

## 7.4. Piramidės metodas

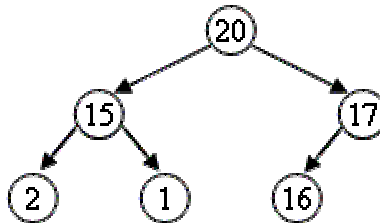
Aptarsime piramidės metodo rūšiavimo algoritmą.

Piramidė – tai speciali dvejetainio medžio rūšis. Paprastai medžiai piešiami šaknimi į viršų, o jo šakos eina žemyn ir baigiasi lapais. Medis gali būti tuščias arba sudarytas iš mazgų su nuorodomis į kitus nesikertančius medžius. Dvejetainis medis skiriasi nuo paprasto tuo, kad kiekvienas mazgas turi ne daugiau kaip dvi šakas. (Natūralu jas vadinti kairiąja ir dešiniąja šakomis). Medžio viršūnę vadinsime „tėvu“, o mazgus, į kurios nukreiptos „tėvo“ šakos vadinsime „sūnumis“. Kadangi kiekviena medžio šaka yra medis, kiekvienas mazgas yra „tėvas“ ir turi ne daugiau kaip du „sūnus“. Piramidė pasižymi šiomis savybėmis:

1. „Sūnų“ reikšmės visada mažesnės už „tėvo“;
2. Piramidę pildome po vieną lygį iš kairės į dešinę;
3. Pilnas medis turi  $2^n - 1$  elementų;
4. Maksimalus elementas visada viršūnėje;
5. i-ojo mazgo „sūnūs“ yra  $2*i$  ir  $2*i+1$  masyvo elementai.

**Pvz.:**

Turime piramidę:



Piramidė vaizduojama masyvu šitaip:

1:	2:	3:	4:	5:	6:
20	15	17	2	1	16

Piramidės metodo rūšiavimo algoritmas skirstomas į du etapus:

1. Iš rūšiuojamųjų elementų sudaroma piramidė;
2. Piramidė transformuojama į surūšiuotą masyvą.
  - 2.1. Pirmąjį masyvo elementą sukeičiame su paskutiniuoju;
  - 2.2. „Atstatome“ piramidę.

**Pvz.:**

1. Tarkime, rūšiuojamas masyvas yra toks:

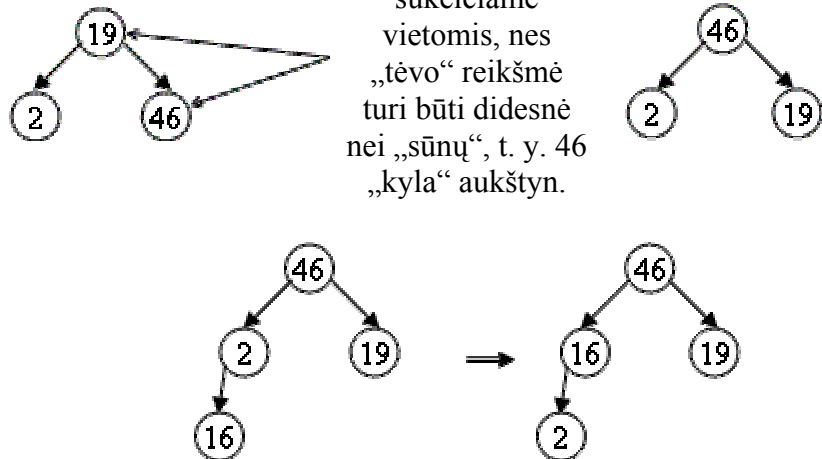
1:	2:	3:	4:	5:	6:	7:	8:	9:	10:	11:	12:
19	2	46	16	12	54	64	22	17	66	37	35

Iš jo sudarysime piramidę, imdami po vieną elementą pradedant nuo pirmojo:

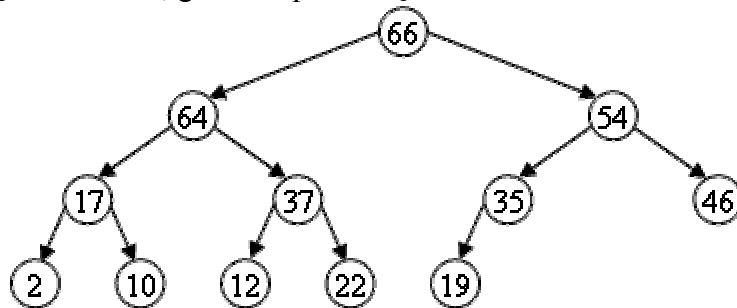


Tai piramidė  
Kadangi  $2 < 19$ , 2  
priskiriamas  
kairiajai šakai.  
Gautasis

dvejtainis medis -  
piramidė  
Elementus masyve  
sukeičiame  
vietomis, nes  
„tėvo“ reikšmė  
turi būti didesnė  
nei „sūnų“, t. y. 46  
„kyla“ aukšтын.



Tokiu būdu tęsdami toliau, gausime piramidę:



2. Šią  
piramidę  
atitinkantis  
masyvas:

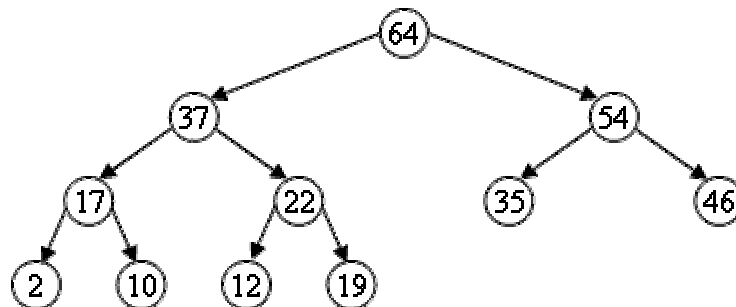
1:	2:	3:	4:	5:	6:	7:	8:	9:	10:	11:	12:
66	64	54	17	37	35	46	2	10	12	22	19

Transformuosime piramidę į surūšiuotą masyvą. Tai darome dviem etapais:  
sukeičiame pirmąjį elementą (nes jis didžiausias) su paskutiniu ir „atstatome“ piramidę.

2.1.Sukeičiame pirmąjį elementą su paskutiniu:

1:	2:	3:	4:	5:	6:	7:	8:	9:	10:	11:	12:
19	64	54	17	37	35	46	2	10	12	22	66

2.2.„Atstatome“ piramidę, tačiau į ją neįtraukiame paskutiniojo masyvo elemento.  
Gautoji piramidė atrodo taip:



Tokiu būdu tęsdami toliau gausime didėjimo tvarka surikiuotą masyvą:

1:	2:	3:	4:	5:	6:	7:	8:	9:	10:	11:	12:
2	10	12	17	19	22	35	37	46	54	64	66

Užrašysime rūšiavimo piramidės metodu algoritmą Paskaliu.

**procedure** Sukeisti (**var** m, n : integer);

**var** k : integer;

**begin**

    k := m;

    m := n;

    n := k

**end;**

**procedure** Žemyn (**var** A : masyvas; k : integer);

**var** s,                   {„sūnus“}

    t : integer;       {„tėvas“}

**begin**

    t := 1;

**while** true **do**

**begin**

**if** 2\*t > k **then** break;                   {t neturi „sūnų“}

            s := 2\*t;

**if** s+1 <= k **then**                   {yra 2 „sūnūs“}

**if** A[s+1] > A[s] **then** s := s+1;

**if** A[t] > A[s] **then** break;

            Sukeisti(A[t], A[s]);

            t := s

**end**

**end;**

**procedure** Aukštyn (**var** A : masyvas; k : integer);

**var** s,                   {„sūnus“}

    t : integer;       {„tėvas“}

**begin**

    s := k;

    t := s **div** 2;

**while** (A[t] < A[s]) **and** (s > 1) **do**

**begin**

            Sukeisti(A[s], A[t]);

            s := t;

**if** s > 1 **then** t := s **div** 2

**end**

**end;**

**procedure** piramidė (**var** A : masyvas; n : integer);

**var** i : integer;

**begin**

**for** i := 2 **to** n **do**                   {1 – as žingsnis}

        Aukštyn(A, i);

**for** i := n **downto** 2 **do**       {2 – as žingsnis}

```

begin
    Sukeisti(A[1], A[i]);
    Žemyn(A, i-1)
end
end;

```

**Piramidės metodo įvertinimas:**

1 etapo sudėtingumas:  $\sum_{i=1}^n \log_2 i \leq \sum_{i=1}^n \log_2 n = n * \log_2 n$

2 etapo sudėtingumas:  $n * \log_2 n$

Algoritmo sudėtingumas:  $O(n * \log_2 n)$ .

## 8. PAIEŠKOS ALGORITMAI

### 8.1. Dvejtainės paieškos algoritmas

Vienas iš svarbiausių programos darbo greičio padidinimo būdų yra sumažinti kartojimų skaičių cikle. Tai galime pasiekti, pavyzdžiui, surikiavę masyvą prieš atlikdami paiešką jame. Dvejtainės paieškos algoritmas ir yra taikomas, norint surasti elementą surūšiuotame masyve.

Tarkime, kad norime surasti elementą **a** surikiuotame masyve **M**. Ieškomo elemento indeksą **k** (**k** = 0, jei elemento masyve nėra) galime surasti tokiu būdu:

- Nustatome, koks yra vidurinio masyvo elemento indeksas: jei masyvas **M** turi **n** elementų, tai vidurinio elemento indeksas yra  $\frac{n+1}{2}$ . Jei paiešką atliekame tam tikrame intervale **[x, y]**, tai indeksas randamas pagal formulę:  $\frac{x+y}{2}$ ;
- Patikriname, ar ieškomas elementas yra didesnis už viduriniojo reikšmę. Jei taip – galime atmesti pirmąją masyvo dalį. Priešingu atveju – antrąją.
- Nustatome likusios masyvo dalies vidurinį elementą ir palyginame jį su ieškoma reikšme. Tokiu būdu paieškos intervalas sumažės dar du kartus.
- Kartosime minėtus veiksmus tol, kol rasime ieškomą elementą **a**, arba kol paieškos intervale liks tik vienas elementas.

**Pvz.:**

Tarkime, kad paiešką atliekame tokiaime surikiuotame masyve **M**:

1:	2:	3:	4:	5:	6:	7:
2	4	5	7	11	18	20

Ieškosime elemento **a** = 11. Randame vidurinio elemento indeksą:  $\frac{7+1}{2} = 4$ . Vidurinis

elementas yra ketvirtas ir lygus 7. Kadangi **a** > 7, tolesnę paiešką atliksime dešinėje masyvo pusėje. Naujas paieškos intervalas: [5; 7]. Vėl randame vidurinio nario indeksą:  $\frac{5+7}{2} = 6$ . **M**[6] = 18. Matome, kad **a** < 18, todėl paiešką tęsime kairėje masyvo dalyje.

Naujas paieškos intervalas: [5; 5]. Kadangi paieškos intervale liko tik vienas narys, patikriname, ar jis lygus ieškomajam elementui. Šiuo atveju jis yra lygus **a**. Taigi ieškomo elemento indeksas yra 5.

Pateiksime du dvejtainės paieškos algoritmo variantus Paskalio kalba.

### 1.1 variantas:

```
type mas = array[1..n] of integer;
var c,
    l,           {kairysis paieškos intervalo režis}
    r : integer; {dešinysis paieškos intervalo režis}
    rasta : boolean;
    M : mas;
begin
    l := 1;
    r := n;
    rasta := false;
    repeat
        c := (l + r) div 2;
        if a < M[c] then r := c - 1
            else if a > M[c] then l := c + 1
                else rasta := true;
    until rasta or (l > r);
    if rasta then ... {atliekame norimus veiksmus su rastu elementu}
end;
```

### 1.2 variantas:

Sąlyginį sakinį cikle galime pakeisti tokiu:

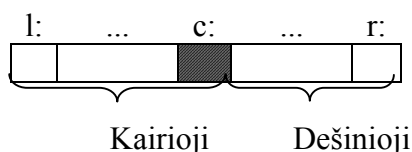
```
if a = M[c] then rasta := true
    else if a < M[c] then r := c - 1
        else r := c + 1;
```

### 2 variantas:

Paieškos intervalą dalijame pusiau tol, kol jame lieka tik vienas elementas. Ir tik pabaigę intervalo dalijimą patikriname, ar likęs vienas elementas yra lygus ieškomam.

```
l := 1;
r := n;
while r > l do
    begin
        c := (l + r) div 2;
        if a <= M[c] then r := c
            else l := c + 1;
    end;
if n > 0 then if a = M[c] then ... {elementas surastas; a = M[c]}
    else ... {elemento masyve nėra}
```

Pastebėsime, kad antru atveju vidurinį elementą priskiriame kairei pusei. Grafiškai tai atrodytų taip:



masyvo dalis      masyvo  
dalys

### Algoritmo įvertinimas:

Tegul  $N$  – masyvo elementų skaičius. Kadangi po kiekvieno palyginimo paieškos intervalas sumažėja pusiau, teisinga tokia formulė:  $C(N) = 1 + C\left(\frac{N}{2}\right)$ . Pažymėkime  $N = 2^n$ .

Tuomet:

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

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

...

$$C(2) = 1 + C(1);$$

$$C(1) = 0;$$

Iš čia gauname, kad  $C(2^n) = n$ . Kadangi  $N = 2^n$ ,  $C(N) = \log_2 N$ .

