Netiesinės duomenų struktūros

Raimondas Čiegis

Matematinio modeliavimo katedra, e-paštas: rc@vgtu.lt

Rugsėjo 1 d., 2022

Dvejetainis medis

Pradedame nagrinėti sudėtingesnes dinamines struktūras, kurios apibrėžia daugiamačius sąryšius.

Dvejetainis medis

Pradedame nagrinėti sudėtingesnes dinamines struktūras, kurios apibrėžia daugiamačius sąryšius.

Pateiksime nepriklausomą dvejetainio medžio apibrėžimą.

Tarkime, turime elementų aibę D.

Dvejetainių medžių (angl. *binary tree*) aibei T priklauso:

- tuščioji aibė;

Dvejetainis medis

Pradedame nagrinėti sudėtingesnes dinamines struktūras, kurios apibrėžia daugiamačius sąryšius.

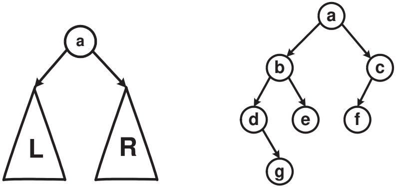
Pateiksime nepriklausomą dvejetainio medžio apibrėžimą.

Tarkime, turime elementų aibę D.

Dvejetainių medžių (angl. *binary tree*) aibei T priklauso:

- tuščioji aibė;
- viena viršūnė $a \in D$;

- visos aibės, sudarytos iš viršūnės $a \in D$, sujungtos su rūšiuota pora (L, R), kur L ir R yra dvejetainiai medžiai (žr. *a* paveikslą).
  Tada a vadinama medžio šaknimi, o L ir R – medžio T kairiuoju ir dešiniuoju pomedžiais.



a) b)

Dvejetainio medžio pavyzdys pateiktas *b* paveiksle.

Medžio viršūnes žymėsime $v_j \in V$.

Medžio viršūnes žymėsime $v_j \in V$.

Jei viršūnė v_j yra sujungta su kita viršūne v_k briauna $e_{jk} = (v_j, v_k) \in E$, tai v_k vadinama viršūnės v_j *vaiku*, o pati v_j – viršūnės v_k *tėvu*.

Medžio viršūnes žymėsime $v_j \in V$.

Jei viršūnė v_j yra sujungta su kita viršūne v_k briauna $e_{jk} = (v_j, v_k) \in E$, tai v_k vadinama viršūnės v_j *vaiku*, o pati v_j – viršūnės v_k *tėvu*.

Viršūnės, kurios neturi vaikų, vadinamos lapais.

Medžio viršūnes žymėsime $v_j \in V$.

Jei viršūnė v_j yra sujungta su kita viršūne v_k briauna $e_{jk} = (v_j, v_k) \in E$, tai v_k vadinama viršūnės v_j *vaiku*, o pati v_j – viršūnės v_k *tėvu*.

Viršūnės, kurios neturi vaikų, vadinamos lapais.

Medžio šaknis yra nulinio lygmens viršūnė. Tada k-tojo lygmens viršūnės vaiko lygmuo yra $(k+1)$-tasis.

Medžio viršūnes žymėsime $v_j \in V$.

Jei viršūnė v_j yra sujungta su kita viršūne v_k briauna $e_{jk} = (v_j, v_k) \in E$, tai v_k vadinama viršūnės v_j *vaiku*, o pati v_j – viršūnės v_k *tėvu*.

Viršūnės, kurios neturi vaikų, vadinamos lapais.

Medžio šaknis yra nulinio lygmens viršūnė. Tada k-tojo lygmens viršūnės vaiko lygmuo yra $(k+1)$-tasis.

Medžio aukštis yra lygus didžiausiam viršūnės lygmeniui.

Medžio viršūnes žymėsime $v_j \in V$.

Jei viršūnė v_j yra sujungta su kita viršūne v_k briauna $e_{jk} = (v_j, v_k) \in E$, tai v_k vadinama viršūnės v_j *vaiku*, o pati v_j – viršūnės v_k *tėvu*.

