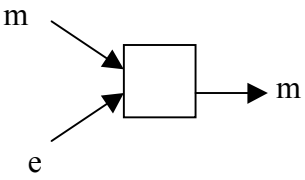
Dvejetainių ir AVL medžių programavimas

Dvejetainiai paieškos medžiai

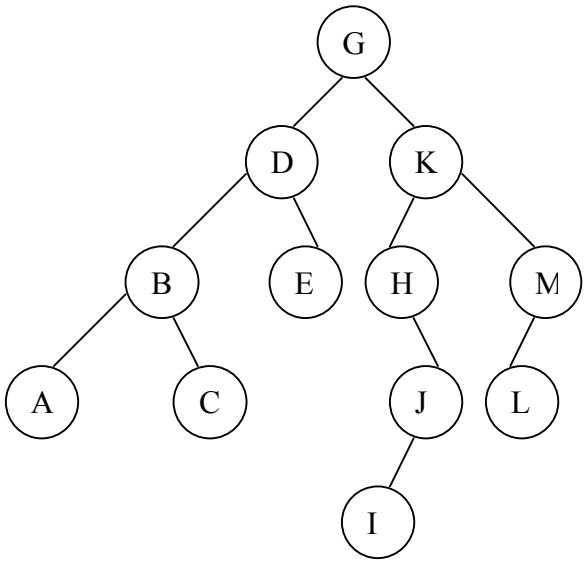
Su dvejetainiais medžiais atliekamos šios operacijos
- elemento įtraukimas
- elemento paieška
- medžio spausdinimas
    * preorder
    * postorder
    * inorder
- elemento pašalinimas

Aptarsime elemento pašalinimą. Konstruosime procedūrą:



Turime medį m ir elementą e. Atlikus procedūrą, turėsime tik medį m.

Tarkime, turime medį:



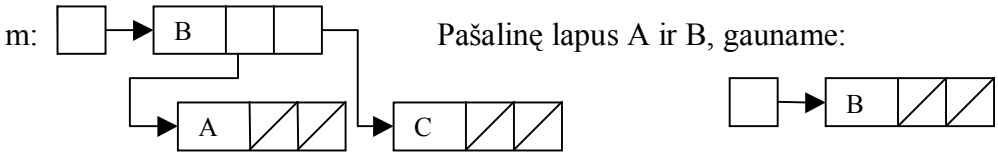
Medis sudarytas atsižvelgiant į leksikografinę tvarką – t.y., pvz., G < D

Šalinant iš medžio, galimi 4 variantai:
1. šalinamas elementas – lapas
   (mūsų pavyzdyje lapai yra A, C, E, I, L)
2. Šalinamas elementas neturi kairės šakos
   (pavyzdyje: H)
3. Šalinamas elementas neturi dešinės šakos
   (M, J)
4. Šalinamas elementas turi ir kairę, ir dešinę šakas
   (G, D, K, B).

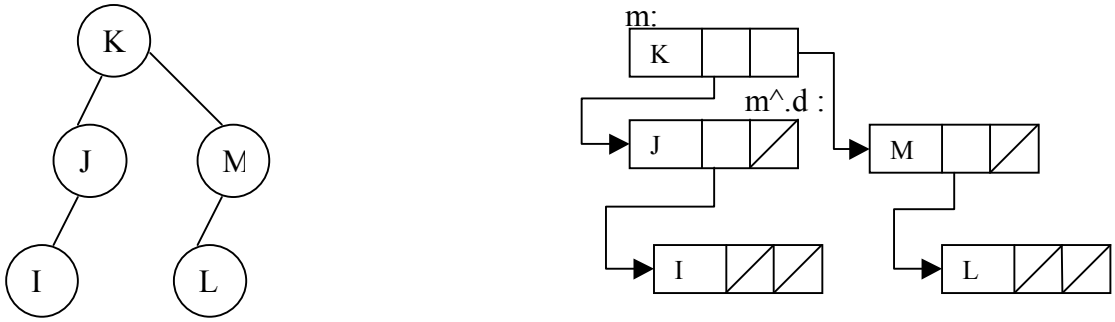
Parengė: Eleonora Kulberkytė, 2-o kurso informatikos specialybės studentė

Aptarsime kiekvieną atvejį.

1. Pirmu atveju mūsų nagrinėjamo medžio dalis ties lapu schematiškai atrodo taip:

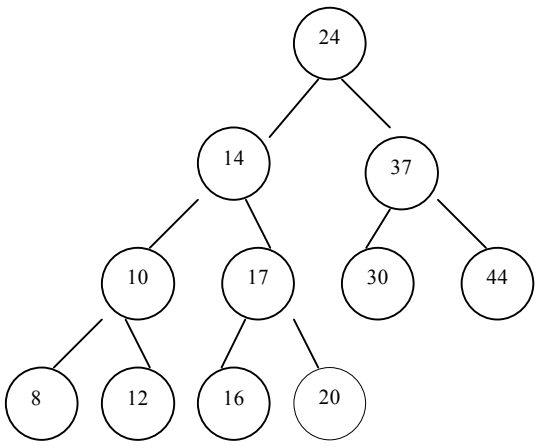


2. Šiuo atveju pašalinę elementą H, turime tokį medžio fragmentą:



3. Analogiškai.

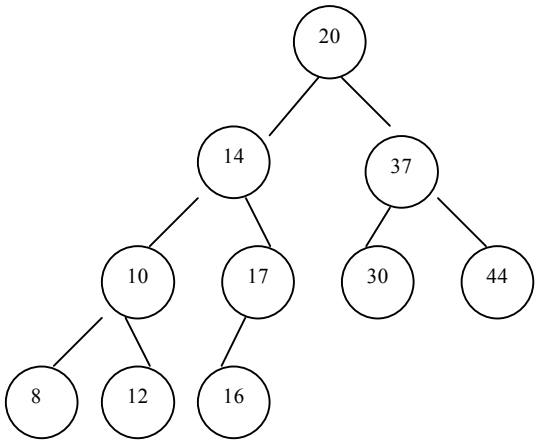
4.



Tarkime, reikia pašalinti viršūnės elementą 24. Tačiau, kad medžio struktūra nesugriūtų, jį turi kažkas pakeisti. Realiausias tam tinkamas skaičius – 20, nes jis yra iš visų kairės šakos elementų pats didžiausias (kairės šakos pats dešiniausias elementas). Medžio struktūra nesugriūtų, jei 24 keistume ir dešinės šakos pačiu kairiausiu elementu (iš visos dešinės šakos pačiu mažiausiu).

Parengė: Eleonora Kulberkytė, 2-o kurso informatikos specialybės studentė

Pašalinus elementą 24, gauname medį:



Vietoj 24 viršūnėje turime 20. Tačiau, kad nesidubliuotų elementai, taip pat pašaliname lapą su skaičiumi 20.