Dvejtainės paieškos algoritmo efektyvumą atspindi ši lentelė:

N	Palyginimų skaičius vykdant tiesinės paieškos algoritimą	Palyginimų skaičius vykdant dvejetainės paieškos algoritimą
8	4	3
1024	514	10
35768	17884	16

## 8.2. Maišos (hash) metodas

Aptarsime dar vieną elemento paieškos masyve algoritmą. Maišos (angl. – hash) metodu parašytas algoritmas pasižymi tuo, jog jo sudėtingumas  $O(1)$ . Kitaip tariant palyginimų skaičius nepriklauso nuo to, kiek elementų yra masyve, kuriame atliksime paiešką. Algoritmą galime suskirstyti į dvi dalis:

1. Elementų patalpinimas į masyvą (hash lentelę);
2. Elemento paieška.

Tarkime, turime  $n$  elementų, kuriuos reikia patalpinti į masyvą  $M$ . Kiekvieną elementą galime išdėstyti masyve taip, kad elemento indeksas būtų lygus jo reikšmei.

**Pvz.:**

Tarkime, skaičiai yra tokie: 20, 41, 5, 75, 11, 18, 2;

Iš jų galime sudaryti masyvą, kurio 2-as elementas yra 2, 5-as – 5, 11-as 11, 18-as – 18 ir t.t..

Turėdami tokį masyvą galime lengvai surasti ieškomą elementą **a** paprasčiausiai patikrindami ar egzistuoja masyve elementas  $M[a]$  (indeksas lygus reikšmei). Tačiau šis būdas yra netinkamas jei masyve yra gana didelių skaičių. Pavyzdžiui jei masyvo didžiausias elementas yra 100000 tuomet tektų sudaryti naują masyvą, kuriame yra 100000 elementų. Deja, to padaryti negalime. Kad to išvengtume sudarysime specialią masyvą, vadinamą „hash lentele“.

Tegul  $x$  – masyvo elementas. Parenkame skaičių  $p$  (dažnai patogų parinkti pirminį). Elementą  $x$  į hash lentelę patalpinsime tokiu būdu: indeksas =  $x \bmod p$ . Tačiau ir vėl susidursime su problemomis:

- Į hash lentelę galime patalpinti ne daugiau kaip  $p$  elementų. Todėl reikia  $p$  parinkti pakankamai didelį (didesnį nei elementų kiekis  $n$  masyve  $M$ );
- Kai kurių masyvo  $M$  elementų  $x$  dalybos iš  $p$  liekana gali sutapti, o tai reiškia, kad turėtume į vieną hash lentelės lauką patalpinti du ar daugiau elementų. To padaryti negalime, todėl, tokiu atveju, elementą patalpiname į artimiausią kitą tuščią lauką.

**Pvz.:**

Tarkime turime masyvą  $M$ :

1:	2:	3:	4:	5:	6:	7:	8:
1234	5021	7423	2000	9043	6296	6620	2013

Nustatome, kuriam hash lentelės laukui reikia priskirti kiekvieną elementą. Parenkame  $p = 13$ :

$1234 \bmod 13 = 12$ ;  
 $5021 \bmod 13 = 3$ ;  
 $7423 \bmod 13 = 0$ ;  
 $2000 \bmod 13 = 11$ ;  
 $9043 \bmod 13 = 8$ ;  
 $6296 \bmod 13 = 4$ ;  
 $6620 \bmod 13 = 3$ ;  
 $2013 \bmod 13 = 11$ ;

Hash

0:	7423
1:	2013
2:	
3:	5021
4:	6296
5:	6620
6:	
7:	
8:	9043
9:	
10:	
11:	2000
12:	1234

lentelė:

Kad galėtume nustatyti, ar laukas hash lentelėje jau užimtas, patogų naudoti papildomą lauką, kurio reikšmė yra *true*, jei laukas užimtas arba *false* priešingu atveju. Žemiau pateiktame algoritme laisvam laukui priskirsime reikšmę *maxint*.

Hash lentelės sudarymas Paskalio kalba:

```

const p = 100;
p1 = p - 1;
type raktotipas = integer;
infotipas = integer;
elem = record
    raktas : raktotipas;
    info : infotipas;
end;
indeksas = 0..p1;
lentelė = array[indeksas] of elem;

function hash (raktas : raktotipas) : indeksas;
begin
    hash := raktas mod p
  
```

```

end;

procedure init (var A : lentelė;
    var i : indeksas;
begin
    for i := 0 to p1 do
        A[i].raktas := maxint {Laukas laisvas, jei A[i].raktas = maxint}
    end;
    { užimtas, jei A[i].raktas <> maxint}

procedure įterpti (var A : lentelė;
    naujasraktas : raktotipas;
    naujasinfo : infotipas;
    var įterpimoindeksas : indeksas);
    var H : indeksas;
begin
    H := hash(naujasraktas);
    while A[H].raktas <> maxint do
        H := (H + 1) mod p;
    A[H].raktas := naujasraktas;
    A[H].info := naujasinfo;
    įterpimoindeksas := H
end;

```

Norint surasti skaičių hash lentelėje, reikia nustatyti ieškomo elemento dalybos iš **p** liekaną **H**. Tada, pradėdant **A[H]** elementu, tikriname, ar jis nėra lygus mūsų ieškomam skaičiui. Jei ne, tikriname ar **A[H+1]** nėra ieškomas skaičius ir t.t., kol elementas surandamas arba randama lauko reikšmė lygi *maxint* (tai reiškia, kad ieškomo skaičiaus masyve nėra). Atkreipkime dėmesį, kad hash lentelė negali būti visiškai užpildyta. Tokiu atveju joje nėra lauko, kurio reikšmė būtų *maxint* ir, jei ieškomo elemento hash lentelėje nėra, pateksime į „amžiną ciklą“. Elemento paieškos hash lentelėje algoritmas:

```

procedure paieška (var A : lentelė;
    ieškomasraktas : raktotipas;
    var rasta : boolean;
    var raktoindeksas : indeksas);
    var H : indeksas;
begin
    H := hash(ieškomasraktas);
    while (A[H].raktas <> ieškomasraktas) and (A[H].raktas <> maxint) do
        H := (H + 1) mod p;
    if A[H].raktas = ieškomasraktas then rasta := true
        else rasta := false;
    raktoindeksas := H
end;

```

## 9. DINAMINĖS DUOMENŲ STRUKTŪROS

### 9.1. Statiniai ir dinaminiai kintamieji

Pagal egzistavimo trukmę ir identifikavimą kintamieji skirstomi į 2 kategorijas:



1. Statiniai kintamieji;
2. Dinaminiai kintamieji.

Pateiksime programos, kurioje naudojami statiniai kintamieji pavyzdį:

```
program kintamieji;  
  var a, b : integer;  
  procedure p;  
    var x, y : real;  
  begin  
    x := 1;  
    y := 1  
  end;  
begin  
  a := 1; b := 2; p;  
  a := 2; b := 3; p;  
  a := 3  
end.
```

Šiame pavyzdyje visi naudojami kintamieji **a**, **b**, **x**, **y** yra statiniai, t. y. jie sukuriama programos vykdymo pradžioje (Paskalio kompiliatorius jiems išskiria tam tikrą atminties kiekį) ir egzistuoja visą programos vykdymo laiką. Tačiau kartais negalime atminties skirstyti statiškai, nes paprastai globaliniams kintamiesiems bei konstantoms išskiriama 64Kb (+16Kb lokaliniais kintamiesiems) atminties, ir toks atminties kiekis yra akivaizdžiai per mažas, jei reikia atlikti veiksmus pavyzdžiui su 500x500 realiųjų skaičių matrica. Tokiu atveju naudojame dinامينius kintamuosius, kurie yra sukuriama ir gali būti panaikinti programos vykdymo metu.

Dinaminų kintamųjų identifikavimui naudojami rodyklės tipo kintamieji. Rodyklė – tam tikras atminties adresas. Vartodami rodykles, į dinaminę sritį galime įrašyti bet kokio tipo duomenis, kurie gali užimti vieną, du ar daugiau iš eilės einančių atminties baitų. Rodyklė rodo tik į pirmą iš jų. Rodyklės yra dviejų tipų: netipizuotos ir tipizuotos. Pastarosios gali būti susietos su tam tikru duomenų tipu, t. y. objekto, į kurį rodo rodyklė tipas yra iš anksto žinomas. Pavyzdžiui, jei kintamojo aprašas yra „**var p** : ^integer;“, tai žymuo **p** reiškia tipizuotos rodyklės tipo kintamąjį **p**, kurio reikšmė – rodyklė (atminties adresas, kur yra saugomas koks nors objektas - šiuo atveju sveikasis skaičius), o **p^** reiškia dinaminį sveikąjo tipo kintamąjį, į kurį rodo ši rodyklė.

Dinaminiai kintamieji yra sukuriama su procedūra **new(p)**, kur **p** – rodyklės tipo kintamasis. Šis kintamasis gali būti pašalintas programos veikimo metu, naudojant procedūrą **dispose(p)**. Tuščios rodyklės (rodyklės, kuri niekur nerodo) reikšmė lygi **nil**.

**Pvz.:**

```
var ri : ^integer;
```

```
new(ri);      {sukuriamas dinaminis kintamasis}  
ri^ := 1;     {dinaminio kintamojo reikšmei priskiriamas 1}  
dispose(ri);  {sunaikinamas dinaminis kintamasis}
```

Aptarsime, kokie veiksmai yra galimi su dinaminiais kintamaisiais:

```
var i : integer;  
    ri1, ri2 : ^integer;
```

```

r : real;
rr : ^real;

```

```

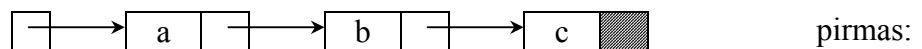
new(ri1);
ri1^ := 1;
new(ri2);
ri2^ := 2;

```

Galimi veiksmai	Negalimi veiksmai	Kodėl?
rr^ := ri1^; ri1 := ri2; ri1^ := ri2^;	ri1 := rr; rr := ri1;	ri1 – sveikojo tipo rodyklė, o rr – realaus, t. y. skirtingų tipų rodyklės negali būti priskirtos viena kitai
	ri1^ := rr^;	Sveikojo tipo dinaminiam kintamajam negalime priskirti realaus tipo dinaminio kintamojo (lygiai taip pat, kaip ir statiniam sveikojo tipo kintamajam negalime priskirti realaus tipo statinio kintamojo).

## 9.2. Sarašai

Sarašas – dinaminė struktūra, kurios kiekvienas elementas sudarytas iš dviejų pagrindinių dalių: informacinės dalies ir rodyklės į kitą elementą. Simboliškai tai būtų galima pavaizduoti taip:



Sąrašo aprašas Paskalio kalba:

```

type sąrašas = ^elementas;
      elementas = record
        info : integer;
        kitas : sąrašas
      end;

```

Kaip matome, sąrašo aprašas yra rekursinis. Šiuo atveju informacinę dalį sudaro sveikojo tipo dinaminis kintamasis. Laukas **kitas** saugo kito elemento adresą atmintyje. Kadangi po paskutinio elemento neina joks kitas elementas, lauko reikšmė lygi **nil**.

Panagrinėkime pavyzdį:

```

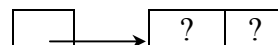
var pirmas : sąrašas;

```

```

new(pirmas);

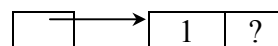
```



```

pirmas^.info := 1;

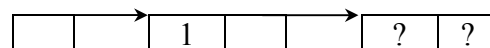
```



```

new(pirmas^.kitas);

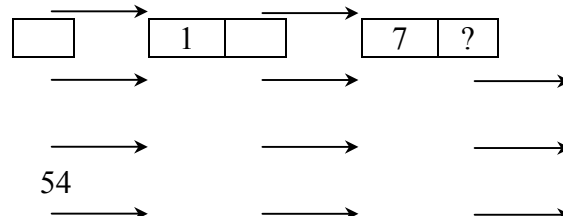
```






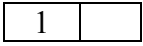
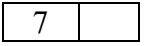
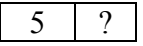