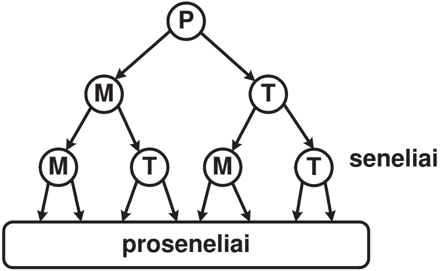
Viršūnės, kurios neturi vaikų, vadinamos lapais.

Medžio šaknis yra nulinio lygmens viršūnė. Tada k-tojo lygmens viršūnės vaiko lygmuo yra $(k+1)$-tasis.

Medžio aukštis yra lygus didžiausiam viršūnės lygmeniui.

Medžio briaunoms gali būti suteiktas svoris, toks medis vadinamas svertiniu.

Genealoginis medis



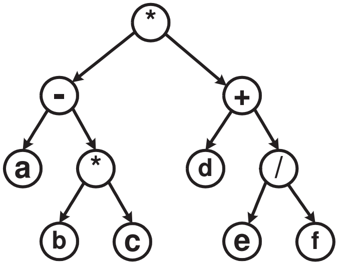
Infix aritmetinės išraiškos užrašymas dvejetainiame medyje

Nagrinėkime aritmetinę išraišką $(a - b * c) * (d + e/f)$.

Infix aritmetinės išraiškos užrašymas dvejetainiame medyje

Nagrinėkime aritmetinę išraišką $(a - b * c) * (d + e/f)$.

Ją užrašome dvejetainiame medyje (nereikia naudoti skliaustų)

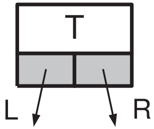


Pirmiausia apibrėžiame vieną atskirą *elementą* (angl. *node*), kurį sudaro informacinė dalis T ir dvi rodyklės, rodančios į kitą *elementą*

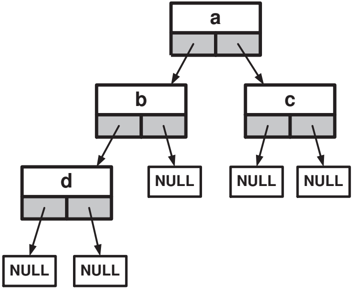
$$
\begin{aligned}
& struct\ node\,\{ \\
& \quad T\ data; \\
& \quad node * \ left; \\
& \quad node * \ right; \\
& \}
\end{aligned}
$$

Pirmiausia apibrėžiame vieną atskirą *elementą* (angl. *node*), kurį sudaro informacinė dalis T ir dvi rodyklės, rodančios į kitą *elementą*

$$
\begin{aligned}
&struct\ node\,\{ \\
&\quad T\ data; \\
&\quad node * \ left; \\
&\quad node * \ right; \\
&\}
\end{aligned}
$$



Sujungdami šiuos elementus, sudarome dvejetainį medį.



Visiškai subalansuotas dvejetainis medis

Daugelio algoritmų, realizuojančių medžio veiksmus, sudėtingumas priklauso nuo medžio aukščio. Todėl siekiame, kad didėjant medžio viršūnių skaičiui, jo aukštis didėtų kuo lėčiau.

Visiškai subalansuotas dvejetainis medis

Daugelio algoritmų, realizuojančių medžio veiksmus, sudėtingumas priklauso nuo medžio aukščio. Todėl siekiame, kad didėjant medžio viršūnių skaičiui, jo aukštis didėtų kuo lėčiau.

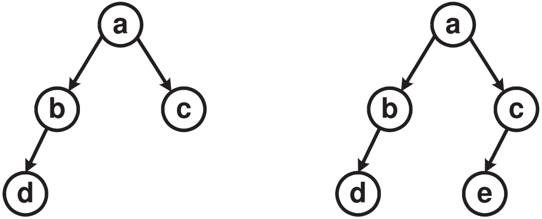
Tada reikia derinti kiekvieno elemento kairiojo ir dešiniojo pomedžių sudarymą.

Visiškai subalansuotas dvejetainis medis

Daugelio algoritmų, realizuojančių medžio veiksmus, sudėtingumas priklauso nuo medžio aukščio. Todėl siekiame, kad didėjant medžio viršūnių skaičiui, jo aukštis didėtų kuo lėčiau.

Tada reikia derinti kiekvieno elemento kairiojo ir dešiniojo pomedžių sudarymą.

