

**IF** einamasis ir sekantis elementas yra bloga tvarka

**THEN** sukeisti juos vietomis

**UNTIL** nebuvo atlikta pakeitimų

Kadangi kiekvienos iteracijos metu didžiausias elementas padedamas į savo vietą, tai galima mažinti vidinio ciklo kartojimų skaičių, tai kažkiek pagreitintų veikimą. Išorinį ciklą irgi galima perrašyti su **FOR**.

##### Burbuliuko algoritmas

**FOR** limit := N-1 **DOWNTO** 1 **DO**

**FOR** i := 1 **TO** limit **DO**

**IF** List[i] > List[i+1] **THEN** swap(List[i], List[i+1]);

Antrasis variantas visada (nepriklausomai nuo pradinio masyvo surūšiavimo) atlieka fiksuotą rakto palyginimo operacijų skaičių  $(N-1)+(N-2)+\dots+1=N*(N-1)/2$ . Pirmasis variantas geriausiu atveju gali atlikti tik N-1 rakto palyginimo operaciją, bet pirmajam variantui, jei vidinio ciklo kartojimų skaičius yra fiksuotas, blogiausiu atveju prireikia dar vienos iteracijos tam, kad įsitikinti, jog viskas

tvarkoje. Reikšmių keitimo operacijų skaičius abiem atvejais yra vienodas ir tiesiogiai priklauso nuo pradinio masyvo surūšiavimo.

Taigi algoritmo sudėtingumas yra kvadratinis  $O(N^2)$ , o geriausiu atveju tiesinis. Bendru atveju, rūšiavimo įterpimu algoritmas yra efektyvesnis, nes burbuliuko metodas atlieka gretimų elementų raktų palyginimo operacijas, nors keitimo ir nereikia.

Pastebėjime, kad burbuliuko metodui blogiausias variantas būtų pavyzdžiui toks: visi masyvo elementai, išskyrus paskutinį, yra surūšiuoti, o paskutinis elementas yra mažiausias. Tada kiekvienos iteracijos metu mažiausias elementas pasislinktų tik per vieną poziciją ir prireiktų  $N-1$  iteracijos. Yra atvirkštinis burbuliuko metodas, kuris šią situaciją sutvarkytų vienu praėjimu. Galima konstruoti apibendrintą metodą, kuris nelyginėse iteracijose dirbtų kaip burbuliuko, o lyginėse kaip atvirkštinis metodas. Tačiau bet kurio iš šių metodų sudėtingumas yra kvadratinis  $O(N^2)$ .

Apžvelgtų vidinio rūšiavimo metodų sudėtingumo lentelė:

Rūšiavimas	Blogiausias atvejis	Tikėtinas efektyvumas	Geriausias atvejis
Paieška (selection)	$O(N^2)$	$O(N^2)$	$O(N^2)$
Burbuliuko	$O(N^2)$	$O(N^2)$	$O(N)$
Įterpimu (insertion)	$O(N^2)$	$O(N^2)$	$O(N)$
Greitas (quicksort)	$O(N^2)$	$O(N \log_2 N)$	$O(N \log_2 N)$

Kai kuriose situacijose prasmingas kelių rūšiavimo metodų panaudojimas, priklausomai nuo situacijos.

### Greitas rūšiavimas (*Quicksort*)

Kai masyvai yra labai dideli ir svarbus rūšiavimo greitis, galima naudoti Greito rūšiavimo (*Quicksort*) metodą. Jis yra greičiausias ir dažniausiai naudojamas vidinio rūšiavimo metodas. Kaip ir buvo galima tikėtis, jis yra ir vienas iš sudėtingiausių. Tačiau programos kodo jam realizuoti reikia stebėtinai mažai, palyginus su puikiu algoritmo greičiu. Šis algoritmas buvo pasiūlytas C.A.R. Hoare ir pirmą kartą publikuotas 1962 metais. Tai rūšiavimo keitimu (*exchange sort*) metodas, besiremiantis tuo faktu, kad greičiau ir lengviau surūšiuoti du mažus sąrašus negu vieną didelį. Taigi pagrindinė metodo strategija "skaldyk ir valdyk". Jis pastoviai dalina rūšiuojamą sąrašą į mažesnius ir mažesnius sąrašėlius, kol jų nebereikia dalinti, t.y. kiekvienas iš jų yra surūšiuotas. Greito rūšiavimo metodą lengviausia realizuoti rekursyviai.

Tarkime, kad turime masyvą Key [First..Last]: 13, 28, 40, 6, 35, 4, 10, 21, 7, 17.

Pasirinkime kažkokią reikšmę iš masyvo ir pavadinkime ją dalinimo reikšme arba ašimi (*partitioning value or pivot*). Tarkime, kad mūsų pavyzdyje dalinimo reikšmė yra 13. Tada padalinkime mūsų masyvą taip, kad mažesnės reikšmės eitų prieš ašį, o visos didesnės reikšmės už jos:

7, 6, 4, 10 (Kairioji dalis), 13, 40, 28, 21, 35, 17 (Dešinioji dalis).

Dabar galima sakyti, kad masyvas yra padalintas (vėliau panagrinėsime patį dalinimo algoritmą) taip: kairiosios dalies elementai yra mažesni už ašį (dalinimo reikšmę), dešinėsios dalies elementai yra didesni už ašį, o tarpe yra pati ašis (dalinimo reikšmė). Dalinimo reikšmė dabar yra savo pozicijoje ir jos *daugiau nebereikės judinti*. Tai svarbus faktas. Dabar turime du mažesnius rūšiavimo uždavinius: reikia surūšiuoti kairiąją ir dešiniąją masyvo dalis, kiekvieną savo viduje. Abiejų dalių dydis (mūsų pavyzdyje) yra maždaug pusė nuo pradinio masyvo dydžio, taigi mes sėkmingai suskaidėme pradinį rūšiavimo uždavinį į du dvigubai mažesnius rūšiavimo uždavinius. Imdami tik kairiąją dalį, daliname ją tokiu pačiu būdu kaip ir pradinį masyvą:

4, 6 (Kairioji dalis), 7, 10 (Dešinioji dalis).

Dabar mes padėjome dalinančią reikšmę (tarkime 7) į jos poziciją ir padalinome senąją kairiąją dalį į dvi mažesnes, kiekviena iš kurių bus rūšiuojama dalinant. Tai tęsiama kol mes gausime mažas mažas dalelytes, kurios bus arba tuščios arba turės tik vieną elementą (taigi daugiau nebereikės dalinti). Šiuo momentu visa pradinė kairioji dalis bus surūšiuota. Analogiškai rūšiuojama ir dešinioji dalis. Visas algoritmas yra nesudėtingai apibūdinamas rekursiškai.

**Greito rūšiavimo (*quicksort*) algoritmas:**

**Tikslas:** Surūšiuoti masyvą Key[First..Last] nedidėjimo tvarka.

**Pradinės sąlygos** (preconditions): First, Last ir Key[First..Last] yra inicializuoti.

**Galinės sąlygos** (postconditions): Key[First] <= Key[First+1] <= ... <= Key[Last]

**Žingsniai** (steps): Jei First < Last tada

1. Pasirinkti dalinančią reikšmę Pivot ir pertvarkyti elementus taip, kad:

a. Dalinanti reikšmė Pivot patalpinama į elementą Key[PivotPos], kuri yra jos galutinė pozicija.

b. Visi elementai Key[First..PivotPos-1] yra mažesni už dalinančią reikšmę Pivot.

c. Visi elementai Key[PivotPos+1..Last] yra didesni už dalinančią reikšmę Pivot.

(Kaip atlikti žingsnį 1 bus paaiškinta vėliau.)

2. Rekursyviai atlikti algoritmą kairiajai daliai.

3. Rekursyviai atlikti algoritmą dešiniajai daliai.

(Algoritmas natūraliai aprašomas rekursiškai. Jis gali būti realizuotas ir iteratyviai, bet tai sudėtingiau.)

Jei visą žingsnį 1 atlieka procedūra Partition (visas pagrindinis darbas joje ir yra), tai procedūrą QuickSort parašyti labai paprasta:

```
procedure QuickSort (var Key : ArrayType; First, Last : Integer);
var PivotPos : SubscriptRange;
begin
  if First < Last then begin
    Partition (Key, First, Last, PivotPos);
    QuickSort (Key, First, PivotPos-1);    QuickSort (Key, PivotPos+1, Last)
  end; {if First < Last}
end; {QuickSort}
```

Algoritmas veikia greičiausiai, jei dalindami mes padaliname masyvą į dvi daugmaž lygias dalis, t.y. jei dalinanti reikšmė yra daugmaž vidurinė reikšmė (*median value*): tokia parinkta iš aibės reikšmė, kad aibėje yra daugmaž vienodai didesnių ir mažesnių už ją reikšmių (nereiktų painioti su vidutine reikšme, kuri yra visų reikšmių aritmetinis vidurkis).