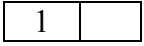
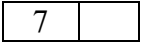



```

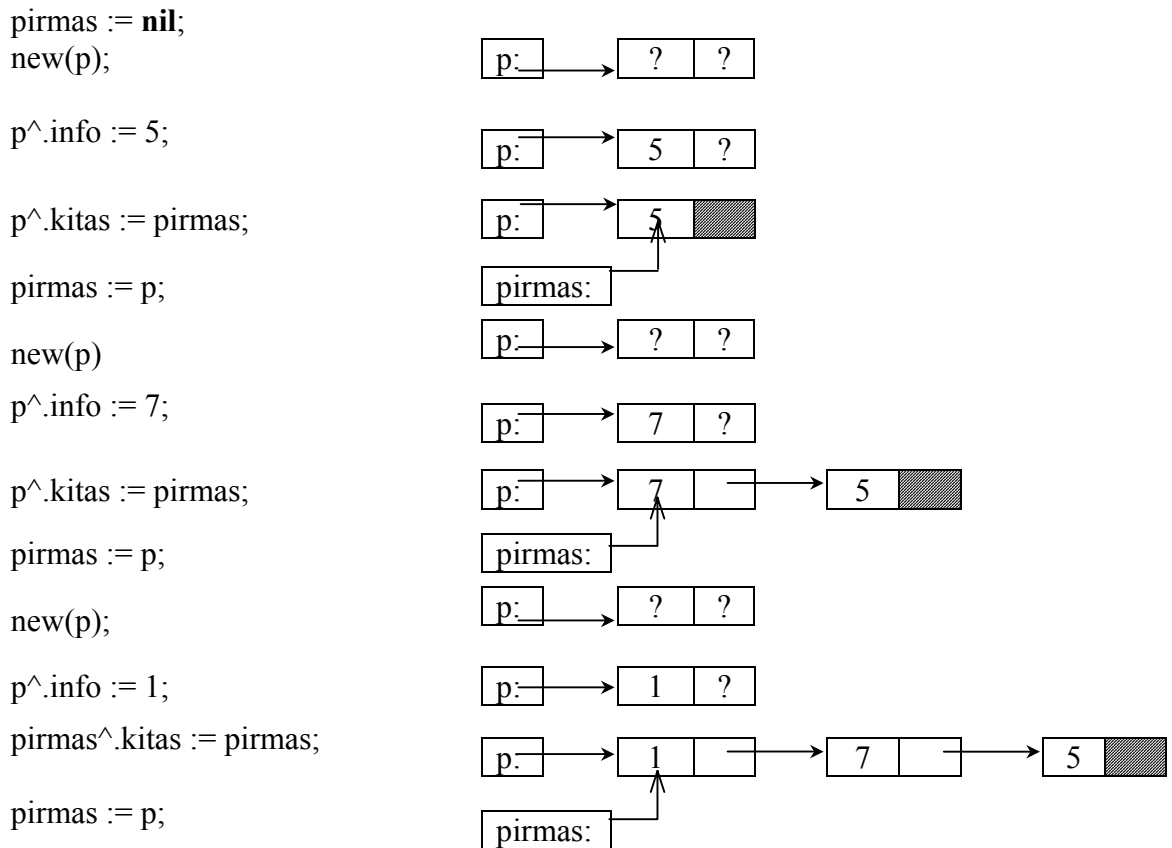
pirmas^.kitas^.info := 7;

```



<code>new(pirmas^.kitas^.kitas);</code>				
<code>pirmas^.kitas^.kitas^.info := 5;</code>				
<code>pirmas^.kitas^.kitas^.kitas := nil;</code>				

Tą patį sąrašą galime sudaryti ir kitu būdu:



Pateiktieji sąrašo sudarymo algoritmai yra neuniversalūs. Todėl sąrašą paprastai sudarome tokiu būdu:

```

pirmas := nil;
readln(x);
while x <> 0 do
begin
  new(p);
  p^.info := x;
  p^.kitas := pirmas;
  pirmas := p;
  readln(x)
end;

```

Sąrašo spausdinimas:

```

while p <> nil do
begin

```

```

        writeln(p^.info);
        p := p^.kitas
    end;

```

### **Elemento paieška sąrašė**

Pateiksime elemento paieškos pagal raktą **L** dinamiame sąrašė algoritmą Paskalio kalba (raktui sukuriame papildomą lauką):

```

elementas = record
    raktas : integer;
    info : integer;
    kitas : sąrašas
end;

```

**1 variantas** (blogas, jei elemento nėra sąrašė);

```

q := pirmas;
while q^.raktas <> L do
    q := q^.kitas;

```

**2 variantas**

```

q := pirmas;
rasta := false;
while not rasta and (q <> nil) do
    if q^.raktas = L then rasta := true
        else q := q^.kitas;
if rasta then writeln(q^.info)
        else writeln('Nerasta');

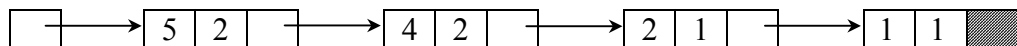
```

### **„Policijos“ uždavinys.**

Tarkime, registruojame eismo pažeidėjus. Reikia suskaičiuoti, kiek kartų kuris pažeidėjas nusižengia. Kadangi nežinome tikslaus pažeidėjų skaičiaus, šiam uždaviniui spręsti panaudosime dinامينius sąrašus. Kiekvieną pažeidėją identifikuosime sveiku skaičiumi.

**Pvz.:**

Tegul, pažeidėjai yra: 5, 4, 2, 5, 1, 4. Iš čia matome, kad 1–as pažeidėjas padarė 1 nusižengimą, 2–as – 1, 4–as – 2 ir 5–as – 2. Simboliškai tai galima pavaizduoti taip:



Kaip matome, šį uždavinį lengva spręsti naudojant dinaminį sąrašą, kurio informacinę dalį sudaro du laukai: pažeidėjo numeris ir pažeidimų skaičius. Jei reikia užregistruoti naują pažeidėją, pakanka į sąrašą įterpti naują elementą.

Pateiksime uždavinio sprendimą Paskalio kalba:

**program** policija;

```

type sąrašas = ^elem;
  elem = record
    NR : integer;
    kiek : integer;
    kitas : sąrašas
  end;

var k : integer;
  p : sąrašas;

procedure įterpti (var s : sąrašas; x : integer);
  var w : sąrašas;
  nerasta : boolean;
begin
  nerasta := true;
  w := s;
  while (w <> nil) and nerasta do
    if w^.NR = x then nerasta := false;
      else w := w^.kitas;
    if nerasta then begin
      new(w);
      w^.NR := x;
      w^.kiek := 1;
      w^.kitas := s;
      s := w
    end
    else
      w^.kiek := w^.kiek + 1
    end;
end;

procedure spausdinti (s : sąrašas);
begin
  while s <> nil do
    begin
      writeln(s^.NR : 9, s^.kiek : 5);
      s := s^.kitas
    end
  end;

begin {pagrindinė programos vykdomoji dalis}
  p := nil;
  readln(k);
  while k <> 0 do
    begin
      įterpti(p, k);
      readln(k);
    end;
    spausdinti(p)
  end;

```

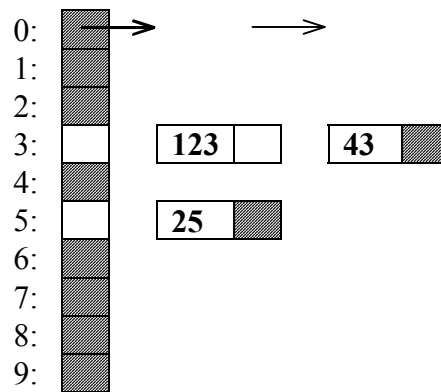
### Paieška hash (maišos) lentelėje

Tarkime, reikia patalpinti į hash lentelę 3 skaičius: 123, 43, 25. Parenkame skaičių  $p = 10$ . Tačiau skaičių 123 ir 43 dalybos iš 10 liekanos sutampa ir dėl to atsiranda kolizija. Šią problemą sprendėme patalpindami skaičių į artimiausią kitą laisvą lauką. Toks hash lentelės sudarymo būdas vadinamas tiesiniu.

Hash lentelė:

0:	
1:	
2:	
3:	123
4:	43
5:	25
6:	
7:	
8:	
9:	

Pastebėsime, jog patogu hash lentelę sudaryti grandininio būdu, naudojant dinامينius sąrašus:



Šiuo atveju naudojame  $p$  dinaminio sąrašų masyvą. Talpinamo skaičiaus dalybos iš  $p$  liekana atitinka vieno iš dinaminio sąrašų indeksą, pvz.  $123 \bmod 10 = 3$ . Todėl skaičių 123 priskirsime sąrašui, kurio indeksas 3. Sudarinėjant hash lentelę grandininio būdu lengvai išvengsime kolizijos, nes tuo atveju kai liekanos sutampa, pakanka į atitinkamą sąrašą įterpti naują elementą.

```
const M = 100;  
    M1 = M - 1;  
type sąrašas = ^elem;  
    elem = record  
        raktas : integer;  
        info : integer;  
        kitas : sąrašas  
    end;  
lentelė = array [0..M1] of sąrašas;  
procedure init (var A : lentelė);  
var i : integer;  
begin  
    for i := 0 to M1 do  
        A[i] := nil
```

```

end;

procedure įterpti (var A : lentelė;
                  naujas_raktas : integer;
                  naujas_info : integer);
var H : integer;
    p : sąrašas;
begin
    H := hash(naujas_raktas);
    new(p);
    p^.raktas := naujas_raktas;
    p^.info := naujas_info;
    p^.kitas := A[H];
    A[H] := p
end;

procedure paieška (var A : lentelė;
                  ieškomas_raktas : integer;
                  var rasta : boolean;
                  var rasta_info : integer);
var H : integer;
    p : sąrašas;
begin
    H := hash(ieškomas_raktas);
    p := A[H];
    rasta := false;
    while (p <> nil) and not rasta do
        if p^.raktas = ieškomas_raktas then rasta := true
            else p := p^.kitas;
        if rasta then rasta_info := p^.info
    end;
end;

```

Tiesinio ir grandininio hash lentelės sudarymo būdų palyginimas:

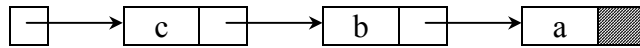
Būdas	Privalumai	Trūkumai
Tiesinis	Greitesnis	Elementų kiekis ribotas: $n \leq M$
Grandininis	Neribojamas elementų kiekis n	Lėtesnis

### 9.3. Stekas

Stekas – dinaminė duomenų struktūra. Steką pildome nuo pabaigos į pradžią, o elementus apeiname nuo pradžios į pabaigą. (t. y. paskutinis elementas, kurį įdėjome į steką bus atspausdintas pirmuoju, priešpaskutinis – antruoju ir t. t.). Stekas sudaromas panašiai kaip ir dinaminis sąrašas.

**Pvz.:**

Sakykime patalpiname į steką tris elementus: a, b, c. Tuomet steką simboliškai galima vaizduoti taip:



Matome, jog stekė elementų tvarka yra atvirkščia. Apeinant šiuos elementus gausime seką c, b, a.

Pateiksime pavyzdį Paskalio kalba:

```

type stekas = ^elem;
  elem = record;
    info : integer;
    kitas : stekas
  end;

```

```

procedure į_steką (var s : stekas; i : integer);
  var p : stekas;
begin
  new(p);
  p^.info := i;
  p^.kitas := s;
  s := p
end;

```

```

procedure iš_steko (var s : stekas;
  var yra : boolean;
  var i : integer);
begin
  if s = nil then yra := false {stekas yra tuščias}
  else
    begin
      yra := true;
      i := s^.info;
      s := s^.kitas;
    end
  end;

```

```

var s : stekas;
  i : integer;

```

...

```

s := nil; {stekas padaromas tuščias}
į_steką(s, 4);
į_steką(s, 7);

```

...

```

iš_steko(s, taip, i);
if taip then {apdorojame elementą i}
  else {veiksmas, kai stekas tuščias}

```

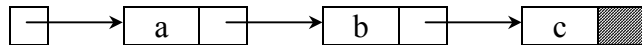


## 9.4. Eilė

Eilė – dinaminė duomenų struktūra, panaši į steką. Pagrindinis skirtumas yra tas, jog eilę pildome nuo pradžios į pabaigą ir elementus apeiname nuo pradžios į pabaigą.

**Pvz.:**

Sakykime patalpiname į eilę tris elementus: a, b, c. Tuomet eilę simboliškai galima vaizduoti taip:

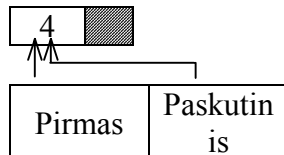


Apeinant eilės elementus gausime seką a, b, c.

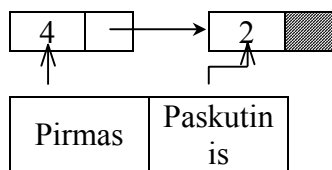
**Pvz.:**

Pateiksime eilės sudarymo pavyzdį. Sakykime, reikia sudaryti eilę iš trijų skaičių: 4, 2, 8.

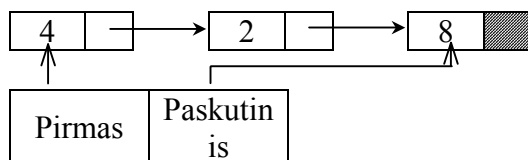
Eilė, sudaryta iš vieno elemento. Jos pradžia ir pabaiga sutampa:



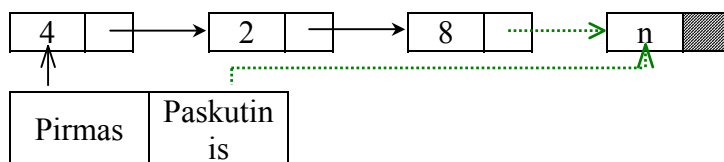
Eilė, sudaryta iš dviejų elementų:



Eilė, sudaryta iš trijų elementų:



Naujo elemento patalpinimo į eilę shema:



**type** sąrašas = ^elem;

elem = **record**;

info : integer;

kitas : stekas

**end**;

eilė = **record**

pirmas, paskutinis : sąrašas

```

    end;

    procedure į_eilę (var e : eilė; i : integer);
        var p : sąrašas;
    begin
        new(p);
        p^.info := i;
        p^.kitas := nil;
        e.paskutinis^.kitas := p;
        e.paskutinis := p
    end;

    procedure iš_eilės (var e : eilė;
        var yra : boolean;
        var i : integer);
    begin
        if e.pirmas = nil then yra := false           {eilė tuščia}
        else begin
            yra := true;
            i := e.pirmas^.info;
            e.pirmas := e.pirmas^.kitas
        end
    end;

    ...
    var ee : eilė;
        ii : integer;
        taip : boolean;
    ...
    ee.pirmas := nil; {eilė padaroma tuščia}
    ...
    į_eilę(ee, 4);
    ...
    į_eilę(ee, 7);
    ...
    iš_eilės(ee, taip, ii);
    if taip then {apdoroti ii}
        else {veiksmas, kai eilė tuščia}

```

## 9.5. Dvejjetainiai medžiai

Tiek steko tiek eilės elementai turi vieną rodyklės lauką, saugantį tolesnio elemento adresą. Dėl to tokias dinamines struktūras galime peržiūrėti tik viena kryptimi. Tačiau kartais naudinga numatyti keletą rodyklių laukų. Dinamines struktūras, kurių elementai turi daugiau nei vieną rodyklę, vadiname daugiaryšėmis.