Dvejetainis medis, kurio kiekvieno elemento kairiojo ir dešiniojo pomedžių elementų skaičiai skiriasi ne daugiau kaip vienu, vadinamas *visiškai subalansuotu* medžiu.



a)　　　　b)

Visiškai subalansuoti medžiai: a) $N = 4$, b) $N = 5$

Pateiksime algoritmą, kuris perkelia duomenis iš masyvo ar rinkmenos į visiškai subalansuotą dvejetainį medį. Tai rekursinis algoritmas.

$$
\begin{aligned}
& node * \ balancedTree(int\ N)\{ \\
& \quad if\ (N == 0)\ return\ (NULL); \\
& \quad nL = N/2;\ nR = N - nL - 1; \\
& \quad x = read(); \\
& \quad node * \ Node = new(node); \\
& \quad Node{-}{>}\ data = x; \\
& \quad Node{-}{>}\ left = balancedTree(nL); \\
& \quad Node{-}{>}\ right = balancedTree(nR); \\
& \quad return\ (Node); \\
& \}
\end{aligned}
$$

Dvejetainio medžio viršūnių apėjimo algoritmai

Susipažinsime su trimis svarbiais dvejetainio medžio viršūnių aplankymo algoritmais. Visi jie rekursiniai, o skiriasi tik pomedžių aplankymo tvarka.

Dvejetainio medžio viršūnių apėjimo algoritmai

Susipažinsime su trimis svarbiais dvejetainio medžio viršūnių aplankymo algoritmais. Visi jie rekursiniai, o skiriasi tik pomedžių aplankymo tvarka.

Jei medyje saugome aritmetinę išraišką, tai šiais algoritmais randame tris pagrindines aritmetinės išraiškos formas: *prefix*, *infix* ir *postfix*, todėl taip vadinsime ir atitinkamus medžio apėjimo algoritmus.

Prefix algoritmas. Pirmiausia aplankome viršūnę-šaknį, paskui jos kairįjį pomedį ir vėliausiai dešinįjį pomedį.

```
preOrder (node* v)
begin
  (1)  if  (v != NULL ) then
  (2)        P(v);
  (3)        preOrder(v->left);
  (4)        preOrder(v->right);
       end if
end preOrder
```

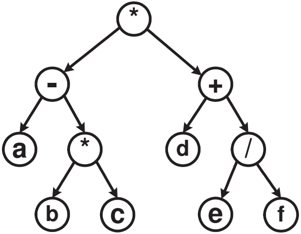
Infix algoritmas. Pirmiausia aplankome kairįjį pomedį, paskui viršūnę-šaknį ir vėliausiai dešinįjį pomedį

```
inOrder (node* v)
begin
  (1)  if  (v != NULL ) then
  (2)        inOrder(v->left);
  (3)        P(v);
  (4)        inOrder(v->right);
       end if
end inOrder
```

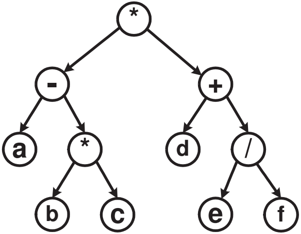
Postfix algoritmas. Pirmiausia aplankome kairįjį pomedį, paskui dešinįjį pomedį ir vėliausiai viršūnę-šaknį.

```
postOrder (node* v)
begin
  (1)  if  (v != NULL ) then
  (2)        postOrder(v->left);
  (3)        postOrder(v->right);
  (4)        P(v);
       end if
end postOrder
```

Pritaikę šiuos algoritmus atspausdiname aritmetinės išraiškos $(a - b * c) * (d + e/f)$ tris skirtingas užrašymo formas:

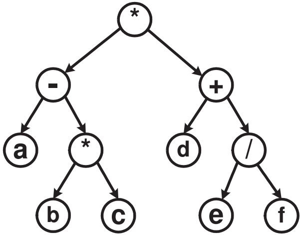


Pritaikę šiuos algoritmus atspausdiname aritmetinės išraiškos $(a - b * c) * (d + e/f)$ tris skirtingas užrašymo formas:



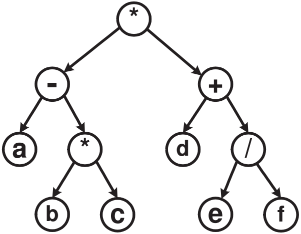
- *Prefix* forma: $* - a * bc + d/ef$,

Pritaikę šiuos algoritmus atspausdiname aritmetinės išraiškos $(a - b * c) * (d + e/f)$ tris skirtingas užrašymo formas:



- *Prefix* forma: $* - a * bc + d/ef$,
- *Infix* forma: $a - b * c * d + e/f$,

Pritaikę šiuos algoritmus atspausdiname aritmetinės išraiškos $(a - b * c) * (d + e/f)$ tris skirtingas užrašymo formas:



- *Prefix* forma: $* - a * bc + d/ef$,
- *Infix* forma: $a - b * c * d + e/f$,
- *Postfix* forma: $abc * -def / + *$.

Dvejetainis paieškos medis

Labai dažnai tenka informaciją saugoti, rūšiuoti ir ieškoti. Sprendžiant šiuos uždavinius, svarbūs yra paieškos medžiai (angl. *binary search tree*).

Dvejetainis paieškos medis

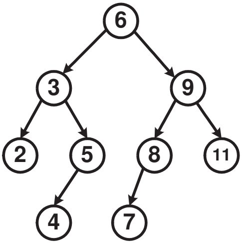
Labai dažnai tenka informaciją saugoti, rūšiuoti ir ieškoti. Sprendžiant šiuos uždavinius, svarbūs yra paieškos medžiai (angl. *binary search tree*).

Tokio medžio kiekvienoje viršūnėje esantis elementas yra didesnis už kairiojo pomedžio elementus ir nedidesnis už dešiniojo pomedžio elementus.

Dvejetainis paieškos medis

Labai dažnai tenka informaciją saugoti, rūšiuoti ir ieškoti. Sprendžiant šiuos uždavinius, svarbūs yra paieškos medžiai (angl. *binary search tree*).

Tokio medžio kiekvienoje viršūnėje esantis elementas yra didesnis už kairiojo pomedžio elementus ir nedidesnis už dešiniojo pomedžio elementus.



Naujos viršūnės įterpimas.

Naujoji viršūnė turi būti įterpta taip, kad, ir atlikus šį veiksmą, medis išliktų paieškos medžiu.

Naujos viršūnės įterpimas.

Naujoji viršūnė turi būti įterpta taip, kad, ir atlikus šį veiksmą, medis išliktų paieškos medžiu.

Nebandome kontroliuoti medžio aukščio, tai įterpimo algoritme surandame atitinkamą medžio šaką ir sukuriame naują viršūnę-lapą.

Naujos viršūnės įterpimas.

Naujoji viršūnė turi būti įterpta taip, kad, ir atlikus šį veiksmą, medis išliktų paieškos medžiu.

Nebandome kontroliuoti medžio aukščio, tai įterpimo algoritme surandame atitinkamą medžio šaką ir sukuriame naują viršūnę-lapą.

Jei pradinis medis yra tuščias, tai ši viršūnė tampa jo šaknimi.

```
insert (node* tree, node* v)
begin
  (1)  while  ( tree != NULL ) do
  (2)             if ( v->data < tree->data ) then
  (3)                tree = tree->left;
                  else
  (4)                tree = tree->right;
                  end if
       end do
  (5)  tree = v;
  (6)  v->left = NULL;
  (7)  v->right = NULL;
end insert;
```

```
insert (node* tree, node* v)
begin
  (1)  while  ( tree != NULL ) do
  (2)           if ( v->data < tree->data ) then
  (3)              tree = tree->left;
                else
  (4)              tree = tree->right;
                end if
       end do
  (5)  tree = v;
  (6)  v->left = NULL;
  (7)  v->right = NULL;
end insert;
```

Pavyzdys: sudarykite paieškos medį, kai turime duomenų seką:

6, 9, 3, 5, 11, 4, 8, 2, 7.

