

**Vilniaus Universitetas**

Algimantas Juozapavičius

**DUOMENŲ STRUKTŪROS IR  
ALGORITMAI**

mokomoji priemonė

**Vilniaus universiteto leidykla**

**1997**

Mokomoji priemonė iš dalies apima dėstomas kursų “Duomenų struktūros”, “Informacinės sistemos”, “Efektyvūs algoritmai ir duomenų struktūrų sintezė” temas. Šie kursai numatyti Matematikos fakulteto informatikos specialybės studentų bakalauro ir magistro studijų programose.

Apsvarstė ir rekomendavo spausdinti Matematikos fakulteto taryba (1996.09.17,-protokolo Nr. 8).

Recenzavo dėst. V. Dičiūnas

Mokomąją priemonę išleisti padėjo TEMPUS S\_JEP-09752-95 programa.

© Algimantas Juozapavičius, 1997

## Turinys.

### Įvadas.

1. Pagrindiniai duomenų struktūrų objektai
  - 1.1. Masyvas (*array*)
  - 1.2. Stekas (*stack*)
  - 1.3. Eilės (*queue*)
  - 1.4. Dekas (*dequeue*)
  - 1.5. Tiesiniai sąrašai (*linked lists*)
  - 1.6. Medžiai (*trees*)
  - 1.7. Medžių dėstymas kompiuterių atmintyje
  - 1.8. Rekursija
  - 1.9. Algoritmų analizė
  - 1.10. Algoritmų diegimas
2. Rūšiavimo metodai
  - 2.1. Elementarūs rūšiavimo algoritmai
    - 2.1.1. Išrinkimo (*selection*) algoritmas
    - 2.1.2. Įterpimo algoritmas
    - 2.1.3. Burbulo algoritmas
    - 2.1.4. Kevalo (*shellsort*) algoritmas
    - 2.1.5. Pasiskirstymo skaičiavimas
  - 2.2. Greito rūšiavimo algoritmas
    - 2.2.1. Greito rūšiavimo algoritmo taikymas elementams išrinkti
    - 2.2.2. Greito rūšiavimo algoritmo sudėtingumas
  - 2.3. Skaitmeninis rūšiavimas (*radix sort*)
  - 2.4. Prioritetinės eilės (*priority queues*)
    - 2.4.1. Duomenų struktūra *heap*
    - 2.4.2. Operacijos su *heap* struktūra
    - 2.4.3. *Heapsort* algoritmas
    - 2.4.4. Netiesioginė *heap* duomenų struktūra
    - 2.4.5. Aibės duomenų struktūra
    - 2.4.6. Aibinis *heapsort* algoritmas
  - 2.5. Sąlajos rūšiavimas (*mergesort*)
  - 2.6. Išorinis rūšiavimas (*external sorting*)
    - 2.6.1. Subalansuota daugybinė sąlaja (*balanced multiway merging*)
    - 2.6.2. Pakeitimo išrinkimas (*replacement selection*)
3. Paieškos metodai
  - 3.1. Elementarūs paieškos metodai
    - 3.1.1. Nuosekli paieška
    - 3.1.2. Dvejetainė paieška
    - 3.1.3. Dvejetainio medžio paieška
    - 3.1.4. Operacija *delete*
    - 3.1.5. Netiesioginiai dvejetainės paieškos medžiai
  - 3.2. Subalansuoti medžiai
    - 3.2.1. 2-3-4 ir 2-3-medžiai
    - 3.2.2. Įterpimo operacija 2-3-medžiuose
    - 3.2.3. Išmetimo operacija 2-3-medžiuose
    - 3.2.4. Duomenų struktūros 2-3-medžiams

- 3.2.5. Raudoni-juodi medžiai
- 3.3. Dėstymo lentelės
  - 3.3.1. Išdėstymo funkcijos
  - 3.3.2. Atskirti sąrašai
  - 3.3.3. Atviro adresavimo dėstymo metodai
- 3.4. Skaitmeninė paieška
  - 3.4.1. Skaitmeniniai paieškos medžiai
  - 3.4.2. Skaitmeniniai paieškos indeksai
  - 3.4.3. Daugybiniai skaitmeniniai paieškos indeksai
  - 3.4.4. *Patricia* algoritmas
- 3.5. Išorinė paieška
  - 3.5.1. Indeksinė-nuosekli paieška
  - 3.5.2. B-medžiai
  - 3.5.3. Išplėstinis dėstymas
- 4. Sekų apdorojimas ir kompresija
  - 4.1. Sekų paieška
  - 4.2. Tiesmukiškas (brutalios jėgos) algoritmas
  - 4.3. Knuth-Morris-Pratt'o algoritmas
  - 4.4. Boyer-Moore'o algoritmas
  - 4.5. Rabin-Karp'o algoritmas
  - 4.6. Dviejų sekų bendras ilgiausias posekis
  - 4.7. Sekų ir failų kompresijos metodai
    - 4.7.1. Vienodų simbolių sekų kodavimas
    - 4.7.2. Kintamo ilgio kodavimas
- 5. Daugiamačių duomenų struktūros ir algoritmai.
  - 5.1. Vidinės atminties struktūros
    - 5.1.1. K-D-medžiai
    - 5.1.2. BSP-medžiai
    - 5.1.3. Ketvirtiniai medžiai
  - 5.2. Antrinės atminties struktūros
    - 5.2.1. Daugiamačiai išplėstinio dėstymo metodai
    - 5.2.2. Daugiamačiai hierarchiniai metodai
    - 5.2.3. Erdviniai kreipties metodai
- 6. Uždaviniai ir pratimai
  - 6.1. Pirmoji uždavinių grupė
  - 6.2. Antroji uždavinių grupė
  - 6.3. Trečioji uždavinių grupė
  - 6.4. Pratimai tiriamajam darbui
- 7. Literatūra

## Ivadas

Mokomosios priemonės tikslas - supažindinti su duomenų struktūrų ir algoritmų teorija ir praktika.

Duomenų struktūros ir algoritmai - tai sritis, formalizuojanti platų spektrą svarbių ir vis labiau plintančių procedūrų, kompiuteriu sprendžiant simbolių skaičiavimų, inžinerijos, kompiuterinės grafikos, duomenų bazių, telekomunikacijų ir kitų sričių uždavinius. Ji jungia įvairius matematinės logikos, kombinatorikos, diskrečiosios matematikos, programavimo, kompiuterinės technikos konstravimo, geometrijos ir algebros metodus. Sukaupę rezultatų gausa ir kokybė jau apie 20 metų išskiria ją kaip atskirą mokslo tyrimų discipliną ir trumpai apibrėžia kaip duomenų dėstymo kompiuterio atmintyje metodų visumą.

Į algoritmus žmonija atkreipė dėmesį jau labai seniai. Dažniausiai yra pateikiamas **EUKLIDO** algoritmas dviejų sveikų skaičių didžiausiam bendram dalikliui skaičiuoti (*greatest common divisor - gcd*), kaip tematikos senumo ir fundamentalumo pavyzdys:

```
program euclid (input, output);
var x,y: integer;
function gcd (u,v: integer): integer;
    var t: integer;
    begin
    repeat
        if u<v then
            begin t := u; u := v; v := t end;
        u := u-v;
    until u = 0;
    gcd := v
    end;
begin
    while not eof do
    begin
        readln (x, y);
        if (x>0) and (y>0) then writeln (x, y, gcd (x, y))
    end;
end.
```

Taikymuose algoritmai remiasi sudėtingais duomenų organizavimo ir manipuliavimo metodais. Šie metodai kartu su duomenų aibėmis vadinami duomenų struktūromis ir yra vieni iš pagrindinių informatikos objektų. Jie neatskiriami nuo algoritmų (kaip veiksmų). Tačiau ryšis tarp jų nevienareikšmis. Kartais paprasti algoritmai gali naudoti sudėtingas duomenų struktūras, o sudėtingi - paprastas. Vienas iš pagrindinių ypatumų, apibrėžiančių algoritmų veiksmingumą, yra juose naudojamos duomenų struktūros. Šioje mokomojoje priemonėje pagrindinis dėmesys skiriamas duomenų struktūroms (kaip veiksmų sudėtinei daliai) ir nagrinėjami su tuo susiję algoritmai.

Algoritmai ir duomenų struktūros yra neatsiejami nuo taikymų. Duomenys taikymuose įgyja turinį ir formą (t.y. sintaksę ir semantiką). Tai daro įtaką operacijoms su duomenimis ir kaip pasekmei - jų kompiuterinei realizacijai. Tas pats algoritmas skirtingiems duomenims gali būti visiškai skirtingai realizuotas. Kokia realizacija kokiems duomenims yra efektyvi - tai irgi vienas iš pagrindinių duomenų struktūrų tyrimo objektų. Papildomas darbo su duomenų struktūromis aspektas, galima sakyti "visuomeninis" - nors ir paprastas algoritmas, bet turintis sudėtingą programinę realizaciją, gali būti daugelio klaidų šaltinis. Todėl reikia nagrinėti ir patikimus duomenų struktūrų programavimo būdus. Visi šie aspektai tik pabrėžia duomenų struktūrų teorijos ir praktikos svarbą.

**Apibrėžimas.** Duomenų struktūra (DS) - tai duomenų aibė, kuriai apibrėžtos tam tikros operacijos ir loginiai ryšiai tarp jų. Literatūroje naudojamas ir kitas pavadinimas (dažnai suprantamas kaip sinonimas) - abstraktūs duomenų tipai (ADT). Kadangi apibrėžime dalyvauja operacijos sąvoka, duomenų struktūros neatskiriamos nuo algoritmų. Šis apibrėžimas labiau panašus į matematinį, nes yra abstraktus. Kompiuterių teorijoje ir praktikoje duomenų struktūrų nagrinėjimas neatskiriamas nuo jų diegimo (realizavimo). Formaliai vienodai apibrėžtos duomenų struktūros diegimas kompiuterio atmintyje gali būti toks skirtingas, kad ši struktūra bus minima įvairiais pavadinimais. Realizacijos skirtumai kartais labai pakeičia tam tikro algoritmo sandarą. Tačiau duomenų struktūros esmė informatikos požiūriu yra tokia:

- **apibrėžti vietas išskyrimo duomenims talpinti kompiuterio atmintyje metodą;**
- **apibrėžti duomenų reikšmių rašymo į šią vietą metodą.**

Literatūroje yra nusistovėjęs tam tikras terminų "abstraktūs duomenų tipai" ir "duomenų struktūros" vartojimo santykis. Kalbant apie abstraktų duomenų tipą, paprastai atsiribojama nuo realizacijos, jos ypatumų, pabrėžiami tik duomenys (tam tikra tvarka dėstomi), ir algoritmai (tam tikra tvarka atliekamos operacijos). Pagrindinė ADT sąvokos įvedimo priežastis - tai noras abstrahuotis nuo konkrečios realizacijos. Šis noras ypač būna svarbus projektuojant dideles ir sudėtingas programas. Tuo tarpu terminas "duomenų struktūros" labiau vartojamas tada, kai norima paminėti ir konkretų algoritmo realizavimą. Be to, dažnai algoritmą formalizuojant reikia nurodyti esmines detales, kurios išskyla realizacijos metu, ir jas įtraukti į algoritmą. Abiejų terminų vartojimo atvejais yra ir tam tikrų kolizijų, kurias ne visada skiria netgi programuotojai. Taigi stekai ir eilės yra abstraktūs duomenų tipai, kai kalbama apie manipuliacijas su elementais, įeinančiais į šią duomenų aibę ir turinčiais gan sudėtingą vidinę struktūrą. Tačiau tai yra duomenų struktūros, kai stekai ir eilės realizuoti sveikiems dvejetainiams skaičiams dauginti ar tvarkyti atmintį, esant procesoriaus darbo pertraukims. Duomenų struktūros masyvai ar jungtiniai sąrašai yra abstraktaus duomenų tipo - tiesinio sąrašo - detalizacijos Paskalio (ar panašioje) programavimo kalboje. Šios detalizacijos priklauso nuo algoritmo ir gali būti esminės.

ADT vartojimo ar apibrėžimo atvejai kartais gali būti labai įvairūs. Daug algoritmų yra formuluojami remiantis tiesiniais sąrašais, pavyzdžiui, rūšiavimo ar paieškos algoritmai. Tačiau jiems efektyviai realizuoti reikia dažniausiai sudėtingų duomenų struktūrų.

Tarp įvairių abstrakčių duomenų tipų ar duomenų struktūrų yra susiklostę įvairūs, kartais gan sudėtingi santykiai. Vieną ADT (ar DS) galime apibrėžti kita. Stekai ir eilės gali būti apibrėžiami kaip tiesiniai sąrašai, o Paskalio kalbos rodyklės ir įrašo tipai gali būti sėkmingai naudojami tiesiniams sąrašams apibrėžti. ADT sąvoka svarbi naudojant skirtingus abstrakčių lygius, konstruojant dideles sistemas.

Knygos turinį sudaro kelios grupės temų:

- pagrindiniai objektai ir metodai (baziniai objektai, elementarios duomenų struktūros, medžiai, rekursija, algoritmų analizės metodai, algoritmų realizavimas);
- rūšiavimo metodai (elementarūs rūšiavimo metodai, greitas rūšiavimas, skaitmeninis rūšiavimas, prioritetinės eilės, rūšiavimo-sąlajos algoritmai, išorinis rūšiavimas);
- paieškos metodai (elementarūs paieškos metodai, subalansuoti medžiai, dėstymo lentelės, skaitmeninė paieška, išorinė paieška);
- sekų apdorojimas (sekų fragmentų paieška, fragmentų tapatinimas, bendrųjų posekių paieška ir tapatinimas, failų kompresija);
- vaizdų apdorojimo ir kompiuterinės grafikos algoritmai, manipuliacijų su duomenimis labai didelėse duomenų bazėse algoritmai.

Mokomosios priemonės turinį (tačiau ne visas temas) labiausiai atitinka [1] vadovėlis. Jame aprašyti algoritmai yra pritaikyti programavimo Paskaliu metodikai, panašiai kaip ir šioje knygoje,

todėl iš jos perimta ir nemažas kiekis programų tekstų pavyzdžių. Programavimo kalbos C++ metodikai pritaikyti pagrindiniai algoritmai yra išdėstyti [2] knygoje. Ši knyga irgi yra labai geras duomenų struktūrų ir algoritmų teorijos vadovėlis. Knyga [3] yra tikra algoritmų, dėstomų studentams, enciklopedija. Joje dėstomi algoritmai ir duomenų struktūros yra formalizuoti, pateikiama išsami algoritmų efektyvumo analizė. Kitos - [5 - 10, 12, 13] knygos yra VU Matematikos fakulteto skaitykloje ir gali būti naudojamos vienai ar kitai temai nagrinėti.

Mokomojoje priemonėje pateikiami uždaviniai skirstomi į keturias grupes. Pirmosios grupės uždavinių tikslas - įgyti pradinių algoritmų analizės įgūdžių. Tai nesudėtingi uždaviniai, kuriems reikia parinkti ekonomiškiausią algoritmą ir šį algoritmą įvertinti. Antrosios grupės uždaviniai skirti tiesioginiam duomenų struktūrų realizavimui programavimo kalba. Trečios grupės uždaviniai - tai savarankiškos problemos, kurias galima efektyviai spręsti, parenkant tinkamas duomenų struktūras. Kompleksinei duomenų struktūrų ir algoritmų analizei skirti ketvirtos grupės uždaviniai.

# 1. Pagrindiniai duomenų struktūrų objektai

## Pagrindiniai duomenų struktūrų objektai:

- duomenų ir algoritmų aprašymo (ir užrašymo) kalba;
- baziniai duomenų tipai ir elementarios duomenų struktūros;
- duomenų struktūrų ir atitinkamų algoritmų analizės ir diegimo metodai.

Algoritmų ir duomenų struktūrų apibrėžimui ir užrašymui naudojama kalba neturi žymiai skirtis nuo realizavimo kalbos, ir, be to, turi būti suderinama su duomenų ir algoritmų analizės metodais. Esant dabartinei situacijai informatikoje, geriausia tam tikslui naudoti daugiau ar mažiau formalizuotą programavimo kalbą. Šioje knygoje pasirinkome Paskalio programavimo kalbą (atitinkančią standartą, išdėstytą [4]). Pagrindinės priežastys - ji dėstoma pirmame kurse ir todėl gerai pažįstama studentams, be to, yra pakankamai struktūrizuota ir patogi algoritmams užrašyti bei duomenų struktūroms apibrėžti. Jos pagrindinės operacijos suderinamos su algoritmų analizėje naudojamais metodais. Ją vartojant realizuota daug populiarių algoritmų. Paskalio neigiama ypatybė - daugeliui taikymų ši kalba nėra pakankamai detali, ir automatiškai neįgalina išreikšti algoritmų su visomis detalėmis. Todėl dabartiniu metu literatūroje algoritmai vis dažniau aprašomi C arba C++ programavimo kalba.

**Baziniai duomenų tipai** - tai duomenų struktūros, įdiegtos programavimo kalboje. Paskalyje kalboje tai yra: sveikas (*integer*), realus (*real*), loginis (*boolean*), simbolinis (*character*), masyvas (*array*), aibė (*set*), rodyklės tipas (*pointer*), įrašo tipas (*record*). Užrašydami visas kitas duomenų struktūras ar algoritmus konkrečioje realizacijoje (programoje), turime jas išreikšti baziniais tipais ir tam tikromis operacijomis su jais.

**Elementarios duomenų struktūros** - struktūros, labai paplitę ir nesudėtingai apibrėžiamos. Jos gali būti realizuotos įvairiais lygiais: techniniu lygiu (operacijų vykdymas įdiegtas elektroninėje schemoje ar kitoje techninėje kompiuterio dalyje), programavimo kalba (realizuotos kalboje baziniais tipais), algoritmu (duomenų struktūros operacijos išreiškiamos algoritmiškai baziniais tipais). Toliau dėstomos pagrindinės elementarios duomenų struktūros.

## 1.1 Masyvas (*array*)

Masyvas - tai vieno tipo tolydžiai išdėstytų kintamųjų aibė, kai kintamųjų skaičius iš anksto žinomas, o kiekvieną kintamąjį galima išrinkti tiesiogiai (naudojant indeksą ar indeksus). Šitoks apibrėžimas pateikia masyvą kaip bazinį tipą (pvz., Paskalio kalboje). Duomenų struktūros atveju masyvas apibrėžiamas kaip fiksuotas kiekis duomenų, kurie atmintyje laikomi nuosekliai, o išrenkami naudojant indeksą ar indeksus, be to, kiekvieno duomens reikšmės formuojamos unifikuotai. Masyvai panaudojami beveik kiekvienoje programoje. Čia pateikiamas pavyzdys, kaip masyvas naudojamas programuojant **Eratosteno rėtį** - metodą pirminiams skaičiams rasti, pasiūlytą trečiame šimtmetyje B.C.:

```

program primes (input, output);
const N=1000;
var a: array[1..N] of boolean; i, j: integer;
begin
  a[1]:= false; for i:= 2 to N do a[i]:= true;
  for i:= 2 to N div 2 do
    for j:= 2 to N div i do a[i*j]:= false;
  for i:= 1 to N do
    if a[i] then write (i:4);
end.

```



Masyvo naudojimas algoritme įgalina:

- efektyviai taikyti tiesioginį duomenų išrinkimą;
- suvienodinti kiekvieno duomens išrinkimo laiką.

Masyvas yra **statinė duomenų struktūra** - jo dydis turi būti žinomas iš anksto. Masyvo paplitimą sąlygoja ir tai, kad faktiškai bet kurio kompiuterio atmintį galima interpretuoti kaip masyvą - indeksų vaidmenį vaidina adresai. Duomenims išrinkti masyve gali būti naudojama ir daugiau kaip vienas indeksas - tada kalbama apie dvimačius, trimačius ir t. t. masyvus. Išrinkimo laikas masyve bet kuriam duomeniui yra fiksuotas. Jis priklauso tik nuo masyvo matavimų skaičiaus, t.y. nuo formulės, kuria naudojantis pagal kelių indeksų reikšmes skaičiuojamas duomens saugojimo atmintyje adresas.

## 1.2 Stekas (stack)

Duomenų struktūra stekas - tai duomenų aibė, kuriai apibrėžtos tokios operacijos:

- inicializuoti steką (išskirti vietą stekui kompiuterio atmintyje);
- įterpti elementą  $x$  į steką (operacija *push(x)*);
- pašalinti elementą iš steko (operacija *pop*);
- skaityti steką;
- panaikinti steką (panaikinti vietą stekui kompiuterio atmintyje).

Duomenims steke taikomi loginiai apribojimai - operacijos *pop* metu iš steko bus pašalintas elementas, įterptas paskutiniu (taip vadinamas *LIFO (Last-In-First-Out)* principas). Todėl stekas yra viena iš riboto išrinkimo klasės struktūrų.

Lietuviškai steką siūloma vadinti dėklu, tačiau pastarasis terminas per daug netiksliai atskleidžia steko esmę.

Steko struktūra naudojama įvairiausiose srityse: kompiuterio aparatūroje, programose, operacinių sistemų architektūroje, kompiliatoriuose, loginių ir simbolių skaičiavimų algoritmuose, kitur.

Labai paplitęs pavyzdys, iliustruojantis steko privalumus, yra algebrinių reiškinių reikšmių skaičiavimas arba šių reiškinių teisingo užrašymo tikrinimas, pvz., ar teisingai išdėstyti skliausteliai reiškinyje. Skaičiuojant aritmetinio reiškinio

$$9 * (((5 + 8) + (8 * 7)) + 3)$$

reikšmę, steko operacijų seka gali būti tokia:

```
push(9);
push(5);
push(8);
push(pop+pop);
push(8);
push(7);
push(pop*pop);
push(pop+pop);
push(3);
push(pop+pop);
push(pop*pop);
writeln(pop).
```

Steko turinio kitimą atspindi 1 pav.



						7							
		8		8		8	56		3				
	5	5	13	13		13	13	69	69	72			
9	9	9	9	9		9	9	9	9	9	648	nil	

1 pav. Steko turinio kitimas, skaičiuojant reiškini  $9*((5+8)+(8*7))+3$ .

Programuojant steko operacijas Paskaliu, reikia naudoti koki nors bazini - rodyklės arba masyvo tipa. Steko operacijas, išreikštos baziniu rodyklės tipu:

```

type link=^node;
      node=record key:integer; next:link; end;
      var head,z:link;
procedure stackinit;
begin
      new(head); new(z);
      head^.next:=z; z^.next:=z
end;
procedure push(v:integer);
      var t:link;
begin
      new(t);
      t^.key:=v; t^.next:=head^.next;
      head^.next:=t
end;
function pop:integer;
      var t:link;
begin
      t:=head^.next;
      pop:=t^.key;
      head^.next:=t^.next;
      dispose(t)
end;
function stackempty:boolean;
begin stackempty:= (head^.next=z) end.

```

Steko operacijos, programuojamos masyvo baziniu tipu:

```

const maxP=100;
    var stack: array[0..maxP] of integer; p:integer;
procedure push(v:integer);
    begin stack[p]:=v; p:=p+1 end;
function pop:integer;
    begin p:=p-1; pop:=stack[p] end;
procedure stackinit;
    begin p:=0 end;
function stackempty:boolean;
    begin stackempty:=(p=<0) end.

```

Algebrinio reiškinių reikšmės skaičiavimas naudojant steką:

**stackinit;**

```

repeat
  repeat read(c) until c<>";
  if c=')' then write(chr(pop));
  if c='+' then push(ord(c));
  if c='*' then push(ord(c));
  while (c=>'0') and (c=<'9') do
    begin write(c); read(c) end;
  if c<>'(' then write("");
until eoln;

```

### 1.3 Eilės (*queues*)

Duomenų struktūra eilė - tai duomenų aibė, kuriai apibrėžtos tokios operacijos:

- inicializuoti eilę (išskirti vietą eilei kompiuterio atmintyje);
- įterpti tam tikrą elementą  $x$  į eilę (operacija *put*( $x$ ));
- pašalinti elementą iš eilės (operacija *get*);
- skaityti eilę;
- panaikinti eilę (panaikinti vietą eilei kompiuterio atmintyje).

Duomenims eilėje taikomi loginiai apribojimai - operacijos *get* metu iš eilės bus pašalintas elementas, įterptas pirmasis (vadinamasis *FIFO (First-In-First-Out)* principas). Todėl eilė, kaip ir stekas, yra viena iš riboto išrinkimo klasės struktūrų.

Eilės struktūra taikoma įvairiose srityse: algoritams realizuoti, programų loginėms schemoms, operacinių ir taikomųjų sistemų architektūroje, kompiliatoriams, loginių ir simbolių skaičiavimų algoritms, matematinio modeliavimo uždaviniams ir pan.

Programuojant eilės (kaip ir stekui) operacijas Paskaliu, reikia naudoti kokią nors bazinį - rodyklės arba masyvo tipą. Eilės operacijos, išreikštos baziniu rodyklės tipu:

```

type link=^node;
  node=record key:integer; next:link; end;
var head,tail,z:link;
procedure queueinit;
begin
  new(head); new(z);
  head^.next:=z; tail^.next:=z; z^.next:=z;
end;
procedure put(v:integer);
var t:link;
begin
  new(t);
  t^.key:=v; t^.next:=tail^.next;
  tail^.next:=t;
end;
function get:integer;
var t:link;
begin
  t:=head^.next;
  get:=t^.key;
  head^.next:=t^.next;
  dispose(t);
end;

```

```

function queueempty:boolean;
begin queueempty:=(head^.next=z;tail^.next=z) end.

```

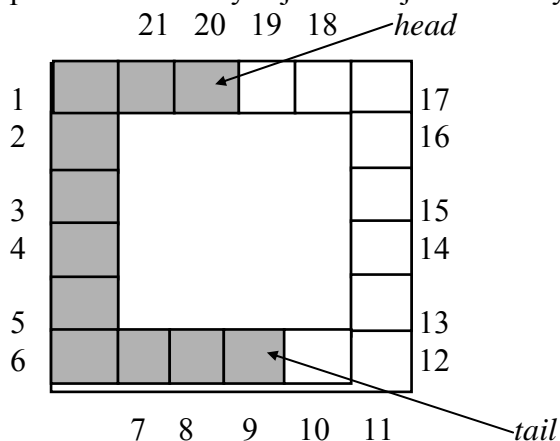
Eilės operacijos, išreikštos baziniu masyvo tipu:

```

const max=100;
var queue:array[0..max] of integer;
    head,tail:integer;
procedure put(v:integer);
begin
    queue[tail]:=v; tail:=tail+1;
    if tail>max then tail:=0
end;
function get: integer;
begin get:=queue[head]; head:=head+1;
    if head>max then head:=0
end;
procedure queueinitialize;
    begin head:=0; tail:=0 end;
function queueempty:boolean;
    begin queueempty:=(head=tail) end.

```

Programuojant eilę masyvu (kaip pateikta), jo pradžią ir pabaigą tikslinga "suklijuoti", t.y. kintamųjų *head* ir *tail*, rodančių eilės pradžią ir pabaigą, reikšmes reikia skaičiuoti **mod** *Max* pagrindu (aišku, su reikalingais tikrinimais). Įterpiant ar išmetant elementus iš eilės, ji turi savybę "šliaužti" per masyvą. Tam, kad efektyviai išnaudotume visą masyvą eilei saugoti, reikia leisti eilės pabaigai užimti ir masyvo pradžios elementus (jei jie laisvi). Būtent tokia programa ir pateikta šiame skyriuje. "Suklijuotas" masyvas eilei saugoti parodytas 2 pav.



2 pav. Eilės dėstymas "suklijuotame" masyve.

## 1.4 Dekas (*dequeue*)

Duomenų struktūra dekas - tai duomenų aibė, kuriai apibrėžtos tokios operacijos:

- inicializuoti deką (išskirti vietą dekui kompiuterio atmintyje);
- įterpti tam tikrą elementą  $x$  į deko pradžią;
- įterpti tam tikrą elementą  $x$  į deko pabaigą;
- pašalinti elementą iš deko pradžios;
- pašalinti elementą iš deko pabaigos;

- skaityti deko pradžią;
- skaityti deko pabaigą;
- panaikinti deką (panaikinti vietą deku kompiuterio atmintyje).

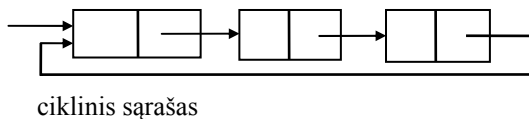
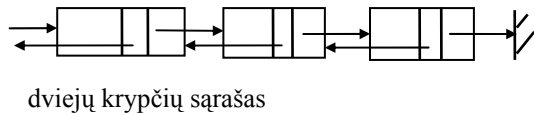
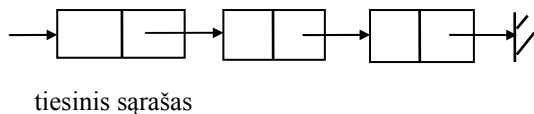
Duomenims deke taikomi loginiai apribojimai - operacijos pašalinimo iš deko pradžios ar pabaigos metu bus išmestas elementas, įterptas vėliausiai atitinkamai į pradžią ar į pabaigą. Todėl dekas, kaip ir eilė ar stekas, yra viena iš riboto išrinkimo klasės struktūrų. Dekas dar vadinamas abipusiu steku (ar abipuse eile). Jo programavimas naudojant bazinį masyvo ar rodyklės tipą panašus į steko ar eilės programavimą, tik reikalauja sudėtingesnės realizacijos.

## 1.5 Tiesiniai sąrašai (*linked lists*)

Duomenų struktūra tiesinis sąrašas - tai duomenų aibė, kurios kiekvienas elementas susideda iš dviejų dalių - informacinės dalies ir rodyklės - ir kuriai taikomos tokios operacijos:

- inicializuoti sąrašą;
- įterpti elementą  $x$  į sąrašo tam tikrą vietą;
- įterpti elementą  $x$  į sąrašą, tenkinant nurodytas sąlygas (pvz., didėjimo tvarka);
- išmesti iš sąrašo elementą, esantį tam tikroje vietoje;
- išmesti elementą  $x$  iš sąrašo, tenkinant nurodytas sąlygas;
- panaikinti sąrašą.

Sąrašai, kaip įprasta juos pateikti programavimo literatūroje, pavaizduoti 3 pav.



3 pav. Sarašų pavyzdžiai

Loginiai apribojimai gali būti taikomi tik sąrašo elementams, bet ne visam sąrašui. Elementams reikia nurodyti jų tipą (programavimo procese). Sąrašas yra dinaminė duomenų struktūra, t.y. jo turis vykdymo metu gali keistis, t.y. jam išskiriama tiek atminties, kiek elementų yra sąrašė. Programuojant sąrašą, reikia remtis rodyklės ar masyvo baziniais tipais. Naudojant bazinį rodyklės tipą, sąrašas programuojamas taip:

```

type link=^node;
      node=record key:integer; next:link end;
var head.z.t:link;
procedure listinitialize;
begin
      new (head); new (z);

```

```

    head^.next:=z; z^.next:=z
end;
procedure deletenext (t:link);
begin
    t^.next:=t^.next^.next;
end;
procedure insertafter (v:integer; t:link);
    var x:link;
begin
    new (x);
    x^.key:=v; x^.next:=t^.next;
    t^.next:=x
end;

```

Sąrašo programavimas, naudojant bazinį masyvo tipą:

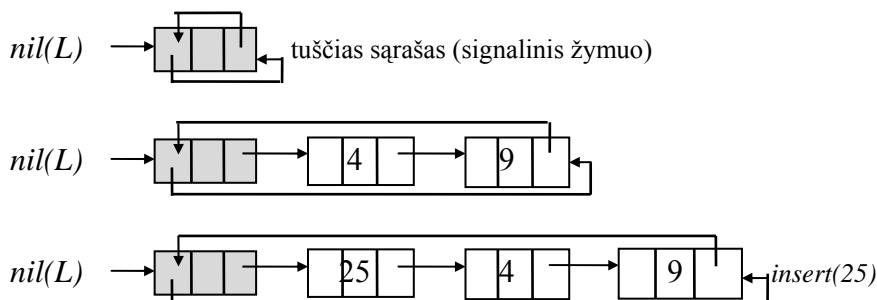
```

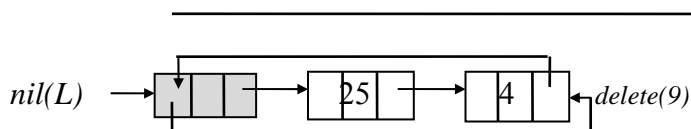
var key,next: array[0..N]of integer; x,head,z: integer;
procedure listinitialize;
begin
    head:=0; z:=1; x:=1;
    next[head]:=z; next[z]:=z;
end;
procedure deletenext(t:integer);
begin
    next[t]:=next[next[t]]
end;
procedure insertafter(v:integer; t:integer);
begin
    x:=x+1;
    key[x]:=v; next[x]:=next[t];
    next[t]:=x
end.

```

Skirtingai nuo steko, eilės ar deko, sąrašas įgalina išrinkti bet kurį elementą, tačiau išrinkimo metodas yra tiesinis, t.y. tam tikrą elementą pasiekiami nuosekliai eidami nuo vieno elemento prie kito. Jei reikia "gudresnių" išrinkimo metodų, sąrašas gali būti apibrėžiamas su keliomis rodyklėmis (kurių apibrėžimas ir naudojimas priklauso nuo duomenų turinio).

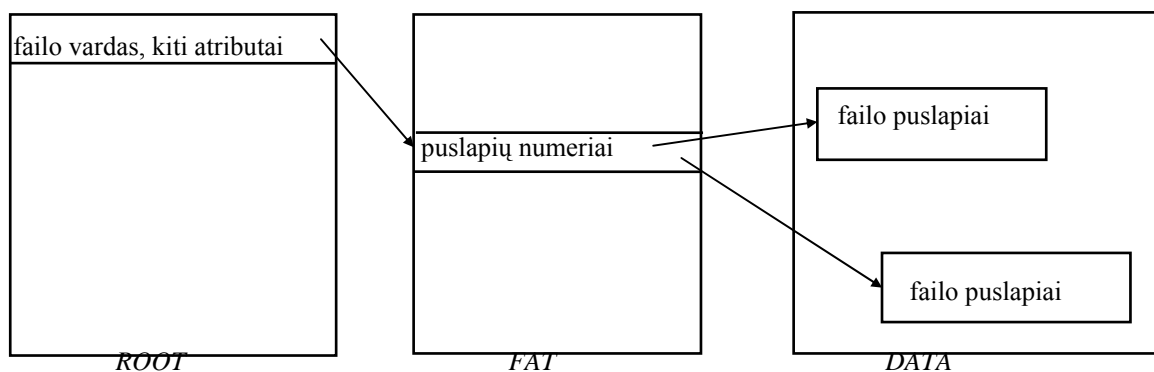
Programuojant sąrašus, (pvz., procedūras *insertafter* ar *deletenext*), patogu ir svarbu apibrėžti signalinį žymenį (*sentinel*). Signalinis žymuo - tai menamas objektas, kuris suprastina sąrašo pirmo ir paskutinio elementų atskyrimą (identifikavimą): signaliniu sąrašo *L* žymeniu būtų objektas *nil(L)*, kurio reikšmė yra *nil*, bet kuris turi visą kitų sąrašo elementų struktūrą. Jis gali būti kintamųjų *head* ir *tail* pakaitalas. Viena iš galimybių, kaip naudoti signalinį žymenį, pavaizduota 4 pav. Ji įgalina atsisakyti kintamojo *head*, nes sąrašo pradžią galima pasiekti per *next(nil(L))*.





4 pav. Sąrašas, naudojantis signalinį žymenį.

Gal būt labiausiai paplitęs sąrašų taikymas - duomenų dėstymas diskinėje atmintyje. Tai tiesiogiai susiję su diskinių įrenginių konstrukcija. Mažiausio adresuojamo informacinio vieneto diske (vadinsime jį toliau puslapiu - angliškai *page*) turis yra keli arba kelios dešimtys kilobaitų. Kai failas rašomas į diskinę atmintį, jis užima dažnai ne vieną puslapį. Šie puslapiai gali būti išdėstyti diske nebūtinai nuosekliai. Todėl būtina formuoti nuorodas iš vieno puslapio į kitą. Jų taip pat reikia identifikuojant laisvą diskinę atmintį, t.y. tuos puslapius, kurie dar nepriklauso jokiai failui. Skirtingos operacinės sistemos formuoja puslapių nuorodas skirtingai. Pavyzdžiui, MS-DOS operacinė sistema nuorodas į failo (t.y. tiesinio sąrašo) pradžią rašo *ROOT* srityje, nuorodas į puslapius (vadinamuosius klasterius - angliškai *cluster*) *FAT* srityje, o patį failą dėsto *DATA* srityje. Loginė tokio dėstymo schema pateikta 5 pav.



5 pav. Loginė disko struktūra.

## 1.6 Medžiai (*trees*)

Iki šiol nagrinėtos duomenų struktūros buvo vienmatės - vienas duomuo sekė po kito. Tokias DS gan paprasta programuoti ir naudoti, tačiau taikymams jos gali būti neefektyvios, t.y. algoritmas, veikiantis jų pagrindu, gali būti per lėtas. Todėl verta nagrinėti sudėtingesnes - dvimates, trimates ir pan. duomenų struktūras. Viena iš tokių yra medžio duomenų struktūra.

Medis - tai duomenų aibė, kuriai apibrėžtos duomenų incidentumo ir eiliškumo operacijos:

- medžio (žymime  $T$ ) elementai yra dvi aibės: viršūnių aibė  $V$  ir briaunų aibė  $E$ ;
- tarp  $V$  ir  $E$  apibrėžtas incidentumo santykis: viena briauna (elementas iš  $E$ ) atitinka dvi ir tik dvi viršūnes (elementų iš  $V$  porą, elementų tvarka poroje yra fiksuota);
- viena viršūnė medyje yra išskiriama ir vadinama šaknimi;
- bet kurias dvi viršūnes jungia vienas ir tik vienas kelias (kelias yra briaunų seka, kurioje kiekvienos dvi gretimos briaunos turi bendrą viršūnę).

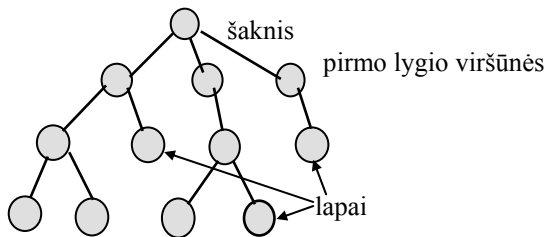
Manipuliavimo su duomenimis operacijos tokios:

- inicializuoti medį (išskirti vietą būsimai medžio struktūrai kompiuterio atmintyje);
- įterpti elementą į medį (įterpti naują viršūnę);
- išmesti elementą iš medžio (pašalinti viršūnę);
- sujungti du medžius;
- išskaidyti medį į du medžius;

- sutvarkyti medį (išdėstyti medžio viršūnes taip, kad pagal tam tikrą viršūnių perrinkimo tvarką, jų reikšmės atitiktų norimas sąlygas);
- panaikinti medį (išlaisvinti medžiui skirtą kompiuterio atmintyje vietą).