Viena iš pagrindinių daugiaryšių dinaminių struktūrų yra dvejetainis medis. Medį apibūdinti galime taip: medis yra arba tuščias arba sudarytas iš elementų su rodyklėmis į nesusikertančius medžius. Dvejjetainis medis ypatingas tuo, jog kiekvienas elementas turi ne daugiau kaip dvi rodykles. Nesunku pastebėti, jog medžio struktūra yra rekursinė:

```

type medis = ^elem;
    elem = record

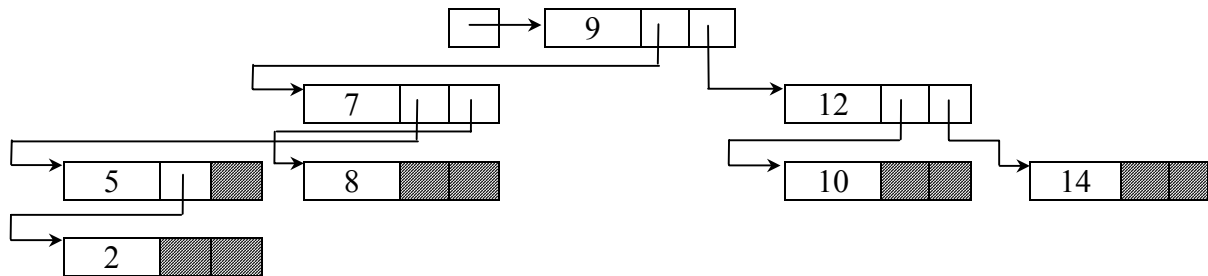
```

```

info : integer;
kairė, dešinė : medis
end;

```

Simboliškai dvejetainį medį, sudarytą iš skaičių 9, 7, 12, 5, 8, 10, 14, 2 galime pavaizduoti taip:



Įterpdami naują elementą į medį, palyginame jį su viršūnėje esančiu elementu. Jei įterpiamasis elementas yra mažesnis už viršūnėje esantį elementą, įterpiamąjį priskirime kairiajai šakai, priešingu atveju – dešiniajai. Kadangi abi šakos yra dvejetainiai medžiai, tą pačią procedūrą kartojame tol, kol „ateiname“ iki elemento, neturinčio tos šakos, kuriai reikia priskirti įterpiamąjį elementą. Pateiksime naujo elemento įterpimo į dvejetainį medį algoritmą Paskalio kalba:

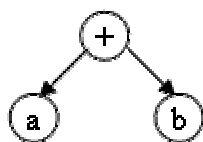
```

procedure į_medį (var m : medis; i : integer);
begin
  if m = nil then begin
    new(m);
    m^.info := i;
    m^.kairė := nil;
    m^.dešinė := nil
  end
  else if i < m^.info then į_medį(m^.kairė, i)
    else į_medį(m^.dešinė, i);
end;

```

Šio algoritmo sudėtingumas  $O(\log_2 n)$ .

Tarkime turime dvejetainį medį:



Yra 3 dvejetainio medžio elementų atspausdinimo (apėjimo) būdai:

- InOrder (atspausdins „a+b“);
- PreOrder (atspausdins „+ab“);
- PostOrder (atspausdins „ab+“).

Pateiksime InOrder algoritmą:

```

procedure SpausdInOrder (m : medis);

```

```

begin
  if m <> nil then
    begin
      SpausdInOrder(m^.kairė);
      writeln(m^.info);
      SpausdInOrder(m^.dešinė);
    end;
  end;

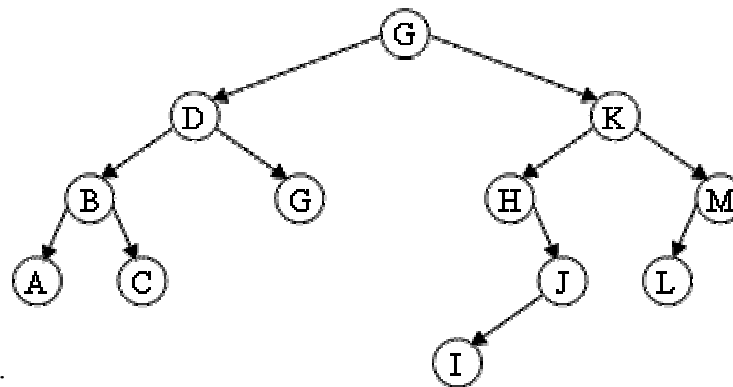
```

### Dvejtainio medžio elemento pašalinimas

Norint pašalinti dvejetainio medžio elementą galimi keturi atvejai:

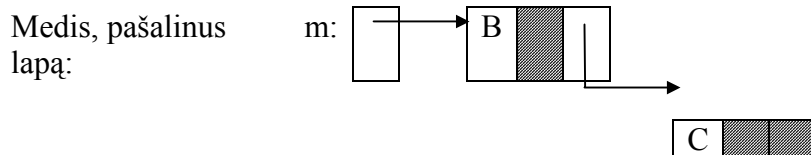
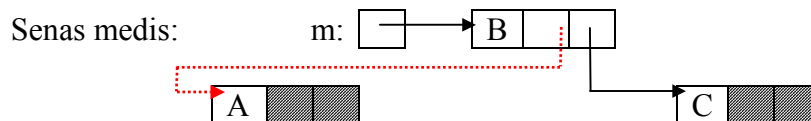
1. Šalinamas elementas yra „lapis“;
2. Šalinamas elementas neturi kairės šakos;
3. Šalinamas elementas neturi dešinės šakos;
4. Šalinamas elementas turi abi šakas.

Aptarsime visus keturis atvejus. Tarkime turime dvejetainį

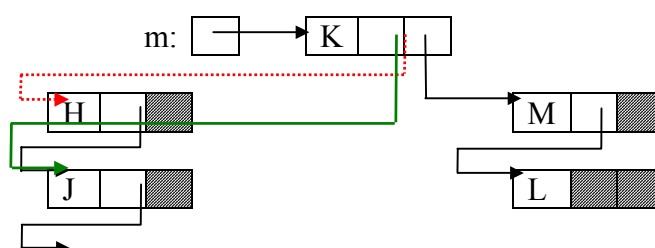


medį:

1. Elementai „lapai“ yra šie: A, C, E, I, L. Norint pašalinti bet kurį iš šių elementų pakanka panaikinti rodyklę, jungiančią „lapą“ su medžiu. Šis pašalinimo būdas yra korektiškas, t. y. nesugriaunama dvejetainio medžio struktūra. Schematiškai „lapo“ A pašalinimas atrodo taip:

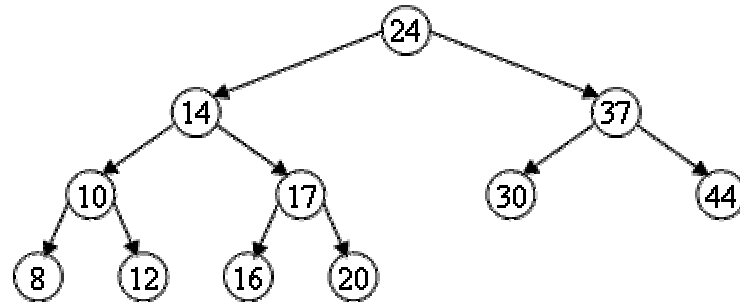


2. Elementas, neturintis kairės šakos yra H. Norėdami pašalinti šį elementą, rodyklę m^.kairė nurodome ne į elementą H, o į J. Šio elemento pašalinimo schema:

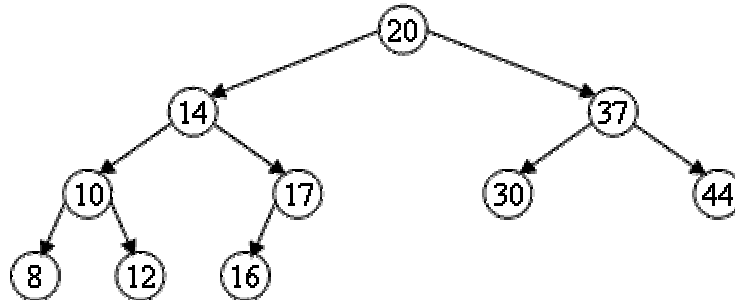


Toks pašalinimo būdas yra korektiškas.

3. Elementai, neturintys dešinės šakos yra: M ir J. Juos šaliname kaip ir 2 atveju.
4. Elementai, turintys abi šakas yra G, D, K, B. Paprastumo dėlei panagrinėkime atskirą pavyzdį:



Šalinsime medžio viršūnę 24. Pašalindami šį elementą sugriautume medžio struktūrą. Kad to išvengtume 24 galime pakeisti kairės šakos didžiausiu elementu (šiuo atveju 20) arba dešinės šakos mažiausiu elementu (30). Tai atlieka žemiau pateikta funkcija „pirmtakas“. Naujas dvejetainis medis atrodo taip:



Elemento pašalinimas Paskalio kalba:

```

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 ... {šaliname elementą}
      else if e > m.info then pašalinti(m.d, e)
      else pašalinti(m.k, e)
    end
  end;

  {šaliname elementą}
begin

```

```

    if m^.k = nil then      {elementas neturi kairės šakos}
        begin
            r := m;
            m := m^.d;
            dispose(r);
        end
    else if m^.d = nil then  {elementas neturi dešinės šakos}
        begin
            r := m;
            m := m^.k;
            dispose(r);
        end
    else                    {elementas turi abi šakas}
        begin
            pirm := pirmtakas(m);
            m^.info := pirm^.info;
            pašalinti(m^.k, m^.info)
        end
    end
end;

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

```

### Duomenų bazės pavyzdys

Pateiksime paprasčiausios duomenų bazės, realizuojančios dvejetainį medį, pavyzdį Paskalio kalba. DB valdymui naudosime tris operatorius:

I – elemento įterpimui;

Q – elemento paieškai;

D – elemento pašalinimui.

Tarkime norime įterpti 50 elementą - rašome: „I 50“, surasti – rašome: „Q 50“ ir pan.

```

program PaprastaDB;
type medis = ^mazgas;
    mazgas = record
        info : integer;
        k, d : medis
    end;
procedure įterpti...
procedure spausd...
procedure pašalinti...
function pirmtakas...
function kiek (m : medis) : integer;
begin

```

```

    if m = nil then kiek := 0
    else kiek := 1 + kiek(m^.k) + kiek(m^.d)
end;
function gylis (m : medis) : integer;
begin
    if m = nil then gylis := 0
    else gylis := 1 + max(gylis(m^.k), gylis(m^.d))
end;
var BD : medis;
    kodas : char;
    i : integer;
begin
    DB := nil;    {Pradinė duomenų bazė padaroma tuščia}
    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(DB, i)
            end;
            writeln('DB turinys:');    {DB turinio spausdinimas}
            spausd(DB);
            writeln('DB gylis:');    {Nustatomas DB gylis}
            writeln(gylis(DB));
            writeln('DB elementų:'); {Nustatomas įrašų kiekis DB}
            writeln(kiek(DB));
        end;
    end;
end;

```

Paieškos arba įterpimo sudėtingumas  $O(\log_2 n)$ .

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

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

## 9.6. 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 Landin.

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

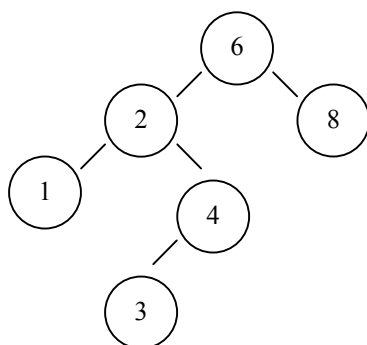
## Dvejetainių (paprastų medžių) ir AVL medžių palyginimas

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

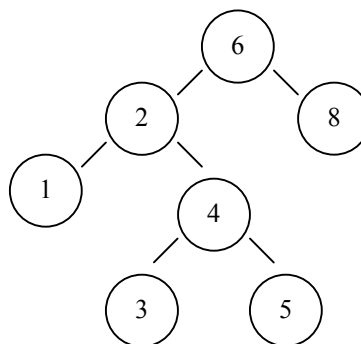
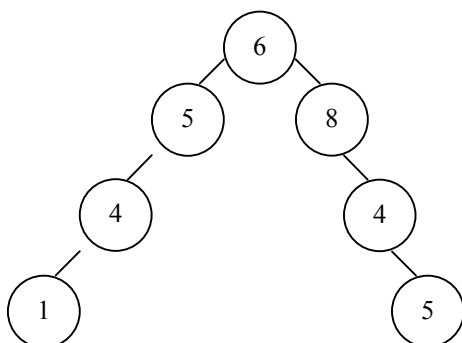
$n$  – elementų (mazgų) kiekis medyje

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;

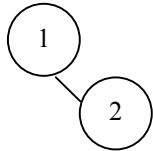
Pvz. Į AVL medį įterpiame tokius skaičius: 1, 2, 3, 4, 5, 6, 7.



