

Datu struktūras (DIP203)

Lekciju materiāls sagatavots projekta
***“RTU akadēmiskās studiju programmas
“Datorsistēmas” kursu pilnveidošana”***
ietvaros

Datu struktūras

Docents Gunārs Matisons

Rīgas Tehniskā universitāte

Datorzinātnes un informācijas tehnoloģijas fakultāte

Lietišķo datorsistēmu institūts

Programmatūras izstrādes tehnoloģijas profesora grupa

Priekšmeta pamatdati

Priekšmeta pieteicējs: Gunārs Matisons

Apjoms: 3 KP

Kontroles veids: Eksāmens

Studiju līmenis: Bakalaura

Semestris: 2. semestris

Priekšmeta mērķi un uzdevumi

Mācību kursa mērķi:

- iepazīstināt studentus ar datu tipu un datu struktūru specifikācijām, ar datu struktūru projektēšanas un veidošanas metodēm un attēlošanas paņēmieniem, ar efektīviem algoritmiem darbā ar bieži lietojamām datu struktūrām;
- iemācīt studentus izvēlēties visoptimālākās datu struktūras un to apstrādes algoritmus un lietot tos praksē programmatūras izstrādes procesā.

Mācību kursa galvenie uzdevumi ir:

- studentiem skaidri jāizprot datu struktūru jēdziens, lietojuma nozīme un klasifikācijas principi;
- studentiem jāapgūst datu struktūru attēlojuma veidi un modeļi;
- studentiem jāapgūst datu struktūru modeļu veidošana un aprakstīšana, datu struktūru projektēšana un ieviešana, jāprot to visu lietot praksē;
- studentam jāprot izvēlēties un lietot vispiemērotākās datu struktūras un efektīvākos datu struktūru attēlojuma modeļus katrā konkrētā informācijas tehnoloģiju lietojuma gadījumā;
- studentiem jāprot izvēlēties visoptimālākos algoritmus datu struktūras apstrādes operāciju izpildei, jāprot novērtēt to efektivitāti

Pamatliteratūra

1. G. Matisons. Datu struktūras. Lekciju konspekts, RTU izdevniecība, Rīga, 2008,
2. Ellis Horowitz, Sartaj Sahni. Fundamentals of Data Structures in Pascal. 4-th edition. W.H. Freeman and Company, NJ, 1990, p.609.
3. D. Wood. Data Structures, Algorithms and Performance. Addison – Wesley Publishing Company, NJ, 1993, p.594.
4. Daniel Stubbs, Neil W. Webre. Data Structures with Abstract Data Types and Pascal. Brooks/Cole Publ. Company, 1989, Ca, p.404.
5. Mark Allen Weiss. Data Structures & Algorithms Analysis in Java. Addison – Wesley Publishing Company, 1999, p.542.
6. Wayne Amsbrery. Data Structures from Arrays to Priority Queues. Wadsworth Publishing Company, 1985, p.516.

7. Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. Data Structures and Algorithms. Addison – Wesley Publishing Company, NJ, 1983.
8. Вирт Н. Алгоритмы и структуры данных / Пер. с англ. – М., Мир, 1989, 360 с.
9. Трамбле Ж., Соресон П. Введение в структуры данных. М., Машиностроение, 1982, 784 с.
10. Бертисс А.Т. Структуры данных / Пер. с англ. М., Статистика, 1974.
11. Макаровский В.Н. Информационные системы и структуры данных. М., Статистика, 1980.
12. http://en.wikipedia.org/wiki/Data_structures
13. http://en.wikipedia.org/wiki/List_of_algorithms

Papildliteratūra

1. Donald E.Knuth. Fundamental Algorithms, volume 1 of The art of Computer Programming. Addison – Wesley Publishing Company, NJ, 1968. Second Edition, 1973. (Русский перевод первого издания: Кнут Д., Искусство программирования для ЭВМ. Т.1.: Основные алгоритмы. М.: Мир, 1976)
2. Donald E.Knuth. Seminumerical Algorithms, volume 2 of The art of Computer Programming. Addison – Wesley Publishing Company, NJ, 1969. Second Edition, 1981. (Русский перевод первого издания: Кнут Д., Искусство программирования для ЭВМ. Т.2.: Получисленные алгоритмы. М.: Мир, 1977)
3. Donald E.Knuth. Sorting and Searching, volume 3 of The art of Computer Programming. Addison – Wesley Publishing Company, NJ, 1973. Second Edition, 1973. (Русский перевод первого издания: Кнут Д., Искусство программирования для ЭВМ. Т.3.: Сортировка и поиск. М.: Мир, 1978)

Atslēgvārdi

Dati

Struktūra

Sistēma

Lietotājs

Datu tips

Domēns

Datu struktūra

Skalārs datu tips

Strukturēts datu tips

Datu struktūras specifikācija

Datu struktūras projektēšana

Datu struktūras ieviešana

Lineāra datu struktūra

Nelineāra datu struktūra

Pamattēmas (1)

1. Mācību priekšmeta mērķi un uzdevumi. Datu jēdziens. Datu tipa jēdziens. Datu struktūras jēdziens. Datu struktūras un to klasifikācija.
2. Biežāk lietotās datu struktūras. Datu struktūru izstrāde. Datu struktūras attēlojuma paņēmieni un modeļi. Datu struktūras elementu identifikācija.
3. Datu struktūras projektējuma vērtēšana. Metrika, efektivitāte, veiktspēja. Rakstzīmju attēlošana ar mainīgu bitu skaitu. Rakstzīmju virknes jēdziens. Rakstzīmju virknes tipa specifikācija. Rakstzīmju virknes attēlojuma modeļi.
4. Masīva jēdziens. Matricas jēdziens un interpretācija. Elementa meklēšana vektorā. Deskriptors un tā lietojums. Vektora adresēšanas funkcijas noteikšana. Divdimensiju masīva adresēšanas funkcija. Vairākdimensiju masīvi un to adresēšanas funkcija.
5. Speciālie masīvi un to lietojums. Diagonālmatrix. Simetriskā matrica. Apakšējā trīsstūrmatrix. Augšējā trīsstūrmatrix. Retinātā matrica.
6. Ieraksta jēdziens. Ieraksta lauku adresēšana un piekļuve. Masīvu ieraksti. Ierakstu masīvi jeb tabulas. Rādītāju masīvi un to lietojums.
7. Saraksta jēdziens. Vektoriālā formā attēlotais saraksts. Vienkāršsaistītais saraksts. Vienkāršsaistītais saraksts bez beigu rādītāja.
8. Divkāršsaistītais saraksts. Vienkāršsaistītais saraksts ar beigu rādītāju. Divkāršsaistītais saraksts bez beigu rādītāja. Cirkulārais saraksts. Divkāršsaistītais cirkulārais saraksts.
9. Divkāršsaistītais gredzens. Vienkāršsaistītais gredzens. Hronoloģiski sakārtotais saraksts. Vektoriālā formā attēlotais hronoloģiski sakārtotais saraksts. Saistīta formā attēlotais hronoloģiski sakārtotais saraksts.

Pamattēmas (2)

10. Sašķirotais saraksts. Vektoriālā formā attēlotais sašķirotais saraksts. Interpolatīvā meklēšana. Multiplikatīvā meklēšana. Saistītā formā attēlotais sašķirotais saraksts.
11. Pēc lietojuma biežuma sašķirotais saraksts. Pašorganizētais saraksts un 3 metodes to veidošanai. Saraksta elementu apmaiņas operācijas 3 saraksta veidošanas metodēm. Lineārās meklēšanas operācija divkāršsaistītajā cirkulārajā pašorganizētajā sarakstā.
12. Daudzkāršsaistītais saraksts. Steks. Vektoriālā formā attēlotais steks. Steka pāris. Saistītā formā attēlotais steks.
13. Rinda. Vektoriālā formā attēlotā rinda. Cirkulārā rinda. Saistītā formā attēlotā rinda.
14. Prioritātes rinda. Vektoriālā formā attēlotā prioritātes rinda. Saistītā formā attēlotā prioritātes rinda. Dekss – rinda ar diviem galiem. Vektoriālā formā attēlotais deks. Saistītā formā attēlotais deks.
15. Koks jeb hierarhiska datu struktūra. Binārā koka attēlojums saistītā formā. Binārā koka apgaita. Binārā koka apstrādes operācijas. Binārā koka attēlojums vektoriālā formā.
16. Binārās meklēšanas koks. AVL – koks. Kaudze

Datu (data) jēdziens

Programmas datu apstrādes procesā operē ar datiem. Izveidojot jaunas programmas, galvenā uzmanība ir jāpievērš ne tikai datu apstrādes algoritmu struktūrai, analīzei un izvēlei, ne tikai pašam programmēšanas procesam vien. Programmēšanas metodoloģijā liela nozīme ir arī datu lietojuma un datu uzbūves aspektiem. Programmu var uzskatīt par konkrētu abstraktu algoritmu realizāciju, balstoties uz datu uzbūvi un reālu attēlojumu. Datu apstrādes algoritma izvēle ir atkarīga no datu uzbūves. Tātad programmas struktūra un datu struktūra ir savā starpā cieši saistīti jēdzieni.

Dati ir primāri, programma ir sekundāra. Dati ir jebkuras programmas neatņemama sastāvdaļa. Dati jāuztver un jāinterpretē kā reālu objektu abstrakcija – veidojumi, kas var arī nebūt paredzēti programmēšanas valodās. Datiem var būt dažāds sarežģītības un organizācijas līmenis.

Skaidrojošā vārdnīca Webster:

Dati ir faktuāla informācija, piemēram, mērījumi, statistika un tml. par objektiem, notikumiem un parādībām, kas kodēta formalizētā veidā, kas derīga šīs informācijas vākšanai, glabāšanai un apstrādei ar nolūku iegūt jaunu informāciju.

Dati un informācija ir sinonīmiski jēdzieni, kas tomēr nav identiski. Dati ir formalizētā veidā attēlota informācija jaunas informācijas iegūšanai.

Datiem ir noteikta uzbūve jeb struktūra. Struktūra ir datu elementu

Datu tipa (data type) jēdziens (1)

Matemātikā datus klasificē pēc noteiktām pazīmēm un īpašībām. Ir skaitliskie dati un loģiskie dati. Skaitliskie dati iedalāmi veselos, reālos un kompleksos skaitļos.

Dati glabājas datoros, un datori operē ar datiem. Šie dati ir klasificējami pēc datu tiem. Katrā programmēšanas valodā ir definēti savi konkrēti datu tipi.

Vienkāršākie datu tipi, kas realizēti valodā Pascal ir šādi:

char

integer (apakštipi: byte, word, shortint, longint)

real (apakštipi: single, double, extended)

boolean

Šiem tiem atbilstošie mainīgie, konstantes un funkcijas pieņem vērtības, ar kurām datu apstrādes procesā tiek izpildītas dažādas darbības: aprēķini, datu ievade, datu izvade u.tml.

Datu tips ir

1) iespējamo vērtību kopums;

2) operāciju kopums šo vērtību apstrādei.

Katram datu tipam atbilst noteikts vērtību kopums, ko sauc par

domēnu (domain).

Datu tipa (data type) jēdziens (2)

Domēns

Tips	Vērtību kopums
char	alfabēta ASCII rakstzīmes – 256 gb.
integer	<u>MinInt</u> .. <u>MaxInt</u> -32768 32767
boolean	false .. true

Datu tipa (data type) jēdziens (3)

Tips	Operāciju kopums
char	piešķire := salīdzināšanas operācijas =, <=, >=, <>, in
integer	piešķire := aritmētiskās operācijas +, -, *, div, mod salīdzināšanas operācijas =, <=, >=, <>, in
boolean	piešķire := lōgiskās operācijas not, and, or, xor salīdzināšanas operācijas =, <=, >=, <>, in
real	piešķire := aritmētiskās operācijas +, -, *, / salīdzināšanas operācijas =, <=, >=, =, <>

Datu struktūras (data structure) jēdziens

Datu organizācijas un datu elementu sasaistes raksturs, iespējamo vērtību kopums un iespējamo operāciju kopums ir datu struktūra.

Datu struktūra ir jebkuram informācijas objektam piemītoša īpašība.

Datu struktūra ir jebkuras programmēšanas sistēmas vai vides neatņemama sastāvdaļa. Teorētiskās un praktiskās zināšanas par datu struktūrām ir nepieciešamas, izstrādājot:

informācijas sistēmas, datu bāzu pārvaldības sistēmas, mākslīgā intelekta sistēmas, lēmumu pieņemšanas un vadības sistēmas, ekspertu sistēmas, imitācijas un modelēšanas sistēmas, programmēšanas valodu kompilatorus, operētājsistēmas u.c.

Datu struktūras (DS) un to klasifikācija (1)

Vienkāršas datu struktūras ir tādas DS, kurām atbilstošās vērtības ir skalāra tipa dati, kas nav sadalāmas sīkākās sastāvdaļās jeb datu elementos. Vienkāršas datu struktūras ir char, integer, real, boolean u.c. skalāra tipa dati.

char – 'A', '7'

integer – 10, -5

boolean – true

real – 1.23

Fundamentālas datu struktūras ir tādas DS, kurām atbilstošās vērtības ir elementu kopums, kas sadalāms sastāvdaļās jeb komponentos. Fundamentālās datu struktūras bieži izmanto, lai veidotu saliktas datu struktūras ar sarežģītāku uzbūvi. Fundamentālām DS atbilstošās vērtības ir strukturēta tipa dati, piemēram,

virknes – predefinēts datu tips string,

masīvi – predefinēts datu tips array,

ieraksti – predefinēts datu tips record.

Piemēram:

var A: array [1 .. 3, 1 .. 3] of integer;

$$A = \begin{array}{ccc|c} & \rightarrow j & & \\ 2 & 5 & -9 & \\ 0 & 4 & 7 & \downarrow i \\ 1 & 3 & 8 & \end{array}$$

$A[i, j]$ – masīva elements, piemēram, $A[1, 2] \Rightarrow 5$

Datu struktūras (DS) un to klasifikācija (2)

Par operatīvām DS sauc tādas DS, kuras tiek izvietotas un apstrādātas datora pamatatmiņā.

DS datora diskatmiņā sauc par failu (datņu) struktūrām. Failu struktūras elements ir faila ieraksts. Savstarpēji saistītu failu struktūrās glabātu sarakstu kopums veido datu bāzi.

Izvēli starp operatīvām DS un failu struktūrām nosaka piekļuves un apstrādes efektivitātes, kā arī atmiņas apjoma apsvērumi.

Svarīga DS īpašība ir DS elementu sasaiste un sakārtotība. Atkarībā no tā DS ir iedalāmas:

- 1) lineārās datu struktūrās (virknes, masīvi, ieraksti, faili, saraksti);
- 2) nelineārās datu struktūrās (koki, grafi, daudzkāŗšsaistīti saraksti).

Datu struktūras (DS) un to klasifikācija (3)

Atkarībā no tā, kā mainās DS uzbūve, izpildot DS apstrādes operācijas, tās ir iedalāmas:

- 1) statiskās datu struktūrās (masīvi, ieraksti, tabulas);
- 2) dinamiskās datu struktūrās (saistīti saraksti, koki, grafi, faili).

Reizēm runā arī par pusstatiskām DS (steki, rindas).

Atkarībā no tā, kā DS elementi tiek izvietoti datora pamatatmiņā, DS ir iedalāmas šādi:

- 1) DS ar elementu secīgu izvietojumu pamatatmiņā (masīvi, ieraksti, tabulas);
- 2) DS ar elementu patvaļīgu izvietojumu pamatatmiņā (saistīti saraksti, koki, grafi).

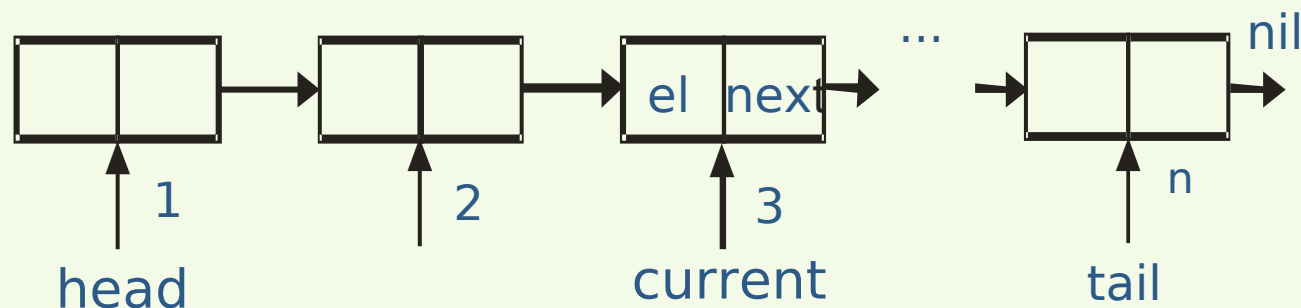
Atkarībā no tā, vai DS elements satur kāda cita DS elementa adresi (rādītāju uz nākamo vai iepriekšējo elementu), DS ir

iedalāmas:

1) saistītās (linked) DS

Datu struktūras (DS) un to klasifikācija (4)

Saistītās DS piemērs:

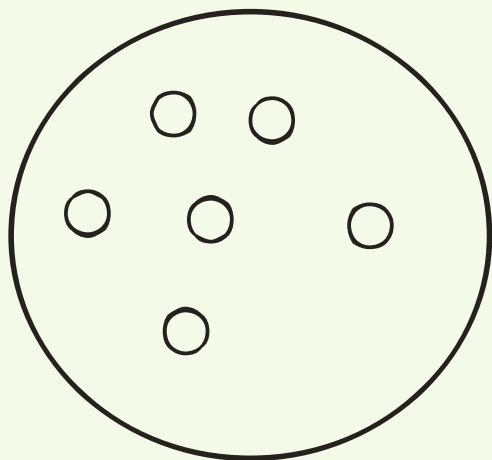


Tabula ir nesaistītas DS piemērs.

Atkarībā no tā, vai elementu izvietojums datu struktūrā ir patvaļīgs un nav determinēts vai arī datu struktūrā elementi izvietoti pēc kādas noteiktas pazīmes, DS ir iedalāmas:

- 1) sakārtotās (ordered) datu struktūrās;
- 2) nesakārtotās (nonordered) datu struktūrās.

Biežāk lietotās datu struktūras (1)



kopa (angl. set)
sasaiste (relationship)

Datu struktūru, kurā starp elementiem nav nekādas citas sasaistes kā vienīgi tās, ka visi elementi pieder pie noteikta datu kopuma, sauc par **kopu**.

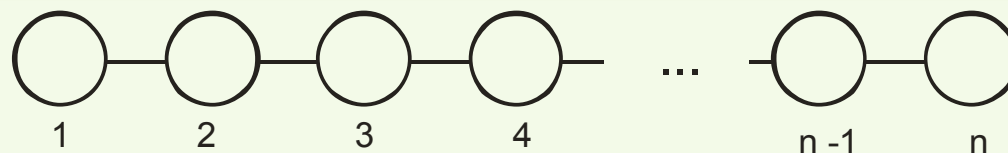
Kopā starp elementiem nav sasaistes.

Kopā nav elementa, ko var saukt par pirmo, pēdējo vai tekošo.

Kopas piemēri:

- 1) studenti grupā, kuri mācās angļu valodu;
- 2) grāmatas par informācijas tehnoloģiju utml.

Biežāk lietotās datu struktūras (2)

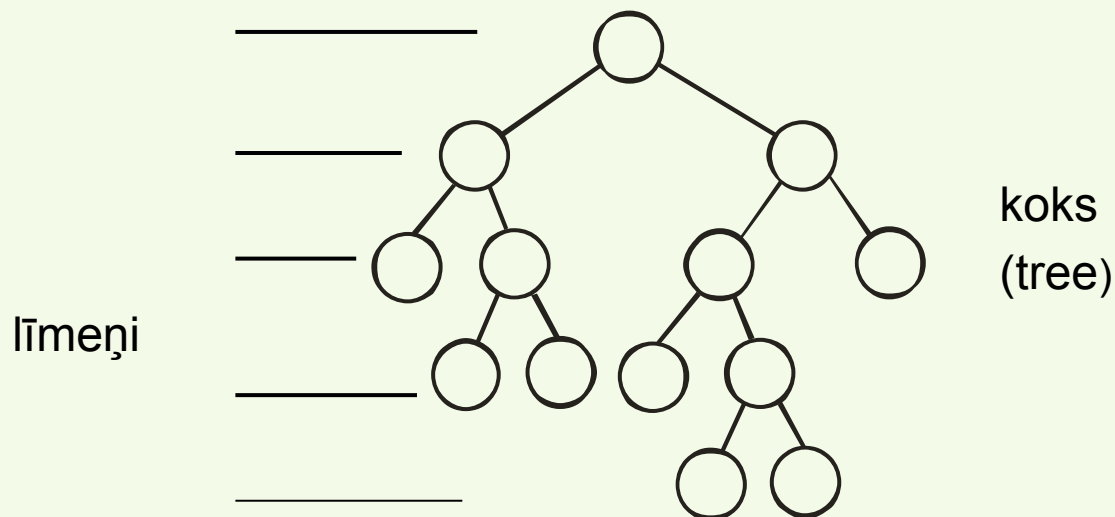


Datu struktūru, kurā elementu sasaistes raksturs ir **"viens ar vienu"** (one-to-one), sauc par **lineāru datu struktūru**.