Šis medžio apibrėžimas yra pritaikytas duomenų struktūroms ir orientuotas į algoritmų teoriją ir jų programavimą. Literatūroje galima rasti ir kitokių medžio apibrėžimų, dažnai bendresnių ir turinčių teorinių ypatumų (gan retai pasitaikančių praktikoje). Todėl čia pateikiami apibrėžimai ir sąvokos yra būtiniausi ir kartais intuityvūs.

Viršūnė medyje dažniausiai yra struktūrizuota (atitinka formuojamas algoritme reikšmes ar sąlygas, dažnai gan sudėtingas). Ji gali turėti vardą ir prie jos papildomą informaciją, taip pat tipą ar netgi duomenų struktūrą. Briauna irgi gali identifikuoti papildomą informaciją (pvz. svorį), gali turėti tipą. Kelias iš šaknies į kurią nors viršūnę yra vadinamas šaka. Medis yra braižomas šakomis žemyn, šaknį dėstant viršuje (6 pav.).



6 pav. Medžio pavyzdys.

Nors briaunos paprastai neturi krypties (medis yra neorientuotas grafas), tačiau programuojant briaunos realizuojamos rodyklėmis ir intuityviai joms priskiriama kryptis. Jei ši kryptis iš viršūnės rodo į kitą viršūnę žemyn nuo šaknies, tai sakoma, kad viršūnė turi sūnų. Briaunos kryptis į priešingą pusę (viršų) nurodo viršūnei jos tėvą. Viršūnė be sūnų vadinama lapu arba išorine viršūne. Visos kitos viršūnės vadinamos vidinėmis. Vieno tėvo sūnūs vadinami broliais (*siblings*). Jei kiekvienas tėvas turi ne daugiau kaip fiksuotą skaičių sūnų (tarkime,  $n$ ), jis vadinamas  $n$ -ariniu medžiu. Algoritmų teorijoje ypač svarbūs yra dvejetainiai medžiai. Medis, prasidedantis kurioje nors viršūnėje, vadinamas pomedžiu. Miškas yra dviejų ir daugiau medžių aibė. Medis turi lygius: šaknis yra nulinio lygio, jos sūnūs yra pirmo lygio, šių sūnų sūnūs yra antro lygio ir t.t. Medžio aukščiui vadinamas lygių maksimumas (arba maksimalios šakos ilgis).

Viena iš svarbių susijusių su algoritmais sąvokų, yra sutvarkytas medis. Medis sutvarkytas, kai jo viršūnėms priskirta tam tikra numeracija (eilės tvarka). Medžio viršūnės gali būti numeruojamos skirtingais būdais (paprastai nuo jų priklauso algoritmo efektyvumas).

Medžio numeracija gali būti žinoma, t.y. nurodyta tvarka, kuria vardinamos (ar spausdinamos, ar peržiūrimos) medžio viršūnės. Algoritmuose ar programavime aptinkamos tokios numeracijos (apibrėžimai rekursyvūs ir pateikti dvejetainiam medžiui):

- infiksiniis medis (angliškai *infix* arba *inorder*) - medžio viršūnės vardinamos taip: kairysis pomedis, viršūnė, dešinysis pomedis;
- prefiksiniis medis (angliškai *prefix* arba *preorder*) - medžio viršūnės vardinamos taip: viršūnė, kairysis pomedis, dešinysis pomedis;
- postfiksiniis medis (angliškai *postfix* arba *postorder*) - medžio viršūnės vardinamos taip: kairysis pomedis, dešinysis pomedis, viršūnė.

Šie apibrėžimai gali būti lengvai apibendrinti  $n$ -ariniams medžiams. Jie yra analogiški algebrinių ar loginių reiškinių infiksiniams, prefiksiniams, postfiksiniams išraiškoms.

**Infiksinė reiškinių išraiška** - tai įprastas reiškinių užrašymas su skliausteliais:

$$(((A + B) * (C - D) + E) * F)$$



**Prefiksinė išraiška** rašoma (rekursyviai) - operacijos ženklas, pirmas operandas, antras operandas:

$$* + * + A B - C D E F$$

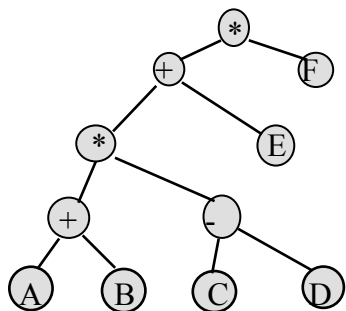
**Postfiksinė išraiška** rašoma (rekursyviai) - pirmas operandas, antras operandas, operacijos ženklas:

$$A B + C D - * E + F *$$

Šią išraišką atitinkantis sintaksinis (*parse*) medis pavaizduotas 7 pav.

Prefiksinėms ir postfiksinėms reiškinių išraiškoms skliausteliai gali būti nenaudojami. Postfiksinė išraiška dar kitaip vadinama atvirkštine lenkiška notacija (pagerbiant lenkų logikus-matematikus, įvedusius tokias išraiškas). Visų šių išraiškų atitikmenys gali būti dvejetainiai medžiai (infiksiniai, prefiksiniai, postfiksiniai), tačiau tik jų elementų numeravimo, o ne skaičiavimo prasme.

Dar vienas galimas medžių tvarkymo būdas - viršūnių numeracija pagal lygius: nulinio lygio viršūnė (šaknis); visos pirmo lygio viršūnės (numeruojamos iš kairės į dešinę); visos antro lygio viršūnės (taip pat iš kairės į dešinę); ir t.t.



7 pav. Reiškinių  $((A + B) * (C - D) + E) * F$  sintaksinis medis.

Medžio struktūra yra iš esmės rekursyvi. Galima pateikti tokį rekursyvų medžio apibrėžimą:

- **Medis yra arba viršūnė, arba šakninė viršūnė, sujungta briaunomis su medžių aibe.**

Elementarios, tačiau naudingos medžio savybės yra tokios:

1. Medis su  $N$  viršūnių turi  $N-1$  briaunų;
2. Dvejetainis medis su  $N$  vidinių viršūnių, turi  $N+1$  išorinių viršūnių;
3. Dvejetainio medžio su  $N$  vidinių viršūnių išorinio kelio ilgis yra  $2N$  didesnis už vidinio kelio ilgį;
4. Pilno dvejetainio medžio aukštis  $n$  su  $N$  vidinių viršūnių gali būti įvertinamas skaičiumi  $\log_2 N$ :

$$2^{n-1} < N + 1 \leq 2^n.$$

## 1.7. Medžių dėstymas kompiuterio atmintyje

Vienas iš labiausiai paplitusių medžių dėstymo atmintyje būdų - tai viršūnių elementų rašymas į įrašus su nuorodomis. Nuorodos nukreiptos iš tėvo viršūnės į sūnus. Nuorodų skaičius įrašė turi būti lygus maksimaliam medžio viršūnių sūnų skaičiui. Jei medis dvejetainis, užtenka dviejų nuorodų: *left* (*l*) ir *right* (*r*). Priklausomai nuo algoritmo kartais reikia turėti dviejų tipų įrašus: vidinėms viršūnėms ir išorinėms viršūnėms, nes jų panaudojimas algoritmo vykdymo metu gali būti specifinis.

Istoriškai svarbus ir esminis yra ryšys tarp dvejetainių medžių (jų tvarkymo) ir algebrinių reiškinių. Kaip matome iš ankstesnio pavyzdžio ir 7 pav., algebriniam reiškiniui galima priskirti sintaksinį (dvejetainį) medį ir jį naudoti reiškiniui analizuoti.

Sintaksinis medis (*parse tree*), atitinkantis algebrinį reiškinį, apibrėžiamas rekursyviai: priskirti operaciją šakniai, pirmo operando reiškinį priskirti kairiajam pomedžiui, antro operando reiškinį priskirti dešiniajam pomedžiui. Turint postfiksiniu būdu užrašytą reiškinį, programa, sukurianti sintaksinį medį, gali būti tokia:

```

type link=^node;
      node=record info:char; l,r:link end;
var x,z:link; c:char;
begin
  stackinit;
  new(z); z^.l:=z; z^.r:=r;
repeat
  repeat read(c) until c<>";
  new(x); x^.info:=c;
  if (c='*') or (c='+')
  then begin x^.r:=pop; x^.l:=pop end
  else begin x^.r:=z; x^.l:=z end;
  push(x)
until eoln;
end.

```

Ši paprasta programa gali būti lengvai modifikuota sudėtingesniems reiškiniams. Tačiau pasiūlytas mechanizmas yra labai bendras: jį galima taikyti reiškinių reikšmėms skaičiuoti, kurti programas reiškinių reikšmėms skaičiuoti, rašyti sintaksinius analizatorius programavimo kalboms ir pan.

Kai medis sukurtas, viena iš pirmiausia iškylančių problemų yra medžio numeravimas, t.y. kokia tvarka apeiti ar perrinkti (*traverse*) medžio viršūnes. Skirtingiems taikymams naudotina skirtinga numeracija. Prefiksinei numeracijai yra tinkama tokia programa:

```

procedure traverse(t:link);
begin
  push(t);
  repeat
  t:=pop;
  visit(t);
  if t<>z then push(t^.r);
  if t<>z then push(t^.l);
  until stackempty;
end;

```

Kitoms dviem medžių numeracijoms - infiksinei ir postfiksinei - programos gali būti parašytos analogiškai, tik šiek tiek modifikuojant pateiktąją.

Tuo tarpu medžių numeracijai pagal lygius programa gali būti gauta iš ankstesnės, operacijas su steku programos struktūroje pakeitus analogiškais operacijomis su eile:

```

procedure traverse(t:link);
begin
  put(t);
  repeat

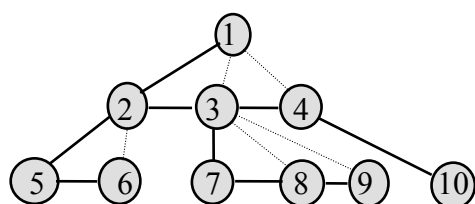
```

```

t:=get;
visit(t);
if t<>z then put(t^.l);
if t<>z then put(t^.r);
until queueempty;
end;

```

Antras medžio struktūros dėstymo kompiuterio atmintyje būdas - naudoti įrašus tik su dviem nuorodomis (bet kuriam n-ariniam medžiui) ir šias nuorodas interpretuojant taip: kairioji (*left*) nuoroda identifikuoja viršūnės kairijį sūnų, tuo tarpu dešinioji nuoroda rodo viršūnės dešinį brolių. Šis dėstymo būdas reikalauja įvesti tam tikrą tvarką tarp brolių (*siblings*), kad atskirtume, kuris iš brolių yra kairėje, o kuris - dešinėje. Tokio dėstymo pavyzdys yra 8 pav.



8 pav. "Kairysis sūnus - dešinysis brolis" būdu išdėstytas medis.

Abu šie metodai nekelia kokių nors reikalavimų įrašams su nuorodomis, daugiausia priklauso nuo taikomojo algoritmo - ar bazinį tipą tokiems įrašams parinksime masyvą, ar rodyklės tipą. Reikia tik turėti omenyje, kad bazinis tipas irgi daro įtaką, ir kartais esminę, algoritmo greičiui (apie tai kituose skyriuose).

Trečias medžio struktūros dėstymo būdas - vadinamieji rekursyvūs medžiai - gali būti naudojamas tada, kai algoritme dažniausiai reikia naudoti nuorodas iš sūnų į jų tėvus. Medžio viršūnės sunumeravus pagal lygius, galima jas rašyti į vienmatį masyvą taip, kad šio masyvo indeksai atitiktų viršūnių numerius, o masyvo elementų reikšmės būtų nuorodos į tėvo numerį. Toks medžio organizavimas įgalina labai greitai "vaikščioti" medžio šakomis nuo sūnų prie tėvų ir nuo vienos viršūnės prie kitos, ir, jei tokios procedūros taikomajame uždavinyje yra pagrindinės, programa dirba labai greitai, jos sudėtingumas yra tiesinis. Medžio iš 8 pav. rekursyvus dėstymas pavaizduotas 9 pav.

1	2	3	4	5	6	7	8	9	10
0	1	1	1	2	2	3	3	3	4

9 pav. Medžio rekursyvus dėstymas.

## 1.8 Rekursija

Rekursija yra viena iš pirminių ir pagrindinių matematikos ir informatikos sąvokų. Rekursija matematikoje vadinamas toks funkcijų apibrėžimo metodas, kai funkcijos reikšmės, atitinkančios bet kokius argumentus, yra apibrėžiamos naudojant tos pačios funkcijos reikšmes, atitinkančias mažesnius argumentus. Analogiškai programavime, rekursija yra programų (procedūrų, algoritmų) apibrėžimo (sudarymo) metodas, kai programa kreipiasi pati į save, esant mažesnėms argumentų (parametrų) reikšmėms. Rekursyviai programai reikia papildomai apibrėžti tą atvejį, kai pasiektos mažiausios galimos argumentų reikšmės (baigmės sąlyga). Rekursyvių programų pavyzdžiai medžiams numeruoti ar viršūnėms perrinkti jau buvo pateikti anksčiau. Žinomi klasikiniai rekursyvių algoritmų pavyzdžiai - tai faktorialo skaičiavimas ar Fibonačio skaičių sekos apibrėžimas:

```

function factorial (N:integer):integer;
begin
    if N=0
    then factorial:=1
    else factorial:=N*factorial(N-1);
end;

```

```

function fibonacci(N:integer):integer;
begin
    if N<=1
    then fibonacci:=1
    else fibonacci:=fibonacci(N-1)+fibonacci(N-2);
end;

```

Rekursijos stiprybė - ji įgalina žymiai suprastinti algoritmo formulavimą ar programos konstrukciją. Tačiau ja naudoti reikia atsargiai. Pavyzdžiui, Fibonačio skaičių sekos apibrėžimas naudojant rekursiją, kaip ką tik pateiktame pavyzdyje, šią programą daro neefektyvią: skaičiai  $fibonacci(N-1)$  ir  $fibonacci(N-2)$  gaunami nepriklausomai vienas nuo kito, tuo tarpu juos nesunku gauti vieną iš kito. Fibonačio algoritmas, formuluojamas be rekursijos, yra efektyvesnis (operacijų skaičius proporcingas N):

```

procedure fibonacci;
const max=25;
var i:integer;
F:array[0..max] of integer;
begin
    F[0]:=1; F[1]:=1;
    for i:=2 to max do F[i]:=F[i-1]+F[i-2]
end;

```

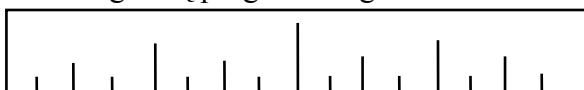
Rekursija glaudžiai susijusi su tam tikra algoritmų konstravimo metodika, vadinamąja skaldyk ir valdyk (*divide and conquer*) paradigma. Pagal tokią paradigmą sukurtas algoritmas naudoja du rekursyvius programos kvietimus, tačiau nesikertančioms argumentų reikšmėms. Jos pavyzdys - pateikiama liniuotės gradavimo programa:

```

procedure rule(l,r,h:integer);
var m:integer;
begin
    if h>0 then
    begin
        m:=(l+r)div2;
        mark(m,h);
        rule(l,m,h-1);
        rule(m,r,h-1)
    end
end;

```

Pagal šią programą sugraduota liniuotė parodyta 10 pav.



10 pav. Graduota liniuotė.

Liniuotės gradavimo nerekursyvus algoritmas gali būti gautas:

- beveik tiesmukiškai iš rekursyvaus algoritmo;
- pakeičiant gradavimo žymių piešimo tvarką.

Kita vertus, pateiktų infiksinio ir pagal lygius sutvarkyto medžių algoritmų rekursyvi realizacija žymiai suprastina algoritmų programavimą. Visa tai gali būti palikta kaip pratimai skaitytojiui.

"Skaldyk ir valdyk" paradigma apima tris tipinius kiekvieno rekursijos lygio žingsnius:

- išskaidyti uždavinį į kelis dalinius uždavinius;
- valdyti rekursyviai kiekvieno dalinio uždavinio sprendimą, jei uždavinys tapo paprastas, išspręsti jį tiesiogiai;
- sujungti dalinių uždavinių sprendinius ir suformuoti išeities uždavinio sprendinį.

Ši paradigma taikoma algoritmams, kaip operacijų sekoms, ir duomenų struktūroms konstruoti. Kiti du plačiai naudojami metodai - tai rekursija ir iteracija. Visos toliau nagrinėjamos struktūros ar su jomis susiję algoritmai bus pagrįsti vienu iš šių metodų.

## 1.9 Algoritmų analizė

Algoritmų analizė yra būtina reikalingiems uždavinio realizavimui kompiuterinės sistemos resursams įvertinti. Resursai - tai uždavinio skaičiavimo laikas (dažniausiai pats svarbiausias resursas), reikalingos atminties tūris, komunikacinių srautų dydis, loginių įvadų (*port*) skaičius ir pan.

Prieš pradedant algoritmo analizę, reikia sukurti formalųjį diegimo technologijos modelį. Nagrinėsime algoritmus darbui vieno procesoriaus aplinkoje (nuoseklus algoritmų vykdymo būdas). Šie algoritmai naudos arba tiesioginio išrinkimo atmintį (*RAM* - *random access memory*), arba diskinę atmintį (toks naudojimas bus specialiai aptartas), disponuos tam tikru kiekiu (*N*) pradinių duomenų (kartais skirstomų į tam tikras klases). Algoritmą analizuosime kaip kompiuterinę programą, vykdomą remiantis tam tikromis bazinėmis operacijomis. Tokių operacijų, jų tipų ar klasių išskyrimas visada susijęs su taikomuoju uždaviniu ir yra būtinas.

Bet kurio net ir paprasčiausio algoritmo analizė - tai tam tikra matematinė problema, kartais net labai sudėtinga. Jai spręsti naudojamos įvairios matematinės priemonės - kombinatorika, elementari tikimybių teorija, įmantrios algebrinės manipuliacijos, kartais ir kitos sudėtingesnės, matematinės teorijos. Be to, dar reikia daug išmonės ir intuicijos.

1. Pradiniai duomenys, jų kiekio apibrėžimas priklauso nuo taikymų. Rūšiavimo uždaviniuose pradiniai duomenys - tai kiekis duomenų, kuriuos turime sutvarkyti. Skaičių daugybės algoritmuose - tai bitų skaičius, reikalingas dauginamiems skaičiams išdėstyti atmintyje. Duomenų bazėse - tai įrašų ar laukų juose kiekis. Telekomunikacijos uždaviniuose - maksimalus siunčiamo pranešimo ilgis, ir t.t.

2. Algoritmo veikimo laikas paprastai yra proporcingas vykdomų operacijų skaičiui. Kadangi skirtingos operacijos gali būti įvykdytos per skirtingą laiką, algoritmų analizės atveju naudojamos operacijos yra skirstomos į grupes, kiekvienai iš jų atskirai įvertinant operacijų kiekį.

Įvertinus algoritmo vykdymo laiką, svarbu žinoti, kaip elgsis algoritmas, jei keisime pradinių duomenų kiekį. Tam nebūtina žinoti tikslų laiką  $L$  išreiškiančių formulių, o tik jų asimptotiką. Todėl algoritmų analizės atveju yra labai populiarūs vertinimai:  $O(1)$  - rodo pastovų operacijos atlikimo laiką;  $o(K)$  - rodo, kad laikas visa eile arba keliomis eilėmis mažesnis už  $K$ ;  $O(K)$  - rodo, kad laikas proporcingas reiškiniui  $K$ , t.y. egzistuoja konstanta  $C$ , kad

$$L < C * K$$

Be to, vertinant algoritmų darbą dažnai tikslinga išskirti du atvejus: vidutinį algoritmo darbo laiką (vidutinis atvejis arba laukiamas atvejis - *average-case analysis*) ir algoritmo darbo laiką tuo atveju, kai jam pateikiami blogiausi duomenys (blogiausias atvejis - *worst-case analysis*). Praktiškai daugiausia dėmesio analizėje yra skiriama blogiausiam atvejui, nes tam yra tokios priežastys:

- blogiausio atvejo laikas yra viršutinė laukiamo atvejo laiko riba, jo įvertinimas garantuoja, kad algoritmas niekada nedirbs ilgiau;
- dažnai taikymuose blogiausias atvejis pasirodo ne taip jau retai, pvz., ieškant tam tikros informacijos duomenų bazėse, blogiausias atvejis paieškos požiūriu iškyla visada, kai tik bazėse tokios informacijos nėra;
- dažnai laukiamas atvejis yra toks pat blogas, kaip ir blogiausias atvejis, t.y. abiem atvejais vertinimai sutampa.

3. Algoritmui veikti reikalingos atminties kiekis irgi yra analizės objektas. Šitokios analizės ypač reikia algoritmams, formuluojamiems remiantis rekursija arba paradigma "skaldyk ir valdyk", nes jų atveju reikia vertinti rekursijos gylį, o kuo gilesnė rekursija, tuo daugiau atminties reikia algoritmo vykdymui. Atminties kiekio vertinimai taip pat reikalingi algoritmams, dinamiškai apibrėžiantiems duomenims vietą atmintyje. Šie vertinimai yra sudėtingesni negu statinių duomenų laikymo atmintyje struktūrų atveju, nes dinamiškai adresuojant duomenis algoritmui kartais reikia labai nedaug atminties.

## 1.10. Algoritmų diegimas

Algoritmai retai egzistuoja "vakuume", be sąsajų su taikymais, nebent tai būtų grynai teoriniai, hipotetiniai algoritmai. Jie paprastai sudaro esminę dalį sistemos, kurioje yra diegiami, nors ta dalis gali būti ir nedidelė. Algoritmų veikimo aplinka būna gan sudėtinga, todėl ir visos realizavimo procedūros nagrinėjimas - tai labai komplikuota analizė. Čia paminėsime tik esminius momentus.

1. **Algoritmo parinkimas.** Paprastai yra kelios ar keliolika procedūrų, kurios tinka realizuoti norimą taikymą. Jų diapazonas būna gan platus - nuo "brutalių jėgų" algoritmų iki logiškai sudėtingos veiksmų sekos. Pirmiausia turime atsakyti į tokius klausimus, kurie yra svarbūs visai vykdymo eigai:

- kaip dažnai programa bus naudojama;
- kokios yra kompiuterinės sistemos charakteristikos;
- ar algoritmas sudaro tik mažą dalį viso uždavinio, ar atvirkščiai?

Pagrindinė taisyklė, kuria vadovaujamasi diegimo metu,- realizuoti patį paprasčiausią uždavinio sprendimui tinkamą algoritmą, nuosekliai įvertinant atsakymus į tik ką pateiktus klausimus. Tai turi būti daroma kruopščiai. Dvi dažniausiai pasikartojančios šioje situacijoje klaidos - tai algoritmo veikimo charakteristikų reikšmės pervertinimas arba nepakankamas vertinimas. Paprastai greitesnį algoritmą sudėtingiau programuoti, bet tam tikram duomenų kiekiui jo greičio rodikliai gali būti esminiai.

2. **Empirinė analizė.** Labai gaila, bet dažniausiai matematinė algoritmo analizė duoda per mažai informacijos apie realų algoritmo efektyvumą, todėl reikia tam tikro empirinio tikrinimo. Pirmiausia čia galioja triviali taisyklė - jei yra du algoritmai, reikia juos abu išmėginti kiek galima panašesnėmis sąlygomis. Vadinasi, ne tik kompiuterinės sistemos ar duomenys turi būti panašūs, bet ir dėmesys algoritmų programavimui ir optimizavimui turi būti tas pats. Po to, greitesnio algoritmo veikimo ar realizavimo sąlygos turi būti kruopščiau analizuojamos.

**3. Programų optimizavimas.** Paprastai šis procesas suprantamas kaip kompiliatoriaus priemonių visuma, kai yra automatiškai generuojamas optimalus kodas. Tačiau čia turimi omenyje būtent algoritmo darbo pagerinimai. Pirmiausia, algoritmą reikalinga realizuoti paprasčiausia galima forma. Antra, reikia minimizuoti bazinių operacijų skaičių, išvengiant bereikalingų ciklų ar kitų pasikartojančių veiksmų sekų. Trečia, algoritmo atliekamus veiksmus reikia analizuoti techninių ir programinių priemonių atžvilgiu, t.y. kokia yra naudojamo kompiuterio architektūra, kaip dirba kompiliatorius ir pan. Tokia periodiškai atliekama empirinė analizė gali kartais padėti atrasti naujus algoritmo darbo aspektus, ir iš esmės pagerinti algoritmo darbą.

**4. Algoritmai ir sistemos.** Tas pats algoritmas naudojamas ir mažose programose, ir sudėtingose sistemose. Šiomis skirtingomis sąlygomis jo realizavimas irgi yra skirtingas. Pavyzdžiui, rūšiavimo algoritmas priklausomai nuo to, ar jis rūšiuoja natūrinius skaičius, ar labai ilgus sudėtingos struktūros įrašus, turės būti skirtingai realizuotas. Vėlgi, jei algoritmas dirba sistemoje, kuri realizuota lokaliame ar globaliame kompiuterių tinkle, jo veikimo charakteristikos bus kitokios negu algoritmo, dirbančio atskirame kompiuteryje. Dažnai tokiais atvejais detali algoritmo analizė sąlygoja išvadą, kad algoritmas gali būti "išlygiagretintas", t.y. jis gali būti išskaidytas lygiagrečiam darbui skirtingose tinklo kompiuteriuose su skirtingomis duomenų aibėmis.

Visi minėti algoritmų analizės ir realizavimo aspektai gali būti taikomi ir atskiroms programoms, ir programų sistemoms. Tačiau sistemų atveju yra ir daug skirtumų. Dabartiniu metu sistemų projektavimo atveju populiariu naudoti objektų orientavimo metodiką, pagal kurią programa turi tenkinti klasės, į kurią ji patenka, duomenų, funkcinių procedūrų ir pan. reikalavimus. Programa turi būti įjungta į sistemą kaip atskiras modulis. Visa tai siejasi su algoritmo sudėtingumu.

## 2. Rūšiavimo metodai

Rūšiavimas - tai viena iš bazinių kompiuterių darbo operacijų, turinčių informatikoje tokią pat reikšmę, kaip matematikoje - sudėties operacija.

Šioje knygoje rūšiavimas - tai algoritmas, dėstantis duomenis tam tikra tvarka. Tokio algoritmo darbą sąlygoja labai daug aspektų. Duomenų tvarkos apibrėžimas, duomenų struktūra, rūšiuojamų laukų išskyrimas, atminties panaudojimas rūšiavimo operacijai, antrinės ar netgi tretinės atminties poreikis, duomenų pateikimo rūšiavimui vienašakiškumas, eiliškumas (nuosekliai ar grupėmis), rūšiavimas lygiagrečiose sistemose, kiti veiksniai - visa tai turi didelę įtaką rūšiavimo algoritmo darbo efektyvumui.

Dėstant rūšiavimo algoritmus, pradedama nuo paprasčiausių ir klasikinių situacijų, pereinama prie sudėtingesnių (tačiau ne visų galimų).

### 2.1 Elementarūs rūšiavimo algoritmai

Rūšiavimo algoritmai priklausomai nuo to, ar naudoja tik vidinę kompiuterio atmintį, ar jiems reikia ir išorinės (antrinės) skirstomi į vidinio rūšiavimo ir išorinio rūšiavimo algoritmus. Pastarieji naudojami, kai duomenys vienu metu negali būti išdėstyti atmintyje.

Papildomos atminties (vidinės) kiekis, reikalingas algoritmų darbui, irgi yra svarbus rodiklis. Rūšiavimo algoritmai gali būti skirstomi į grupes pagal tai, kiek papildomos atminties naudoja (nenaudoja visiškai; naudoja papildomai tik atmintį, reikalingą rodyklėms išdėstyti; naudoja papildomai tiek pat atminties, kiek yra duomenų).

Rūšiavimo algoritmai gali būti stabilūs ir nestabilūs. Stabilūs algoritmai nekeičia lygių (lyginimo požįūriu) elementų tvarkos.

Nagrinėdami algoritmus, tarsime, kad duomenys, kuriuos algoritmas turi rūšiuoti - tai paprasti skaičiai, laisvai talpinami kintamųjų ląstelėse. Taip galime geriau sukcentruoti dėmesį į algoritmo darbo specifiką.

Paprasčiausias ir primityviausias rūšiavimo algoritmas (rūšiuojantis tris elementus):

```
program threesort (input , output );
const maxN=100;
var a : array [1 .. maxN] of integer;
N , i : integer;
procedure sort3 ;
var t : integer ;
begin
if a [1] >a [2] then
begin t:=a [1] ; a [1] :=a [2] ; a [2] :=t end ;
if a[1] >a [3] then
begin t :=a [1]; a [1] :=a [3] ; a [3] :=t end ;
if a [2] >a [3] then
begin t:=a [2] ; a [2] :=a [3] ; a [3] :=t end ;
end;
begin
readln (N) ;
for i:=1 to N do read (a [i]) ;
if N=3 then sort3;
for i:=1 to N do write (a [i]) ;
writeln
end.
```

#### 2.1.1. Išrinkimo (*selection*) algoritmas



Vienas iš paprasčiausių rūšiavimo algoritmų - išrinkimo vadovaujasi principu: minimalų elementą reikia rašyti į pirmą duomenų sekos vietą. Po to tą patį principą reikia taikyti posekiui be pirmojo elemento, ir t.t. Programa, realizuojanti šį algoritmą:

```
procedure selection;
var i,j,min, t: integer ;
begin
    for i:=1 to N -1 do
        begin
            min :=i ;
            for j :=i+1 to N do
                if a [j]<a [min] then min :=j ;
            t:=a [min]; a [min]:=a [i]:=t
            end;
        end;
end;
```

Nors šis algoritmas priklauso "brutalios jėgos" algoritmams, jis dažnai naudojamas labai ilgiems įrašams su trumpais laukais rūšiuoti. Tą sąlygoja šio algoritmo ypatybė, kad kiekvienas iš elementų bus perkeltas į kitą vietą ne daugiau kaip vieną kartą.

Pagrindinės algoritmo darbo operacijos - tai duomenų lyginimas ir keitimas vietomis. Vertinant algoritmo efektyvumą, nesunku įrodyti, kad jis naudoja apytikriai  $N^2/2$  lyginimų ir  $N$  keitimų.

### 2.1.2. Įterpimo algoritmas

Įterpimo algoritmas yra beveik toks pat paprastas kaip ir išrinkimo, bet labiau lankstus. Pradedant nuo pirmojo, kiekvienas sekos elementas  $x_i$  yra lyginamas su prieš jį esančiu, ir, jei jų tvarka netinkama, sukeičiami vietomis. Jei toks keitimas įvyko, lyginama pora  $(x_{i-1}, x_i)$ , ir t.t. Lyginimų serija nutraukiama, jei kurios nors poros elementų sukeisti vietomis nereikia. Šį algoritmą realizuoja tokia programa:

```
procedure insertion;
    var i,j,v: integer;
begin
    for i:=2 to N do
        begin
            v:=a[i]; j:=i;
            while a[j-1]>v do
                begin a[j]:=a[j-1]; j:=j-1 end;
            a[j]:=v
        end
    end;
```

Šis algoritmas gerai žinomas bridžo žaidimo mėgėjams - taip jie rūšiuoja turimas kortas. Taikomuojų požiūriu jis labiau tinkamas situacijai, kai duomenų keitimo vietomis operacija yra lengvai vykdoma. Programuojant šį algoritmą, reikia parinkti tinkamą bazinę duomenų struktūrą (masyvas čia mažiau tinka). Naudojant sąrašus, tikslinga į juos įvesti ir signalinį žymenį (*sentinel*), tai žymiai palengvina programos konstravimą.

Pagrindinės algoritmo operacijos yra tokios pat, kaip ir ankstesnio. Įterpimo algoritmas laukiamu atveju naudoja apytikriai  $N^2/4$  lyginimų ir  $N^2/8$  keitimų vietomis ir dvigubai daugiau operacijų blogiausiu atveju. Įterpimo metodas yra tiesinis beveik surūšiuotiems duomenims.

### 2.1.3. Burbulo algoritmas

Šio algoritmo idėja - nuosekliai iš dešinės į kairę peržiūrint gretimų elementų poras ir jei reikia, elementus sukeičiant vietomis, į sekos pradžią perkelti mažesnius elementus. Taip elgiantis, mažesni elementai pasislenka į duomenų sekos pradžią, o pirmoje sekos vietoje atsiduria mažiausias elementas. Po to galima pritaikyti tą patį principą duomenų posekiui be pirmojo elemento, ir t.t. Tai panašu į virimo procesą, kai oro burbuliukai kyla į paviršių - iš čia kilęs algoritmo pavadinimas. Algoritmą realizuoja tokia programa:

```

procedure bubble;
  var i,j,t: integer;
begin
  for i:=N downto 1 do
    for j:=2 to i do
      if a[j-1]>a[j] then
        begin t:=a[j-1]; a[j-1]:=a[j]; a[j]:=t end
end;

```

Burbuliuko algoritmas apytikriai naudoja  $N^2/2$  lyginimų ir  $N^2/2$  keitimų vietomis ir laukiamu, ir blogiausiu atvejais.

Nė vienam iš trijų pateiktų algoritmų nereikia papildomos arba tarpinės atminties. Tolesnis algoritmų darbo arba jų tinkamumo vienai ar kitai situacijai vertinimas turi apimti lyginimo ir keitimo vietomis operacijų "kainą" ir jų santykį. Operacijų vykdymo laikas akivaizdžiai priklauso nuo įrašų ilgio, laukų, pagal kuriuos rūšiuojama, santykio, įrašų ilgių santykio ir pan. Jei šie faktoriai napalankūs, galima keisti algoritmų realizaciją. Pavyzdžiui, **rūšiuojant ilgus įrašus**, galima įvesti papildomą įrašų nuorodų masyvą ar sąrašą ir lyginimo atveju manipuluoti šio masyvo reikšmėmis. Taip galima pasiekti, kad bet kuris iš jau nagrinėtų algoritmų naudotų tik N įrašų keitimų vietomis. Pavyzdžiui, įterpimo algoritmas šiuo atveju būtų programuojamas taip:

```

procedure insertion;
  var i,j,v: integer;
begin
  for i:=1 to N do p[i]:=i;
  for i:=2 to N do
    begin
      v:=p[i]; j:=i;
      while a[p[j-1]]>a[v] do
        begin p[j]:=p[j-1]; j:=j-1 end;
      p[j]:=v
    end
end;

```

Po to, kai nuorodų masyvas surūšiuotas, reikia spręsti įrašų keitimo vietomis problemą. Optimaliai tai galima daryti fiksuojant vietas, kur koks įrašas turi būti:

```

procedure insitu;
  var i,j,k,t: integer;
begin
  for i:=1 to N do
    if p[i]<>i then
      begin t:=a[i]; k:=i;

```

```

repeat j:=k; a[j]:=a[p[j]];
k:=p[j]; p[j]:=j;
until k=i;
a[j]:=t
end;
end.

```

#### 2.1.4. Kevalo (*shellsort*) algoritmas

Išvardyti algoritmai yra universalūs ta prasme, kad nekreipia dėmesio į duomenų sekos ypatumus. Tačiau jei šie ypatumai yra žinomi, algoritmus galima pertvarkyti taip, kad jie dirbtų efektyviau. Kevalo algoritmas yra nesudėtinga įterpimo metodo modifikacija. Duomenų seka interpretuojama kaip  $h$  surūšiuota, t.y. imant nuo bet kurios vietos kiekvieną  $h$ -ąjį elementą gaunamas surūšiuotas posekis. Skaičių  $h$  keičiant pagal kokį nors dėsnį, pvz.,  $h := 3 * h + 1$ , kevalo algoritmas visiškai apibrėžtas, jo realizavimas pateikiamas programoje:

```

procedure shellsort;
label 0;
    var i,j,h,v: integer;
begin
    h:=1; repeat h:=3*h+1 until h>N;
    repeat
    h:=h div 3;
    for i:=h+1 to N do
    begin
    v:=a[i]; j:=i;
    while a[j-h]>v do
    begin
    a[j]:=a[j-h]; j:=j-h;
    if j <= h then goto 0
    end;
0:a[j]:=v
    end
    until h=1;
end;

```

Šio algoritmo darbo efektyvumas priklauso nuo skaičiaus  $h$  kitimo dėsnio parinkimo ir nuo duomenų sekos. Algoritmo sudėtingumą atskleidžia toks teiginys, kurį sunku matematiškai įrodyti:

Kevalo algoritmas naudoja ne daugiau kaip  $N^{3/2}$  lyginimų (esant  $h$  kitimo dėsniui 1, 4, 13, 40, 121, ...).

Kitas geras šio metodo ypatumas - jį lengva programuoti netgi sudėtingesniems duomenims. Todėl jis dažnai aptinkamas įvairiuose taikymuose.

#### 2.1.5. Pasiskirstymo skaičiavimas

Labai paprasta rūšiuoti specifinėje situacijoje, kai skaičiai keičiasi nuo 1 iki  $N$ , belieka duomenis perkelti į vietas, kurių numeriai lygūs jų reikšmėms:

```

for i:= 1 to N do t[a[i]]:= a[i];

```

Jei turima  $N$  skaičių, kurie yra intervale nuo 1 iki  $M$ , rūšiuoti galima irgi panašiu būdu, tik reikia žinoti, kiek kartų kiekvienas skaičius sekoje pasikartoja. Kai  $M$  nėra labai didelis, seką galima pereiti du kartus, pirmą kartą apskaičiuojant elementų pasikartojimus sekoje, o antrą kartą perkeltiant tuos elementus į jiems priklausančias vietas. Šitokią algoritmą realizuoja programa:

```

for j:=0 to M-1 do count[j]:=0;
    for i:=1 to N do
        count[a[i]]:=count[a[i]]+1;
for j:=1 to M-1 do
    count[j]:=count[j-1]+count[j];
for i:=N downto 1 do
    begin
        b[count[a[i]]]:=a[i];
        count[a[i]]:=count[a[i]]-1
    end;
for i:=1 to N do a[i]:=b[i];

```

## 2.2 Greito rūšiavimo algoritmas

Šis metodas, angliškai vadinamas *quicksort*, pasiūlytas C. A. R. Hoare 1962-ais metais. Jis yra labai paplitęs ir aptinkamas daugelyje taikymų. Algoritmo realizavimo paprastumas ir efektyvus atminties naudojimas sąlygojo jo populiarumą. Algoritmo teigiamos savybės:

- papildoma atmintis beveik nenaudojama;
- algoritmo sudėtingumas yra  $O(N \log N)$  laukiamu atveju;
- algoritmo realizavime gali būti apibrėžti labai trumpi vidiniai ciklai.