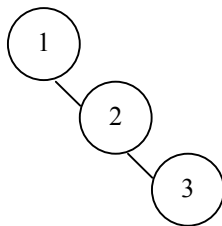
Įterpiame 1:



Įterpiame 2:

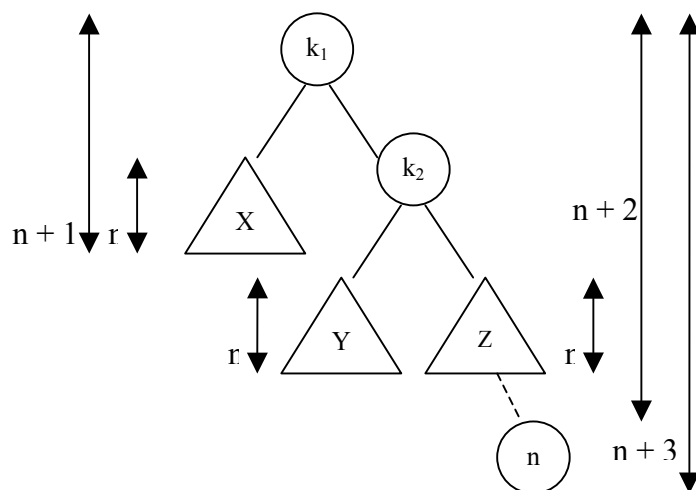


Įterpiame 3:



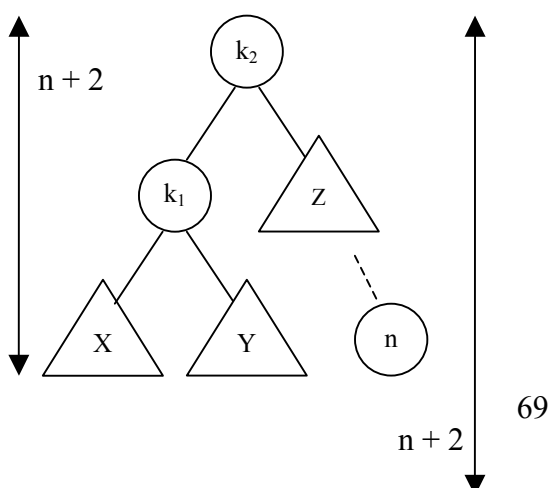
Matome, kad tai nėra AVL medis. Vadinasi, Reikia atlikti tam tikrą transformaciją, kuri transformuotų duotą medį į AVL medį.

### Posūkis kairėn.



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

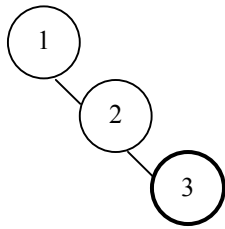
Atlikus posūkį:



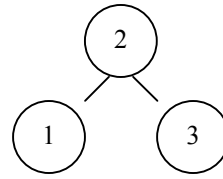
Kai jau atliktas posūkis, tai gauname AVL medį.  
Įrodome, kad tai korektiškas AVL medis.  
Medyje, prie kurio prijungėme elementą  $n$ , turime:  
 $k_1 < k_2$   
 $X < k_1$   
 $Y > k_1$

Matome, kad tokius pat santykius turime ir transformuotame medyje.

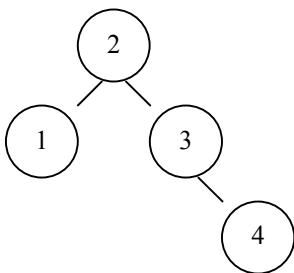
Pavyzdyje turejome:



Transformuojame (atliekame posūkį kairėn):



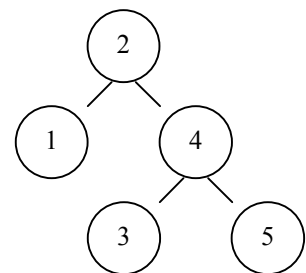
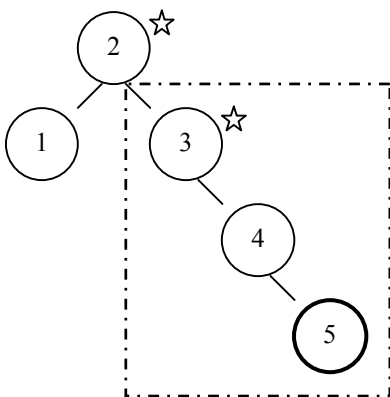
Prie gauto medžio prijungiamo 4:



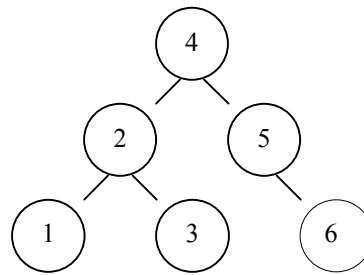
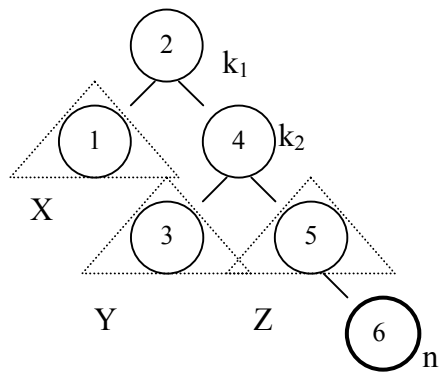
Gautas medis – taisyklingas AVL medis.

Prijungiamo 5:

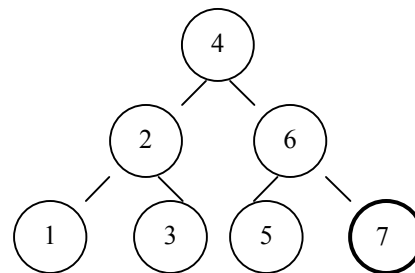
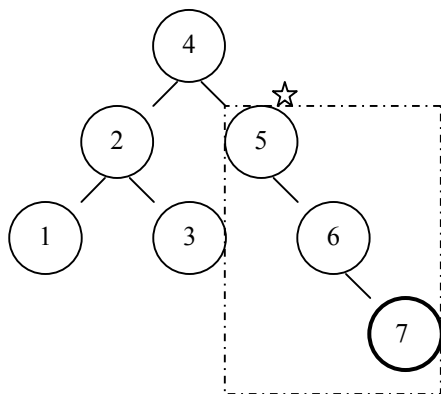
Turime du pažeidimus AVL medžio struktūroje. Taikysime posūkį kairėn (pažymėtą medžio šaką (submedį) transformuojame ir prijungiamo prie pradinio medžio):



Prijungiame 6 ir atliekame transformaciją kairėn:



Prijungiame 7 ir atliekame transformaciją kairėn:

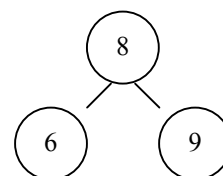
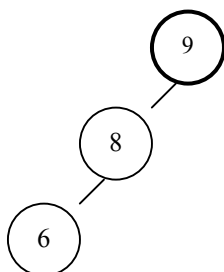


### Posūkis dešinėn.

Posūkis dešinėn atliekamas analogiškai kaip ir posūkis kairėn, pvz:

Turime medį:  
medis:

Atlikus posūkį dešinėn, gautas AVL

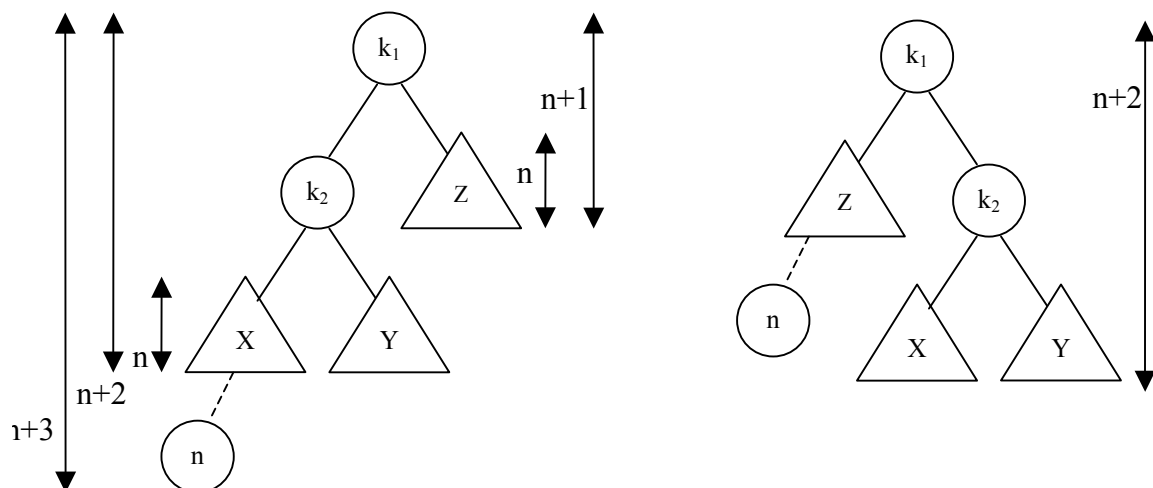


Bendra taisyklė posūkiui dešinėn.  
medis:

Atlikus posūkį dešinėn, gautas AVL

Turime tokios struktūros medį (X,  
Y, Z – submedžiai), į kurį

Itrauktas naujas elementas  $n$  (lapas):



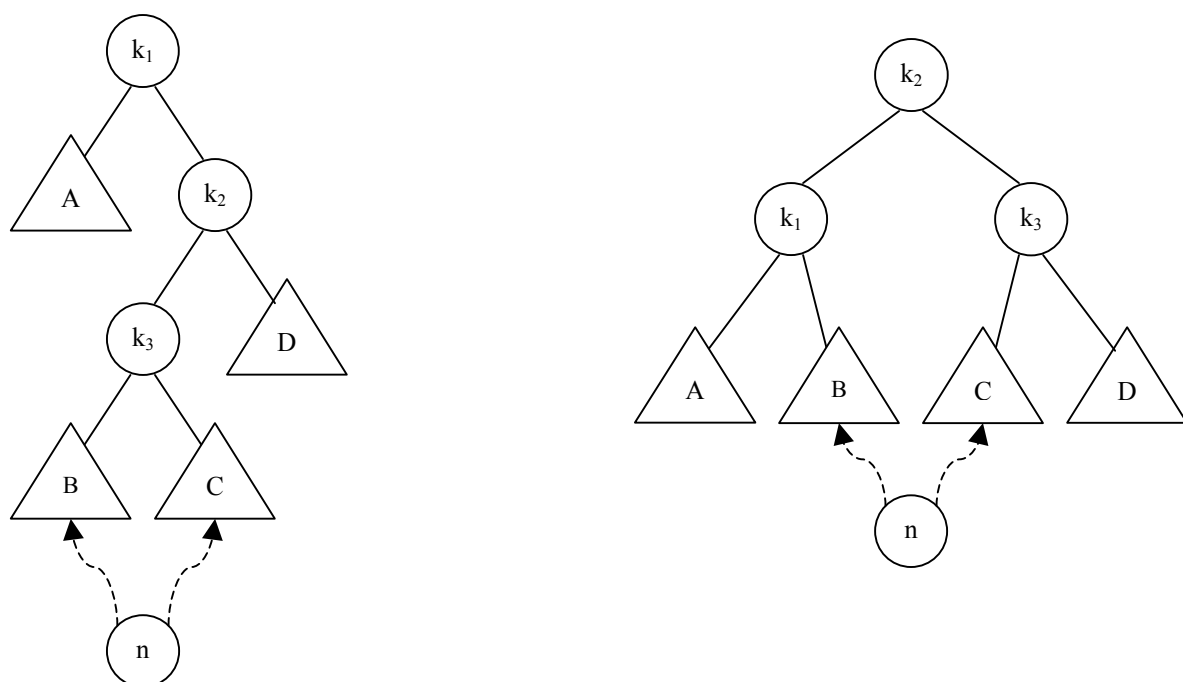
Kad gautasis medis – korektiškas AVL medis, įrodome panašiai, kaip ir posūkio kairėn atveju.

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

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

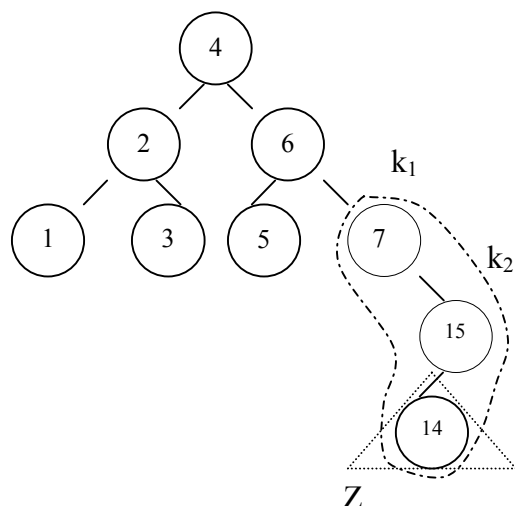
### Dvigubas posūkis.

Apibendrinta dvigubo posūkio kairėn taisyklė.

