2.1. Paprasčiausios tiesinės duomenų struktūros

Šiame skyriuje susipažinsime su naudingomis duomenų struktūromis, kurias gauname iš vienakrypčio tiesinio sąrašo apribodami pastarojo veiksmų galimybes. Atkreipsime skaitytojų dėmesį į pastarąjį faktą: naujosios duomenų struktūros yra efektyvios ne todėl, kad išplečiame dinaminio sąrašo galimybes (vėliau įsitikinsime, kad ir šis kelias yra labai produktyvus), bet, atvirkčiai, pasinaudosime griežčiau apibrėžtomis jų veiksmų savybėmis.

2.1.1. Stekas

Stekas (angl. *stack*) yra labai dažnai naudojama duomenų struktūra. Ją apibūdina principas "paskutinis įeina, pirmasis išeina" (angl. *LIFO - Last In, First Out*). Elementai įrašomi ir šalinami iš sąrašo pradžios.

Tokią taisyklę naudojame ir buityje, pavyzdžiui pažvelkime į padėklų krūvą valgykloje: padėklas, kuris paskutinis padedamas ant viršaus, bus paimtas pirmas, o apatinis padėklas, kuris buvo padėtas pirmas, bus paimtas paskutinis.

Kaip ir kitoms duomenų struktūroms turime apibrėžti tokius steko veiksmus:

- sukurti tuščią steką;
- įterpti elementą į steką;
- patikrinti, ar stekas yra tuščias;
- perskaityti viršutinio elemento reikšmę;
- išimti viršutinį elementą iš steko;

Priklausomai nuo steko realizavimo būdo, prieš įterpiant naują elementą, gali tekti patikrinti ar stekas nėra pilnas, o prieš išimant elementą iš steko tenka patikrinti ar jis nėra tuščias.

Nagrinėsime du steko realizavimo būdus.

Steko realizacija masyve

Steką apibrėžiame panaudodami įrašo duomenų struktūrą: jį sudaro du laukai – elementų masyvas ir viršūnės rodyklė.

```
struct stack{
  T data[N];
  int top=0;
}
```

Kadangi masyvo dydis yra fiksuotas, tai tokiame steke galėsime saugoti ne daugiau nei N elementų. Tačiau tokia situacija yra būdinga daugeliui taikomųjų uždavinių, aprašomų steko duomenų struktūra. Pavyzdžiui padėklai saugomi stove, kuriame galime padėti tik numatytą jų skaičių, į pistoleto apkabą taip pat galime įdėti tik fiskuotą skaičių šovinių.

Tuščiame steke viršūnės rodyklė *top* yra lygi nuliui. Stekas yra pilnas, jei $top = N$.

Elementas įterpiamas į steką tokia procedūra

```
push(T e){
  if (top < N){
    data[top] = e;
    top += 1;
  }
}
```

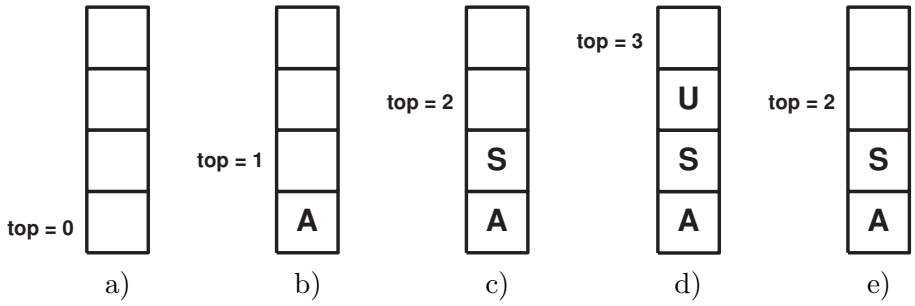
Šioje realizacijoje nieko nedarome, jei stekas jau pilnas. Todėl vartotojas turi pats pasirūpinti steko pilnumo sąlygos tikrinimu

```
int stackFull(){
  int full = 0;
  if (top ==  N)
    full = 1;
return (full);
}
```

2.1 paveiksle pavaizduotas stekas, į kurį įterpiamos raidės A, S, U, o po to raidė U yra pašalinama.

Elementas išimamas iš steko `pop`() procedūros pagalba

```
T pop(){
  if (top > 0){
    top -= 1;
    return (data[top]);
  }
}
```



2.1 pav. Steko realizavimas masyve: a) tuščias masyvas, b) push(A), c) push(S), d) push(U), e) pop().

Ir šioje procdūroje nedarome nieko, jei stekas yra tuščias. Taigi vartotojas pats turi patikrinti ar stekas yra tuščias, tai atlieka procedūra

```
int stackEmpty(){
  int empty = 0;
  if (top ==  0)
    empty = 1;
return (empty);
}
```

Procedūra readTop tik perskaito viršutinio elemento reikšmę, bet jo neišima iš steko:

```
T readTop(){
  if (top > 0)
    return (data[top-1]);
}
```

Steko realizacija dinaminiame sąraše

Naudodami tiesinį dinaminį sąrašą galime saugoti bet kokio ilgio steką, visada bus išskirta tik tiek kompiuterio atminties, kiek yra elementų. Įterpimo į steką veiksmas push vykdomas tiesinio sąrašo procedūra insertFront, o elemento šalinimas iš steko pop atliekmas panaudojant deleteFront veiksmą. Kaip pavyzdį pateiksime push procedūrą:

```
push(T e) {
  node *v;
  v = new node;
  (*v).T = e;
  if ( listStart == NULL) // Pirmas elementas
    (*v).next = NULL ;
  else
    (*v).next = (*listStart).next;
  listStart = v;
}
```

Kaip matome steko duomenų struktūra gali būti realizuota keliais būdais. Tokioje situacijoje labai patogu naudoti objektinio programavimo technologiją, pvz. sukurti C++ specialią steko klasę. Šios klasės vartotojas gali ir nežinoti, kaip realizuotas pats stekas, o jo programos veikimo teisingumas (bet ne efektyvumas) nepriklauso nuo steko realizavimo būdo.