Lineārā datu struktūrā katram elementam ir noteikts kārtas numurs, tajā ir elements, ko sauc par pirmo, un elements, kas ir pēdējais. Visiem elementiem, izņemot pirmo un pēdējo, ir viens vienīgs priekštecis (predecessor) un viens vienīgs pēctecis (successor). Pirmajam elementam nav priekšteca, bet pēdējam elementam nav pēcteča.

Lineārās datu struktūras lieto visbiežāk. Lineārās datu struktūras ir masīvi, ieraksti, faili un saraksti. Tās arī izmanto kā pamatelementus, veidojot datu struktūras ar sarežģītu uzbūvi.

Biežāk lietotās datu struktūras (3)

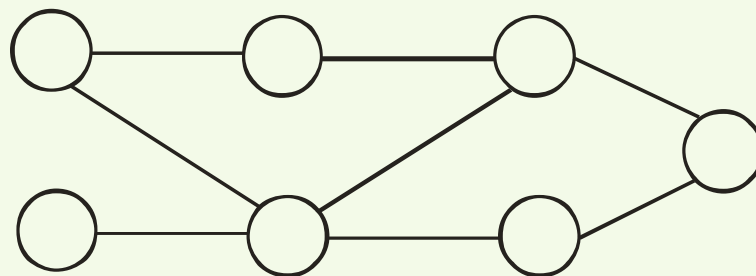


Datu struktūru, kurā elementu sasaistes raksturs ir **"viens ar vairākiem"** (one-to-many), sauc par **koku** (tree) jeb hierarhisku datu struktūru. Hierarhisks nozīmē to, ka datu struktūras elementi izvietoti vairākos līmeņos.

Kokā ir unikāls elements, ko sauc par saknes virsotni. Katram elementam ir viens, vairāki vai neviens pēctecis, ko sauc par bērnu (child) un viens vienīgs priekštecis, ko sauc par vecāku (parent). Saknes virsotnei nav priekšteča, bet var būt pēcteči. Koka virsotnes, kurām nav pēcteču, sauc par lapām (leaf).

Visbiežāk lieto bināros kokus, kuros katrai virsotnei nav vairāk kā 2 pēcteči. Katrs pēctecis ir kaisais bērns vai labais bērns. Virsotnes (node) koka savienotas ar šķautnēm (edge).

Biežāk lietotās datu struktūras (4)



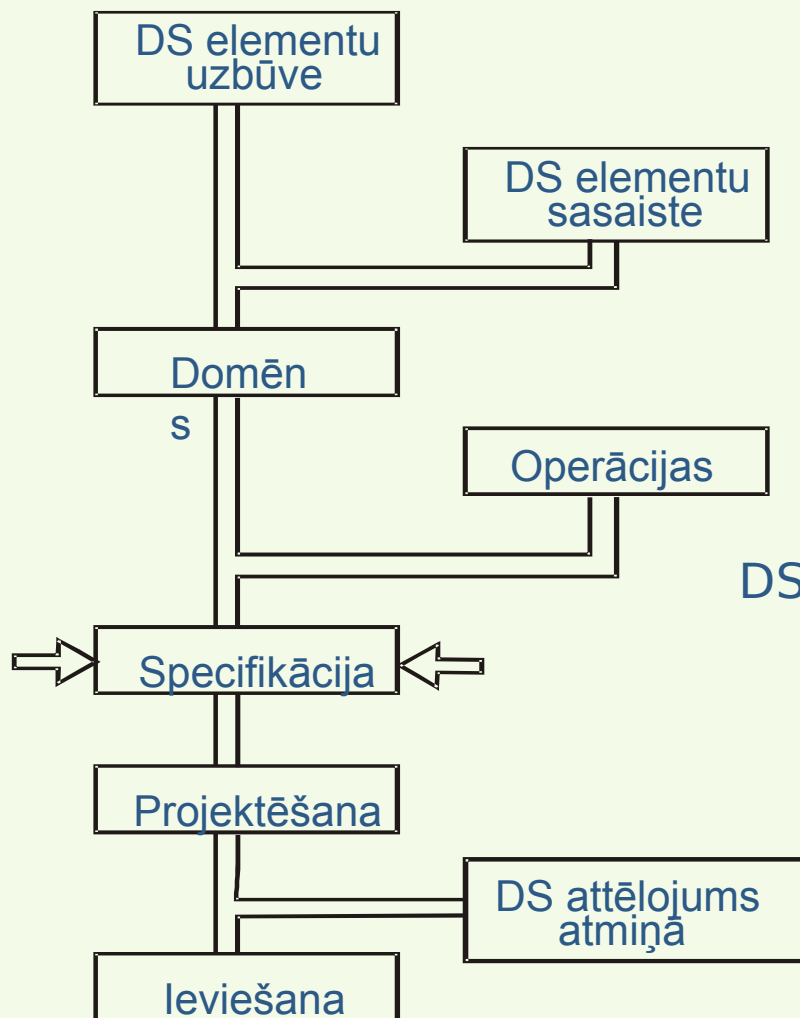
Datu struktūru, kurā elementu sasaistes raksturs ir **"vairāki ar vairākiem"** (many-to-many), sauc par **grafu** (graph) jeb tīklveida datu struktūru.

Grafā nav elementa, ko sauc par pirmo vai par pēdējo. Katram elementam ir vairāki pēcteči un vairāki priekšteči. Elementi savienoti ar lokiem. Darbā ar grafiem svarīga operācija ir īsākā ceļa meklēšana starp virsotnēm.

Grafus plaši lieto, uzdodot dažādus procesus, to stāvokļus un norises, ar grafu palīdzību risina arī optimizācijas problēmas.

Kokus un grafus sauc arī par **nelineārām datu struktūrām**.

Datu struktūru izstrāde



DS izstrāde (development):

- 1) specifikācija (specification);
- 2) projektēšana (design);
- 3) ieviešana (implementation).

Datu struktūras attēlojuma paņēmieni un modeļi (1)

1. paņēmiens:

adrese

vārds

1000	Aivars
1008	Markus
1016	Edgars
1024	Dainis
1032	Centis

Ja zināma i -tā saraksta elementa adrese, tad $i + 1$ – saraksta elementa adrese ir aprēķināma šādi:

$$\text{adrese}_{i+1} = \text{adrese}_i + l \quad (l = 8, \quad i = 1, 2, \dots, n-1)$$

Šo paņēmieni sauc par pozicionēšanu vektoriālā formā attēlotā modelī (array representation using positioning).

Datu struktūras attēlojuma paņēmieni un modeļi (2)

2. paņēmiens:

adrese

vārds

1000	Aivars
1008	---
1016	Centis
1024	Dainis
1032	Edgars
1040	---

...

1096	Markus
------	--------

Saraksta elementa adreses aprēķins:

$1000 + l * (\text{ord}(\text{pb}) - \text{ord}('A'))$ ($l = 8$, $\text{ord}('A') = 65$,
pb – vārda pirmais burts).

Šo paņēmieni sauc par jaukšanu jeb hešēšanu vektoriālā formā attēlotā modelī (array representation uing hashing).

Izmantojot šo paņēmieni, var rasties situācija, ko sauc par kolīziju, kad vairāki elementi pretendē uz vienu un to pašu vietu vektorā. Ir dažādi paņēmieni kolīziju novēršanai.

Datu struktūras attēlojuma paņēmiens un modeļi (3)

3. paņēmiens:

adrese vārds

pēcteča adrese

988	---	1000	-	sākumadrese
1000	Aivars	1024		
1012	---	---		
1024	Centis	1036		
1036	Dainis	1048		
1048	Edgars	1136		
1052	---	---		
...				
1136	Markus	nil	-	saraksta beigas

Šo paņēmieni sauc par saistīšanu (linking, linked representation), un to lieto saistītajā formā attēlotajā DS modelī. Katrs DS elements satur arī nākamā elementa adresi. Ir zināma pirmā elementa adrese, kas glabājas speciālā rādītāja laukā, kas vienmēr norāda uz saraksta sākumu. Pēdējam elementam nav pēcteča, tāpēc pēdējā elementa pēcteča adreses vērtība ir *nil*, kas nozīmē to, ka pēcteča adrese nav zināma (neeksistē).

Datu struktūras elementu identifikācija (1)

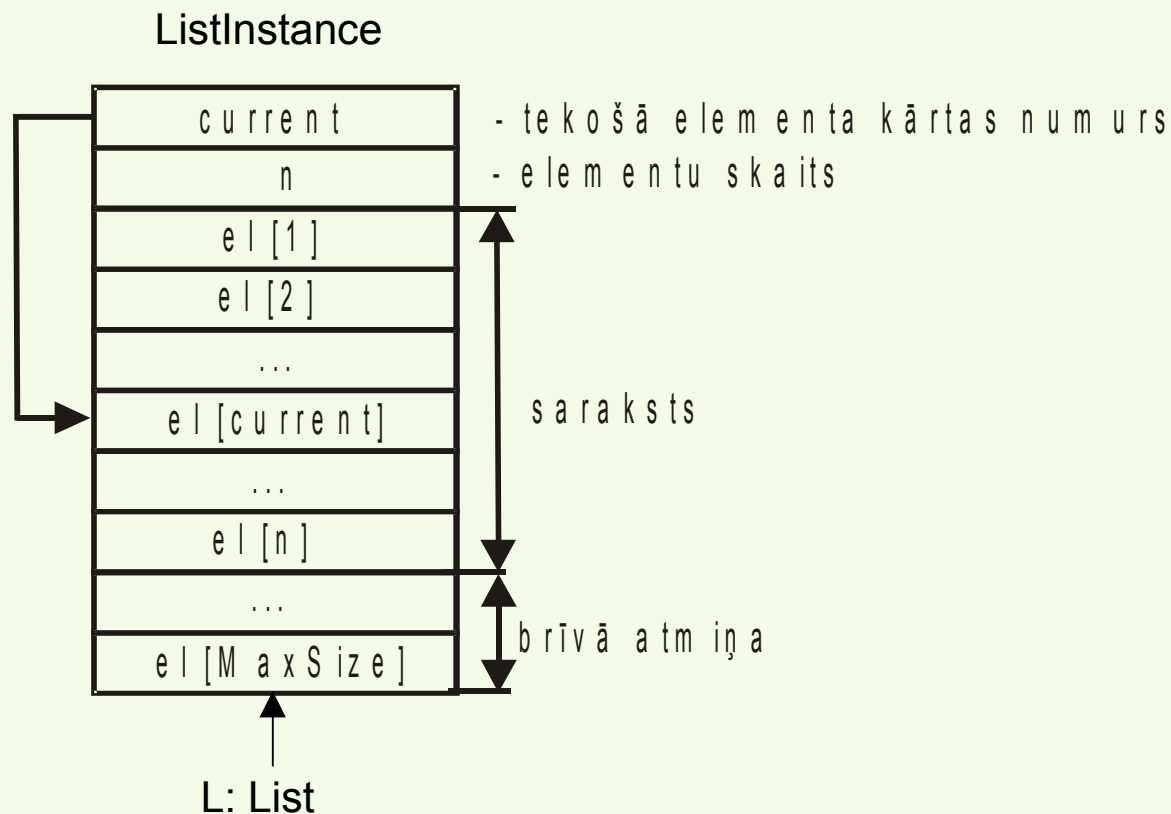
Lineāru un nelineāru DS elementiem ir vienāda uzbūve. Elementu veido 2 lauki:

1) informatīvs lauks **data**, kurā sakopota plaša un daudzveidīga informācija par DS elementu. Visai bieži šo lauku definē kā ierakstu, vienkāršākajā gadījumā – kā rakstzīmju virkni;

2) atslēgas lauks **key**, kas satur unikālu informāciju jeb kodu, kas viennozīmīgi identificē DS elementu. Parasti DS nav vairāki elementi ar vienu un to pašu atslēgu. Atslēgas laukam var uzdot jebkuru skalāru ordinālo tipu vai virknes tipu string.

Datu struktūras elementu identifikācija (2)

1) vektoriālajā formā attēlotā saraksta
modelis:



Datu struktūras

elementu identifikācija (3)

DS elementa uzbūve:

$el[i], \quad i = 1, 2, \dots, n$

datu tips

DataType

jebkurš tips,

strukturēts

const vai skaits $MaxSize = 100;$

skaitis}

type

tips}

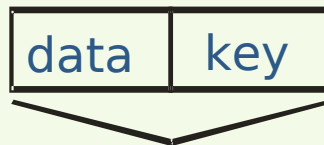
string}

StdElement = record

data: DataType;

key: KeyType

end;



datu tips

datu tips

KeyType

jebkurš skalārs ordināls

tips

vai virknes tips string

- ieraksta tips

{maksimālais elementu

{jebkurš datu

{skalārs ordināls datu tips vai

{saraksta elementa tips}

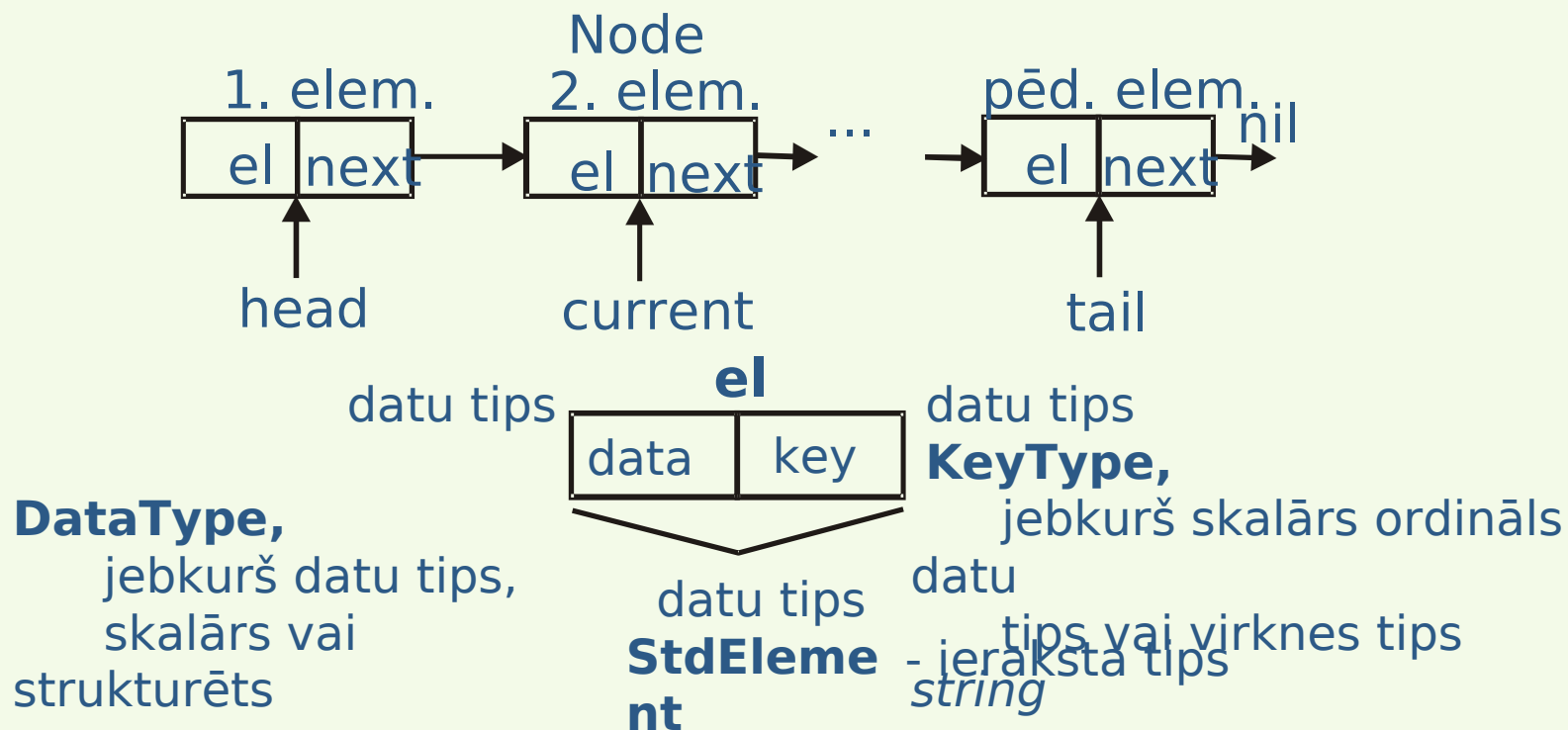
{informatīvs datu lauks}

{unikālas atslēgas lauks}

Datu struktūras

elementu identifikācija (4)

2) saistītajā formā attēlotā saraksta modelis:



Datu struktūras

elementu identifikācija (5)

```

const MaxSize = 100;           {maksimālais elementu
skaits}

type DataType = string;       {jebkurš
datu tips}

    KeyType = integer;        {skalārs ordināls datu tips vai
tips string}

    StdElement = record
        data: DataType;       {informatīvs datu
lauks}
        key: KeyType           {unikālas atslēgas
lauks}
    end;

    NodePointer = ^ Node;      {rādītāja datu
tips}

```

Node = record

{saraksta

32

Datu struktūras projektējuma vērtēšana (1)

Projektējot DS, jārisina šādas problēmas:

- 1) jānovērtē datu struktūras apstrādes operāciju izpildes laiks;
- 2) jāizvēlas, vai tiks veidota statiska vai dinamiska datu struktūra;
- 3) jānoskaidro, vai iespējams, ka datu struktūras elementiem varētu būt mainīgs garums;
- 4) jānovērtē algoritmu izpildes efektivitāte (sarežģītības pakāpe);
- 5) jāizvēlas, vai datu struktūra, strādājot ar to, tiks izvietota pamatatmiņā vai diskatmiņā.

Piemēram: izveidots saraksts ar N elementiem, katrs saraksta elements ir kāds vārds, piemēram, *Alvars*. Sarakstā jāsamēklē

Datu struktūras projektējuma vērtēšana (2)

1. paņēmieni – saraksts izvietots pamatatmiņā:

```
const N = 500;           {elementu skaits  
sarakstā}  
  
type Name = string [8]; {saraksta  
elementu tips}  
  
    Arr = array [1 .. N] of Name;           {masīva  
tips}  
  
var List: Arr;  
{saraksts}  
  
    i: 0 .. N;  
    Test: Name;  
  
    i:= 0;  
    repeat                               {meklēšana  
sarakstā}
```

Datu struktūras

projektējuma vērtēšana (3)

2. paņēmieni – saraksts izvietots diskatmiņā:

```

const N = 500;           {elementu skaits
sarakstā}
type Name = string [8];  {saraksta
elementu tips}
FL = file of Name;       {faila
tips}
var List: FL;
{saraksts}
i: 0 .. N;
Test: Name;
i:= 0;
repeat                   {meklēšanā
sarakstā}
    i:= i+1; read (List, Test)
until (Test = 'Edgars') or (i = N);

```

Priekšrocības un trūkumi:

RTU akadēmiskās studiju programmas "Datorsistēmas" kursu pilnveidošana

2005/0125/VFD1/FSF/PIA/04/APK/3.2.3-2/0012/0007

35

1. paņēmieni – meklēšanas operācija ir ātrdarbīga, bet DS

Metrika, efektivitāte, veikspēja (metrics, efficiency, performance) (1)

Bieži izpildāmas operācijas:

- 1) meklēšanas operācija – reducējama uz salīdzināšanu. Operācijas izpildes ātrumu nosaka salīdzinājumu skaits;
- 2) elementa dzēšana sarakstā – reducējama uz elementu pārvietošanu par 1 pozīciju virzienā uz dzēšamo elementu.

Lai novērtētu algoritma efektivitāti izmanto:

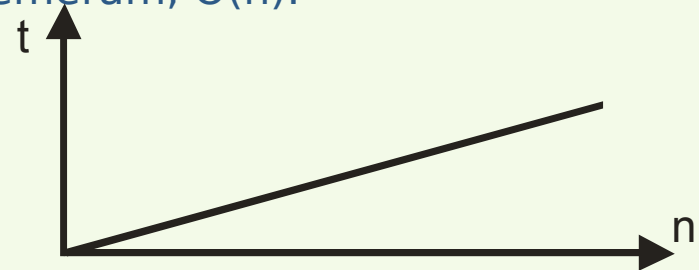
- 1) salīdzināšanas vai pārsūtīšanas operāciju skaitu;
- 2) kopējo operatoru skaitu, alternatīvo zarojumu daudzumu, cikla izvietojuma dziļumu;
- 3) pierakstu matemātiskās kārtas veidā, piemēram, $O(n)$.

Salīdzinājumu skaits meklēšanas procesā

$$E_c = \frac{n+1}{2}, \text{ kur } n - \text{elementu skaits.}$$

Meklēšanas laiks:

$$t = C_1 E_c + C_0 = C_1 \frac{n+1}{2} + C_0 = C'_1 n + C'_0 - \text{lineāra funkcija}$$



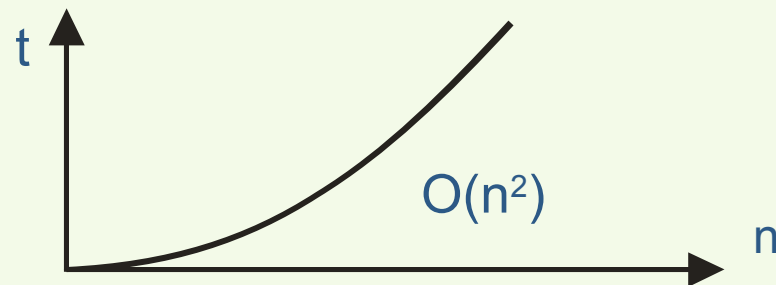
Izteiksmes lieluma kārtu nosaka tikai vislielākais operands, mazākie operandi vērtējumu būtiski neietekmē.