Kaip parinkti vidurinę reikšmę? Žinoma, mes galime parašyti algoritmą, kuris peržiūrėdamas visą masyvą nustato vidurinę reikšmę skaičiuodamas, bet tai reikalauja nemažai skaičiavimų. Galimas kitas kraštutinis: tiesiog atsitiktinai pasirinkti bet kurią reikšmę kaip vidurinę, jei masyvo elementai išsidėstę visai atsitiktinai vienodai tikėtina, kad bet kuris elementas bus vidurine reikšme. Toks paprastas metodas veikia pakankamai gerai, išskyrus atvejus, kai masyvas yra beveik surūšiuotas arba surūšiuotas atvirkščiai. Galimas ir kitas nesudėtingas būdas: atsitiktinai pasirinkti tris reikšmes ir tada pasirinkti vidurinę iš jų. Šis metodas beveik visada duoda gerus rezultatus, be to, jį paprasta realizuoti.

Sudėtinga greito rūšiavimo dalis yra dalinimo algoritmas, realizuojantis žingsnį 1. Buvo pasiūlyta daug dalinimo algoritmų, bet galbūt vienas iš lengviausiai suprantamų yra algoritmas, aprašytas Dž. Bentli (Jon Bentley) ir jo priskirtas N. Lomuto (Nico Lomuto):

**Lomuto dalinimo algoritmas greito rūšiavimo (quicksort) metodui:**

**Tikslas:** Padalinti masyvą Key[First..Last] taip, kad Key[First..PivotPos-1] < Key[PivotPos] ir Key[PivotPos] <= Key[PivotPos+1..Last].

**Pradinės sąlygos** (preconditions): First, Last ir Key[First..Last] yra inicializuoti.

**Galinės sąlygos** (postconditions): Key[First..PivotPos-1] < Key[PivotPos] ir

Key[PivotPos] <= Key[PivotPos+1..Last]

**Žingsniai** (steps):

1. Pasirinkti tinkamą dalinančią reikšmę ir, sukeitus elementus, patalpinti ją į Key[First].

2. Naudoti indeksą PivotPos žymėjimui einamajai dalinančios reikšmės pozicijai. Iš pradžių PivotPos lygi First.

3. Visoms indeksų reikšmėms nuo First+1 iki Last:

Jei Key[indeksas] yra mažesnė už dalinančią reikšmę,

a. Padidinti vienetu PivotPos.

b. Sukeisti vietomis Key[indeksas] elementą su Key[PivotPos].

4. PivotPos dabar yra už kairiosios dalies, t.y. pozicijoje kur dalinanti reikšmė turėtų būti, todėl sukeisti vietomis Key[First] elementą su Key[PivotPos].

Išbandykime dalinimo algoritmą su anksčiau pateiktu pavyzdžiu. Jei pasirinkta dalinanti reikšmė 13 (nesvarbu koku būdu), tai situacija prieš 3-čią žingsnį bus tokia:

13<sub>First</sub>, PivotPos, 28<sub>Indeksas</sub>, 40, 6, 35, 4, 10, 21, 7, 17<sub>Last</sub>

Indeksas didėja, kol randama reikšmė (6) mažesnė už dalinančią (13):

13<sub>First</sub>, PivotPos, 28, 40, 6<sub>Indeksas</sub>, 35, 4, 10, 21, 7, 17<sub>Last</sub>

Tada PivotPos reikšmė didinama vienetu:

13<sub>First</sub>, 28<sub>PivotPos</sub>, 40, 6<sub>Indeksas</sub>, 35, 4, 10, 21, 7, 17<sub>Last</sub>

Ir elementai keičiami vietomis:

$13_{\text{First}}, 6_{\text{PivotPos}}, 40, 28_{\text{Indeksas}}, 35, 4, 10, 21, 7, 17_{\text{Last}}$

Vėl indeksas didėja, kol randama reikšmė (4) mažesnė už dalinančią (13):

$13_{\text{First}}, 6_{\text{PivotPos}}, 40, 28, 35, 4_{\text{Indeksas}}, 10, 21, 7, 17_{\text{Last}}$

Tada PivotPos reikšmė didinama vienetu:

$13_{\text{First}}, 6, 40_{\text{PivotPos}}, 28, 35, 4_{\text{Indeksas}}, 10, 21, 7, 17_{\text{Last}}$

Ir elementai keičiami vietomis:

$13_{\text{First}}, 6, 4_{\text{PivotPos}}, 28, 35, 40_{\text{Indeksas}}, 10, 21, 7, 17_{\text{Last}}$

Šis procesas kartojamas, kol indeksas pasiekia masyvo pabaigą. tada turime situaciją:

$13_{\text{First}}, 6, 4, 10, 7_{\text{PivotPos}}, 40, 28, 21, 35, 17_{\text{Last}}$

Dabar PivotPos nurodo korektišką dalinančios reikšmės poziciją, todėl pakeičiame vietomis elementą Key[First] su Key[PivotPos]:

$7_{\text{First}}, 6, 4, 10, 13_{\text{PivotPos}}, 40, 28, 21, 35, 17_{\text{Last}}$

Taigi masyvas padalintas taip, kaip reikėjo. Kaip jau minėjome, greito rūšiavimo algoritmo darbas priklauso nuo dalinančios reikšmės pasirinkimo. Jeigu kaip dalinanti reikšmė imamas pirmas elementas, mūsų nagrinėtame pavyzdyje, kartais gauname dalis, kuriose nėra nė vieno elemento. Kai greito rūšiavimo algoritmas veikia geriausiai, kiekvienas dalinimas padalina einamąjį segmentą daugmaž pusiau. Dalinimas peržiūri reikšmes nuo First+1 iki Last, t.y. atlieka N-1 rakto palyginimo operaciją. Jeigu N elementų surūšiavimui reikia atlikti C(N) rakto palyginimo operacijų, tai

$$C(N) = (N-1) + 2 \cdot C(N/2)$$

Tai rekurentinė formulė rakto palyginimo operacijų skaičiui rasti. Šioje formulėje N/2 turi prasmę tik tuo atveju, jei tai sveikas skaičius. Paprastumo dėlei tarkime, kad N yra ne tik lyginis, bet ir 2 laipsnis, t.y.  $N=2^L$ . Šiuo atveju turėtume:

$$C(2^L) = (2^L - 1) + 2C(2^{L-1}) = (2^L - 1) + 2C(2^{L-1})$$

Padalinę abi puses iš  $2^L$  turėtume:

$$C(2^L)/2^L = 1 - 2^{-L} + C(2^{L-1})/2^{L-1}$$

Analogiškai įsistatydami  $C(2^{L-1})$  ir t.t. reikšmes gautume:

$$C(2^L)/2^L = (1 + 1 + \dots + 1) - (2^{-L} + 2^{1-L} + \dots + 2^{L-1-L}) + C(1)$$

Pirmoje sumoje yra L narių, todėl ji lygi L. O  $C(1)$  yra rakto palyginimo operacijų skaičius, reikalingas tam, kad surūšiuoti vieną elementą, t.y. 0. Todėl gauname:

$$C(2^L)/2^L = L - (1/2^L + 1/2^{L-1} + \dots + 1/2)$$

Suma skliaustuose yra  $(1/2 + 1/4 + 1/8 + \dots + 1/N)$ . Kai N yra be galo didelis, ji konverguoja į 1.

Taigi galutinis rezultatas:

$$C(N) = C(2^L) \approx 2^L (L-1) = N (\log_2 N - 1) = N \log_2 N - N = O(N \log_2 N)$$

Parodėme, kad geriausiu atveju, kai masyvas yra dalinamas į dvi daugmaž lygias dalis, greito rūšiavimo algoritmo sudėtingumas yra  $O(N \log_2 N)$ , kas žymiai geriau negu  $O(N^2)$ . Šis pranašumas prarandamas, jei blogai parenkamas dalinantis elementas. Blogiausiu atveju, kai masyvas yra beveik surūšiuotas arba surūšiuotas atvirkščia tvarka, viena dalis gaunama tuščia ir kiekviename žingsnyje vienintelis dalinantis elementas padedamas į savo vietą. Kadangi yra  $O(N)$  rekursyvių kreipinių, kiekvienas iš kurių atlieka  $O(N)$  rakto palyginimo operacijų, tai blogiausiu atveju greito rūšiavimo algoritmo sudėtingumas yra  $O(N^2)$ .