2.1.2. Steko taikymai

Stekai labai plačiai naudojami įvairių algoritmų realizacijose, informatikoje, kompiuterių programų vykdyme. Pavyzdžiui iš programos teksto sudaromas vykdomųjų komandų stekas.

Šiame paragrafe susipažinsime su trijų uždavinių sprendimu: rekursijos algoritmų realizavimu, aritmetinių išraiškų atvaizdavimu *postfix* forma ir šia forma užrašytų aritmetinių išraiškų skaitinės reikšmės skaičiavimu.

Aritmetinės išraiškos vaizdavimas postfix forma

Dviejų skaičių a ir b sumą dažniausiai žymime $a + b$, toks užrašas yra vadinamas aritmetinės išraiškos *infix* forma. Čia aritmetinis veiksmas yra užrašomas tarp dviejų operandų.

Kalkuliatoriuose aritmetinės išraiškos užrašomos *prefix* forma, kai iš pradžių rašomas aritmetinis veiksmas, o po to operandai. Sumos prefix forma yra $+ab$.

Šiuolaikiniuose kompiuteriuose aritmetinės išraiškos vaizduojamos *postfix* forma, kai iš pradžių rašomi operandai, o po to aritmetinis veiksmas. Sumos postfix forma yra $ab+$.

Prefix ir postfix formos yra labai patogios todėl, kad nereikia naudoti skliaustų, norint nurodyti veiksmų atlikimo eiliškumą.

2.1 pavyzdys. Aritmetinių išraiškų postfix formos analizė. Nagrinėkime aritmetinę išraišką $a+b*c$. Užrašysime jos postfix formą

$$\begin{aligned} a+b*c &\longrightarrow a+(b*c) \longrightarrow a+(b\,c*) \\ &\longrightarrow a\,(b\,c*)+ \longrightarrow a\,b\,c\,*\,+\,. \end{aligned}$$

Kitos aritmetinės išraiškos $(a+b)*c$ postfix forma yra skirtinga ir jai užrašyti nereikia skliaustų, kurie buvo būtini infix formos išraiškoje

$$(a+b)*c \longrightarrow (a\,b+)*c \longrightarrow (a\,b+)\,c* \longrightarrow a\,b\,+\,c\,*\,.$$

Priminsime aritmetinių veiksmų prioritetus:

- Aukščiausią prioritetą turi kėlimo laipsniu veiksmas, kurį žymėsime simboliu $\wedge$:
$$a^b = a \wedge b\,.$$
Šis veiksmas yra atliekamas iš dešinės į kairę, t.y. naujasis laipsnio kėlimas yra aukštesnio prioriteto veiksmas
$$a \wedge b \wedge c = a \wedge (b \wedge c)\,.$$

- Žemesnį prioritetą turi daugybos ir dalybos veiksmai. Jų eiliškumas yra iš kairės į dešinę
$$a * b/c = (a * b)/c\,.$$

- Sumavimo ir atimties veiksmų prioritetas yra žemiausias, jų eiliškumas irgi iš kairės į dešinę
$$a - b + c = (a - b) + c\,.$$

- Skliaustuose esantis reiškinys interpretuojamas kaip vienas operandas ir apskaičiuojamas prieš kitus veiksmus (jo prioritetas yra aukštesnis už bet kurią aritmetinę operaciją).

Dabar pateiksime algoritmą, kuriuo *infix* formos išraišką konvertuojame į *postfix* formą. Jį realizuodami esminiai remsimės *steko* savybėmis.

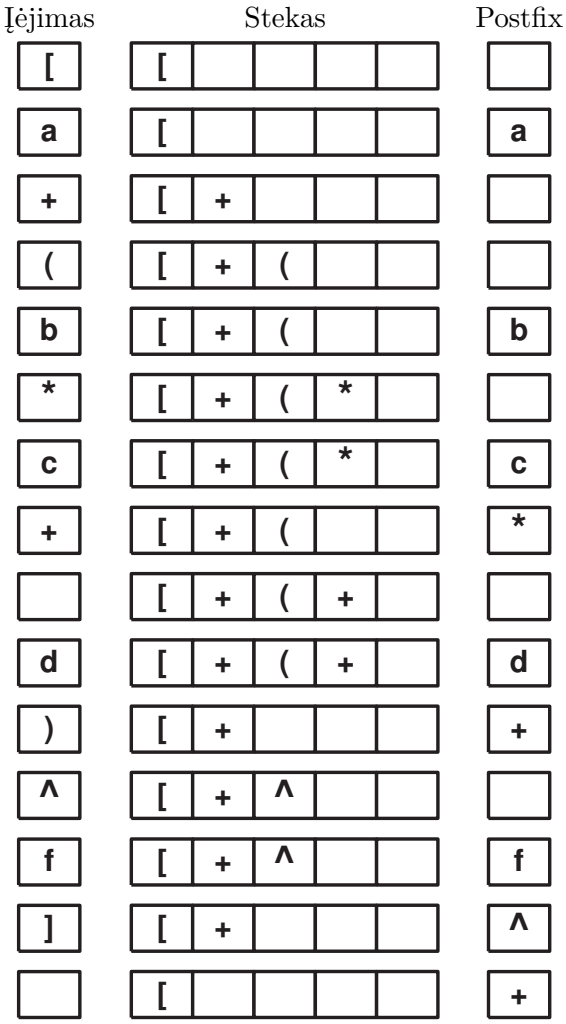
Infix išraiškos užrašymas postfix forma

```
(1)  Infix išraišką papildome pradžios " [ " ir pabaigos " ] " simboliais.
(2)  s = readDataFile();
(3)  select case (s):
     (4)  case ( "[",  "(" )
               (5)  push(s);
     (6)  case ( s  yra operandas  )
               (7)  print(s);
     (8)  case ( ∧,  *,  /,  +,  − )
               (9)  h = readTop();
               (10) select case (h):
                    (11) case ( "[",  "(",  h ≺ s )
                              (12) push(s);
                    (12) case ( s ≺= h )
                              (13) h = pop();
                              (14) print(h);
                              (15) goto 9;
     (16) case ( ")" )
               (17) h = pop();
               (18) while ( h != "(" )
                         (19) print(h);
                         (20) h = pop();
     (21) case ( "]" )
               (22) h = pop();
               (23) while ( h != "[" )
                         (24) print(h);
                         (25) h = pop();
(26) if (s != "]") goto 2;
```