Algoritmo neigiamos savybės:

- algoritmas rekursyvus, todėl diegimas komplikluotas, kai nėra rekursijos mechanizmo;
- algoritmo sudėtingumas yra  $O(N^2)$  blogiausiu atveju;
- algoritmas labai jautrus programavimo klaidoms.

*Quicksort* remiasi paradigma "skaldyk ir valdyk". Pagrindinė idėja - išskaidžius duomenų seką į dvi dalis taip, kad vienoje iš jų visi elementai būtų mažesni už kitos dalies elementus, toliau šias dvi dalis galima rūšiuoti nepriklausomai viena nuo kitos. Todėl parinkus elementą, kuris galima sakyti, jau yra savo vietoje ir naudotinas kaip slenkstis, galima toliau skaidyti seką į dvi dalis tokio elemento atžvilgiu. Tai daro *partition* procedūra šioje scheminėje programoje:

```

procedure quicksort(j,r:integer);
var i;
begin
    if r>l then
        begin
            i:=partition(l,r);
            quicksort(l,i-1);
            quicksort(i+1,r);
        end
    end;

```

Parametrai  $l$  ir  $r$  rodo, kuri sekos dalis šiuo metu yra rūšiuojama, parametras  $i$  rodo slenkstinio elemento numerį. Kruopščiai išanalizavus reikalingus veiksmus, algoritmą galima

vykdyti gan tobulai, pvz., suradus kairėje sekos pusėje elementą, kuris didesnis už slenkstį ir kurį reikia perkelti į dešinę pusę, galima prieš tai dar surasti elementą dešinėje, kuris mažesnis už slenkstį, ir tik tada abu šiuos elementus sukeisti vietomis - taip yra taupomi veiksmai ir atmintis. Realizavus *partition* procedūrą ir tik ką aprašytą principą, algoritmo tekstas būtų toks:

```
procedure quicksort (l,r: integer);
var v,t,i,j: integer;
begin
  if r>l then
    begin
      v:=a[r]; i:=l-1; j:=r;
      repeat
        repeat i:=i+1 until a[i]>=v;
        repeat j:=j-1 until a[j]<=v;
        t:=a[i]; a[i]:=a[j]; a[j]:=t;
      until j<=i;
      a[j]:=a[i]; a[i]:=a[r]; a[r]:=t;
      quicksort (l,i-1);
      quicksort (i+1,r)
    end
end;
```

Aišku, šis algoritmas yra nestabilus, jis gali ne tik keisti lygių elementų tvarką, bet ir juos dėstyti skirtingose vietose. Jis taip pat nestabilus *partition* procedūros atžvilgiu. Algoritmo efektyvumo ypatumus sąlygoja tokie faktoriai:

- **rekursijos eliminavimas** (reikia turėti omenyje, kad būtina kruopščiai valdyti algoritmo vykdymo eigą ir iš anksto numatyti nepalankių arba išsigimusių sekų atvejus):

```
procedure quicksort;
  var t,i,l,r: integer;
begin
  l:=1; r:=N; stackinit;
  push(l); push(r);
  repeat
    if r>l then
      begin
        i:=partition(l,r);
        if (i-l)>(r-i)
          then begin push(l); push(i-1); l:=i+1 end
          else begin push(i+1); push(r); r:=i-1 end;
        end
      else
        begin r:=pop; l:=pop end;
      until stackempty;
  end;
```

- **trumpi posekiai** - kai reikia rūšiuoti trumpą posekį, tarkime, tokį, kad  $r - l \leq M$ , tikslinga naudoti kokį nors tiesioginį metodą, pvz., įterpimą, ribinį skaičių M parenkant tokį, kokio reikia;

- **slenksčio parinkimas** - ši problema irgi svarbi greito rūšiavimo algoritme; dažniausiai čia naudojamas arba didesnio iš pirmųjų dviejų nesutampančių sekos elementų principas, kaip aprašyta anksčiau, arba vadinamasis medianos iš trijų elementų principas.

### 2.2.1. Greito rūšiavimo algoritmo taikymas elementams išrinkti

Pagrindinė algoritmo idėja gali būti panaudota ne vien duomenims rūšiuoti. Labai dažnas toks uždavinys: iš skaičių sekos išrinkti  $k$  mažiausių. Kitas pavyzdys - medianos apskaičiavimas. Šitokius ir panašius uždavinius (dažnai minimus ranginės statistikos vardu), aišku, galima spręsti rūšiuojant turimą skaičių seką. Tačiau tai yra neefektyvu. Kitoks galimas sprendimo būdas - parinkti tinkamą elementą ir pritaikius *partition* procedūrą, suskaidyti seką į dvi dalis, iš kurių vienoje yra  $k$  mažesnių už parinktą elementą, o kitoje yra  $(N-k)$  didesnių. Atitinkama programa būtų tokia:

```

procedure select( $l, r, k$ : integer);
var  $i$ ;
begin
    if  $r > l$  then
        begin  $i := \text{partition}(l, r)$ ;
        if  $i > l + k - 1$  then select( $l, i - 1, k$ );
        if  $i < l + k - 1$  then select( $i + 1, r, k - 1$ );
        end
    end;

```

Kadangi ši procedūra iškviečia save tik vieną kartą, ji iš esmės nėra rekursyvi ir eliminuoti rekursiją nėra sudėtinga. Tam nebūtina naudoti netgi steko, nes procedūra, grįždama vėl į savo pradžią, atnaujinama parametrų reikšmės. Be to, galime sumažinti skaičiavimų kiekį, jei parinkdami slenkstį naudosime skaičių  $k$ :

```

procedure select( $k$ : integer);
var  $v, t, i, j, l, r$ : integer;
begin
     $l := 1$ ;  $r := N$ ;
    while  $r > l$  do
        begin
             $v := a[r]$ ;  $i := l - 1$ ;  $j := r$ ;
            repeat
                repeat  $i := i + 1$  until  $a[i] \geq v$ ;
                repeat  $j := j - 1$  until  $a[j] \leq v$ ;
                 $t := a[i]$ ;  $a[i] := a[j]$ ;  $a[j] := t$ ;
            until  $j < i$ ;
             $a[j] := a[i]$ ;  $a[i] := a[r]$ ;  $a[r] := t$ ;
            if  $i \geq k$  then  $r := i - 1$ ;
            if  $i \leq k$  then  $l := i + 1$ ;
        end;
    end;

```

### 2.2.2. Greito rūšiavimo algoritmo sudėtingumas

Intuityviai aišku, kad efektyviausiai *Quicksort* algoritmas veikia, kai slenkščiai skaido duomenis į dvi dalis po lygiai. Šiuo atveju labai paprasta teigti, kad algoritmo sudėtingumas yra

$O(N \log N)$ , o lyginimo operacijų yra naudojama apie  $2N \log_2 N$ . Galima teigti, nors įrodyti yra sudėtingiau, kad ir laukiamu atveju galioja tokie įvertinimai [3].

Į trumpus posekius, rekursijos eliminavimą ir slenksčio parinkimą dėmesys buvo atkreiptas ne veltui, nes šių ypatybių tinkamas realizavimas gali pagreitinti *Quicksort* laiką apie 25 - 30%.

Elementų išrinkimo sudėtingumas taikant *Quicksort* metodą laukiamu atveju yra tiesinis [3].

### 2.3. Skaitmeninis rūšiavimas (*radixsort*)

Rūšiuojami duomenys dažnai yra sudėtingi, pvz., įrašai duomenų bazėse, ar telefonų sąrašai, ar bibliotekiniai katalogai. Apie šių įrašų laukų reikšmes nieko apibrėžto negalima pasakyti. Tada galima tik išskirti dvi pagrindines operacijas - lyginimo ir keitimo vietomis - ir jų atžvilgiu vertinti vieną ar kitą rūšiavimo algoritmą. Taip buvo elgiama iki šiol. Jei apie įrašų ar laukų reikšmes galima pasakyti ką nors papildomo, šią informaciją tikslinga panaudoti rūšiavimo procedūroms. Šiame skyriuje bus nagrinėjami algoritmai, kai duomenų reikšmės yra skaitmeninės ir priklauso kokiam nors skaitiniam intervalui ar išsiskiria panašiomis savybėmis.

Skaitmeninio rūšiavimo algoritmuose duomenų reikšmės interpretuojamos kaip skaičiai  $M$ -ainėje skaičiavimo sistemoje. Priklausomai nuo reikšmių  $i$ -oje pozicijoje skaičiai gali būti suskirstyti į  $M$  grupių, po to kiekviena iš šių grupių gali būti lygiai taip pat suskirstyta į  $M$  pogrupių, priklausomai nuo reikšmių  $j$ -oje pozicijoje, ir t.t.. Įvedus vienokį ar kitokį pozicijų parinkimo dėsnį, gaunami išsamiai apibrėžti skaitmeninio rūšiavimo algoritmai.

Čia bus nagrinėjami algoritmai esant  $M = 2$  (turint omenyje kompiuterinę realizaciją), o pozicijos keisis pereinant nuo  $i$  prie  $i+1$  arba  $i-1$  arba jas grupuojant po dvi ir pan. Šioje situacijoje, programuojant skaitmeninio rūšiavimo algoritmus, tikslinga turėti funkcijas tiesioginiam darbui su bitais. Standartinis Paskalis tokių galimybių neturi, nors daugelio dabartinių Paskalio realizacijų darbo su bitais funkcijos vienaip ar kitaip yra realizuotos. Užrašant algoritmus, programavimo kalbos tekste bus naudojama funkcija: *function bits (x, k, j: integer): integer*. Ji apibrėžia dvejetainį skaičių, kurį sudaro  $j$  bitų, paimtų iš skaičiaus  $x$  dvejetainės išraiškos, praleidus  $k$  bitų iš dešinės. Ši funkcija ekvivalenti tokiai Paskalio instrukcijai:  $x \text{ (div } 2^k) \bmod 2^j$ .

Reikia pastebėti, kad taikant skaitmeninį rūšiavimą skaičiai turi būti ne tokie jau maži. Jei skaičiai nedideli ir kiekvieną jų sudaro ne daugiau kaip  $b$  bitų, naudojant pasiskirstymo skaičiavimo algoritmą duomenis galima rūšiuoti taip, kad rūšiavimo laikas tiesiškai priklausytų nuo duomenų kiekio. Blieka tik atmintyje išskirti  $2^b$  dydžio lentelę, kad  $b$  bitų ilgio skaičiais galima būtų laisvai disponuoti.

Dažniausiai naudojamos dvi skaitmeninio rūšiavimo procedūros: skaitmeninio keitimo (*radix exchange sort*) ir tiesioginio skaitmeninio rūšiavimo (*straight radix sort*), besiskiriančios apdorojamų bitų tvarka. Pirmasis metodas pagrįstas leksikografinė tvarka ir bitų pozicijas numeruoja iš kairės į dešinę. Tai reiškia, kad skaičiai, prasidedantys dvejetainiu nuliu, rūšiuojamoje sekoje pasirodys anksčiau negu skaičiai, prasidedantys dvejetainiu vienetu. Kai reikia dvejetaines pozicijas analizuoti ir skaičius keisti vietomis, procedūrą galima taikyti, panašiai kaip *Quicksort* algoritme:

```

procedure radixexchange(l,r,b: integer);
    var t,i,j: integer;
begin
    if (r>l) and (b>=0) then
        begin i:=l; j:=r;
        repeat
            while (bits(a[i],b,1)=0) and (i<j) do i:=i+1;
            while (bits(a[j],b,1)=1) and (i<j) do j:=j-1;
            t:=a[i]; a[i]:=a[j]; a[j]:=t;
        until j=i;
    end

```

```

if bits(a[r],b,1)=0 then j:=j+1;
radixexchange(l,j-1,b-1);
radixexchange(l,r,b-1);
end
end;

```

Tiesioginio skaitmeninio rūšiavimo metodas, skirtingai nuo skaitmeninio keitimo algoritmo, analizuoja bitus iš dešinės į kairę. Jis iš pradžių rūšiuoja duomenis pagal dešiniausią poziciją, po to pagal prieš tai esančią poziciją, po to dar prieš tai ir t.t. Elementams, kurių bitai  $j-1$  pozicijoje sutampa, šis metodas nekeičia tarpusavio tvarkos, t.y. jis yra stabilus. Labai panašiai elgdavosi senos skaičiavimo mašinos, rūšiuodamos perfokortas:

```

procedure straightradix;
  var i,j,pass:integer;
count:array[0..M]of integer;
begin
  for pass:=0 to (w div m)-1 do
    begin
      for j:=0 to M-1 do count[j]:=0;
      for i:=1 to N do
        count[bits(a[i],pass*m,m)]:=count[bits(a[i],pass*m,m)]+1;
      for j:=1 to M-1 do
        count[j]:=count[j-1]+count[j];
      for i:=N downto 1 do
        begin
          b[count[bits(a[i],pass*m,m)]]:=a[i];
          count[bits(a[i],pass*m,m)]:=count[bits(a[i],pass*m,m)]-1;
        end;
      for i:=1 to N do a[i]:=b[i];
    end;
  end;
end;

```

Abiejų skaitmeninio rūšiavimo algoritmų savybės:

1. *Radixexchange* metodas naudoja apie  $N \lg N$  bitų lyginimų.
2. Abu skaitmeniniai metodai, rūšiuodami  $N$  skaičių, kurių kiekvienas yra  $b$  bitų ilgio naudoja mažiau negu  $Nb$  bitų lyginimų.
3. Tiesioginis metodas rūšiuoja  $N$  skaičių, kurių kiekvienas  $b$  bitų ilgio, kartodamas algoritmą  $b/m$  kartų (jei išskiriama papildoma atmintis  $2^m$  skaitliukų saugoti, o taip pat buferis pertvarkyti failą).

## 2.4 Prioritetinės eilės (*priority queues*)

Daugelio taikymų duomenis tenka rūšiuoti tik iš dalies, nes pilnas rūšiavimas nėra būtinas, arba kartais tenka rūšiuoti tik duomenų poaibius. Pavyzdžiui, duomenų bazėse dažnai tenka sukaupti tam tikrą kiekį duomenų, iš jų išrinkti didžiausią (mažiausią), po to vėl papildomai kaupti aibę duomenų, iš jų vėl išrinkti didžiausią, ir t.t. Šias operacijas atitinkančiai duomenų struktūrai netinka naudoti eilės ar steko, ar kitos iki šiol nagrinėtos duomenų struktūros. Žemiau nagrinėjama duomenų struktūra yra geriau pritaikyta tokiai situacijai.

Duomenų struktūra, vadinama **prioritetine eile**, pritaikyta operacijoms:

- inicializuoti aibę iš  $N$  elementų;



- konstruoti iš aibės prioritetinę eilę (*construct*);
- įterpti naują elementą (*insert*);
- išmesti didžiausią elementą (*remove*);
- pakeisti didžiausią elementą nauju, jei šis nėra didžiausias (*replace*);
- pakeisti elemento prioritetą (*change*);
- išmesti bet kurį nurodytą elementą (*delete*);
- sujungti dvi prioritetines eiles į vieną (*join*).

Šios operacijos nėra tiksliai apibrėžtos. Formalizuojant situaciją, reikia jas patikslinti. Pirmiausia, jei aibėje yra kelios didžiausios reikšmės, imama bet kuri iš jų. Antra, operacija "pakeisti didžiausią elementą nauju" yra beveik ekvivalenti operacijoms "įterpti" ir "išmesti", o skirtumas tarp jų yra toks: naudojant dvi operacijas, prioritetinė eilė laikinai padidėja vienu elementu. Trečia, operacija "sujungti" kelia tam tikrus reikalavimus duomenų saugojimo atmintyje struktūrai. Taikymuose panašiai reikia analizuoti ir kitas duomenų struktūros operacijas. Reikia turėti omenyje, kad prioritetinėms eilėms, nors ir turinčioms detalų formalizuotą aprašymą, skirtingos DS realizacijos sąlygoja skirtingus algoritmų darbo rezultatus. Tačiau šis realizacijos faktorius yra vienintelis, dalyvaujantis duomenų struktūros apibrėžimo kontekste.

Prioritetinių eilių struktūros nagrinėjimas pradedamas nuo elementarios realizacijos (nesutvarkyto sąrašo), kai programuojamos pagrindinės operacijos naudojant masyvą:

**procedure** *insert*(*v:integer*);

**begin**

*N:=N+1; a[N]:=v;*

**end;**

**function** *remove:integer*;

**var** *j,max: integer*;

**begin**

*max:=1;*

**for** *j:=2 to N do*

**if** *a[j]>a[max]* **then** *max:=j;*

*remove:=a[max];*

*a[max]:=a[N]; N:=N-1;*

**end;**

Šitaip programuojant, skirtingų operacijų vykdymo laikas gali labai skirtis, operacija *insert* visada vykdoma per fiksuotą laiką, o operacijos *remove* ar *delete* savo ruožtu gali pareikalauti perrinkti visus masyvo elementus.

Kai prioritetinė eilė programuojama naudojant sutvarkytą sąrašą, operacijų atlikimo laikas iš esmės keičiasi - dabar jau operacijoms *insert* ar *delete* reikia laiko, proporcingo  $\log N$ , o operacijai *remove* - fiksuoto laiko.

Kai programuojant prioritetinę eilę masyvas keičiamas tiesiniu sąrašu, operacijų vykdymo laikas iš esmės nesikeičia, nepriklauso nuo to, ar tiesinį sąrašą interpretuosime kaip sutvarkytą ar nesutvarkytą sąrašą, operacijų sudėtingumo analizė bus tokia pat. Pastebėsime tik, kad operacijos *delete* ir *join* atliekamos per pastovų laiką.

#### 2.4.1. Duomenų struktūra *heap*

Viena iš tinkamų duomenų struktūrų, garantuojančių efektyvų prioritetinės eilės operacijų vykdymą, yra vadinamoji *heap* struktūra (kartais vadinama piramidės metodu). *Heap* struktūra - tai pilnas dvejetainis medis, realizuotas naudojant masyvą, kuriame nuorodos nukreiptos iš tėvo viršūnės į sūnus. Medis turi būti sutvarkytas pagal lygius, t.y. kiekvienai viršūnei patenkinama "*heap*" sąlyga:

### tėvo reikšmės turi būti didesnės už jo vaikų reikšmes.

Iš čia išplaukia, kad didžiausias aibės elementas turi būti medžio šaknyje. Kadangi medis pilnas, nesunku apskaičiuoti tėvų ar sūnų vietas masyve: jei tėvas yra  $j$ -oje masyvo vietoje, jo sūnūs bus  $2j$ -oje ir  $(2j+1)$ -oje masyvo vietose. Todėl nereikia naudoti jokių specialių nuorodų, o pereinant nuo tėvo prie sūnaus ar atvirkščiai, pakanka padauginti atitinkamą indeksą iš 2 (ir gal būt pridėti 1) arba atitinkamai atlikti **div** 2 operaciją. Realizuojant prioritetinės eilės operacijas šioje struktūroje, reikės vaikščioti medžio šakomis. Šakų didžiausias ilgis yra  $\log N$ , todėl visos prioritetinės eilės operacijos, išskyrus *join*, bus  $\log N$  sudėtingumo. Pavadinimo *heap* sinonimas - masyvas, turintis dalinai sutvarkyto medžio savybę.

#### 2.4.2. Operacijos su *heap* struktūra

Visos prioritetinės eilės operacijos, realizuojamos *heap* struktūroje, paprastai atliekamos taip:

- masyve apibrėžiama vieta, nuo kurios pradedama operacija, pvz., naujas elementas visada įterpiamas į fiksuotą (dažniausiai paskutinę) vietą masyve;
- įterptas elementas (ar elementas, esantis parinktoje vietoje) automatiškai interpretuojamas kaip tam tikros viršūnės sūnus;
- šakoje nuo įterpto elemento iki šaknies yra tikrinama *heap* savybė;
- jei elementų reikšmės neatitinka *heap* sąlygos, jie keičiami vietomis.

Programuojant *heap* struktūrą, atmintyje reikia apibrėžti masyvą ir dar vieną kintamąjį, kuriame bus fiksuojama operacijos pradžios vieta (šį kintamąjį žymėsime raide  $N$ ). Pavyzdžiui, operacija *insert* bus realizuojama taip: naują elementą įterpiame masyve į  $N+1$  vietą, o po to šakoje tarp šios vietos ir šaknies (t.y. masyvo pirmo elemento) tikriname *heap* savybę, ir, jei reikia, elementus keičiame vietomis. Šitokioje aplinkoje pastoviai bus naudojama operacija, kuri ieškos tėvo viršūnės, ir kurią tikslinga programuoti kaip atskirą procedūrą:

```

procedure upheap(k:integer);
var v:integer;
begin
    v:=a[k]; a[0]:=maxint;
    while a[k div 2]<=v do
        begin a[k]:=a[k div 2]; k:=k div 2 end;
    a[k]:=v
end;

```

Įterpimo operacijos programa atrodys taip:

```

procedure insert(v:integer);
begin
    N:=N+1; a[N]:=v;
    upheap(N)
end;

```

Prioritetinės eilės operacija *replace* gali būti realizuota taip: naują elementą rašome į medžio šaknį, (pirmą elementą masyve), po to stumiame tą elementą žemyn viena ir kita medžio šaka tol, kol jis atsiduria savo vietoje. Šaką, kurią reikia parinkti stumiant elementą žemyn, sąlygoja didesnis viršūnės sūnus.

Operacija *remove* atliekama labai panašiai: į medžio šaknies vietą rašomas N-asis masyvo elementas, po to jis stumiamas žemyn, panašiai kaip operacijoje *replace* tol, kol atsistos į savo vietą. Abi šios operacijos naudoja viršūnės sūnaus radimo procedūrą:

```
procedure downheap(k:integer);
label 0;
var i,j,v:integer;
begin
    v:=a[k];
    while k<=N div 2 do
        begin
            j:=k+k;
            if j<N then if a[j]<a[j+1] then j:=j+1;
            if v>a[j] then goto 0;
            a[k]:=a[j]; k:=j;
        end;
    0: a[k]:=v;
end;
```

Tada operacijos *replace* ir *remove* operacijos bus programuojamos taip:

```
function replace(v:integer):integer;
begin
    a[0]:=v;
    downheap(0);
    replace:=a[0];
end;
```

```
function remove:integer;
begin
    remove:=a[1];
    a[1]:=a[N]; N:=N-1;
    downheap(1);
end;
```

Visų trijų prioritetinės eilės operacijų sudėtingumas bus toks:

- Jei aibėje yra N elementų, šioms operacijoms atlikti reikės mažiau negu  $2\log N$  lyginimų.

### 2.4.3. Heapsort algoritmas

Naudojant išvardytas prioritetinės eilės operacijas, galima įvesti ir realizuoti naują rūšiavimo algoritmą (vadinamąjį *heapsort*): duomenų aibei sudaryti *heap* struktūrą, po to spausdinti elementą, išmetamą *remove* operacijos metu, ir taip elgtis tol, kol aibė nebus tuščia. Visi išmetami elementai išsirikiuos mažėjančia tvarka. Formaliai užrašytas algoritmas bus toks:

```
N:=0;
for k:=1 to M do insert (a[k]);
for k:= M downto 1 do a[k]:= remove;
```

Šiam algoritmui nereikia papildomos atminties, jo sudėtingumas bus  $M\log M$ . Aišku, anksčiau parašytos programos tekstas formaliai neatitinka duomenų tipų taisyklių, nes tas pats

kintamasis interpretuojamas ir kaip masyvo, ir kaip *heap* elementas, tačiau šitoks užrašymas labai vaizdus. Realiai programuojant, programos tekstą reikia rašyti vientisą, vengiant procedūrų bereikalingų iškvietimų.

Truputį geriau (logiškiau) konstruoti *heap* struktūrą iš apačios į viršų. Tada algoritmas būtų programuojamas taip:

```
procedure heapsort;
var k,t:integer;
begin
    N:=M;
    for k:=M div 2 downto 1 do downheap(k);
    repeat
        t:=a[1]; a[1]:=a[N]; a[N]:=t;
        N:=N-1; downheap(1)
    until N=<1;
end;
```

Tokio algoritmo realizavimo sudėtingumas bus toks:

- *heap* struktūra bus sudaroma per tiesinį laiką;
- *heapsort* algoritmui reikės mažiau negu  $2M \lg M$  lyginimų.

#### 2.4.4. Netiesioginė *heap* duomenų struktūra

Daugelyje taikomųjų uždavinių nepageidautina įrašus kilnoti iš vienos vietos į kitą, tačiau juos rūšiuoti vis tiek reikia. *Heapsort* algoritmas gali būti efektyviai pritaikytas ir tokiai aplinkai. Vietoje rūšiuojamų masyvo **a** įrašų galima operuoti su jo indeksų masyvu **p**, apibrėžiant jį taip, kad elementas **a[p[k]]** nurodo į *heap* struktūros k-ąjį elementą. Dėl pilnumo reikia įvesti kitą masyvą **q**, kuriame būtų laikomos nuorodos į vietą, kurią *heap* struktūroje užima k-asis masyvo **a** elementas. Naudojant šiuos masyvus, *heap* struktūrą galima sudaryti taip:

```
procedure pqconstruct;
var k:integer;
begin
    N:=M;
    for k:=1 to N do
        begin p[k]:=k; q[k]:=k end;
    for k:=M div 2 downto 1 do pqdownheap(k);
end;
```

Programuojant netiesioginę *heap* algoritmą, reikia modifikuoti tik tas ankstesnės programos vietas, kur kreipiamasi į masyvo **q** elementus, pakeičiant juos atitinkamomis masyvo **p** reikšmėmis:

```
procedure pqdownheap(k:integer);
label 0;
var j,v:integer;
begin
    v:=p[k];
    while k<N div 2 do
        begin
            j:=k+k;
```

```

if j<N then if a[p[j]]<a[p[j+1]] then j:=j+1;
if a[v]>=a[p[j]] then goto 0;
p[k]:=p[j]; q[p[j]]:=k; k:=j;
end;
0: p[k]:=v; q[v]:=k
end;

```

Kitos netiesioginės *heap* duomenų struktūros operacijos - *pqinsert*, *pqremove*, ir t.t., - gali būti panašiai aprašytos. Jei masyvo **p** reikšmės yra nuorodos į atskirai saugomus įrašus, o ne į masyvo **a** elementus, kaip ką tik nagrinėtoje situacijoje, programuojant didesnis dėmesys turi būti kreipiamas į masyvo **q** sudarymą ir manipuliavimą jo elementais.

Čia nagrinėtos prioritетinių eilių realizacijos neapėmė operacijos *join*. *Heap* struktūra yra per daug nelanksti ir per griežta, kad leistų realizuoti operaciją *join* logaritminiame laike kaip kitas prioritетinių eilių operacijas. Tam sukurtos kitos detalesnės ir lankstesnės struktūros, tinkamos visoms prioritетinių eilių operacijoms. Jos gali būti naudojamos ne vien tik operacijos *join*, bet ir kitais atvejais, pvz., kai duomenis reikia laikyti ne viename vientisame masyve, bet keliuose atskiruose, logiškai susietuose į vieną aibę. Tokios struktūros taip pat padeda laikytis principo, kad visas operacijas turime atlikti pagal vieningą metodiką ir jos turi būti vienodo sudėtingumo (pvz., logaritminis laikas).

#### 2.4.5. Aibės duomenų struktūra

Su ką tik nagrinėta duomenų struktūra *heap* yra susijusi **aibės** duomenų struktūra, t.y. duomenys, kuriems būdingos tokios operacijos:

- inicializuoti aibę **S** (išskirti vietą atmintyje ir parinkti dėstymo metodą joje);
- *memberof* (**x**, **S**) - funkcija, kuri patikrina ar elementas **x** priklauso aibei **S**;
- *insert* (**x**, **S**) - įterpti elementą **x** į aibę **S**;
- *delete* (**x**, **S**) - pašalinti elementą **x** iš aibės **S**;
- *join* (**S1**, **S2**, **S3**) - sujungti dvi aibes **S1** ir **S2** į vieną **S3**;
- *find* (**x**) - rasti aibę, kurioje yra elementas **x**;
- *disjoin* (**x**, **S**) - aibę **S** išskaidyti į dvi aibes, iš kurių vienoje yra visi elementai, mažesni ar lygūs **x**, o kitoje - didesni už **x** (aibę **S** tiesiškai sutvarkyta);
- *min* (**S**) - rasti minimalų aibės **S** elementą.

Operacijoms su aibėmis taip pat priklauso aibių sąjunga, aibių sankirta, aibių skirtumas ir aibių simetrinis skirtumas (operacijos, gerai žinomos matematikoje). Jos nagrinėjamos rečiau. Dvi aibės yra **atskirtos** (*disjoined*), jei jos neturi bendrų elementų. Daugelį praktikoje aptinkamų uždavinių galima suformuluoti taip, kad jie būtų sprendžiami vykdant vienokių ar kitokių operacijų su aibėmis sekas. Ypač tai pasakytina apie atskirtas aibes. Įvairiose operacijose su jomis kiekvienai iš atskirtų aibių gali atstovauti vienas elementas, pvz., minimalus (jei aibė sutvarkyta), o tai dažnai būna paranku kompiuteriniam algoritmui, nes mažina reikalingos atminties kiekį ir operacijų atlikimo sudėtingumą.

**Aibių** elementai kompiuterio atmintyje dėstomi įvairiais būdais. Vienas iš jų - kiekvienai aibei išskirti po tiesinį sąrašą. Kiti galimi būdai - dėstyti masyvuose ar medžiuose. Kiekvienas iš jų turi savų privalumų ir trūkumų. Toliau skyrelyje *heapsort* algoritmas bus pateiktas ir realizuotas, naudojant **aibės** duomenų struktūros operacijas.

#### 2.4.6. Aibinis *heapsort* algoritmas

Turime aibę  $S$ , kurios elementus reikia sutvarkyti didėjančia tvarka. Aibės  $S$  elementams rašyti kompiuterio atmintyje naudosime tiesinį sąrašą  $L$ . Tada *heapsort* algoritmas gali būti suformuluotas taip:

```
for x on list L do
    insert (x,S);
while not empty(S) do
begin
    y:=min (S);
    writeln(y);
    delete (y,S)
end
```

Kad būtų patogiau manipuluoti tiesinio sąrašo elementais, tarkime, kad jis išdėstytas masyve  $A$ . Tada *heap* savybės tikrinamos taip:

```
procedure pushdown (first, last: integer);
var r: integer;
begin
    r:=first;
    while r<last div 2 do
        if r=last div 2 then begin
            if A[r].key>A[2*r].key then
                swap(A[r],A[2*r]); r:=last
            end else
            if A[r].key>A[2*r].key and A[2*r].key<=A[2*r+1].key
            then begin
                swap(A[r],A[2*r]); r:=2*r
            end else
            if A[r].key>A[2*r+1].key and A[2*r+1].key<A[2*r].key
            then begin
                swap(A[r],A[2*r+1]); r:=2*r+1
            end else r:=last
        end
    end;
```

Pilna *heapsort* programa būtų tokia:

```
procedure heapsort;
var i: integer;
begin
    for i:=n div 2 downto 1 do
        pushdown(i,n);
    for i:=n downto 2
    begin swap(A[1],A[i]);
        pushdown(1,i-1)
    end
end;
```

## 2.5 Šalajos rūšiavimas (*mergesort*)

Labai dažnai apdorojant duomenis reikia į gan didelį surūšiuotą failą įterpti tam tikrą kiekį naujų duomenų, po to jį vėl surūšiuoti. Galima elgtis dvejopai: iš pradžių duomenis įterpti į failą,

o po to rūšiuoti, arba atvirkščiai, iš pradžių duomenis surūšiuoti, o po to abu failus sulieti. Būtent pastarąjį metodą ir nagrinėsime.

Sąlajos (*merge*) procesas sujungia dvi arba kelias duomenų aibes į vieną ir tam tikra prasme yra priešingas išrinkimo (*selection*) operacijai. Šiame skyriuje nagrinėsime algoritmus, kai liejami du failai. Jei jie jau surūšiuoti, nesunku sugalvoti procedūrą, kuri per tiesinį laiką, t.y. po vieną kartą nuskaitydama kiekvieną abiejų failų elementą, sukurtų naują surūšiuotą failą:

```
procedure merge (a, b);
var a,b,c: array [1..M + N] of integer;
begin
  i:=1; j:=1;
  a[M+1]:=maxint; b[N+1]:=maxint;
  for k:=1 to M+N do
    if a[i]<b[j]
      then begin c[k]:=a[i]; i:=i+1 end
      else begin c[k]:=b[j]; j:=j+1 end;
  end;
```

Ši programa yra maksimaliai suprastinta, joje netgi panaudoti signaliniai žymenys (kintamasis **maxint**), kurių reikšmės didesnės už bet kokias galimas masyvų **a** ir **b** elementų reikšmes. Be to, akivaizdu, kad masyvas **c** sujungia masyvus **a** ir **b** ir jam reikia papildomos atminties. Ši programa naudoja M+N lyginimų. Ją galima tobulinti, stengiantis daugiausia sumažinti naudojamos atminties tūrį, taikant įterpimo metodą duomenų įvedimo metu ir pan. Tačiau tokie patobulinimai kelia daug sąlygų realizacijai ir ją komplikuoja.

*Merge* procedūra tinkama manipuliacijoms su tiesiniais sąrašais. Papildoma atmintis tokiu atveju naudojama tik vienai nuorodai rašyti, o programa atrodo taip:

```
type link=^node;
node=record key: integer; next: link end;
var t,z:link; N: integer;
function merge(a,b:link):link;
var c:link;
begin c:=z;
  repeat
    if a^.key=<b^.key
      then begin c^.next:=a; c:=a; a:=a^.next end
      else begin c^.next:=b; c:=b; b:=b^.next end
    until c^.key=maxint;
  merge:=z^.next; z^.next:=z;
end;
```

Sąlajos procedūrą galima taikyti rūšiavimui, sukurti keli rūšiavimo algoritmai. Vienas iš jų paprastas pateiktas šiame skyrelyje. Idėja yra paprasta: duomenis skaidome į dvi dalis, kiekvieną iš jų rūšiuojame, po to suliejame. Tam, kad surūšiuoti kiekvieną iš suskaidytų dalių, rekursyviai taikome joms tą pačią procedūrą. Pateikiamoje programoje masyvas a[1..r] skirtas išeities duomenims saugoti, o masyvas b[1..r] yra pagalbinis:

```
procedure mergesort(l,r: integer);
var i,j,k,m: integer;
begin if r-l>0 then
  begin m:=(r+1) div 2;
    mergesort(l,m); mergesort(m+1,r);
```

```

for i:=m downto 1 do b[i]:=a[i];
for j:=m+1 to r do b[r+m+1-j]:=a[j];
for k:=1 to r do if b[i]<b[j]
then begin a[k]:=b[i]; i:=i+1 end
else begin a[k]:=b[j]; j:=j-1 end;
end;
end;

```

Sąlajos algoritmas, realizuotas naudojant tiesinį sąrašą, rūšiuoja duomenis nekeldamas juos iš vienos vietos į kitą ar keisdamas vietomis, bet keisdamas nuorodų reikšmes. Be to kiekvienas sąrašas, atstovaujantis duomenų dalims, kurios formuojamos programoje, baigiasi nuoroda į tą patį unifikotą elementą. Programa pateikiama kaip funkcija, kurios argumentas yra nuoroda į nesurūšiuotą failą, o reikšmė yra nuoroda į surūšiuotą failą:

```

function mergesort(c:link):link;
var a,b:link;
begin if c^.next=z then mergesort:=c else
  begin
    a:=c; b:=c^.next; b:=b^.next; b:=b^.next;
    while b<>z do begin c:=c^.next; b:=b^.next; b:=b^.next end;
    b:=c^.next; c^.next:=z;
    mergesort:=merge(mergesort(a),mergesort(b));
  end;
end;

```

Kaip ir bet kurio kito, šio algoritmo rekursyvią realizaciją galime pakeisti nerekursyvia. Sąlajos rūšiavimo atveju tam reikia keisti duomenų grupavimo tvarką ir naudoti vadinamąjį iš apačios į viršų (*bottom-up*) metodą: imami atskiri elementai ir interpretuojami kaip vienetinio ilgio duomenų rinkiniai, po to jie suliejami, gaunant surūšiuotus 2 ilgio rinkinius, po to šie rinkiniai suliejami, gaunant 4 ilgio duomenų rinkinius, ir t.t.:

```

function mergesort(c:link):link;
var a,b,head,todo,t:link;
i,N: integer;
begin
  N:=1; new(head); head^.next:=c;
repeat
  todo:=head^.next; c:=head;
  repeat
    t:=todo; a:=t;
    for i:=1 to N-1 do t:=t^.next;
    b:=t^.next; t^.next:=z; t:=b;
    for i:=1 to N-1 do t:=t^.next;
    todo:=t^.next; t^.next:=z;
    c^.next:=merge(a,b);
    for i:=1 to N+N do c:=c^.next
  until todo=z;
  N:=N+N;
until a=head^.next;
  mergesort:=head^.next
end;

```



Sąlajos rūšiavimo algoritmo sudėtingumas ir savybės yra tokios:

- nepriklausomai nuo duomenų, naudoja apytikriai  $N \log N$  lyginimų;
- naudoja papildomą atmintį, kurios tūris proporcingas duomenų kiekiui  $N$ ;
- algoritmas yra stabilus;
- algoritmo darbas nepriklauso nuo duomenų tvarkos;
- darbą galima pagerinti, kombinuojant jį su kitais rūšiavimo metodais.

## 2.6. Išorinis rūšiavimas (*external sorting*)

Apdorojant informaciją labai dažnai pasitaiko atveju, kai disponuojama labai dideliais duomenų failais ar rinkiniais, kurie yra rūšiuojami ir netelpa kompiuterio vidinėje atmintyje. Tuo atveju anksčiau nagrinėti algoritmai netinka, vietoj jų yra naudojami vadinamieji išoriniai rūšiavimo metodai. Du pagrindiniai faktoriai iš esmės skiria vidinio ir išorinio rūšiavimo algoritmus:

- kadangi dauguma duomenų saugomi antrinėje (diskinėje) ar netgi tretinėje atmintyje, elemento išrinkimas iš išorinės atminties trunka kur kas ilgiau, negu tūkstančiai lyginimo ar keitimo vietomis operacijų ar kitokių skaičiavimų vidinėje atmintyje;
- duomenų failai, saugomi išorinėje atmintyje, turi savus duomenų išrinkimo ar paieškos metodus, kurių negalima atsisakyti ar pakeisti (pvz., magnetinėje juostoje duomenis galima išrinkti tik nuosekliai).

Todėl išorinio rūšiavimo atveju, kuriant ar taikant vienokį ar kitokį algoritmą, reikia atsižvelgti į visus duomenų apdorojimo veiksmus ir etapus. Pilnos duomenų apdorojimo sistemos sąvoka čia lygiai tokia pat svarbi kaip ir algoritmo.