Galime parašyti elemento šalinimo procedūrą:

```
Type medis = ^mazgas;
Mazgas = record
              info: integer;
              k, d : medis;
            end;
procedure pašalinti (var m : medis; e : integer);
var r, pirm : medis;
begin
  if m <>  nil
     then begin
            if m^.info = e
              then …
              else if e > m^.info
                   then pašalinti (m^.d, e)
                   else pašalinti (m^.k, e)
          end;
end;
```

Tai, ką sąlygos sakinyje žymejome daugtaškiu, parašome:

```
Begin
    If m^.k =  nil                    {nėra kairės medžio šakos}
      Then begin
             r := m;
             m := m^.d;
             dispose (r);
           end
      else if m^.d = nil
           then begin
                  r := m;
                  m := m^.k;
                  dispose (r);
                end
```

```
        else  {yra abi šakos}
              then begin
                     pirm := pirmatkas (m);
                     m^.info := pirm^.info;
                     pašalinti (mŠ.k, mŠ.info);
                   end;
end;
```

Mūsų parašytoje sąlygos sakinio dalyje yra funkcija *pirmatakas* - ji pakeičia viršūnės pašalintą elementą tokiu, kuris leidžia išlaikyti medžio struktūrą. Ją parašysime (pirmtaku laikome kairės šakos didžiausią elementą):

```
Function pirmtakas (m: medis): medis
var p : medis;
begin
    p := m^.k;
    while p^.d <>  nil do
        p := p^.d;
    pirmtakas := p;
end;
```

Dvejetainiai medžiai naudojami aprašant (generuojant) duomenų bazes.
Panagrinėkime pačios paprasčiausios duomenų bazės pavyzdį. Tarkime, su duomenų baze galima atlikti tokias tris operacijas (tranzakcijas):

- Elemento įterpimas
- elemento paieška
- elemento pašalinimas
- duomenų bazės turinio spausdinimas

Susitarkime, kad tranzakcijų kodai bus tokie:

I – įterpimas
Q – elemento paieška
D – elemento pašalinimas

Parašome programą tokios duomenų bazės kūrimui ir jos tvarkymui šiomis tranzakcijomis:

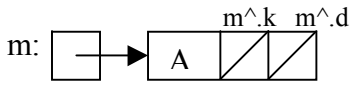
```
Program PaprastaDB;
Type medis = ^mazgas;
       Mazgas = record
                    Info: integer;
                    k, d : medis;
                  end;

var
  DB: medis;
 Kodas: char;
 i : integer;
```

```
begin
    Db := nil;
    While not eof do
        Begin
            ReadLn (kodas, i);
            Case kodas of
                'I' : Įterpti (Db, i);
                'Q' : if yra (DB, i)
                          then WriteLn (i, , yra DB')
                          else WriteLn (i, 'nėra DB);
                'D' : pašalinti (i, DB);
            end;
        end;
    Writeln ('DB turinys');
    Spausdinti (DB);
    Writeln (gylis (DB));
    WriteLn (kiek (DB));
End.
```

Šioje paprasčiausią duomenų bazę generuojančioje programoje taip pat be jau minėtų tranzakcijų turime ir funkcijas, nustatančias dvejetainio medžio, kitaip – duomenų bazės gylį (funkcija *gylis (DB)*), jos elementų skaičių (funckija *Kiek (DB)*). Aprašysime jas:

```
Function gylis (m: medis): integer;
Begin
    If m = nil
      Then gylis = 0
      Else gylis := 1 + max (gylis (m^.k), gylis (m^.d);
End;
```

```
Function Kiek (m: medis): integer;
Begin
    If m = nil
      Then kiek := 0
      Else kiek := 1 + kiek (m^.d) + kiek (m^.k)
End;
```



Taigi turime visas funkcijas ir procedūras, reikalingas apdoroti pačią primityviausią duomenų bazę:

Funkcija *Įterpti*;
Procedūra *Spausdinti*;
Procedūra *pašalinti*;
Funkcija *pirmtakas*;
Funkcija *Kiek*;
Funkcija *Gylis*;

AVL medžiai.

AVL medžiai – tai tokia dvejetainių medžių rūšis, kuriai priklauso medžiai, suformuoti pagal tam tikrą balansavimo taisyklę. AVL medžius ištyrinėjo du rusų mokslininkai: Adelson – Velskij ir Landis.

AVL medžių balansavimo taisyklė yra tokia – Tokio medžio kairiosios ir dešinios šakos skiriasi ne daugiau kaip per vieną mazgą.

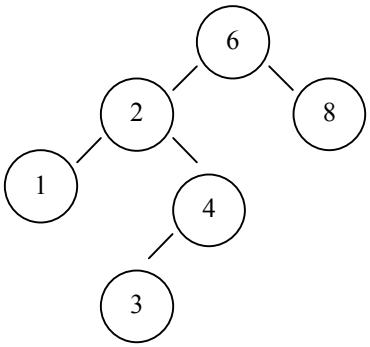
Dvejetainių (paprastų medžių) ir AVL medžių palyginimas

	Palyginimų skaičius (blogiausias atvejis)	Palyginimų skaičius (vidutiniškai)
Dvejetainiai medžiai	n	$1{,}39 \log_2 n$
AVL medžiai	$1{,}44 \log_2 n$	$1{,}04 \log_2 n$

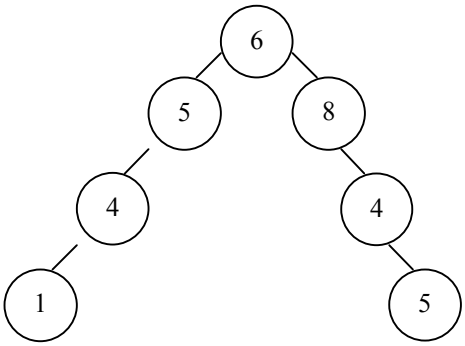
n – elementų (mazgų) kiekis medyje

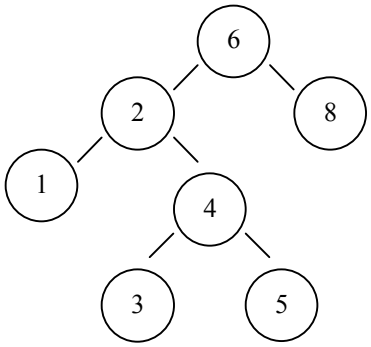
AVL ir ne avl medžių pavyzdžiai:

AVL medis



Ne AVL medžiai:





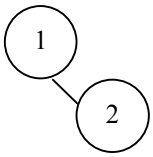
Įterpimas į AVL medį.

Norint įterpti į duotą medį, reikalingos 4 korekcijos (transformacijos)
- posūkis kairėn
- posūkis dešinėn
- dvigubas posūkis kairėn
- dvigubas posūkis dešinėn;

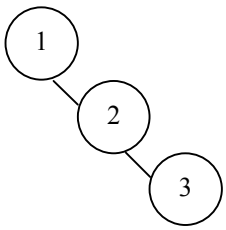
Pvz. Į AVL medį įterpiame tokius skaičius: 1, 2, 3, 4, 5, 6, 7.

Įterpiame 1:

Įterpiame 2:

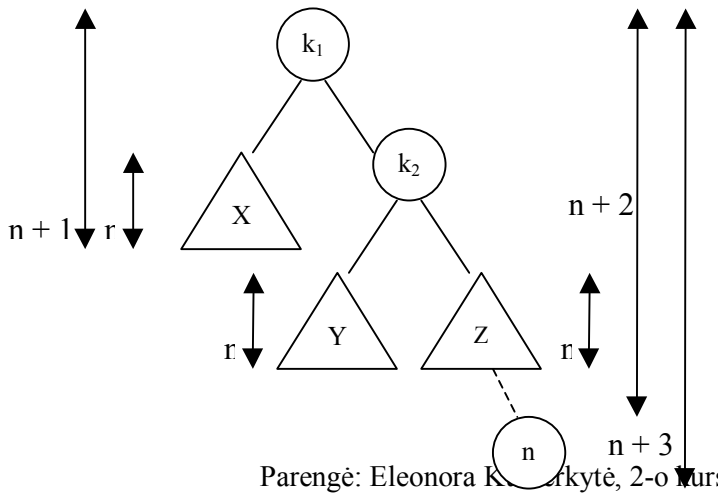


Įterpiame 3:



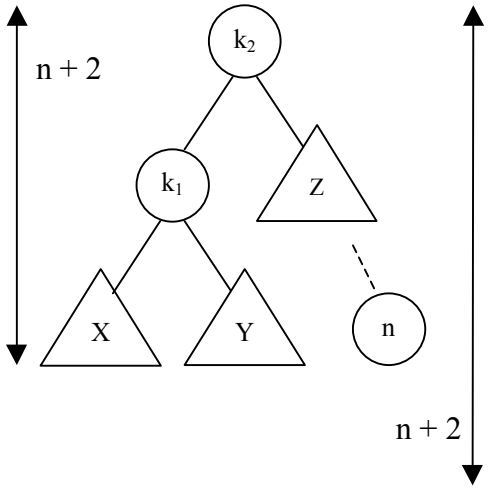
Matome, kad tai nėra AVL medis. Vadinasi, Reikia atlikti tam tikrą transformaciją, kuri transformuotų duotą medį į AVL medį.