2.2 pavyzdys. Infix aritmetinės išraiškos užrašymas postfix forma. Nagrinėkime *infix* forma užrašytą aritmetinę išraišką $a + (b * c + d) \wedge f$. Panaudodami pateiktąjį algoritmą užrašysime ją *postfix* forma. Iš pradžių šią išraišką papildome pradžios ir pabaigos požymiais $[a + (b * c + d) \wedge f]$. Tolimesnė skaičiavimo eiga pateikta 2.2 paveiksle.

Taigi aritmetinės išraiškos $a + (b * c + d) \wedge f$ postfix forma yra

$$abc * d + f \wedge + .$$



2.2 pav. Aritmetinės išraiškos konvertavimas į postfix formą.

Postfix formos išraiškos skaitinės reikšmės skaičiavimas

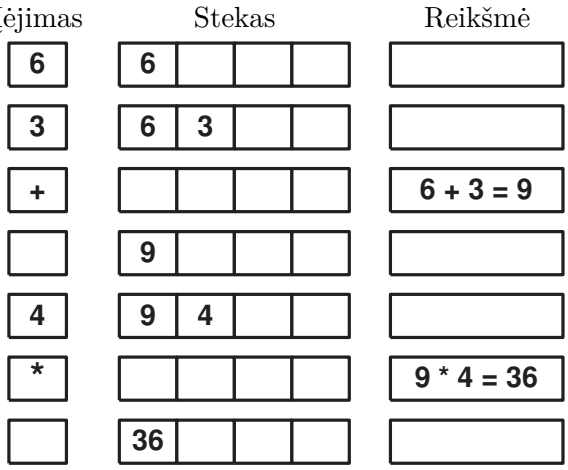
Dabar sudarysime algoritmą, kuriuo apskaičiuojame aritmetinės išraiškos, užrašytos *postfix* forma, reikšmę. Vėl naudosime steko duomenų struktūrą.

Pagrindinis algoritmo sudarymo principas yra paprastas: naujos aritmetinės operacijos operandai yra prieš tai apskaičiuotos dviejų aritmetinių veiksmų reikšmės.

Postfix išraiškos reikšmės skaičiavimas

```
(1) s = readDataFile();
(2) if (s yra skaičius)
        (3) push(s);
(4) else
        (5) b = pop();
        (6) a = pop();
        (7) c = a op(s) b;
        (8) push(c);
(9) if (yra duomenų) goto 1;
```

2.3 pavyzdys. Aritmetinės išraiškos reikšmės skaičiavimas. Raskime *postfix* išraiškos $6\ 3 + 4*$, kuri apibrėžia aritmetinę išraišką $(6 + 3) * 4$, reikšmę. Skaičiavimo eiga pateikta 2.3 paveiksle.



2.3 pav. Aritmetinės išraiškos $6\ 3 + 4*$ reikšmės skaičiavimas.

2.2. Dvejetainis medis

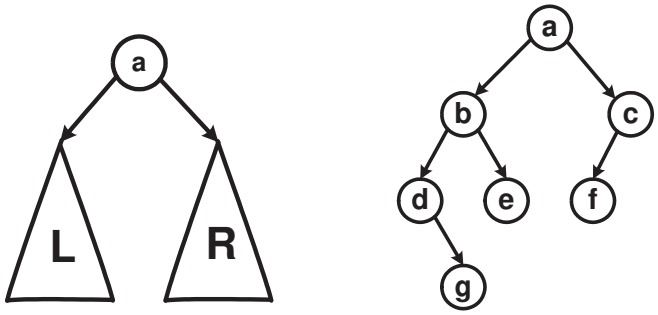
Pradedame nagrinėti sudėtingesnes dinamines struktūras, kurios apibrėžia daugiamačius sąryšius.

Medį galime apibrėžti kaip atskirą bendresnės duomenų struktūros, grafo, atvejį – tai ciklų neturintis orientuotas grafas. Tačiau šiame paragrafe pateksime nepriklausomą dvejetainio medžio apibrėžimą.

Tarkime turime elementų aibę D. *Dvejetainių medžių* (angl. *binary tree*) aibei T priklauso

- tuščia 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. 2.4*a* paveikslą). Tada a vadinama medžio *šaknimi*, o L ir R – medžio T kairiuoju ir dešiniuoju pomedžiais.