Dauguma išorinio rūšiavimo metodų naudoja tokią strategiją: jie peržiūri failą ir suskaido jį į blokus, tinkamus rašyti vidinėje atmintyje; kiekvieną iš šių blokų surūšiuoja; surūšiuotus blokus sulieja į vis didesnius, procesas tęsiamas tol, kol gaunamas vienas surūšiuotas failas. Šitokia strategija verčia programą daug kartų kreiptis į išorinį įrenginį, skaitant failą ar rašant į jį. Todėl norint parinkti efektyvų išorinio rūšiavimo algoritmą, reikia kreipti dėmesį ne tik į lyginimo ar keitimo vietomis operacijų skaičių, kaip buvę daroma anksčiau, bet ir mažinti įvedimo ar išvedimo operacijų skaičių ir jų vykdymo laiką.

Išorinio rūšiavimo metodai, sukurti, kai duomenys dar buvo saugomi perfokortose ir popierinėse perfojuostose, naudojami ir dabar rūšiuoti diskuose ir magnetinėse juostose, jie bus reikalingi ir ateityje, rūšiuojant duomenis domeninėje atmintyje (*bubble-memory*) ir videodiskuose. Veržlus technologijų vystymasis kelia jiems daug reikalavimų, verčia juos adaptuoti vis sudėtingesnei kompiuterinei aplinkai. Pvz., rūšiavimo metodai dabartinėse *multimedia* sistemose kompiuterinę atmintį skirsto į pirminę, antrinę, tretinę, iš kurių kiekviena turi didelės įtakos rūšiavimo greičiui ir algoritmams. Iš tikrųjų situacija yra dar sudėtingesnė. Šiuolaikiniuose kompiuteriuose vidinė atmintis nėra vienuarūšė, jos sudėtinės ar ją aptarnaujančios dalys yra ir superoperatyvioji atmintis (*cache memory*), ir buferinė atmintis (*buffer memory*). Duomenų išrinkimo greičiai šiose dalyse irgi yra skirtingi. Aišku, kad rūšiavimo metodai galingsiose kompiuterinėse sistemose turi gerai išnaudoti ir šiuos skirtumus.

Išorinio rūšiavimo algoritmai ne tik rūšiuoja duomenis failuose, bet ir juos sulieja, todėl dar jie vadinami rūšiavimo-sąlajos algoritmais (*sort-merge*).

### 2.6.1. Subalansuota daugybinė sąlaja (*balanced multiway merging*)

Tarkime, reikia surūšiuoti pakankamai didelio failo įrašus, o vidinėje atmintyje telpa tik trys įrašai. Sakykime, kad turime neribotą kiekį magnetinių juostų (nuoseklus išrinkimo įrenginių), ir eiliniam suliejimui naudosime bet kurias tris iš jų. Tada pirmiausia iš pirminio failo skaitome

nuosekliai po tris įrašus, juos rūšiuojame ir blokus po tris įrašus pakaitomis ir nuosekliai rašome į tris skirtingas juostas. Toliau vykdomos suliejimo procedūros. Po vieną įrašą iš kiekvienos juostos skaitoma į atmintį ir mažiausias iš jų rašomas į naują juostą. Vėl kreipiamasi į juostą, kurioje buvo mažiausias įrašas, ir iš jos skaitomas naujas įrašas, vėl mažiausias iš jų rašomas į juostą. Taip tęsiama tol, kol nepasibaigs blokas juostoje, iš kurios skaitoma, po to ta juosta ignoruojama, o iš likusių dviejų juostų skaitomi ir suliejami likusieji įrašai. Taip naujoje juostoje suformuojamas blokas iš devynių elementų. Jei dar blokų yra, ši procedūra gali būti tęsiama. Po to vėl tą pačią porcedūrą galima taikyti naujai suformuotoms trimis juostoms, kuriose išdėstyti devynių elementų ilgio blokai.

Ši aprašyta procedūra taikoma daugeliui gan efektyviai veikiančių rūšiavimo-sąlajos algoritmų, kurie subalansuotai naudoja nuoseklaus išrinkimo išorinius įrenginius. Balansuotumas reiškia tolygų išorinių atminties įrenginių darbo paskirstymą.

### 2.6.2. Pakeitimo išrinkimas (*replacement selection*)

Rūšiavimo-sąlajos algoritmams galima natūraliai ir efektyviai pritaikyti prioritetines eiles. Pirmiausia sąlajos metu, kai reikia išrinkti iš suliejamų elementų minimalų, tikslinga naudoti *heap* struktūrą ir jos operaciją *replace*, kuri pakeičia iš karto ankstesnio algoritmo dvi operacijas - minimalaus elemento rašymą į išorinę atmintį ir naujo elemento įterpimą. Taip sumažinamas atliekamų operacijų skaičius. Aišku, *heap* struktūrą reikia naudoti nuosekliai, t.y. jau pirminiame etape, kai elementai skirstomi į blokus ir rūšiuojami, tikslinga juos išdėstyti į *heap* struktūrą ir surūšiuoti tik iš dalies.

Be to (tai yra dar svarbiau), prioritetinių eilių struktūros panaudojimas įgalina sąlajos metu gauti ilgesnius surūšiuotus blokus, negu jie tilptų į vidinę atmintį. Kai išeities duomenys yra tvarkomi į *heap* struktūrą ir minimalus elementas yra keičiamas nauju, reikia papildomai naudoti tokią taisyklę: jei naujas elementas, rašomas vietoje minimalaus elemento senoje *heap* struktūroje, yra mažesnis už jį, reikia nuo šio elemento pradėti naują bloką ir naują *heap* struktūrą, interpretuojant jį esant didžiausiu šioje struktūroje. Algoritmai, naudojantys šias taisykles, yra vadinami pakeitimo išrinkimo vardu.

Algoritmai, sukurti naudojant išdėstytus ir panašius principus, turi tokias jų efektyvumą vertinančias savybes:

- rūšiavimo-sąlajos algoritmai, rūšiuojantys  $N$  įrašų, kai vidinė atmintis talpina  $M$  įrašų ir duomenys rašomi į  $(P+1)$ -ą išorinį įrenginį, perrinks šiuos duomenis  $1 + \log_P(N/2M)$  kartų;
- pakeitimo išrinkimo algoritmai, esant atsitiktiniams duomenims, formuoja duomenų blokus dvigubai ilgesnius negu naudojama *heap* struktūra.

## 3. Paieška

### 3.1. Elementarūs paieškos metodai

Paieška yra viena iš fundamentalių kompiuterių darbo operacijų. Jos esmė - dideliame informacijos rinkinyje rasti tam tikrą duomenų elementą arba patvirtinti, kad jo ten nėra. Remiantis kai kuriais vertinimais, paieškos operacijos užima iki 30% kompiuterio darbo laiko [1]. Duomenų vienetai, su kuriais tenka manipuliuoti šiose operacijose, dažniausiai atitinka įrašus, susidedančius savo ruožtu iš laukų. Tokiu atveju paieškos metu reikia rasti įrašus, kurių tam tikras laukas turi konkrečią reikšmę. Ši situacija kompiuterių taikymuose pasitaiko dažniausiai, todėl ji pagrindinai ir bus nagrinėjama. Įrašo laukas, kurio reikšmė naudojama paieškoje, vadinamas raktu (*key*).

Paieškos procese dažnai pasitaikančių duomenų struktūrų pavyzdžiai yra žodynai ir simbolių lentelės. Beveik kiekvienas tekstų redaktorius suteikia galimybę vartotojui patikslinti žodžių teisingą rašymą (*spelling*). Tam naudojami anglų ar kokios kitos kalbos žodynai. Tikslinant rašybą, kompiuteriui reikia rasti žodyne visus žodžius, besiskiriančius nuo duoto viena ar dviem raidėmis, nesvarbu, kurioje pozicijoje. Akivaizdu, kad šį norą lengva formuluoti, bet jam realizuoti reikia sudėtingo algoritmo. Simbolių lentelių formavimas ir paieška jose yra kitas labai panašus uždavinys, dažnai sutinkamas programavime. Simbolių lentelės yra pagrindiniai duomenys kompiliatoriams, programų bibliotekoms ir pan. Paieška jose vyksta lygiai taip pat, kaip ir žodynuose. Paieškos laukai yra kintamųjų ar kitų programų simbolių vardai, o įrašai - informacija, aprašanti vardo vietą programoje, kiti atributai.

Paieškos metu naudojamos tokios operacijos:

- *initialize* (išskirti vietą kompiuterio atmintyje);
- *search* (rasti įrašą, kuriame yra konkreti rakto reikšmė);
- *insert* (įterpti naują įrašą);
- *delete* (išmesti tam tikrą įrašą);
- *join* (sulieti du žodynus į vieną);
- *sort* (rūšiuoti įrašus arba žodyną).

Gana dažnai įvairiuose taikomuosiuose algoritmuose išvardytos operacijos naudojamos ne tik savarankiškai ar paskirai, bet ir jungiamos į grupes. Taip paaiškinama didelė paieškos procese dalyvaujančių duomenų struktūrų įvairovė bei jų specifika. Didelės įtakos paieškos algoritmams turi įrašai su pasikartojančiomis raktų reikšmėmis. Jie yra apdorojami kitokiais algoritmais, negu įrašai be pasikartojančių raktų reikšmių.

Pagrindinės algoritmų ir duomenų struktūrų, nagrinėjamų šiame skyriuje, operacijos - *initialize*, *search*, *insert*, ir *delete*. Kartais reikės operacijos *sort*. Kaip ir prioritetinių eilių atveju, operacijai *join* realizuoti reikia sudėtingesnės procedūros, kuri galėtų būti nagrinėjama kituose specialiuose kursuose.

Algoritmus, atliekančius paiešką įrašuose be pasikartojančių raktų reikšmių, galima pritaikyti paieškai ir įrašuose su pasikartojančiomis reikšmėmis. Tai galima padaryti keliais būdais. Vienas iš jų - paieškos operacijos vykdomos įrašams be pasikartojančių raktų reikšmių, o įrašai su ta pačia rakto reikšme rašomi į atskirą sąrašą su nuoroda į šį sąrašą iš tam tikro įrašo. Kitas būdas - visi įrašai rašomi kompiuterio atmintyje naudojant vieningą duomenų struktūrą, o paieškos metu randamas tik vienas įrašas iš kelių galimų. Šis būdas paprastesnis realizuoti, bet gali neatitikti kitų keliamų paieškai reikalavimų. Šį būdą tenka papildyti kitais algoritmais, kai reikia rasti visus įrašus su tam tikru raktu. Trečias būdas - kiekvienam įrašui priskirti unikalų identifikatorių (kodą) ir jį naudoti paieškos operacijų metu (vietoje paieškos rakto reikšmės).

#### 3.1.1. Nuosekli paieška

Paprasčiausias paieškos algoritmas yra nuosekli paieška - nuoseklus vieno po kito elementų perrinkimas. Tariant, kad elementai yra saugomi masyve, paiešką lengva apibrėžti šiomis operacijomis:

- operacija *insert* - elemento rašymas į laisvą vietą masyve;
- operacija *search* - nuoseklus masyvo elementų perrinkimas iki tol, kol bus sutikta reikiama elemento reikšmė;
- operacija *delete* - surasto elemento pašalinimas iš masyvo.

Paieškos algoritmą realizuojanti programa būtų tokia:

```

type node=record key,info: integer end;
var a: array [0..maxN] of node; N: integer;
procedure initialize;
  begin N:=0 end;
function seqsearch(v:integer; x:integer):integer;
begin
  a[N+1].key:=v;
  if x=<N then
    repeat x:=x+1 until v=a[x].key;
  seqsearch:=x
end;
function seqinsert(v:integer):integer;
begin
  N:=N+1; a[N].key:=v;
  seqinsert:=N;
end;

```

Vertinant algoritmo sudėtingumą, tikslinga žinoti lyginimų skaičių, nes tai pagrindinė algoritmo operacija. Jis reikalauja vidutiniškai  $(N + 1)/2$  lyginimų.

Nuoseklią paiešką galima realizuoti ir naudojant sąrašą. Šiuo atveju įrašai turi būti surūšiuoti, nes tai pagreitina algoritmo darbą.

Programa, realizuojanti nuoseklią paiešką, būtų tokia:

```

type link=^node;
  node=record key, info:integer; next :link end;
var head,t,z:link; i:integer;
procedure initialize;
begin
  new(z); z^.next :=z;
  new(head); head^.next :=z;
end;
function listsearch(v:integer; t :link):link;
begin
  z^.key:=v;
  repeat t :=t^.next until v<=t^.key;
  if v=t^.key then listsearch:=t
  else listsearch:=z
end;

```

Algoritmas pagreitėja todėl, kad paieškos atveju suradus raktą, didesnę už duotą, galima teigti, kad paieška buvo nesėkminga, ir baigti darbą. Tokiu atveju vidutiniškai reikės lyginti tik

pusę įrašų. Tačiau tokios prielaidos atveju reikia, kad operacijos *insert* ir *delete* irgi nekeistų surūšiuotų elementų tvarkos. Įterpimo operacijos programa šiuo atveju būtų tokia:

```
function listinsert(v:integer; t :link):link;
var x:link;
begin
    z^.key:=v;
    while t^.next^.key<v do t:=t^.next ;
    new(x); x^.next :=t^.next ; t^.next :=x; x^.key:=v;
    listinsert:=x;
end;
```

Nuosekli paieška naudos  $N/2$  lyginimų ir sėkmingu ir nesėkmingu atvejais.

Jei ieškomų raktų dažnumai žinomi, tai algoritmo darbą gali pagreitinti įrašų rūšiavimas jų dažnumų mažėjimo tvarka. Jei tokie dažnumai nežinomi, efektyvi priemonė yra patalpinti pirmą kartą surastą įrašą į sąrašo pradžią.

### 3.1.2. Dvejetainė paieška

Jei įrašų, kuriuose ieškoma, yra daug, nuoseklios paieškos laikas juose gali būti labai ilgas. Jis gali žymiai sutrumpėti, taikant paieškos algoritmams "skaldyk ir valdyk" paradigmą. Šiuo principu yra sudaromi dvejetainės paieškos algoritmai, kurie veikia panašiai kaip *quicksort*. Duomenys yra laikomi surūšiuotame faile. Paieškos metu jie yra skaidomi į dvi lygias dalis (kairę ir dešinę) ir išskiriamas ribinis elementas (mažiausias tarp dešinės dalies ir didžiausias tarp kairės dalies elementų). Ieškomas raktas lyginamas su ribiniu elementu, ir jei jis mažesnis už ribinį, paieška tęsiama kairiojoje dalyje, o jei didesnis - dešinėje dalyje. Programa, realizuojanti tokį algoritmą, atrodytų taip:

```
function binarysearch(v: integer): integer;
var x,l,r: integer;
begin
    l:=1; r:=N;
    repeat
        x:=(l+r) div 2;
        if v<a[x].key then r:=x-1 else l:=x+1
        until (v=a[x].key) or (l>r);
        if v=a[x].key then binarysearch:=x
        else binarysearch:=N+1
    end;
```

Panašiai kaip algoritmai *quicksort* ir *radix*, šis metodas naudoja rodykles l ir r, išskiriant analizuojamų duomenų dalį. Dvejetainė paieška reikalauja  $\lg N+1$  lyginimų sėkmingu ar nesėkmingu atvejais. Tačiau laikas, reikalingas įterpimo operacijai, kai programuojama naudojant masyvą, yra didelis. Elementus masyve reikia laikyti surūšiuotus, vadinasi, įterpiant naują elementą į tam tikrą vietą kitus reikia perkelti toliau - į masyvo pabaigą. Todėl dvejetainės paieškos algoritmą verta naudoti tada, kada iš pradžių galima beveik pilnai suformuoti failą (t.y. paieškos metu beveik nepasitaikys įterpimo operacijų), o po to naudoti tik paiešką (daug kartų).

Šį algoritmą sunkiau programuoti, kai yra pasikartojantys raktai, nes ribinis elementas gali atsidurti tarp kelių lygių. Tokiu atveju bus sunku spręsti, kurią dalį - kairę ar dešinę - toliau analizuoti, ypač kai reikia rasti visus įrašus su tuo pačiu raktu.

Dvejetainės paieškos algoritmą galima patobulinti, dalijant duomenis ne į dvi lygias dalis, o nustatant ribinį elementą pagal ieškomo rakto reikšmę. Tarkime, telefonų knygoje reikia rasti žodį,

prasidedantį raide H. Knygą nebūtina ir netikslinga versti ties viduriu, čia efektyvu yra įvesti interpoliacinę paiešką. Formaliai tai reiškia, kad intervalą tarp l ir r reikia dalyti į dvi dalis ne per pusę, bet interpoliacinės formulės

$$l + (v - a[l].key) * (r - l) \div (a[r].key - a[l].key)$$

rekomenduojamu santykiu. Čia rakto v reikšmė yra spėjama vieta tarp raktų a[l] ir a[r] reikšmių. Tokio modifikuoto algoritmo sudėtingumas bus apie  $\lg \lg N + 1$  lyginimų.

Tačiau šis metodas gali būti efektyviai taikomas tik kai raktų reikšmės yra pasiskirstę tolygiai. Be to, mažoms N reikšmėms funkcijos lg ir  $\lg \lg$  nedaug skiriasi. Kita vertus, interpoliaciją tikslinga naudoti, kai duomenų failai dideli, įrašų ar raktų lyginimai reikalauja daug laiko sąnaudų arba duomenys laikomi išorinėje atmintyje ir jų išrinkimas (t.y. perkėlimas į vidinę atmintį) yra labai ilgas.

### 3.1.3. Dvejetainio medžio paieška

Dvejetainio medžio paieškos algoritmas naudoja dvejetainės paieškos ir interpoliacinės paieškos principus, juos pritaikydamas visoms operacijoms (įterpimo, išmetimo, ir t.t.). Tai paprastas, efektyvus dinaminis metodas, jis yra vienas iš fundamentalių algoritmų informatikoje aibės. Duomenims dėstyti algoritmas naudoja dvejetainį medį, o įterpimo ar išmetimo operacijoms - tokią taisyklę: visos viršūnės, mažesnės už duotą, turi būti talpinamos kairiajame pomedyje, o visos didesnės ar lygios - dešiniajame. Paieškos operacija šiame algoritme - tai tik ėjimas tam tikra medžio šaka. Dvejetainė paieška skiriasi nuo dvejetainio medžio paieškos ne tik tuo, kaip dalijami elementai į dalis, bet ir tuo, kad pastaruoju atveju yra formuojama duomenų struktūra. Programa, realizuojanti algoritmą, būtų tokia:

```

type link = ^node;
node = record key, info: integer; l, r: link end;
var t, head, z: link;
function treearch(v: integer; x: link): link;
begin
    z^.key := v;
    repeat
    if v < x^.key then x := x^.l else x := x^.r
    until v = x^.key;
    treearch := x
end;

```

Algoritmą programuojant, verta apibrėžti kintamąjį *head*, kurio dešinė nuoroda būtų nukreipta į medžio šaknį, kairė nuoroda būtų *nil*, o reikšmė būtų mažesnė už visų kitų viršūnių reikšmes. Norint išsaugoti kintamųjų vienaarūšiškumą, visų kitų viršūnių nuorodas, kai jos laisvos, reikia nukreipti į vieną (*dummy*) viršūnę z. Kitas galimas atvejis - reikia išskirti du viršūnių (ir įrašų) tipus: vidines ir išorines viršūnes. Tuščio medžio inicializacijos programa būtų tokia:

```

procedure treeinitialize;
begin
    new(z); z^.l := z; z^.r := z;
    new(head); head^.key := 0; head^.r := z;
end;

```

Norint įterpti viršūnę į medį, reikia atlikti nesėkmingą paiešką ir naują viršūnę įterpti kaip sūnų prie tos viršūnės, kur paieška užsibaigė:

```

function treeinsert(v:integer; x: link): link;
var p:link;
begin
    repeat
        p:=x;
        if v<x^.key then x:=x^.l else x:=x^.r
        until x=z;
        new(x); x^.key:=v; x^.l:=z; x^.r:=z;
        if v<p^.key then p^.l:=x else p^.r:=x;
        treeinsert:=x
    end;

```

Norint įterpti naują įrašą tuo atveju, kai jau toks dvejetainiame paieškos medyje yra, reikia jį įterpti kaip dešinę sūnų tos viršūnės, kuri jam lygi.

Jei reikia surūšiuoti duomenis, belieka tik esamą medį atspausdinti, perenkant visas jo viršūnes iš kairės į dešinę, ignoruojant jas jungiančius ryšius.

```

procedure treeprint(x: link);
begin
    if x<>z then
        begin
            treeprint(x^.l);
            printnode(x);
            treeprint(x^.r)
        end
    end;

```

Šis rūšiavimo metodas labai panašus į *quicksort* algoritmą, skirtumas tik toks, kad pastarasis medžio rūšiavimas naudoja gan daug papildomos atminties ryšiams tarp viršūnių išsaugoti, tuo tarpu *quicksort* algoritmas tokiems ryšiams atminties naudoja labai mažai.

Algoritmo efektyvumas labai priklauso nuo dvejetainio medžio formos, teisingiau nuo jo subalansuotumo. Jei dvejetainis paieškos medis yra subalansuotas, algoritmas naudos  $2\ln N$  lyginimų. Blogiausiu atveju algoritmas naudos  $N$  lyginimų.

### 3.1.4. Operacija *delete*

Įterpimo, paieškos ir rūšiavimo operacijos dvejetainiame paieškos medyje yra realizuojamos gan paprastai ir tiesmukiškai. Tačiau elemento išmetimo (*delete*) operacija yra žymiai sudėtingesnė programuoti, ją verta nagrinėti kaip rekurentinės procedūros paieškos algoritmuose pavyzdį.

Pašalinti viršūnę iš medžio yra nesudėtinga, kai ji neturi sūnų arba turi tik vieną sūnų. Tačiau pašalinant viršūnę, turinčią du sūnus, reikia viršūnių vietas medyje pertvarkyti. Dvejetainiams paieškos medžiams šiuo atveju naudojamas toks principas - į išmetamos viršūnės vietą reikia įterpti viršūnę, kitą pagal didumą. Aišku, kad šis principas yra optimalus paieškai, bet jo realizavimas gali kartais priversti iš pagrindų pakeisti medžio struktūrą. Tokią situaciją verta iliustruoti pavyzdžiais:

```

procedure treedelete(t,x: link);
var p,c:link;
begin
    repeat

```

```

p:=x;
if t^.key<x^.key then x:=x^.l else x:=x^.r
until x=t;
if t^.r=z then x:=x^.l
else if t^.r^.l=z then
  begin x:=x^.r; x^.l:=t^.l end
else begin
  c:=x^.r; while c^.l^.l<>z do c:=c^.l;
  x:=c^.l; c^.l:=x^.r;
  x^.l:=t^.l; x^.r:=t^.r
end;
if t^.key<p^.key then p^.l:=x else p^.r:=x;
end;

```

Programuojant šį algoritmą, reikia perrinkti ne tik daug atvejų, bet ir naudoti įvairius programavimo triukus. Pavyzdžiui, tikslinga ieškoti kitos dar prieš išmetant tam tikrą viršūnę, o ne po to. Algoritmas taip, kaip čia pateiktas, atrodo asimetris - visada išrenkamas pirmiausia dešinys sūnus, nors kartais duomenys yra taip išsidėstę, kad tikslingiau būtų išrinkti kairįjį sūnų. Tokių programos patobulinimų gali būti daug.

### 3.1.5. Netiesioginiai dvejetainės paieškos medžiai

Dažniausiai reikia tik įrašų paieškos, o ne įterpimo, išmetimo ar kitų tiesioginio manipuliavimo su įrašais operacijų. Tokio taikymo pavyzdys - bibliotekinės informacinės sistemos. Jose įrašai būna suformuoti iš anksto arba tai daroma išskirtiniu laiko momentu, o informacinių sistemų funkcionavimo metu vykdomos tik paieškos operacijos. Norint pagreitinti tokios sistemos darbą, kartais reikia sukurti papildomas rodykles, kurios padėtų lengvai rasti įrašo vietą dokumentų rinkinyje. Taigi, reikia sukurti indeksą, nurodantį masyvo elementus.

Dvejetainės paieškos medžius taikyti tokiai situacijai galima keliais būdais. Vienas iš jų - viršūnių informacinę dalį naudoti ne raktų reikšmėms saugoti, o įrašo vietų masyve nuorodoms. Antras būdas - viršūnių informacinėse dalyse sudaryti raktų reikšmių kopijas. Trečias - raktų reikšmės kopijuoti į atskirą masyvą, o viršūnėse saugoti šio naujo masyvo elementų nuorodas.

Taikant visus šiuos ar kitus netiesioginio paieškos medžio realizavimo būdus, turimus ryšius, t.y. nuorodas, galima realizuoti naudojant rodyklės tipą arba masyvą. Jei ryšiai realizuojami naudojant masyvą, bet kuri nuoroda yra rašoma masyvo elementuose. Tada nėra būtinybės kurti naujus įrašus (procedūra *new*). Šitoks būdas naudojamas daugelyje taikomųjų programų, ypač kai dirbama su dideliais duomenų kiekiais (tačiau jų skaičius tačiau iš anksto gali būti įvertintas).

Dvejetainių paieškos medžių į sūnus nuorodoms realizuoti galima naudoti ir kelis lygiagrečius masyvus, vienas iš jų gali rodyti į kairiuosius sūnus, kitas į dešiniuosius, ir t.t. Dar kiti masyvai gali būti naudojami atskiroms medžių šakoms ar -pan. išskirti. Tokios programos dirba labai greitai, yra lanksčios, jas nesudėtinga papildyti, nors jos ir netaupo atminties.

### 3.2 Subalansuoti medžiai.

Algoritmai, naudojantys dvejetainius medžius, gali dirbti labai greitai, bet gali ir labai lėtai, jų efektyvumas labai svyruoja (dažniausiai priklausomai nuo duomenų). Ypač jų efektyvumas krinta blogiausiu atveju. Nepalankūs atvejai yra, kai duomenys pilnai surūšiuoti, arba išdėstyti atvirkštine tvarka, arba paeiliui didelis keičia mažą, arba išdėstyti vienas po kito kokia nors kitokia reguliaria tvarka. Šitokia ypatybė buvo aptikta ir anksčiau, kituose algoritmuose. Pavyzdžiui, *quicksort* algoritmo atveju prieš rūšiuojant duomenis rekomenduotina atsitiktinai sumaišyti arba

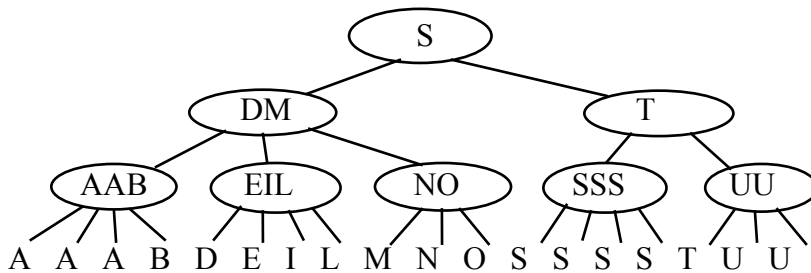


bent jau atsitiktiniu būdu rinkti dalijantį elementą. Tokios rekomendacijos nors ir mažina blogiausio atvejo tikimybę, tačiau turi tik statistinį pobūdį ir negarantuoja, kad jo galima išvengti.

Dvejetainių medžių atveju yra geriau, jiems galima apibrėžti reguliarias struktūras, valdančias algoritmų efektyvumą ir įgalinančias iš viso išvengti blogiausių atvejų. Pagrindinės ir labiausiai paplitusios tokios struktūros yra subalansuoti medžiai. Jie remiasi vadinamaisiais balansavimo metodais - paradigma, panašia į “skaldyk ir valdyk”. Tačiau balansavimo metodai, gerinantys algoritmų efektyvumą, turi vieną bendrą blogą savybę - juos lengviau formuluoti, negu realizuoti.

### 3.2.1. 2-3-4 ir 2-3-medžiai

Blogiausias dvejetainių medžių algoritmų atvejis yra, kai dauguma viršūnių turi tik po vieną sūnų, ir todėl struktūra yra beveik “tiesinė”. Balansavimo principas draudžia tokią būseną, t.y. kiekviena viršūnė turi turėti daugiau negu vieną sūnų. Kita vertus, struktūra turi būti pakankamai lanksti, kad leistų aprašyti daugelį realių situacijų. Todėl subalansuotiems medžiams leidžiama turėti ne tik 2, bet ir daugiau sūnų. 2-3-4-medžiuose kiekvienai viršūnei leidžiama turėti ne mažiau kaip 2 ir ne daugiau kaip 4 sūnus. Tokio medžio viršūnės vadinamos 2-viršūne (atitinkamai 3-viršūne, 4-viršūne), jei ji turi 2 sūnus (atitinkamai 3 sūnus, 4 sūnus). Be to, kiekvienai viršūnei yra priskiriami įrašai: 2-viršūnei - vienas įrašas; 3-viršūnei - du įrašai; 4-viršūnei - trys įrašai. Jų reikšmės skirtos pomedžių minimalioms reikšmėms saugoti. Vieninteliam 2-viršūnės įraše bus saugoma minimali reikšmė, kurią galima rasti dešinio sūnaus pomedyje. Viename 3-viršūnės įraše bus saugoma minimali reikšmė, kurią galima rasti vidurinio sūnaus pomedyje, o kitame įraše - minimali reikšmė, kurią galima rasti dešinio sūnaus pomedyje. Viename 4-viršūnės įraše bus saugoma minimali reikšmė, kurią galima rasti antro sūnaus pomedyje, kitame įraše bus saugoma minimali reikšmė, kurią galima rasti trečio sūnaus pomedyje, o trečiame įraše - minimali reikšmė, kurią galima rasti dešinio sūnaus pomedyje. Taip suformuoti įrašai viršūnėje leidžia automatiškai nustatyti minimalų ir maksimalų elementus, esančius kiekviename pomedyje, ir tokiu būdu paieškos operaciją atlikti per logaritminį laiką. Kad paieška būtų iš tikrųjų vienerūšė, apibrėžiant subalansuotą medį dar reikalaujama, kad visos šakos būtų vienodo ilgio. Paieška pradedama nuo medžio šaknies, ir toliau tęsiama tame pomedyje, kurio reikšmių intervale yra ieškoma rakto reikšmė. Ji visada užbaigiama šakos lape. 2-3-4-medžio pavyzdys parodytas 11 pav.



11 pav. Sakinio “SUBALANSUOTAS MEDIS” 2-3-4-medis

Įterpimo procedūra 2-3-4-medyje vykdoma taip: atliekame nesėkmingą paiešką ir nustatome vietą, kur naują viršūnę turi būti įterpta. Jei tai 2- ar 3-viršūnė, tai prijungdami naują sūnų ją pertvarkome į 3- ar 4-viršūnę. Jei tai 4-viršūnė, ją skaidome į dvi 2-viršūnes ir prie vienos iš jų prijungiame naują. Jei viršūnė, kurios sūnus buvo pastaroji 4-viršūnė, yra 2- arba 3-viršūnė, įterpimo procedūra baigta. Jei ši viršūnė savo ruožtu buvo 4-viršūnė, ją irgi reikia skaidyti į dvi viršūnes. Šis procesas kartojamas tol, kol naujai pertvarkoma viršūnė yra 4-viršūnė.

Įterpimo operacijos metu 4-viršūnės yra skaidomos į dvi viršūnes. Jei skaidoma taip, kaip ką tik aprašyta, medžių pertvarkymas vyksta iš apačios į viršų, ir jie vadinami *bottom-up* 2-3-4-

medžiais. Tačiau 4-viršūnės galima skaidyti paieškos metu, kai tik jos aptinkamos einant medžio šakomis iš viršaus į apačią. Tada medžiai vadinami *top-down* 2-3-4-medžiais.

**Teiginys:** Paieškos metu 2-3-4-medyje, turinčiame  $N$  viršūnių, visada aptinkama ne daugiau  $\lg N + 1$  viršūnių. Įterpimo metu tokiam medyje yra skaidoma ne daugiau kaip  $\lg N + 1$  viršūnė blogiausiu atveju, ir tikėtina (tą teigia statistiniai testai), kad skaidoma mažiau negu 1 viršūnė vidutiniškai.

Panašiai 2-3-4-medžiuose galima apibrėžti ir išmetimo operaciją, tik vietoje viršūnės skaidymo į dvi, čia dvi viršūnės iš kurių viena turi tik vieną sūnų, bus jungiamos į vieną. Šios operacijos sudėtingumą vertins toks pat analogiškas teiginys.

Nors algoritmai 2-3-4-medžiams yra pilnai apibrėžti ir teoriniai jų vertinimai geresni negu dvejetainių medžių, bet jų programavimas yra sudėtingas, o programos, realizuotos tiesiogiai perrašant algoritmus, gali būti labai lėtos, netgi lėtesnės negu dvejetainiams medžiams. Todėl taikomiosiose programose "tikrų" 2-3-4-medžių pasitaiko ne taip dažnai. Vietoj jų naudojami arba 2-3-medžiai, arba raudoni-juodi medžiai.

2-3-medžiai turi tokias savybes:

- kiekviena vidinė viršūnė turi du arba tris sūnus;
- kiekviena šaka nuo šaknies iki lapo yra vienodo ilgio (arba gali skirtis tik vienu lygmeniu).

Įrašai 2-3-medyje, kaip ir 2-3-4-medyje, yra surūšiuoti ir talpinami lapuose. Lygiai taip, kaip ir anksčiau, apibrėžiamos 2-viršūnės ir 3-viršūnės, ir joms priskiriami įrašai. 2-3-medis, kurio šakos yra ilgio  $k$ , turi ne mažiau kaip  $2^{k-1}$  ir ne daugiau kaip  $3^{k-1}$  lapų. Toks medis iš  $N$  elementų turės ne mažiau kaip  $1 + \log_3 N$  lygių ir ne daugiau kaip  $1 + \log_2 N$  lygių.

### 3.2.2. Įterpimo operacija 2-3-medžiuose

Šiame skyrelyje pateikiamos programos yra daugiau aprašomojo pobūdžio, bet jos gerai iliustruoja manipuliavimo su medžio viršūnėmis operacijas. Algoritmas, įterpiantis naują sūnų vienai viršūnei, būtų toks:

```
procedure insert1 (node:^twothreeNode);
x:elementType; { x is to be inserted into the subtree of node }
var pnew:^twothreeNode; { pointer to new node created to right of node }
var low:real; { smallest element in the subtree pointed to by pnew }
begin
  pnew:=nil;
  if node is a leaf then begin
    if x is not the element at node then begin
      create new node pointed to by pnew;
      put x at the new node;
      low:=x.key
    end end
  else begin { node is an interior node }
    let w be the child of node to whose subtree x belongs;
    insert1 (w,x,pback,lowback);
    if pback<>nil then begin
      insert pointer pback among the children of node just
      to the right of w;
      if node has four children then begin
        create new node pointed to by pnew;
        give the new node the third and the fourth children of node;
        adjust lowofsecond and lowofthird in node and the new node;
```

```

        set low to be the lowest key among the children of the new node
    end end
end
end; { insert1 }

```

Ta pati programa, perrašyta vietoje komentarų įterpiant Paskalio operatorius, atrodytų taip:

```

procedure insert1 (node:^twothreenode);
x: elementtype;
var pnew:^twothreenode; var low:real);
var pback:^twothreenode; lowback:real;
child:1..3; { indicates which child of node is followed in recursive call }
w:^twothreenode; { pointer to the child }
begin
    pnew:=nil;
    if node^.kind=leaf then begin
        if node^.element.key<>x then begin
            new(pnew,leaf);
            pnew^.element:=x;
            low:=x.key
        end end
        else begin { node is an interior node }
            if x.key<node^.lowofsecond then begin
                child:=1; w:=node^.firstchild end
            else if (node^.thirdchild=nil) or (x.key<node^.lowofthird) then begin
                child:=2; w:=node^.secondchild end
            else begin { x is in third subtree }
                child:=3; w:=node^.thirdchild end;
            insert1 (w,x,pback,lowback);
            if pback <> nil then
                if node^.thirdchild=nil then begin
                    if child=2 then begin
                        node^.thirdchild:=pback;
                        node^.lowofthird:=lowback
                    end
                    else begin { child=1 }
                        node^.thirdchild:=node^.secondchild;
                        node^.lowofthird:=node^.lowofsecond;
                        node^.secondchild:=pback;
                        node^.lowofsecond:=lowback
                    end
                end
            else begin { node already had three children }
                new (pnew,interior);
                if child=3 then begin
                    pnew^.firstchild:=node^.thirdchild;
                    pnew^.secondchild:=pback;
                    pnew^.thirdchild:=nil;
                    pnew^.lowofsecond:=lowback;
                    low:=node^.lowofthird;
                    node^.thirdchild:=nil
                end
                else begin { child=<2; move third child of node to pnew }

```

```

    pnew^.secondchild:=node^.thirdchild;
    pnew^.lowofsecond:=node^.lowofthird;
    pnew^.thirdchild:=nil;
    node^.thirdchild:=nil end
if child=2 then begin
    pnew^.firstchild:=pback;
    low:=lowback end
if child=1 then begin
    pnew^.firstchild:=node^.secondchild;
    low:=node^.lowofsecond;
    node^.secondchild:=pback;
    node^.lowofsecond:=lowback end
end
end
end; { insert1 }

```

Visa naujo elemento įterpimo į 2-3-medį programa atrodytų taip:

```

procedure insert (x:elementtype; var S: set);
var pback:^twothreenode; { pointer to new node returned by insert1 }
    lowback:real; { low value in subtree of pback }
    saveS: set; { place to store a temporary copy of the pointer S }
begin
    { checks for S being empty or a single node should occur here, and an appropriate insertion
    procedure should be included }
    insert1 (S,x,pback,lowback);
    if pback<> nil then begin
    { create new root; its children are now pointed to by S and pback }
    saveS:=S; new(S);
    S^.firstchild:=saveS;
    S^.secondchild:=pback;
    S^.lowofsecond:=lowback;
    S^.thirdchild:=nil;
    end
end; { insert }

```

### 3.2.3. Išmetimo operacija 2-3-medžiuose

Neformaliai užrašyta programa, realizuojanti elemento išmetimą iš vienos 2-3-medžio viršūnės, būtų tokia:

```

function delete1 (node:^twothreenode; x:elementtype):boolean;
var onlyone:boolean; { to hold the value returned by a call to delete1 }
begin
    delete:=false;
    if the children of node are leaves then begin
    if x is among those leaves then begin
    remove x;
    shift children of node to the right of x one position left;
    if node now has one child then
    delete1:=true end
    end
end

```

```

else begin { node is at level two or higher }
    determine which child of node could have x as a descendant;
    onlyone:=delete1 (w,x); { w stands for node^.firstchild,
    node^.secondchild, or node^.thirdchild, as appropriate }
if onlyone then begin { fix children of node }
    if w is the first child of node then
    if y, the second child of node, has three children then
        make the first child of y be the second child of w
    else begin { y has two children }
        make the child of w be the first child of y;
        remove w from among the children of node;
        if node now has one child then
            delete1:=true end
        if w is the second child of node then
        if y is the first child of node, has three children then
            make the third child of y be the first child of w
        else { y has two children }
        if z, the third child of node, exists and has three children then
            make first of z be the second child of w
        else begin { no other child of node has three children }
            make the child of w be the third child of y;
            remove w from among the children of node;
            if node now has one child then
                delete1:=true end;
            if w is the third child of node then
            if y, the second child of node, has three children then
                make thr third child of y be the second child of w
        else begin { y has two children }
            make the child of w be the third child of y;
            remove w from among the children of node
        end { note node surely has two children left in this case }
    end
end
end; { delete1 }

```

### 3.2.4. Duomenų struktūros 2-3-medžiams

Realizuojant algoritmus su medžiais, išskiriamos dviejų tipų medžio viršūnės - vidinės ir išorinės. Šis išskyrimas yra esminis, nes tokias viršūnes suvienodinus gaunami visiškai kiti algoritmai. Jie bus nagrinėjami vėliau.