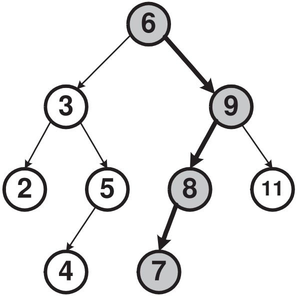
Viršūnės paieškos algoritmas

```
node* find (node* tree, T inf)
begin
 (1)   while  ( tree != NULL && tree->data != inf ) do
 (2)             if ( tree->data < inf ) then
 (3)                tree = tree->right;
                 else
 (4)                tree = tree->left;
                 end if
       end do
 (5)   return tree;
end find
```

Viršūnės paieškos algoritmas

```
node* find (node* tree, T inf)
begin
 (1)  while  ( tree != NULL && tree->data != inf ) do
 (2)           if ( tree->data < inf ) then
 (3)              tree = tree->right;
               else
 (4)              tree = tree->left;
               end if
      end do
 (5)  return tree;
end find
```

Jei tokio elemento medyje nėra, tai procedūra grąžina nuorodą į tuščią viršūnę (`NULL` nuorodą).



Viršūnės, kurioje saugomas skaičius 7, paieškos kelias.

Viršūnės šalinimas

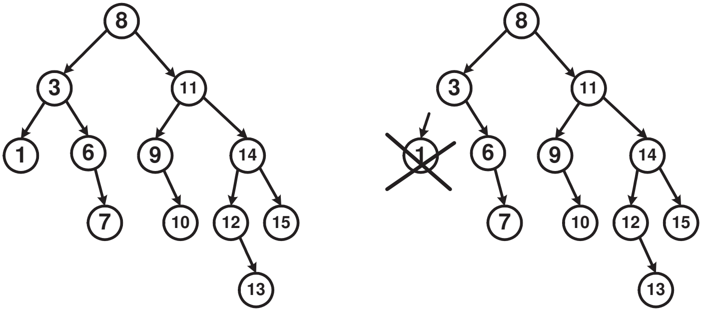
Pašalinti viršūnę iš paieškos medžio yra sudėtingiau – reikia garantuoti, kad ir atlikus šį veiksmą turėsime dvejetainį pieškos medį.

Viršūnės šalinimas

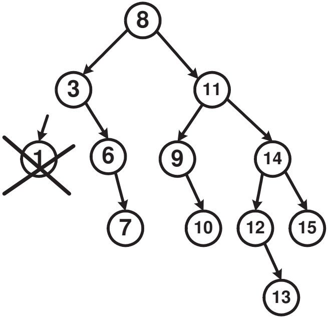
Pašalinti viršūnę iš paieškos medžio yra sudėtingiau – reikia garantuoti, kad ir atlikus šį veiksmą turėsime dvejetainį pieškos medį.

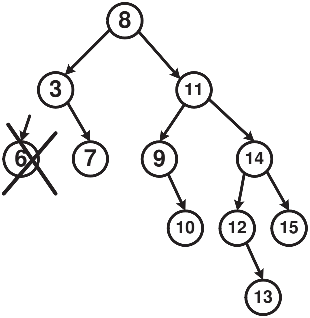
Panagrinėsime tik pavyzdžius, algoritmai yra aprašyti vadovėliuose, jų detales galėsite išnagrinėti savarankiškai, kai spręsdami taikomuosius uždavinius naudosite šią duomenų struktūrą.

Imkime dvejetainį pieškos medį ir iš jo pašalinkime viršūnę–lapą 1.

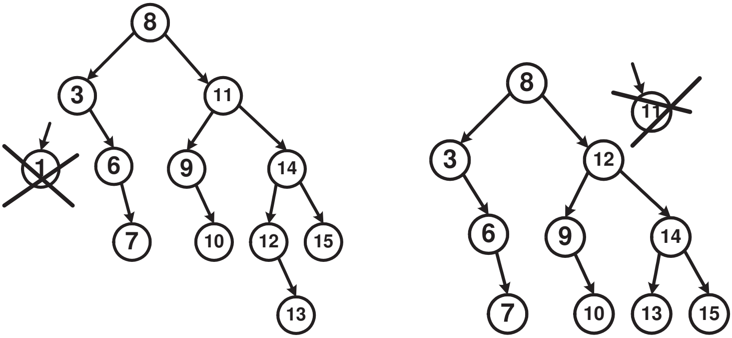


Iš gautojo dvejetainio medžio pašalinkime viršūnę, kuri turi tik vieną vaiką – 6.





Iš to pačio dvejetainio medžio pašalinkime viršūnę 11.



Paieškos algoritmo sudėtingumas

Kiek užtruks informacijos paieška dvejetainio medžio duomenų struktūroje?