Dvejetainio medžio pavyzdys pateiktas 2.4*b* paveiksle.



a) b)

2.4 pav. Dvejetainio medžio schema ir pavyzdys

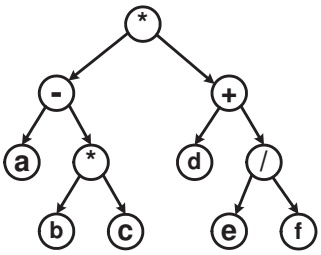
Taigi į bet kurią medžio viršūnę, išskyrus šaknį, įeina viena briauna, o išeina ne daugiau kaip dvi. Jeigu dvi medžio viršūnes galime sujungti keliu, tai šis kelias yra vienintelis.

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-ojo lygmens viršūnės vaiko lygmuo yra $(k+1)$-asis. Medžio *aukštis* yra lygus didžiausiam viršūnės lygmeniui.

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

Dvejetainis medis yra *sutvarkytas*, kai jo viršūnėms suteikti eilės numeriai.



2.5 pav. Aritmetinė išraiška $(a - b * c) * (d + e/f)$

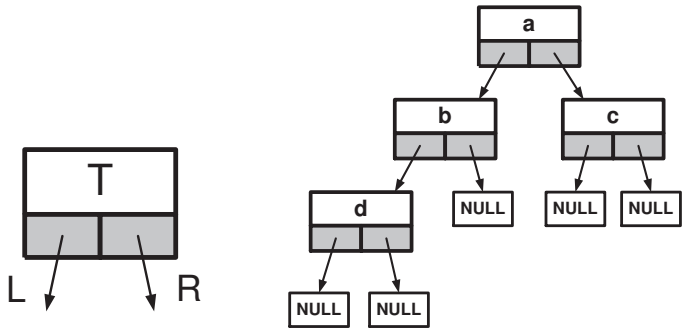
2.4 pavyzdys. Infix išraiškos įrašymas dvejetainiame medyje. Parodysime, kaip medyje galime saugoti informaciją apie aritmetinę išraišką. Vėliau įsitikinsime, kad egzistuoja paprasti medžio viršūnių aplankymo algoritmai, leidžiantys *infix* formos išraišką užrašyti *prefix* ar *postfix* forma. Nagrinėkime aritmetinę išraišką $(a-b*c)*(d+e/f)$. Ją užrašome į dvejetainį medį, kuris pateiktas 2.5 paveiksle.

2.2.1. Medžio veiksmai

Ir šiai duomenų struktūrai turime apibrėžti pagrindinius veiksmus:

- sukurti tuščią medį;
- perkelti duomenis iš failo į medį;
- įterpti ar pašalinti elementą;
- apeiti medžio viršūnes, surasti reikiamą viršūnę.

Pirmiausia apibrėžiame vieną atskirą *elementą* (angl. *node*), kurį sudaro informacinė dalis T (ją patogu realizuoti kaip įrašą) ir dvi rodyklės, rodančios į *elemento* tipo objektą (žr. 2.6 *a* pav.):



2.6 pav. a) Atskiras medžio elementas, b) dvejetainis medis.

```
struct node{
  T data;
  node* left;
  node* right;
}
```

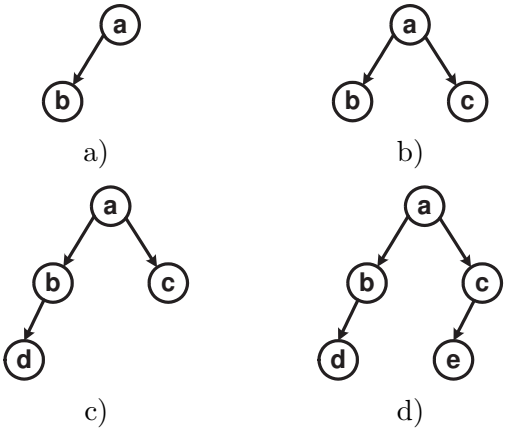
Po to sujungdami šiuos elementus sudarome dvejetainį medį (žr. 2.6 *b* pav.)

Visiškai subalansuotas dvejetainis medis

Daugelio algoritmų, realizuojančių medžio veiksmus, sudėtingumas priklauso nuo medžio aukščio. Todėl siekiame, kad didėjant medžio viršūnių skaičiui jo aukštis didėtų kiek galima lėčiau. Akivaizdu, kad tada reikia derinti medžio kairiojo ir dešiniojo pomedžių sudarymą.

Dvejetainis medis, kurio kiekvieno elemento kairiojo ir dešiniojo pomedžių elementų skaičiai skiriasi ne daugiau nei vienetu, yra vadinamas *visiškai subalansuotu*. 2.7 paveiksle pateikti tokių medžių pavyzdžiai.

Pateiksime algoritmą, kuris perkelia duomenis iš masyvo ar failo į visiškai subalansuotą dvejetainį medį. Tai rekursinis algoritmas ir toks metodas yra būdingas daugeliui algoritmų, realizuojančių medžio veiksmus.