Metrika, efektivitāte, veiktspeja (efficiency, performance) (2)

Dažu izteiksmju kārtas:

$$\frac{n(n-1)}{2}$$

$$O(n^2)$$

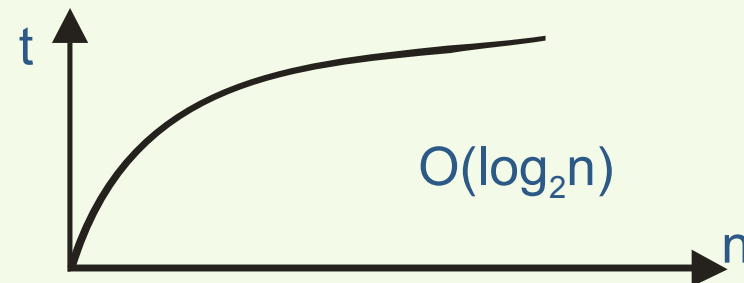


$$15 \log_2 n + 3n + 7$$

$$O(n)$$

$$2n \log_2 n + 0,1n^2 + 5$$

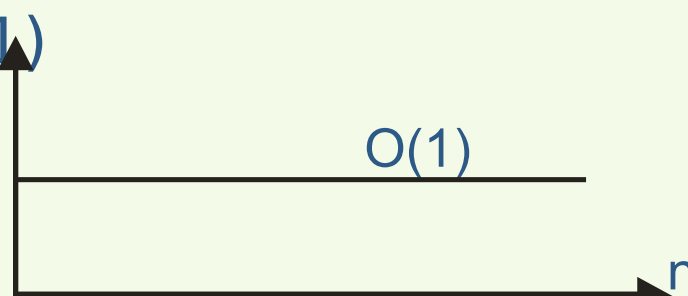
$$O(n^2)$$



$$\frac{6 \log_2 n + 3n + 7}{2n - 5}$$

$$2n - 5$$

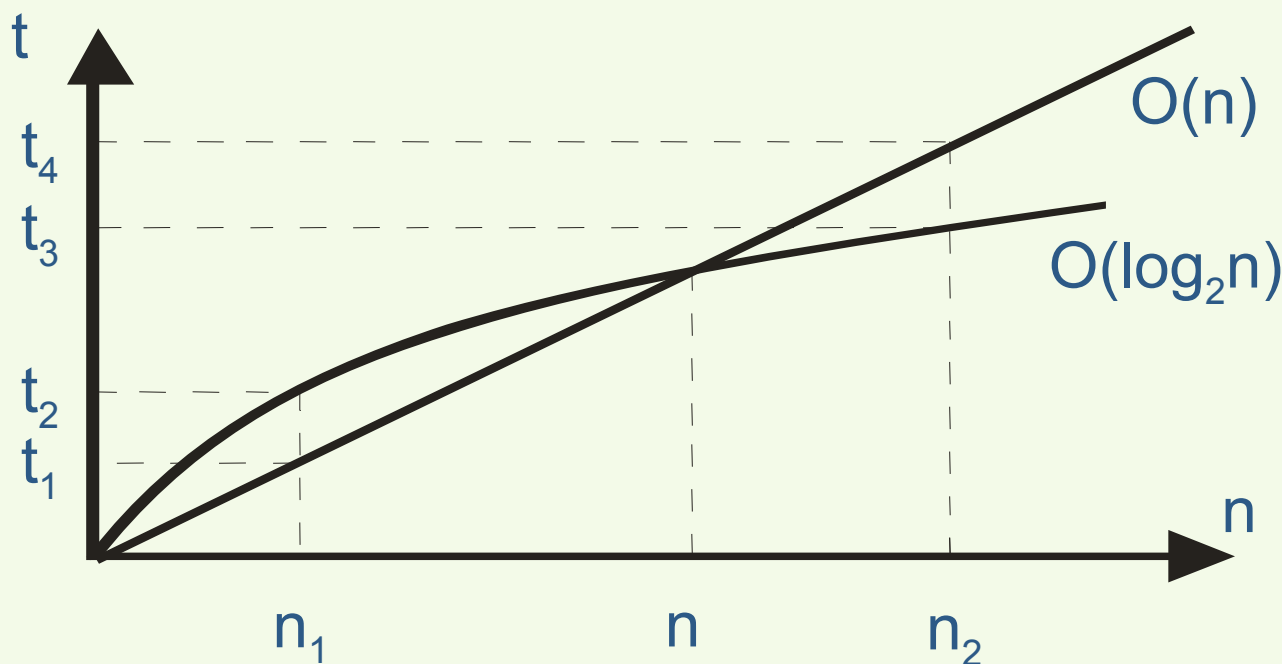
$$O(1)$$



Metrika, efektivitāte, veiktspeja (efficiency, performance) (3)

Kārta	$n = 8$	$n = 128$	$n = 1024$	$n = 10^6$
$O(n)$	8	128	$1024 = 2^{10}$	10^6
$O(n^2)$	64	16 384	$1\,048\,576 \approx 10^6$	10^{12}
—				
$O(\sqrt{n})$	≈ 3	≈ 11	32	10^3
$O(\log_2 n)$	3	7	10	20
$O(n \log_2 n)$	24	896	10 240	$2 \cdot 10^7$

Metrika, efektivitāte, veiktspeja (efficiency, performance) (4)



Meklēšanas laiks

$t_1 < t_2$ - ja elementu skaits neliels,

$t_3 < t_4$ - ja sarakstā daudz elementu.

Rakstzīmju attēlošana ar mainīgu bitu skaitu (1)

Ja bitu skaits rakstzīmes kodā $n = 8$, iespējams kodēt $2^8 = 256$ rakstzīmes.

Pieņemsim, ka ir teksts, kurā ir tikai 8 rakstzīmes.

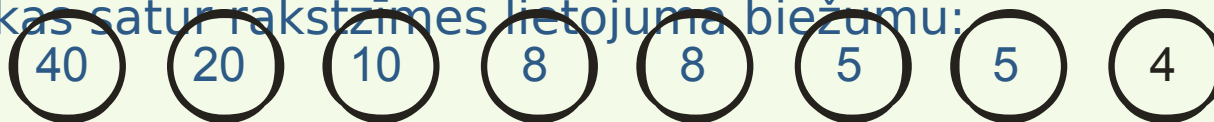
Tā kā $8 = 2^3$, tad šo tekstu ir iespējams kodēt arī tā, ka katrai rakstzīmei paredz tikai 3 bitu kombināciju.

Ir zināms šo 8 rakstzīmju lietojuma biežums (%):

a	b	c	d	e	f	g	h
40	20	10	8	8	5	5	4

Uzdevums: atrast tādu attēlojuma formu, lai šis teksts aizņemtu vismazāk vietas atmiņā.

Sāk ar to, ka izveido mežu, ko veido koki ar vienu vienīgu saknes virsotni, kas satur rakstzīmes lietojuma biežumu:

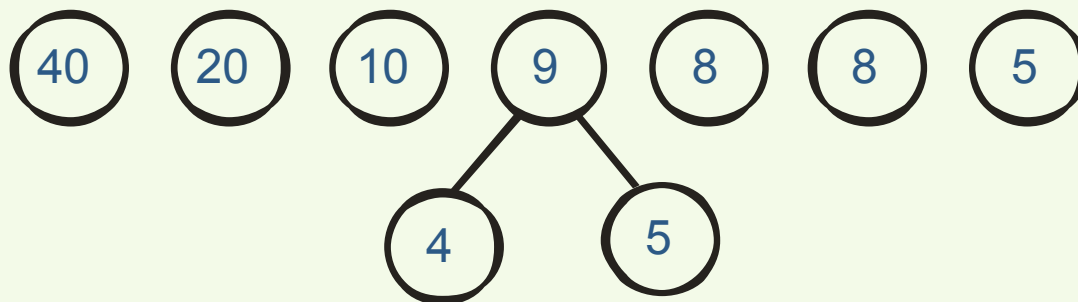


Virsotnes sakārtotas lietojuma biežuma dilšanas secībā.

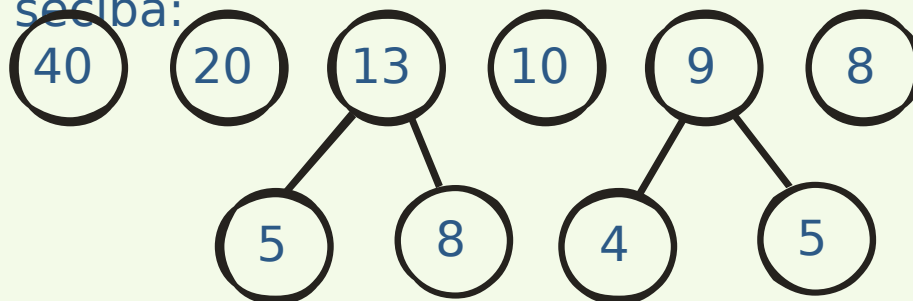
Rakstzīmju attēlošana ar mainīgu bitu skaitu (2)

Divas virsotnes, kurām ir viszemākais lietojuma biežums, apvieno binārajā kokā ar jaunu saknes virsotni un 2 zarojuma virsotnēm:

Kokus atkal sakārto virsotņu vērtību dilšanas secībā:

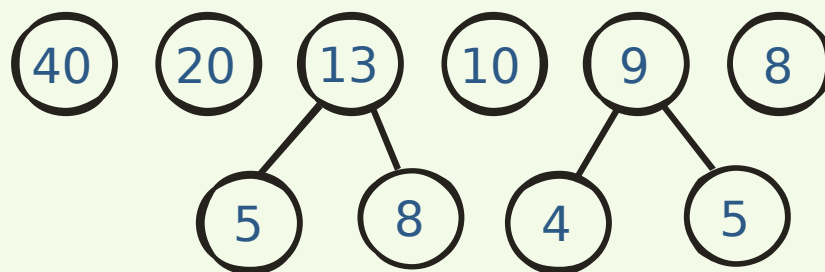


Procesu turpina, vēlreiz apvienojot 2 virsotnes ar viszemākajiem lietojuma biežumiem, pēc tam kokus atkal sakārto saknes virsotņu vērtību dilšanas secībā:

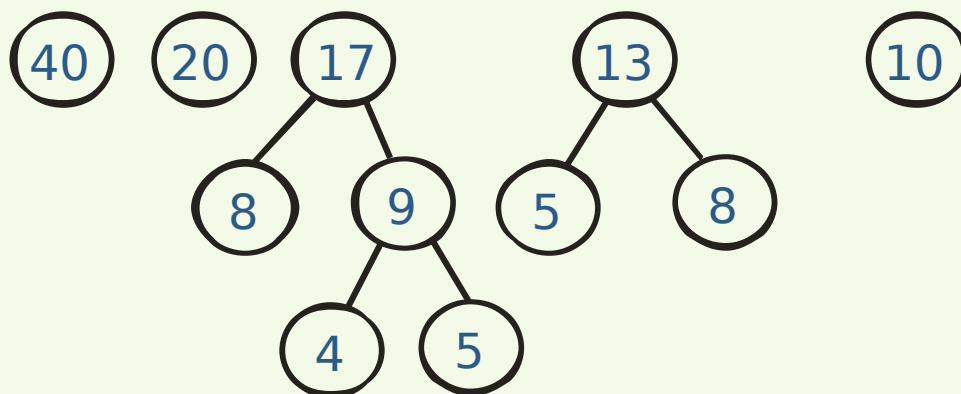


Rakstzīmju attēlošana ar mainīgu bitu skaitu (3)

Vēlreiz atkārtojot procesu, iegūst šādu bināro koku:



Vēlreiz atkārtojot procesu, iegūst nākamo bināro koku:

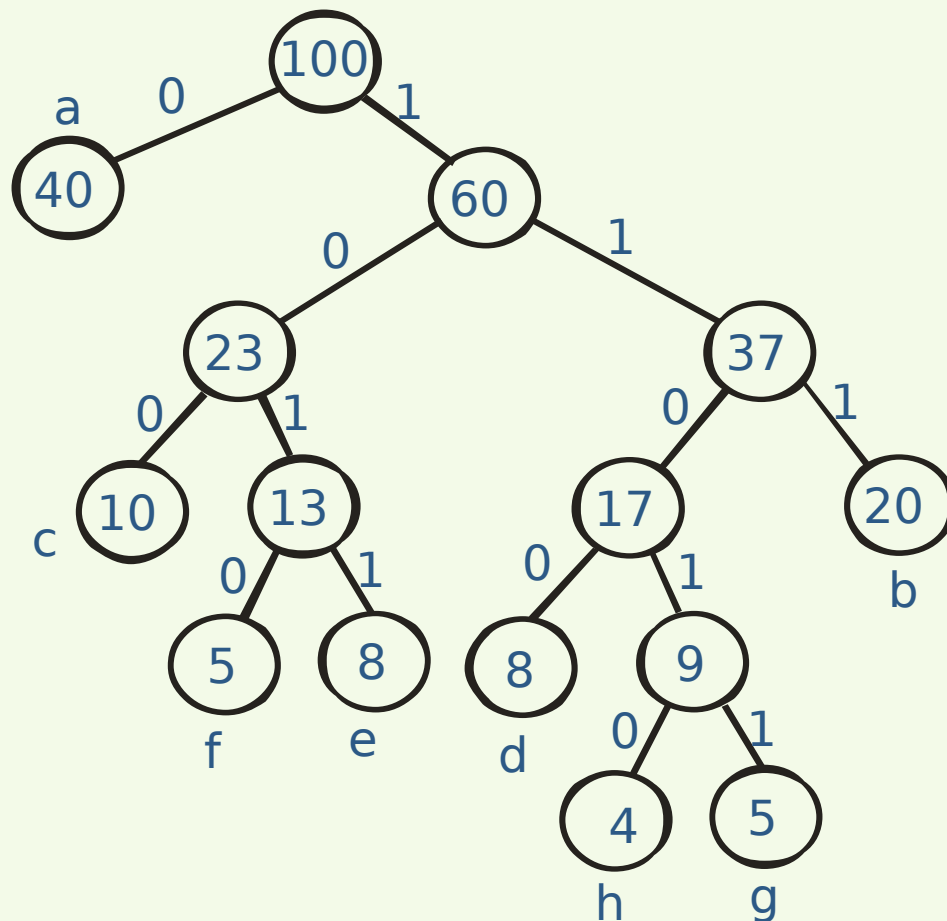


Rakstzīmju attēlošana ar mainīgu bitu skaitu (4)

Līdzīgā veidā procesu atkārtojot vēl 4 reizes, iegūst rezultējošo bināro koku:

Rakstzīmju
lietojuma
biežuma vērtības
atrodas
binārā koka lapu
virsoņos.

Binārajā kokā
katra kreisajai
šķautnei piešķir
vērtību 0, labajai – 1.



Rakstzīmju attēlošana ar mainīgu bitu skaitu (3)

Katras rakstzīmes attēlojuma kods ir bitu virkne
ceļā no saknes virsotnes uz lapas virsotni. Ir 8 šādi
ceļi:

rakstzīme	kods	lietojuma biežums
a	0	40
b	111	20
c	100	10
d	1100	8
e	1011	8
f	1010	5
g	11011	5
H	11010	4

Rakstzīmju attēlošana ar mainīgu bitu skaitu (4)

Iegūto kodu sauc par Hafmena (Huffman) kodu. Tam piemīt tāda īpašība, ka nevienas rakstzīmes kods nav vienāds ar prefiksu kādas citas rakstzīmes kodā. Tāpēc no Hafmena koda var iegūt oriģinālo 8 bitu kodu.

Ievietojot Hafmena kodu teksta kodēšanai, būtu nepieciešami

$$\frac{100}{100} (40 \cdot 1 + 20 \cdot 3 + 10 \cdot 3 + 8 \cdot 4 + 8 \cdot 4 + 5 \cdot 4 + 5 \cdot 5 + 4 \cdot 5) = 2,59n$$
 biti

Atmiņas ietaupījums:

$8n / 2,59n \approx 3$ – Hafmena kodu salīdzinot ar 8 bitu kodu,

$3n / 2,59n \approx 1,15$ – Hafmena kodu salīdzinot ar 3 bitu

kodu.

Rakstzīmju virknes jēdziens (1)

Valodā Pascal rakstzīmju virkni iespējams definēt divējādi:

1) kā mainīga garuma rakstzīmju virkni, virknes aprakstā izmantojot

predefinēto datu tipu **string**:

```
type Text1 = string;
```

```
Text2 = string [80];
```

```
var S: Text1;
```

```
Q: Text2 ;
```

```
S:= 'RTU'; Q:='RIGA';
```

```
S:= ''; read(Q); writeln(Q);
```

$1 \leq \text{maksimālais garums} \geq 255$

$0 \leq \text{tekošais garums} \leq \text{maksimālais garums}$



Rakstzīmju virknes jēdziens (2)

Tekošais garums aizņem 0. baitu, tā maksimālā vērtība:

$$11111111_2 = FF_{16} = 255_{10}$$

Tekošā garuma baits apstrādei tieši nav pieejams:

```
var S: string [80];
    L: byte absolute S;
    S:='ABC';    writeln(L, S);
```

2) **kā fiksēta garuma rakstzīmju virkni**, virknes aprakstā izmantojot

predefinēto datu tipu **array**:

```
type Text1 = array [1..255] of char;
```

```
    Text2 = array [1..80] of char;
```

```
var S:= Text1;
```

```
    Q:= Text2:
```

```
    S:= 'RTU';    Q:= 'RIGA';
```

```
    writeln(S, Q);
```

{izvadīs 335

rakstzīmes}

Rakstzīmju virknes tipa specifikācija (1)

Elementi: rakstzīmju virknes elementi ir alfabēta ASCII rakstzīmes. Katra rakstzīme atmiņā aizņem 1 baitu.

Struktūra: rakstzīmju virknes elementiem ir lineāra sasaiste. Katram elementam ir unikāla pozīcija rakstzīmju virknē. Pirmais elements atrodas 1. pozīcijā.

Domēns: visas iespējamās rakstzīmju kombinācijas ar garumu 0, 1, ..., MaxLength. Maksimālo garumu MaxLength definē kā konstanti, piemēram:

```
const MaxLength = 500;
```

Tipi: String – rakstzīmju virknes rādītāja tips,
StringPos = 1 .. MaxLength – rakstzīmju virknes
pozīcijas tips,
StringLen = 0 .. MaxLength – rakstzīmju virknes tekošā
garuma

tips.

Rakstzīmju virknes tipa specifikācija (2)

Operācijas:

Apkalpošanas operācijas

Create
Terminate
Length
Empty
Full

Pamatoperācijas

Append
Concatenate
Substring
Delete
Insert
Match
Find
ReadString
WriteString

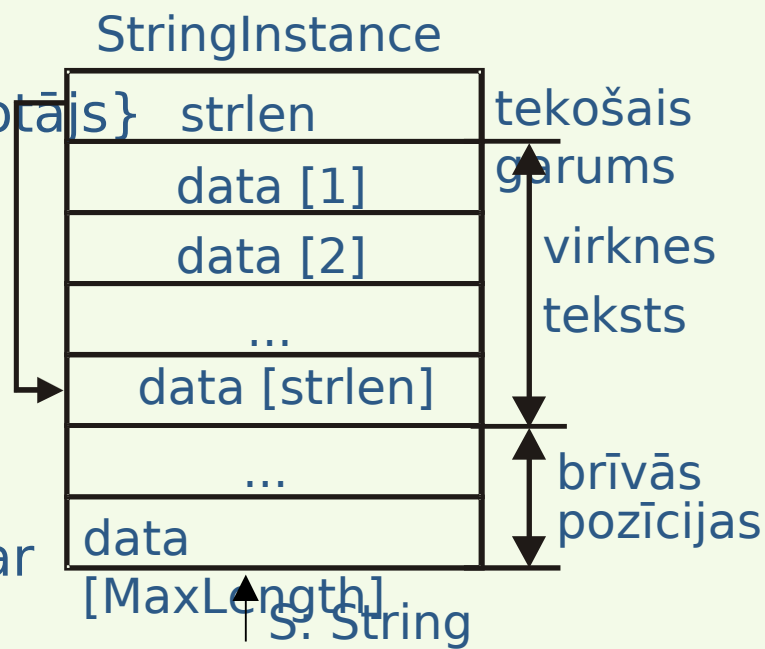
Papildoperācijas

MakeEmpty
Remove
Equal
Reverse
Polindrome
u.c.