Paieškos algoritmo sudėtingumas

Kiek užtruks informacijos paieška dvejetainio medžio duomenų struktūroje?

Bazine algoritmo operacija laikome dviejų raktų palyginimą.

Paieškos algoritmo sudėtingumas

Kiek užtruks informacijos paieška dvejetainio medžio duomenų struktūroje?

Bazine algoritmo operacija laikome dviejų raktų palyginimą.

Elemento paieškos sudėtingumas priklauso nuo dvejetainio medžio aukščio.

Paieškos algoritmo sudėtingumas

Kiek užtruks informacijos paieška dvejetainio medžio duomenų struktūroje?

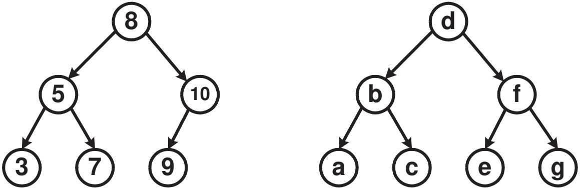
Bazine algoritmo operacija laikome dviejų raktų palyginimą.

Elemento paieškos sudėtingumas priklauso nuo dvejetainio medžio aukščio.

Aišku, kad bet kuriam elementų išsidėstymui geriausio atvejo sudėtingumas yra lygus vienetui – ieškomasis elementas yra saugomas medžio šaknyje.

Palankiausias variantas, kai informacija yra saugoma idealiai subalansuotame paieškos medyje.

Palankiausias variantas, kai informacija yra saugoma idealiai subalansuotame paieškos medyje.



a) b)

Idealiai subalansuoti dvejetainiai paieškos medžiai: a) skaičių aibė $\{3, 5, 7, 8, 9, 10\}$, b) raidžių aibė $\{a, b, c, d, e, f, g\}$

Jeigu saugome N elementų, tai subalansuoto medžio aukštis yra $H = \log(N+1) - 1$.

Jeigu saugome N elementų, tai subalansuoto medžio aukštis yra $H = \log(N+1) - 1$.

Taigi net ir blogiausio atvejo paieškos sudėtingumas yra

$$W_b = \log(N+1).$$

Jeigu saugome N elementų, tai subalansuoto medžio aukštis yra $H = \log(N+1) - 1$.

Taigi net ir blogiausio atvejo paieškos sudėtingumas yra

$$W_b = \log(N+1).$$

Dažniausiai svarbu žinoti vidutinį (tikėtinąjį) paieškos algoritmo sudėtingumą, kai vienodai dažnai tikrinami visi N elementai. Gauname tokį įvertį

$$W_v = \frac{1}{N} \sum_{i=0}^{H} 2^i (i+1) = \log N + \mathcal{O}(1).$$

Nepalankiausiu atveju dvejetainis paieškos medis išsigimsta į tiesinį sąrašą. Taip bus, kai duomenis skaitome iš rinkmenos rakto didėjimo ar mažėjimo tvarka.

Nepalankiausiu atveju dvejetainis paieškos medis išsigimsta į tiesinį sąrašą. Taip bus, kai duomenis skaitome iš rinkmenos rakto didėjimo ar mažėjimo tvarka.

Tokiame paieškos medyje vidutinis paieškos algoritmo sudėtingumas yra

$$W_v = \frac{1}{N}\sum_{i=1}^{N} i = \frac{(N+1)}{2} = \frac{N}{2} + \mathcal{O}(1)\,.$$

Nepalankiausiu atveju dvejetainis paieškos medis išsigimsta į tiesinį sąrašą. Taip bus, kai duomenis skaitome iš rinkmenos rakto didėjimo ar mažėjimo tvarka.

Tokiame paieškos medyje vidutinis paieškos algoritmo sudėtingumas yra

$$W_v = \frac{1}{N} \sum_{i=1}^{N} i = \frac{(N+1)}{2} = \frac{N}{2} + \mathcal{O}(1)\,.$$

Blogiausiojo atvejo sudėtingumas $W_b = N$.