2.7 pav. Visiškai subalansuoti dvejetainiai medžiai: a) N=2, b) N=3, c) N=4, d) N=5.

```
node* balancedTree(int N){
  if ( N == 0 ) return(NULL);
  nL = N/2; nR = N - nL -1;
  x = read();
  node* Node= new(node);
  (*Node).data = x;
  (*Node).left = balancedTree(nL);
  (*Node).right = balancedTree(nR);
  return(Node);
}
```

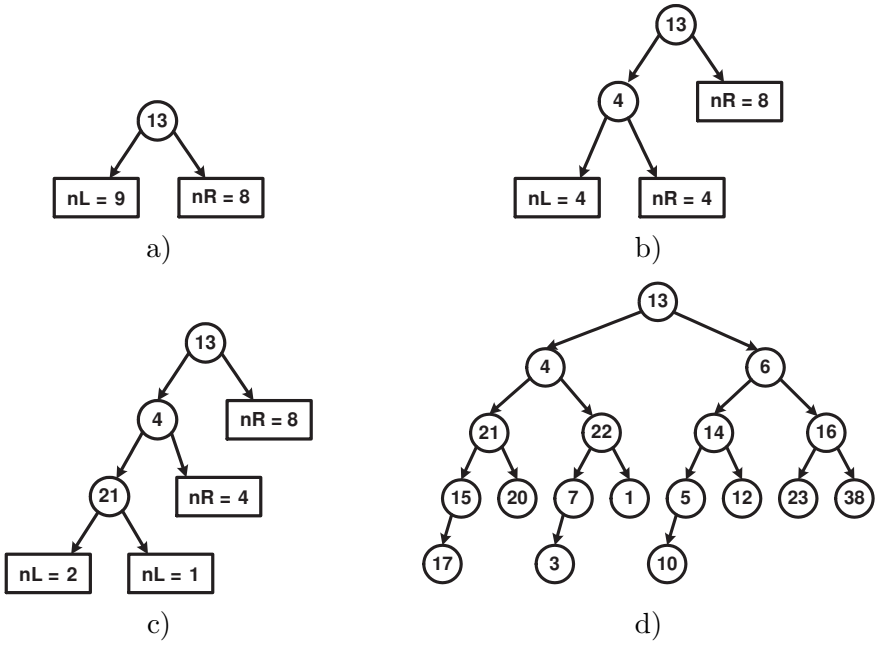
2.5 pavyzdys. Dvejetainio medžio sudarymas. Faile F saugome $N = 18$ skaičių:

13, 4, 21, 15, 17, 20, 22, 7, 3, 1, 6, 14, 5, 10, 12, 16, 29, 38.

Perkelsime juos į visiškai subalansuotą dvejetainį medį. Medžio sudarymo eiga pavaizduota 2.8 paveiksle.

Dvejetainio medžio viršūnių apėjimo algoritmai

Medis yra išsišakojanti duomenų struktūra, todėl egzistuoja daug skirtingų visų jo viršūnių aplankymo variantų. Į bet kurią viršūnę galime patekti tik iš jos tėvo viršūnės.



2.8 pav. Visiškai subalansuoto dvejetainio medžio sudarymas.

Susipažinsime su trim labai svarbiais dvejetainio medžio viršūnių aplankymo algoritmais. Visi jie rekursiniai, o skiriasi tik pomedžių aplankymo tvarka. Vėliau įsitikinsime, kad jei medyje saugome aritmetinę išraišką, tai šiais algoritmais randame tris pagrindines aritmetinės išraiškos formas: *prefix*, *infix postfix*, todėl taip vadinsime ir atitinkamus medžio apėjimo algoritmus.

Prefix algoritmas. Šiame algoritme pirmiausia aplankome viršūnę–šaknį, po to jos kairįjį pomedį ir vėliausiai dešinįjį pomedį. Pažymėkime $P(v)$ medžio viršūnės v aplankymo metodą (pavyzdžiui, tai gali būti informacinės dalies spausdinimas).

```
preOrder(node* v){
  if ( v <> NULL ){
    P(v);
    preOrder((*v).left);
    preOrder((*v).right);
  }
}
```

Infix algoritmas. Šiame algoritme pirmiausia aplankome kairįjį pomedį, po to viršūnę–šaknį ir vėliausiai dešinįjį pomedį.

```
inOrder(node* v){
  if ( v <> NULL ){
    inOrder((*v).left);
    P(v);
    inOrder((*v).right);
  }
}
```

Postfix algoritmas. Šiame algoritme pirmiausia aplankome kairįjį pomedį, po to dešinįjį pomedį ir vėliausiai viršūnę–šaknį.

```
postOrder(node* v){
  if ( v <> NULL ){
    postOrder((*v).left);
    postOrder((*v).right);
    P(v);
  }
}
```

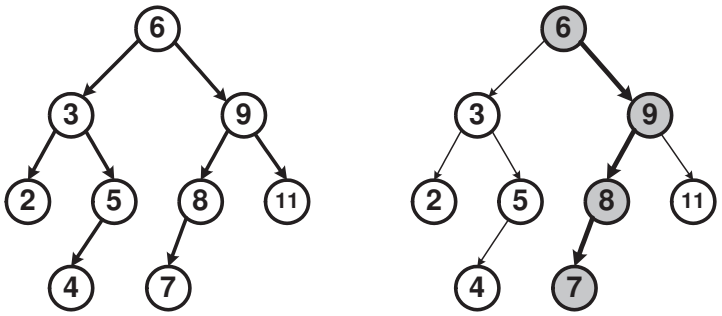
Pritaikę šiuos algoritmus 2.4 pavyzdžio medžiui, atspausdiname aritmetinės išraiškos $(a - b * c) * (d + e/f)$ tris skirtingas užrašymo formas

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

2.2.2. Dvejetainis paieškos medis

Dvejetainiai medžiai labai dažnai naudojami informacijos saugojimui, rūšiavimui ir paieškai. Tada ypač 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.

Dažniausiai elementus rūšiuojame pagal įrašo specialaus lauko data.key reikšmes. Dvejetainio paieškos medžio pavyzdys yra pateiktas 2.9*a* paveiksle.



2.9 pav. a) Dvejetainis paieškos medis, b) viršūnės, kurioje saugomas skaičius 7, paieškos kelias.