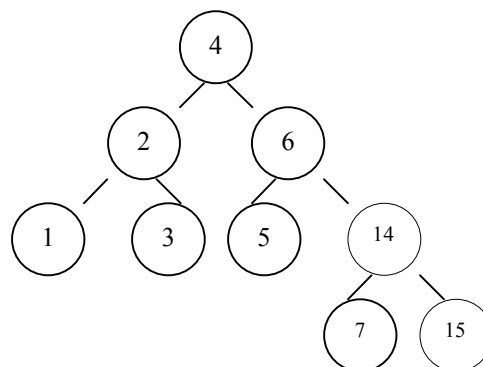


Pavyzdžiui, prie jau turimo AVL medžio prijunkime skaičių 15 (prijungus tik 15, gauname tvarkingą AVL medį) ir skaičių 14:

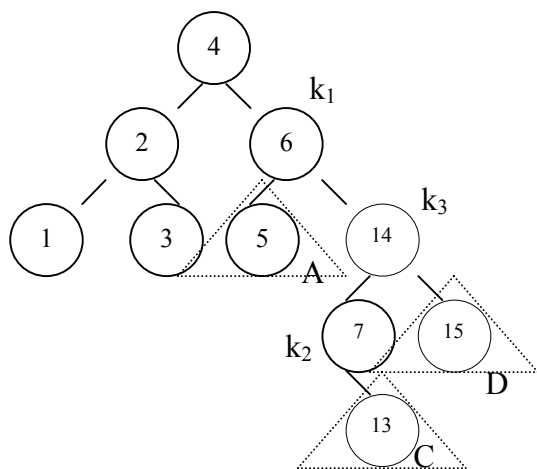
Gautas medis nėra AVL medis.



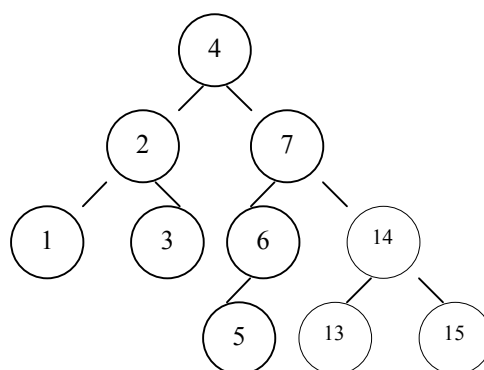
Atliekame dvigubą posūkį kairėn ir gauname taisyklingą AVL medį:



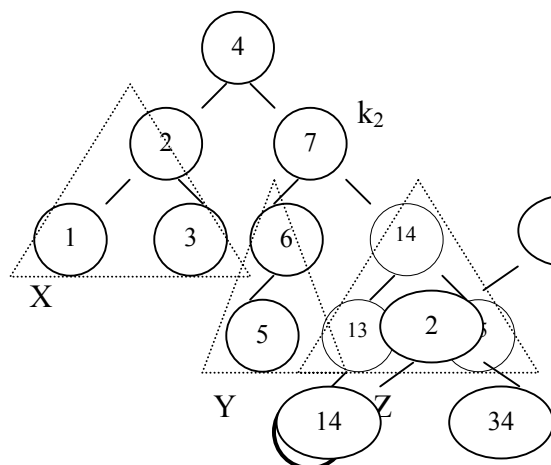
Į turimą medį įjungsime skaičių 13:



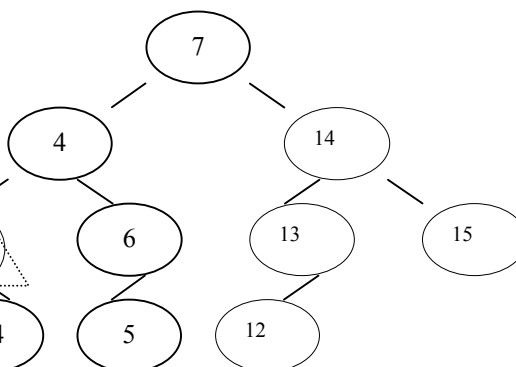
Atliekame dvigubą posūkį kairėn:



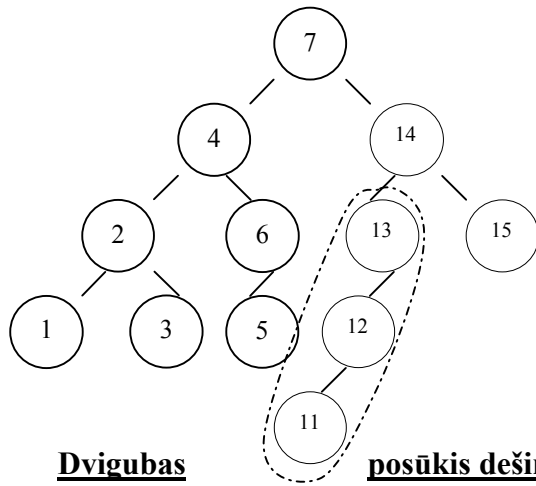
Į turimą medį įjungiame 12:



Transformuojame:

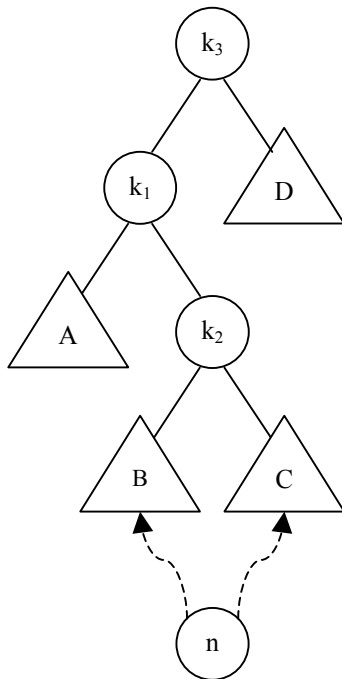


Į turimą medį įjungiame skaičių 11:  
atliekame

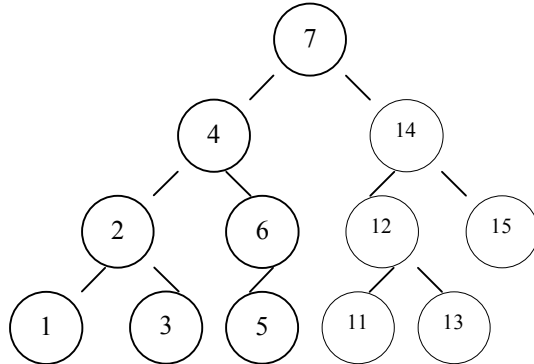


**Dvigubas** **posūkis dešinėn.**

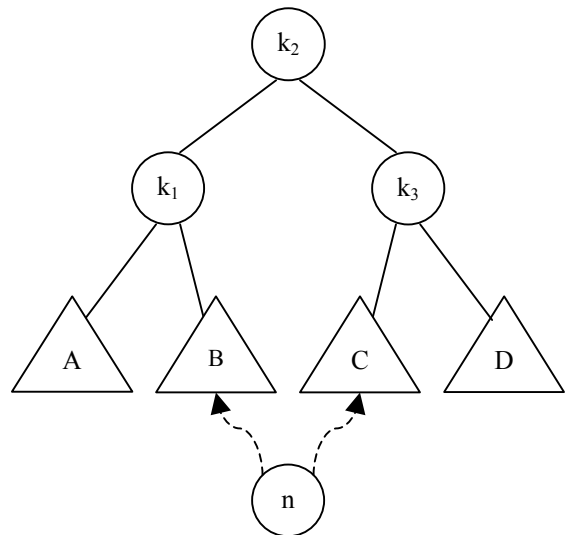
Apibendrinta taisyklė. Turime medį, prie kurio šakos B arba C prijungiame naują elementą n. Medį reikia transformuoti taip, kad gautume taisyklingą AVL medį.



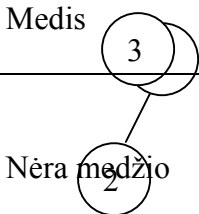
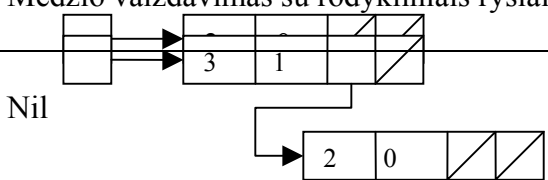
Šiam medžiui transformuoti į AVL medį  
tik paprastą (nedvigubą) posūkį dešinėn:



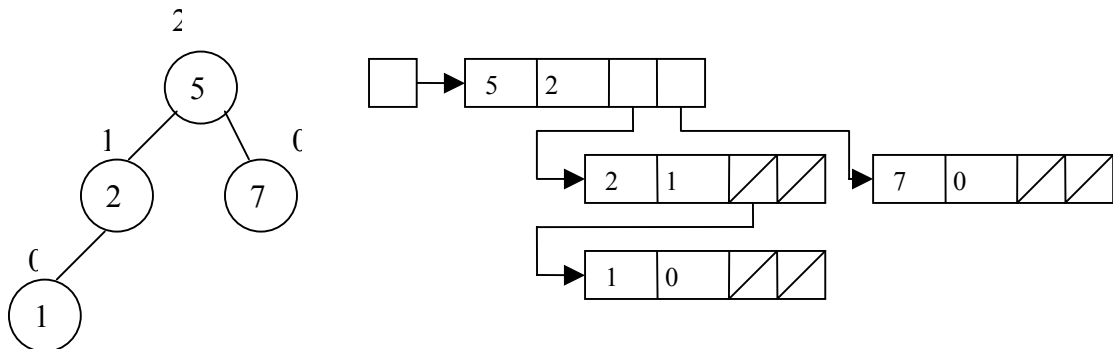
Atliekamas posūkis dešinėn.  
Gautas medis – taisyklingas  
AVL medis. Elementas n Gali  
būti B arba C lapas.



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

Medis	Medžio aukštis	Medžio vaizdavimas su rodykliniais ryšiais
	-1	
	0	
	1	

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



Apsirašome 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

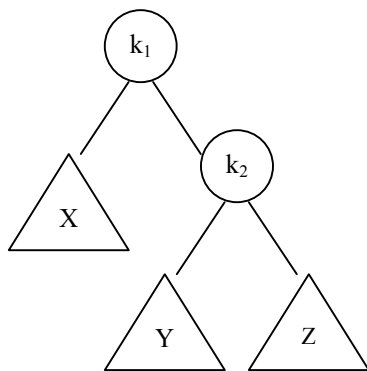
then aukštis := -1

else aukštis := m^.aukštis

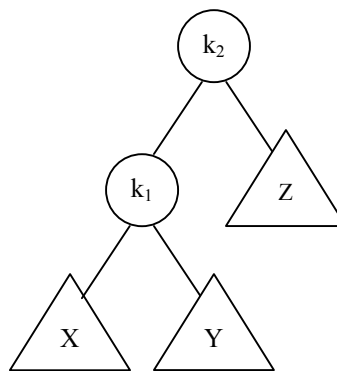
**end;**

**Posūkis kairėn**

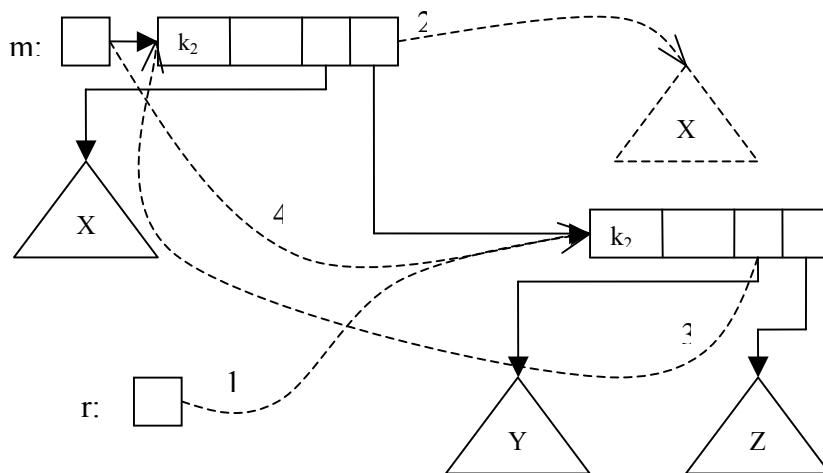
Turime tokį medį:



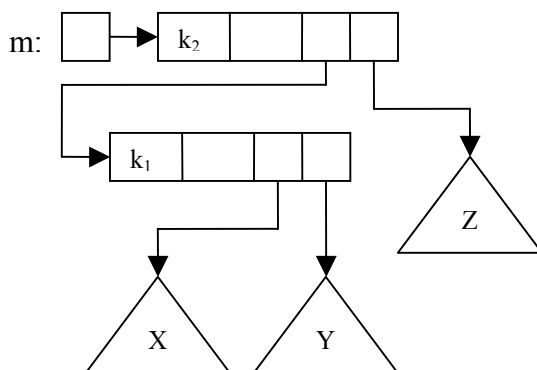
Atlikus transformaciją, gautas toks AVL medis:



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



gautas rezultatas:



Parašysime procedūrą posūkiui kairėn:

```

procedure kairėn (var m : medis);
  var r: medis

```



**begin**

$r := m^{\wedge}.d;$  (1)

$m^{\wedge}.d := r^{\wedge}.k;$  (2)

....