Reziumuojant pažymėkime, kad greito rūšiavimo algoritmas netinkamas, jei masyvas yra beveik surūšiuotas, tada geriau tinka rūšiavimo įterpimu (*insertion sort*) metodas. Tačiau jis yra greičiausias bendros paskirties metodas atsitiktiniams masyvams rūšiuoti ir dažniausiai pasirenkamas, kai dirbama su dideliais masyvais, tačiau telpiančiais operatyvioje atmintyje.

### Rūšiavimas sujungimu (*merge sort*)

Ne visada rūšiuojamą masyvą galima nuskaityti į operatyvią atmintį. Tokiais atvejais tenka naudoti išorinio rūšiavimo metodus. Vienas iš jų yra rūšiavimo sujungimu (*merge sort*) metodas.

Panagrinėkime jį, remdamiesi pavyzdžiu. Tarkime turime failą skaičių, išdėstytą atsitiktine tvarka, o reikia juos surūšiuoti nemažėjimo tvarka.

Failas\_1: 13 4 2 7 15 12 15 5 10

Mes galime sugrupuoti failo skaičius į poras ir kiekvienoje poroje juos surūšiuoti (paskutinis skaičius 10 lieka vienas):

Failas\_1: (4 13) (2 7) (12 15) (5 15) (10 )

Dabar mes pradedame kartotinį procesą vadinamą skirstymu ir jungimu (*distributing and merging*). Pirmiausia **paskirstome** (*distribute*) gautas poras į atskirus failus:

Failas\_2: (4 13) (12 15) (10 )

Failas\_3: (2 7) (5 15)

Dabar **sujunkime** (*merge*) pirmąją 2-o failo porą su pirmąja 3-o failo pora, antrąją 2-o failo porą su antrąja 3-o failo pora ir t.t., patalpindami rezultatus į pradinį failą. (Sujungimo metu palaikoma nedidėjanti elementų tvarka). Po šio veiksmo gausime elementų grupes dvigubai didesnes nei prieš tai:

Failas\_1: (2 4 7 13) (5 12 15 15) (10 )

Dabar pakartokime skirstymo-jungimo procesą. Skirstome į skirtingus failus:

Failas\_2: (2 4 7 13) (10 )

Failas\_3: (5 12 15 15)

Jungiame į pradinį failą:

Failas\_1: (2 4 5 7 12 13 15 15) (10 )

Skirstome į skirtingus failus:

Failas\_2: (2 4 5 7 12 13 15 15)

Failas\_3: (10 )

Jungiame į pradinį failą:

Failas\_1: (2 4 5 7 10 12 13 15 15)

Po kiekvieno skirstymo-jungimo žingsnio surūšiuotos grupės darosi vis ilgesnės (dvigubai, išskyrus, paskutinę) ir ilgesnės, kol visas failas tampa viena surūšiuota grupe.

Dalinimo procesas pakankamai elementarus, o sujungimą panagrinėkime detaliau. Turime du failus, kiekviename iš kurių yra skaičiai surūšiuoti nemažėjimo tvarka. Tikslas yra sujungti šiuos failus į vieną failą, kuriame skaičiai būtų išdėstyti nemažėjimo tvarka.

Failas\_1: 2 5 11 12 16 <eof>

Failas\_2: 7 9 10 15 <eof>

Mes norime sujungti šiuos du failus į vieną rezultatų failą MergedFile, kuris pradiniu momentu yra tuščias. Kiekviename faile turime nuorodas (rodyklės) į atitinkamą einamąjį elementą. Pradiniu momentu abi rodyklės rodo į pirmuosius elementus.

Failas\_1: ↑ 2 5 11 12 16 <eof>

Failas\_2: ↑ 7 9 10 15 <eof>

MergedFile: ↑

Kadangi 2 yra mažiau negu 7, kopijuojame 2 į rezultatų failą ir perstumiamo pirmo failo rodyklę:

Failas\_1: 2 ↑ 5 11 12 16 <eof>

Failas\_2: ↑ 7 9 10 15 <eof>

MergedFile: 2 ↑

5 < 7, tai nukopijuojame 5 į rezultatų failą ir perstumiamo pirmo failo rodyklę:

Failas\_1: 2 5 ↑ 11 12 16 <eof>

Failas\_2: ↑ 7 9 10 15 <eof>

MergedFile: 2 5 ↑

11 > 7, tai nukopijuojame 7 į rezultatų failą ir perstumiamo antro failo rodyklę:

Failas\_1: 2 5 ↑ 11 12 16 <eof>

Failas\_2: 7 ↑ 9 10 15 <eof>

MergedFile: 2 5 7 ↑

Šį procesą galima aprašyti tokiu fragmentu:

```
while not eof(File_1) and not eof(File_2) do
  if File_1↑ < File_2↑ then
    begin
      File_1↑ kopijuoti į MergedFile↑
      Perstumti File_1 rodyklę
      Perstumti MergedFile rodyklę
    end
  else
    begin
      File_2↑ kopijuoti į MergedFile↑
      Perstumti File2 rodyklę
      Perstumti MergedFile rodyklę
    end
  end
end
```

Procesas baigiasi, kai pasiekiamas vieno iš failų pabaiga (mūsų pavyzdyje, antrojo):

Failas\_1: 2 5 11 12 ↑16 <eof>

Failas\_2: 7 9 10 15 ↑<eof>

MergedFile: 2 5 7 9 10 11 12 15 ↑

Tada belieka nukopijuoti likusią failo “uodegą” į rezultatų failą, pavyzdžiui, taip:

```
while not eof(File_1) do
  begin
    File_1↑ kopijuoti į MergedFile↑
    Perstumti File1 rodyklę
    Perstumti MergedFile rodyklę
  end
end
```






# Perrinkimas

Šiuolaikiniame gyvenime tenka susidurti su užduotimis, kuriose reikia rasti sprendinį iš daug galimų variantų. Vienas iš sprendimo būdų yra perrinkti (patikrinti) galimus variantus, bet variantų dažnai būna tiek daug, kad perrinkimas visų galimų variantų yra praktiškai neįmanomas (*angl. computationally unfeasible*). Reikia kažkaip apriboti nagrinėjamus variantus, tačiau apribojant negalima prarasti egzistuojančio sprendinio. Dažnai sprendinio radimui pakanka perrinkti tik tam tikrus variantus. Šios problemos sprendimo būdą detaliau panagrinėsime su tokiu uždaviniu: šachmatų lentoje sustatyti 8 karalienes taip, kad nei viena iš jų nekirstų kitos.

Viena iš strategijų yra perrinkti visus galimus variantus ir iš jų išrinkti tinkamą. Tačiau iš viso yra **4 426 165 368** galimi aštuonių karalienių išdėstymo variantai, kas gerai parodo, kad tokia strategija visai netinkama. Galima pastebėti, kad yra ypatybė, kuri žymiai sumažina perrinkimų skaičių: vienoje eilutėje ir viename stulpelyje negali būti pastatytos dvi karalienės. Kitaip sakant, kiekvienas stulpelis ir kiekviena eilė gali turėti tik vieną karalienę. Tokiu būdu, perrinkimų skaičius sumažėja iki **8!=40 320**, kas yra labiau įvykdoma. Prieš detalų šio algoritmo nagrinėjimą reikėtų susipažinti su tokia strategija kaip **valdymas su grįžimais** (*backtracking*).






Apibrėžimas. Valdymas su grįžimais (*Backtracking*) – tai strategija, kuri leidžia grįžti atgal po žingsnį ir pasirinkti kitą sprendimo vystymo kryptį, jei atsirado atvejis, kai sprendimo nėra ar jis netinkamas.

(1) Padarom prielaidą, kad pirmą karalienę stovės pirmo stulpelio pirmame kvadrato. Tada antra karalienė stovės antro stulpelio trečiame kvadrato, nes pirmame ir antrame kvadratuose ją kirstų pirmą karalienę, t.y. sekančią karalienę statome į pirmą neužimtą sekančio stulpelio kvadratą. Brėžinys (1) vaizduoja tokiu būdu sudėliotas penkias karalienes. Brėžinyje taškais pažymėti kvadratai, į kuriuos negalima statyti karalienės, nes ją kirstų kita, anksčiau pastatyta, karalienė.