Vidinėms ir išorinėms viršūnėms realizuoti reikia naudoti skirtingas duomenų struktūras. Paskalio kalboje tai galima daryti naudojant **case** operatorių. Viršūnę galima apibrėžti taip:

```

type elementtype = record
    key: real;
    {other fields as warranted}
end;
nodetypes = (leaf, interior);
twothreenode = record
    case kind: nodetypes of leaf: (element: elementtype);
    interior: (firstchild, secondchild, thirdchild: ^twothreenode;
        lowofsecond, lowofthird: real)

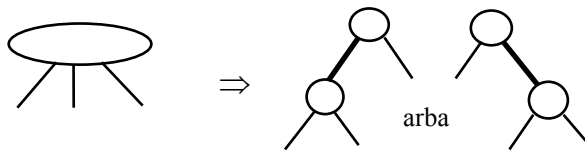
```

**end;**  
 set = ^twothreenode;

Lygiai tokius pat programavimo principus galime taikyti ir 2-3-4-medžiams. Vidinės viršūnės gali turėti nuorodas į 2 ar 3 ar 4 sūnus. Išorinės viršūnės yra elementaraus tipo ir nuorodų neturi. Aišku, programos, realizuojančios paiešką 2-3-4-medyje, bus dar sudėtingesnės, bet manipuliavimo su jų viršūnėmis principai lieka tie patys.

### 3.2.5. Raudoni - juodi medžiai

Be 2-3-medžių, yra ir kitų būdų palengvinti algoritmų, besiremiančių 2-3-4-medžiais, realizavimą. Vienas iš jų - naudoti raudonus -juodus medžius. Iš esmės tai tie patys aukščiau aprašyti dvejetainiai medžiai, tik kiekvienai jų viršūnei išskiriamas papildomas bitas nuorodos iš viršūnės į jos sūnus spalvai, kuri gali būti raudona ar juoda, saugoti. Pagrindinė mintis - vaizduoti 2-3-4-medžio 3-viršūnės ir 4-viršūnės kaip mažus dvejetainius medžius, sujungtus raudonomis briaunomis, tuo tarpu paties 2-3-4-medžio briaunos yra vaizduojamos juoda spalva. 4-viršūnę atitiks trys dvejetainės viršūnės, sujungtos raudonomis jungtimis, o 3-viršūnę atitiks dvi dvejetainės viršūnės, irgi sujungtos raudonai. Jų galima išraiška parodyta 12 pav. Jame raudona spalva pažymėta storesne linija. Programuojant kiekvienai viršūnei išskiriamas papildomas bitas, rodantis jungties spalvą.



12 pav. 3-viršūnės išraiška raudonu-juodu medžiu.

3-viršūnė gali būti išreikšta nevienareikšmiškai, vaizduojant raudoną nuorodą į sūnų, nukreiptą kairėn ar dešinėn. Todėl tą patį 2-3-4-medį galima išreikšti skirtingais raudonais-juodais medžiais. Jų žymėjimo suvienodinimas priklauso nuo konkrečių algoritmo sąlygų. Raudoniems-juodiems medžiams galima formuluoti tam tikras jų struktūrines savybes:

- raudonas-juodas medis neturi dviejų greta esančių vienoje šakoje raudonų jungčių;
- visos šakos turi vienodą kiekį juodų jungčių;
- skirtingų šakų ilgiai (skaičiuojant ir raudonas jungtis) gali skirtis ne daugiau kaip dvigubai (bet visų jų ilgis proporcingas  $\log N$ ).

Aišku, kad raudonų-juodų medžių programavimo sudėtingumas yra toks pat kaip dvejetainių, ir daug lengvesnis negu 2-3-4-medžių. Raudoni-juodi medžiai turi ir kitų gerų savybių. Pavyzdžiui., juose nesunku dėstyti ir programuoti pasikartojančių elementų reikšmes. Aišku, kad bet kuris subalansuoto medžio algoritmas turi leisti pasikartojančioms reikšmėms "kristi" į abi puses nuo viršūnės, kurios reikšmė irgi tokia pati. Šiuo atveju paieškos procedūrą reikia nežymiai modifikuoti, pritaikant ją apeiti visą nagrinėjamą pomėdį, kaip tai buvo padaryta anksčiau procedūroje *treeprint*.

Kita gera šių medžių savybė yra ta, kad paieškos paprastame dvejetainiame medyje procedūros gali būti įdiegtos, ir beveik be modifikacijų. Jungčių spalvą galima realizuoti, įvedant loginį kintamąjį, kuris bus "true", kai jungtis raudona, ir "false", kai jungtis juoda. Be to, paieškos metu į jungties spalvą galima nekreipti dėmesio. Papildomos spalvos analizės reikia elementų įterpimo ar išmetimo metu.

Įterpimo operacijos "papildomos pastangos" irgi yra nedidelės. Jos reiškia, kad vidiniame paieškos cikle reikia tikrinti, ar abi nuorodos į sūnus yra raudonos (t.y. ar yra 4-viršūnė), o tam

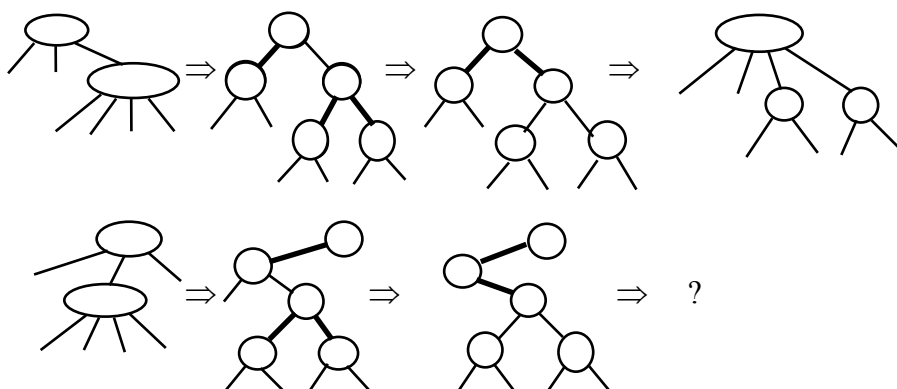
programoje reikia įsiminti prosenelio (gg), senelio (g) ir tėvo (p) nuorodas. Įterpimo procedūros programa būtų tokia:

```

function rbtreeinsert(v: integer; x: link): link;
var gg,g,p:link;
begin
    p:=x; g:=x;
    repeat
        gg:=g; g:=p; p:=x;
        if v<x^.key then x:=x^.l else x:=x^.r;
        if x^.l^.red and x^.r^.red then x:=split(v,gg,g,p,x);
    until x=z;
    new(x); x^.key:=v; x^.l:=z; x^.r:=z;
    if v<p^.key then p^.l:=x else p^.r:=x;
    rbtreeinsert:=x;
    x:=split(v,gg,g,p,x);
end;

```

Nuoroda gg (senelio nuoroda) reikalinga *split* operacijoje, būtent, kai 2-viršūnės vienas iš sūnų yra 4-viršūnė, jos abi ir nauja viršūnė turi būti transformuojamos į 3-viršūnę ir du jos sūnus - į 2-viršūnes; kai 3-viršūnės vienas iš sūnų yra 4-viršūnė, jos abi ir nauja viršūnė transformuojamos į 4-viršūnę ir du jos sūnus - į 2-viršūnes. Raudoniems-juodiems medžiams taip išreikšta skaidymo operacija reiškia, kad reikia pakeisti nuorodų spalvą - dviejų sūnų raudonas jungtis reikia padaryti juodomis, o jų tėvo jungtį su seneliu padaryti raudona, kaip parodyta 13 pav.



13 pav. Spalvų transformacija raudonuose-juoduose medžiuose.

13 pav. matome, kad po viršūnių skaidymo raudonų-juodų medžių vienoje šakoje gali susidaryti viena po kitos einančios dvi raudonos jungtys - struktūriniu požiūriu neleistina situacija. Tai reiškia, kad reikia papildomų veiksmų su viršūnėmis. Analizė rodo, kad taip gali atsitikti dėl “neteisingos” sūnų orientacijos - 4-viršūnė yra kairysis ar vidurinis (o ne dešinysis, kaip pirmuose dviejuose pavyzdžiuose) sūnus. Aišku, kad viršūnes reikia perorientuoti.

Raudonų-juodų medžių atveju šitoki perorientavimą galima suformuluoti kaip gan paprastai išreiškiamą sukimo operacija (*rotate*). Būtent turi būti keičiamos vietomis trys jungtys: tėvo kairioji jungtis nukreipiama į jo kairiojo sūnaus dešinį anūką, kairiojo sūnaus jungtis pertvarko tėvą į dešinį sūnų, o senelio dešinioji jungtis nukreipiama į kairįjį sūnų. Be to, dviejų jungčių spalvos sukeičiamos tarpusavyje.

Ši paprasta sukimo operacija gali būti apibrėžta ne tik raudonam-juodam, bet ir bet kuriam dvejetainiam medžiui. Savo ruožtu ji buvo panaudota keliems subalansuotų medžių algoritmams. Operacija *rotate* išsaugo paieškos medžio esmę ir lokaliai keičia tik tris viršūnes. Tačiau svarbu

pažymėti, kad ši operacija neturi tendencijos gerinti medžio balansą, nes viršūnės, kuri po sukimo tampa dešiniuoju sūnumi, sūnūs pažemėja vienu lygiu.

Atliekant *rotate* operaciją, modifikuojama medžio struktūra, todėl tai reikia daryti atsargiai. Tinkamas pavyzdys yra pateikiama programa, kai viršūnės *v* tėvas yra viršūnė *y*:

```
function rotate(v: integer; y: link): link;
var c,gc: link;
begin
  if v<y^.key then c:=y^.l else c:=y^.r;
  if v<c^.key
    then begin gc:=c^.l; c^.l:=gc^.r; gc^.r:=c end
    else begin gc:=c^.r; c^.r:=gc^.l; gc^.l:=c end;
  if v<y^.key then y^.l:=gc else y^.r:=gc;
  rotate:=gc
end;
```

Atskirai verta pateikti procedūros *split* programą, kuri taip pat iliustruoja, kaip reikia įsiminti jungtis:

```
function split(v: integer; gg,g,p,x: link): link;
begin
  x^.red:=true; x^.l^.red:=false; x^.r^.red:=false;
  if p^.red then begin
    g^.red:=true;
    if (v<g^.key)<>(v<p^.key) then p:=rotate(v,g);
    x:=rotate(v,gg); x^.red:=false
  end;
  head^.r^.red:=false;
  split:=x
end;
```

Tai turi įtakos raudono-juodo medžio inicializacijos procedūrai:

```
type link=^node;
  node=record key, info:integer; l,r:link; red: boolean end;
var head,z:link;
procedure rbtreeinitialize;
begin
  new(z); z^.l:=z; z^.r:=z; z^.red:=false;
  new(head); head^.key:=0; head^.r:=z;
end;
```

Sujungus šių programų tekstus į vientisą, gaunamas efektyvus pakankamai paprastas algoritmas, įterpiantis viršūnę į dvejetainį medį ir garantuojantis, kad paieškai ir įterpimams bus naudojamas logaritminis lyginimų kiekis:

- vienai paieškai raudoname-juodame medyje reikia apie  $\lg N$  lyginimų;
- vienam įterpimui vidutiniškai reikia mažiau negu vieno sukimo.

Raudoni-juodi medžiai vertingi ir tuo, kad jie efektyviai dirba ir blogiausiu atveju. Paieškai raudoname-juodame medyje blogiausiu atveju reikia mažiau negu  $2\lg N + 2$  lyginimų, o įterpimo



operacijai - ketvirtadaliu mažiau sukimų negu lyginimų. Vadinasi, turint apie pusę milijono įrašų, 2-3-4-medžio (arba raudono-juodo medžio) aukštis bus ne didesnis už 20 lygių.

### 3.3 Dėstymo lentelės

Dėstymo lentelės, dar kartais vadinamos išdėstymo funkcijomis (angliškai *hashing*) atstovauja visiškai kitokiais principais pagrįstiems paieškos algoritmams. Tai metodai, kurie rašo įrašus tiesiogiai į lenteles (masyvus), adresus įrašams skaičiuodami įrašų reikšmių pagrindu (atlikdami su įrašų reikšmėmis aritmetines operacijas). Tarkime, jei žinoma, kad rakto reikšmės yra skirtingos ir keičiasi nuo 1 iki  $N$ , tai įrašą, atitinkantį rakto reikšmę  $i$ , reikia rašyti į matricos ar lentelės  $i$ -ąją vietą, tokiu būdu naudojant tiesioginio išrinkimo metodą. Išdėstymo lentelių algoritmai kuriami apibendrinant šį trivialų metodą.

Paieškos operacijos išdėstymo lentelėse atliekamos paprastai: elementui, kurį norima rasti, pagal *hash* funkcija skaičiuojamas adresas lentelėje ir lyginami tuo adresu esantys vienas ar keli elementai. Įterpimo operacija vykdoma irgi taip pat: elementui, kurį reikia įterpti, skaičiuojamas adresas ir tuo adresu elementas rašomas į lentelę. Panaši ir išmetimo operacija.

Išdėstymo lentelių algoritmai paprastai susideda iš dviejų dalių. Pirmas žingsnis - tai adreso lentelėje skaičiavimas (*hash* funkcijos reikšmės skaičiavimas). Jei tokia funkcija parinkta idealiai, tai kiekvienam įrašui skaičiuojamas vienintelis galimas adresas, tada kiekvienas įrašas turi vieną ir tik vieną vietą lentelėje, ir algoritmą galima tobulinti nebent tik trumpinant funkcijos reikšmių skaičiavimo laiką. Tačiau tokių *hash* funkcijų beveik neegzistuoja, todėl paprastai reikalingas papildomas žingsnis: kolizijų išskyrimas, t.y. kai keliems įrašams skaičiuojama ta pati vieta lentelėje (tas pats adresas), reikia nuspręsti, kaip išskirti įrašus. Tada yra galimos skirtingos kolizijų išskyrimo strategijos, t.y. skirtingi algoritmai.

Išdėstymo funkcijų algoritmai priklauso informatikos klasikais. Tokių funkcijų ir algoritmų yra pasiūlyta daug, ir jie detalčiai ištirti. Išdėstymo funkcijoms yra žinoma daug rekomendacijų, kurios įgalina efektyviai išnaudoti norimą ar turimą laiko ir atminties santykį ir jį balansuoti.

#### 3.3.1. Išdėstymo funkcijos

Dėstymo lentelių algoritmams pirmiausia reikia funkcijos, kuri transformuotų rakto reikšmės (simbolių sekas, ar skaičius, ar kitokius duomenis) į lentelės adresus. Galima tarti, kad tai yra skaičiai, išsidėstę intervale  $[1 \dots M]$ ; čia  $M$  reikšmė parenkama praktiniais sumetimais (pvz., kiek turima atminties). Funkcija beveik visad yra aritmetinė. Ideali *hash* funkcija yra tokia, kurios reikšmės galima lengvai ir greitai skaičiuoti ir kuri visas galimas reikšmes "paskirsto" tolygiai, t.y. bet kuri gaunama reikšmė vienodai tikėtina. Tokių funkcijų parinkimo problemos labai panašios į atsitiktinių skaičių generatorių sudarymą.

Skaičiuojant tokių funkcijų reikšmes, pirmiausia reikia įrašą ar raktą interpretuoti kaip skaičių. Dažniausiai tokiu atveju rakto dvejetainė išraiška (dvejetainė eilė, kai kiekvienas simbolis pakeičiamas atitinkančiu dvejetainiu kodu) yra skaitoma kaip dvejetainis sveikas skaičius be ženklų. Po to šį skaičių (jei jis neperdaug ilgas) verčiant adresu, yra naudojama (dažniausiai) **mod** funkcija. Adresą skaičiuojanti išdėstymo funkcija užrašoma taip:  $h(k) = k \bmod M$ . Norint patenkinti tolydumo reikalavimą,  $M$  turi būti pirminis skaičius.

Kai dvejetainė eilė ilga, ją pateikti kompiuteriui kaip skaičių - kintamojo reikšmę - yra sudėtinga. Tokiu atveju naudojami kitokie metodai, pvz., ilga dvejetainė seka interpretuojama kaip kokios nors pozicinės skaičiavimo sistemos (32-ainės) skaičius, o adreso reikšmėms skaičiuoti taikomas Hornerio metodas:

```
h:=key[1];
for j:=2 to keysize do
    h:=(h * 32) + key[j] mod M;
```

Išdėstymo funkcijos detalai aprašytos D. Knuto [12] knygos 3-me tome. Sudėtingesnės funkcijos, įdomios ir matematikos, ne tik informatikos požiūriu, aprašytos [11] knygoje.

### 3.3.2. Atskirti sąrašai.

Išdėstymo funkcijos tą patį adresą gali priskirti keliems įrašams. Todėl būtina išspręsti tokio pobūdžio kolizijas. Atskirtų sąrašų algoritmas teigia, kad kiekvieną įrašų, turinčių tą patį adresą, grupę reikia formuoti kaip atskirą tiesinį sąrašą. Vadinasi yra  $M$  sąrašų, o *hash* funkcijos adresai yra ne kas kita, kaip nuorodos į kažkurio sąrašo pradžią. Kartais sąrašo elementus naudinga laikyti surūšiuotus. Programuojant šį algoritmą, reikia naudoti nuorodų masyvą. Programa šiam metodui atrodytų taip:

```

type link = ^node;
      node = record key.info: integer; next: link end;
var heads: array[0..M] of link; t,z:link;
procedure initialize;
var i:integer;
begin
    new(z); z^.next:=z;
    for i:=0 to M-1 do
      begin new(heads[i]); heads[i]^.next:=z end;
end;

```

**Algoritmo savybės.** Naudojamų lyginių skaičius yra lygus nuoseklos paieškos lyginių skaičiui, padalytam iš  $M$ . Kompiuteris naudoja papildomą atmintį  $M$  nuorodoms saugoti.

Nuorodų masyvo galima būtų išvengti, jei funkcija nuorodas formuotų tiesiogiai į pirmą reikiamo sąrašo elementą. Tačiau tai komplikuoja algoritmo programavimą.

### 3.3.3. Atviro adresavimo dėstymo metodai

Dėstymo metodus naudoja kompiliatoriai, ypač sudarydami simbolių lenteles. Šiuo atveju atminties yra pakankamai, todėl lentelių dydis didesnis už elementų kiekį. Dėstymo metodai tokiu atveju vadinami atviro adresavimo metodais. Tačiau *hash* funkcijų skaičiuojami adresai skirtingiems įrašams vis dėlto gali sutapti, todėl algoritmai turi nagrinėti ir kolizijų atvejus. Čia bus išdėstyti du tokie algoritmai: tiesinio dėstymo (*linear probing*) ir dvigubo dėstymo (*double hashing*).

**Tiesinis dėstymas.** Tiesinio dėstymo algoritmas, paskaičiavęs elemento adresą, tikrina, ar jau tokiu adresu nurodyta vieta yra užimta, ir jei taip - nuosekliai vieną po kitos tikrina kitas pozicijas tol, kol suranda laisvą vietą. Įterpimo operacijos metu naujas elementas rašomas į šią vietą, o paieškos operacija užbaigiama. Šiame algoritme galimi keli skirtingi lyginimai: galima lyginti raktus arba *hash* funkcijos reikšmes raktams (priklausomai nuo taikymo). Programa, realizuojanti algoritmo operacijas, būtų tokia:

```

procedure hashinitialize;
var i:integer;
begin
    for i:=0 to M do a[i].key:=maxint;
end;
function hashinsert(v: integer): integer;
var x:integer;
begin

```

```

x:=h(v);
while a[x].key<>maxint do x:=(x+1) mod M;
a[x].key:=v;
hashinsert:=x;
end;

```

**Savybė.** Remiantis statistiniais vertinimais, galima teigti, kad tiesinis dėstymas naudoja vidutiniškai mažiau negu penkis nuoseklius perrinkimus, kai lentelė užpildyta daugiau negu dviem trečdaliais.

**Dvigubas dėstymas.** Tiesinio dėstymo algoritmas gali dirbti labai lėtai, jei lentelė būna beveik užpildyta. Tada vietoje jo tikslinga naudoti kitą algoritmą. Vienas iš tokių yra dvigubo dėstymo algoritmas. Jame siūloma pakeisti vietų lentelėje perrinkimo strategiją - vietoje tiesinio perrinkimo naudoti kintamo žingsnio perrinkimą. Kitaip naudojama *hash* funkcija: po to, kai apskaičiuojamas rakto adresas ir, jei ta vieta užimta, reikia pereiti prie kito elemento, tikslinga naudoti kintamą žingsnį:  $x := (x + u) \bmod M$ . Kad algoritmas sėkmingai veiktų, žingsnis  $u$  ir skaičius  $M$  turi būti tarpusavyje pirminiai. Be to,  $u$  gali priklausyti nuo rakto  $x$  reikšmės.

Todėl dažniausiai tokiu atveju apibrėžiama nauja (antra) išdėstymo funkcija. Pavyzdžiui, ji gali būti tokia:  $h_2(k) = (M - 2 - k) \bmod (M-2)$ . Įvairiose dvigubo dėstymo algoritmo modifikacijose pasiūlyta ir daugiau tokių funkcijų [11]. Kaip ir anksčiau, skaičiai  $M$  ir  $M-2$  turi būti pirminiai.

**Empirinė savybė.** Dvigubas dėstymas laisvos vietos paieškoje paprastai reikalauja mažiau bandymų negu tiesinis dėstymas.

Dėstymo algoritmų srityje yra atlikta daug tyrimų ir pasiūlyta daug funkcijų, kurios dažniausiai testuotos ir įvertintos empiriškai. Matematinė tokių funkcijų analizė ir vertinimai surinkti [13 knygoje, jie yra gan sudėtingi.

### 3.4 Skaitmeninė paieška

Paieškos ar kokių nors kitų operacijų tikslams dažnai galima panaudoti raktų ar įrašų reikšmių dvejetainę (skaitmeninę) išraišką (o ne įrašų reikšmes), kurią šie įrašai turi kompiuterio atmintyje. Įrašų išraiškų bitai gali būti analizuojami ar lyginami tarpusavyje, šių lyginimų pagrindu daromos išvados, tokios pat kaip ir anksčiau nagrinėtais paieškos algoritmų atvejais. Tokie algoritmai vadinami skaitmeninės paieškos (*radix-searching*) algoritmais. Juos patogiau naudoti tada, kai su dvejetainėmis įrašų išraiškomis lengva manipuluoti, ir kai jos vienokia ar kitokia prasme pakeičia įrašų reikšmes.

#### 3.4.1. Skaitmeniniai paieškos medžiai

Tai paprasčiausias skaitmeninės paieškos algoritmas, labai panašus į dvejetainės paieškos medį. Jų skirtumas tas, kad pereinant iš vienos viršūnės į kitą yra lyginamos ne raktų (viršūnių) reikšmės, bet šių reikšmių dvejetainių išraiškų bitai. Pereinant nuo medžio šaknies į jos sūnus, lyginamas 1-as bitas, iš šių sūnų į jų sūnus - 2-as bitas ir t.t., iki tol, kol nebus pasiektas lapas (šakos pabaiga). Jei bito reikšmė lygi nuliui, pereinama į kairįjį sūnų, jei vienetui - į dešinį sūnų. Rašant šio algoritmo programą, tikslinga apibrėžti funkciją, kuri iš tam tikrų raktų reikšmių išskirtų reikalingus bitus:  $bits(x, k, j)$  - iš reikšmės  $x$  iškiria  $j$  bitų, pradedant nuo  $k$ -ojo bito. Algoritmo programos tekstas tada būtų toks:

```

function digitalsearch (v: integer; x: link): link;
var b:integer;
begin
  z^.key:=v; b:=maxb;
  repeat

```

```

    if bits(v,b,1)=0 then x:=x^.l else x:=x^.r;
    b:=b-1;
until v=x^.key;
digitalsearch:=x
end;

```

Duomenų struktūros, naudojamos šiame algoritme, yra tokios pat kaip ir dvejetainių medžių. Sutampančios raktų reikšmės skaitmeninėje paieškoje (kaip ir skaitmeniniame rūšiavime) sudaro esminį kliuvinį. Todėl šiame skyriuje bus nagrinėjami tik tie atvejai, kai visų raktų reikšmės yra skirtingos.

Skaitmeninės paieškos įterpimo procedūra irgi yra tik lengva dvejetainio paieškos medžio modifikacija:

```

function digitalinsert (v: integer; x: link): link;
var p: link; b: integer;
begin
    b:=maxb;
    repeat
        p:=x;
        if bits(v,b,1)=0 then x:=x^.l else x:=x^.r;
        b:=b-1;
    until x=z;
    new(x); x^.key:=v; x^.l:=z; x^.r:=z;
    if bits(v,b+1,1)=0 then p^.l:=x else p^.r:=x;
    digitalinsert:=x
end;

```

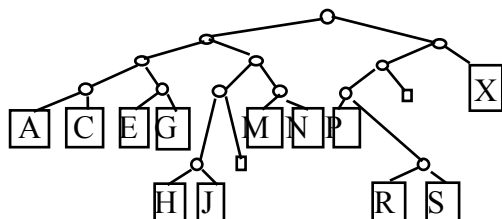
**Algoritmo sudėtingumas:** Paieškai ar įterpimui skaitmeniniame paieškos medyje, kurį sudaro  $b$  bitų ilgio  $N$  viršūnių, reikia vidutiniškai apie  $\lg N$  lyginimų (ir  $b$  lyginimų blogiausiu atveju).

### 3.4.2. Skaitmeniniai paieškos indeksai

Skaitmeniniuose paieškos medžiuose elementų reikšmės, nors ir dvejetainės, yra saugomos (ir lyginamos, ar apdorojamos, t.y. iš jų išskiriami bitai ar jų grupės) viršūnėse. Jei šios reikšmės yra ilgos, jos gali labai sumažinti algoritmo efektyvumą.

Šiame skyrelyje bus aptariami algoritmai, kurie išnaudoja skaitmeninių paieškos medžių privalumus, bet įgalina elemento reikšmę paieškos operacijoje lyginti tik vieną kartą. Idėja paimta iš gyvenimo: telefonų knygoje puslapių viršuje rašomos pirmos pavardžių raidės, bet ne pilnos pavardės. Todėl paieška joje atliekama ieškant pirmos reikalingo žodžio raidės, o tik po to viso žodžio. Plėtojant šią idėją, 1960-ais metais E. Fredkin pasiūlė tokios paieškos duomenų struktūrą ir algoritmą [12]. Ši struktūra žinoma "*trie*" vardu (nuo žodžių *tree* ir *retrieval*). Jos esmė yra tokia:  $M$ -ariniame medyje skiriamos vidinės ir išorinės viršūnės, prie kiekvienos vidinės viršūnės rašomas  $(M-1)$ -matis vektorius (skaitmeninis ar simbolinis), kurio reikšmės indeksuoja jungtis ir nurodo, kuria iš jų reikia eiti toliau. Elementų reikšmės rašomos išorinėse viršūnėse (kurios yra medžio lapai). Dvejetainėje *trie* struktūroje vidinėse medžio viršūnėse yra rašomi nuliai ar vienetai (bitai), o viršūnių numeriai atitinka bito poziciją įrašo dvejetainėje išraiškoje. Pavyzdžiui, norint rasti elementus, kurie trečio bito pozicijoje turi reikšmę, lygią vienetui, reikia iš trečios viršūnės pereiti į dešinį sūnų ir taip tęsti toliau. Ši struktūra lietuviškai bus vadinama "skaitmeniniais paieškos indeksais" (iki šiol lietuvių kalboje nusistovėjusio termino nėra). *Trie* struktūros įvairios realizacijos, modifikacijos ir analizė pateikta [12] knygoje.

Skaitmeniniai paieškos indeksai nepriklauso nuo elementų įterpimo tvarkos - tiems patiems duomenims gaunamas tas pats medis nepriklausomai nuo to, kokia tvarka elementai buvo įtraukti. Įterpimo operacijos metu reikia atlikti nesėkmingą paiešką ir prie vidinės viršūnės prijungti išorinę, kartu pakeičiant ją atitinkantį indeksą. *Trie* struktūros pavyzdys pateiktas 14 pav.



14 pav. *Trie* struktūra raidžių penkių bitų kodams.

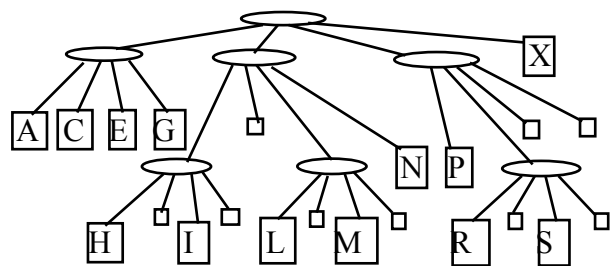
Programuoti *trie* struktūrą nėra paprasta, nes ji reikalauja specialių operacijų su bitų ar panašių simbolių indeksais. Knygose [1, 12] aptarta daug programavimo ir programų optimizavimo metodų. Ši struktūra yra geras algoritmo, kurį verta programuoti žemo lygio programavimo kalba, pavyzdys.

**Algoritmų sudėtingumas.** Skaitmeninės paieškos indeksų struktūra turi vidutiniškai apie  $N/\lg N$  (t.y. apie 1,44N) vidinių viršūnių.

Viena iš *trie* struktūros problemų - galimi skirtingi paieškos medžio šakų ilgiai. Ji gali būti sprendžiama arba dvejetainį medį pertvarkant į M-arinį, arba trumpinant šakas, kuriose yra daug ilgų vienodų indeksų.

### 3.4.3. Daugybiniai skaitmeniniai paieškos indeksai

Struktūros, kurios vadinamos daugybiniais skaitmeniniais paieškos indeksais, siūlo pereinant nuo vienos viršūnės prie kitos naudoti kelių bitų sekas (pvz., b ilgio) ir sudaryti M-arinis medžius ( $M=2^b$ ). Šis metodas tinkamas tuo atveju, kai yra daug elementų, kurių dvejetainėje išraiškoje tam tikros bitų sekos kartojasi. M-arinis medis bus šiuo atveju labiau subalansuotas ir jo šakos trumpesnės (šakų ilgis bus  $\log_M N$ ). Vadinasi ir paieška bus vykdoma greičiau. Tokio daugybinio medžio pavyzdys pateiktas 15 pav.



15 pav. Keturių lygių *trie* struktūra raidžių kodams.

Šis metodas labai efektyvus, kai turima daug raktų ir jų reikšmių - matuojamų tūkstančiais. Kita vertus, šioje struktūroje yra paslėpta galimybė užteršti atmintį, nes gali būti daug neišnaudotų jungčių.

### 3.4.4. Patricia algoritmas

Skaitmeninės paieškos medžiai turi dvi savybes, kurios potencialiai trukdo efektyviam algoritmo veikimui ir jo realizacijai:

- potenciali ir dažnai pasitaikanti galimybė turėti daug neišnaudotų jungčių;
- paieškos medžio viršūnės turi būti dviejų skirtingų tipų.

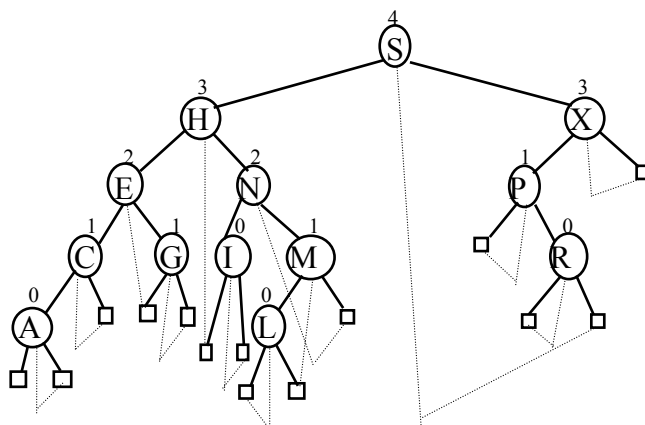
Šios neigiamos savybės yra eliminuotos D. R. Morrison'o pasiūlytame *Patricia* algoritme. Jame sudaromas medis iš  $N$  viršūnių (kiekvienam elementui po viršūnę), o kiekvienai viršūnei priskiriama informaciją leidžia naudoti tik vieną pilną elementų lyginimą per vieną paieškos operaciją.

Neišnaudotos jungtys eliminuojamos paprastai - kiekvienai viršūnei priskiriamas papildomas indeksas - bito numeris, kurį reikia lyginti norint nuspręsti, kuria šaka toliau eiti. Elementų reikšmės saugomos viršūnėse, todėl išnyksta skirtumas tarp vidinių ir išorinių viršūnių. Kiekviena viršūnė turi dvi jungtis, iš kurių kiekviena gali būti nukreipta ne tik į sūnus, bet ir į pačią viršūnę ar į kurią nors aukštesnio lygio viršūnę.

Kad būtų vaizdžiau, verta išnagrinėti pavyzdį. Tarkime, turime tokias raides ir jų dvejetainius kodus:

A	00001	S	10011
E	00101	R	10010
C	00011	H	01000
I	01001	N	01110
G	00111	X	11000
M	01101	P	10000
L	01100		

Nuosekliai, tokia tvarka, kaip išdėstyti elementai lentelėje, įterpiant viršūnes į *Patricia* medį gaunama 16 pav. pavaizduota struktūra.



16 pav. *Patricia* medis raidžių penkių bitų kodams.

*Patricia* medyje paieška vykdoma taip: jei reikia rasti, tarkime, elementą  $X$ , yra skaitoma jo dvejetainio kodo pozicija, kuri pažymėta ties šaknine viršūne, ir tokiu būdu nusprendžiama, ar eiti į kairįjį ar į dešinįjį sūnų. Ši operacija kartojama tol, kol prieinama nuoroda, nukreipta arba į tą pačią viršūnę, arba į aukščiau esančią viršūnę. Tada yra lyginamos ieškomo elemento  $X$  ir nurodytos viršūnės reikšmės ir paieška užbaigiama. Jei reikšmės sutapo, paieška sėkminga, jei ne - nesėkminga. Reikia pastebėti, kad būtent *Patricia* medžio konstrukcija atspindi specifinius skirtumus tarp elementų. Pavyzdžiui., elemento  $S$  kodas yra toks, kad dvejetainis posekis  $10 \cdot 11$  tarp kitų elementų nėra sutinkamas. Šis skirtumas ir yra priežastis, kodėl iš apačios į viršų nuoroda (iš artimiausio elementui  $S$  elemento  $R$ ) buvo nukreipta į elementą  $S$ .

Įterpimo operacija yra sudėtingesnė, jos realizacijoje reikia skirti du atvejus, kai viršūnė įterpiama medžio šakoje (vidinė viršūnė) ir kai lape (išorinė viršūnė). Tokios paieškos ir įterpimo operacijų programos galėtų būti tokios:

```

type link:=^node;
node = record key. info, b: integer; l,r: link end;
var head,z: link;
function patriciasearch(v:integer; x:link):link;
var p:link;
begin
    repeat
        p:=x;
        if bits(v,x^.b,1)=0 then x:=x^.l else x:=x^.r;
        until p^.b=<=x^.b;
        patriciasearch:=x
    end;

function patriciainsert(v:integer; x: link): link;
label 0;
var t,p:link; i:integer;
begin
    t:=patriciasearch(v,x);
    if v=t^.key then goto 0;
    i:=maxb;
    while bits(v,i,1)=bits(t^.key,i,1) do i:=i-1;
    repeat
        p:=x;
        if bits(v,x^.b,1)=0 then x:=x^.l else x:=x^.r;
    until (x^.b=<=i) or (p^.b=<=x^.b);
    new(t); t^.key:=v; t^.b:=i;
    if bits(v,t^.b,1)=0
        then begin t^.l:=t; t^.r:=x end
        else begin t^.r:=t; t^.l:=x end;
    if bits(v,p^.b,1)=0 then p^.l:=t else p^.r:=t;
    0:patriciainsert:=t
end;

```

**Algoritmo sudėtingumas.** *Patricia* medis, sudarytas iš atsitiktinių  $b$  bitų ilgio  $N$  elementų, turi  $N$  viršūnių ir paieškos operacijai vidutiniškai naudoja  $\lg N$  lyginimų.

### 3.5 Išorinė paieška

Praktiškai dažnai reikia ieškoti duomenų dideliuose failuose, kurie laikomi išorinėje atmintyje. Kaip ir išorinio rūšiavimo atveju, tokia paieška vadinama išorine. Nagrinėjami algoritmai bus pritaikyti darbui antrinėje, t.y. diskinėje atmintyje. Šią atmintį galima šiek tiek formalizuoti, skirtingai negu tretinę atmintį, kurioje gausu labai įvairių ir skirtingų techninių sprendimų, darančių įtaką algoritmų darbui.

Paieška yra viena iš labai dažnų ir fundamentalių operacijų su diskiniiais įrenginiais. Nors diskinių įrenginių konstrukcijos skiriasi, visoms joms galima naudoti tam tikrą modelį. Tą daryti verčia ir manipuliacijų su duomenimis pobūdis - operacijų metu reikia pakeisti, įterpti, išmesti ar rasti kelis bitus ar baitus labai dideliuose failuose. Failai paprastai yra rašomi taip, kad išnaudotų visas technines diskinių įrenginių charakteristikas ir išrinktų duomenis kuo greičiau. Todėl labai sudėtinga rasti objektyvius kriterijus, kurie pagelbėtų vertinant paieškos metodų efektyvumą įvairiuose taikymuose, nes labai daug kas priklauso nuo techninės ir programinės įrangos. Formalus diskinių įrenginių modelis padeda vaizdžiai ir matematiškai griežtai pateikti išorinės

paieškos algoritmus. Šie, skirtingai nuo išorinio rūšiavimo, nelabai skiriasi nuo vidinės paieškos algoritmų, tik yra labiau pritaikyti dinaminei duomenų dėstymo situacijai.