$m := r;$  (3)

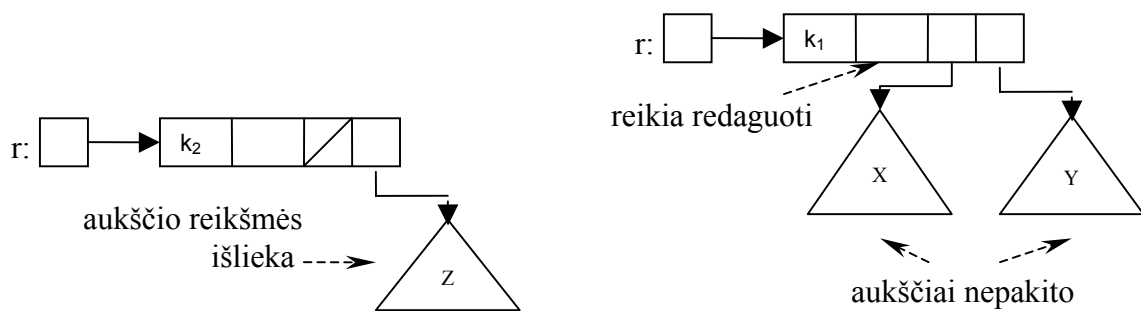
**end;**

*{medžio aukščio sutvarkymas}*

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

$m^{\wedge}.aukštis := \max(\text{aukštis}(m^{\wedge}.k), \text{aukštis}(m^{\wedge}.d)) + 1;$

$r^{\wedge}.aukštis := \max(m^{\wedge}.aukštis, \text{aukštis}(r^{\wedge}.d)) + 1;$



Dabar aprašysime analogišką procedūrą posūkiui dešinèn:

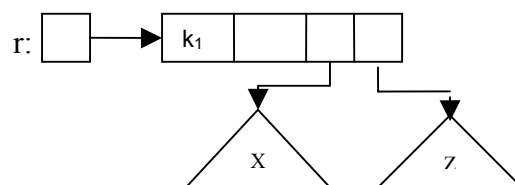


**Procedure** dešinèn (**var** m: medis);

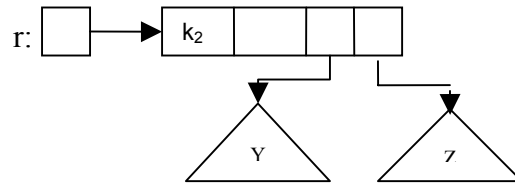
**var** r: medis;

**begin**

$r := m^{\wedge}.k;$



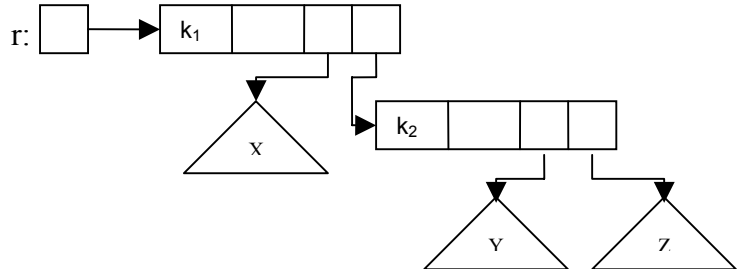
$m^{\wedge}.k := m^{\wedge}.d;$



....

*{medžio aukščio sutvarkymas}*

$m := r;$



**end;**

Sutvarkome medžio aukštį:

$m^{\wedge}.aukštis := \max (\text{aukštis } (m^{\wedge}.k), \text{aukštis } (m^{\wedge}.d)) + 1;$

$r^{\wedge}.aukštis := \max (m^{\wedge}.aukštis, \text{aukštis } (r^{\wedge}.d)) + 1;$

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}*

$m^{\wedge}.info := i;$

$m^{\wedge}.aukštis := 0;$

$m^{\wedge}.k := \text{nil};$

$m^{\wedge}.d := \text{nil};$

**end**

**else**

**end;**

Užbaigiame sąlygos sakinį, pažymėtą :

**if** i <  $m^{\wedge}.inf$

**then begin**

Įterpti\_į\_AVL (i,  $m^{\wedge}.k$ );

**if** aukštis ( $m^{\wedge}.k$ ) – aukštis ( $m^{\wedge}.d$ ) = 2

*{AVL struktūra pažeista}*

**then if** i <  $m^{\wedge}.k^{\wedge}.inf$

**then** dešinèn (m)

**else** dvigubas\_dešinèn (m)

```

    else m^.aukštis := max (aukštis (m^.k), aukštis (m^.d)) + 1;
  end
else begin
  ....
  end;

```

*{analogiški sakiniai dešiniajai šakai}*

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.

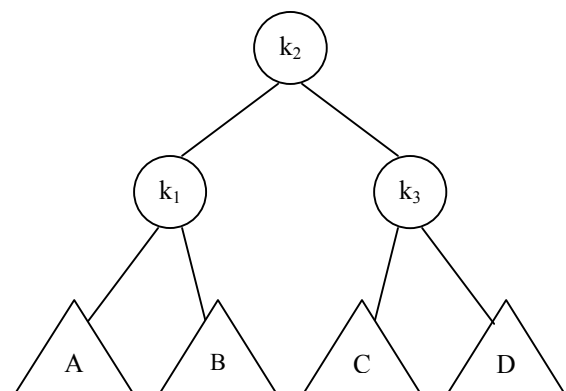
```

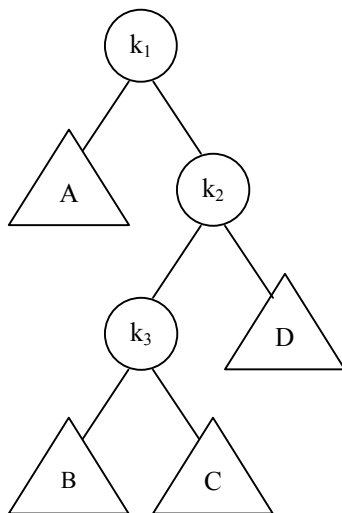
**end.**

**Dvigubas posūkis kairėn.**

Turime tokios struktūros dvejetainį medį:

Atlikus dvigubo posūkio kairėn transformaciją, gautas medis:



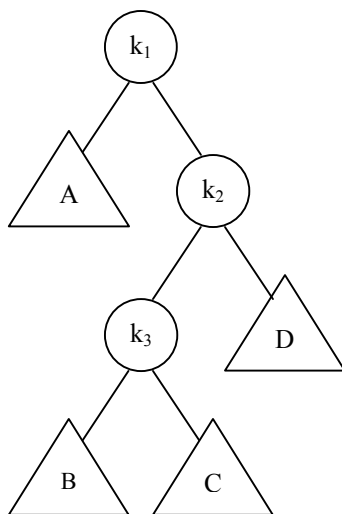


Galima pastebėti, kad dvigubas posūkis kairėn ekvivalentus tokiai transformacijų sekai:

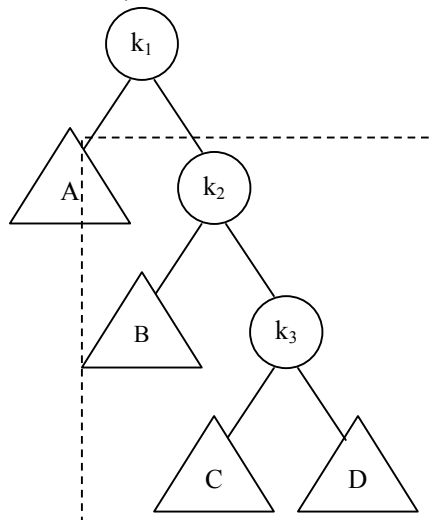
- posūkis dešinėn (viršūnė  $k_3$ )
- posūkis kairėn (viršūnė  $k_1$ )

Įrodome, kad būtent taip ir yra:

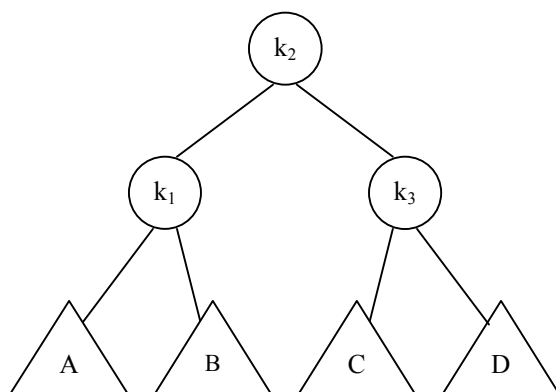
pradinis medis:



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
```

dešinèn (m <sup>^</sup> .d);	<i>{medis sukamas apie viršūnę k<sub>3</sub>}</i>
kairèn (m);	<i>{medis sukamas apie viršūnę k<sub>1</sub>}</i>

**end;**

## **10. REKURSIJOS REALIZACIJA KOMPIUTERYJE**

Rekursija gali būti apibrėžta:

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

2. Funkcijos apraše;
3. Procedūros apraše.

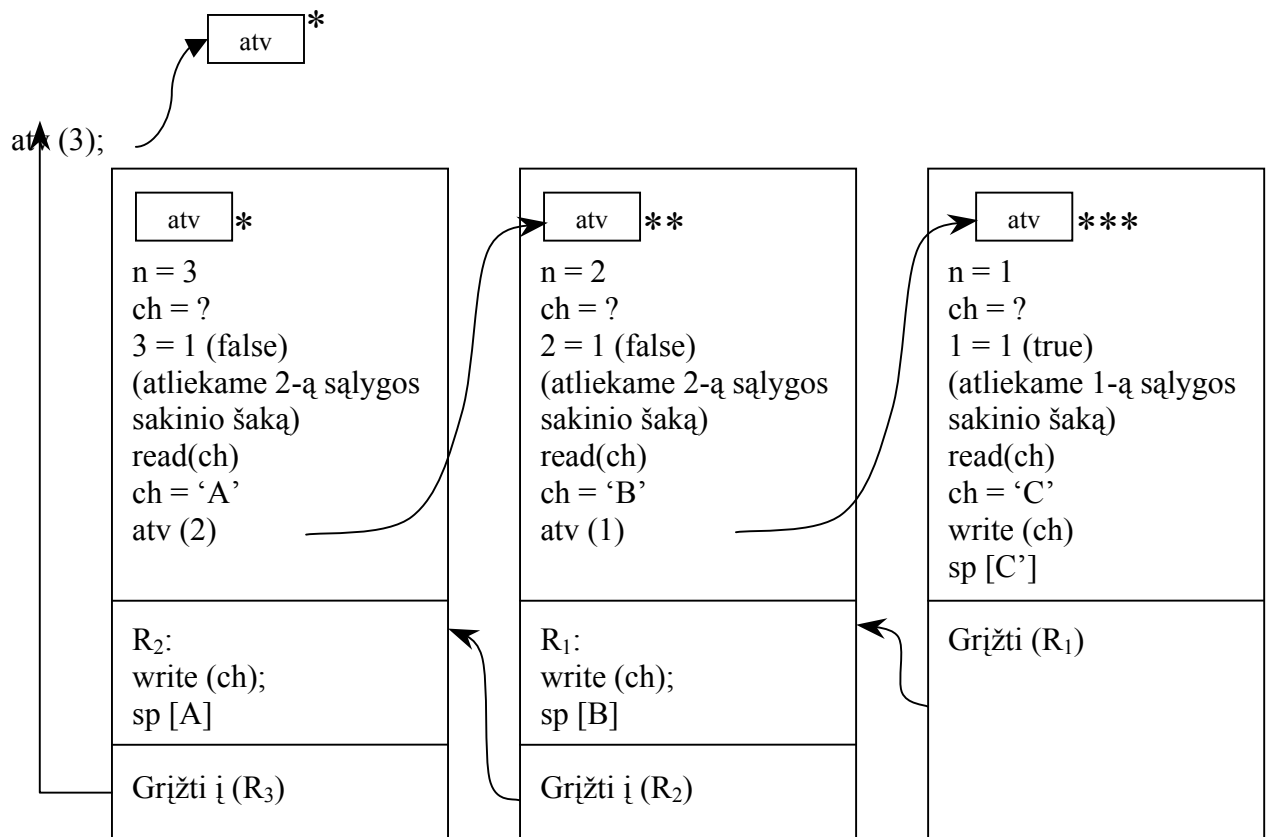
Pavyzdžiai.

Turime rekursinę procedūrą, parašančią raidės atvirkščia tvarka, nei jos yra įvestos:

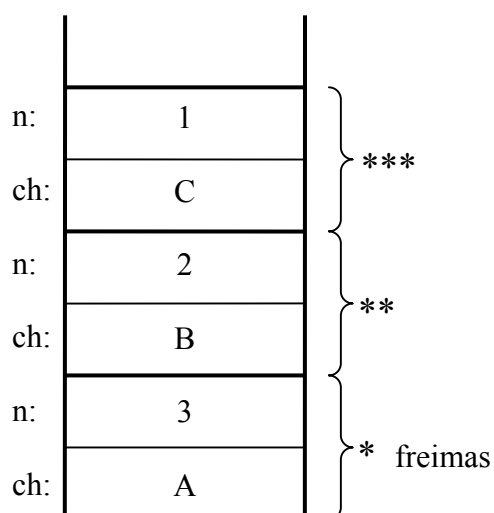
```
procedure atv (n : integer);
var
  ch : char;
begin
  if n = 1
    then begin
      readln (ch);
      write (ch);
    end
    else begin
      readln (ch);
      atv (n - 1);
      write (ch);
    end;
end;
```

procedūros vykdymas:

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

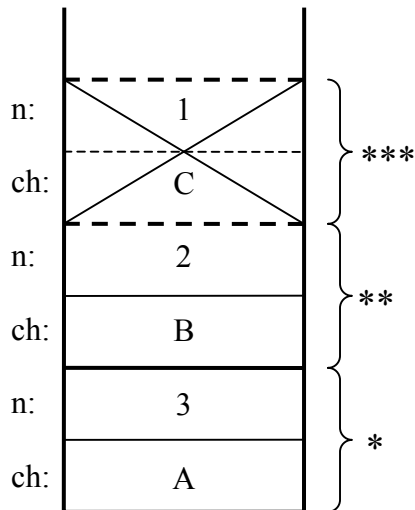


Tokie rekursiniai kreipiniai kompiuteryje naudoja tam tikrą duomenų struktūrą :



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).