Rakstzīmju virknes attēlojuma modeļi (1)

1.paņēmiens – modelī paredzēts speciāls lauks tekošā garuma attēlošanai, virknes attēlošanai izmanto vektoriālā formā attēlotu modeli, paredzot arī speciālu lauku tekošā garuma attēlošanai:

```
const MaxLength = 500; {uzdod lietotājs}
type StringLen = 0 .. MaxLength;
StringPos = 1 .. MaxLength;
String = ^ StringInstance;
StringInstance = record
    strlen: StringLen;
    data: array [StringPos] of char
end;
```

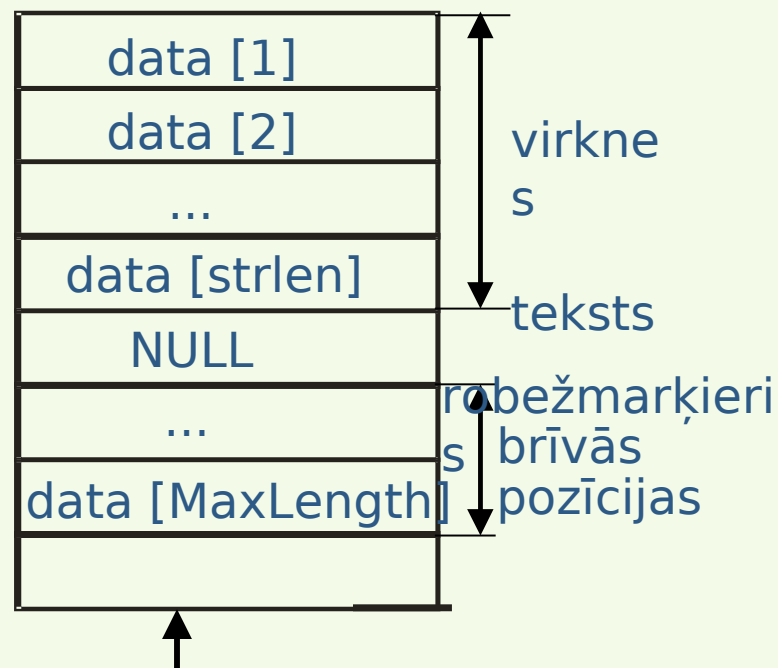


Rakstzīmju virknes attēlojuma modeļi (2)

2. paņēmieni – lieto vektoriālo attēlojuma formu, bet nav paredzēts lauks tekošā garuma attēlošanai. Aiz virknes pēdējās rakstzīmes ieraksta virknes beigu pazīmi. Parasti izmanto vadības rakstzīmi **NULL**, kuras kods ir 00_{16} . Var paredzēt arī tādu paņēmieni, ka viss pārpalikušais vektors tiek aizpildīts ar šo vadības rakstzīmi.

Robežmarķiera metode:

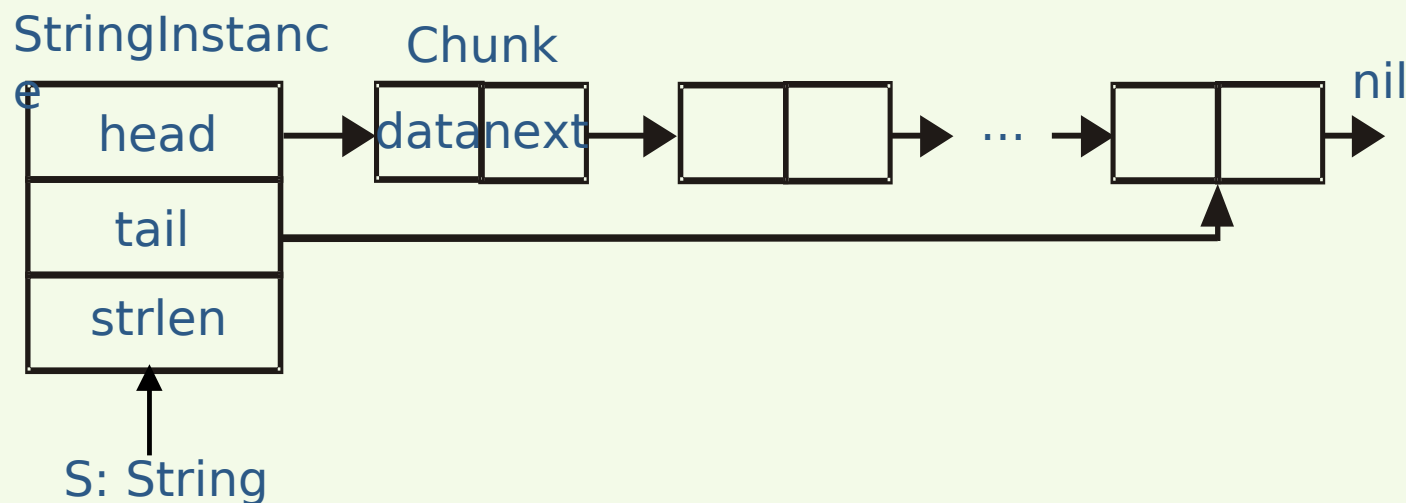
```
const MaxLength = 500;
type StringLen = 0 .. MaxLength;
    StringPos = 1 .. MaxLength;
    String = ^ StringInstance;
    StringInstance =
        array[1..MaxLength+1]
            of char;
```



s: String

Rakstzīmju virknes attēlojuma modeļi (3)

3. paņēmiens: virknes attēlošanai izmanto saistītā formā attēlotu modeli. Rakstzīmju virknes teksts tiek sadalīts fragmentos (chunk) ar vienādu garumu, izņemot pēdējo posmu, kas var būt arī īsāks.



Rakstzīmju virknes attēlojuma modeļi (4)

const MaxLength = 500;	{virknes maksimālais
garums}	
ChunkSize = 10;	{fragmenta
garums}	
type ChunkPos = 1.. ChunkSize	{fragmenta
pozīcijas tips}	
StringLen = 0 .. MaxLength;	{tekošā
garuma tips}	
ChPointer = ^ Chunk;	{elementu
rādītāja tips}	
Chunk = record	{virknes elementa
fragments}	
data: array [ChunkPos] of char;	
next: ChPointer	
end;	
String = ^ StringInstance;	{virknes rādītāja
tips}	
StringInstance = record	{virknes vadības
data: array [0..MaxLen-1] of char;	
next: ChPointer;	
end;	

53

Masīva jēdziens (1)

Masīvs (array) – vienkāršākais strukturētais datu tips.

Vēsturiski – pirmā programmēšanas valodā realizētā datu struktūra.

Masīvi – visbiežāk lietotās fundamentālās datu struktūras.

Masīvs – regulāra datu struktūra. Masīva elementi izvietoti dimensiju virzienā, katram elementam masīvā ir noteikts pozīcijas numurs, piemēram, $A[2, 1, 3]$.

Masīvs ir homogēna (viendabīga) datu struktūra. Visiem masīva elementiem ir vienāda uzbūve un viens un tas pats tips, ko sauc par bāzes tipu.

Masīvs – datu struktūra, kuras elementu pieejai izmanto brīvpiekļuves metodi (random access method). Apstrādei pieejams jebkurš elements jebkurā secībā, izmantojot indeksizteiksmes, piemēram,

$A[i, j, k]$

Masīvs - lineāra datu struktūra.

Masīva jēdziens (2)

Masīva tipa apraksts valodā Pascal:

```
type T = array [I] of B;  
var A: T;
```

I – indeksa tips, par to var būt tikai skalārs ordināls tips. Parasti to definē kā diapazona tipu ar indeksa augšējo un apakšējo robežvērtību:

lo .. hi

B – bāzes tips, par to var būt jebkurš datu tips, skalārs vai strukturēts datu tips, predefinēts vai lietotāja definēts datu tips.

Viendimensijas masīva X apraksta piemērs:

```
var X: array [1 .. 100] of real;  
                  I                  B
```

X [1] – masīva X pirmais elements,

X [100] – masīva X pēdējais elements,

X [i], i = 2, 3, ..., 99 - masīva X tekošais elements,

X [i+1] – tā pēctecis,

X [i-1] – tā priekštecis.

Masīva jēdziens (3)

Masīva elementu sasaistes raksturs: viens – ar – vienu.

Ir 3 pamatooperācijas masīva elementu apstrādei, pie kam 3. operācija ir arī realizējama, izmantojot pirmās divas.

```
type T = array [I] of B;
```

```
var X, Y: T;
```

```
C: B;
```

1) $C := X[i]$; kur i – tipam I atbilstoša indeksizteiksme.

{izguves operācija}

Retrieve}

2) $X[i] := e$; kur e – bāzes tipam B atbilstoša izteiksme.

{labošanas operācija}

Update}

3) $Y := X$; ekvivalents ar for $i := 1$ to n do $Y[i] := X[i]$;

{kopēšanas operācija}

Copy}

Viendimensijas masīvu sauc par vektoru.

Divdimensijas masīvu sauc par matricu.

Masīva dimensiju skaitu praktiski ierobežo datorresursi, teoretiski

Masīva jēdziens (4)

Piemēri:

- 1) type Row = array [1 .. 100] of real;
 Card = array [1 .. 80] of char;
 Vector = array [1 .. 15] of integer;
var A: Row;
 X, Y: Card;
 Q: Vector;

- 2) type Ind1 = 1 .. 10;
 Ind2 = 1 .. 12;
 Matrix = array [Ind1, Ind2] of real;
var M: Matrix;

- 3) type Ind1 = 1 .. 10;
 Ind2 = 1 .. 12;
 Matrix = array [Ind1] of array [Ind2] of real;
 I B
var M: Matrix;

Matricas jēdziens un interpretācija (1)

Valodā Pascal iespējami 2 matricas interpretācijas veidi:

1) matrica ir divdimensiju masīvs, kura elementi izvietoti rindās un kolonnās, šādi matrica tiek interpretēta matemātikā. Programmēšanas valodās matricas interpretācija ir plašāka.

```
const lo1 = 1; hi1 = 3;
```

```
lo2 = 1; hi2 = 4;
```

```
type Ind1 = lo1 .. hi1;
```

```
Ind2 = lo2 .. hi2; B = real;
```

```
Matrix = array [Ind1, Ind2] of B;
```

var A: Matrix;

Matricas jēdziens un interpretācija (2)

		→	j	
	A [1,1]	A[1,2]	A[1,3]	A[1,4]
↓ i	A [2,1]	A[2,2]	A[2,3]	A[2,4]
	A [3,1]	A[3,2]	A[3,3]	A[3,4]

Masīva elements ir mainīgais ar indeksiem:

A [i,j], $i = 1, 2, 3; \quad j = 1, 2, 3, 4.$

(vispārējā gadījumā $i = lo1, lo1+1, \dots, hi1,$
 $j = lo2, lo2+1, \dots, hi2).$

Katrai dimensijai jāuzdod sava indeksizteiksme.

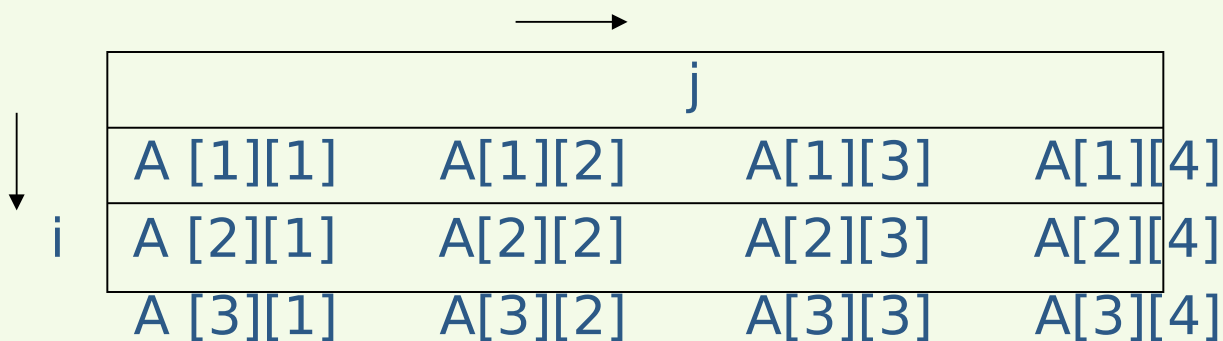
Mainīgais **A** – pārstāv visus masīva elementus, piemēram:
 writeln (A);

Matricas jēdziens un interpretācija (3)

2) matrica ir vektors, kura elementi savukārt ir vektori (array of array).

Šādi masīvu var interpretēt, piemēram, valodā Pascal:

```
const lo1 = 1; hi1 = 3;
      lo2 = 1; hi2 = 4;
type  Ind1 = lo1 .. hi1;
      Ind2 = lo2 .. hi2;  B = real;
      Matrix = array [Ind1] of array [Ind2] of B;
var   A: Matrix;
```



The diagram shows a 3x4 matrix A. A vertical arrow on the left points downwards and is labeled 'i', indicating the row index. A horizontal arrow at the top points to the right and is labeled 'j', indicating the column index. The matrix is represented as a table with three rows and four columns. Each cell in the table contains an element of the matrix, labeled as A[i][j], where i is the row index and j is the column index. The rows are indexed 1 to 3, and the columns are indexed 1 to 4.

	j			
i ↓	A [1][1]	A[1][2]	A[1][3]	A[1][4]
	A [2][1]	A[2][2]	A[2][3]	A[2][4]
	A [3][1]	A[3][2]	A[3][3]	A[3][4]

Matricas jēdziens un interpretācija (4)

Masīva elements ar ar vienu indeksizteiksmi:

A[i] , $i = lo1, lo1+1, \dots, hi1$, pārstāv visus elementus kādā rindā,

piemēram:

$A[3] := A[1];$ {masīva rindas
piešķire}

Mainīgais ar indeksiem **A[i] [j]** – pārstāv vienu noteiktu elementu, kas atrodas i-tā vektora (rindas) j-tā pozīcijā, $i = lo1, lo1+1, \dots, hi1$,

$j = lo2, lo2+1, \dots, hi2$.

Piemēram:

$A[lo1] [lo2] := 0;$

Mainīgais A – pārstāv visus masīva elementus, piemēram:

writein(A);

Elementa meklēšana vektorā (1)

Bieži lietota operācija darbā ar datu struktūrām.

Ir 3 meklēšanas algoritmi (metodes):

1) lineārā jeb secīgā meklēšana:

```
const N = 500;
type I = 1 .. N;  I1 = 0 .. N;
      B = integer;
      T = array [I] of B;
var A: T;
    k: I1;
    X: B;
```

atslēga}

{meklēšanas

...

```
k:= 0;
repeat
meklēšana}
```

{elementa

```
      k:= k + 1
until (A[k] = X) or (k = N);
if A[k] <> X then writeln ('Nesekmīga meklēšana')
else writeln (k, X);
```

Elementa meklēšana vektorā (2)

2) lineārā meklēšana, izmantojot robežmarkiera metodi:

```
const N = 500;
type I = 1 .. N + 1; I1 = 0 .. N + 1;
      B = integer;
      T = array [I] of B;
var A: T; k: I1; X: B;
```

...

```
A [N + 1] := X;
```

{robežmarkieris}

```
k := 0;
```

[meklēšana]

```
repeat
```

{elementa

```
k := k + 1
```

[until A [k] = X;

```
if k > N then writeln ('Nesekmīga meklēšana')
else writeln (k, X);
```

Lineārās meklēšanas operācijas izpildes efektivitāte, izmantojot

robežmarkieri, $O(n)$

Elementa meklēšana vektorā (3)

3) binārā meklēšana (dihotomijas metode):

```
const N = 500;
type I = 1 .. N;
      B = real;
      T = array [I] of B;
var A: T; i, j, k: I;
    X : B;
```

{meklēšanas

atslēga}

...

```
diapazons}
  i:= 1; j:= N;
```

{meklēšanas

```
  repeat
meklēšana}
```

{elementa

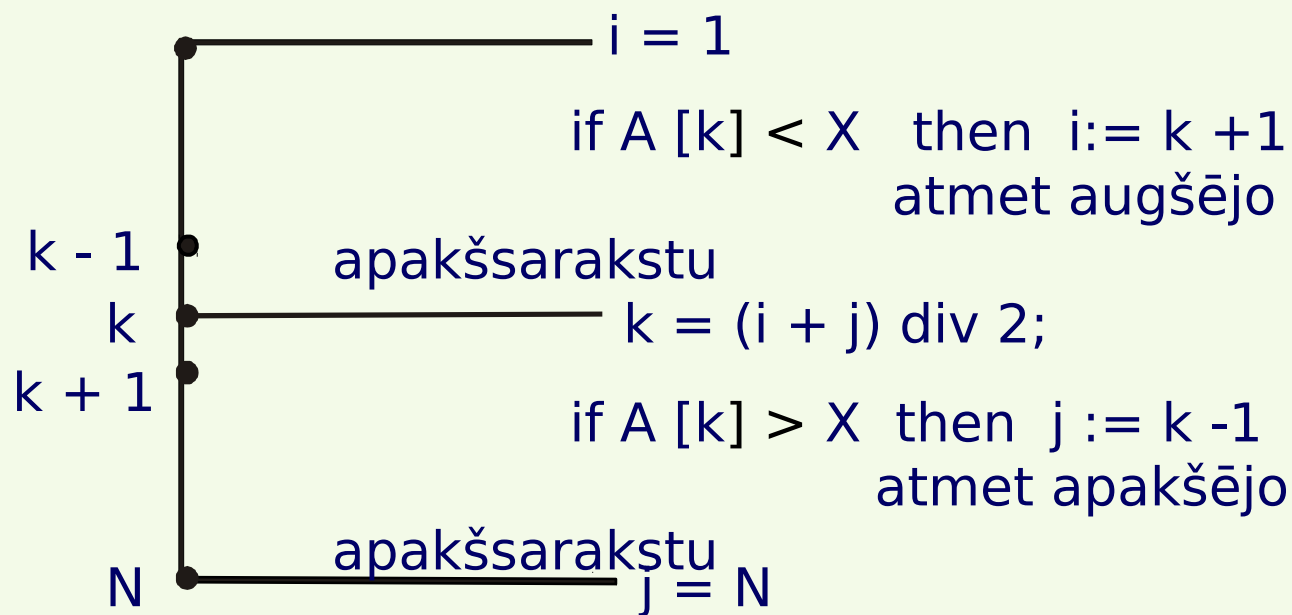
```
    k = (i + j) div 2;
{viduspunkts}
```

```
    if A [k] < X then i:= k + 1 {atmet augšējo
apakšsarakstu}
```

```
  else j:= k - 1 {atmet apakšējo
```

```
apakšsarakstu}
```


Elementa meklēšana vektorā (4)



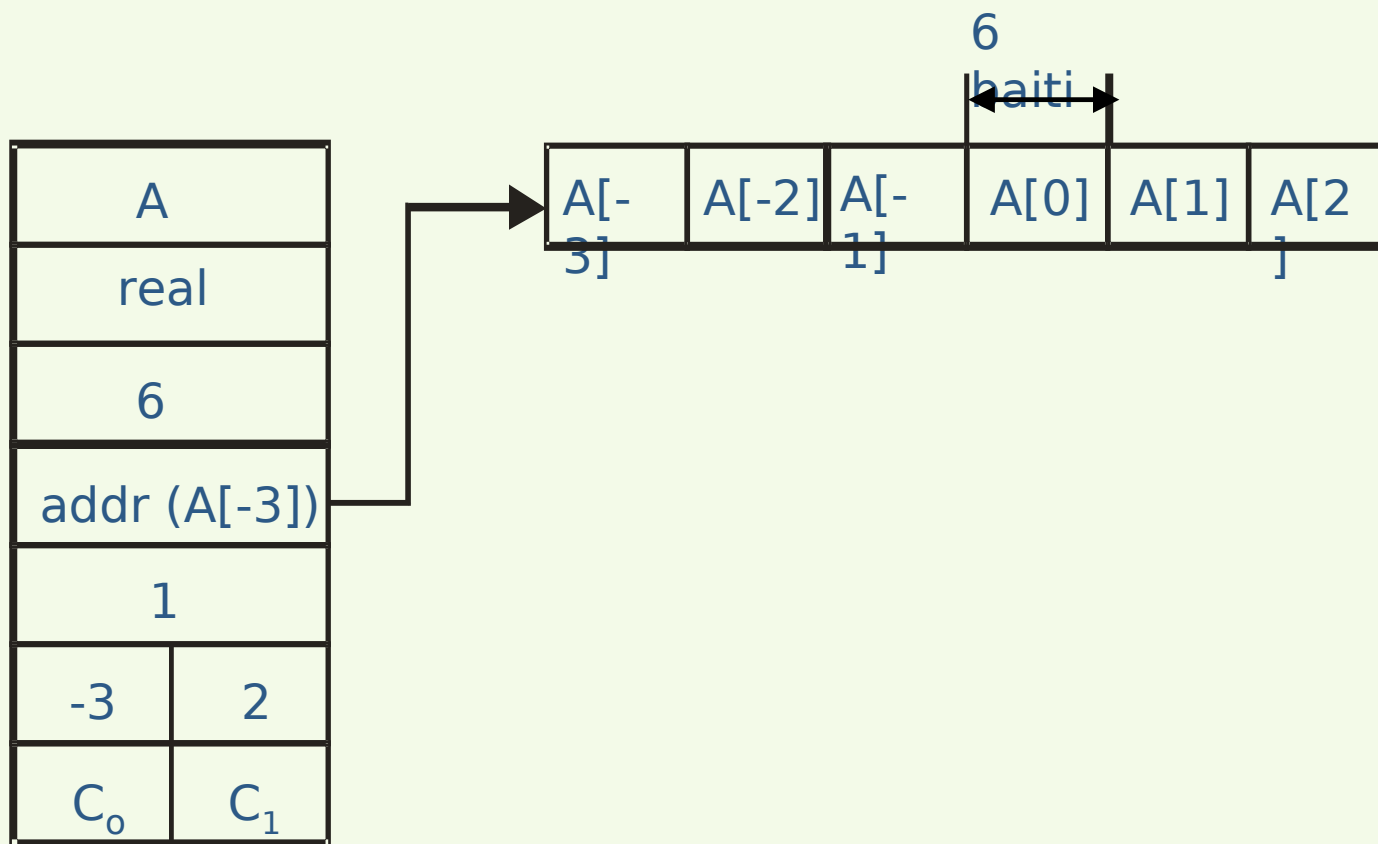
Deskriptors un tā lietojums (1)

Fiziskai datu struktūrai, kas ir masīvs, tiek piekārtots informatīvs ieraksts, ko sauc par deskriptoru un kurā tiek sakopotas vispārīgas ziņas par attiecīgo masīvu. Deskriptoru parasti izveido kompilators, un tas paredzēts, lai masīvu apstrādes procesā indeksizteiksmju vērtības pārveidotu fiziskas datu struktūras lauka adresēs.