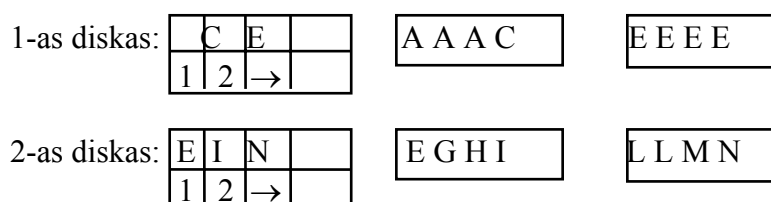
Nagrinėjami išorinės paieškos metodai yra naudojami ir naudingi ne tik failams ar jų sistemoms, bet ir duomenų bazėms. Tačiau duomenų bazės turi labai didelę inerciją, t.y. kai koks nors algoritmas jose yra įdiegtas, visą manipuliavimo su duomenimis struktūrą labai sunku pakeisti. Paprastai įdiegti algoritmai gyvuoja tol, kol gyvuoja duomenų bazių taikymas. Nauji, dinamiški paieškos metodai diegiami tik kartu su naujomis duomenų bazėmis.

Diskinės atminties modelyje atmintis skirstoma į atskiras sekcijas, vadinamuosius puslapius. Kiekvienas diskas turi po fiksuotą kiekį puslapių (nagrinėjamuose algoritmuose - po tris). Į puslapį galima rašyti fiksuotą kiekį įrašų (nagrinėjamuose algoritmuose - po keturis), kurie puslapio viduje išrenkami nuosekliai. Skaitymo-rašymo (*input/output* - *I/O*) operacijos reikalauja daug daugiau laiko negu visos kitos kompiuterio operacijos. Skaitymo-rašymo operacijos metu visas puslapis perkeliamas į vidinę atmintį (arba atvirkščiai).

### 3.5.1. Indeksinė-nuosekli paieška

Paprasčiausias nuoseklos paieškos algoritmas gali būti pritaikytas ir paieškai diskinėje atmintyje. Tokiu atveju įrašus reikia dėstyti nuosekliai didėjimo tvarka disko puslapiuose. Tada paieška atliekama kreipiantis į kiekvieną puslapį ir nuskaitant jo turinį į atmintį. Algoritmas dirbs tol, kol bus sutiktas įrašas, didesnis už ieškomą, ir jo veikimo laikas bus tiesiogiai proporcingas nuskaitytų puslapių skaičiui.

Nuoseklos paieškos algoritmą galima akivaizdžiai patobulinti. Diske esantiems puslapiams galima sukurti rodyklę (angliškai *index*), kurioje nurodoma, kokia įrašo reikšmė prasideda ar baigiasi kiekvienas puslapis. Rodyklei skiriamas atskiras puslapis. Paieška šioje struktūroje yra daug greitesnė: norint rasti reikiamą įrašą, yra nuskaitomas rodyklės puslapis ir po to - tas puslapis, kuriame gali būti ieškomasis įrašas. Šitai paieškai paprastai reikia dviejų *I/O* operacijų. Jei yra keli diskai, galima dar kurti rodyklių rodyklę (*master index*), kurioje rašomi įrašų reikšmės, esančių kiekviename diske diapazonai. Paprastai tokia rodyklė yra maža ir paieškos metu ją galima laikyti vidinėje atmintyje. Šitokia duomenų struktūra vadinama indeksiniu-nuosekliu failu ar paieška, jos pavyzdys pateiktas 17 pav. Ji labai tinka greitai duomenų paieškai, bet jos įrašų įterpimo ar išmetimo operaciją vykdo labai ilgai, nes gali tokiu atveju prireikti pakeisti visų disko ar diskų puslapių turinį.



17 pav. Indeksinė-nuosekli struktūra diskuose.

**Algoritmo sudėtingumas:** Paieškos operacija indeksiniame-nuosekliame faile vyksta pastoviu laiku, tačiau įterpimo ar išmetimo operacijos gali pareikalauti pertvarkyti visą failą (t.y. eksponentiniu laiku).

### 3.5.2. B-medžiai

Kai duomenys kokioje nors aibėje keičiasi dinamiškai, t.y. kai paieškos, įterpimo ar išmetimo operacijos yra vienodai tikėtinos, geriausia naudoti vienodo (ir mažiausio galimo) sudėtingumo struktūras. Viena iš tokių tinkamų struktūrų - subalansuoti medžiai. Išorinės paieškos atveju medžiai paprastai susieja disko ar diskų puslapius ir todėl *I/O* operacijų atžvilgiu turi būti optimizuoti, t.y. jie turi būti gan šakoti. Algoritmai, kurie čia naudojami, gali apibendrinti (ar



pritaikyti) 2-3-4-medžius. Medžių viršūnės gali turėti priskirtas iš tam tikro intervalo (nuo 1 iki  $M-1$ ) reikšmes, t.y. gali turėti nuo 2 iki  $M$  nuorodų. Nuorodos gali būti nukreiptos į puslapius, kuriuose įrašyti tame intervale esantys įrašai (teisingiau į puslapį, kuriame prasideda įrašai). Šitokia struktūra yra vadinama B-medžiais, ją 1972-ais metais pasiūlė autoriai R. Bayer ir McCreight [2]. Jos pavyzdys pateiktas 18 pav.

1-as diskas:	<table><tr><td>2</td><td>D</td><td>2</td><td>M</td><td>2</td></tr><tr><td>0</td><td></td><td>1</td><td></td><td>2</td></tr></table>	2	D	2	M	2	0		1		2	<table><tr><td>3</td><td>T</td><td>β</td><td></td></tr><tr><td>0</td><td></td><td>1</td><td></td></tr></table>	3	T	β		0		1		
2	D	2	M	2																	
0		1		2																	
3	T	β																			
0		1																			
2-as diskas:	<table><tr><td>A</td><td>A</td><td>A</td><td>B</td></tr></table>	A	A	A	B	<table><tr><td>D</td><td>E</td><td>I</td><td>L</td></tr></table>	D	E	I	L	<table><tr><td>M</td><td>N</td><td>O</td></tr></table>	M	N	O							
A	A	A	B																		
D	E	I	L																		
M	N	O																			
3-as diskas:	<table><tr><td>S</td><td>S</td><td>S</td><td>S</td></tr></table>	S	S	S	S	<table><tr><td>T</td><td>U</td><td>U</td></tr></table>	T	U	U	<table><tr><td></td><td></td><td></td></tr></table>											
S	S	S	S																		
T	U	U																			

18 pav. Sakinio “SUBALANSUOTAS MEDIS” (žr. 11 pav.) B-medis diske.

Paieškos, įterpimo ar išmetimo operacijos šioje struktūroje atliekamos kaip ir 2-3-4-medžiuose. Formaliai B-medžiuose yra dviejų tipų viršūnės: vidinės ir išorinės. Tačiau jų realizacija panaikina šį skirtumą. Jei medis yra mažas, jo visos vidinės viršūnės gali tilpti vidinėje atmintyje ir joms galima taikyti vieną iš anksčiau nagrinėtų struktūrų. Šio atveju nuorodos į išorines viršūnes bus diskų ir jų puslapių adresai. Jei medis didelis ir vidinėje atmintyje netelpa, kiekvienai vidinei viršūnei (išskyrus medžio šaknį) galima priskirti atskirą puslapį, turintį savą struktūrą (18 pav.). Įrašų adresavimas bus kompleksinis: disko numeris ir puslapio numeris. Saugant diske viršūnes, joms skirtuose puslapiuose yra saugomi ir visi jungtims reikalingi adresai, tačiau puslapiuose, saugančiuose įrašus, jungčių saugoti nereikia. Medžio šaknį verta laikyti vidinėje atmintyje, nes pradedant paiešką visad reikia formuoti reikalingus adresus. Aišku, B-medžio atminties kiekis priklauso nuo puslapių, įrašų dydžio ir nuorodų kiekio.

B-medžių įterpimo ir išmetimo operacijos, kai vieną viršūnę reikia skaidyti į dvi ar dvi viršūnes sujungti į vieną, formaliai yra tokios pat, kaip ir 2-3-4-medžiuose. Tačiau skiriasi jų realizacija. Pagrindinį dėmesį reikia sutelkti ne į struktūros elementų keitimą, o į tai, kad reikia minimizuoti operacijų su puslapiais skaičių.

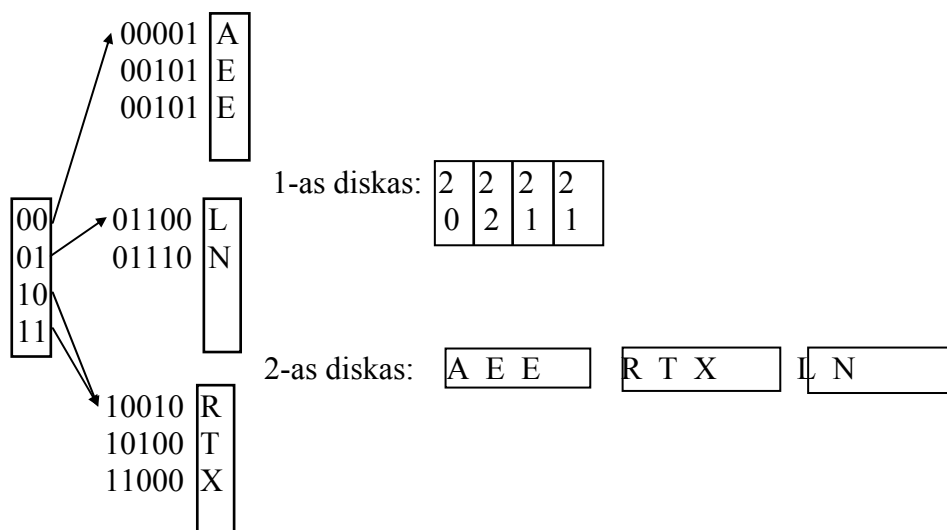
**Algoritmo sudėtingumas.** Paieškos, įterpimo ar išmetimo operacijoms B-medyje, turinčiame  $N$  elementų ir  $M$  jungčių, reikia  $\log_{M/2} N$  disko  $I/O$  operacijų. Praktiškai, kai skaičius  $M$  yra pakankamai didelis, tam reikia pastovaus laiko. Viršūnių skaičius, reikalingas tokiam medžiui, gali būti įvertintas apytikriai  $1,44N/M$ .

B-medis išsiskiria dar keliomis savybėmis, kurias galima išnaudoti algoritmo darbui pagerinti. Viena iš jų - viršūnėse galima naudoti kintamą jungčių (ir intervalų) skaičių. Viršutiniuose lygiuose galima naudoti daugiau jungčių, apatiniuose - mažiau, taip galima sutaupyti reikalingų medžiui saugoti puslapių skaičių, kartu  $I/O$  operacijų skaičių. Kita savybė - viršūnių įrašuose galima saugoti ne įrašų reikšmes, o jų kodus ar reikšmių prefiksus. Tokia procedūra gali sutaupyti laiko ir vietos. Literatūroje pasiūlyta daug tokių galimų modifikacijų (ir daug iš jų yra realizuota). Apie tai liudija ir aibė pavadinimų: prefix-B-medžiai, B+-medžiai, prefix-B+-medžiai, K-D-B-medžiai ir pan.

### 3.5.3. Išplėstinis dėstymas

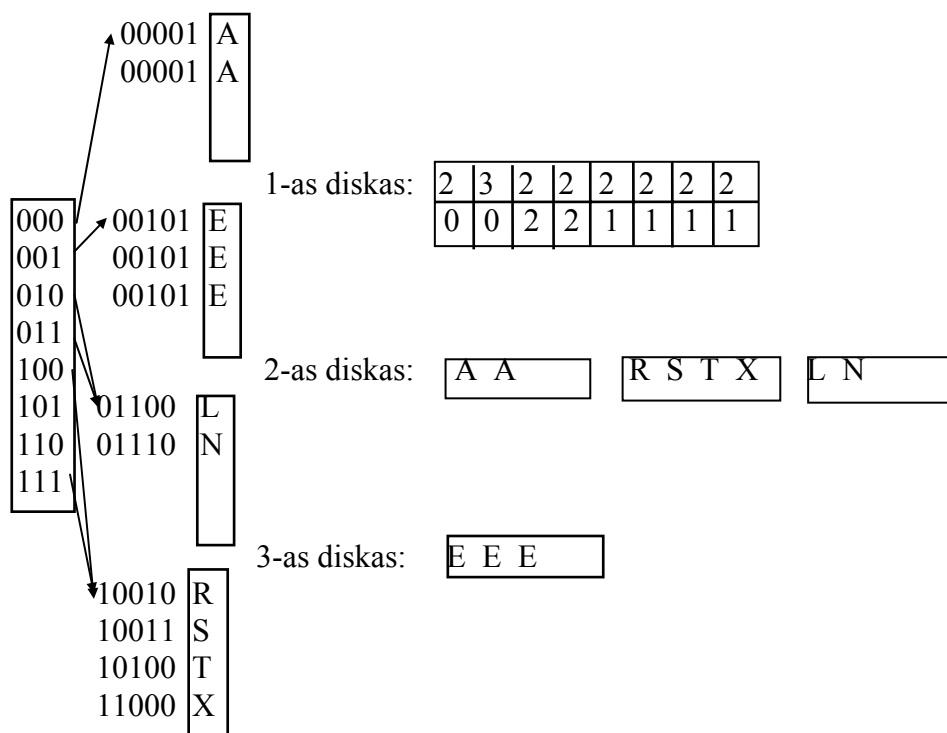
Išplėstinis dėstymas (*extendable hashing*) yra tam tikra alternatyva B-medžiams. Šis metodas pritaiko skaitmeninius paieškos metodus išorinei atminčiai, pirmą kartą buvo pasiūlytas 1978 metais [1], vėliau buvo paskelbta daug modifikacijų ir taikymų. Jo pagrindinis privalumas yra tai, kad tipinėse ir labai dažnai pasitaikančiose duomenų dėstymo situacijose paieškai reikia tik dviejų  $I/O$  operacijų ir panašiai tiek pat naujoms elementams įterpti ar išmesti. Išplėstinio dėstymo atveju, kaip ir B-medžiuose, įrašai ir jų indeksai saugomi disko puslapiuose. Indeksai

sudaromi naudojant dvejetainius įrašų kodus. Duomenų saugojimo šioje struktūroje schema parodyta 19 pav.



19 pav. Išplėstinio dėstymo struktūra diske.

Operacijos su indeksais išplėstinio dėstymo algoritmuose yra dvilypės: indekso struktūra disko puslapyje yra vienaip modifikuojama, kai keičiasi analizuojamų bitų skaičius įterpian ar išmetant elementus, ir kitap modifikuojama, kai reikia vieno puslapio turinį skaidyti į du puslapius arba du puslapius sujungti į vieną. Duomenų struktūra, gauta į 19 pav. struktūrą įterpus papildomus elementus, pavaizduota 20 pav.



20 pav. Elementų įterpimas išplėstinio dėstymo struktūroje.

Puslapio viduje įrašai saugomi surūšiuoti. Tai nėra esminė algoritmo darbo savybė, nes daugiausia laiko sunaudoja I/O operacijos. Puslapių modifikacija atliekama vadovaujantis tokiais principais: kai puslapį reikia skaidyti į dvi dalis, ir dalį įrašų perkelti į kitą puslapį, jie turi būti

rašomi taip, kad viename puslapyje liktų įrašai, prasidedantys 0, o į kitą patektų prasidedantys 1. Šią principą reikia taikyti ir toliau, sekoms 00, 01, 10, 11, ir t.t. Kai reikia pereiti prie ilgesnių skaitmeninių sekų, tenka padvigubinti rodyklę (indeksą). Rodyklę naudoja tik puslapių nuorodas - todėl jai reikia mažai vietos. Aišku, kad rodyklės skaidymas ir yra ta operacija, kuri gali labai pabloginti algoritmo darbą, nes gali tekti skaidyti ar sujungti daug puslapių. Tačiau tokia situacija būna ne taip jau dažnai. Todėl algoritmas dažniausiai dirba labai efektyviai. Jei rodyklę užima daugiau nei vieną puslapį, vidinėje atmintyje galima suformuoti rodyklės puslapių nuorodas ir su jų pagalba kreiptis tik į tą puslapį, kuriame yra reikiamas rodyklės fragmentas.

Tačiau yra ir neigiamų išplėstinio dėstymo algoritmo savybių. Algoritmas yra jautrus dvejetainių reikšmių pasiskirstymui. Jei daugumos įrašų reikšmės prasideda tuo pačiu dvejetainiu fragmentu, rodyklės tūris gali nepagrįstai išsipūsti, nes prie dvejetainio fragmento pridedant vieną bitą, rodyklės tūris padvigubėja. Esant minėtam netolygumui, patartina vietoje dvejetainio fragmento naudoti vienokią ar kitokią išdėstymo funkciją.

Kita algoritmo ypatybė - jo priklausomumas nuo puslapio dydžio. Teisingiau, šio algoritmo darbui yra svarbus įrašų, puslapių ir išdėstymo funkcijos reikšmių dydžių santykis. Tai daro įtaką ir rodyklės dydžiui, ir nuorodų skaičiui, ir užimamos atminties dydžiui. Aišku, toks faktorius yra svarbus, kai duomenų pakankamai daug, pvz., konstruojant labai dideles duomenų bazių sistemas.

Dar viena algoritmo elgsenos ypatybė, į kurią reikia kreipti dėmesį - nuorodų perteklius esant dideliame lygių įrašų kiekiui, t.y. kai vienas įrašas kartojasi daug kartų. Šie įrašai dirbtinai išpučia rodyklę, nes atsiranda daug nuorodų į tą pačią reikšmę. Ši problema panaši į dvejetainių fragmentų pasiskirstymo netolygumus ir gali būti sprendžiama panašiai. Tačiau jei kartojasi tik vienas arba tik keli įrašai, galima naudoti fiktyvias reikšmes, identifikuojančias pasikartojančių įrašų tinkamus poaibius.

**Algoritmo sudėtingumas.** Jei į diskinės atminties puslapį telpa  $M$  įrašų, atminties tūris, reikalingas saugoti  $N$  įrašų, bus apytikriai lygus  $1,44N/M$  puslapių. Išplėstinio dėstymo algoritmo rodyklė turės apie  $N^{1+1/M}/M$  nuorodų.

Reikia pastebėti, kad išplėstinis dėstymas yra labai panašus į virtualios atminties formavimo metodus. Virtuali atmintis, kuri yra ne kas kita, kaip bendros paskirties išorinės paieškos metodas, formuoja tiesioginius adresus į išorinės atminties puslapius ir remiasi prielaida, kad įrašai, esantys kompiuterio vidinėje atmintyje, ir įrašai, esantys diske, fiziškai yra netoli vienas nuo kito ir, be to, susiję. Nuorodų formavimo principai abiem algoritmams yra tokie patys. Tačiau virtualios atminties formavimo metodo, skirtingai negu išplėstinio dėstymo algoritmo, negalima tiesiogiai pritaikyti duomenų bazių valdymo sistemoms. Duomenų bazių ar failų sistemoms reikia, kad be jokių išankstinių susiejimų būtų galima efektyviai kreiptis iš bet kurios duomenų vietos į bet kurią kitą.

## 4. Sekų apdorojimas ir kompresija

### 4.1 Sekų paieška

Šiame skyriuje sąvoka "seka" bus interpretuojama kaip duomenų aibė, kuri negali būti išskaidyta į logiškai nepriklausomus įrašus taip, kad kiekvienas jų turėtų nesudėtingai identifikuojamą dalį. Sekos - tai dviejų skirtingų kompiuterių programų sistemų šeimų pagrindinis objektas: tekstų apdorojimo sistemų, vadinamųjų dokumentų procesorių (simbolių sekos), ir kompiuterinės grafikos sistemų (dvejetainės sekos). Šitie du atvejai turi daug skirtumų ir panašumų. Panašus uždavinio formulavimas ir sekų apdorojimo procedūros, bet skirtingi alfabetai, simbolių kiekis, ir t.t. Šie faktoriai dažnai sąlygoja skirtingus algoritmus.

Pagrindinės operacijos, atliekamos su sekomis - tai posekių paieška ir sekų fragmentų identifikavimas. Posekių paieškos operacijos labai skiriasi nuo nagrinėtų paieškos algoritmų, todėl čia bus nagrinėjamos atskirai. Sekų fragmentų identifikavimo (*pattern matching*) uždavinys gali būti formuluojamas taip: yra  $N$  simbolių ilgio seka ( $N$  yra didelis), ir  $M$  simbolių ilgio fragmentas (*pattern*), reikia rasti visus fragmento pasikartojimus tekste. Pavyzdžiui, rasti visas vietas, kur šioje knygoje sutinkamas fragmentas "duomenų struktūra". Toks formulavimas nėra paprasta simbolio paieška, ir todėl algoritmai gali labai skirtis.

Kaip visuomet, svarbi algoritmų sudėtingumo problema. 1970-ais metais buvo paskelbtas teiginys, kad egzistuoja posekių paieškos algoritmas, kurio sudėtingumas netgi blogiausiu atveju yra  $M+N$ . Vėliau per kelerius metus buvo sugalvoti ir įdiegti konkretūs algoritmai, kurie atitiko teorinius rezultatus. Šie algoritmai nagrinėjami šiame skyriuje.

Daugelis posekių paieškos algoritmų turi įdomią ir retai pasitaikančią savybę - jų teoretiškai vertinamas sudėtingumas paprastai yra blogesnis už praktinį. Pavyzdžiui, brutalios jėgos algoritmo teorinis sudėtingumas yra lygus  $MN$ , o jo kruopštus realizavimas leidžia sumažinti jo sudėtingumą iki  $M+N$ .

Įdomu pastebėti, kad keli plačiai naudojami fundamentalūs ir efektyvūs posekių paieškos algoritmai yra labai paprastai formuluojami (ir šia savo savybe vos ne genialūs), bet buvo pasiūlyti tik prieš kelis metus. Natūralu pagalvoti, kodėl šie algoritmai nebuvo atrasti anksčiau?

### 4.2. Tiesmukiškas (brutalios jėgos) algoritmas

Vienas iš posekių paieškos metodų yra akivaizdus - reikia nuosekliai vieną po kito tikrinti kiekvieną sekos simbolį tol, kol jis sutampa su pirmuoju fragmento simboliu, ir jei taip - lyginti fragmento ir sekos antruosius simbolius, po to trečiuosius, ir t.t. Jei bent vienoje vietoje simboliai nesutampa, reikia pereiti prie sekančio sekos simbolio ir tikrinimą pradėti iš naujo. Jei seka ir fragmentas išdėstyti masyvuose, programa realizuojanti šį algoritmą, būtų tokia:

```
function brutearch: integer;
  var i, j: integer;
  begin i:= 1; j:= 1;
  repeat    if a[i] = p[j]
            then begin i:= i+1; j:= j+1 end
            else begin i:= i - j +2; j:= 1 end;
  until ( j > M ) or ( i > N );
  if j > M then brutearch:= i - M else brutearch:= i
  end;
```

Čia  $i$  ir  $j$  yra atitinkamai sekos ir fragmento nagrinėjamų simbolių rodyklės. Algoritmo efektyvumas labai priklauso nuo konkrečių sekų ir fragmentų savybių. Pavyzdžiui, tekstų apdorojimo metu labai retai pasitaiko, kad vidinis ciklas kartotųsi tiek kartų, kiek numatyta,

dažniausiai jis net neinicijuojamas. Be vidinio ciklo algoritmo sudėtingumas būtų  $M+N$ . Kaip jau buvo minėta anksčiau, algoritmo teorinis sudėtingumas reikalauja  $MN$  lyginimų paieškos metu.

### 4.3. Knuth-Morris-Pratt'o algoritmas

Pagrindinis principas, kuriant šį algoritmą, buvo toks: fragmentas yra žinomas iš anksto ir ši informacija turi būti panaudota. Pavyzdžiui, jei fragmento pirmasis simbolis toliau nesikartoja, o fragmento ir sekos lyginimo metu sutapo  $(j-1)$  simbolių, ir  $j$ -asis nesutapo, tai faktiškai yra žinoma daug daugiau - sekoje nuo tos vietos, kur buvo pradėta tikrinti, galima praleisti  $(j-1)$  simbolį ir pereiti prie  $j$ -ojo lyginimo, nes nė vienas iš šių praleistų simbolių tikrai nesutaps su pirmuoju fragmento simboliu. Todėl brutaliųjų jėgų algoritme rodyklės  $i$  reikšmę galima keisti  $(i+1)$ , o ne  $(i+j+2)$ , kaip anksčiau. Šios idėjos įgyvendinimas ir sudaro nagrinėjamo algoritmo turinį. Be to, remiantis įvairiais papildomais pastebėjimais galima pasiekti, kad rodyklės  $i$  reikšmė niekad nemažėtų.

Programuojant šį algoritmą, yra apibrėžiamas naujas masyvas **next**[1 .. M], kurio elementų reikšmės skaičiuojamos taip:

- **next**[1]:=0;
- jei  $j > 1$ , tai **next**[j] reikšmė yra lygi maksimaliam  $k < j$ , kai pirmieji  $(k-1)$  fragmento simbolių sutampa su paskutiniais  $(k-1)$  simbolių fragmento sekos, susidedančioje iš  $j-1$  elementų.

Funkcija, apskaičiuojanti grįžimo indeksą, būtų tokia:

```
function kmpsearch: integer;
  var i, j: integer;
  begin i:= 1; j:= 1; initnext;
  repeat if (j = 0) or (a[i] = p[j])
    then begin i:= i+1; j:= j+1 end
    else begin j:= next[j] end;
  until (j > M) or (i > N);
  if j > M then kmpsearch:= i - M else kmpsearch:= i;
  end;
```

Algoritmą realizuojanti programa gali būti parašyta taip, kad masyvo **next**[1 .. M] reikšmių skaičiavimas būtų realizuotas pačioje programoje kartu naudojant papildomus programavimo triukus:

```
procedure initnext;
  var i, j: integer;
  begin i:= 1; j:= 0; next[1]:= 0;
  repeat if (j = 0) or (p[i] = p[j])
    then begin i:= i+1; j:= j+1; next[i]:= j; end
    else begin j:= next[j] end;
  until i > M;
  end;
```

Jei fragmentas keičiasi retai ir fiksuotas per visą programos vykdymo laiką, galima tokį masyvą sudaryti iš anksto ir įterpti į programą. Pateikiamas tokio algoritmo realizavimo pavyzdys (fragmentui 10100111):

```
  i:=0;
0: i:= i+1;
1: if a[i] <> '1' then goto 0; i:= i+1;
```

```

2: if a[i] <> '0' then goto 1; i:= i+1;
3: if a[i] <> '1' then goto 1; i:= i+1;
4: if a[i] <> '0' then goto 2; i:= i+1;
5: if a[i] <> '0' then goto 3; i:= i+1;
6: if a[i] <> '1' then goto 1; i:= i+1;
7: if a[i] <> '1' then goto 2; i:= i+1;
8: if a[i] <> '1' then goto 2; i:= i+1;
   search:= i - 8;

```

Ši programa - tai sekos paieškos kompiliatoriaus pavyzdys. Naudojant papildomus pastebėjimus, kuriuos galima realizuoti šiame algoritme, galima pasiekti tokį jo **sudėtingumą**: Knuth-Morris-Pratt'o algoritmas naudoja ne daugiau kaip  $(M+N)$  simbolių lyginimų.

Nors lyginimų skaičiumi šis algoritmas daug efektyvesnis už brutalią jėgą algoritmą, tačiau jo reikšmė iš tikrųjų ne tokia jau didelė. Taikymuose ne taip jau dažnai pasitaiko pasikartojančio teksto pasikartojančių fragmentų paieška. Šis metodas turi kitą privalumą - sekoje simboliai peržiūrimi nuosekliai ir niekada negrįžtama atgal. Todėl tokį algoritmą yra patogu realizuoti situacijose, kai simboliai įvedami vienas po kito, iš šių simbolių yra išrenkami reikalingi fragmentai, tačiau nereikia naudoti papildomos atminties anksčiau buvusiems sekos simboliams įsiminti.

#### 4.4. Boyer-Moore'o algoritmas

Jeigu sekos analizės metu galima laisvai pereiti nuo vienos pozicijos prie kitos ir grįžti į bet kurią poziciją atgal, taikomas greitesnis posekio paieškos metodas, patį fragmentą analizuojant iš dešinės į kairę. Ši idėja gali būti pagrįsta tokiu pavyzdžiu: jei fragmento 10100111 8-a, 7-a ir 6-a pozicijos sutampa su sekos simboliais, bet 5-a pozicija nesutampa, iš karto fragmento analizė perkeliama per 7 pozicijas ir tikrinama nuo 15-os pozicijos. Šitokį sprendimą galima pagrįsti tuo, kad fragmento 6, 7, 8 pozicijos (t.y. fragmento dalis 111) niekur daugiau fragmente nesikartoja, todėl paties fragmento pirmose septyniuose sekos pozicijose negali būti. Aišku, nesikartojančios fragmento dalys masyve **next** turi būti identifikuotos iš anksto. Šis masyvas turi būti modifikuotas skaičiuoti pozicijoms iš dešinės į kairę.

Verta pastebėti, kad turbūt pagrindinis skiriamasis šio algoritmo bruožas yra tai, kad jo veiksmams valdyti yra naudojamas simbolių nesutapimo požymis, ne taip kaip anksčiau - simbolių sutapimo požymis. Programa, realizuojanti algoritmą, būtų tokia:

```

function mischarchsearch: integer;
  var i, j: integer;
  begin i:= M; j:= M; initskiip;
  repeat if a[i] = p[j]
    then begin i:= i-1; j:= j-1 end
    else begin if M-j+1 > skip[index(a[i])]
      then i:= i + M - j + 1 else i:= i + skip[index(a[i])];
      j:= Mp end;
  until (j < 1) or (i > N);
  mischarchsearch:= i+1;
end;

```

**Algoritmo sudėtingumas.** Boyer-Moore posekių paieškos algoritmas visada naudoja ne daugiau kaip  $(M+N)$  simbolių lyginimų. Jam reikia apie  $N/M$  lyginimų, jei alfabetas nėra labai mažas, o fragmentas nėra labai ilgas.

Pastaroji savybė, aišku, yra euristinė. Ji pagrįsta pastebėjimu, kad jei simbolių alfabete yra daug, o fragmentas ne toks jau ilgas, daugelio simbolių arba nebus fragmente iš viso, arba jie pasikartos ne daugiau kaip vieną kartą.

#### 4.5. Rabin-Karp'o algoritmas

Posekių paieškos procedūroms galima pritaikyti ir išdėstymo funkcijų metodus. Tokiu atveju fragmentą reikėtų interpretuoti kaip galimą išdėstymo funkcijos reikšmę. Jei fragmentas yra trumpas, dėstymo algoritmus galima taikyti tiesiogiai, t.y. fragmentą interpretuoti kaip *hashing* adresą visiems galimiems sekos M ilgio posekiams. Jei fragmentas nėra trumpas ir M ilgio posekių analizė atmintyje komplikuota, reikia ir fragmentams, ir posekiams taikyti tinkamą išdėstymo funkciją. Teigiama šios paieškos ypatybė yra ta, kad nereikia atmintyje saugoti pačios išdėstymo lentelės, nes domina tik fragmento ir posekio vienkartinis sutapimas.

Algoritmas, naudojantis *hash* funkciją, formuluojamas paprastai:

- apibrėžiama kokia nors paprastai ir greitai apskaičiuojama hash funkcija, pvz.,  $h(k)=k \bmod q$ , čia  $q$  yra pirminis skaičius;
- apskaičiuojama ir saugoma šios funkcijos reikšmė fragmentui;
- perrenkant visus galimus sekos M ilgio posekius, pradedant nuo 1-os pozicijos, jiems skaičiuojama *hash* funkcijos reikšmė ir lyginama su fragmento reikšme;
- jei abi reikšmės sutapo, tiesiogiai lyginami fragmentas ir posekis, jei ne - analizuojamas kitas posekis.

**Algoritmo sudėtingumas.** Rabin-Karp'o algoritmas, vykusiai parinkus hash funkciją, naudoja apytikriai  $M+N$  lyginimo operacijų, nors jo teorinis sudėtingumas blogiausiu atveju yra  $O(MN)$ .

#### 4.6. Dviejų sekų bendras ilgiausias posekis

Problema, nagrinėjama šiame skyrelyje ir dažnai pasitaikanti praktikoje, gali būti formuluojama taip: reikia išskirti bendrą ilgiausią dviejų sekų posekį (t.y. seką, gautą išmetus tam tikrą kiekį, galbūt nulinį, elementų iš sekos). Šis uždavinys taip dažnai iškyla taikomosiose programose, kad, pavyzdžiui, UNIX operacinėje sistemoje yra net speciali failų lyginimo komanda *diff*, surandanti didžiausią posekį, kuris yra bendras dviems failams, jei jie yra nelygūs. Šioje programoje įrašai yra interpretuojami kaip sekos elementai.

Algoritmas yra formuluojamas aibių duomenų struktūros terminais, o pagrindinės naudojamos operacijos su aibėmis yra šios:

- **FIND(r)** - suranda aibę, kurioje yra elementas  $r$ ;
- **SPLIT(S, T, U, r)** - aibę  $S$  suskaido į dvi aibes  $T$  ir  $U$  taip, kad aibėje  $T$  atsiduria elementai, mažesni už  $r$ , o aibėje  $U$  - didesni už  $r$  ar lygūs jam;
- **MERGE(S, T, U)** - aibes  $S$  ir  $T$  sujungia į vieną ir rašo į aibę  $U$ .

Uždavinio tikslas yra surasti ilgiausią bendrą dviejų simbolių sekų  $A$  ir  $B$  posekį. Algoritmo darbas pradedamas nuo to, kad sekai  $A$  sudaromas aibių rinkinys:

$$PLACES(a) = \{i - \text{numeris} \mid \text{kad simbolis } a \text{ yra } i\text{-oje pozicijoje}\}.$$

Šitą aibių rinkinį galima efektyviai realizuoti, sudarant pozicijų sąrašus kaip tam tikrų pozicijų nuorodų rinkinį. Jei simboliai, sudarantys sekas  $A$  ir  $B$ , yra pakankamai sudėtingi, pvz. failo įrašai (tekstinio failo eilutės), galima šiam tikslui naudoti kokią nors tinkamą išdėstymo (*hash*) funkciją. Aibės  $PLACES$  sudarymo sudėtingumas būtų  $O(n)$ .

Sudarius PLACES, toliau jau tiesiogiai galima rasti ilgiausią bendrąjį posekį. Čia pateikiamas algoritmas sugebės nustatyti maksimalų bendrojo posekio ilgį. Lengva algoritmo modifikacija, reikalinga surasti pačiam posekiui, paliekama skaitytojui.

Algoritmas darbą pradeda, nuosekliai analizuodamas vieną po kito sekos B elementus. Kiekvienam aibės B elementui  $b_j$ ,  $0 < j < m$ , reikia išskirti ilgiausią bendrąjį posekį, kurį turi sekos  $\{a_1, \dots, a_i\}$  ir  $\{b_1, \dots, b_j\}$ . Tam tikslui simbolių pozicijų indeksai  $i$  jungiami į aibes  $S_k$ , kurias sudaro tokios pozicijų  $i$  reikšmės, kad aibių  $\{a_1, \dots, a_i\}$  ir  $\{b_1, \dots, b_j\}$  ilgiausio bendrojo posekio ilgis yra lygus  $k$  ( $k$  kinta nuo 0 iki  $n$ ). Reikia pastebėti, kad aibė  $S_k$  visada susideda iš didėjimo tvarka išsidėsčiusių skaičių, ir skaičiai, esantys aibėje  $S_{k+1}$ , yra didesni už skaičius, esančius aibėje  $S_k$ .

Algoritmas dirba iteratyviai. Tariant, kad visos aibės  $S_k$ , esant sekos B pozicijai  $j-1$ , jau sudarytos, yra nagrinėjama pozicija  $j$  ir jai sudaromos naujos aibės  $S_k$ . Tam tikslui reikia tik modifikuoti jau turimas aibes  $S_k$ . Išskiriama aibė  $PLACES(b_j)$ . Kiekvienam elementui  $r$  iš šios aibės reikia nuspręsti, ar jis galės pailginti ilgiausią bendrąjį posekį. Jeigu taip, ir, be to, jei abu elementai  $(r-1)$  ir  $r$  bus vienoje aibėje  $S_k$ , tada visi elementai  $s \geq r$  turi priklausyti aibei  $S_{k+1}$ . Toks tikrinimas atliekamas modifikuojant aibes taip:

- $FIND(r)$  - norint surasti reikalingą aibę  $S_k$ ;
- Jei  $FIND(r-1)$  nėra lygi aibei  $S_k$ , tai naujas elementas  $b_j$  negali pailginti bendrojo posekio, todėl nieko modifikuoti nereikia ir algoritmas šį etapą turi užbaigti;
- Jei  $FIND(r-1) = S_k$ , yra vykdoma operacija  $SPLIT(S_k, S_k', r)$ , siekiant iš aibės  $S_k$  išskirti elementus, didesnius už  $r$ ;
- $MERGE(S_k', S_{k+1}, S_{k+1})$  - išskirti elementai turi būti perkelti į aibę  $S_{k+1}$ .

Kai visi aibės B elementai yra perrinkti, ilgiausio bendrojo posekio ilgis yra saugomas aibėje  $S_k$ , čia  $k$  - gautas maksimalus skaičius.

Nesunku pastebėti, kad algoritmo darbą galima padaryti efektyvesnį, aibėse  $PLACES(b_j)$  imant analizuojamus skaičius mažėjimo tvarka. Tai gali sumažinti operacijų skaičių.

**Algoritmo sudėtingumas.** Operacijų skaičius gali būti įvertintas asimptotika  $O(p \log n)$ : čia  $n$  yra sekų ilgių maksimumas, o  $p$  - skaičius porų pozicijų, kur sutampa sekų simboliai. Blogiausiu atveju skaičius  $p$  yra lygus  $n^2$ .

## 4.7. Sekų ir failų kompresijos metodai

Iki šiol nagrinėtuose algoritmuose daugiausia dėmesio buvo skiriama procedūrų vykdymo greičiui, ir ne visada kreipiamas dėmesys į papildomos atminties reikalingumą ar panaudojimą. Duomenų kompresija skirta būtent taupyti kompiuterio atmintį - vidinę ar išorinę. Daugelis kompiuterinių failų yra perteklingi, todėl kompresijos metodai įgalina juos suspausti. Šiuos metodus tikslingiausia taikyti tekstiniais failams, rastriniams (kompiuterinės grafikos) failams, taip pat garso kodavimo failams, kuriuose įrašyti dažni pasikartojimai.