## 11. 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 skaičius, 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):

$r = a \bmod b$  (liekana, dalijant vieną skaičių iš kito)  
 jei  $r = 0$ , dbd = b  
 $\text{dbd}(a, b) = \text{dbd}(b, r)$

parašysime rekursinę funkciją dviejų skaičių didžiausiam bendrajam dalikliui rasti:

```
function dbd (a, b : integer): integer;
var
  r : integer;
begin
```

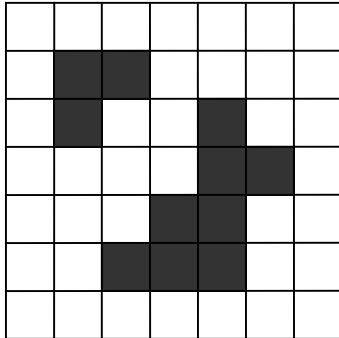


```

r := a mod b;
if r = 0
  then dbd = 0
  else dbd := dbd (b, r);
end;

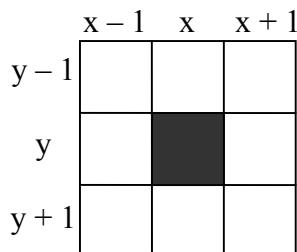
```

#### 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 kuri patenka nurodytas langelis.

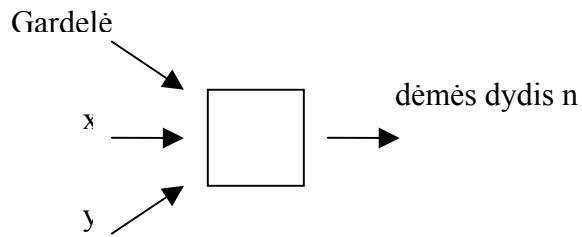
Jei nurodytas langelis patenka į dėmę, tai jis turi daugiausia 8 kaimyninius langelius, kurie gali būti arba užkrėsti, arba ne. Kiekvienas jų turi taip pat daugiausia 8 kaimynus, tie savo ruožtu – taip pat, ir t.t. kaimyninių langelių koordinatės:



bakteriologinė dėmė bus apibrėžiama:

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

Reikia sudaryti funkciją:



Parašome funkciją šiems veiksams atlikti:

**const**

MaxX = 100;

maxY = 100;

**type**

būsena = (tuščia, užpildyta);

gardtipas = array [1 .. MaxX, 1 .. MaxY] of būseną;

**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);

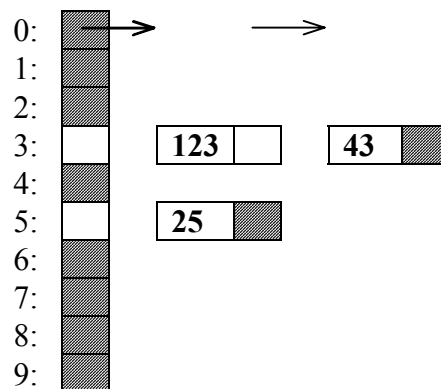
**end;**

**end;**

**begin**

Bdėmė := bd (X,Y);

**end;**



Šiuo atveju naudojame **p** dinaminių sąrašų masyvą. Talpinamo skaičiaus dalybos iš **p** liekana atitinka vieno iš dinaminių sąrašų indeksą, pvz.  $123 \bmod 10 = 3$ . Todėl skaičių 123 priskirsime sąrašui, kurio indeksas 3. Sudarinėjant hash lentelę gradiniui būdu lengvai išvengsime kolizijos, nes tuo atveju kai liekanos sutampa, pakanka į atitinkamą sąrašą įterpti naują elementą.

```

const M = 100;
      M1 = M - 1;
type sąrašas = ^elem;
      elem = record
          raktas : integer;
          info : integer;
          kitas : sąrašas
      end;
      lentelė = array [0..M1] of sąrašas;
procedure init (var A : lentelė);
var i : integer;
begin
for i := 0 to M1 do
    A[i] := nil
end;

procedure įterpti (var A : lentelė;
                  naujas_raktas : integer;
                  naujas_info : integer);
var H : integer;
      p : sąrašas;
begin
    H := hash(naujas_raktas);
    new(p);
    p^.raktas := naujas_raktas;
    p^.info := naujas_info;
    p^.kitas := A[H];
    A[H] := p
end;

procedure paieška (var A : lentelė;
                   ieškomas_raktas : integer;
                   var rasta : boolean;
                   var rasta_info : integer);
var H : integer;
      p : sąrašas;
begin
    H := hash(ieškomas_raktas);
    p := A[H];
    rasta := false;
    while (p <> nil) and not rasta do
        if p^.raktas = ieškomas_raktas then rasta := true
            else p := p^.kitas;
        if rasta then rasta_info := p^.info
    end;

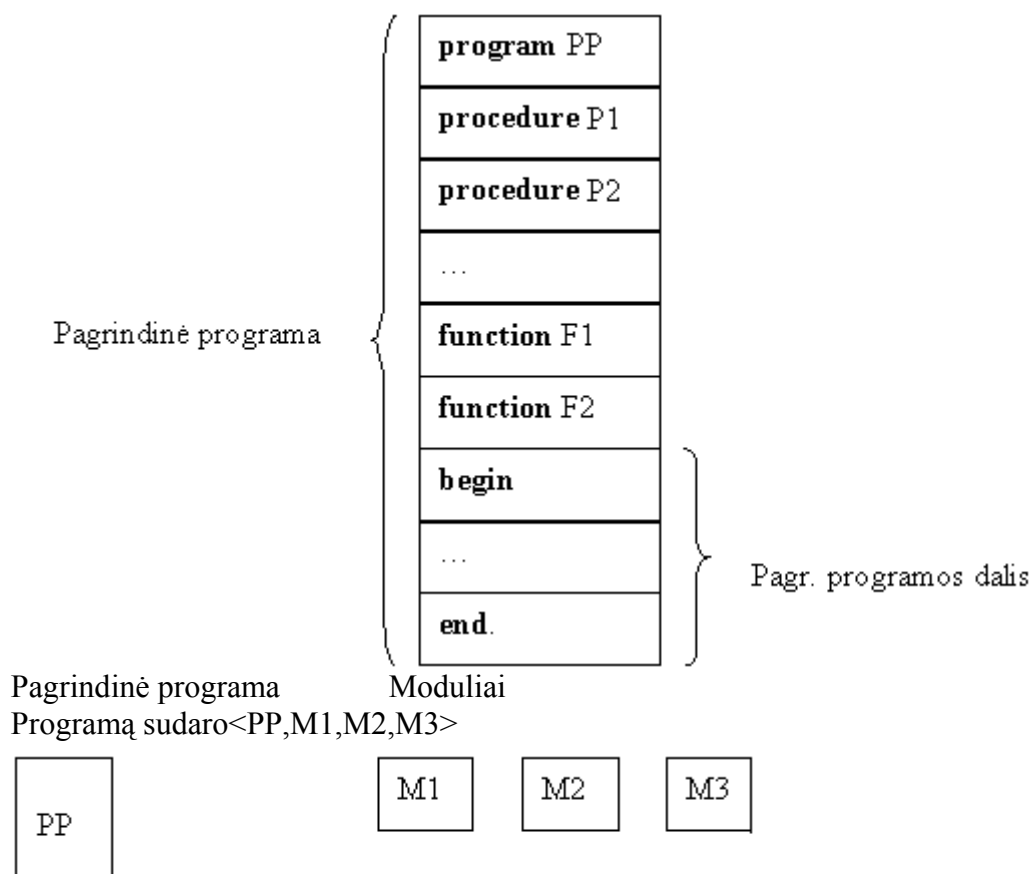
```

Tiesinio ir grandininio hash lentelės sudarymo būdų palyginimas:

Būdas	Privalumai	Trūkumai
Tiesinis	Greitesnis	Elementų kiekis ribotas: $n \leq M$
Grandininis	Neribojamas elementų kiekis n	Lėtesnis

## 12. MODULIAI

Programos struktūrizacija



Modulį sudaro

- Apibrėžimo dalis (**interface**)
- Veiksmų dalis, dar vadinama realizacijos dalimi (**implementation**)

Apibrėžimo dalyje nurodyti resursai, kuriuos teikia modulis, pvz., procedūros, funkcijos, konstantos, tipai ir t.t.

Veiksmų dalyje nurodoma resursų realizacija.

Modulio pavyzdys. Veiksmai su kompleksiniais skaičiais

**unit** komplmod;

```

interface
  type kompl=record
    re,m:real
  end;
  procedure sudėtis(x,y:kompl; var z:kompl);
  procedure atimtis(x,y:kompl; var z:kompl);
  procedure daugyba(x,y:kompl; var z:kompl);
implementation
  procedure sudėtis(x,y:kompl; var z:kompl);
    begin
      z.re:=x.re+y.re;
      z.m:=x.m+y.m
    end;
  procedure atimtis;           {parametrų sąrašas praleistas, jei jis kartojamas}
    begin                     {turi atitikti esantį apibrėžimų dalyje}
      z.re:=x.re-y.re
      z.m:=x.m-y.m
    end;
  procedure daugyba;
    begin
      z.re:=x.re*y.re-x.m*y.m;
      z.m:=x.re*y.m+x.m*y.re
    end;
end.

program panaudojimas
  uses komplmod;      nurodoma, kad bus naudojamas sukurtas modulis 'komplmod'
  var a,b,c:kompl;
begin
  a.re:=2.5;
  a.m:=1.0;
  b.re:=1.3;
  b.m:=2.5;
  sudėtis(a,b,c);
  writeln(c.re,' ',c.m)
end.

```

Programa, naudojanti modulius vykdoma tokiu būdu:

1. Modulį įrašome į failą modulio vardu, t. y., šiuo atveju, vardu "komplmod.pas"
2. Sukurtas failas kompiliuojamas atskirai režimu "Destination Disk"  
Kompiliavimo rezultatas: "komplmod.tpu";
3. Pagrindinė programa kompiliuojama įprasta tvarka  
(gaunama programa sujungta su moduliu "komplmod.tpu").
4. Vykdoma gauta programa.

Pačiame modulyje gali būti naudojami kiti moduliai:

```

unit M1
uses M11, M12;

```

```

...
unit M2
  uses M21, M22;
...
program PPP;
  uses M1,M2;
...
Šiuo atveju programą sudaro <PPP,M1,M2,M11,M12,M21,M22>

```

### **Vardų kolizija**

Pagrindinėje programoje galima naudoti tuos pačius vardus kaip ir modulyje, tačiau prieš naudojant pastarojo resursus reikės nurodyti modulio vardą.  
Pvz.

```

program vardai;
  uses komplmod;
  var a,b,c:kompl;
    u,v,w:integer;
  procedure sudėtis(a,b:integer;var c:integer);
    begin
      ...
    end;
  begin
    komplmod.sudėtis(a,b,c); {bus naudojama modulio procedūra}
    sudėtis(u,v,w)           {programoje apibrėžta procedūra}
  end.

```

### **Sisteminiai moduliai**

**System**—realizuotos standartinės funkcijos ,procedūros.Jis ypatingas tuo, kad jo nereikia įtraukti sakiniu **uses**.

**Crt**—displėjaus,klaviatūros,spalvų, garsų galimybės.

**Dos**—operacinės sistemos galimybės.

**Graph**—realizuoja grafiką.

**Overlay**—naudojamas, kai programos vykdymui, reikia daugiau atminties, nei yra operatyviosios.Tai reiškia programa suskaidoma dalimis ir kiekviena dalis vykdoma atskirai.

**Strings**—naudojamas eilučių apdorojimui.

Sisteminiai moduliai saugojami faile “turbo.tpl”, norint jais pasinaudoti, jie nurodomi sakiniu **uses**(išskyrus **System** modulį).

## **13. DINAMINIŲ MASYVŲ MODELIAVIMAS**

Turbo paskalio masyvi statiniai, tai reiškia, kad jų dydis turi būti žinomas kompiliavimo metu. Pvz. toks sakiny s paskalyje neleidžiamas:  
readln(n);

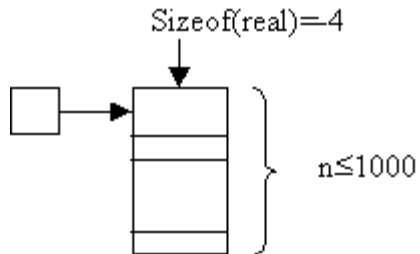
**var** a:**array** [1..n] **of** real.

Dinaminis masyvas kuriamas naudojant rodyklės kintamuosius ir procedūrą GetMem(r,n), r-rodyklės tipio kintamasis, n-sveikas sk.  
Pvz.

```

type mas=array[1..1000] of real;
var a:^mas;
...
readln (n);
GetMem(a,n*sizeof(real));    {sukuriama rudyklė, kuri rodo į n-atį real tipo masyvą}
for i=1 to n do
    readln (a^[i]);

```



### Sukūrimas

Atminties atlaisvinimas

```

1. new(a);
2. GetMem (a,1000*sizeof(real));
3. GetMem (a,n*sizeof(real));
dispose (a);
FreeMem (a,1000*sizeof(real));
FreeMem (a,n*sizeof(real))

```

### Literatūra

1. Tumasonis V. Paskalis ir Turbo Paskalis 7.0. – V.:Ūkas, 1993.
2. McCracken D.D. A second course in computer science with Pascal. 1987.
3. Weiss M.A. Data Structures and Algorithm Analysis. 1992.