Deskriptors ir ieraksts, kas sastāv no laukiem, kuru skaits, garums un raksturlielumi ir atkarīgi no masīva apraksta, piemēram:

```
var A: array [ -3 .. 2 ] of real;
```

Deskriptors un tā lietojums (2)



Vektora adresēšanas funkcijas noteikšana (1)

(address mapping function, AMF)

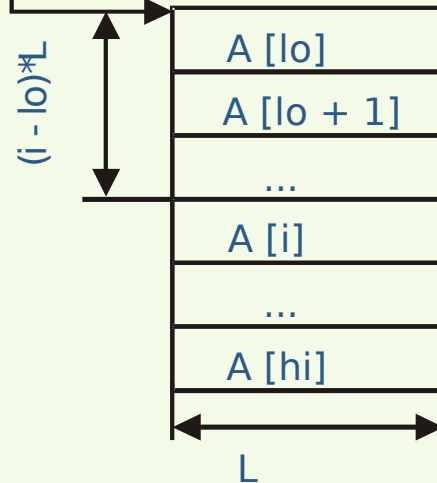
```
const lo = ... ;   hi = ... ;
    {uzdod lietotājs}
type   Ind = lo .. hi;
    {indeksa tips}
    B = ... ;
    {bāzes tips - uzdod lietotājs}
var   A: array [Ind] of B;

Elementu skaits
```

$$N = hi - lo + 1$$

A - vekt. nosaukums	
Bāzes tips B	
Elementa garums L	
Vektora sāk.adrese b	
Dimensiju skaits d	
lo	hi
C_0	C_1

parasti 1 - 8 rakstzīmes
parasti koda veidā
vesels skaitlis
adrese - vesels skaitlis
vesels skaitlis, $d = 1$
indeksa robežvērtības
adresēšanas funkcijas konstantes



$b + (i - lo) * L$ - attālums baitos līdz
 $(i - lo) * L$ - elementa attālums baitos no

$$\begin{aligned} \text{sākumadrese} + (i - lo) * L &= \\ &= (b - L * lo) + i * L = \mathbf{C_0 + C_1 * i} \end{aligned}$$

$$\begin{aligned} C &= L \\ C_0^1 &= b - lo * C_1 \end{aligned}$$

L
baiti

Vektora adresēšanas funkcijas noteikšana (2)

(address mapping function, AMF)

Piemēram: var A: array [3 .. 7] of integer;

lo = 3; hi = 7; L = 2;

Pieņemsim, ka b = 500.

$C_1 = L = 2$; $C_0 = b - lo * C_1 = 500 - 3 * 2 = 494$

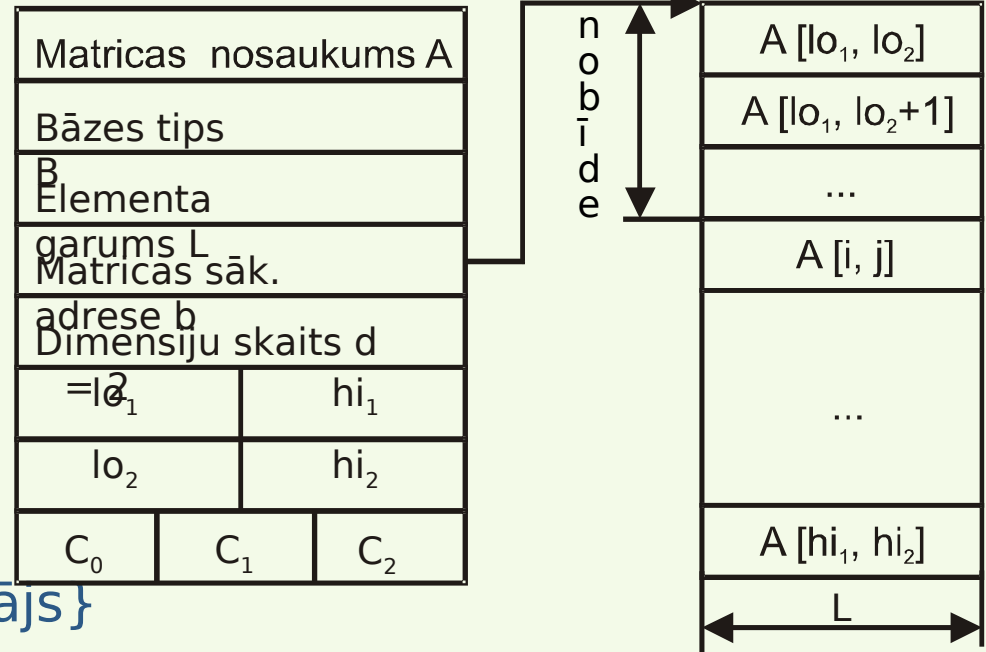
$\text{addr}(A[i]) = \mathbf{494 + 2i}$ - lineāra funkcija

	Adrese	Element s
$\text{addr}(A[3]) = 494 + 2 * 3 = 500$;	500	A [3]
$\text{addr}(A[5]) = 494 + 2 * 5 = 504$;	502	A [4]
$\text{addr}(A[7]) = 494 + 2 * 7 = 508$;	504	A [5]
	506	A [6]
	508	A [7]

Divdimensiju masīva adresēšanas funkcija (1)

```

const lo1 = ... ;    hi1 = ... ;
    {uzdod lietotājs}
    lo2 = ... ;    hi2 = ... ;
type  Ind1 = lo1 .. hi1;
    {indeksu tipi}
Ind2 = lo2 .. hi2;
B = ... ;
    {bāzes tips - uzdod lietotājs}
Matrix = array [Ind1, Ind2] of B;
var    A: Matrix;
  
```



Divdimensiju masīva adresēšanas funkcija (2)

$$\text{addr}(A[i, j]) = b + \underbrace{(i - \text{lo1}) (\text{hi2} - \text{lo2} + 1) * L}_{\text{attālums līdz i-tai rindai}} + \underbrace{(j - \text{lo2}) * L}_{\text{attālums līdz j-ajam elementam rindā i}}$$

$$= C_0 + C_1 * i + C_2 * j - \text{divargumentu lineāra}$$

funkcija

$$C_2 = L; \quad C_1 = (\text{hi2} - \text{lo2} + 1) * C_2; \quad C_0 = b - C_1 * \text{lo1} - C_2 * \text{lo2}$$

Piemēram:

```
const  lo1 = 1;   hi1 = 3;
       lo2 = 1;   hi2 = 4;
type   Ind1 = lo1 .. hi1;   Ind2 = lo2 .. hi2;
       B = real;
       T = array [Ind1, Ind2] of B;
var     A: T;
```

Divdimensiju masīva adresēšanas funkcija (3)

$$lo_1 = 1; \quad hi_1 = 3;$$

$$lo_2 = 1; \quad hi_2 = 4; \quad L = 6; \quad b = 500;$$

$$C_2 = L = 6;$$

$$C_1 = (hi_2 - lo_2 + 1) * C_2 = (4 - 1 + 1) * 6 = 24;$$

$$C_0 = b - C_1 * lo_1 - C_2 * lo_2 = 500 - 24 * 1 - 6 * 1 = 470.$$

$$\text{addr}(A[i,j]) = \mathbf{470 + 24i + 6j}$$

$$\text{addr}(A[1, 2]) = 470 + 24 * 1 + 6 * 2 = 506$$

$$\text{addr}(A[1, 1]) = 470 + 24 * 1 + 6 * 1 = 500$$

$$\text{addr}(A[3, 1]) = 470 + 24 * 3 + 6 * 1 = 548$$

$$\text{addr}(A[3, 4]) = 470 + 24 * 3 + 6 * 4 = 566$$

Elementu skaits $n = 12$.

Lauka garums $= 12 * 6 = 72$ baiti.

Pēdējā elementa adrese $= 500 + (72 - 6) = 566$.

A[1,1]	b = 500
A[1,2]	506
A[1,3]	512
A[1,4]	518
A[2,1]	524
...	
A[3,1]	548
...	
A[3,4]	566

Vairākdimensiju masīvi un to adresēšanas funkcijas (1)

Masīva dimensiju skaits – to praktiski ierobežo tikai datorsistēmas arhitektūra un resursi.

Atmiņas apjoms un tā apstrādes laiks strauji pieaug, pieaugot dimensiju skaitam.

Ja definēts masīvs ar **d** dimensijām un **L** baitiem viena elementa attēlošanai atmiņā, tad viss masīvs atmiņā aizņems

$L * (hi_1 - lo_1 + 1) * (hi_2 - lo_2 + 1) * \dots * (hi_d - lo_d + 1)$ baitus,
piemēram:

```
var A: array [1 .. 100, 1 .. 100, 1 .. 4] of integer;
L = 2; d = 3. Masīvs atmiņā aizņems apmēram 80 000 baitus.
```

Uzskatīsim, ka vispārējā gadījumā definēts šāds vairākdimensiju masīvs:

```
var A: array [lo1 .. hi1, ..., lod .. hid] of B;
```

Vairākdimensiju masīvi un to adresēšanas funkcijas (2)

Mēģināsim vispārināt adresēšanas funkcijas konstanšu C_0, C_1, \dots un C_d noteikšanas metodiku un formulas:

$$C_d = L \quad (\text{elementu garums baitos})$$

...

$$C_{k-1} = (hi_k - lo_k + 1) * C_k, \quad k = d, d-1, \dots, 2$$

...

$$C_0 = b - C_1 * lo_1 - C_2 * lo_2 - \dots - C_d * lo_d$$

$$\text{addr} (A [i_1, i_2, \dots, i_d]) = C_0 + C_1 * i_1 + C_2 * i_2 + \dots + C_d * i_d$$

Vairākdimensiju masīvu elementi datora atmiņā tiek izvietoti viens aiz otra tā, ka visstraujāk izmainās pēdējais indekss, bet vislēnāk – pirmais indekss.

Vairākdimensiju masīvi un to adresēšanas funkcijas (3)

Piemērs:

Definēts trīsdimensiju masīvs:

var A: array [1 .. 2, 1 .. 2, 1 .. 2] of integer;

N = 8; Atmiņas lauka garums ir $8*2 = 16$ baiti.

b = 500; L = 2; d = 3.

Adresēšanas funkcijas konstantes:

$$C_3 = L = 2;$$

$$C_2 = (hi_3 - lo_3 + 1) * C_3 = 2 * 2 = 4;$$

$$C_1 = (hi_2 - lo_2 + 1) * C_2 = 2 * 4 = 8;$$

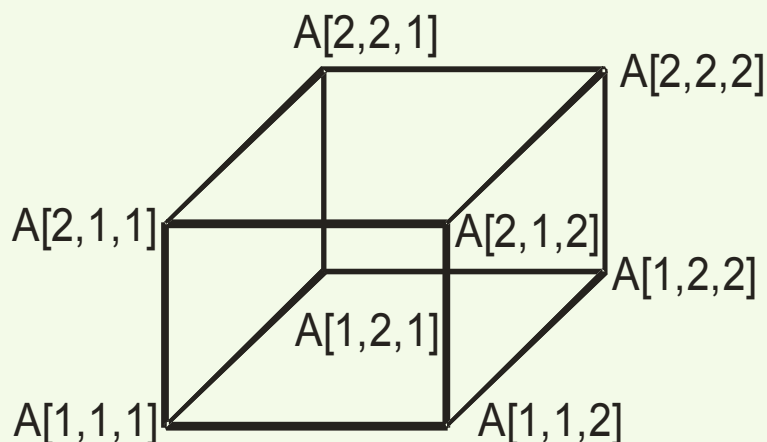
$$C_0 = b - C_1 * lo_1 - C_2 * lo_2 - C_3 * lo_3 = 500 - 8 * 1 - 4 * 1 - 2 * 1 = 486;$$

Adresēšanas funkcija:

$$\text{addr}(A[i, j, k]) = 486 + 8i + 4j + 2k;$$

Trīsdimensiju masīva attēlojums

Masīva loģiskā struktūra



Fiziskā struktūra

500	A[1,1,1]
502	A[1,1,2]
504	A[1,2,1]
506	A[1,2,2]
508	A[2,1,1]
510	A[2,1,2]
512	A[2,2,1]
514	A[2,2,2]

$$\text{addr}(A[1,1,2]) = 486 + 8 \cdot 1 + 4 \cdot 1 + 2 \cdot 2 = 502;$$

$$\text{addr}(A[2,2,2]) = 486 + 8 \cdot 2 + 4 \cdot 2 + 2 \cdot 2 = 514;$$

$$\text{addr}(A[1,1,1]) = 486 + 8 \cdot 1 + 4 \cdot 1 + 2 \cdot 1 = 500;$$

Speciālie masīvi un to lietojums

Pie speciālajiem masīviem pieskaitāmi:

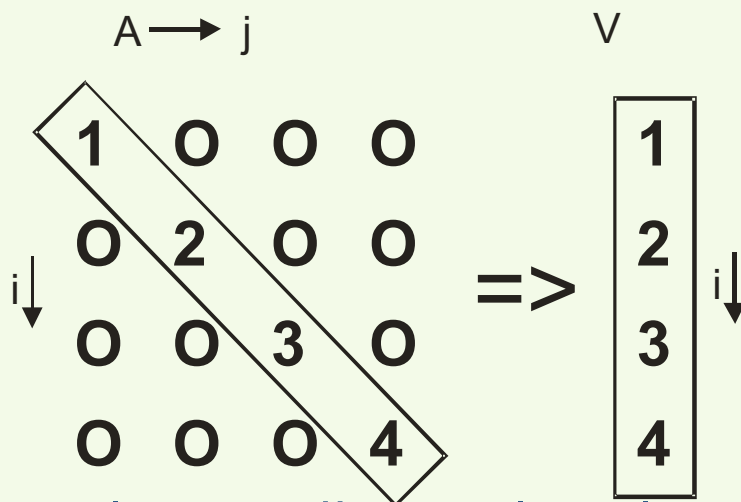
- 1) diagonālmaticas;
- 2) simetriskās matricas;
- 3) trijstūrmaticas;
- 4) retinātās matricas.

Galvenās risināmās problēmas:

- 1) kā visefektīvāk speciālo masīvu attēlot datora atmiņā;
- 2) kā visefektīvāk organizēt piekļuvi masīva elementiem un izpildīt to meklēšanas operāciju.

Diagonālmatrixa (1)

(diagonal array)



$$A[i,j] = 0,$$

$$A[i,i] \neq 0,$$

$$i = lo, lo+1, \dots, hi,$$

$$j = lo, lo+1, \dots, hi.$$

Galvenās diagonāles elementu skaits: $N = hi - lo + 1$.

Diagonālmatrixas modeļa apraksts un meklēšanas operācija:

```

const lo = 1; hi = 4;                                {uzdod
lietotājs}
type Ind = lo .. hi;                                  {indeksa
tips}
B = integer;                                           {matricas elementa
tips}
DATA = array [Ind] of B;                               {vektora
tips}

```

Diagonālmatrixa (2)

(diagonal array)

```
function DArrFind (V: DArr; i, j: Ind): B;  
  {Diagonālmatrixas A elementa meklēšana vektorā V,  
  izmantojot indeksu i un j vērtības}  
begin  
  if i = j then DArrFind:=V[i]  
    else DArrFind:=0  
end;
```

Masīva elementa $A[i,j]$ vietā lieto funkcijas izsaukumu:

$A[i,j] \Rightarrow \text{DArrFind}(V, i, j)$

Simetriskā matrica (1)

(symmetrical array)

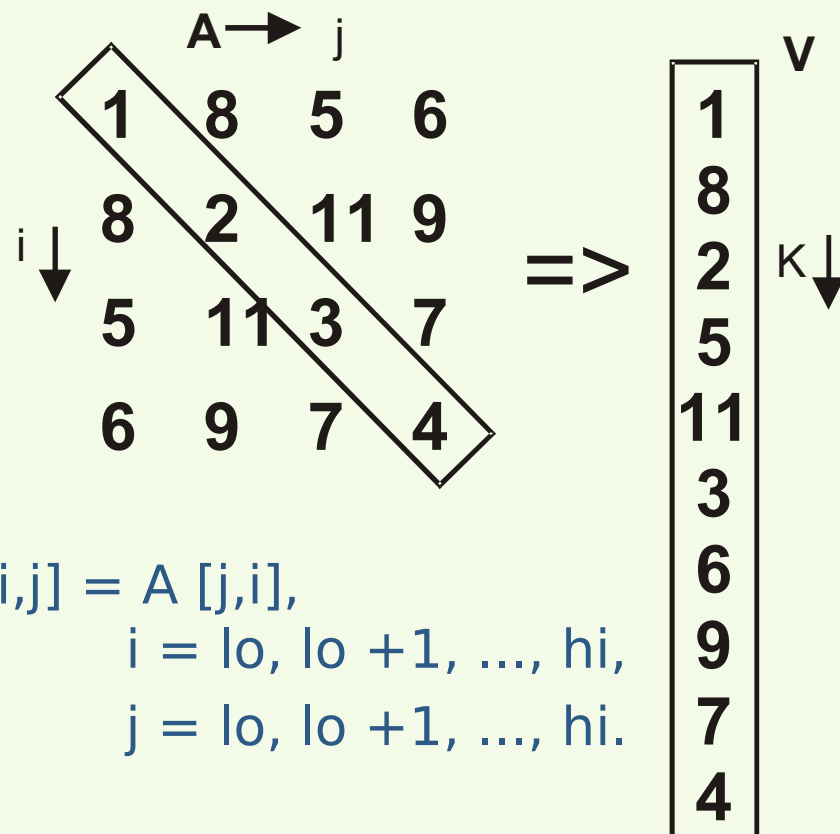
Elementu skaits vektorā V (t.i., simetriskās trijstūrmaticā):

$$N = \frac{(hi - lo + 1)(hi - lo + 2)}{2}$$

Ja $lo=1$, tad
$$N = \frac{hi(hi+1)}{2}$$

Piemērā $lo = 1$, $hi = 4$,

$$N = \frac{4 * 5}{2} = 10.$$



Simetriskā matrica (2)

(symmetrical array)

Simetriskās matricas modeļa apraksts un meklēšanas operācija:

```
const lo = 1; hi = 4;    {indeksu i un j robežvērtības – uzdod  
lietotājs}  
hk = (hi - lo + 1) * (hi - lo + 2) div 2;    {indeksa k  
robežvērtība}  
type Ind = lo .. hi;    {indeksu i un j datu  
tips}  
Indk = lo .. hk;    {indeksa k datu  
tips}  
B = integer;    {simetriskās matricas elementa  
datu tips}  
SymArr = array [Indk] of B;    {vektora V  
tips}  
var V: SymArr;  
function SymArrFind (V: SymArr; i, j: Ind): B;  
{Simetriskās matricas A elementa meklēšana vektorā V,  
indeksu i un j vērtībās}
```

Simetriskā matrica (3)

(symmetrical array)

Masīva elementa $A[i,j]$ vietā, izpildot dažādas darbības ar simetriskās matricas elementiem, lieto funkcijas izsaukumu:

$A[i,j] \Rightarrow \text{SymArrFind}(V, i, j)$.