Iš viso galime sudaryti $N!$ skirtingų paieškos medžių. Tai milžiniškas skaičius, net kai turime nedaug elementų, pvz., $N = 20$, nes remiantis Stirlingo formule

$$20! \geqslant \sqrt{40\pi} \left(\frac{20}{e} \right)^{20} > 2 \cdot 10^{18} \,.$$

Iš viso galime sudaryti $N!$ skirtingų paieškos medžių. Tai milžiniškas skaičius, net kai turime nedaug elementų, pvz., $N = 20$, nes remiantis Stirlingo formule

$$20! \geqslant \sqrt{40\pi} \left(\frac{20}{e} \right)^{20} > 2 \cdot 10^{18} \, .$$

Pažymėkime a_N paieškos algoritmo vidutinį sudėtingumą, kai nagrinėjame visus galimus paieškos medžius ir kai vienodai dažnai tikrinami visi N medžio elementai.

Iš viso galime sudaryti $N!$ skirtingų paieškos medžių. Tai milžiniškas skaičius, net kai turime nedaug elementų, pvz., $N = 20$, nes remiantis Stirlingo formule

$$20! \geqslant \sqrt{40\pi} \left(\frac{20}{e} \right)^{20} > 2 \cdot 10^{18} \,.$$

Pažymėkime a_N paieškos algoritmo vidutinį sudėtingumą, kai nagrinėjame visus galimus paieškos medžius ir kai vienodai dažnai tikrinami visi N medžio elementai.

Šiame pavyzdyje puikiai matysime teorinės analizės galimybes. Atlikę pakankamai paprastus veiksmus, galėsime įvertinti šio milžiniško variantų skaičiaus vidutinį sudėtingumą.

Nesunku įrodyti, kad teisinga tokia rekurentinė lygtis

$$a_N = 1 + \frac{2}{N^2} \sum_{j=1}^{N-1} j\, a_j, \quad a_1 = 1.$$

Nesunku įrodyti, kad teisinga tokia rekurentinė lygtis

$$a_N = 1 + \frac{2}{N^2} \sum_{j=1}^{N-1} j\, a_j, \quad a_1 = 1.$$

Žinant šią lygtį, galime skaičiuoti paieškos algortmo vidutinį sudėtingumą net ir labai dideliam elementų skaičiui N.

Nesunku įrodyti, kad teisinga tokia rekurentinė lygtis

$$a_N = 1 + \frac{2}{N^2} \sum_{j=1}^{N-1} j\, a_j, \quad a_1 = 1.$$

Žinant šią lygtį, galime skaičiuoti paieškos algortmo vidutinį sudėtingumą net ir labai dideliam elementų skaičiui N.

Jeigu pratęsime teorinę analizę, tai galime sprendinį užrašyti ir analizine forma.

Nesunku įrodyti, kad teisinga tokia rekurentinė lygtis

$$a_N = 1 + \frac{2}{N^2} \sum_{j=1}^{N-1} j\, a_j, \quad a_1 = 1.$$

Žinant šią lygtį, galime skaičiuoti paieškos algortmo vidutinį sudėtingumą net ir labai dideliam elementų skaičiui N.

Jeigu pratęsime teorinę analizę, tai galime sprendinį užrašyti ir analizine forma.

Apibrėžkime harmoninės eilutės dalinės sumos funkciją

$$H_N = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{N}\,.$$

Tada nesunku patikrinti, kad lygties sprendinys yra užrašomas taip:

$$a_N = 2\frac{N+1}{N} H_N - 3\,.$$

Atsižvelgę į tai, kad $H_N = \ln N + \gamma + \mathcal{O}\left(\frac{1}{N^2}\right)$, gauname, kad paieškos algoritmo vidutinis sudėtingumas yra

$$a_N = 2\ln N + \mathcal{O}(1)\,.$$

Atsižvelgę į tai, kad $H_N = \ln N + \gamma + \mathcal{O}\left(\frac{1}{N^2}\right)$, gauname, kad paieškos algoritmo vidutinis sudėtingumas yra

$$a_N = 2 \ln N + \mathcal{O}(1)\,.$$

Priminsime, kad idealiai subalansuoto paieškos medžio atveju $W_v = \log N$. Kadangi

$$\frac{2 \ln N}{\log N} = 2 \ln 2 = 1.386,$$

tai vidutinis paieškos algoritmo sudėtingumas dvejetainiame paieškos medyje yra tik 1.386 karto didesnis nei idealiai subalansuotame medyje.