	•	•	•	•	•		
	•	•		•	•		
		•		•	•		
		•			•		
					•		
					•		
					•		
					•		



(2) Reikia pastebėti, kad taip sustatytos penkios karalienės kerta visus šešto stulpelio kvadratus, t.y. pastatyti karalienės šeštame stulpelyje negalima. Todėl reikia sugrįžti atgal į penktą stulpelį ir perstatyti karalienę į kitą kvadratą (*backtracing* strategija), pavyzdžiui, į paskutinį, kaip pavaizduota brėžinyje (2).

	●	●	●	●	●		
	●	●		●	●		
		●		●	●		
		●		●	●		
				●	●		
				●	●		
				●	●		
					●		

(3) Tačiau tai dar neišsprendžia problemos. Penkios karalienės vis tiek kirs šeštąjį kiekviename šešto stulpelio langelyje. Todėl reikia grįžti prie ketvirto stulpelio ir perstatyti ketvirtą karalienę, pavyzdžiui, į septintą kvadratą, o penktą karalienę - pavyzdžiui, į penkto stulpelio antrą kvadratą (brėžinys(3))

	●	●	●	●			
	●	●	●				
		●	●				
		●	●				
			●				
			●				
							

Štai dabar galima pastatyti šeštą karalienę į šešto stulpelio ketvirtą kvadratą. Tuo pačiu būdu galima sustatyti ir likusias karalienes.

**Realizacija:** iš pirmo žvilgsnio suprantama, kad paprasčiausia tokios užduoties realizacija yra panaudojant rekursiją.

## Prioritetinės eilės (*Priority Queues*)

Daugelyje taikymų tam, kad gauti reikiamus rezultatus, užtenka rūšiuoti duomenis tik iš dalies, nes pilnas rūšiavimas nėra būtinas, arba kartais tenka rūšiuoti tik duomenų poaibius. Pavyzdžiui, duomenų bazėse dažnai tenka sukaupti tam tikrą kiekį duomenų, iš jų išrinkti didžiausią arba mažiausią, po to vėl papildomai kaupti aibę duomenų, iš jų vėl išrinkti didžiausią arba mažiausią, ir t.t. Šias operacijas atitinkančiai duomenų struktūrai netinka naudoti eilę ar steką. Žemiau nagrinėjama duomenų struktūra – prioritetinė eilė – yra geriau pritaikyta tokiai situacijai.

### Prioritetinėms eilėms reikalingos tokios operacijos:

- Sukurti tuščią prioritetinę eilę (*Create*);
- Nustatyti, ar eilė tuščia (*IsEmpty*);
- Įdėti naują elementą (*Insert*);
- Išmesti elementą su didžiausiu prioritetu (*Remove*);
- Sujungti dvi prioritetines eiles į vieną (*Join*).

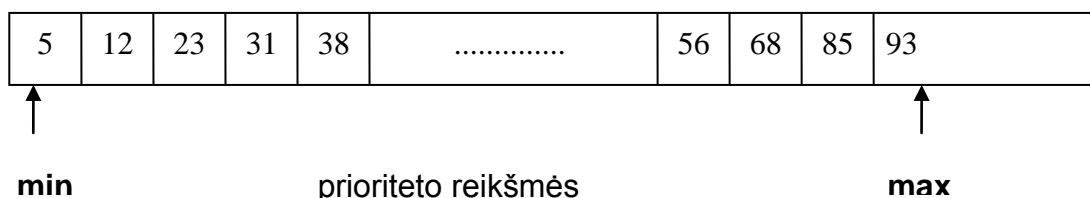
Reikėtų atkreipti dėmesį į tai, kad operacija *Remove* išima ne elementą su didžiausia reikšme, o elementą, kurio prioritetas didžiausias. Jeigu elementų su tuo pačiu prioritetu yra daugiau nei vienas, tai išimamas elementas pagal eilės principą, t. y. tas elementas, kuris buvo seniausiai įdėtas.

### Prioritetinių eilių realizacijos:

#### 1. Masyvas.

Realizacijoje masyvu dažniausiai elementai išdėstomi prioritetų didėjimo tvarka. Tai užtikrina, kad elementas su didžiausia prioriteto reikšme bus masyvo gale kaip pavaizduota brėžinyje.

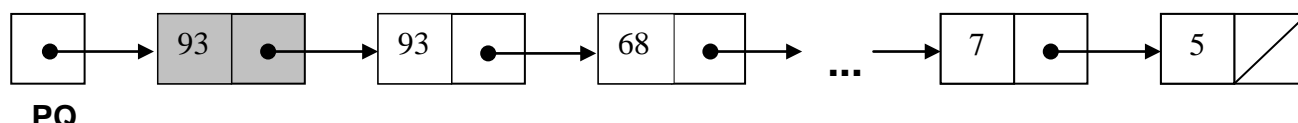
**Pastaba:** šiame ir tolimesniuose pavyzdžiuose vaizduojami tik elementų prioritetai, o pačios elementų reikšmės nevaizduojamos.



Realizacijoje masyvu sunkumai gali kilti su įterpimo operacija *Insert*. Tam, kad įterpti elementą reikėtų rasti teisingą poziciją – elementas turi būti įterptas pagal savo prioritetą o tarp turinčių tokį prioritetą kaip pirmasis – o po to perstumti visus elementus, kad atsirastų vietos naujam elementui.

#### 2. Tiesinis sąrašas.

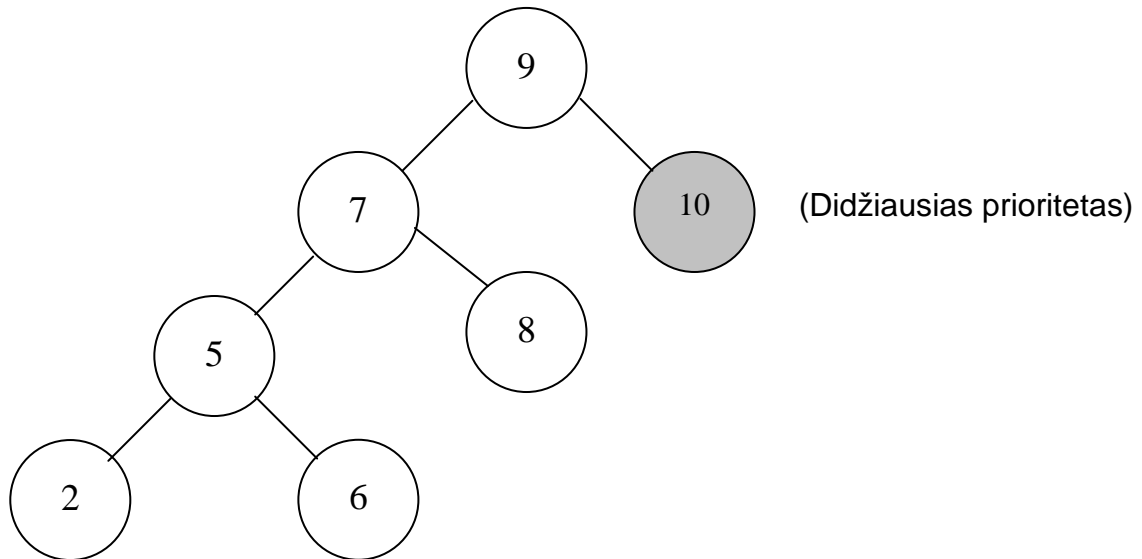
Iš kitos pusės, realizacijoje tiesiniu sąrašu dažniausiai elementai išdėstomi prioritetų mažėjimo tvarka, t.y. elementas su didžiausia prioriteto reikšme yra sąrašo pradžioje.



Tokiu būdu, operacija *Remove* paprasčiausiai grąžina reikšmę, į kurią rodo rodyklė PQ, o po elemento išmetimo rodyklė turi rodyti į sekantį elementą. Tuo tarpu, operacija *Insert* turi eiti sąrašu tol, kol ras elementui tinkamą vietą – naujas elementas turi būti įterptas prieš pirmą elementą, turintį mažesnę prioritetą už naują elementą.