<u>Posūkis kairėn.</u>



Bendras atvejis. Naujas elementas n įterpiamas į medį – jis tampa lapu. Matome, kad atsirado pažeidimas. X, Y, Z – submedžiai.

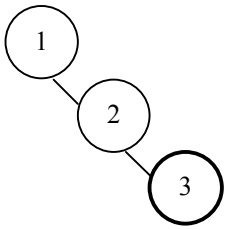
Atlikus posūkį:



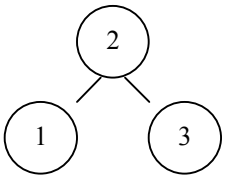
Kai jau atliktas posūkis, tai gauname AVL medį.
Įrodome, kad tai korektiškas AVL medis.
Medyje, prie kurio prijungėme elementą n, turime:
$k_1 < k_2$
$X < k_1$
$Y > k_1$

Matome, kad tokius pat santykius turime ir transformuotame medyje.

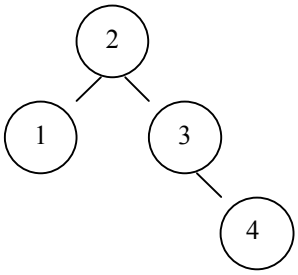
Pavyzdyje turejome:



Transformuojame (atliekame posūkį kairėn):



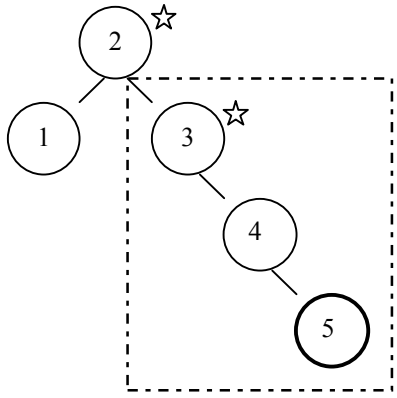
Prie gauto medžio prijungiame 4:

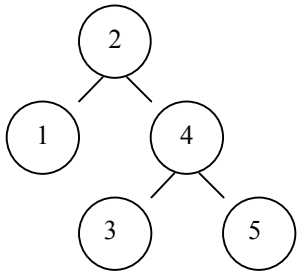


Gautas medis – taisyklingas AVL medis.

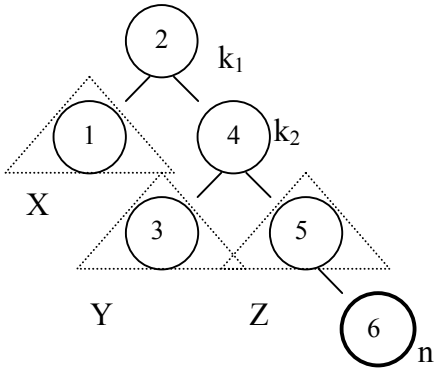
Prijungiame 5:

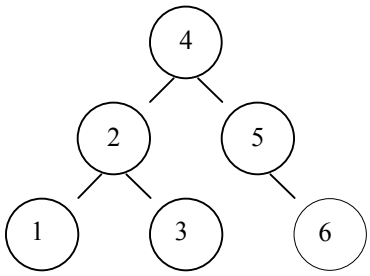
Turime du pažeidimus AVL medžio struktūroje. Taikysime posūki kairėn (pažymėtą medžio šaką (submedį) transformuojame ir prijungiame prie pradinio medžio:



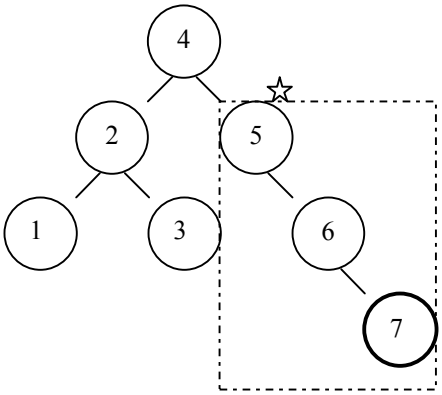


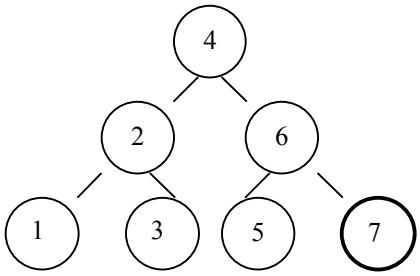
Prijungiame 6 ir atliekame transformaciją kairėn:





Prijungiame 7 ir atliekame transformaciją kairėn:

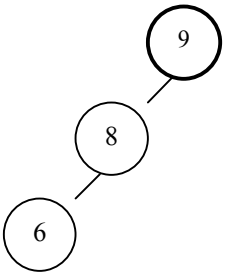




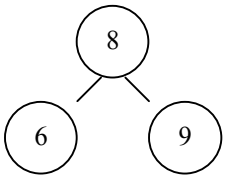
<u>Posūkis dešinėn.</u>

Posūkis dešinėn atliekamas analogiškai kaip ir posūkis kairėn, pvz:

Turime medį:

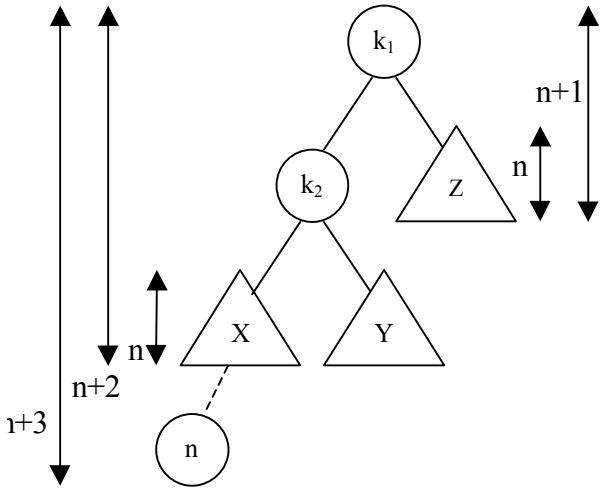


Atlikus posūkį dešinėn, gautas AVL medis:

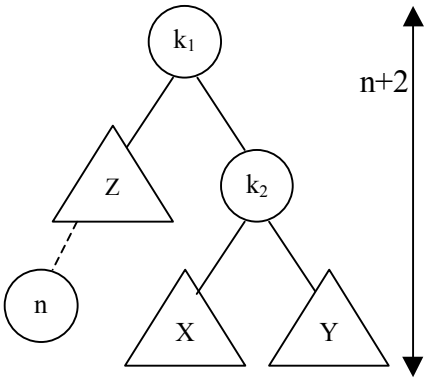


Bendra taisyklė posūkiui dešinėn.

Turime tokios struktūros medį (X, Y, Z – submedžiai), į kurį Įtrauktas naujas elementas n (lapas):



Atlikus posūkį dešinėn, gautas AVL medis:



Kad gautasis medis – korektiškas AVL medis, įrodome panašiai, kaip ir posūkio kairėn atveju.