Failų kompresijai būdingos tokios charakteristikos: tekstiniai failai sumažėja vidutiniškai nuo 20 iki 50%, dvejetainiai failai - nuo 50 iki 90%. Tačiau kai kurių tipų failams kompresija gali nepadėti, jų tūris kartais gali net padidėti.

Failų ar duomenų kompresija yra prieštaringas reiškinys. Kompiuterinė atmintis pastoviai pinga, atsiranda vis naujos technologijos, kai diskinė atmintis padidėja dešimtimis ir šimtais kartų, dažnai paprasčiau saugoti ar naudoti failus nesuspaustus. Kita vertus, kaip tik dėl atminties technologijos pasikeitimų, išrinkimo greičio antrinėje ar tretinėje atmintyje padidėjimo suspaustos informacijos saugojimas tampa vis labiau patogesnis ir efektyvesnis ir plačiau naudotinas.

### 4.7.1. Vienodų simbolių sekų kodavimas



Perteklingų simbolių sekos, turinčios paprasčiausią galimą struktūrą ir gan dažnai pasitaikančios - tai ilgos paeiliui einančių vienodų simbolių sekos. Jas galima keisti į skaičių ir simbolių kombinacijas: pvz., vietoj AAAAAAA galima rašyti 7A - seka akivaizdžiai sutrumpėja. Šitoks metodas be didelių pakeitimų gerai tinka situacijai, kai tekste yra vien tik raidės ir jame nėra tokio fragmento 7A ar panašaus. Jei tekste yra ir raidžių, ir skaičių, metodą galima ir reikia modifikuoti. Šį metodą paprasta ir lengva taikyti dvejetainiams failams - skaičiai 0 ir 1 yra alternuojantys, todėl vienodų simbolių sekas galima tiesiog užrašyti skaičiais:

0000000111111000111111111 galima keisti skaičių seka 7639

Šitoks kompresijos metodas patogus vadinamiesiems „*bitmap*” tipo failams (rastriniams piešiniais, piešiamiems šriftams, ir pan.), failo suspaudimo santykis gali būti labai didelis. Neigiama jo ypatybė - metodas tinka tik tam tikriems specializuotiems failams.

Kitas elementarus būdas - vadinamųjų *escape* (išeities) sekų naudojimas. Koduojamos sekos pradžioje rašomas specialus simbolis, o po jo dvi ar kelios raidės, išreiškiančios sekos turinį. Pavyzdžiui, simbolių seką AAA reikia koduoti seka QCA, kuri turi būti suprantama taip: raidė Q yra specialus *escape* simbolis, raidė C atitinka skaičių 3, nes tai trečia raidė alfabete (3 yra koduojamo simbolio pasikartojimų skaičius), o raidė A - koduojamą simbolį. Kadangi *escape* seka visada užima ne mažiau kaip 3 pozicijas, varta pradėti koduoti sekas, ne trumpesnes kaip 4 simbolių. Jei pats *escape* simbolis yra tekste, jam galima naudoti koduojančią kombinaciją Q[tarpas], t.y. nulinio ilgio seką. Labai ilgoms vienodų simbolių sekoms galima naudoti pasikartojančias *escape* sekas. Reikia pastebėti, kad taikymuose *escape* sekos yra naudojamos ne tik sekų ar failų kompresijai realizuoti, bet ir vykdomoms simbolių sekoms (t.y. komandoms) nuo informacinių simbolių atskirti, pvz., failuose, kurie į spausdintuvą atmintį įrašo šriftus ar pan.

#### 4.7.2. Kintamo ilgio kodavimas

Sekų kompresijai yra taikoma ir daug kitokių metodų. Viena iš labiausiai paplitusių idėjų, kurios realizavimas sugeba sutrumpinti sekos ilgį ir kurios pagrindu sukurta daug failų kompresijos programų, yra tokia: kuo dažniau simbolis ar simboliai kartojasi sekoje, tuo trumpesnis turi būti jam(-iems) konstruojamas kodas. Tada galima tikėtis, kad ir simbolių sekos ilgis bus trumpesnis. Įvairioms galimybėms ir apribojimams apžvelgti, varta pateikti detalų ir paprastą pavyzdį.

Turint tekstinius failus, simbolių alfabetas susideda iš 26 lotyniškų raidžių, todėl kiekvienam simboliui galima priskirti penkių bitų ilgio kodą. Užkoduotos sekos AAAAABBBCCD ilgis tokiu atveju bus 55 bitai. Jei raidėms būtų priskirti kodai pagal jų dažnumus: A-0, B-1, C-00, D-01, seka transformuotųsi į dvejetainę seką 00000111000001, kurios ilgis bus tik 14 bitų. Nors sekų ilgiai labai sutrumpėja, tačiau šiuo atveju iškyla kita - vienareikšmio dekodavimo problema. Ją galima spręsti keliais būdais, iš kurių vienas - įvedant tarp kodų specialius skyriklius. Tarkime, jei skyriklis užims vieną bitą (paprastai jis yra daug ilgesnis), užkoduota seka pailgės 10 bitų, t.y. bus lygi 24 bitams, ir ji vistiek bus trumpesnė už pradinę seką. Atrodytų, šitoks kodavimas yra efektyvus. Tačiau analizuojant situaciją iš esmės, iškyla daug problemų. Šitoks kodavimo būdas yra nestabilus, t.y. šitaip užkoduotoje sekoje pametus nors vieną bitą (taip dažnai atsitinka telekomunikacijų kanaluose), bus dekodotas visiškai kitas tekstas. Šį metodą sunku realizuoti (iškyla daug techninių detalių) kompiuterių programose.

Siūlant išeitį iš tokios situacijos ir įvertinant vienareikšmio kodavimo svarbą, daugelyje algoritmų buvo sėkmingai panaudota vadinamoji prefikso taisyklė: koduojama turi būti taip, kad vienas kodas negalėtų būti kito kodo pradžia. Esant tokiam kodavimui, nereikės jokių skyriklių, visi koduojančio alfabeto simboliai bus lygiaverčiai, o tai žymiai palengvins algoritmo realizavimą.

Toks principas ir kodavimo būdas buvo pasiūlyti Huffman'o (tariama Hafmen) 1952-ais metais ir yra žinomas jo vardu. Šis algoritmas yra realizuotas labai daugelyje kompiuterinių programų, programų sistemų, techninėje įrangoje (dažnai tokiose telekomunikacijos prietaisuose,

kaip faksas, modemas ar pan.). Toliau bus nagrinėjami šio algoritmo principai ir jo realizavimo detalės.

Tarkime, kad simbolių sekos susideda iš lotyniškų raidžių, kurias reikia perkoduoti į dvejetainių simbolių sekas. Pirmas algoritmo žingsnis - paskaičiuoti išeities teksto simbolių dažnumus. Programa, realizuojanti šią procedūrą, atrodytų taip:

```
for i:= 0 to 26 do count[i]:= 0;
for i:= 1 to M do
    count [index (a [i] ) ]:= count [index (a [i] ) ] + 1;
```

Šioje programoje raidžių kodai naudojami kaip masyvo indeksai. Žinant teksto raidžių dažnumus, galima konstruoti pačius dvejetainius kodus. Konceptualiai, tačiau nekonstruktyviai tai galima aprašyti taip: visas raides reikia suskirstyti į dvi grupes taip, kad vienos grupės raidžių dažnumų suma būtų lygi (arba apytikriai lygi) kitos grupės raidžių dažnumų sumai. Visų vienos grupės raidžių kodo pradžios simbolis turi būti nulis, o kitos grupės raidžių - vienetą. Po to kiekvieną iš šių grupių reikia vėl perskirti į dvi grupes ir taikyti tą pačią procedūrą kiekvienai iš jų. Šį procesą reikia tęsti tol, kol grupėje bus bent dvi raidės.

Programuojant šį kodavimo procesą, reikia turėti algoritmą, įgalinantį efektyviai skirstyti raides į grupes. Pasirodo, kad tai galima pasiekti naudojant skaitmeninėje paieškoje išdėstytos *trie* struktūros principus, kuriais remiantis gaunamas efektyvus tiesinio sudėtingumo algoritmas.

Raides (kurių yra  $N$ ) reikia koduoti konstruojant dvejetainį medį iš apačios į viršų pagal jų dažnumus. Tam tikslui visos raidės yra išdėstomos apačioje, būsimo medžio lapuose. Kiekvienai raidei priskiriama viršūnė, šalia kurios rašomas dažnumas. Po to išrenkamos dvi viršūnės su mažiausiais dažnumais ir joms priskiriama nauja viršūnė, jos pačios tampa šios viršūnės kairiuoju ir dešiniuoju sūnumis, o naujai viršūnei yra priskiriamas suminis jos sūnų dažnumas. Toliau turint seką viršūnių iš  $(N-1)$ -os viršūnės vėl atrenkamos dvi viršūnės su mažiausiais dažnumais ir jos padaromos sūnumis naujos viršūnės, kartu jai prirašant jos sūnų dažnumų sumą. Taip turi būti tęsiama tol, kol bus gauta tik viena viršūnė. Tokio skaidymo metu kodai kiekvienai iš raidžių priskiriami automatiškai, kai dvi viršūnės tampa naujos viršūnės sūnumis, tada viena iš briaunų (kairioji) gauna kodą 0, o kita briauna (dešinioji) - 1. Raidės kodas bus lygus bitų sekai, kuri gaunama einant vienintele šaka iš medžio šaknies į raidės lapą.

#### **Kodo savybės:**

- užkoduoto pranešimo ilgis bus lygus vidutiniam pasvertam šakų ilgių vidurkiui Huffman'o dažnumų medyje;
- bet kuris kitas dažnumų medis duos kodo ilgį didesnį, negu kad Huffman'o medis.

## 5. Daugiamačių duomenų struktūros ir algoritmai

Šiame skyriuje pateikiamos duomenų struktūros ir algoritmai, būdingi geografinėms informacinėms sistemoms (*GIS*), kompiuterinės grafikos ir kompiuterinio konstravimo sistemų (*CAD*) vaizdų analizei, daugiaterpės duomenų bazėms (*multimedia DB*) ir labai didelėms duomenų bazėms (*VLDB*). Jie papildo ir apibendrina ankstesniuose skyriuose nagrinėtus algoritmus tekstiniais duomenimis apdoroti.

Daugiamačiai duomenys skirtinguose taikymuose yra skirtingai suvokiami. Sudėtingos integroschemų kompiuterinio konstravimo sistemos (*VLSI CAD*) remiasi daugiausia dvimačiais arba išsluoksniuotais dvimačiais duomenimis, kurių bazinius elementus sudaro stačiakampiai, daugiakampiai, orientuoti pagal koordinačių ašis. Tipinės tokių duomenų operacijos yra sankirta, sąjunga, trasavimas. Kartografinės sistemos irgi naudoja dvimačius duomenis, kurių baziniai elementai yra taškai, tiesės, uždaros sritys plokštumoje. Bazinės operacijas sudaro erdvinė paieška, sričių sankirta, atstumų skaičiavimas. Mechaninėse *CAD* sistemose duomenys yra trimačiai, jose operuojama su kietais kūnais erdvėje, o duomenų dėstymo formatai gali būti labai įvairūs.

Dažnas terminas yra ir vaizdų duomenų bazės. Jose paprastai laikomi neanalizuoti vaizdiniai duomenys, pvz., rentgeno nuotraukos, palydovinės nuotraukos, ir pan., kurių pagrindu vėliau būna apskaičiuojamos tam tikros charakteristikos. Kompiuterio atmintyje tokie vaizdiniai yra laikomi rastriniuose failuose. Metodai, analizuojantys vaizdinius duomenis, sėkmingai taikomi ir kitose srityse, pvz., infraraudoniesiems sensoriniams signalams arba garsams. Tuo tarpu daugiamačių duomenų bazės kaupia ir apdoroja detalią informaciją apie objektus, jų savybes, vietą erdvėje ir pan., o duomenys jose yra laikomi vadinamaisiais vektoriniais formatais.

Detaliau apibūdžiant daugiamačius duomenis, galima išskirti tokias jų savybes:

- daugiamačiai duomenys yra sudėtingos struktūros - jų objektai kartais susideda iš daugelio tūkstančių taškų, įvairiai pasiskirsčiusių erdvėje, jų negalima atvaizduoti vienoje fiksuotos struktūros reliacinėje lentelėje;
- daugiamačiai duomenys yra dinaminiai - įterpimas ar išmetimas kaitaliojasi su papildymo operacijomis, ir duomenų struktūros turi būti pritaikytos tokiems procesams;
- daugiamačių duomenų bazės yra didelės - paprastai yra operuojama gigabaitiniais atminties tūriais;
- daugiamačiai duomenys neturi nusistovėjusios standartinių algebrinių operacijų ar specifikacijų aibės - manipuliacijos su duomenimis labai priklauso nuo dalykinės srities;
- daug daugiamačių duomenų operacijų nėra uždaros - jų vykdymo metu galima gauti figūras, visiškai kitokias negu pradinės (dviejų daugiakampių sankirtoje galima gauti paskirus taškus, kelis daugiakampius, kitokias figūras);
- daugiamačių duomenų bazės yra lėtesnės negu įprastos (jų operatoriai paprastai reikalauja daugiau kompiuterio operacijų negu reliaciniai ar kitokie operatoriai).

Išvardytos įvairialypės ir sudėtingos duomenų savybės neleidžia pateikti vieningos tokių bazių teorijos. Viena iš svarbių grupių, išskiriančių manipuliacijas su daugiamačiais duomenimis, yra vadinamieji geometriniai operatoriai. Pagrindiniai iš jų yra erdviniai paieškos operatoriai, kurių efektyvi realizacija reikalauja specialių fizinio lygio algoritmų ir struktūrų. Šie operatoriai paieškai naudoja ne tik skaitinius ar simbolinius, bet ir kitokius atributus, pvz., elemento vietą erdviniam objekte.

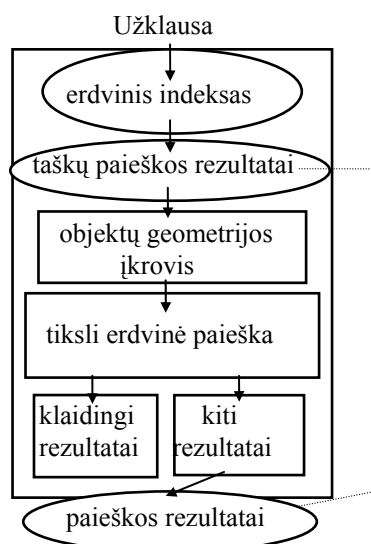
Anksčiau nagrinėtų paieškos metodų negalima tiesmukiškai panaudoti erdvinei paieškai, nebent tik išimtiniais atvejais. Viena iš pagrindinių kliūčių - erdviniams duomenims nėra tokių vieningų tvarkinių, kurie išsaugotų nuotolius tarp objektų. O visi anksčiau nagrinėti metodai remiasi tokiais tvarkiniais. Todėl norint panaudoti tokius metodus erdviniams duomenims, reikėtų daugiamačius objektus bijektyviai atvaizduoti į vienmačius taip, kad atstumai tarp objektų prieš

atvaizdavimą ir po jo liktų nepakitę. Konkrečioje situacijoje rasti tokį atvaizdavimą ne visada pavyksta.

Manipuliuojant su erdviniais duomenimis, kreipties metodai, atitinkantys naudojamas struktūras ir algoritmus, turi tenkinti reikalavimus, kylančius iš erdvinių duomenų savybių:

1. **Dinamiškumas.** Įterpiant ar išmetant elementus, kreipties metodai turi padaryti visus su tuo susijusius pakeitimus.
2. **Antrinės ir tretinės atminties valdymas.** Erdvinių duomenų tūriai tokie dideli, kad jų neįmanoma laikyti vidinėje atmintyje. Kreipties metodai turi laisvai valdyti antrinę ir tretinę atmintį.
3. **Platus operacijų spektro palaikymas.** Kreipties metodai turi vienodai efektyviai vykdyti visas operacijas (vykdymo sudėtingumas visoms operacijoms turi būti tas pats).
4. **Nepriklausomybė nuo duomenų įvedimo.** Kreipties metodai turi būti efektyvūs ir laukiamu, ir blogiausiu atveju.
5. **Paprastumas.** Painūs kreipties metodai, turintys daug išskirtinių atvejų, dažnai nėra pakankamai patikimi, nes jų realizacijoje tikėtinos klaidos.
6. **Invariantiškumas.** Kreipties metodai turi gerai adaptuotis prie augančio duomenų kiekio.
7. **Efektyvumas.** Kadangi erdvinės paieškos operacijos naudoja daug procesoriaus laiko ir daug I/O operacijų, kreipties metodai turi minimizuoti ir kreipinių į antrinę atmintį skaičių, ir procesoriaus naudojimo laiką.

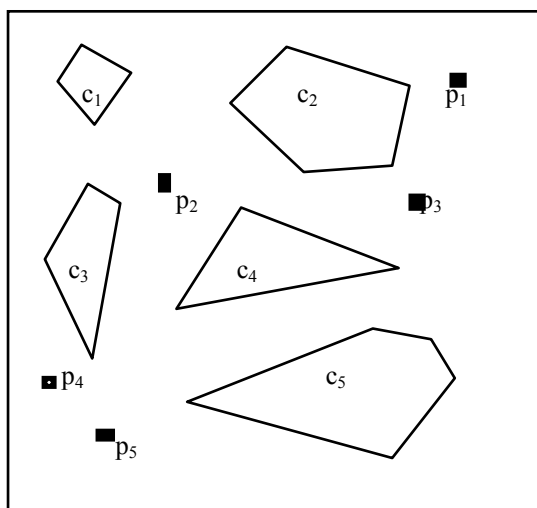
Daugiamačių duomenų kreipties metodai, realizuojantys paiešką, sudaro didelę operacijų klasę. Verta išskirti taškinius (*point access methods*) ir erdvinius kreipties metodus (*spatial access methods*). Taškiniai metodai skirti erdvinei paieškai taškinėse duomenų bazėse, kuriose taškai gali būti dviejų ar daugiau matavimų, bet be erdvinių charakteristikų. Erdviniai kreipties metodai, be kitų, valdo tokius objektus, kaip tiesės, daugiakampiai, daugiamačiai briaunainiai ir pan. Papildomai dar galima išskirti nedidelę klasę metodų, kurie pritaikyti erdviniams duomenims, bet nevaldo išorinės atminties (vadinamieji vidinės atminties metodai). Tačiau šie kreipties metodai yra tik vienas žingsnis erdvinės paieškos procese. Taškinių ir erdvinių kreipties metodų santykį tarp jų iliustruoja 21 pav.



21 pav. Daugiamačių duomenų paieška

## 5.1. Vidinės atminties struktūros

Tolesniam ir pilnesniam temos nagrinėjimui tikslinga susikurti modelinį uždavinį. Tarkime, kad stačiakampėje geometrinėje srityje turime penkis daugiakampius  $r_i$  ir penkis taškus  $p_i$ . Daugiakampiams gali atstovauti jų centroidai  $c_i$  (arba minimalūs gaubiantys stačiakampiai) (22 pav.).

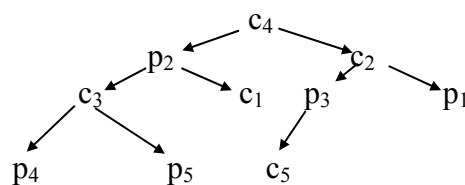
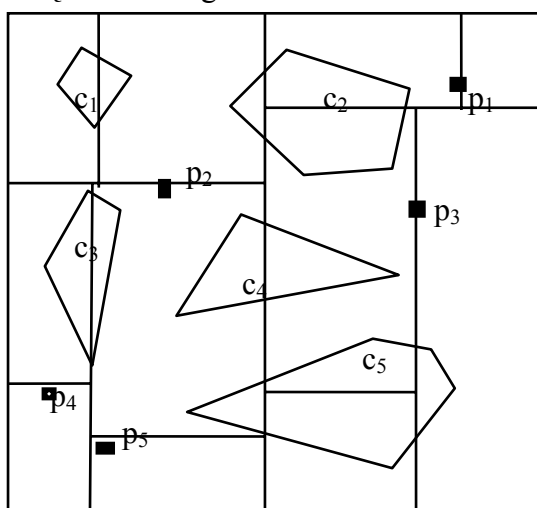


22 pav. Modelinis pavyzdys.

### 5.1.1. K-D-medžiai

Viena iš labai paplitusių erdvinės paieškos struktūrų yra vadinamieji **k-d-medžiai**. Tai dvejetainės paieškos medžiai, fiksuojantys rekursyvų geometrinės erdvės skaidymą ( $d-1$ ) matmenų hiperplokštumomis. Šios plokštumos yra lygiagrečios su koordinatinių plokštumomis, o jų orientacija kiekvienu skaidymo atveju yra keičiama viena iš  $d$  galimų krypčių (nustatyta seka). Kiekviena plokštuma turi turėti bent vieną duomenų tašką, kuris ir naudojamas plokštumai medyje žymėti. Be to, kiekvienai vidinei tokiu būdu gaunamo medžio viršūnei priskiriama diskriminanto funkcijos reikšmė, kuri naudojama paieškos metu. Paieškos ir įterpimo tokioje struktūroje operacijos atliekamos tiesiogiai, be komplikuočių alternatyvų. Tačiau išmetimo operacija gali būti sudėtingesnė, nes kartais reikia partvarkyti visą pomedį, prasidedantį išmetamoje viršūnėje.

Dvejetainis medis, atitinkantis pavaizduotą 22 pav. geometriją, parodytas 23 pav. Kadangi medžio viršūnės turi atitikti taškus, plokštumas identifikuoja daugiakampių centroidai ir taškai. Pirmoji kertanti plokštuma eina vertikaliai per centroidą  $c_4$ , kuris saugomas medžio šaknyje. Kitos dvi plokštumos yra horizontalios ir eina per taškus  $p_2$  ir  $c_2$ , kurie yra šaknies sūnūs. Savo ruožtu šių sūnų sūnūs bus gaunami skaidant stačiakampį vertikaliomis plokštumomis, ir t.t.



23 pav. Modelinio uždavinio k-d-medis.

Aišku, kad k-d-medžiai yra paprasta ir efektyvi erdvinės paieškos struktūra, įgalinanti greitai rasti ar apibrėžti taškus ir kitas geometrines figūras erdvėje. Tačiau jos blogybės yra tai, kad medžio forma labai priklauso nuo taškų įterpimo tvarkos ir duomenų taškai išmėtomi po visą medį. Buvo pasiūlytos kelios šio medžio modifikacijos, išvengiančios šių blogių algoritmui atvejų.

Viena iš jų - **adaptyvūs k-d-medžiai**. Jose skaidančias plokštumas siūloma parinkti taip, kad abiejose pusėse būtų apytikriai po vienodą kiekį elementų, kartu atsisakant taisyklės, kad plokštumos turi turėti duomenų taškus. Tokiame medyje duomenys yra talpinami medžio lapuose. Vidinės viršūnės žymimos skaidančios plokštumos dimensija ir koordinatėmis, o skaidymas vyksta tol, kol kiekviename poerdvyje lieka tik po vieną tašką. Tačiau adaptyvus medis nėra visiškai dinaminė struktūra. Jį sunku laikyti subalansuotą, kai elementai dažnai įterpiami ir išmetami. Geriausia, kai duomenų papildymai žinomi iš anksto arba yra reti.

Kitas k-d-medžių patobulintas variantas - **bin-medžiai**. Ši struktūra siūlo rekursyviai skaidyti erdvę į d-mačius lygius briauninius tol, kol kiekviename iš jų liks po vieną elementą. Jos privalumas tas, kad visada tiksliai žinomas skaidančių plokštumų išsidėstymas ir galima su jomis manipuluoti, pvz., optimizuoti jų parinkimą priklausomai nuo duomenų. Tačiau norint šį medį išlaikyti adaptyviai prisitaikantį prie duomenų reikia daug daugiau pastangų.

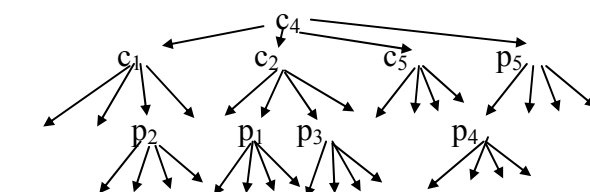
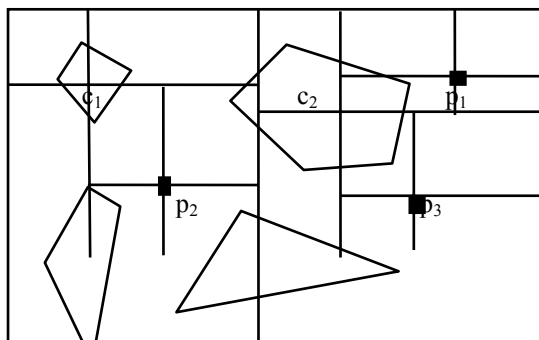
### 5.1.2. BSP-medžiai

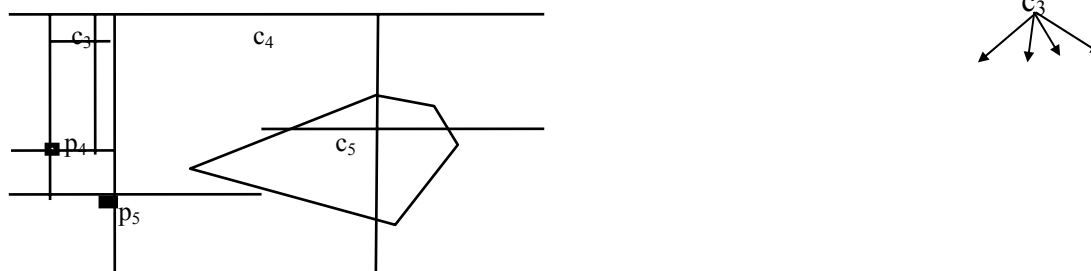
Tai, kad geometrinę erdvę skaidančios hiperplokštumos yra lygiagrečios su koordinatėmis, apriboja struktūros efektyvumą. Todėl buvo pasiūlytas nemažas kiekis metodų, leidžiančių skaidyti erdvę bet kuriomis kryptimis. Galbūt žinomiausias iš tokių metodų - BSP-medžiai (*binary space partitioning*). Kaip ir k-d-medžiai, BSP-medžiai yra dvejetainiai, atitinkantys rekursyvų erdvės skaidymą į poerdvius (d-1)-matėmis plokštumomis. Kiekvienas poerdvis yra skaidomas individualiai, nepriklausomai nuo kitų poerdvių ir nuo to, kaip jis buvo gautas. Skaidančių plokštumų parinkimas priklauso tik nuo taškų, esančių poerdvyje, išsidėstymo. Poerdviai skaidomi tol, kol juose esančių taškų skaičius viršija tam tikrą slenkstį. BSP-medžiuose kiekvieną skaidančią plokštumą atitinka vidinė viršūnė, o poerdvius - lapai. Lapai saugo nuorodas į tuos taškus, kurie yra poerdvyje.

Paieškos medyje metu taško koordinatės įterpiamos į šaknį ir nustatoma, kurioje hiperplokštumos pusėje yra taškas, po to pereinama į atitinkamą sūnų, ir operacija kartojama. Kai tokiu būdu pasiekiamas lapas, ieškomasis taškas yra lyginamas su kiekvienu tašku lapui atitinkamame puslapyje. Tokiu būdu surandama (arba nesurandama), ar taškas yra geometrinėje erdvėje. BSP-medžiai lengvai adaptuojasi prie taškų pasiskirstymo duomenų bazėje pobūdžio. Juos lengva pritaikyti ir ranginei paieškai. Tačiau BSP-medžiai turi ir neigiamų savybių. Paprastai šie medžiai yra nebalansuoti, jie kartais turi labai gilius vienšakius pomedžius, kurie pablogina algoritmų efektyvumą. Be to, BSP-medžiams saugoti reikia daugiau atminties, palyginus su k-d-medžiais.

### 5.1.3. Ketvirtainiai medžiai

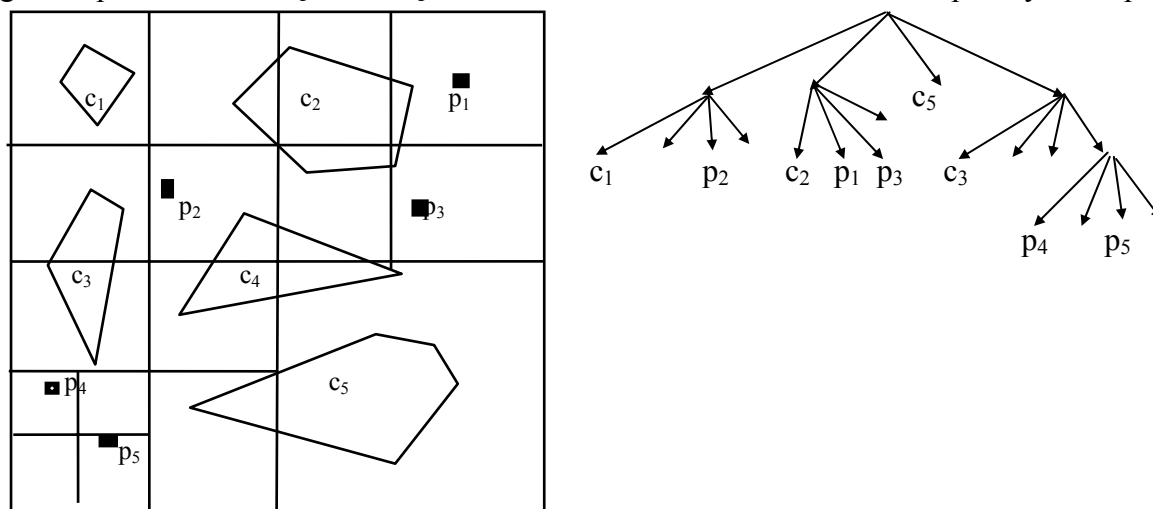
Ketvirtainiai medžiai (angliškai *quadtrees*), nors ir artimi k-d-medžiams, nėra dvejetainiai. Jei jie konstruojami d-matei erdvei, kiekviena viršūnė turi  $2^d$  sūnų. Jų pavadinimas kilęs iš dvimatės plokštumos, ir šiuo atveju kiekviena vidinė viršūnė turi po keturis sūnus. Nors ketvirtainiai medžiai turi daug variantų, pagrindinė idėja taikoma visiems iš jų: d-matė geometrinė erdvė yra skaidoma (d-1)-matėmis lygiagrečiomis su koordinatėmis plokštumomis į  $2^d$  poerdvius; pačiam skaidiniui priskiriama medžio viršūnė, o kiekvienam poerdviui - po sūnų; visiems šiems poerdviams nustatytas tvarkinys (kokia tvarka juos galima perrinkti), nurodantis ir viršūnės sūnų numeraciją. Ketvirtainis medis, atitinkantis modelinį uždavinį, parodytas 24 pav. Jis yra dar vadinamas taškiniu ketvirtainiu medžiu (nes erdvės skaidymo viršūnės parenkamos duomenų bazės taškuose).





24 pav. Modelinio uždavinio taškinis ketvirtainis medis.

Kitas paplitęs ketvirtainių medžių variantas yra vadinamieji sektoriniai ketvirtainiai medžiai (*region quadtree*). Jų viršūnės nepriklauso nuo duomenų bazės taškų, o kiekviena jų atitinka reguliarių erdvės skaidymą į sektorius, sutvarkytus pagal kokį nors tvarkinį. Duomenys yra saugomi lapuose. Modelinį uždavinį atitinkantis sektorinis ketvirtainis medis parodytas 25 pav.



25 pav. Modelinio uždavinio sektorinis ketvirtainis medis.

Paieškos požiūriu ketvirtainiai medžiai yra nesubalansuoti ir todėl kartais gali būti neefektyvūs. Tačiau jie labai dažnai pasitaiko, kai iš rastrinio vaizdo (piešinio) yra gaunama vektorinė struktūra (pvz., geografinėse informacinėse sistemose).

## 5.2. Antrinės atminties struktūros

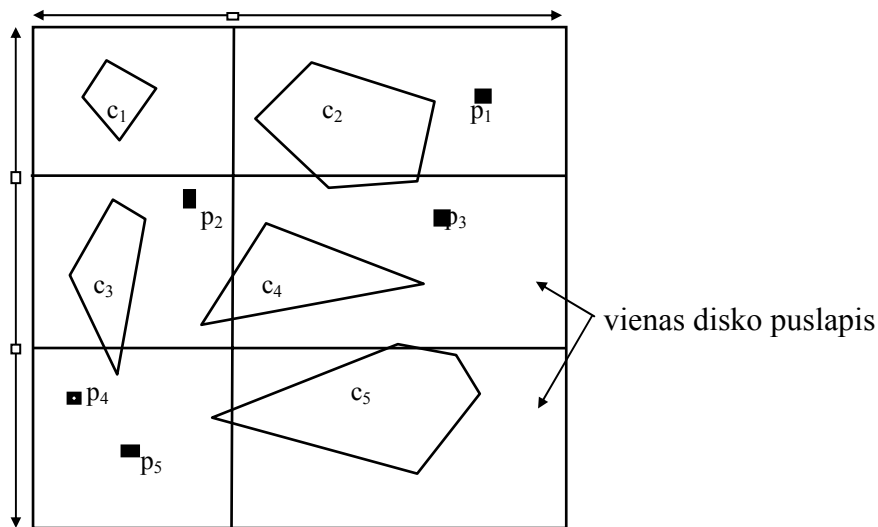
Daugiamatės duomenų struktūros, nagrinėtos ankstesniame skyrelyje, nepritaikytos duomenims, esantiems išorinėje kompiuterio atmintyje, valdyti. Tuo tarpu praktiškai duomenų negalima rašyti vien tik vidinėje atmintyje. Jei šias struktūras tiesmukiškai taikytume ir išorinei atminčiai, algoritmai būtų labai neefektyvūs.

Duomenų struktūras, arba kreipties metodus, valdančius daugiamatius duomenis, galima skirstyti į taškinis ir sektorinius (arba sritis). Duomenų bazių taškai (arba pakankamai mažos sritys) yra skirstomi į vadinamuosius buketus, kurie parenkami taip, kad kiekvieną iš jų galima rašyti į atskirą diskinės atminties puslapį. Be to, šių buketų atributai turi būti pritaikyti prie kompiuterio operacinės sistemos ypatumų. Vizualiai buketai gali būti vaizduojami tam tikromis, dažnai paprastomis, geometrinėmis figūromis, pvz., stačiakampiais.

### 5.2.1. Daugiamatiai išplėstinio dėstymo metodai

Nors dviejų ar daugiau matavimų duomenys paprastai neturi kokio nors tvarkinio, išsaugančio atstumus tarp jų, tačiau literatūroje yra daug bandymų įvesti tokį sutvarkymą, bent jau

iš dalies. Tikimasi, kad tai gali žymiai sumažinti kreipčių į diskinę atmintį skaičių. Aišku tokie bandymai yra euristiniai, jie remiasi noru objektus, esančius geometrinėje erdvėje arti vienas kito, ir antrinėje atmintyje dėstyti arti vienas kito. Vienas iš populiariausių metodų yra išplėstinis dėstymas, apibendrintas daugiamačiam atvejui - vadinamasis *grid file*. Yra daug jo modifikacijų, tačiau visos jos remiasi viena idėja - geometrinės erdvės skaidymu į gardeles. Gardelės yra gaunamos skaidant erdvę plokštumomis, lygiagrečiomis su koordinatėmis, tačiau jos gali būti nevienodo dydžio ir formos. Gardelės gali būti sujungtos į buketus taip, kad vienas buketas atitiktų vieną atminties puslapį. Kaip ir išplėstinio dėstymo algoritme, gardelėms sudaroma rodyklė. Ji irgi turi būti pritaikyta saugoti išorinėje atmintyje. Rodyklės pagrindą sudaro koordinatinių ašių dalijimo intervalai, kuriais remiantis formuojama vadinamoji liniuotė, pritaikyta saugoti vidinėje atmintyje. Modeliniam uždaviniui sudarytas *grid file* parodytas 26 pav. Jame vienas skaidymo metu gautas stačiakampis atitinka vieną disko puslapį.



26 pav. Modelinio uždavinio *grid file*.

Paieška *grid file* struktūroje atliekama taip: iš pradžių randamas ieškomojo taško koordinatinių išsidėstymas liniuotėje ir nusprendžiama, kurį gardelių rodyklės puslapį diske reikia skaityti; po to analizuojamas nuskaitytas puslapis ir nustatoma, kurį gardelės puslapį diske reikia skaityti, kad galutinai nuspręsti, ar taškas yra geometrinėje erdvėje. Todėl paieškos metu reikia (mažiausiai) du kartus kreiptis į išorinę atmintį - tokia yra paieškos operacijos kaina. Paieškos operacija reikalinga ir įterpiant naują elementą į struktūrą. Jos metu surandama gardelė, o teisingiau disko puslapis, į kurį įterpiamas elementas. Jei puslapis nepilnas, įterpimo operacija papildomų veiksmų nereikalauja. Tačiau jei puslapis pilnas, į liniuotę įvedama nauja plokštuma, duomenų buketas yra skaidomas šia plokštuma į du buketus ir atitinkamas disko puslapis perrašomas į du puslapius. Šiuo atveju reikia pakeisti ir gardelės rodyklę. Visi šie veiksmai reikalauja papildomų I/O operacijų.

Tačiau dar daugiau struktūros ir disko pertvarkymų gali prireikti išmetimo operacijos atveju. Jei pašalinus elementą iš puslapio jame likęs informacijos kiekis tampa mažesnis už leistiną, puslapį reikia sujungti su kitu gretimu. Tačiau tai galima tik tada, kai ir gretimame informacijos kiekis yra mažesnis už leistiną, ir norint tai pasiekti gali tekti pertvarkyti visą geometrinės erdvės išdėstymą diske. Būtent tokiam pertvarkymui efektyviai realizuoti yra pasiūlyti papildomi algoritmai ir struktūros (vadinamieji *neighbor system* ir *buddy system*).