Tada nesunku patikrinti, ar medyje yra saugoma reikalinga informacija. Paieškos algoritmas yra panašus į rekursinius algoritmus, tačiau jo realizacijai rekursijos nereikia, nes visada pasirenkamas tik vienas iš dviejų pomedžių. Jei tokio elemento medyje nėra, tai procedūra grąžina nuorodą į tuščia viršūnę (NULL nuorodą).

```
node*  find(node* tree, T inf){
  while ( (tree <> NULL) && ((*tree).data <> inf) ){
    if ( (*tree).data < inf )
      tree = (*tree).right;
    else
      tree = (*tree).left;
  }
  return tree;
}
```

2.9*b* paveiksle parodyta, kaip ieškome viršūnės, kurioje saugomas skaičius 7.

Naujos viršūnės įterpimas. Naujoji viršūnė turi būti įterpta taip, kad ir atlikus šį veiksmą medis išliktų paieškos medžiu. Kadangi kol kas nebandome kontroliuoti medžio aukščio, tai įterpimo algoritme surandame atitinkamą medžio šaką ir sukuriame naują jos lapą. Jei pradinis medis yra tuščias, tai naujoji viršūnė tampa jo šaknimi. Pastebėsime, kad algoritme vėl nereikia naudoti rekursijos.

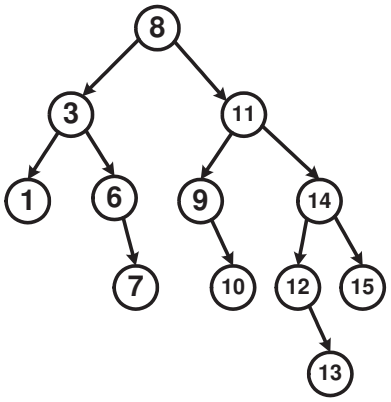
```
insert(node* tree, node* v){
```

```
  while ( tree <> NULL ){
    if ( (*v).data < (*tree).data )
      tree = (*tree).left;
    else
      tree = (*tree).right;
    }
  }
  tree = v;
  (*v).left = NULL;
  (*v).right = NULL;
}
```

2.6 pavyzdys. Dvejetainio paieškos medžio sudarymas. Faile F saugome skaičius:

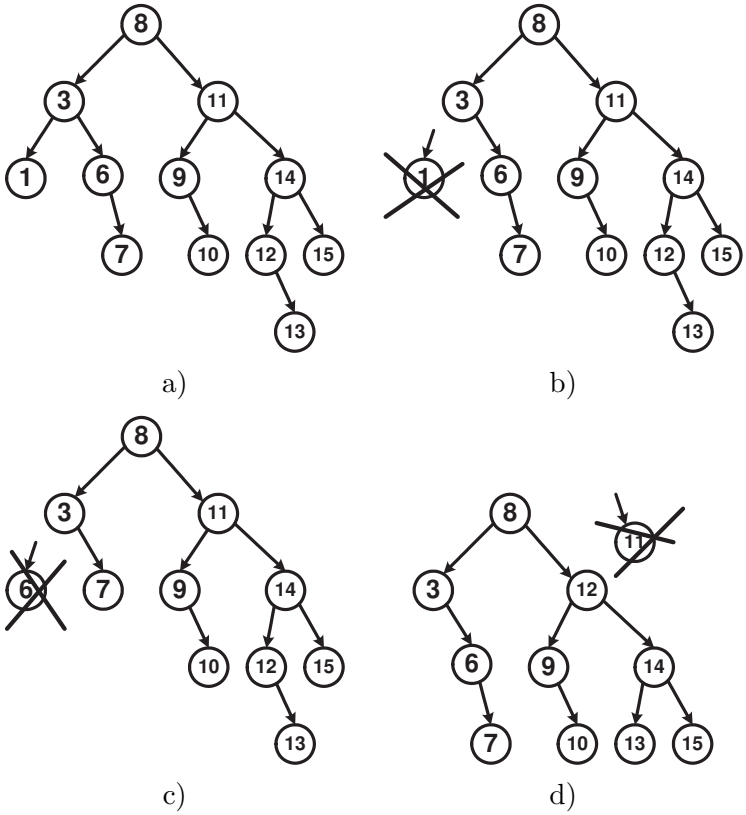
$$8,\ 11,\ 9,\ 3,\ 1,\ 14,\ 6,\ 12,\ 10,\ 7,\ 13,\ 15\,.$$

Perkelsime juos į dvejetainį paieškos medį, kuris pavaizduotas 2.10 paveiksle.



2.10 pav. Dvejetainis paieškos medis.

Viršūnės šalinimas. Pašalinti viršūnę iš paieškos medžio yra daug sudėtingiau, nei iš tiesinių duomenų struktūrų. Reikia garantuoti, kad ir atlikus šį veiksmą turėsime dvejetainį medį, o jo viršūnės bus išdėstytos medžio šakose taip, kad liks išpildyta pieškos medžio sąlyga.



2.11 pav. a) Dvejetainis paieškos medis, b) pašalinta "1" viršūnė, c) pašalinta "6" viršūnė, d) pašalinta "11" viršūnė.

Išskirsime tris viršūnės šalinimo atvejus.

- Jei iš medžio šaliname viršūnę–*lapą*, tai jos tėvo atitinkamai rodyklei priskiriame nuorodą NULL, viršūnę atkabiname ir ją sunaikiname (taip šaliname 2.10 paveiksle pavaizduoto medžio "1" viršūnę).

- Jei šaliname viršūnę, kuri turi tik vieną vaiką, tai viršūnės tėvo nuorodą nukreipiame į vaiką, o pačią viršūnę sunaikiname (taip šaliname 2.10 paveiksle pavaizduoto medžio "6" viršūnę).