Pārbaude:

$$A[1,1] \quad i = 1, j = 1, \quad k = 1 + (1^2 - 1) / 2 = 1;$$

$$A[1,2] \quad i = 1, j = 2, \quad k = 1 + (2^2 - 2) / 2 = 1 + 2 / 2 = 2;$$

$$A[2,1] \quad i = 2, j = 1, \quad k = 1 + (2^2 - 2) / 2 = 1 + 2 / 2 = 2;$$

$$A[4,4] \quad i = 4, j = 4, \quad k = 4 + (4^2 - 4) / 2 = 4 + 12 / 2 = 10;$$

Apakšējā trīsstūrmatrixa (1)

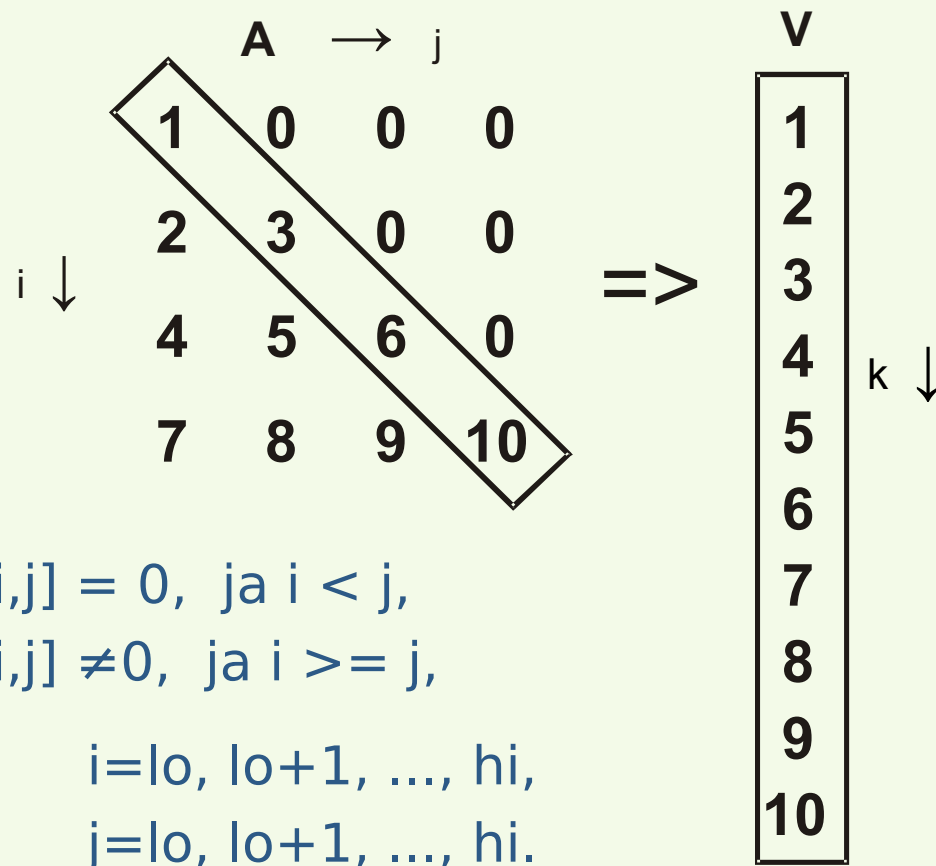
(lower triangular array)

Nesingulāro elementu skaits

$$N = \frac{(hi - lo + 1)(hi - lo + 2)}{2}$$

Ja $lo = 1$, tad

$$N = \frac{hi(hi + 1)}{2}$$



Apakšējā trīsstūrmatrixa (2)

(lower triangular array)

Apakšējās trijstūrmatrixas modeļa apraksts un meklēšanas operācija:

```

const lo = 1; hi = 4;    {indeksu i un j robežvērtības – uzdod lietotājs}
hk = (hi - lo + 1)*(hi - lo + 2) div 2;    {indeksa k robežvērtība}
type Ind = lo .. hi;    {indeksu i un j datu tips}
Indk = lo .. hk;    {indeksa k datu tips}
B = integer;    {apakšējās trīsstūrmatrixas elementa datu tips}
LTArr = array [Indk] of B;    {vektora V datu tips}
var V: LTArr;
function LTArrFind (V: LTArr; i, j: Ind): B;
{Apakšējās trīsstūrmatrixas A elementa meklēšana vektorā V, izmantojot indeksu i un j vērtības}
var k: Indk;
begin
  if i >= j then

```

Apakšējā trīsstūrmatrixa (3)

(lower triangular array)

Masīva elementa $A[i,j]$ vietā, izpildot dažādas darbības ar apakšējās trīsstūrmatrixas elementiem, lieto funkcijas izsaukumu:

$A[i, j] \Rightarrow \text{LTArrFind}(V, i, j)$

Pārbaude:

$A[1,1] \quad i = 1, j = 1,$

$$k = \frac{1*(1-1)}{2} + 1 = 1;$$

$A[3, 2] \quad i = 3, j = 2,$

$$k = \frac{3*2}{2} + 2 = 5;$$

$A[3, 3] \quad i = 3, j = 3,$

$$k = \frac{3*2}{2} + 3 = 6;$$

$A[4, 4] \quad i = 4, j = 4,$

$$k = \frac{4*3}{2} + 4 = 10;$$

Augšējā trīsstūrmatrixa (1)

(upper triangular array)

Nesingulāro elementu skaits:

$$N = \frac{(hi-lo+1)(hi-lo+2)}{2}$$

Ja $lo = 1$, tad

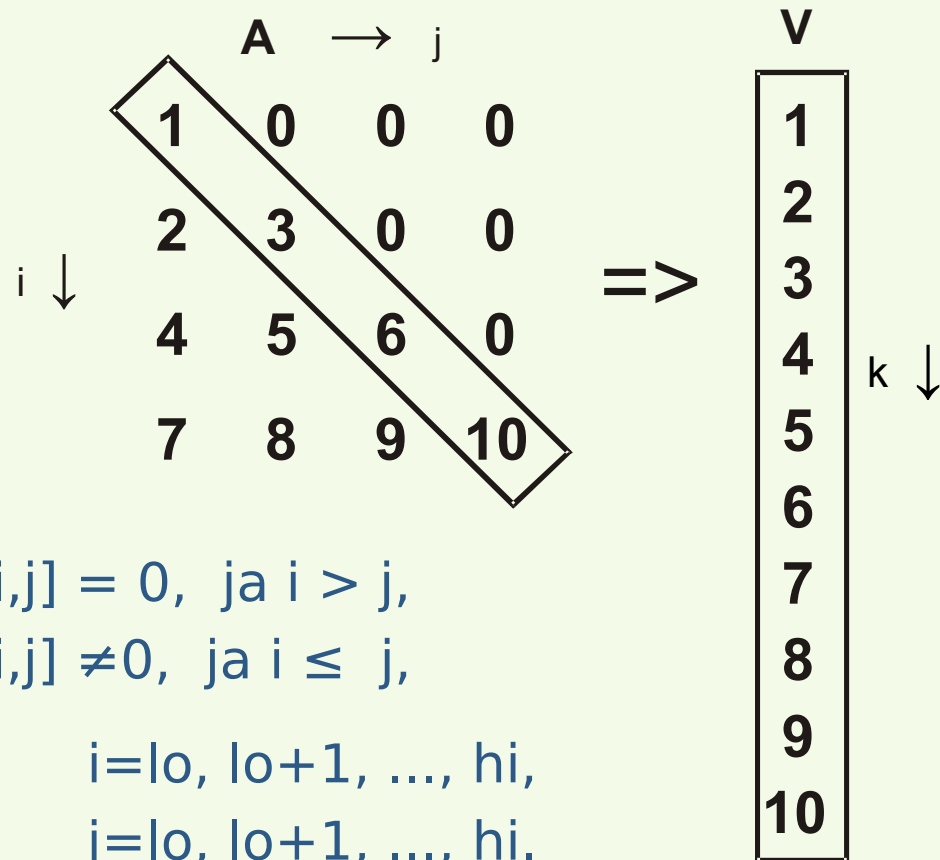
$$N = \frac{hi(hi+1)}{2}$$

$A[i,j] = 0$, ja $i > j$,

$A[i,j] \neq 0$, ja $i \leq j$,

$i = lo, lo+1, \dots, hi$,

$j = lo, lo+1, \dots, hi$.



Augšējā trijstūrmatrixa (2)

(upper triangular array)

Apakšējās trijstūrmatrixas modeļa apraksts un meklēšanas operācija:

```

const lo = 1; hi = 4;    {indeksu i un j robežvērtības – uzdod lietotājs}
hk = (hi - lo + 1)*(hi - lo + 2) div 2;    {indeksa k robežvērtība}
type Ind = lo .. hi;    {indeksu i un j datu tips}
Indk = lo .. hk;    {indeksa k datu tips}
B = integer;    {augšējās trīsstūrmatrixas elementa datu tips}
LTArr = array [Indk] of B;    {vektora V datu tips}
var V: UTArr;
function UTArrFind (V: UTArr; i, j: Ind): B;
{Augšējās trijstūrmatrixas A elementa meklēšana vektorā V, izmantojot indeksu i un j vērtības}
var k: Indk;
begin

```

Augšējā trijstūrmatrixa (3) (upper triangular array)

Masīva elementa $A[i,j]$ vietā, izpildot dažādas darbības ar
aupšējās trīsstūrmatrixas elementiem, lieto funkcijas izsaukumu:

$$A[i, j] \Rightarrow \text{UTArrFind}(V, i, j)$$

Pārbaude:

$$A[1,1], \quad i = 1, j = 1; \quad k = ((2*4 - 1 + 1) * 1) / 2 - 4 + 1 = 4 - 4 + 1 = 1$$

$$A[1,2], \quad i = 1, j = 2; \quad k = ((2*4 - 1 + 1) * 1) / 2 - 4 + 2 = 4 - 4 + 2 = 2$$

$$A[2,2], \quad i = 2, j = 2; \quad k = ((2*4 - 2 + 1) * 2) / 2 - 4 + 2 = 7 - 4 + 2 = 5$$

$$A[3,4], \quad i = 3, j = 4; \quad k = ((2*4 - 3 + 1) * 3) / 2 - 4 + 4 = 9 - 4 + 4 = 9$$

Retinātā matrica (1)

(sparse array)

Masīvu, kurā vairums elementu ir vienādi ar kādu singulāru vērtību, (piemēram, ar nulli), sauc par retinātu matricu. Tikai dažas vērtības ir nesignulāras, un tās matricā izvietotas nevienmērīgi.

Ja definēta retināta matrica:

```
var A: array [1 .. hi1, 1 .. hi2] of integer;
```

un N_z elementi nav vienādi ar 0 ($N_z = 1, 2, \dots$), tad pārējie singulārie elementi atmiņā aizņemtu $(hi1 * hi2 - N_z) * L$ baitus, ja izmanto parasto masīva elementu izvietojanas paņēmieni datora atmiņā.

Retinātā matrica (2)

(sparse array)

Piemērs:

A $\rightarrow j = 1..7$

$i = 1..6$ ↓

0	0	6	0	9	0	0
2	0	0	7	8	0	4
10	0	0	0	0	0	0
0	0	12	0	0	0	0
0	0	0	0	0	0	0
0	0	0	3	0	0	5

No 42 elementiem tikai $N_z = 10$ nav vienādi ar 0.

Viens no paņēmieniem, kā glabāt retināto matricu atmiņā, ir izveidot ierakstu vektoru. Katrs vektora elements satur nesingulāro vērtību vai un tai atbilstošo indeksu i un j vērtības.

Retinātā matrica (3) (sparse array)

Vektora indekss	val	i	j
1	9	1	3
2	6	1	5
3	2	2	1
4	7	2	4
5	8	2	5
6	4	2	7
7	10	3	1
8	12	4	3
9	3	6	4
10	5	6	7

Nz= 10;

v
↓ K = 1 .. Nz

Retinātā matrica (4)

(sparse array)

Retinātās matricas modeļa apraksta un meklēšanas operācija:

```

const NzMax = 20;           {nesingulāro vērtību skaits - uzdod
lietotājs}

    lo1 = 1;  hi1 = 6;           {indeksa i
robežvērtības}

    lo2 = 1;  hi2 = 7;           {indeksa j
robežvērtības}

type   Ind1 = lo1 .. hi1;           {indeksa i datu
tips}

    Ind2 = lo2 .. hi2;           {indeksa j datu
tips}

    NzRange = 0 .. NzMax;           {vektora indeksa k
datu tips}

    B = integer;                   {retinātās matricas
elementa tips}

    Condensed = array [NzRange] of record {ierakstu
vektora tips}

```

val: B;
i: Ind1

Retinātā matrica (5)

(sparse array)

```
function SparseFind (V: Condensed; i: Ind1; j: Ind2): B;  
{Retinātas matricas elementa A[i,j] meklēšana vektorā V}  
var k: NzRange;  
begin  
    k:= 0;  
    repeat    k:= k + 1  
    until ((V[k] . i = i) and (V[k] . j = j)) or (k = NzMax);  
    if ((V[k] . i = i) and (V[k] . j = j)) then SparseFind:= V[k] .  
val  
                                                else SparseFind:= 0  
end;
```

Masīva elementa A[i,j] vietā, operējot ar to, lieto funkcijas izsaukumu:

$A[i,j] \Rightarrow \text{SparseFind}(V, i, j)$

Retinātās matricas (6)

(sparse arrays)

Piekluve:

Matricas elementu gadījumi piekluves vietā notiek elementa meklēšana ierakstu vektorā. Vidēji nepieciešamas $(N_z + 1) / 2$ caurskates sekmīgas meklēšanas gadījumā.

Trūkumi:

- 1) iepriekš jāuzdod N_z Max vērtība, kas katras retinātas matricas gadījumā ir atšķirīga un grūti prognozējama;
- 2) retinātas matricas attēlojuma modelis faktiski ir tabula ar 3 vērtībām katra elementa rindā, vismaz 2 baiti nepieciešami laukiem i un j .

Attēlojuma efektivitātes novērtējums:

Ja lauku val , i un j garums ir L baiti ($L=2$), tad katrs vektora elements atmiņā aizņems $3 * L$ baitus, un tiks ietaupīti $(hi1 * hi2 - N_zMax) * 3 * L$ baiti atmiņas.

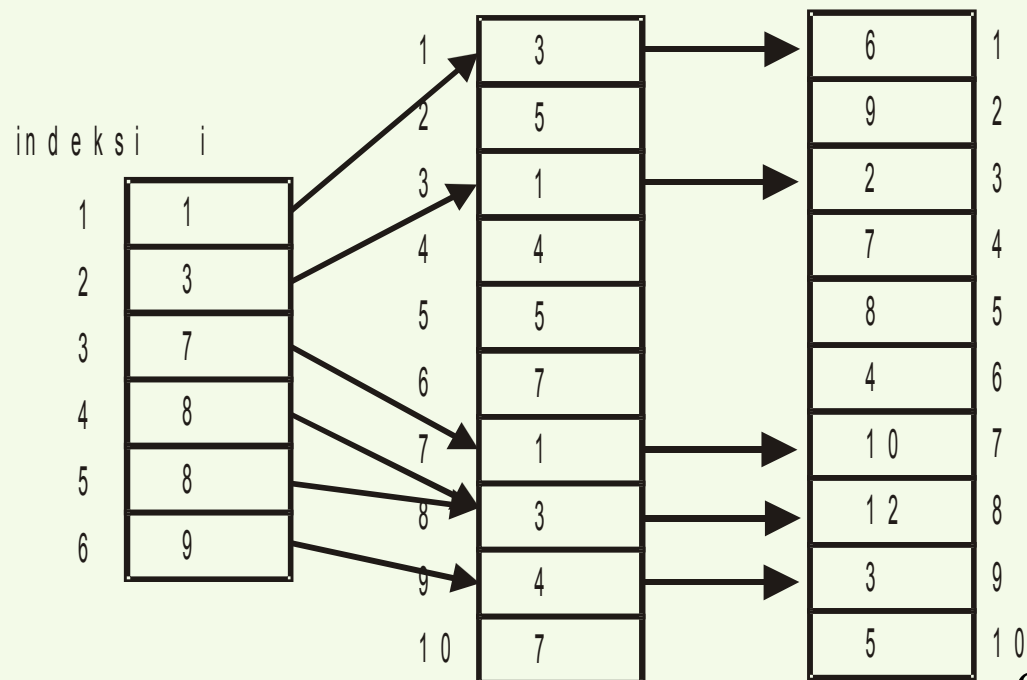
Atmiņā ietaupījums baitos vispārējā gadījumā:

$$(hi1 * hi2 - N_zMax) * (2 * L1 + L2),$$

Retinātā matrica (7) (sparse array)

2. paņēmiens: Retinātās matricas attēlojuma modeli veido trīs atsevišķi vektori. Vektorā i tiek attēlots pirmās nesingulārās vērtības kārtas numurs katrā rindā, skaitot pa rindām uz priekšu, vektorā j – katras nesingulārās vērtības val kolonnas indekss rindā i , vektorā val – retinātās matricas nesingulārās vērtības.

Retinātajā matricā
ir 5 rindas ar
nesingulāriem
elementiem



Ieraksta (record) jēdziens (1)

Ieraksts ir nehomogēna datu struktūra, kurā sakopoti dažāda tipa un dažāda garuma dati. Atsevišķu ieraksta lauku sauc par ieraksta komponenti. Datu tips ir jāuzdod katrai ieraksta komponentei, datu tips var būt jebkurš - skalārs vai strukturēts.

Visās mūsdienu modernajās programmēšanas valodās ir līdzekļi ierakstu definēšanai, piemēram:

```
1) type Name = string[30];  
    computer = record  
        System: Name;  
        Manufacturer: Name;  
        Speed: 500..10000;  
        WordSize: 8 .. 64;  
        Serial_ports: 0 .. 10;  
        Parallel_ports: 0 .. 3  
    end;
```


Ieraksta (record) jēdziens (2)

2) type Complex = record

Re, Im: real

end;

3) type Date = record

Day: 1 .. 31;

Month: 1 .. 12;

Year: 1900 .. 2100

end;

work = record

id: string [20];

start: Date;

stop: Date

end;

Ieraksta (record) jēdziens (3)

Ieraksta lauki var būt izvietoti vairākos hierarhijas līmeņos. Lai organizētu piekļuvi ieraksta laukiem, lieto selektoru – saliktu nosaukumu, kurā ieraksta mainīgo un atsevišķas komponentes nosaukumu atdala ar punktu. Katram hierarhijas līmenim jāparedz punkts un lauka nosaukums.

Piemēram:

var Q: work;

Saliktie nosaukumi:

Q.id – piekļuve darba nosaukumam,

Q.start.Year – piekļuve darba uzsākšanas gadam,

Q.stop.Day – piekļuve darba pabeigšanas dienai.

writeln(Q.id, Q.start.Year, Q.stop.Day);

Ieraksta lauku apstrādei plaši lieto operatoru with, piemēram:

with Q do writeln(id, start.Year, stop.Day);

Ieraksta (record) jēdziens (4)

Ieraksts ir lineāra datu struktūra. Ir 3 pamatoperācijas darbā ar ierakstiem, pie kam trešā operācija arī realizējama, izmantojot pirmās divas:

```
type T1 = real;
```

```
    ....
    TN = string;
```

```
    Rec = record
```

```
        S1: T1;
```

```
        ...
        SN: TN
```

```
    end;
```

```
var   P, Q: Rec;
```

```
      V: T1;
```

```
    ...
1) V := Q.S1;
```

Retrieve}

{ieraksta lauka izguves operācija

```
2) Q.S1 := 3.5 * V - exp (V + 1);
```

Update}

{labošanas operācija

tipam T1 atbilstošā izteiksme

```
3) P := Q;
```

{ieraksta kopēšanas operācija

Ieraksta (record) jēdziens (5)

Vispārējā gadījumā ieraksta struktūra ir šāda:

ieraksta pamatdaļa

ieraksta variantdaļa

Kā pamatdaļa, tā arī variantdaļa nav obligāta, katru no tām var arī izlaist.

Ieraksta tipa apraksta sintakse vispārējā gadījumā:

```
type T = record
```

```
  S1: T1;
```

```
  S2: T2;
```

```
  ...  
  Sn-1: Tn-1;
```

```
  case Sn: Tn of
```

```
    C1: (V1: R1);
```

```
    C2: (V2: R2);
```

```
    ...  
    Cm: (Vm: Rm);
```

```
  end; var Q: T;
```

S_i, V_j – lauka nosaukums
vai

nosaukumu

pamatdaļa

saraksts,

S_n – variantdaļas

selektora

variantdaļas
virsraksts

nosaukums,

T_n – variantdaļas selektora

variantdaļa

tips,

T_i, R_j – lauka tips, jebkurš,

C_j – selektoram atbilstošs

vērtību saraksts,

Ieraksta (record) jēdziens (6)

Piemēram:

```
1) type StateType = (solid, liquid, gas);
```

```
    Substance = record
```

```
        Name: string [20];
```

```
        Number: integer;
```

```
        case state: StateType of  
            solid: (hardness: real);
```

```
            liquid: (boil, freeze: real);
```

```
            gas: ();
```

```
    end
```

```
var S: Substance;
```

pamatdaļa

variantdaļas
virsraksts

variantdaļa

Ieraksta (record) jēdziens (7)

```
2) type Figura = (TA, TR, RI);  
    GeomFig = record  
    case Veids: Figura of  
    TA: (Garums, Platums : real);  
    TR: (Mala1, Mala2, Lenkis: real);  
    RI: (Radiuss: real)  
    end;  
  
function Laukums (Fig: GeomFig): real;  
{Ģeometrisku figūru laukumu aprēķināšana}  
begin  
    with Fig do  
    case Veids of  
    TA: Laukums:= Garums * Platums;  
    TR: Laukums:= 0.5 * Mala1* Mala2 * sin(Lenkis);  
    RI: Laukums:= PI * sqr(Radiuss)  
    end  
  
end;
```

Ieraksta (record) jēdziens (8)

```
3) type CoordMode = (cartesian, polar);
   Coordinate = record
       case Kind: CoordMode of
       cartesian: (X, Y: real);
       polar: (R, Phi: real)
   end;
```

Noteikumi, veidojot ieraksta tipa aprakstu:

- 1) lauku nosaukumiem jābūt atšķirīgiem, pat tad, ja tie sastopami dažādos variantos;
- 2) tukšu variantdaļas lauku uzdod formā **konstante: ()**;
- 3) katrā ierakstā var uzdot tikai vienu variantdaļu, kura savukārt var saturēt citu variantdaļu;
- 4) varianta selektoram jādefinē tikai skalārs ordināls tips;
- 5) variantdaļā nedrīkst definēt file tipa laukus;
- 6) ieraksta variantdaļas lauki programmā apstrādei būs pieejami tikai tad, ja programmā iepriekš variantdaļas selektoram tiek piešķirta vērtība;
- 7) katram selektora vērtības gadījumam jāparedz savs lauks vai to

Ieraksta lauku

Jābūt mehānismam, kas programmā lietotos ieraksta lauku saliktos nosaukumus pārveido reālās atmiņas adresēs. Šim nolūkam kompilators katram programmā lietotam ierakstam piekārto speciāli izveidotu nobīžu sarakstu (offset list). Tajā katrai ieraksta komponentei tiek fiksēti 4 raksturlielumi:

- 1) ieraksta lauka nosaukums;
- 2) ieraksta lauka tips;
- 3) ieraksta lauka tipam atbilstošs garums;
- 4) ieraksta lauka nobīde (baitos) attiecībā pret ieraksta sākumadresi **b** datora pamatatmiņā. Nobīdes vērtības aprēķina kompilātors. Sākumadrese **b** ir zināma tikai tad, kad programmu ielādē atmiņā pilnveidē.