Išplėstinio dėstymo idėja daugiamačiu atveju apibendrinta ir kitaip - algoritmuose *EXCELL*, *BANG file*, dviejų lygių *grid file*, dvynukų *grid file*, kituose metoduose. *EXCELL* (žodžių *Extendible CELL* sutrumpinimas) metodu geometrinę erdvę siūloma skaidyti į reguliarias vienodo dydžio gardeles, o įterpimo ar išmetimo atveju visas jas dvigubinti ar perpus mažinti, atitinkamai didinant ar mažinant skaidymo rodyklę. Greitai paieškai rodyklėje šis metodas siūlo naudoti hierarchines struktūras. Kad hierarchijos nebūtų per daug gilia, jos norimu momentu gali būti



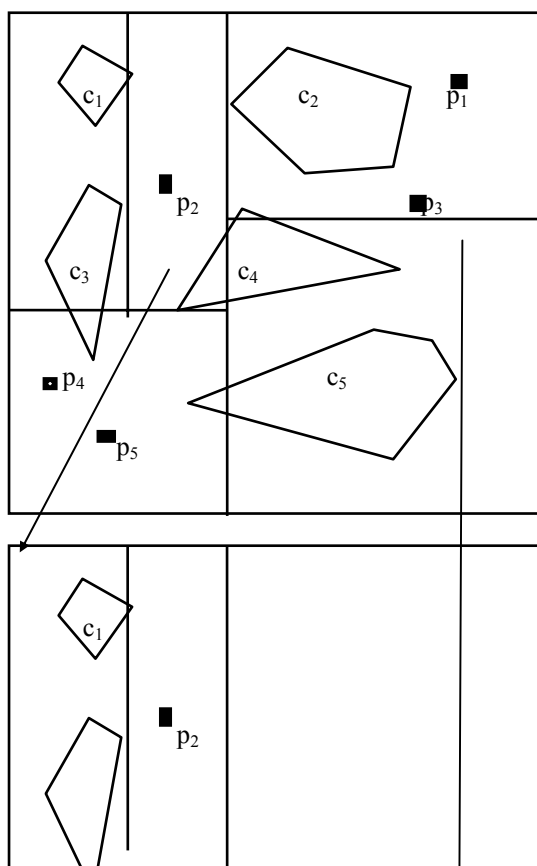
užbaigiamos specialiai konstruojamais puslapiais (vadinamaisiais *overflow pages*). Dviejų lygių *grid file* metodas gardelės rodyklei siūlo taikyti gardelės konstrukciją, t. y. rodyklę interpretuoti kaip geometrinę erdvę ir ją skaidyti į gardeles. Metodas *BANG file* (žodžių *Balanced And Nested Grid* sutrumpinimas) gardeles siūlo konstruoti bet kokio tinkamo daugiakampio formos, be to, leidžia jų sankirtai būti netuščiai. Netuščios sankirtos gali būti naudojamos hierarchiniams medžiams, priskirtiems gardelių rodyklei, išbalansuoti. Dvynukų *grid file* metodas siūlo geometrinę erdvę skaidyti į dvi skirtingas gardelių sistemas, susiejant jas taip, kad elementų įterpimo ar išmetimo atveju duomenis būtų galima dinamiškai perkelti iš vienos sistemos į kitą.

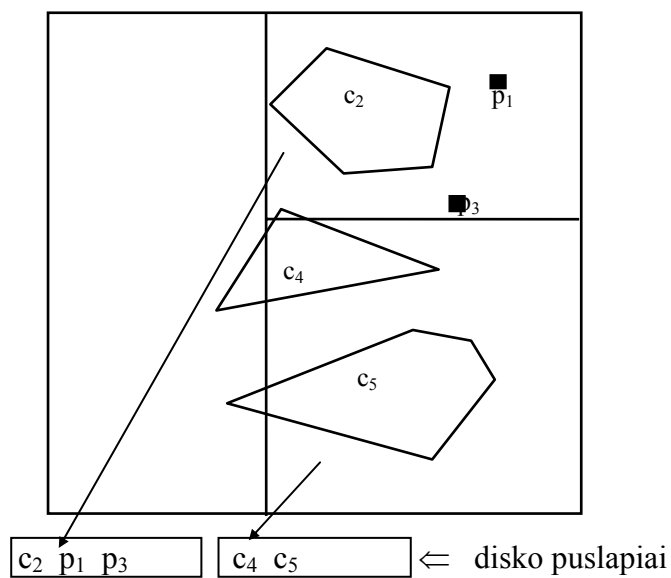
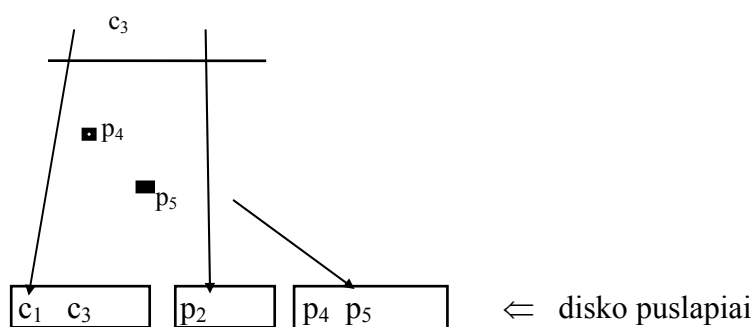
Be išplėstinio dėstymo daugiamačių apibendrinimų, kurie čia buvo paminėti, dar yra daug kitokių metodų, besiremiančių išdėstymo funkcijomis. Tai ir daugiamatis tiesinis išdėstymas, ir vadinamieji *buddy*-medžiai, ir daugelis kitų. Visi jie turi teigiamų ir neigiamų savybių. Vienuose iš jų netolygiai pasiskirstęs skirtingų operacijų sudėtingumas, kiti labai sudėtingi programuoti, tretieji naudoja per daug papildomos atminties. Tačiau kiekvienas iš jų yra tinkamas tam tikriems duomenims ir taikymams. Todėl prieš nusprendžiant, kuriuos iš jų panaudoti, o kuriuos atmesti, turi būti atlikta detali taikomojo uždavinio analizė.

### 5.2.2. Daugiamačiai hierarchiniai metodai

Hierarchiniai metodai remiasi dvejetainiais ar daugiamačiais medžiais, jie kitaip negu išdėstymo metodai, neskaičiuoja atminties adresų, tačiau duomenis kaip ir anksčiau jungia į buketus. Kiekvienas medžio lapas atitinka duomenų buketą ir puslapį diske, o vidinės medžių viršūnės skiriamos nuorodoms saugoti ir todėl atitinka tam tikrus geometrinius poerdvius. Skirtumai tarp šių struktūrų ir jų sudėtingumų bei realizacijų daugiausia remiasi erdvių ypatybėmis.

**k-d-B-medžiai** jungia adaptivių k-d-medžių ir B-medžių konstravimo ypatumus. Jie skaido geometrinę erdvę lygiai taip, kaip adaptivių k-d-medžiai ir gautiems poerdviams priskiria vidines viršūnes. Poerdviai, priskiriami broliams, nesikerta, o jų sąjunga apima visą erdvę. Duomenų taškai, priskiriami atskiriems medžio lapams, yra iš vieno skaidinio ir saugomi viename disko puslapyje. k-d-B-medžiai yra visiškai subalansuoti medžiai, pritaikyti esamam duomenų pasiskirstymui. Modelinį uždavinį atitinkantis k-d-B-medis pavaizduotas 27 pav.

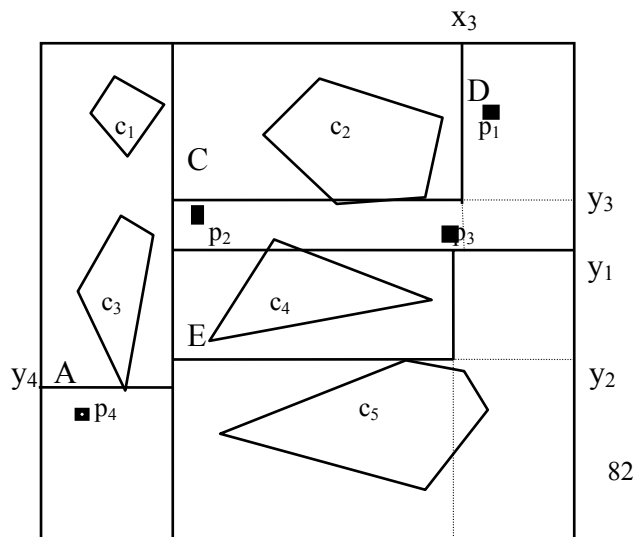


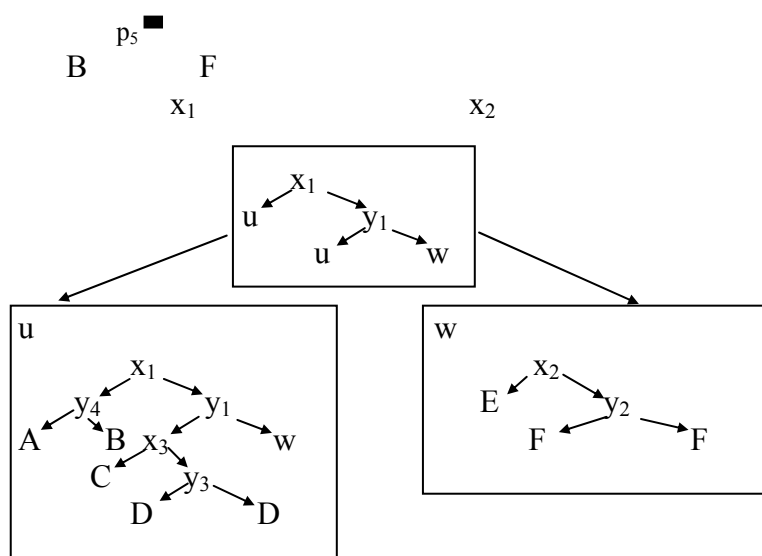


27 pav. Modelinio uždavinio k-d-B-medis.

Paieškos operacija atliekama lygiai taip kaip ir adaptyviame k-d-medyje. Prieš įterpiant elementą, paieškos metu randamas reikiamas duomenų buketas ir, jei disko puslapyje yra vietos, elementas ten įrašomas. Nesant vietos puslapyje, buketas skaidomas į du, atitinkamos vidinės viršūnės keičiamos taip, kaip tai buvo daroma 2-3-4-medyje (arba B-medyje). Vidinių viršūnių keitimas gali apimti kelis medžio lygius. Dėl šios priežasties negalima sakyti, kad k-d-B-struktūra visada efektyvi - galimi atvejai, kai atmintis bus labai netaupiai naudojama. Lygiai tokiomis pat ypatybėmis pasižymi ir išmetimo operacija.

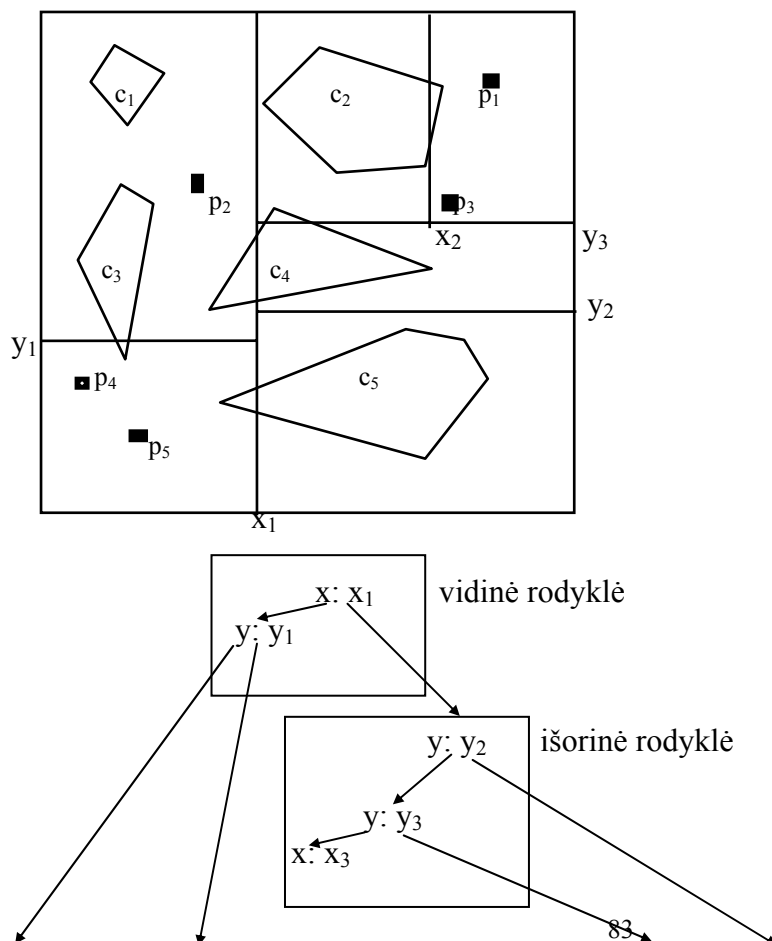
**hB-medžiai** (*holey brick tree*) konstravimo maniera yra giminingi k-d-B-medžiams, skiriasi tik poerdvių skaidymo taisyklės - poerdvių taškams gali būti taikomi skirtingi atributai. Todėl vidinės viršūnės atitinka ne d-mačiai intervalai, bet tokie, iš kurių gali būti gaunami intervalų poaibiai. Erdvinė struktūra, charakteringa hB-medžiams, labai panaši į fraktalinį piešinį. Modelinį uždavinį atitinkantis hB-medis parodytas 28 pav.

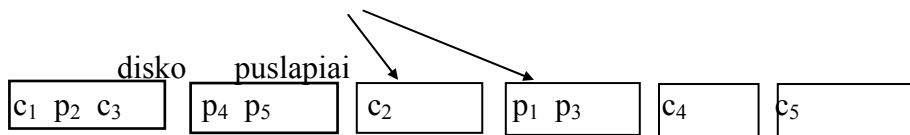




28 pav. Modelinio uždavinio hB-medis.

**LSD-medis** geometrinės erdvės skaidyme remiasi adaptyviais k-d-medžiais, tačiau erdvę skaido į nelygias gardeles. Šitai algoritmas efektyviai prisitaiko prie esamų duomenų pasiskirstymo, o naudojant specialų puslapiavimo algoritmą gaunamas subalansuotas medis. Jei duomenys netelpa vidinėje atmintyje, algoritmas identifikuoja pomedžius, kuriuos tikslinga perkelti į išorinę atmintį. Detalus algoritmo aprašymas ir ypač jo realizavimas labai priklauso nuo duomenų ir geometrinės erdvės. Modelinio uždavinio LSD-medžio vaizdas pateiktas 29 pav.





29 pav. Modelinio uždavinio LSD-medis.

### 5.2.3. Erdviniai kreipties metodai

Manipuliacijos su erdviniais duomenimis yra labai skirtingos ir todėl joms taikomi labai skirtingi principai. Iki šiol nagrinėtos duomenų struktūros buvo suformuluotos ir tinkamos daugiausia tam atvejui, kai geometrinėje figūroje reikia ieškoti taškų arba jais identifikuojamų objektų. Tačiau geografinių duomenų bazės, operuojančios daugiakampiais, ar mechaninės CAD-sistemos, kurių pagrindiniai objektai yra trimačiai briaunainiai, reikalauja daug sudėtingesnių algoritmų. Norint pritaikyti taškų paieškos algoritmus tokiems ir panašioms taikymams, reikia papildomai naudoti šių algoritmų modifikavimo procedūras:

- transformacijas (*object mapping*);
- sričių persidengimą (*overlapping regions, object bounding*);
- sričių atkartojimą (*clipping, object duplication*).

**Transformacijos.** Objektus, sudarytus iš paprastų geometrinių figūrų (apskritimų, trikampių, stačiakampių ir pan.), galima transformuoti į taškus didesnių matavimų erdvėje. Pavyzdžiui, atkarpą ir jos vietą plokštumoje pilnai apibrėžia atkarpos galų koordinatės, t.y. dvi skaičių poros, o iš jų gautą skaičių ketvertą galima interpretuoti kaip tašką keturmatėje Euklido erdvėje. Tą pačią atkarpą galima aprašyti ir kitaip: jos centro koordinatėmis, atkarpos ilgiu ir kampų, kurį atkarpa sudaro su x ašimi. Visi šie skaičiai irgi charakterizuoja tašką keturmatėje erdvėje. Galimi ir kitokie figūrų atvaizdavimai į taškus didesnių matavimų erdvėje. Kad ir koks būtų atvaizdavimas, objektams kompiuterio atmintyje dėstyti ir ieškoti gali būti naudojami taškiniai paieškos metodai.

Nusprendus, kokias transformacijas iš pradinės erdvės į didesnių matavimų erdvę ir kokius taškinis kreipties metodus nagrinėjamame uždavinyje tikslinga naudoti, reikia formuluoti taisykles, kaip erdvinės paieškos užklauso, manipuliacijos su duomenų sritimis, duomenų įterpimas ar išmetimas (pradinėje erdvėje) bus transformuojami į operacijas su taškais (vaizdo erdvėje). Dažniausiai šios operacijos susiveda į taškų identifikavimo ir duomenų intervalų ar buketų išskyrimą. Realizuojant tokį mechanizmą, labai svarbus yra operacijų efektyvumas. Praktiškai užklausų formulavimas ir vykdymas vaizdo erdvėje yra kur kas sudėtingesnis, negu pradinėje erdvėje, todėl būtina kompleksinė viso šio proceso analizė. Be to, reikia turėti omenyje, kad transformacijas galima gan paprastai formuluoti ir naudoti tik nesudėtingiems geometriniams objektams. Jei objektai sudėtingesni, pvz., įvairių formų daugiakampiai, sunku rasti kokią nors akivaizdų jų atvaizdavimą į didesnių matavimų taškus. Priklausomai nuo konkretaus uždavinio galima arba šiuos objektus aproksimuoti paprastesniais (pvz., naudoti minimalius gaubiamuosius stačiakampius ar trikampus), arba objektus ir taškus erdvėje sutvarkyti specialiu būdu (pvz., užpildyti geometrinę erdvę fraktalinėmis kreivėmis), arba pereiti prie kitokių vaizdavimo būdų. Per pastaruosius 5-7 metus šioms problemoms spręsti buvo pasiūlyta daug įvairių metodų ir algoritmų.

**Sričių persidengimas.** Taškinis kreipties metodus galima modifikuoti, pritaikyti erdvinėms sritims saugoti ir su jomis manipuliuoti, leidžiant duomenis iš susikertančių sričių (t.y. pasikartojančius taškus ar figūras) dėstyti skirtinguose duomenų buketuose. Yra nemažai sistemų, kuriose ši idėja realizuota. Jų patirtis teigia, kad algoritmų funkcionalumas išlieka beveik toks pat (nes hierarchinių struktūrų gylis padidėja nežymiai). Tačiau kitas sistemos darbo parametras - našumas - gali labai keistis priklausomai nuo to, kokią geometrinės erdvės dalį užpildo duomenys. Jei įterpiami duomenys visą laiką didina susikertančių sričių skaičių ir sankirtų tūrį, tam tikros

duomenų rodyklės gali tapti labai didelės ir neefektyvios. Dar vienas, nors ir mažesnis pavojus, slypintis tokiose struktūrose - neapibrėžtumas įterpiant naujus elementus į susikertančias sritis (t.y. kuriems buketams reikia priskirti naujus elementus, kai jie yra susikertančiose srityse). Nesiimant tam specialių priemonių, paieškos struktūros gali būti vaizduojamos nesubalansuotais medžiais arba įterpimo operacijos metu gali prireikti daug parengiamųjų skaičiavimų. Ši problema iki šiol buvo sprendžiama euristiniais metodais.

**Sričių atkartojimas.** Paieškos schemas, pagrįstos sričių atkartojimu, eliminuoja sričių netuščias sankirtas ir išvengia jų tolimensniame konstravime. Netuščioms sankirtoms yra priskiriami keli duomenų buketai, saugantys įvairiarūšę informaciją: sankirtos duomenų turinį, atributus, srities geometriją ir pan. Vadinasi, hierarchinių duomenų struktūrų paieškos šakos yra vienareikšmės, o objektų atributai atskirti (informacija apie objekto formą saugoma atskirai nuo objekto turinio). Aišku, kad tokioje struktūroje paieška yra greita, ar tai būtų objektų turinio, ar jų formos užklausa. Šio metodo problemos susiję su duomenų įterpimo ir išmetimo operacijomis. Įterpiant elementus, gali prireikti sritis skaidyti specialiomis hiperplokštumomis, o informaciją apie objektą saugoti keliuose puslapiuose, viename iš jų rašant srities geometrines charakteristikas, kitame - turinį, trečiame - specialius atributus, ir pan. Šitoks duomenų saugojimo perteklingumas kelia pavojų, kad struktūra gali išsigimti, ne tik neleistina ilgainiai paieškos laiką, bet ir didindama sankirtoms atstovaujančių buketų skaičių. Be to, įterpimo metu gali didėti sričių sankirtos dydis. Taigi, sričių atkartojimo atveju geometrinėje erdvėje gali neužtekti vietos duomenų išplėtimams dėstyti, atsiras vadinamoji aklavietė (*deadlock*), kuriai išspręsti gali prireikti pertvarkyti visus struktūros duomenis. Dar viena galima problema - duomenų buketų skaidymas. Paprastai kiekvienas buketas atitinka tam tikrą geometrinę sritį arba jos atributus. Tačiau elementų įterpimo metu disko puslapyje turi būti pakankamai vietos naujiems duomenims. Jei vietos puslapyje neužtenka, duomenų buketą reikia skaidyti, tačiau nekeisti pačios srities. Vadinasi, vienai sričiai ar vienam srities aprašymui gali būti skirti keli duomenų buketai. Aišku, kad šitokia situacija gali pabloginti paieškos charakteristikas ir algoritmo realizavimą.

Daugiamačių, erdvinių ir pan. duomenų įvedimo į kompiuterių atmintį, saugojimo joje, paieškos ir išvedimo algoritmai bei atitinkamos duomenų struktūros yra šiuo metu gan plačiai ir intensyviai tiriamos, joms dar nėra susiklosčiusios vieningos teorijos. Todėl dar nėra ir vadovėlių, kuriuose būtų nagrinėjamos šios struktūros. Čia pateikta medžiaga yra surinkta daugiausia iš straipsnių žurnaluose, techninių aprašų, monografijų, kitokių panašaus pobūdžio leidinių. Dalis pateikiamų struktūrų aprašyta (dažniausiai kitokiame kontekste) [14] leidinyje.

Be to, daugiamačių duomenų struktūrų ir algoritmų yra pasiūlyta labai daug. Jei tekstinio tipo duomenims yra susiklosčiusi tam tikra klasikinė teorija ir jos pagrindinės struktūros šiame leidinyje buvo pateiktos, tai apie erdvines arba daugiamačias struktūras to negalima pasakyti. Liko visiškai neaprašyti daugiamačių erdvių sutvarkymo metodai (pvz., užpildančios erdvę kreivės ir z-tvarkiniai), daugiasieniai medžiai, R-medžiai ir jais besiremiantys algoritmai, hierarchinių daugiamačių briauninių struktūrų, daugiasluoksnių gardelių failai, ir pan.

## 6. Uždaviniai ir pratimai

Čia pateikiami uždaviniai yra suskirstyti į grupes. Kiekvienos grupės uždaviniams, ypač jų sprendimo būdai, keliama kitokie reikalavimai.

### 6.1. Pirmoji uždavinių grupė

Šios grupės uždaviniai nesudėtingi, pagrindinis tikslas yra parašyti šiems uždaviniams tokias programas, kuriose nebūtų nei vienos bereikalingai vykdomos operacijos, ir įrodyti, kad programos išsiskiria būtent šia savybe.

1. Surasti porą didžiausių skaičių, kurių didžiausias bendrasis daliklis ( $gcd$ ) yra lygus  $m$  ir kurie Paskalio sistemoje gali būti apibrėžti kaip *integer* tipo kintamieji.
2. Parašyti programą, skaičiuojančią Fibonačio skaičių seką iki duoto skaičiaus  $n$ .
3. Realizuoti dviejų dvejetainių  $n$ -bitinių skaičių, saugomų masyvuose  $A(n)$  ir  $B(n)$ , sudėtį. Sumą rašyti į masyvą  $C(n+1)$ .
4. Realizuoti  $n$ -ojo laipsnio vieno kintamojo polinomo reikšmių skaičiavimą pagal Hornerio schemą.
5. Surūšiuoti masyve  $A(n)$  saugomus skaičius, randant mažiausią iš jų ir rašant jį į masyvą  $B(n)$ , po to taikant šią procedūrą pakartotinai.
6. Masyve  $A(n)$  yra  $n$  skirtingų skaičių. Inversija vadinama situacija, kai indeksai  $i < j$ , o  $A(i) > A(j)$ . Suskaičiuoti inversijų skaičių masyvui  $A$ .
7. Parašyti programą, kuri iš  $N$  skaičių masyvo išrinktų tris skirtingus skaičius taip, kad jų suma dalytųsi iš  $k$ .
8. Skaičių seka išsiskiria savybe:  $a_1 < a_2 < a_3 < \dots < a_n$ . Parašyti programą, kuri nustatytų, kada  $a_i = i$ .
9. Svarbiausiu masyvo elementu vadinamas toks, kuris pasikartoja daugiau negu  $n/2$  kartų. Sudaryti programą, kuri rastų svarbiausią elementą, arba nustatytų, kad tokio nėra.
10. Parašyti programą, kuri tikrintų dalumo iš skaičiaus  $n$  požymius.
11. Sudauginti du didelius skaičius  $M$  ir  $N$  greitos daugybos metodu, taikant jį rekursyviai:  $M = 10nK + L$ ,  $N = 10nP + R$ .
12. Parašyti programą, kuri sudarytų daugybos lentelę.
13. Sukurti algoritmą ir parašyti programą, kuri vieną simbolių seką  $x$  pervestų į kitą simbolių seką  $y$ , įterpdama simbolį arba išmesdama jį ir atlikdama minimalų kiekį tokių operacijų.
14. Sukurti algoritmą ir parašyti programą, surandančią, kurios iš atkarpų plokštumoje yra lygiagrečios.
15. Sukurti algoritmą ir parašyti programą, surandančią, kiek atkarpų porų plokštumoje kertasi.
16. Parašyti programą realizuojančią **div** ir **mod** procedūras.
17. Parašyti programą, kuri sveikiems teigiamiesiems skaičiams  $a$ ,  $b$ ,  $c$  rastų tiesinės lygties  $a*x + b*y = c$  sveikaskaičius sprendinius, arba nustatytų, kad tokių nėra.
18. Parašyti programą, skaičiuojančią funkcijos  $\sin(x)$  reikšmes.
19. Parašyti programą, skaičiuojančią derinių skaičių  $C_n^k$ .

### 6.2. Antroji uždavinių grupė

Šios grupės uždavinių tikslas - patikrinti duomenų struktūrų programavimo įgūdžius.

1. Realizuoti operacijas su dvejetainiu prefiksiniu medžiu masyvo pagrindu.
2. Realizuoti operacijas su dvejetainiu infiksiniu medžiu masyvo pagrindu.
3. Realizuoti operacijas su dvejetainiu postfiksiniu medžiu masyvo pagrindu.
4. Realizuoti operacijas su dvejetainiu infiksiniu medžiu rodyklės pagrindu.
5. Realizuoti operacijas su dvejetainiu prefiksiniu medžiu rodyklės pagrindu.

6. Realizuoti operacijas su dvejetainiu postfiksiniu medžiu rodyklės pagrindu.
7. Realizuoti operacijas su rekursyviu medžiu.
8. Realizuoti operacijas su medžiu (nuorodos į kairinį sūnų ir dešinį broį) masyvo pagrindu.
9. Realizuoti operacijas su medžiu (nuorodos į kairinį sūnų ir dešinį broį) rodyklės pagrindu.
10. Realizuoti operacijas su aibėmis *bitmap* vektoriaus pagrindu.
11. Sudaryti programą, realizuojančią atviro adresavimo dėstymo algoritmą operacijoms *insert*, *search*, *delete*.
12. Sudaryti programą, realizuojančią dvigubo dėstymo metodą operacijoms *insert*, *search*, *delete*.
13. Realizuoti *Patricia* algoritmą.
14. Realizuoti skaitmeninės paieškos algoritmą (abiem knygoje aprašytais atvejais).
15. Realizuoti operaciją *delete* B-medyje.
16. Realizuoti *extendible hashing* paieškos algoritmą: įterpimo ir paieškos operacijas.
17. Realizuoti *bottom-up mergesort* rūšiavimo metodą.
18. Realizuoti *top-down mergesort* rūšiavimo metodą.
19. Realizuoti *heapsort* metodą.
20. Realizuoti įterpimo ir išmetimo operacijas *heap* struktūroje.
21. Realizuoti dviejų prioritetinių eilių suliejimo operaciją.
22. Realizuoti Knuth-Morris-Pratt'o algoritmą.
23. Realizuoti Boyer-Moore'o algoritmą.
24. Realizuoti Rabin-Karp'o algoritmą.

### 6.3. Trečioji uždavinių grupė

Šios grupės uždaviniai yra sudėtingesni, jų sprendime ir realizavime reikia taikyti įvairius duomenų struktūrų algoritmus.

1. Rasti dviejų aukšto laipsnio polinomų  $g(x)$  ir  $h(x)$  didžiausią bendrąjį daliklį.
2. Rasti dviejų didelių skaičių (kelių dešimčių ar šimtų skaitmenų eilės)  $M$  ir  $N$  didžiausią bendrąjį daliklį.
3. Kiekvieną skaičių nuo 1 iki  $n$  išskaidyti ne daugiau kaip  $k$  dėmenų suma. Kaip parinkti tokius skirtingus dėmenis, kad jų skaičius būtų minimalus?
4. Sukurti algoritmą ir parašyti programą, kuri generuotų atsitiktinę  $N$ -bitų ( $N > 1000$ ) seką, o po to rastų visus posekius, lygius paskutiniams  $k$  bitų.
5. Sukurti algoritmą ir parašyti programą, kuri dviejų matavimų tekste rastų visas vietas, kuriose yra tam tikras posekis.
6. Sukurti algoritmą ir parašyti programą, kuri sekų aibėje  $S = (y_1, y_2, \dots, y_n)$  išskirtų tas, kurios įeina į seką  $x$ .
7. Sukurti algoritmą ir parašyti programą, kuri surastų ilgiausią bendrąjį tolydų posekį dviejuose duotose sekose.
8. Sukurti algoritmą ir parašyti programą, kuri iš aibės  $S$  poaibių sistemos  $\{s_1, s_2, \dots, s_n\}$  išrinktų minimalų skaičių poaibių, kurių sąjunga padengtų visą aibę.
9. Sukurti algoritmą ir parašyti programą, kuri naudodama medžio duomenų struktūrą suspaustų tekstinį failą.
10. Sukurti algoritmą ir parašyti programą, kuri perkoduotų (ir dekoduo) bet kokį failą į tekstinį.
11. Sukurti algoritmą ir parašyti programą, kuri aibę iš  $n$  skaičių iš pradžių rūšiuoja *Quicksort* metodu, o likus mažesniems aibėms, pereina prie įterpimo metodo. Kada toks perėjimas tikslingas?
12. Sukurti efektyvų algoritmą ir parašyti programą, kuri iš  $n$  skaičių surastų  $k$  mažiausių.
13. Parašyti programas ir empiriškai palyginti metodus, kai formuojamos *heap* aibės iš atsitiktinių skaičių, naudojant dvi procedūras: *bottom-up heap* ir *top-down heap*.
14. Parašyti programas *heapsort*, kada naudojamos dvi skirtingos operacijos: išmesti mažiausią ir išmesti didžiausią (*heap* aibė formuojama taip, kaip buvo nagrinėta).

15. Parašyti programą, kuri natūraliai užrašytą algebrinį reiškinių (su skliausteliais) pertvarkytų į postfiksinių reiškinių. Kurias struktūras geriau naudoti: steką, eilę, deką?
16. Sukurti algoritmą ir parašyti programą, kuri polinomus vieną iš kito dalytų greitos daugybos metodu (polinomų laipsnis gali būti labai didelis).
17. Skaičių aibė yra skaidoma į kelias dalis taip, kad šių dalių sumos būtų kiek galima labiau viena kitai artimos. Šį principą rekursyviai galima pritaikyti vis mažesnėms aibėms. Panaudoti dvejetainius ar 2-3, ar 2-3-4-medžius. Parašyti atitinkamą programą ir atlikti uždavinio analizę.
18. Sudaryti programą, kuri naudodama atsitiktinių skaičių generatorių, sudarytų raudonus-juodus medžius (ne mažiau 1000 viršūnių) ir įvertintų vidutinį šakų ilgį nuo šaknies iki lapų.
19. Sudaryti programą, kuri pagal raudoną-juodą medį atstatytų 2-3-4-medį, ir atvirkščiai.
20. Realizuoti operacijas su *trie* struktūra, kai duomenys yra saugomi sąrašo pavidalu masyvo pagrindu.
21. Realizuoti dviejų išretintų matricių daugybą. Algoritmo efektyvumą palyginti su standartiniais metodais.
22. Realizuoti dviejų išretintų polinomų daugybą. Algoritmo efektyvumą palyginti su standartiniais metodais.
23. Realizuoti dviejų labai didelių dvejetainių skaičių daugybą. Algoritmo sudėtingumą palyginti su standartiniais metodais.
24. Sudaryti algoritmą ir parašyti programą, kuris algebrinį reiškinių pertvarkytų į prefiksinių pavidalą.
25. Parašyti programą, kuri dvejetainiame paieškos medyje rastų visas viršūnes, kurios patenka į tam tikrą reikšmių diapazoną.
26. Parašyti programą, kuri 2-3-4-medyje rastų visas viršūnes, patenkančias į tam tikrus diapazonus (jų gali būti keli).

#### 6.4. Pratimai tiriamajam darbui

1. Kompleksiškai ištirti dvejetainę paiešką ir dvejetainius paieškos medžius: išvesti jų sudėtingumo priklausomai nuo duomenų struktūros formules; pateikti rekursyvias šių algoritmų realizacijos programas.
2. Ištirti dvejetainius paieškos medžius: nustatyti, ar verta įterpti įrašus į medį jų dažnumų mažėjimo tvarka, kai iš anksto žinomi paieškos raktų dažnumai; modifikuoti dvejetainius medžius, kad jie galėtų turėti pasikartojančius raktus; sudaryti rekursyvias programas, spausdinančias medžių viršūnes eilės tvarka.
3. Ištirti raudonus-juodus medžius: naudojant atsitiktinių skaičių generatorius, ištirti šių medžių statistines savybes (šakų minimalius, maksimalius, vidutinius ilgius; santykius tarp juodų ir raudonų-juodų šakų; kitas savybes); palyginti šiuos medžius su nebalansuotais dvejetainiais paieškos medžiais.
4. Sukurti keturių spalvų ir aštuonių spalvų medžių teoriją (kokio tipo paieškos struktūroms gali atstovauti medžiai, kai raudoname-juodame medyje spalvai žymėti yra naudojami du arba trys bitai), pateikti pavyzdžių.
5. Išanalizuoti įvairias *hash* funkcijas: kaip galima *hash* funkcijas keisti atsitiktinių skaičių generatoriais ir atvirkščiai; kokias funkcijas reikia naudoti, kai žinomi reikšmių diapazonai, elementų lentelėje skaičius, kai laukiama, kad bus daug pasikartojančių raktų reikšmių; kaip tirti įvairių *hash* funkcijų statistines savybes.
6. Ištirti skaitmeninius paieškos medžius: naudojant atsitiktinių skaičių generatorius, ištirti šių medžių statistines savybes; surasti duomenų savybes, sąlygojančias blogą struktūros *trie* išbalansavimą; surasti kaip reikia taisyti struktūrą tokiu atveju: 26-ųjų nuorodų *trie* struktūrą naudojama lotyniško alfabeto raidės identifikuoti, tačiau kai kurios raidės labai retai sutinkamos.



7. Kompleksiškai ištirti *Patricia* algoritmą: naudojant atsitiktinių skaičių generatorius, ištirti jo statistines savybes; rasti duomenų savybes, kurios *Patricia* medį labai debalansuoja; kada verta parašyti programas, spausdinančias *Patricia* medžio viršūnes didėjimo tvarka.
8. Kompleksiškai išanalizuoti B-medžius: naudojant atsitiktinių skaičių generatorius, ištirti šių medžių statistines savybes; kokie turi būti santykiai tarp puslapio dydžio, sūnų skaičiaus vienai viršūnei, viršūnių skaičiaus, prefikso dydžio, kad paieška (įterpimo operacija, išmetimo operacija) medyje būtų efektyviausia.
9. Kompleksiškai ištirti išplėstinį dėstymą: naudojant atsitiktinių skaičių generatorius, ištirti šio algoritmo statistines savybes; parašyti elemento išmetimo programą ir išanalizuoti jos sudėtingumą; kaip turi būti modifikuotas išplėstinio dėstymo algoritmas, kai jis dirba vidinėje atmintyje.
10. Modifikuoti visus išdėstytus sekų paieškos algoritmus, kai fragmentas turi vieną ar kelis *wild-card* simbolius, ištirti jų sudėtingumą.
11. Kompleksiškai ištirti Huffman'o kodo savybes: kaip kodo ilgis priklauso nuo statistinio dažnumų pasiskirstymo; koks yra dvejetainio failo Huffman'o kodas; jei raidžių dažnumai visi skirtingi, ar Huffman'o kodas yra vienareikšmis; kaip taupyti atmintį realizuojant Huffman'o algoritmą.

## 7. Literatūra

1. Sedgewick R. Algorithms. Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.
2. Ammeraal L. Algorithms and Data Structures in C++. John Wiley & Sons Ltd., Chichester, 1996.
3. Cormen Th. H., Leiserson Ch. E., Rivest R. L. Introduction to Algorithms. The MIT Press, Cambridge, Massachusetts, 1993.
4. Jensen K., Wirth N. Pascal User Manual and Report. Springer-Verlag, Berlin, 1974.
5. Horowitz E., Sahni S., Anderson-Freed S. Fundamentals of Data Structures in C. Computer Science Press, New York, 1993.
6. Carrano F., Helman P., Veroff R. Data Structures and Problem Solving with Turbo Pascal. Walls and Mirrors. The Benjamin/Cummings Publishing Company Inc., 1993.
7. Naps Th. L., Nance D. W., Singh B. Introduction to Computer Science: Programming, Problem Solving and Data Structures. West Publishing Company, St. Paul, Minnesota, 1989.
8. Naps Th. L., Singh B. Program Design with Pascal. West Publishing Company, St. Paul, Minnesota, 1988.
9. Gabrini Ph. J., Kurtz B. L. Data Structures and Algorithms with Modula-2. Addison-Wesley Publishing Company, Reading, Massachusetts, 1993.
10. Stubbs D. F., Webre N. W. Data Structures with Abstract Data Types and Modula-2. Brooks Cole Publishing Company, Monterey, California, 1987.
11. Aho A. V., Hopcroft J. E., Ullman J. D. The Design and Analysis of Computer Algorithms. Addison-Wesley Publishing Company, Reading, Massachusetts, 1976 (yra vertimas į rusų kalbą).
12. Knuth D. E. The Art of Computer Programming. Addison-Wesley Publishing Company, Reading, Massachusetts 1968. 1, 1969. 2, 1973. 3 (yra vertimai į rusų kalbą).
13. Greene D. H., Knuth D. E. Mathematics for the Analysis of Algorithms. Birkhäuser, Boston - Basel - Stuttgart, 1982 (yra vertimas į rusų kalbą).
14. Khoshafian S., Baker A. B. Multimedia and Imaging Databases. Morgan Kaufmann Publishers, Inc. San Francisco, California, 1996.