- Jei šaliname viršūnę v, kuri turi abu vaikus, tai procedūra yra sudėtingesnė. Panaudodami *infix* viršūnių apėjimo algoritmą randame gretimą iš dešinės pusės viršūnę v'. Tarkime, kad 2.10 paveikslo medyje norime pašalinti viršūnę "11". Tada gretima jai viršūnė yra "12" (pastebėsime, kad ši viršūnė negali turėti kairiojo pomedžio vaiko).

Rodyklę, kuri rodė į v, nukreipiame į viršūnę v', o rodyklę, kuri rodė į v', nukreipiame į $(*v').right$ pomedį. Viršūnės v' kairiuoju ir dešiniuoju pomedžiais tampa v atitinkami pomedžiai. Vietoj paskutinių dviejų rodyklių pakeitimų galime kopijuoti viršūnės v' informacinę dalį į viršūnę v, bei šalinti v'.

Elementų šalinimo veiksmai yra pavaizduoti 2.11 paveiksle.

Sudarysime procedūrą, kuri patikrina ar medyje yra ieškomas elementas ir pašalina surastą viršūnę. Joje naudojame jau anksčiau aprašytą paieškos procedūrą `find`().

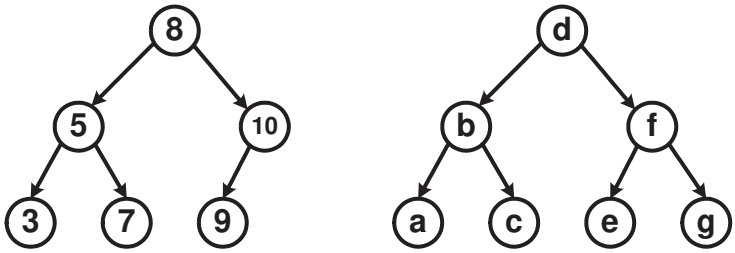
```
deleteNode(node* v, T a){
  node* p, q;
  v = find(v, a);
  if ( v <>  NULL ){ // Elementas medyje surastas
    //  Įsimename šalinamo elemento nuorodą
    q = v;
    if ( (*v).left ==  NULL ){
      // Ne daugiau kaip vienas, dešinysis vaikas
      v = (*v).right;
    }else
      if ( (*v).right ==  NULL ) {
        // Ne daugiau kaip vienas, kairysis vaikas
        v = (*v).left;
      }else{  // v turi du vaikus
        // Randame gretimą elementą
        p = (*v).right;
        while ( (*p).left <> NULL )
          p = (*p).left;
        v = p;  // Perkeliame rodyklę į surastąjį elementą
        p = (*p).right;
        (*v).left = (*q).left;
        (*v).right = (*q).right;
      }
    delete q;
  }
```

2.2.3. Paieškos algoritmo sudėtingumas

Šiame skirsnyje ištirsime sukonstruoto paieškos medžio savybes. Mus domina klausimas, kiek užtruks informacijos paieška tokioje duomenų bazėje. Bazine algoritmo operacija laikome dviejų *raktų* palyginimą. Jei raktas yra *skaičius*, tai lyginame pagal aritmetikos taisykles, jei raktas yra *tekstinė* konstanta, tai naudojame abecėlės tvarką.

Elemento paieškos sudėtingumas priklauso nuo dvejetainio medžio aukščio. Tarkime medyje yra N viršūnių ir visi N skaičių yra vienodai dažnai tikrinami.

Analizę pradėsime nuo idealiai subalansuoto paieškos medžio, kai turime surūšiuotą aibę ir ją perkeliame į dvejetainį paieškos medį. Tokių medžių pavyzdžiai yra pateikti 2.12 paveiksle. Taip saugoma informacija žodynuose ir enciklopedijose. Tarkime enciklopediją sudaro trisdešimt du tomai, tada paieškos medžio šaknis yra septynioliktasis tomas, kairįjį jo pomedį sudaro enciklopedijos pirmieji šešiolika tomų, o dešinįjį – tomai, kurių numeriai yra nuo aštuoniolikto iki trisdešimt antro.



2.12 pav. Surūšiuotų aibių saugojimas idealiai subalansuotame dvejetainiame paieškos medyje: a) skaičių aibė $\{3, 5, 7, 8, 9, 10\}$, b) raidžių aibė $\{a, b, c, d, e, f, g\}$.

Nagrinėkime nepalankiausią atvejį, kai $N = 2^n$, tada idealiai subalansuoto medžio aukštis yra $h = n$. Taigi vykdydami paiešką blogiausiu atveju atliksime

$$W_b = \log N + 1$$

raktų palyginimo veiksmų.

Labai dažnai svarbiau yra žinoti vidutinį (tikėtiną) paieškos algoritmo sudėtingumą, kai vienodai dažnai tikrinami visi N elementai. Dabar nepalankiausias yra atvejis, kai $N = 2^n - 1$, tada idealiai subalansuoto medžio aukštis yra $h = n-1$, o $n = \log(N+1)$. Tokiame medyje pilnai užpildyti visi

jo lygmenys. Tada i-ajame lygmenyje yra 2^i viršūnių, paieškos algoritmę iki šių viršūnių atliekame po $(i+1)$ tikrinimą. Todėl gauname tokį paieškos algoritmo vidutinio sudėtingumo įvertį

$$W_v = \frac{1}{N}\sum_{i=0}^{n-1}(i+1)\,2^i\,.$$

Pasinaudosime skaitinių eilučių išvestinės skaičiavimo formule:

$$\begin{aligned}\sum_{i=0}^{n-1}(i+1)\,x^i &= \left(\sum_{i=0}^{n-1}x^{i+1}\right)' \\ &= (x+x^2+\cdots+x^n)' = \left(\frac{x^{n+1}-x}{x-1}\right)' \\ &= \frac{((n+1)x^n-1)(x-1)-(x^{n+1}-x)}{(x-1)^2},\end{aligned}$$

imdami $x=2$ apskaičiuojame sumą

$$\sum_{i=0}^{n-1}(i+1)\,2^i = (n-1)2^n+1.$$

Taigi atliekant paiešką idealiai subalansuotame medyje algoritmo sudėtingumas vidutiniu atveju yra

$$W_v = \frac{\log(N+1)-1)(N+1)+1}{N} = \log N + \mathcal{O}(1).$$

Matome, kad tiek vidutiniu, tiek ir blogiausiu atveju paieškos algoritmo sudėtingumas yra $W=\mathcal{O}(\log N)$.