### 3. Dvejtainis paieškos medis.

Prioritetinės eilės realizacija pavaizduota brėžinyje. Laikomasi taisyklės: *kiekvienos viršūnės dešiniame pomedyje yra elementai su didesniais prioritetais, o kairiame pomedyje su mažesniais prioritetais.*

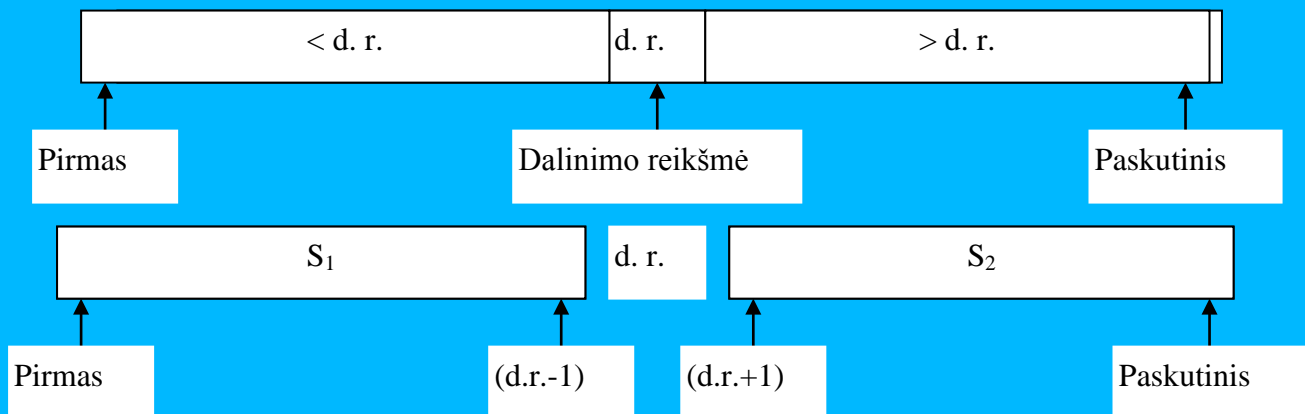


Tokios realizacijos ypatumas yra tame, kad elementas su didžiausia prioriteto reikšme bus pačioje dešinėje viršūnėje. Tokiu būdu, reikia sekti dešiniuosius vaikus, kol atsiras viršūnė be dešiniojo vaiko. Be to, šios viršūnės išėmimas gana lengvas, nes dažniausiai viršūnė turi tik vieną vaiką.

## Greito rūšiavimo (*Quick Sort*) metodas elementams išrinkti

Pagrindinė greito rūšiavimo (*Quick Sort*) metodo idėja gali būti panaudota ne vien duomenims rikiuoti. Labai dažnas toks uždavinys: iš skaičių sekos išrinkti **k-ąjį** pagal dydį elementą. Tokį uždavinį, aišku, galima spręsti rikiuojant turimą skaičių seką. Tačiau tai nėra efektyvu. Kitoks galimas sprendimo būdas – parinkti tinkamą elementą ir, pritaikius *partition* procedūrą, suskaidyti seką į dvi dalis, iš kurių vienoje yra elementai mažesni už parinktą, o kitoje – didesni.

Procedūra *partition*.



Skaidymo procedūra dalina elementų seką į dvi atskiras sekas:  $S_1$ , kurioje elementai yra mažesni už dalinimo reikšmę, ir  $S_2$ , kurioje elementai didesni už dalinimo reikšmę. Toks suskaidymas padeda dalinimo reikšmę į teisingą sekos vietą.

Pseudokodas

```
function Select ( $k, A, First, Last$ ) : grąžina reikšmę iš masyvo  $A[First..Last]$ ;  
begin  
  Parinkti dalinimo reikšmę  $dr$  iš  $A[First..Last]$ ;  
  Suskaidyti  $A[First..Last]$  procedūra Partition ir gauname surikiuoto  $dr$  indeksą.  
  //Nustatome mūsų ieškomo elemento padėtį dalinimo reikšmės atžvilgiu.  
  If  $k < dr - First + 1$  then  $Select := Select(k, A, First, dr - 1)$   
  else if  $k = dr - First + 1$  then  $Select := dr$   
    else  $Select := Select(k - (dr - First + 1), A, dr + 1, Last)$   
end;
```

Funkcijos Select ypatumas yra tame, kad ji tik grąžina **k** mažiausio elemento reikšmę nekeisdama masyvo (masyvą keičia iškviesta procedūra Partition). Be to, funkcija rekursyviai iškviečia save tik toje masyvo vietoje, kurioje yra ieškomas elementas, ir visiškai neiškviečia, jei ieškomas elementas yra dalinimo reikšmė.

## Vidinis rikiavimas sujungimu (*Merge sort*)

Nagrinėsime vidinio rikiavimo algoritmą *Merge sort*. Tarkime, imamas masyvas ir jis dalinamas pusiau, tada kiekviena gauta jo dalis vėl yra dalinama pusiau. Tai lengviausia realizuoti rekursija, nes dalinant kartojame tuos pačius žingsnius iki išsigimusio atvejo (*degenerate case*), kai suskaidytų dalių dydis tampa lygus vienam elementui. Šis **I etapas** vadinamas skaidymu. Skaidant į dalis, elementų tvarka nekeičiama.

**II etapas** yra vadinamas jungimu (suliejimu) (*merge steps*), jo metu suskaidytos dalys yra jungiamos, tvarkant elementus, kas ir duoda surūšiavimą.

Pavyzdžiui, turime masyvą iš 6 elementų:

I algoritmo dalis – skaidymas.

1. (38 16 27 39 12 27)

Jį daliname pusiau (galime pasinaudoti  $mid := (pirmas + paskutinis) \div 2$ ):

2. (38 16 27)                      (39 12 27)

Gautas puses vėl daliname pusiau:

3. (38 16) (27)                      (39 12) (27)

4. (38) (16)                      (39) (12)

Dabar prasideda II etapas, t.y. jungimo žingsniai (atliekant antrojo etapo žingsnius, suliejant dalis, elementus rikiuojame):

5. (16 38) (27)                      (12 39) (27)

6. (16 27 38)                      (12 27 39)

7.                      (12 16 27 27 38 39)

**Duomenų struktūros:** šiam algoritmui realizuoti naudojami masyvai bei rekursija, bet galima realizuoti ir be rekursijos.

**Privalumai:** greitas algoritmas laiko atžvilgiu.

**Trūkumai:** Jungiant duomenis, jam reikalingas papildomas masyvas, kuris yra tokio pat dydžio (jungiant suskaidytus masyvo elementus, mes naudojame laikiną masyvą, į kurį įrašome sulietus ir surikiuotus elementus).

## Skaitmeninis rikiavimas (*Radix sort*)

Algoritmas pagrįstas grupių formavimu, suskirstant duomenis pagal atitinkamoje pozicijoje esančius duomenų elementus, pradedant nuo mažiausiai svarbių (mažiausią vertę turinčių) – paskutinės raidės ar paskutinio skaitmens.

Pirmiausia suvienodinami rikiuojamų duomenų ilgiai, nekeičiant jų reikšmės: jei rikiuojamos simbolių eilutės, iš dešinės prirašomi tarpai; jei rikiuojami skaičiai, iš kairės pusės prirašomi nuliai.

Tada atliekamas skirstymas į grupes pagal paskutinį simbolių/skaitmenį, nekeičiant elementų tvarkos. Grupių skaičius priklauso nuo galimų reikšmių: rūšiuojant skaičius, formuojama 10 grupių, nes yra 10 skaitmenų; rikiuojant simbolių eilutes, grupių skaičius lygus skirtingų simbolių skaičiui (anglų kalbos žodžiams rūšiuoti reikia formuoti 27 grupes: 26 raidės ir tarpo simbolis, reikalingas žodžių ilgiam suvienodinti). Po to duomenys apjungiami pagal jų priklausymą grupei: pirmos grupės duomenys, antros grupės duomenys ir t.t. Rezultate gauname duomenis surūšiuotus pagal paskutinį simbolių/skaitmenį.

Po to tokios pat skirstymo ir jungimo operacijos atliekamos pagal priešpaskutinį simbolį/skaitmenį, kas garantuoja surūšiavimą pagal 2 paskutinius simbolius/skaitmenis.

Pakartojus šiuos veiksmus visiems duomenų simboliams/skaitmenims, gauname pilnai surūšiuotus duomenis.