Ieraksta lauku adresēšana un piekļuve (2)

Ieraksta lauka nobīdes vērtība nemainās, mainoties ieraksta sākumadresei **b**.

Ieraksta lauka adrese b_c ir nosakāma šādi:

$$\mathbf{b}_c = \mathbf{b} + \mathbf{O}_c, \text{ kur } O_c - \text{ieraksta lauka nobīde.}$$

Kāda ieraksta lauka nobīdes O_c vērtību aprēķina šādi:

$$Q_c = I_1 + I_2 + \dots + I_{c-1}$$

t.i., sasummējot visu ierakstu lauku garumus ceļā no sākumadreses **b** līdz ieraksta laukam ar kārtas numuru **c**.

Ieraksta lauku adresēšana un piekļuve (3)

Piemēram:

```
type Date = record
```

```
    Day = 1..31;
```

```
    Month = 1..12;
```

```
    Year = 1900..2100
```

```
end;
```

```
work = record
```

```
    id: string [20];
```

```
    start: Date;
```

```
    stop: Date;
```

```
end;
```

```
var Q: work;
```

Ieraksta lauku adresēšana un piekļuve (4)

Nosaukums	Tips	Gar.	Nob		id	0
				.		21
				star	Day	23
Q.id	string[20]	21	<u>0</u>	t	Month	23
Q.start	Date	6	<u>21</u>		Year	23
Q.start.Day	integer	2	21			27
Q.start.Mont	integer	2	23	stop	Day	29
h	integer	2	25		Month	31
Q.start.Year	Date	6	<u>27</u>		Year	
Q.stop	integer	2	27			
Q.stop.Day	integer	2	29			
Q.stop.Mont	integer	2	31			
h						
Q.stop.Year						

Masīvu ieraksti (records of arrays)

Masīvi var būt ieraksta lauki dažādos hierarhijas līmeņos. Masīvu ieraksti pieder pie saliktajām struktūrām.

Piemērā dots ieraksts, kuru izmanto, lai fiksētu informāciju par noteiktu notikumu, kas var notikt 20 dažādās vietās un 15 dažādos datumos katrā vietā:

```

type RA = record
    event: string [30];
    place: array [1 .. 20] of
        record
            placeName: string [20];
            date: array [1..15] of record
                dy: 1 .. 31;
                mo: 1 .. 12;
                yr: 1900 .. 2100
            end
        end
    end;
var Q: RA;
```

$i = 1, 2, \dots, 20,$
 $j = 1, 2, \dots, 15.$

Saliktie nosaukumi:

Q.place[i].date[j].mo –
 uzdod notikumu
 Q.event

i-tās vietas ajā
 datumā.

Q.place[i].placeName –
 uzdod i-tā notikuma
 vietu.

Q.event – uzdod
 notikuma
 nosaukumu.

Ierakstu masīvi jeb tabulas (arrays of records or tables)

Viendimensijas ierakstu masīvi ir plaši lietota datu struktūra, ko sauc arī par tabulu, un kuru izmanto, lai sakopotu informāciju par objektiem vai personām, kas pieder pie vienas grupas. Ierakstu masīvs pieskaitāms pie saliktajām datu struktūrām.

Piemērā dota tabula, kurā sakopota informācija par studentiem grupā:

```
const N = 25;
type text1 = string [20];
    text2 = string [30];
    text3 = string [50];
    Studenti = array [1 .. N] of record
```

```
    Nr: 1..N;
```

```
    Vards: text1;
```

```
    Uzv: text2;
```

```
    st_apl: string [11];
```

```
    adrese: text3;
```

```
    telefons: string[12];
```

```
end;
```

```
var S: Studenti;
```

Lai organizētu piekļuvi tabulā grupas i-tā studenta ieraksta laukiem, jālieto saliktie mainīgie ar indeksiem:

S[i].Nr, S[i].Vards, ... ,

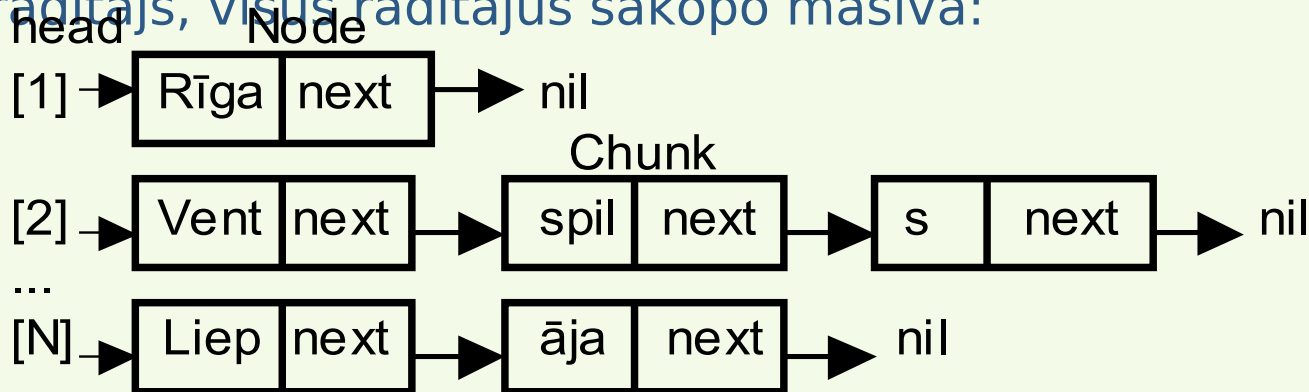
S[i].telefons, i = 1, 2, 3, ...,

N.

Nrsu pilnveidošana

Rādītāju masīvi un to lietojums (1)

- 1) rādītāji uz masīva elementiem, kuriem var būt mainīgs garums.
 Masīva elementi tiek dinamiski izveidoti, uz katru elementu norāda
 savs rādītājs, visus rādītājus sakopo masīvā:



```

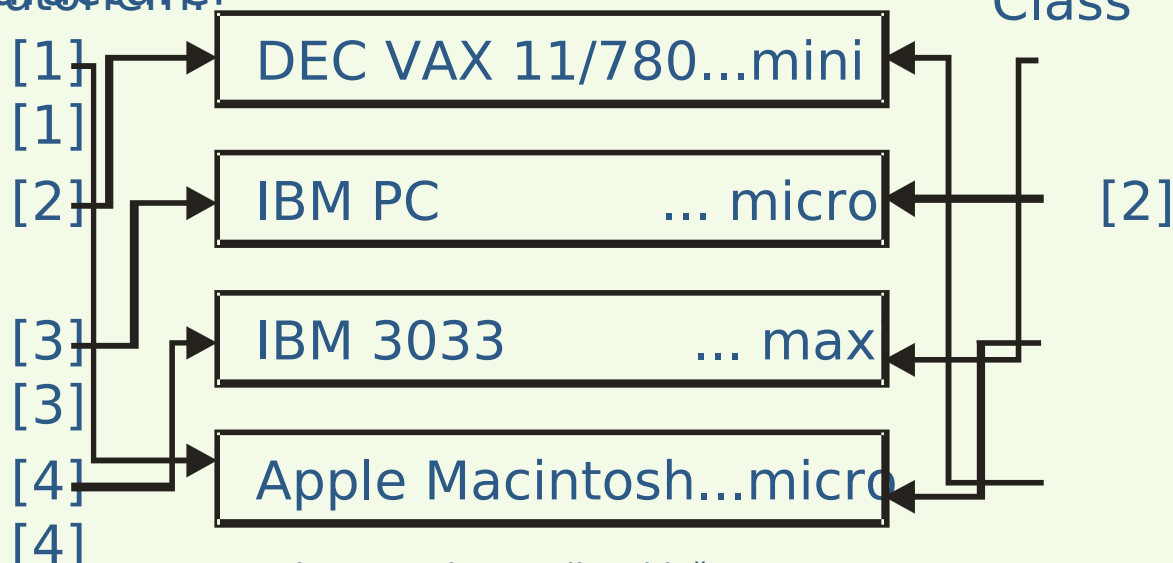
const N = 100;           {elementu skaits}
masīvā}
ChunkSize = 4;           {fragmenta garums, ko uzdod
lietotājs}
type NodePointer = ^Node; {rādītāja datu
tips}
Node = record             {masīva
elements}
  ChunkArray[1..ChunkSize] of char;
  110
  {fragmenta

```

Rādītāju masīvi un to lietojums (2)

2) saraksta vai vektora elementu sasaiste ar rādītājiem, uzdodot noteiktu sasaistes likumu, piemēram, nosaukumu sasaiste alfabētiskā secībā. Pēc šāda principa veido vairākkārtīgi saistītus sarakstus. Katram sasaistes kontūram paredz savu rādītāju vektoru. Šādus vektorus sauc par datu indeksvektoriem. Ar to palīdzību ērti nodrošināt piekļuvi datiem un organizēt vajadzīgo datu sameklēšanu.

Apskatīsim tabulu, kurā ir vairākas kolonnas un kurā sakopota informācija par datoriem.



Rādītāju masīvi un to lietojums (3)

```

const N = 4;                                {elementu skaits tabulā - uzdod
lietotājs}
type Computers = array [1 .. N] of record    {tabulas
datu tips}
    Company: string [20];
    System: string [10];
    ...
    Systype: string [5]
end;
Table = ^Computers;                        {rādītāju
datu tips}
var Q: Table;
    Manufacturer = array [1 .. N] of Table;  {ražotājfirmas rādītāju
masīvs}
    Class = array [1 .. N] of Table;         {datorklases rādītāju
masīvs}
    {Rādītāju iestatīšana uz kādu noteiktu tabulas elementu:}
    Manufacturer [1]:= addr (Q ^ [4].Company);
    Manufacturer [2]:= addr (Q ^ [1].Company);
    Manufacturer [3]:= addr (Q ^ [2].Company);
    Manufacturer [4]:= addr (Q ^ [3].Company);
    Class [1]:= addr (Q ^ [3].Systype);
    Class [2]:= addr (Q ^ [3].Systype);
    Class [3]:= addr (Q ^ [3].Systype);
    Class [4]:= addr (Q ^ [3].Systype);

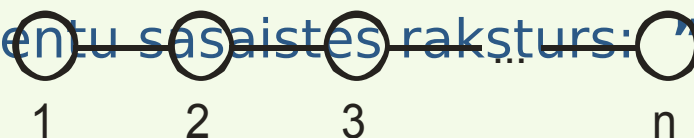
```


Saraksta (list) jēdziens (1)

Saraksts ir lineāra datu struktūra, kurā, ja saraksts nav tukšs:

- 1) ir unikāls elements, ko sauc par pirmo;
- 2) ir unikāls elements, ko sauc par pēdējo;
- 3) visiem saraksta elementiem, izņemot pirmo un pēdējo, ir unikāls priekštecis un pēctecis. Pirmajam elementam ir tikai pēctecis, bet pēdējam elementam ir tikai priekštecis;
- 4) saraksts var būt tukšs.

Saraksta elementu saistītais raksturs: **"viens ar vienu":**



Visiem saraksta elementiem ir tips *StdElement*, kas paredz, ka saraksta datu laukam *data* tiek pievienots atslēgas lauks *key*.

Saraksta elementu atslēgām *key* jābūt atšķirīgām, jo tās viennozīmīgi identificē elementu sarakstā.

Viens no elementiem sarakstā vienmēr ir tekošais (*current*).

Saraksta (list) jēdziens (2)

Saraksta attēlojums:

- 1) vektoriālajā formā, izmantojot saraksta pozicionēšanu vai hešēšanu (jaukšanu);
- 2) saistītajā formā, visus elementus sarakstā saistot ar rādītājiem.

Nesakārtotos sarakstos elementu izvietojums var būt patvalīgs.

Sakārtotos sarakstos elementu izvietojums atbilst noteiktam kārtošanas kritērijam.

Sakārtotie saraksti:

- 1) hronoloģiski sakārtotie saraksti;
- 2) pēc lietojuma biežuma sakārtotie saraksti. Pie tiem pieskaitāmi arī pašorganizētie saraksti;
- 3) sašķirotie saraksti.

Tekošo elementu sarakstā iestata:

- 1) meklēšanas operācijas **Findxxx**, ja saraksts nav tukšs, 114
niskās studiju programmas "Datorsistēmas" kursu pilnveidošana
2) jauna elementa pievienošanas operācija *Insert*;

Saraksta (list) jēdziens (3)

Neatkarīgi no saraksta attēlojuma modeļa, darbā ar sarakstiem paredzētas šādas operācijas:

1) apkalpošanas (servisa) operācijas:

Create	Empty
Terminate	Full
Size	First
CurPos	Last

2) meklēšanas operācijas:

FindFirst	FindLast
FindNext	FindPrior
FindKey	Findith

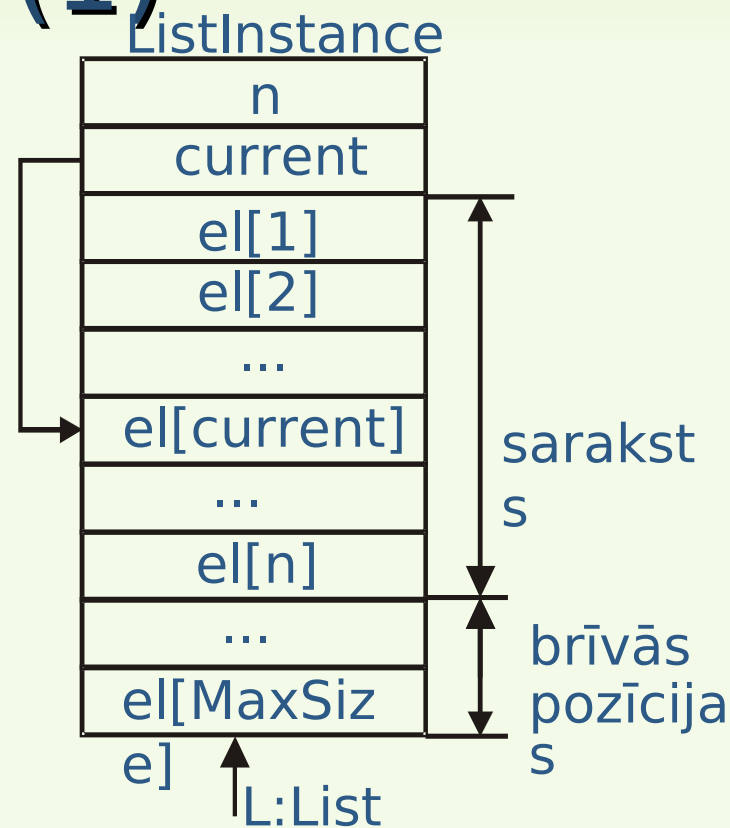
3) pamatoperācijas:

Insert	Retrieve
InsertAfter	Update
InsertBefore	Delete

Vektoriālā formā attēlotais saraksts (1)

Lai organizētu piekļuvi saraksta laukiem, lietojamas šādas norādes:

$L.n$ – elementu skaita laukam,
 $L.current$ – tekošā elementa kārtas numura laukam,
 $L.el[i]$ – saraksta i -jam elementam, $i=1,2,\dots,n$,
 $L.el[i].data$ – saraksta i -tā elementa datu laukam,
 $L.el[i].key$ – saraksta i -tā elementa atslēgas laukam



Saraksta elements $el[i]$
 $i = 1, 2, \dots, n$

dat	: DataType
key	: KeyType

Vektoriālā formā attēlotais saraksts (2)

```

const MaxSize = 100;           {maksimālais elementu
skaits}
type Position = 1 .. MaxSize;   {elementa
pozīcijas tips}
Count = 0 .. MaxSize;          {elementu
skaits tips}
[
  Edit = 1 .. 3;                {labošanas variantu
tips}
  DataType = string [20];       {datu lauka
tips}
  KeyType = integer;            {atslēgas
lauka tips}
  StdElement = record           {saraksta
elementa tips}
    data: DataType;
    key: KeyType
  end;
  ListInstance = record

```

Vektoriālā formā attēlotais saraksts (3)

```
procedure Create (var L: List; var created: boolean);  
{Izveido jaunu tukšu sarakstu L^}  
begin  
    new(L);  
    L^.n:= 0;  
    L^.current:= 0;  
    created:= true  
end;  
  
procedure Terminate (var L: List; var created: boolean);  
{Likvidē sarakstu L^}  
begin  
    if created then  
        begin  
            dispose (L);  
            created:= false  
        end  
    end;  
end;
```

Vektoriālā formā attēlotais saraksts (4)

```
function CurPos (L: List): Count;  
{Nosaka tekošā elementa pozīcijas numuru sarakstā L^}  
begin  
    CurPos:= L^.current  
end;  
  
function Size (L: List): Count;  
{Nosaka elementu skaitu sarakstā L^}  
begin  
    Size:= L^.n  
end;  
  
function Empty (L: List): boolean;  
{Pārbauda, vai saraksts L^ ir tukšs}  
begin  
    Empty:= L^.n = 0  
end;
```

Vektoriālā formā attēlotais saraksts (5)

```
function Full (L: List): boolean;  
{Pārbauda, vai saraksts L^ ir pilns}  
begin
```

```
    Full:= L^.n = MaxSize
```

```
end;
```

```
function First (L:List): boolean;  
{Pārbauda, vai pirmais elements ir tekošais sarakstā L^}  
begin
```

```
    First:= L^.current = 1
```

```
end;
```

```
function Last (L: List): boolean;  
{Pārbauda, vai pēdējais elements ir tekošais sarakstā L^}  
begin
```

```
    Last:= L^.current = L^.n
```

```
end;
```


Vektoriālā formā attēlotais saraksts (6)

```
procedure FindNext (var L: List);  
  {Sarakstā L^ meklē tekošā elementa pēcteci, kas kļūst par  
tekošo  
  elementu}  
begin  
  if CurPos(L) <> Size(L) then L^.current:= L^.current + 1  
end;  
  
procedure FindPrior (var L: List);  
  {Sarakstā L^ meklē tekošā elementa priekštecī, kas kļūst par  
tekošo  
  elementu}  
begin  
  if CurPos(L) > 1 then L^.current:= L^.current - 1  
end;
```

Vektoriālā formā attēlotais saraksts (7)

```
procedure FindFirst (var L: List);  
  {Sarakstā L^ meklē pirmo elementu, kas kļūst par tekošo  
elementu}  
begin  
  if CurPos(L) > 1 then L^.current:= 1  
end;
```

```
procedure FindLast (var L:List);  
  {Sarakstā L^ meklē pēdējo elementu, kas kļūst par tekošo  
elementu}  
begin  
  if CurPos(L) <> Size(L) then L^.current:= L^.n  
end;
```

Vektoriālā formā attēlotais saraksts (8)

```

procedure FindKey1 (var L: List; tkey: KeyType; var found:
boolean);
{Sarakstā L^ meklē elementu, kura atslēgas lauka vērtība ir
tkey. Ja meklēšana ir sekmīga, sameklētais elements kļūst par tekošo
elementu}
var k: Position;
    done: boolean;
begin
    found:= false;
    if not Empty (L) then with L^ do
        begin
            k:= 1; done:= false;
            while (not done) and (not found) do {meklē
elementu}
                if tkey = el [k].key then
                    begin {sekmīga
meklēšana}
                        current:= k; found:= true
                    end
                k:= k + 1;
            end
        end
    end
end

```

Vektoriālā formā attēlotais saraksts (9)

```
procedure FindKey2 (var L: List; tkey: KeyType; var found:
boolean);
  {Lineārā meklēšana, izmantojot robežmarķieri}
  var k: Position;
  begin
    found:= false;
    if not Empty (L) then with L^ do
      begin
        found:= true;
        el [n + 1].key:= tkey;                                {izvieto
robežmarķieri}
        k:= 1;
        while el [k].key <> tkey do k:= k +1                  {meklē
elementu}
        if k = n + 1 then found:= false
        else current:= k
      end
    end
```

Vektoriālā formā attēlotais saraksts (10)

```
procedure Findith (var L: List; i: Position);  
{Sarakstā L^ meklē elementu ar kārtas numuru i. Ja meklēšana  
ir  
sekmīga, sameklētais elements kļūst par tekošo elementu}  
begin  
    if (not Empty(L)) and (i <= Size (L)) then L^.current:= i  
end;
```