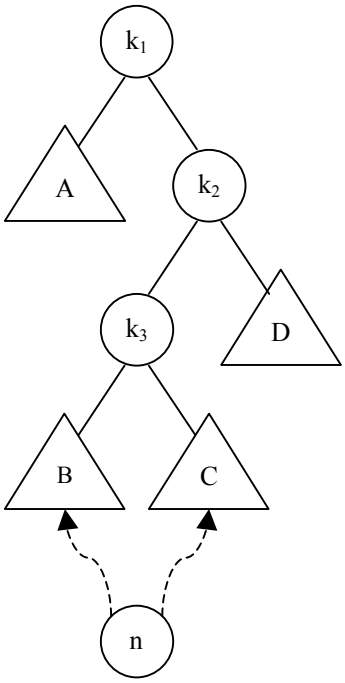
Tokio tipo posūkiai (kairėn arba dešinėn) naudojami tokiais atvejais:

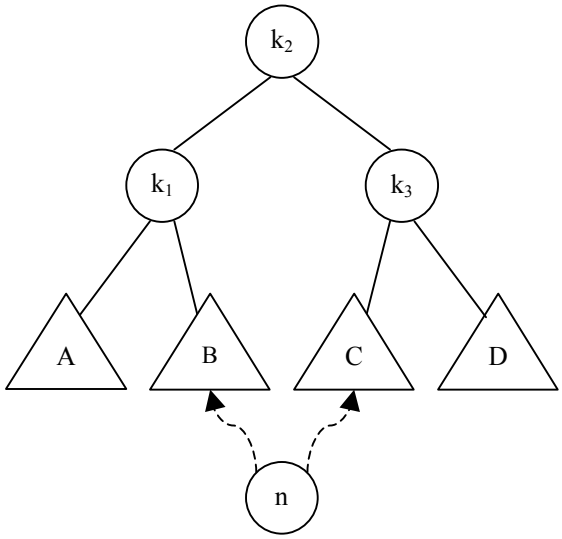
Posūkio tipas	**n santykis su medžio elementais**
Posūkis kairėn	$n > k_1 > k_2$
Posūkis dešinėn	$n < k_1 < k_2$

Parengė: Eleonora Kulberkytė, 2-o kurso informatikos specialybės studentė

Dvigubas posūkis.

Apibendrinta dvigubo posūkio kairėn taisyklė.

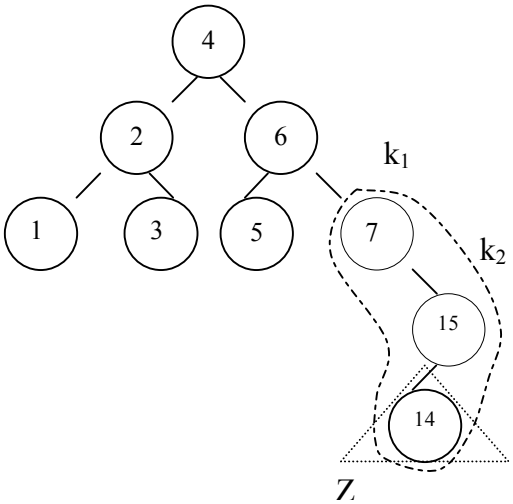


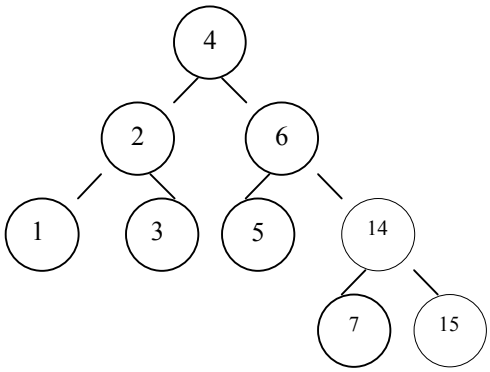


Pavyzdžiui, prie jau turimo AVL medžio prijunkime skaičių 15 (prijungus tik 15, gauname tvarkingą AVL medį) ir skaičių 14:

Gautas medis nėra AVL medis. Atliekame dvigubą posūkį kairėn ir

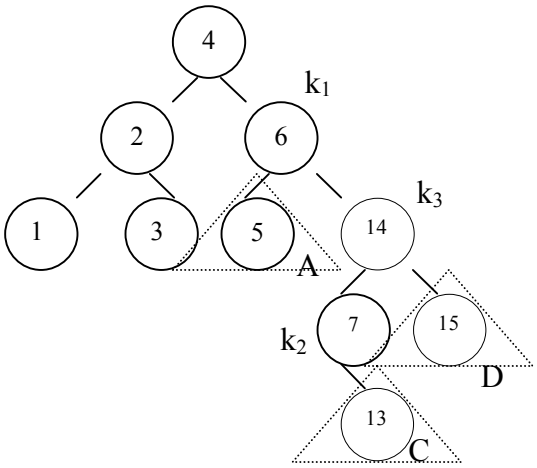
gauname taisyklingą AVL medį:



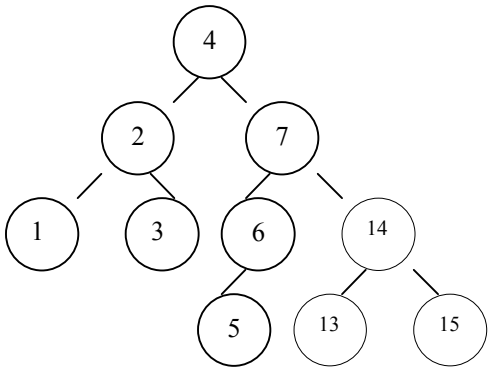


Parengė: Eleonora Kulberkytė, 2-o kurso informatikos specialybės studentė

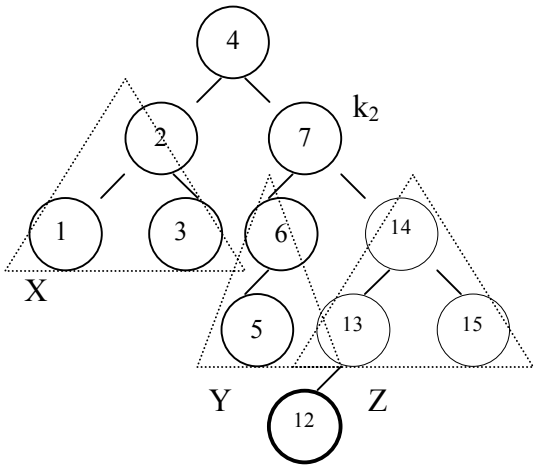
Į turimą medį įjungsime skaičių 13:



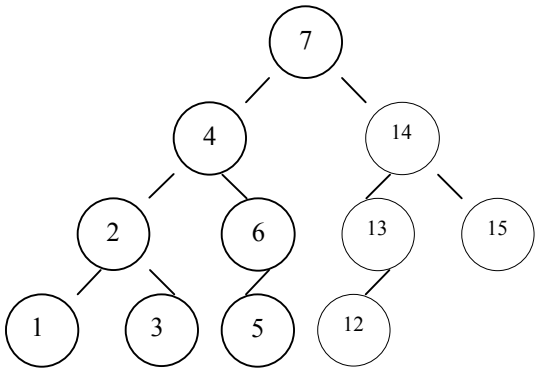
Atliekame dvigubą posūkį kairėn:



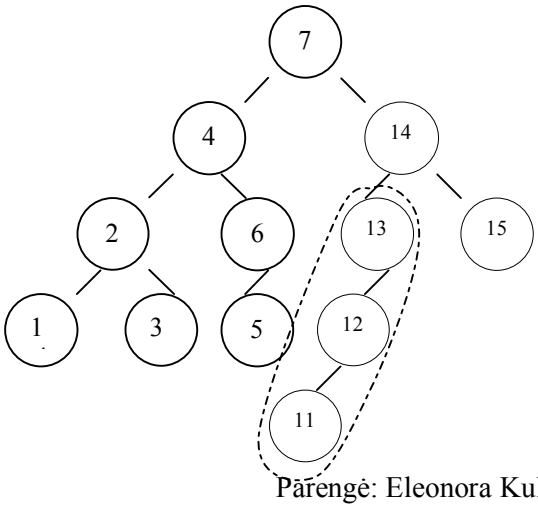
Į turimą medį įjungiame 12:



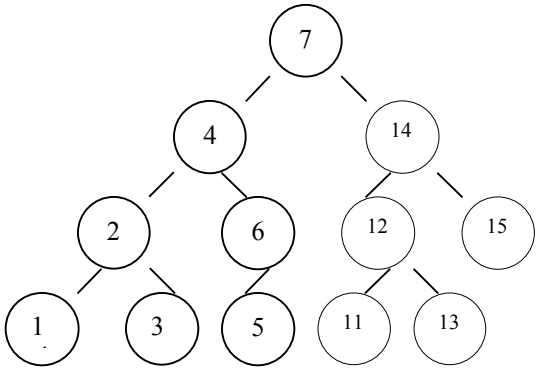
Transformuojame:



Į turimą medį įjungiame skaičių 11: atliekame



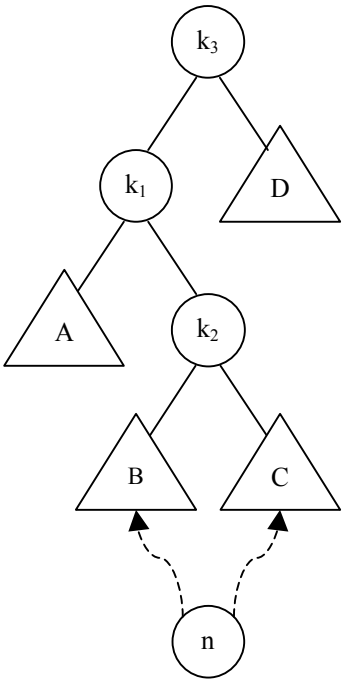
Šiam medžiui transformuoti į AVL medį tik paprastą (nedvigubą) posūkį dešinėn:



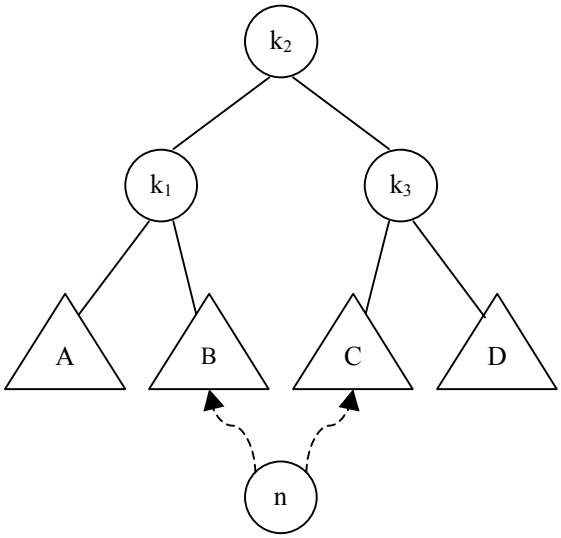
Parengė: Eleonora Kulberkytė, 2-o kurso informatikos specialybės studentė

Dvigubas posūkis dešinėn.

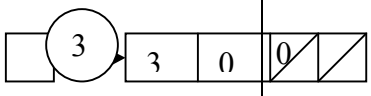
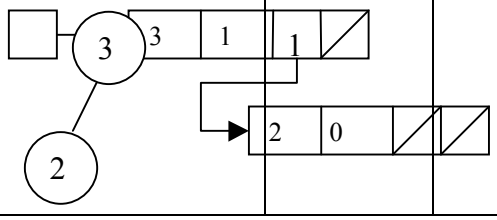
Apibendrinta taisyklė. Turime medį, prie kurio šakos B arba C prijungiame naują elementą n. Medį reikia transformuoti taip, kad gautume taisyklingą AVL medį.



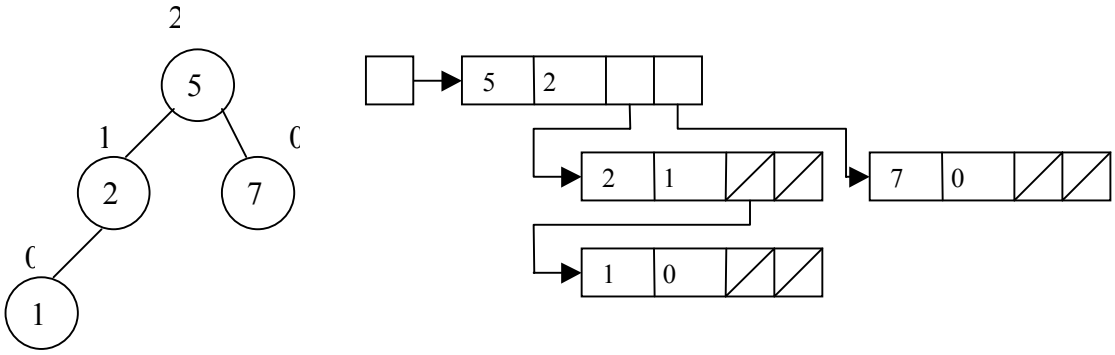
Atliekamas posūkis dešinėn. Gautas medis – taisyklingas AVL medis. Elementas n Gali būti B arba C lapas.



Norint aprašyti įterpimo į AVL medį procedūrą, būtina žinoti medžio aukštį (gylį). Susitarsime, kad:

Medis	Medžio aukštis	Medžio vaizdavimas su rodykliniais ryšiais
Nėra medžio	-1	Nil
		
		

Turime medį, kurio šakos yra tokių aukščių (pačio medžio aukštis – viršūnės aukštis):



Aprašome tipus, kuriuos naudosime procedūroms ir funkcijoms:

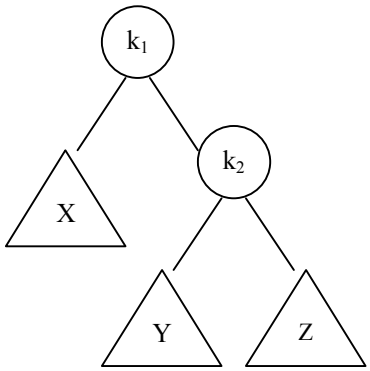
```
Type
    medis = ^mazgas;
    mazgas = record
                 inf: integer;
                 aukštis : integer;
                 k, d : medis;
             end;
```

Parašysime funkciją apskaičiuoti medžio aukščiui:

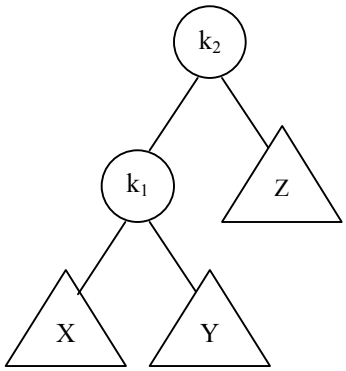
```
function aukštis (m: medis) : integer;
begin
  if m = nil
    then aukštis := -1
    else aukštis := m^.aukštis
end;
```

<u>Posūkis kairėn</u>

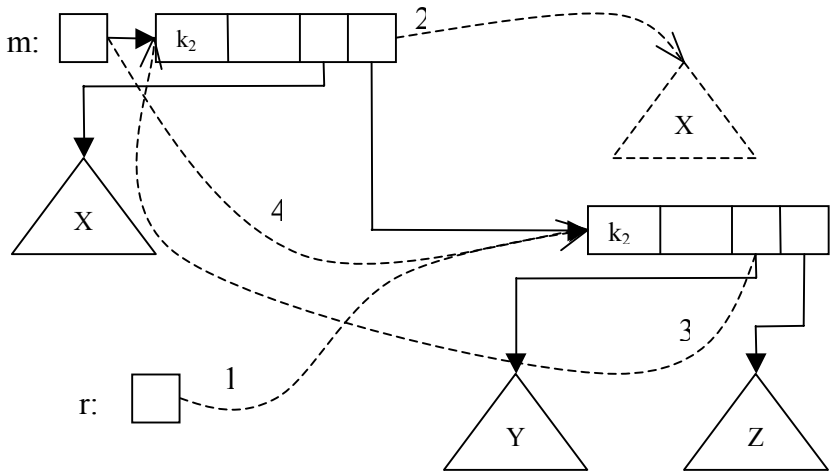
Turime tokį medį:



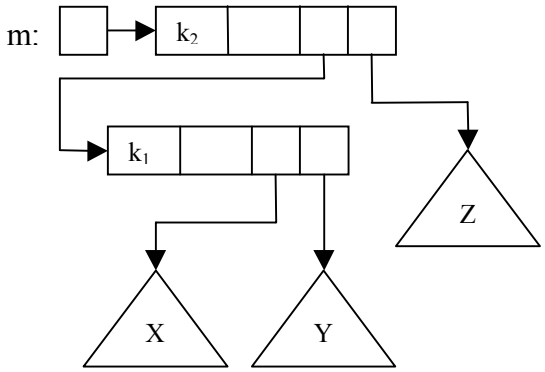
Atlikus transformaciją, gautas toks AVL medis:



Norint iš pradinio medžio gauti transformuotą, būitina atlikti tokius veiksmus:



gautas rezultatas:

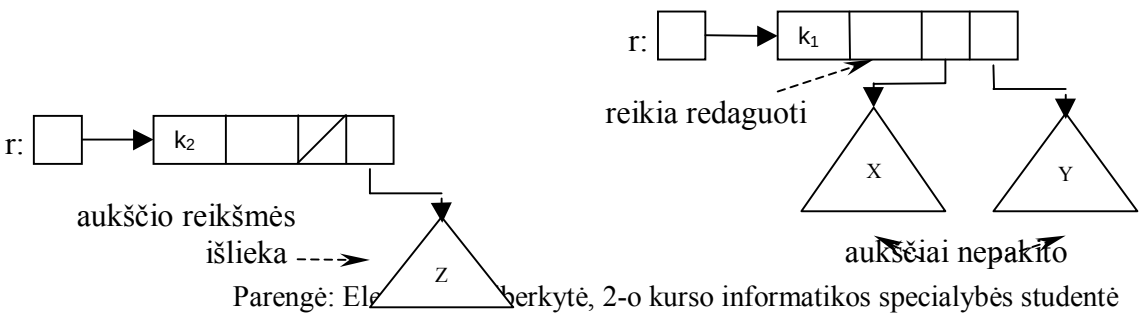


Parašysime procedūrą posūkiui kairėn:

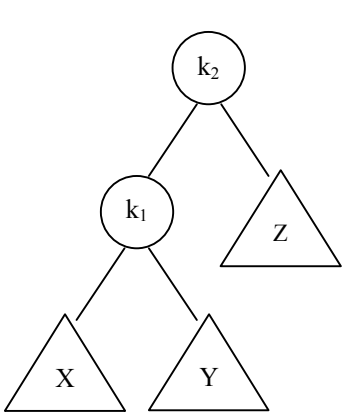
```
procedure kairėn (var m : medis);
  var r: medis
begin
    r := m^.d;              (1)
    m^.d := r^.k;           (2)
    ….                                  {medžio aukščio sutvarkymas}
    m := r;                 (3)
end;
```

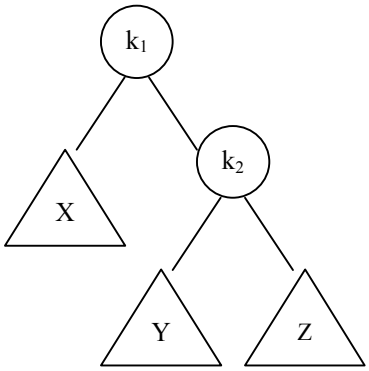
Transformuoto medžio šakų aukščiams sutvarkyti reikalingi tokie sakiniai, kuriuos atliekant posūkio kairėn procedūra kreipiasi į jau anksčiau aprašytą funkciją *aukštis*:

```
m^.aukštis := max (aukštis (m^.k), aukštis (m^.d)) + 1;
r^.aukštis := max (m^.aukštis, aukštis (r^.d)) + 1;
```

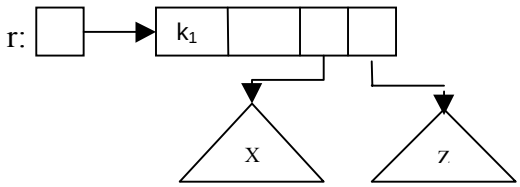


Dabar aprašysime analogišką procedūrą posūkiui dešinėn:

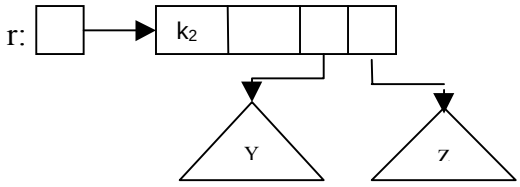




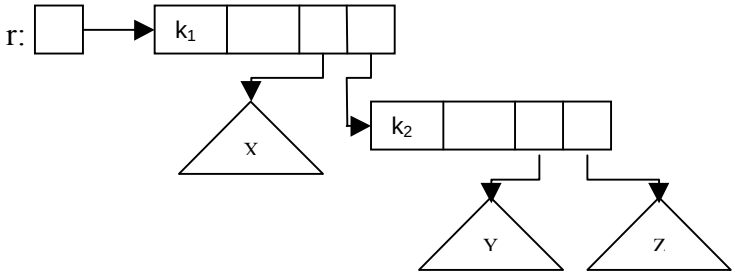

```
Procedure dešinėn (var m: medis);
  var  r: medis;
begin
  r := m^.k;
```




```
  m^.k := m^.d;
```




```
  ….                                  {medžio aukščio sutvarkymas}

  m:= r;
```



```
end;
```

Sutvarkome medžio aukštį:

```
m^.aukštis := max (aukštis (m^.k), aukštis (m^.d)) + 1;
r^.aukštis := max (m^.aukštis, aukštis (r^.d)) + 1;
```

Aprašome pagrindinę procedūrą, kuri įterpia elementą į AVL medį (reikiant AVL medį transformuoja):

```
Procedure Įterpti_į_AVL (i: integer; var m : medis);
begin
  if m = nil                                  {jei AVL medis tuščias}
    then begin
           new (m);                           {kuriamas AVL medis iš vieno elemento}
           m^.info := i;
           m^.aukštis := 0;
           m^.k := nil;
           m^.d := nil;
         end
    else ▦
end;
```

Užbaigiame sąlygos sakinį, pažymėtą ▦ :

```
if i < m^.inf
  then begin
         Įterpti_Į_AVL (i, m^.k);
         if aukštis (m^.k) – aukštis (m^.d) = 2          {AVL struktūra pažeista}
           then if i < m^.k^.inf
                  then dešinėn (m)
                  else dvigubas_dešinėn (m)
           else m^.aukštis := max (aukštis (m^.k), aukštis (m^.d)) + 1;
       end
  else begin
         ….                          {analogiški sakiniai dešiniajai šakai}
       end;
```

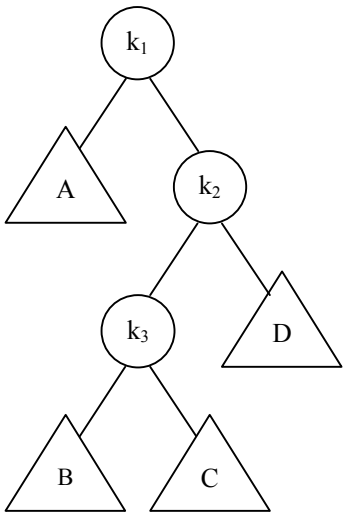
Pagrindinė programa:

```
Program AVL_medis;
Type
    medis = ^mazgas;
    mazgas = record
                 inf: integer;
                 aukštis : integer;
                 k, d : medis;
               end;
var
  M : medis;
```

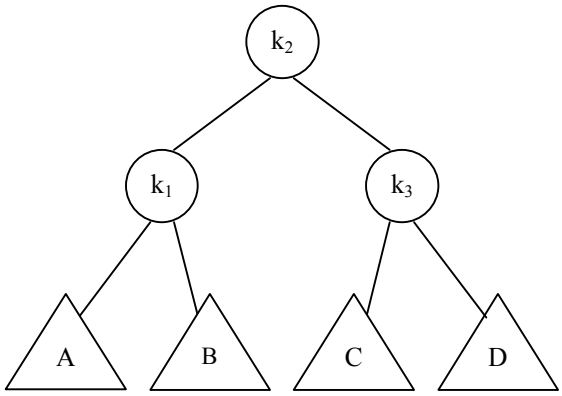
```
  j : integer;
begin
  M := nil;
  readln (j);
  while j > 0 do
      begin
        Įterpti_Į_AVL (j, M);
        readln (j);
      end
end.
```

Dvigubas posūkis kairėn.