Pavyzdžiui, duota skaičių seka: 5 15 9 13 69 1 3

Sulyginame jų ilgius:

05 15 09 13 69 01 03

Suskirstome į grupes pagal paskutinį skaitmenį, elementų tvarka grupėse išlieka:

Grupės	0	1	2	3	4	5	6	7	8	9
Duomenys	( )	(01)	( )	(13 03)	( )	(05 15)	( )	( )	( )	(09 69)

Apjungiame grupių duomenis į vieną seką:

01 13 03 05 15 09 69

Suskirstome į grupes pagal prieš paskutinį skaitmenį:

Grupės	0	1	2	3	4	5	6	7	8	9
Duomenys	(01 03 05 09)	(13 15)	( )	( )	( )	( )	(69)	( )	( )	( )

Apjungiame grupių duomenis į vieną seką:

01 03 05 09 13 15 69

**Duomenų struktūra:** algoritmą realizuoti galima naudojantis masyvais

**Pseudokodas:**

```
RadixSort (A, N, d) {rikiuoja n sveikojo (integer) tipo skaitmenų (iš d elementų) A masyve}
  for j:=d downto 1 do
    begin
      Priskiriame 10 grupių tuščias reikšmes
      Priskiriame kiekvienos grupės skaitikliui su reikšmę 0
      For i:= 1 to n do
        Begin
          K:= j-tasis skaitmuo iš masyvo elemento A[i]
          Patalpiname A[i] k-os grupės gale
          Padidiname k-os grupės skaitliuką vienetu
        End {end for i}
      Pakeičiame visus masyvo A elementus elementais iš grupių 1..10
    End {end for j}
```

**Privalumai:** tai yra labai greitas rikiavimo algoritmas. Remiantis pseudokodu, matome, kad algoritmas reikalauja  $N$  žingsnių kiekvieną kartą, kai jis formuoja grupes bei  $N$  žingsnių sujungti vėl į vieną grupę. Šiuos  $2N$  žingsnius algoritmas kartoja  $d$  kartų. Todėl *Radix sort* reikalauja  $2 \cdot N \cdot d$  žingsnių surūšiuoti  $N$  elementų, kurie sudaryti iš  $d$  simbolių/skaitmenų. Atsižvelgiant, kad algoritmui palyginimai nėra reikalingi, laikome, kad jo sudėtingumas yra  $\Theta(N)$ .

**Trūkumai:** nėra labai paplitęs rikiavimo algoritmas, nes rikiuojant duomenis pagal abėcėlę, kurioje yra 27 simboliai, prireiks 27 masyvų skirtų realizuoti grupes, o tai užims daug vietos (galimas sprendimas: grupėms naudoti sąrašus).

## Rekursija – Hanojaus bokštų uždavinys

Duoti trys stulpai ir N diskų. Ant vieno (pradinio) stulpo sumauti diskai didėjimo tvarka, einant iš viršaus į apačią. Reikia visus diskus perkelti nuo pradinio stulpo ant laisvo (tikslo) stulpo, pasinaudojant atsarginiu stulpu.

Apribojimai:

1. Per vieną ėjimą galima nuimti tik vieną diską ir jį būtina iš karto uždėti ant kito stulpo.
2. Didesnio disko negalima dėti ant mažesnio.

Visai nesunku sugalvoti rekursinį šio uždavinio sprendimo algoritmą:

Jei diskas tik vienas (N=1), uždavinys elementarus – tiesiog perkeliame šį 1 diską nuo pradinio stulpo ant tikslo stulpo.

Tarkime, kad mes mokame išspręsti uždavinį su N-1 disku. Tada norėdami perkelti N diskų elgiamės taip:

- N-1 diską nuo pradinio stulpo perkeliame ant atsarginio stulpo (pagal prielaidą tai mes jau mokame padaryti);
- tada ant pradinio stulpo likusį 1 diską (didžiausią) tiesiog perkeliame ant tikslo stulpo;
- N-1 diską nuo atsarginio stulpo perkeliame ant tikslo stulpo (pagal prielaidą tai mes jau mokame padaryti).

Žemiau pateikiami 2 variantai šį algoritmą realizuojančios rekursinės procedūros pseudokodo (naudojami pažymėjimai: *count* - diskų skaičius, *source* - pradinis stulpas, *dest* - tikslo stulpas, *spare* - tarpinis stulpas):

```
Towers (count, source, dest, spare)
if count = 1 then tiesiog perkelti 1 diską nuo pradinio stulpo (source) ant tikslo stulpo (dest)
else
  begin
    Towers (count-1, source, spare, dest)
    Towers (1, source, dest, spare)
    Towers (count - 1, spare, dest, source)
  end
```

arba

```
Towers (count, source, dest, spare)
if (count > 0)
  begin
    Towers (count - 1, source, spare, dest)
    perkelti 1 diską nuo pradinio stulpo (source) ant tikslo stulpo (dest)
    Towers (count - 1, spare, dest, source)
  end
```

Dabar nustatysime, kiek reikės žingsnių N diskų perkėlimui. Pažymėkime **žingsniai(N)** žingsnių skaičių, reikalingą perkelti N diskų.

Kai N = 1, tai **žingsniai(1) = 1**.

Kai N > 1, procedūra *Towers* iškviečiama 3 kartus: perkelti N-1 diską, perkelti 1 diską ir vėl perkelti N-1 diską. Taigi galime apskaičiuoti **žingsniai(N)** pagal tokią rekurentinę formulę:

$$\text{žingsniai}(N) = \text{žingsniai}(N-1) + \text{žingsniai}(1) + \text{žingsniai}(N-1) = 2 * \text{žingsniai}(N-1) + 1$$

Žingsnių skaičių galima apskaičiuoti pagal tokią formulę:

$$\mathbf{\text{žingsniai}(N)} = 2^N - 1 \ (\forall N \geq 1)$$

Šios formulės teisingumą įrodysime matematinės indukcijos metodu.

Patikriname jos teisingumą, kai  $N=1$ .

$$\mathbf{\text{žingsniai}(1)} = 2^1 - 1 = 2 - 1 = 1$$

Tarkime, kad ji teisinga su  $N-1$  (t.y.  $\mathbf{\text{žingsniai}(N-1)} = 2^{N-1} - 1$ ), tada:

$$\mathbf{\text{žingsniai}(N)} = 2 * \mathbf{\text{žingsniai}(N-1)} + 1 = 2 * (2^{N-1} - 1) + 1 = 2^N - 2 + 1 = 2^N - 1$$

Įrodymas baigtas.



### Šelo rūšiavimas (*Shell sort*)

Šį algoritmą 1959 metais sukūrė Donaldas Šelas (*Donald Shell*) ir jis buvo pavadintas pagal autorių: Šelo rūšiavimas (*Shell sort*).

Šelo rūšiavimas yra rūšiavimo įterpimu (*insertion sort*) modifikacija, atlikta atsižvelgiant į du aspektus:

- rūšiavimas įterpimu yra efektyvus, jeigu duomenys yra beveik surūšiuoti;
- rūšiavimas įterpimu nėra labai efektyvus, nes peržiūri reikšmes pasistumdamas tik per vieną poziciją.

Šelo rūšiavimas veikia panašiai kaip rūšiavimas įterpimu, tačiau pertvarko duomenis didesniu „atstumu“ (*gap*) tarp lyginamų elementų. Pavyzdžiui, jei atstumas yra 3, tai šioje iteracijoje sutvarkomos (surūšiuojamos) 3 duomenų aibės:

- 1) elementai esantys pozicijose 1, 4, 7, ...
- 2) elementai esantys pozicijose 2, 5, 8, ...
- 3) elementai esantys pozicijose 3, 6, 9, ...

Tada „atstumas“ yra mažinamas (pavyzdžiui, 2 kartus), kol galiausiai su „atstumu“ lygiu 1 atliekamas paprasčiausias rūšiavimas įterpimu, bet jį atliekant duomenys jau būna beveik surūšiuoti.

Šis algoritmas yra gana greitas ir paprastai realizuojamas, bet įvertinti jo sudėtingumą yra pakankamai sudėtinga. Žemiau pateikiamas bazinio Šelo rūšiavimo algoritmo kodas C kalba:

```
void shellSort(int numbers[], int array_size)
{
    int i, j, increment, temp;

    increment = initial_gap;
    while (increment > 0)
    {
        for (i=0; i < array_size; i++)
        {
            j = i;
            temp = numbers[i];
            while ((j >= increment) && (numbers[j-increment] > temp))
            {
                numbers[j] = numbers[j - increment];
                j = j - increment;
            }
            numbers[j] = temp;
        }
        if (increment/2 != 0)
            increment = increment/2;
        else if (increment == 1)
            increment = 0;
        else
            increment = 1;
    }
}
```

Šiame variante buvo paimtas pradinis „atstumas“ (*initial\_gap*) ir po kiekvienos iteracijos jis mažinamas pusiau, tačiau specialiai parenkant „atstumų“ sekas galima gauti geresnius (algoritmo sudėtingumo atžvilgiu) rezultatus.