Norėdami sukonstruoti idealiai subalansuotą medį, pradžioje turime surūšiuoti duomenis, o tai daug sudėtingesnis uždavinys. Tokią pagalbinę operaciją tikslinga atlikti tik tada, kai duomenys ilgai nekinta ir juos daug kartų naudojame paieškos metu (pvz. žodynai). Jeigu duomenų bazę dažnai tenka papildyti naujais įrašais ir šalinti kai kuriuos senus įrašus, tai naudojame paprastenį dvejetainį paieškos medį, kuriame šios operacijos atliekamos labai efektyviai.

Tačiau nepalankiausiu atveju dvejetainis medis išsigimsta į tiesinį sąrašą. Taip bus, jei duomenis įrašysime rakto didėjimo ar mažėjimo tvarka. Tokiame paieškos medyje vidutinis paieškos algoritmo sudėtingumas yra

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

Įvertinsime vidutinį algoritmo sudėtingumą, kai nagrinėjame visus galimus variantus. Išviso galime sudaryti $N!$ skirtingus paieškos medžius. Tai milžiniškas skaičius net kai turime nedaug elementų, pvz. $N = 20$, nes remiantis Stirlingo formule

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

Jeigu per vieną sekundę sudarytume vieną miliardą paieškos medžių, tai darbą pabaigtume tik po septyniasdešimt septynių metų. Padidinus elementų skaičių dar vienu, jau neužtektų ir pusantro tūkstančio metų.

Atlikdami tokių algoritmų analizę ir parodome didelę teorinių įverčių naudą, nes jie leidžia įvertinti algoritmų sudėtingumą atrodytų beviltiškose situacijose.

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

Tarkime, kad elementai yra numeruojami jų didėjimo tvarka. Elementas i bus medžio šaknimi su tikimybe $\dfrac{1}{N}$. Kadangi tai dvejetainis paieškos medis, tai žinome, kad kairiajame jo pomedyje bus $(i-1)$ elementas, o dešiniajame – $(N-i)$ elementų. Pažymėkime $a_N^{(i)}$ paieškos algoritmo vidutinį sudėtingumą, kai medžio šaknyje yra i-tasis elementas, tada

$$a_N = \frac{1}{N}\sum_{i=1}^{N} a_N^{(i)}\,.$$

Atsižvelgę į tai, kad

- bet kurios kairiojo pomedžio viršūnės paieškos vidutinis ilgis yra $a_{i-1}+1$,
- šaknies i paieškos ilgis lygus 1,
- bet kurios dešiniojo pomedžio viršūnės paieškos vidutinis ilgis yra $a_{N-i}+1$,

gauname lygybę

$$a_N^{(i)} = \big(a_{i-1}+1\big)\frac{i-1}{N} + \frac{1}{N} + \big(a_{N-i}+1\big)\frac{N-i}{N}\,.$$

Tada paieškos algoritmo vidutinis sudėtingumas tekina lygtį

$$a_N = 1 + \frac{1}{N^2}\sum_{i=1}^{N}\big[(i-1)a_{i-1} + (N-i)a_{N-i}\big] = 1 + \frac{2}{N^2}\sum_{i=1}^{N}(i-1)a_{i-1}\,.$$

Pakeitę sumavimo indeksą $j = i - 1$, gauname lygtį

$$a_N = 1 + \frac{2}{N^2} \sum_{j=1}^{N-1} j\, a_j \,. \tag{2.1}$$

Imdami reikalingą N reikšmę, kompiuteriu nesunkiai galime suskaičiuoti tokios lygties sprendinį. Tačiau mūsų tikslas yra sudaryti analizinį paieškos algoritmos sudėtingumo asimptotinį įvertį.

Gautoje lygtyje rekurentinis sąryšis sieja visus N nežinomųjų. Parodysime, kad galime pertvarkyti lygtį taip, kad ji apibrėžtų tik dviejų gretimų nežinomųjų sąryšį. Užrašykime pagrindinę lygtį ir nežinomajam a_{N-1}:

$$a_{N-1} = 1 + \frac{2}{(N-1)^2} \sum_{j=1}^{N-2} j\, a_j \,.$$

Padauginę šią lygtį iš $\dfrac{(N-1)^2}{N^2}$ ir atėmę iš (2.1) lygties, gauname dvižingsnę tiesinę lygtį

$$a_N = \frac{1}{N^2}\big((N^2-1)a_{N-1} + 2N - 1\big) \,.$$

Apibrėžkime harmoninės eilutės dalinės sumos funkciją

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

Tada dvižingsnės lygties sprendinys yra

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

Atsižvelgę į tai, kad

$$H_N = \ln N + \gamma + \mathcal{O}\Big(\frac{1}{N^2}\Big) \,,$$

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. Paieškos medyje galime labai taupiai įterpti naujas ir pašalinti nereikalingas viršūnes.