Turime tokios struktūros dvejetainį medį:



Atlikus dvigubo posūkio kairėn transformaciją, gautas medis:



Galima pastebėti, kad dvigubas posūkis kairėn ekvivalentus tokiai transformacijų sekai:

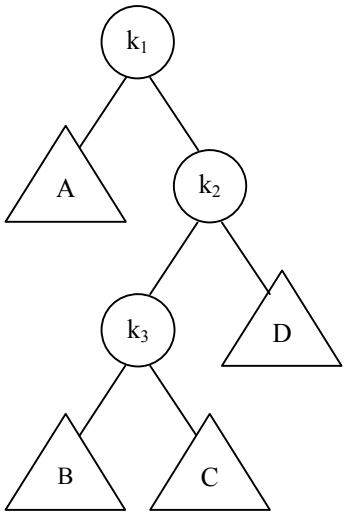
- posūkis dešinėn (viršūnė k_3)
- posūkis kairėn (viršūnė k_1)

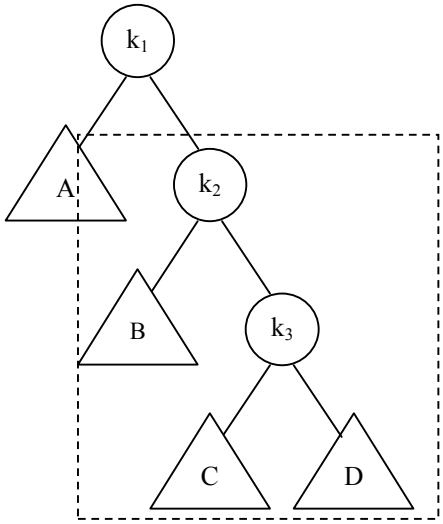
Įrodome, kad būtent taip ir yra:

pradinis medis:

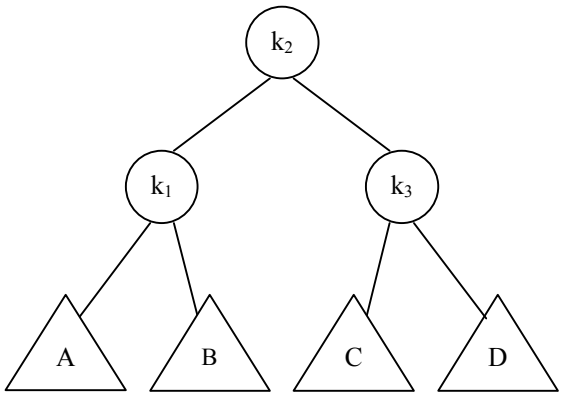
Atlikus posūkį dešinėn (apie viršūnę

k_3)





gautą medį transformuojame posūkiu kairėn (apie viršūnę k_1):



Parašysime procedūrą dvigubam posūkiui kairėn remdamiesi aukščiau išdėstytais pastebėjimais:

```
procedure dvigubas_kairėn (var m : medis);
begin
     dešinėn (m^.d);          {medis sukamas apie viršūnę k3}
    kairėn (m);               {medis sukamas apie viršūnę k1}
end;
```

Rekursijos realizacija kompiuteryje

Rekursija gali būti apibrėžta:

1. Apibrėžime: $u! := \begin{cases} 1, & u = 1; \\ u * (u - 1), & u > 1; \end{cases}$

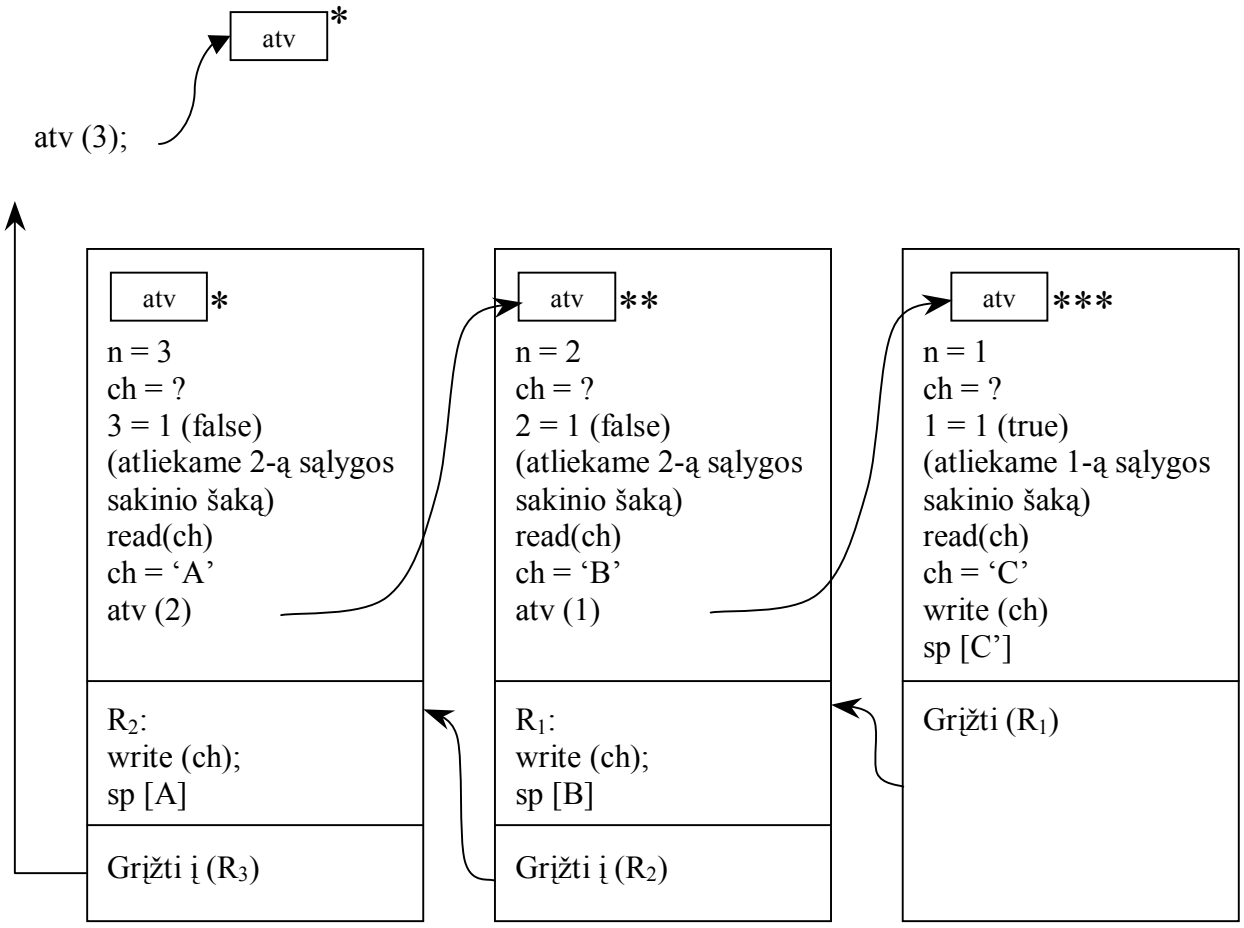
2. Funkcijos apraše;
3. Procedūros apraše.