Toliau pateikiamas pavyzdys su 16 iš anksto numatytų „atstumų“- žingsnių.

```

void shellsort (int[] masyvas, int elem_sk)
{
    int i, j, k, zingsnis, elem;
    int[] zingsniai = {4356424, 1355339, 543749, 213331, 84801, 27901,
                      11969, 4711, 1968, 815, 277, 97, 31, 7, 3, 1};
    for (k=0; k<16; k++)
    {
        zingsnis = zingsniai[k];
        for (i=zingsnis; i<elem_sk; i++)
        {
            elem=masyvas[i];
            j=i;
            while ((j=>zingsnis) && (masyvas[j-zingsnis]>elem))
            {
                masyvas[j]=masyvas[j-zingsnis];
                j=j-zingsnis;
            }
            masyvas[j]=elem;
        }
    }
}

```

Kaip jau buvo minėta, algoritmo sudėtingumas priklauso nuo pasirinktos „atstumų“ sekos. Keli pasiūlyti variantai:

**Pratt:** su „atstumų“ seka

1, 2, 3, 4, 6, 8, 9, 12, 16, ...,  $2^q 3^p$

Šelo rūšiavimo sudėtingumas yra  $O(N (\log N)^2)$

**Hibbard:** su „atstumų“ seka

1, 3, 7, 15, 31, 63, 127, ...,  $2^k - 1$

Šelo rūšiavimo sudėtingumas yra  $O(N^{3/2})$

**Knuth:** su „atstumų“ seka

1, 4, 13, 40, 121, ...,  $(3^s - 1)/2$

Šelo rūšiavimo sudėtingumas yra  $O(N^{3/2})$

**Sedgewick:** su „atstumų“ seka

1, 5, 19, 41, 109, ...

„atstumas“  $h_s$  :

$9 \cdot 2^s - 9 \cdot 2^{s/2} + 1$ , jei  $s$  lyginis

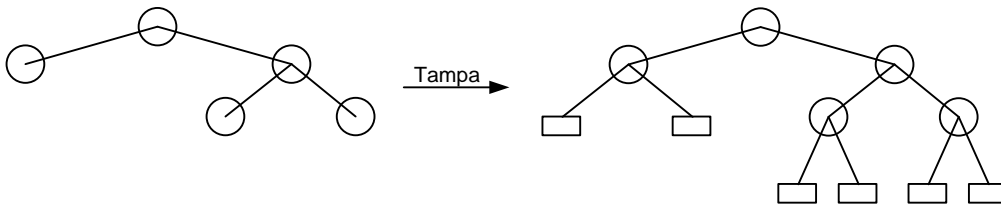
$8 \cdot 2^s - 6 \cdot 2^{(s+1)/2} + 1$ , jei  $s$  nelyginis

Šelo rūšiavimo sudėtingumas yra  $O(N^{4/3})$

## Dvejetainiai paieškos medžiai ir optimalumas.

Iš pradžių reikia išsiaiškinti, ar verta vidutiniškai laikyti dvejetainį paieškos medį subalansuotą ar balansuoti jį atskira operacija. Jei turėsime omeny, kad reikšmės į medį ateina atsitiktine tvarka, kiek vidutiniškai daugiau palyginimo operacijų prireiktų, ieškant elemento gautame medyje negu ieškant elemento visiškai subalansuotame medyje.

Tam kad atsakyti į šį klausimą, iš pradžių paverskime medį į 2-medį. Visas medžio viršūnes laikykime skrituliais, o visus tuščius pomedžius (nulines rodykles) pavaizduokime kvadratais. Visos duoto medžio viršūnės tampa naujo 2-medžio vidinėmis viršūnėmis, o naujos – išorinėmis (lapais). Sėkminga paieška sustos prie vidinės 2-medžio viršūnės, nesėkminga – prie lapo. Toks medis pavaizduotas **Pav 1**.



**Pav 1. Medis su nulinėmis rodyklėmis**

Turėsime omeny, kad visos  $n!$  elementų išdėstymo variacijos yra vienodai tikėtinos. Jei medyje yra  $n$  viršūnių, tai  $S(n)$  žymėsime palyginimo operacijų skaičių sėkmingos paieškos atveju, o  $U(n)$  – nesėkmingos paieškos atveju.

Palyginimo operacijų skaičius, reikalingas norint surasti bet kurią medžio reikšmę, yra vienetu didesnis negu palyginimo operacijų skaičius, reikalingas tos reikšmės įterpimui, o įterpimui prireiktų tiek pat palyginimo operacijų kiek ir nesėkmingos paieškos atvejui, norint parodyti, kad tokio elemento medyje dar nėra. Iš to išplaukia:

$$S(n) = 1 + \frac{U(0) + U(1) + \dots + U(n-1)}{n}$$

Ryšys tarp vidinio ir išorinio kelių ilgių yra:

$$S(n) = (1 + 1/n)U(n) - 1$$

Iš abiejų lygybių išplaukia:

$$(n+1)U(n) = 2n + U(0) + U(1) + \dots + U(n-1)$$

Šią rekurentinį sąryšį pertvarkome  $n-1$  atžvilgiu:

$$nU(n-1) = 2(n-1) + U(0) + U(1) + \dots + U(n-2) \Rightarrow U(n) = U(n-1) + \frac{2}{n+1}$$

Suma  $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$  vadinama  $n$ -tuoju harmoniniu skaičiumi ir apytiksliai lygi

$\ln n$ . Tuomet žinodami, kad  $U(0) = 0$ , galime įvertinti  $U(n)$ :

$$U(n) = 2 \left[ \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n+1} \right] = 2H_{n+1} - 2 \approx 2 \ln n$$

**Teorema:** Vidutinis palyginimo operacijų skaičius vidutiniame dvejetainiame paieškos medyje su  $n$  viršūnių apytiksliai lygus  $2 \ln n = (2 \ln 2)(\lg n)$ .

**Išvada:** Vidutiniškam dvejetainiam paieškos medžiui apytiksliai reikia  $2 \ln 2 \approx 1.39$  karto daugiau palyginimo operacijų negu visiškai subalansuotam medžiui.

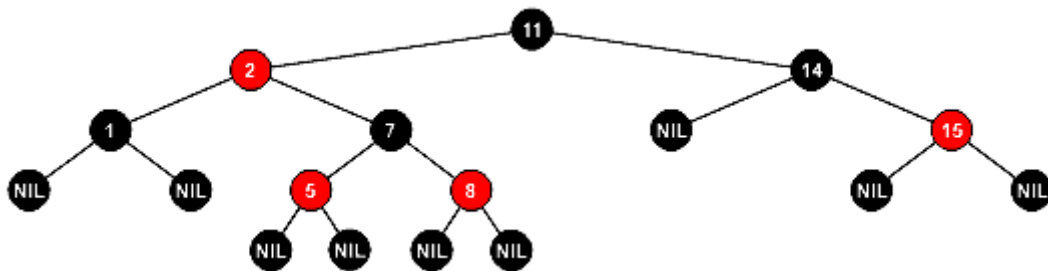
Kitaip tariant, vidutinė nesubalansuoto dvejetainės paieškos medžio kaina yra 39 procentais didesnė. Programose, kur optimalumas labai svarbus, ši kaina turi būti palyginta su papildomomis išlaidomis balansavimo operacijai arba medžio palaikymui subalansuotoje būsenoje.

## Raudonai-juodi medžiai

Raudonai-juodi medžiai – tai dvejetainiai paieškos medžiai, kurių kiekviena viršūnė dar turi ir savo spalvą – raudoną arba juodą. Be to, kiekvienas raudonai-juodas medis pasižymi tokiomis savybėmis:

1. kiekviena viršūnė yra raudona arba juoda;
2. medžio šaknis visada yra juoda;
3. lapai (fiktyvios viršūnės) yra juodi;
4. raudonos viršūnės vaikai yra juodi;
5. kiekvienas kelias nuo viršūnės iki jos lapų turi vienodai juodų viršūnių.

Šie medžiai leidžia palaikyti optimalų variantą. Įterpus ar pašalinus elementą, medis išlieka daugiau ar mažiau panašus į idealiai subalansuotą. Raudonai-juodo medžio, su  $n$  vidinių viršūnių, aukštis visada yra nedidesnis už  $2 \log_2(n + 1)$ . Įterpiant elementą į šį medį reikia nedaugiau dviejų sukimo operacijų.