Pavyzdžiai.
Turime rekursinę procedūrą, parašančią raides atvirkščia tvarka, nei jos yra įvestos:

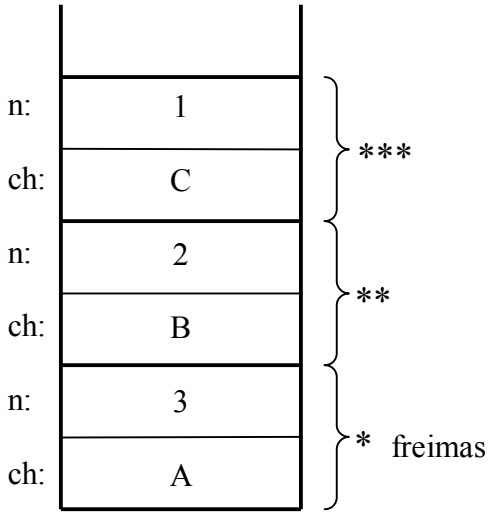
```
procedure atv (n : integer);
var
  ch : char;
begin
    if n = 1
      then begin
             readln (ch);
             write (ch);
           end
      else begin
             readln (ch);
             atv (n – 1);
             write (ch);
           end;
end;
```

procedūros vykdymas:

procedūroje pirmą kartą kreipiamės į ją pačią su n reikšme, lygia 3. Toliau vykdome kitus kreipinius:

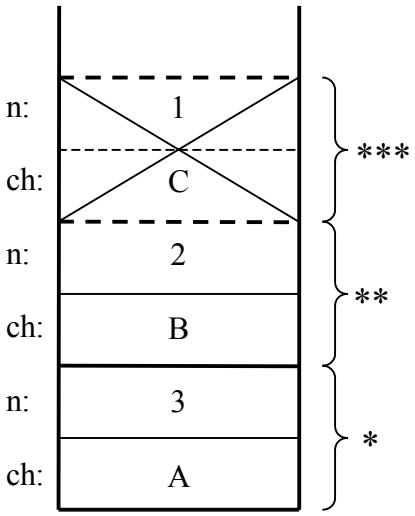


Tokie rekursiniai kreipiniai kompiuteryje naudoja tam tikrą duomenų struktūrą :



kiekvienas naujas kreipimasis į procedūrą sukuria naują freimą saugomoms reikšmėms.

Ši duomenų struktūra vadinama LIFO (Last In First Out) arba kitaip - stekas



Kai procedūroje atliekamas grįžimo veiksmas, kompiuterio atmintyje freimai yra pažingsniui naikinami (pirmas sunaikinamas tas, kuris buvo sukurtas paskutinis (LIFO struktūra))

Iš to, kas išdėstyta, galima daryti išvadą, jog, vykdant rekursiją kompiuteryje, freimų gali būti sukurta be galo daug, ir kompiuterio atmintis bus perkrauta. Tokiu atveju bus pateikta klaida:

202 stack overflow error

Taigi aprašant rekursiją reikia tikrinti, ar ji nėra begalinė (ar i rekursinę funkciją ar procedūrą nėra kreipiamasi be galo daug kartų).

Rekursinių funkcijų pavyzdžiai

1. Skaičiaus faktorialas

Parašysime rekursinę funkciją skaičiaus faktorialui apskaičiuoti:

```
function fakt (n: integer) : integer;
begin
  if n = 1
    then fakt = 1
    else fakt = n * fakt (n – 1);
end;
```

parašysime funkciją apskaičiuoti skaičiaus faktorialą nenaudodami rekursijos:

```
function fakt (n: integer): integer;
  var
    p, i : integer;
begin
    p := 1;
    for i := 1 to n do
    p := p * i;
    fakt := p;
end;
```

2. Fibonačio skaičiai.

Matematinis Fibonačio skaičių apibrėžimas yra toks: $f_0 = 0$, $f_1 = 1$, kiekvienas kitas yra gaunamas sudedant du skačius, esančius prieš jį. Užrašyta benra formule tai atrodo:

$$f_n = f_{n-1} + f_{n-2}$$

Parašysime rekursinę funkciją n-tajam Fibonačio skaičiui rasti:

```
function fib (n: integer): integer;
begin
    if n = 0
      then fib := 0
      else if n = 1
            then fib := 1
            else fib := fib (n – 1) + fib (n – 2);
end;
```

Funkcija n-tajam Fibonačio skaičiui be rekursijos atrodys taip:

```
function fib (n : integer): integer:
  var
      f1, f2, f, i : integer;
begin
    f1 := 0;
    f2 := 1;
    for i := 2 to n do
      begin
        f := f1 + f2;
        f2 := f;
        f1:= f2;
      end;
```

```
end;
```

3. Didžiausias bendras daliklis

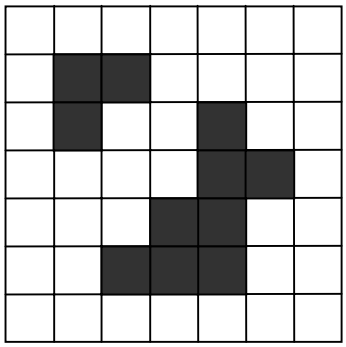
Algoritmas dižiausiam bendrajam dalikliui rasti matematiškai aprašomas taip (Euklido algoritmas):

r = a mod b (liekana, dalijant vieną skaičių iš kito)
jei r = 0, dbd = b
dbd (a, b) = dbd (b, r)

parašysime rekursinę funkciją dviejų skaičių didžiausiam bendrajam dalikliui rasti:

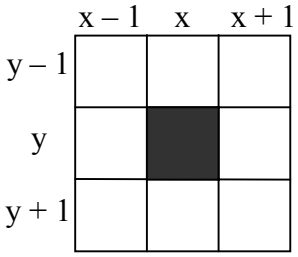
```
function dbd (a, b : integer): integer;
  var
      r : integer;
begin
    r := a mod b;
    if r = 0
      then dbd = 0
      else dbd : = dbd (b, r);
end;
```

4. Vietovės bakteriologinis užteršumas



Turime vietovę, kurią suskirstome langeliais (kiekvieno langelio koordinatės x ir y yra žinomos). Apkrėstą langelį žymėsime juoda spalva. Procedūros užduotis – rasti dėmės dydį (langelių skaičių) į kurį patenka nurodytas langelis.

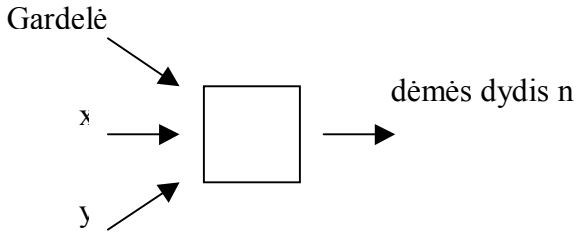
Jei nurodytas langelis patenka į dėmę, tai jis turi daugiausia 8 kaimyninius langelius, kurie gali būti arba užkrėsti, arba ne. Kiekvienas jų turi taip pat daugiausia 8 kaimynus, tie savo ruožtu – taip pat, ir t.t.
kaimyninių langelių koordinatės:



bakteriologinė dėmė bus apibrėžiama:

$$bd(x, y) \begin{cases} 0, \text{ jei langelis (su koordinatėmis x ir y) yra tuščias} \\ 1 + bd(x-1, y-1) + bd(x, y-1) + bd(x+1, y-1) + bd(x-1, y) + \\ + bd(x+1, y) + bd(x-1, y+1) + bd(x, y+1) + bd(x+1, y+1); \end{cases}$$

Reikia sudaryti funkciją:



Parašome funkciją šiems veiksmams atlikti:

const

```
  MaxX = 100;
  maxY = 100;

type
būsena = (tuščia, užpildyta);
gardtipas = array [1 .. MaxX, 1 .. MaxY] of būsena;

function Bdėmė ( gard: gardtipas; X, Y : integer): integer;
     function bd (X, Y: integer): integer;
     begin
       if (X < 1) or (X > MaxX) or (Y < 1) or (Y > MaxY)
          then bd := 0
          else if gard [X, Y] = tuščia
                  then bd := 0
                  else begin
                           gard [X, Y] := tuščia
                                bd := 1 + bd (x – 1, y – 1) + bd ( x, y – 1) + bd (x + 1, y - 1) +
    bd (x – 1, y) +  bd (x +  1, y) + bd ( x – 1, y + 1) + bd ( x, y + 1) + bd (x + 1, y + 1);

                       end;
     end;
begin
   Bdėmė := bd (X,Y);
end;
```