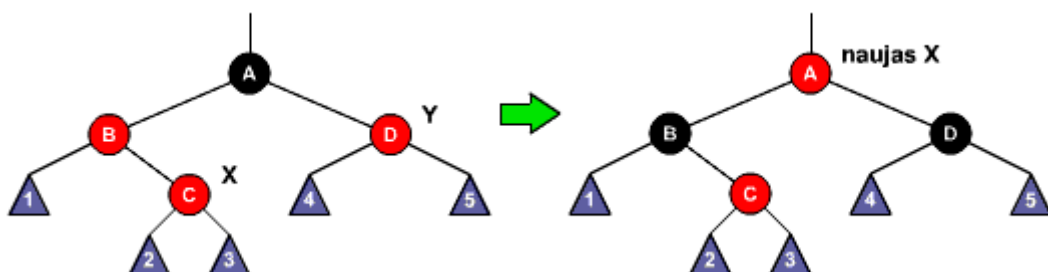


Raudonai-juodo medžio pavyzdys

### Elemento įterpimas

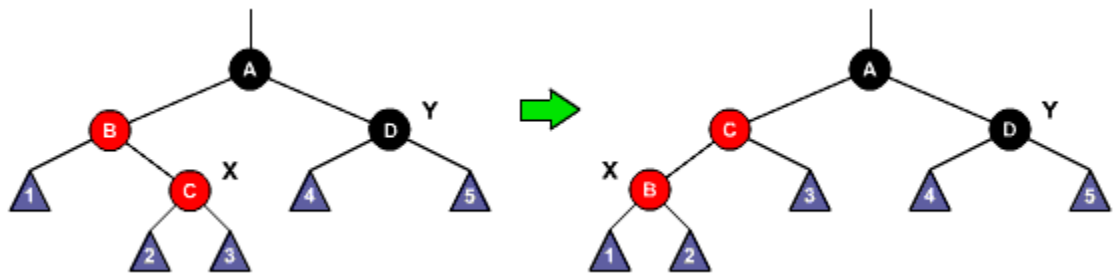
Įterpimas nėra sudėtingas. Pirmu žingsniu elementas įterpiamas kaip į paprastą dvejetainį paieškos medį. Įterpus jam priskiriama raudona spalva. Toliau prasideda medžio koregavimas. Pažymėkime  $X$  naujai įterptą elementą,  $Y$  – elemento  $X$  dėdę. Trikampiu žymėsime viršūnių pomedžius. Tada galimos tokios situacijos:

1.  $X$  tėvas ir  $Y$  yra abu raudoni, tada:
  - $X$  tėvas ir  $Y$  nudažomi juodai;
  - $X$  tėvo tėvas nudažomas raudonai;
  - $X$  tėvo tėvas tampa  $X$ -u.



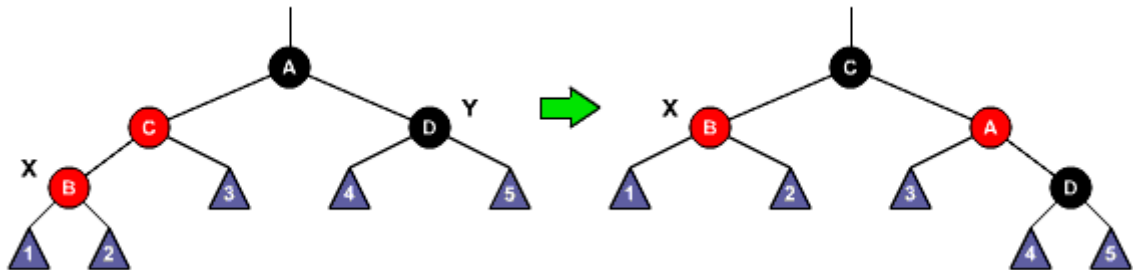
*Pastaba: paveikslėlyje vaizduojamas tik medžio fragmentas, ir naujas  $X$  nėra medžio šaknis.*

2.  $X$  tėvas yra raudonas,  $Y$  yra juodas ir  $X$  yra dešinysis vaikas, tada:
  - $X$  tėvas tampa  $X$ ;
  - atliekama sukimo į kairę operacija apie  $X$  elementą (visada po šio sukimo gauname 3-ią situaciją);



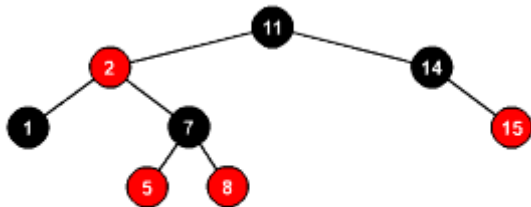
3. X tėvas yra raudonas, Y yra juodas ir X yra kairysis vaikas, tada:

- X tėvo spalva pakeičiama į juodą;
- X tėvo tėvo spalva pakeičiama į raudoną;
- atliekama sukimo į dešinę operacija apie X elemento tėvo tėvą;

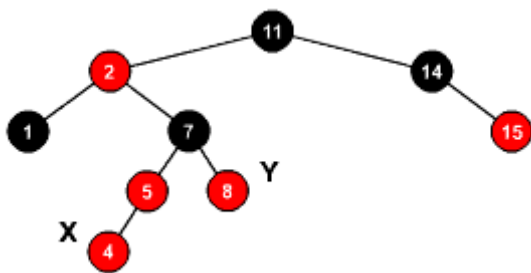


Koregavimas yra baigiamas, kai X tėvo spalva tampa juoda arba kai X tampa šaknine viršūne.

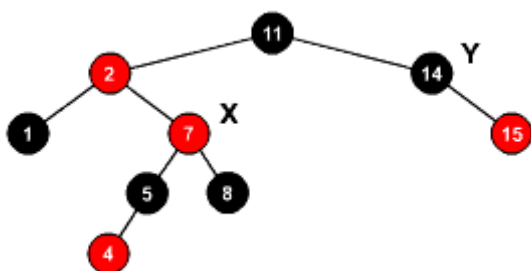
Įterpkime į aukščiau pavyzdyje duotą raudonai-juodą medį elementą 4 (paprastumo dėlei, fiktyvių viršūnių NIL nevaizduosime):



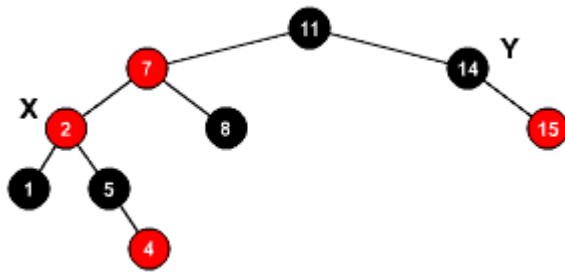
Pradinis medis.



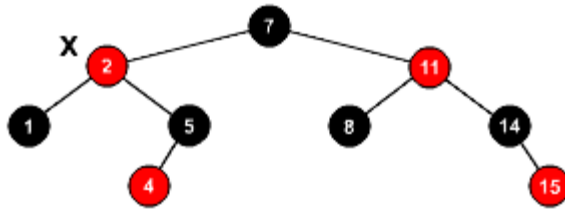
Į medį įterpiamas elementas 4, jam priskiriama raudona spalva. Tėvas ir dėdė raudoni (1 situacija), todėl koreguojama toliau.



Keičiamos 5, 7, 8 spalvos. X pakyla dviem lygiais aukščiau. Dabar tėvas raudonas, dėdė juodas, X – dešinysis vaikas (2 situacija), todėl koreguojama toliau.



X-u tampa 2. Atliekama sukimo į kairę operacija apie viršūnę X. Gaunama prieš tai buvusi situacija, tačiau dabar X yra kairysis vaikas (3 situacija). Koreguojama toliau.



Keičiamos 7 ir 11 spalvos. Atliekama sukimo į dešinę operacija apie viršūnę 11. X tėvo spalva juoda, todėl koregavimas baigiamas.

Elemento įterpimo į raudonai-juodą medį pseudo-kodas:

```
rbInsert(Tree, current) {
    bstInsert(Tree, current); /*įterpimas į dvejetainį paieškos medį */
    color of current = red;
    while ((current is not root) && (color of parent is red)) {
        if (color of uncle is red) {
            color of parent = black;
            color of uncle = black;
            color of grandparent = red;
            current = grandparent;
        }
        else
            if (current is a right child) {
                set current = parent;
                rbRotateLeft(Tree, current);
            }
            else {
                color of parent = black;
                color of grandparent = red;
                rbRotateRight(Tree, grandparent);
            }
    }
}
```

### Elemento pašalinimas

Elemento pašalinimas iš raudonai-juodo medžio yra kur kas sudėtingesnis nei įterpimas, todėl čia aptarsime tik patį principą. Pašalinimas vyksta kaip ir paprastame dvejetiniame paieškos medyje. Tačiau reikia sekti, kad medis išliktų raudonai-juodas. Jei trinama viršūnė yra raudona arba viršūnė yra lapas, tai nieko keisti nereikia. Jei ištrinta viršūnė yra juoda, tada medį reikia koreguoti. Koreguojant yra ieškoma artimiausia tinkanti raudona viršūnė, kuri yra perdažoma juodai ir pakeičia ištrintąją.