Vektoriālā formā attēlotais saraksts (11)

```
procedure InsertAfter (var L:List; e: StdElement);  
  {Sarakstā L^ aiz tekošā elementa pievieno jaunu elementu e,  
  kas kļūst  
  par tekošo elementu}  
  var k: Position;  
  begin  
    if not Full (L) then with L^ do  
      begin  
        if not Last (L) then  
          for k:= n downto current + 1 do  
            el [k+1]:= el [k];           {atbrīvo vietu  
elementam}  
          current:= current + 1;  
          el [current]:= e;               {izvieto  
elementu}  
          n:= n + 1  
        end  
      end;  
  end;
```

Vektoriālā formā attēlotais saraksts (12)

```
procedure InsertBefore (var L:List; e: StdElement);  
  {Sarakstā  $L^{\wedge}$  pirms tekošā elementā pievieno jaunu elementu  
e, kas  
  kļūst par tekošo elementu}  
begin  
  if not Empty(L) then  $L^{\wedge}.current := L^{\wedge}.current - 1$ ;  
  InsertAfter (L, e)  
end;  
  
procedure Retrieve (L: List; var e: StdElement);  
  {Tekošā elementa izguve sarakstā  $L^{\wedge}$ }  
begin  
  if not Empty (L) then  $e := L^{\wedge}.el [L^{\wedge}.current]$   
end;
```

Vektoriālā formā attēlotais saraksts (13)

```
procedure Delete (var L: List)
{Sarakstā L ^ dzēš tekošo elementu}
var k: Position;
begin
    if not Empty (L) then with L ^ do
        begin
            for k:= current +1 to n do                {pārvieto
elementus}
                el [k - 1]:= el [k];
            if n = 1 then current:= 0
                else if current = n then current:= n -1;
            n:= n -1
        end
    end;
end;
```


Vektoriālā formā attēlotais saraksts (14)

```
procedure Update (var L: List; e: StdElement; k: Edit);  
  {Sarakstā  $L^{\wedge}$  labo tekošo elementu atbilstoši labošanas  
variantam k}  
begin  
  if not Empty (L) then with  $L^{\wedge}$  do  
    case k of  
      1: el [current].data:= e.data;  
      2: el [current].key:= e.key;  
      3: el [current]:= e  
    end  
  end;  
end;
```

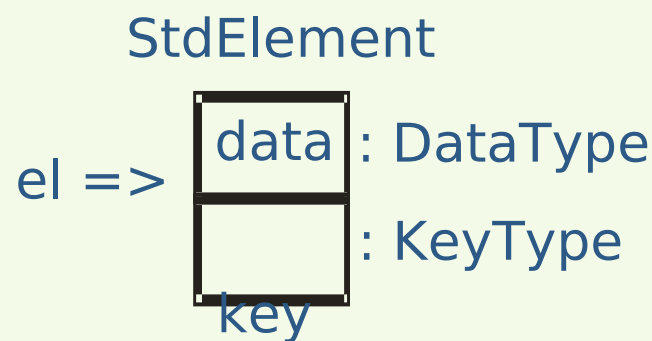
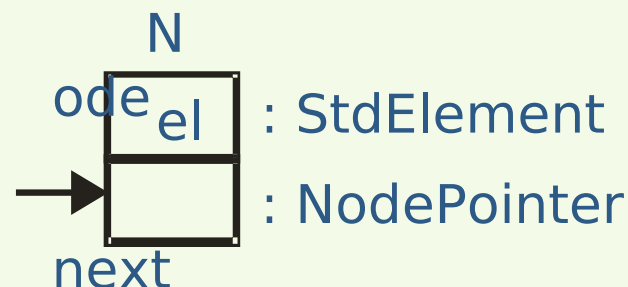
Vienkāršsaistītais saraksts (1)

Veidojot vienkāršsaistītu sarakstu, katram saraksta elementam papildus pievieno rādītāja lauku *next*, kurā glabājas nākamā elementa adrese:

```
type Node = record
    el: StdElement;
    next: NodePointer
end;
NodePointer = ^ Node;
```

Saraksta elementa uzbūve:

```
type StdElement = record
    data: DataType;
    key: KeyType
end;
```



Vienkāršsaistītais saraksts (2)

Lai organizētu piekļuvi saistītam sarakstam, lieto šādus rādītājus:

- 1) **L** – kas norāda uz saistītā saraksta vadības struktūru;
- 2) **head** – kas vienmēr norāda uz pirmo elementu sarakstā, t.i.,

glabā

saraksta sākumadresī;

- 3) **current** – kas norāda uz tekošo elementu sarakstā. Strādājot

ar

sarakstu, par tekošo var kļūt jebkurš saraksta elements.

Saraksta

apstrādes operācijas vienmēr tiek izpildītas attiecībā pret

tekošo

elementu;

- 4) **tail** – kas norāda uz pēdējo elementu sarakstā. Rādītājs tail

nav

obligāts, bez tā var iztikt, tomēr tā lietojums piekļuvi

sarakstam

padara ērtāku. Ja rādītāju tail nelieto, tad par saistīta saraksta

beigu pazīmi izmanto pēdējā elementa rādītāja lauka next

vērtību

nil.

Katram elementam sarakstā atbilst noteikts kārtas numurs, kas

131

glabājas laukā **current**. Informācija par elementu skaitu sarakstā

Vienkāršsaistītais saraksts (3)

Pirmā un pēdējā elementa sameklēšana sarakstā:

```
current:= head;   icurrent:= 1;
```

```
current:= tail;    icurrent:= n;
```

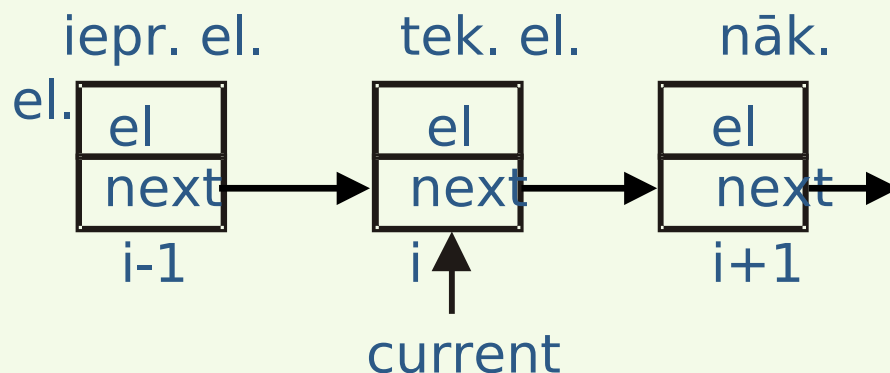
Saraksta pēdējā elementa sameklēšana, ja rādītāju tail nelieto:

```
while current^.next <> nil do
```

```
    current:= current^.next;
```

Par tekošo kļūst nākamais elements:

```
current:= current^.next;   icurrent:= icurrent+1;
```



Vienkāršsaistītais saraksts (4)

Saraksta vadības struktūras lauku identifikācija un piekļuve:

$L^{\wedge}.current,$ $L^{\wedge}.icurrent,$ $L^{\wedge}.n$
 $L^{\wedge}.head,$ $L^{\wedge}.tail$ vai

e	→
h	

 ni
 ext

Saraksta tekošā elementa lauku identifikācija un piekļuve:
 $current^{\wedge}.el$ – elementa informatīvā lauka piekļuve,

$current^{\wedge}.el.data$ – datu lauka piekļuve,

$current^{\wedge}.el.key$ – atslēgas lauka piekļuve,

$current^{\wedge}.next$ – rādītāja lauka piekļuve,

$current := current^{\wedge}.next$ – rādītāja pārcelšana uz nākamo elementu,

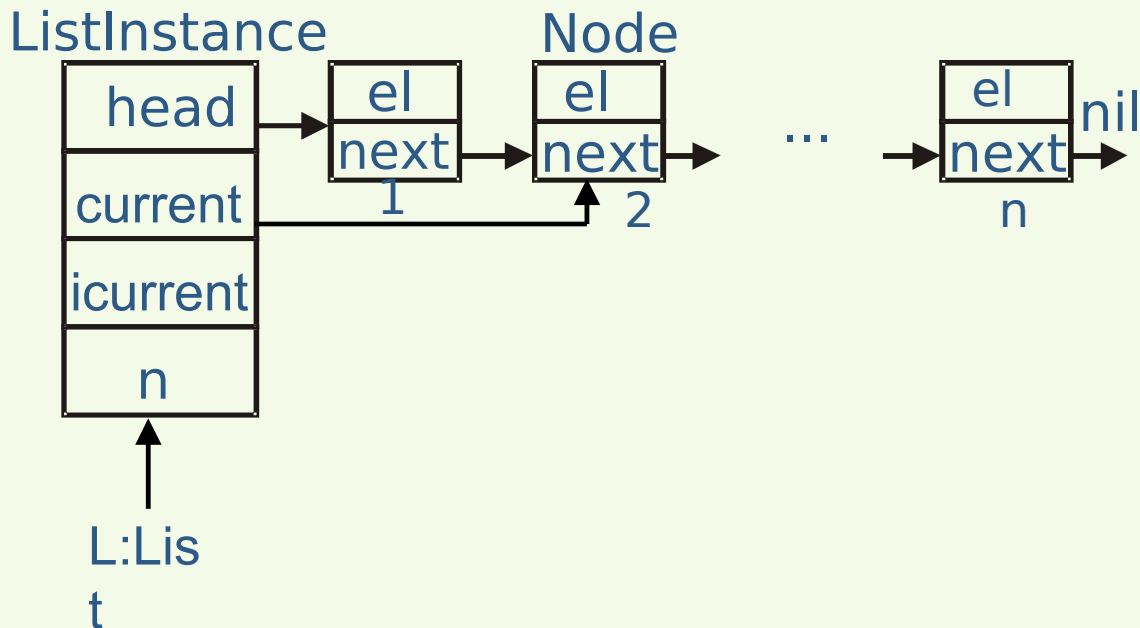
$current := current^{\wedge}.next^{\wedge}.next$ – rādītāja pārcelšana uz aiznākamo

elementu.

Vienkāršsaistītais saraksts bez beigu rādītāja (1)

```

const MaxSize = 100;
type Count = 0 .. MaxSize; ListInstance
DataType = string;
KeyType = integer;
Edit = 1 .. 3;
StdElement = record
    data: DataType;
    key: KeyType
end;
Node = record
    el: StdElement;
    next: NodePointer
end;
NodePointer = ^ Node; {rādītāja tips}
List = ^ ListInstance;
ListInstance = record
    head, current: NodePointer;
    icurrent, n: Count
end;
  
```



Vienkāršsaistītais saraksts bez beigu rādītāja (2)

```
procedure Create (var L: List; var created: boolean);  
{Izveido jaunu tukšu sarakstu L^}  
begin  
    new (L);  
    with L^ do  
        begin  
            head:= nil;  
            current:= nil;  
            icurrent:= 0;  
            n:= 0  
        end;  
    created:= true  
end;
```

Vienkāršsaistītais saraksts bez beigu rādītāja (3)

```
function Size (L: List): Count;  
{Nosaka elementu skaitu sarakstā L^}  
begin  
    Size:= L^.n  
end;  
function CurPos (L: List): Count;  
{Nosaka tekošā elementa kārtas numuru sarakstā L^}  
begin  
    CurPos:= L^.icurrent  
end;  
function Empty (L: List): boolean;  
{Pārbauda, vai saraksts L^ ir tukšs}  
begin  
    Empty:= L^.n = 0  
end;
```


Vienkāršsaistītais saraksts bez beigu rādītāja (4)

```
function Full (L: List): boolean;  
{Pārbauda, vai saraksts L^ ir pilns}  
begin  
    Full:= L^.n = MaxSize  
end;  
  
function First (L: List): boolean;  
{Pārbauda, vai pirmais elements ir tekošais sarakstā L^}  
begin  
    First:= L^.icurrent = 1  
end;  
  
function Last (L: List): boolean;  
{Pārbauda, vai pēdējais elements ir tekošais sarakstā L^}  
begin  
    Last:= L^.icurrent = L^.n  
    {Last:=  
L^.current^.next = nil}  
end;
```

Vienkāršsaistītais saraksts bez beigu rādītāja (5)

```

procedure FindFirst (var L: List);
{Sarakstā L^ meklē pirmo elementu, kas kļūst par tekošo
elementu}
begin
    if CurPos(L) > 1 then with L^ do
        begin
            curent:= head;  icurrent:= 1
        end
    end;
procedure FindLast (var L: List);
{Sarakstā L^ meklē pēdējo elementu, kas kļūst par tekošo
elementu}
begin
    if CurPos (L) <> Size (L) then with L^ do
        begin
            while current^.next <> nil do           {while not
Last(L)}
                current:= current^.next;
        end;
{FindNext(L)}
    icurrent:= n

```

Vienkāršsaistītais saraksts bez beigu rādītāja (6)

```
procedure FindNext (var L: List);  
  {Sarakstā L^ meklē tekošā elementa pēcteci, kas kļūst par  
tekošo  
  elementu}  
begin  
  if CurPos(L) <> Size(L) then with L^ do  
    begin  
      current:= current^.next  
      icurrent:= icurrent + 1  
    end  
end;  
end;
```

Vienkāršsaistītais saraksts bez beigu rādītāja (7)

```
procedure FindPrior(var L: List);
{Sarakstā L^ meklē tekošā elementa priekštecī, kas kļūst par
tekošo elementu}
var p, q: NodePointer;
begin
    if CurPos(L) > 1 then with L^ do
        begin
            p:= head;  q:= nil;
            while p <> current do {meklē
priekštecī}
                begin
                    q:= p;  p:= p^.next
                end;
            current:= q;
            icurrent:= icurrent -1
        end
    end;
end;
```

Vienkāršsaistītais saraksts bez beigu rādītāja (8)

```

procedure FindKey (var L: List; tkey: KeyType; var found:
boolean);
{Sarakstā L^ meklē elementu, kura atslēgas lauka vērtība ir
tkey.
meklēšana ir sekmīga, sameklētais elements kļūst par tekšo
elementu}
var p: NodePointer; k: Count;
begin
    found:= false;
    if not Empty(L) then with L^ do
        begin
            p:= head; k:= 1;
            while (p^.next <> nil) and (p^.el.key <> tkey)
do
                begin
                    {meklē
                    elementu}
                    p:= p^.next; k:= k + 1
                end;
            if p^.el.key = tkey then
                begin
                    {sekmīga
                    meklēšana}
                    current:= p; icurrent:= k; found:= true;

```

Vienkāršsaistītais saraksts bez beigu rādītāja (9)

```

procedure FindIth (var L: List; i: Count);
{Sarakstā L^ meklē elementu ar kārtas numuru i. Ja
meklēšana ir
sekmīga, sameklētais elements kļūst par tekošo elementu}
begin
    if (not Empty(L)) and (i <= Size(L)) then
        with L^ do
            begin
                current:= head;    icurrent:= 1;
{FindFirst(L);}
                while i <> icurrent do
                    begin
                        current:=current^.next;
{FindNext(L);}
                        icurrent:= icurent + 1
                    end
            end
        end
    end
end

```

Vienkāršsaistītais saraksts bez beigu rādītāja (10)

```
procedure Retrieve (L: List; var e: StdElement);
```

```
{Tekošā elementa izguve sarakstā L^}
```

```
begin
```

```
    if not Empty(L) then with L^ do e:= current^.el
```

```
end;
```

```
procedure Update(var L: List; k: Edit; e: StdElement);
```

```
{Sarakstā L^ labo tekošo elementu atbilstoši labošanas  
variantam k}
```

```
begin
```

```
    if not Empty(L) then with L^ do
```

```
        case k of
```

```
            1: current^.el.data:= e.data;
```

```
            2: current^.el.key:= e.key;
```

```
            3: current^.el:= e
```

```
        end
```

```
end;
```

Vienkāršsaistītais saraksts bez beigu rādītāja (11)

```

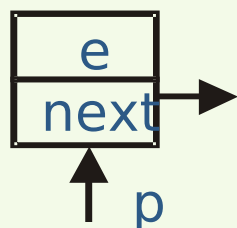
procedure Insert (var L: List; e: StdElement);
{Sarakstā L^ aiz tekošā elementa pievieno jaunu elementu e, kas
klūst par
tekošo elementu}
var p: NodePointer;
begin
    if not Full(L) then with L^ do
        begin
            new(p); p^.el:= e;
            if Empty(L) then {saraksts ir
tukšs}
                begin
                    head:= p; p^.next:= nil
                end
            else {saraksts nav
tukšs}
                begin {izkārto 2
saites}
                    p^.next:= current^.next;
                    current^.next:= p
                end
            end;
            current:= p; icurrent:= icurrent + 1; n:= n + 1
        end
    end;
end;

```


Vienkāršsaistītais saraksts bez beigu rādītāja (12)

Elementu dinamiski izveido:

Node

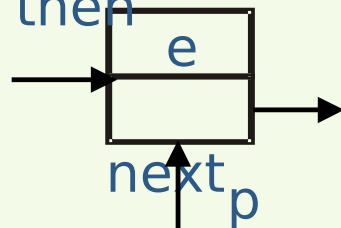


`new(p);`

`p^.el := e;`

if Empty(L)
then

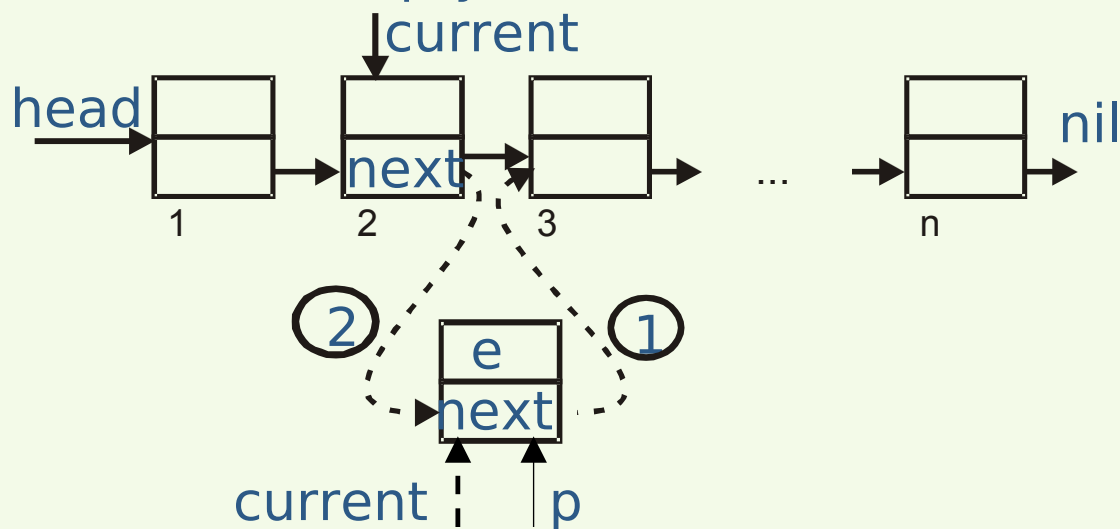
head



1 `head := p;`

`p^.next := nil;`

if not Empty(L) then



1) `p^.next := current^.next;`

2) `current^.next := p;`

`current := p;`

Vienkāršsaistītais saraksts bez beigu rādītāja (13)

```

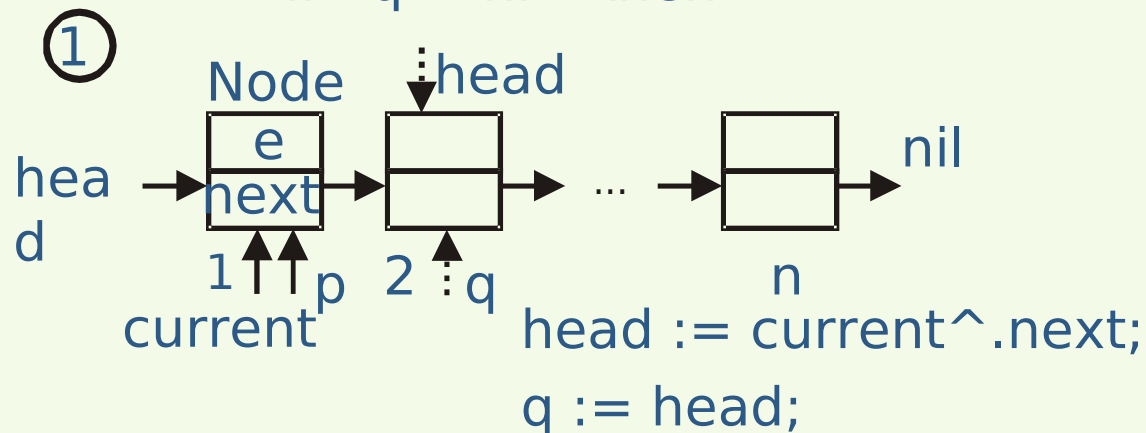
procedure Delete (var L: List);
{Sarakstā L^ dzēš tekošo elementu un vairumā gadījumu
iepriekšējais elements
klūst par tekošo elementu}
var p, q: NodePointer;
begin
  if not Empty(L) then with L^ do
    begin
      p:= head;  q:= nil;
      while p <> current do
        begin
          q:= p;  p:= p^.next
        end;
      if q = nil then
        begin
          head:= current^.next;  q:= head;
          if n = 1 then icurrent:= 0
        end
      else
        begin
          q^.next:= current^.next;  icurrent:= icurrent - 1;
          dispose(current);  current:= q;  n:= n - 1
        end
      end
    end
  end
end;

```

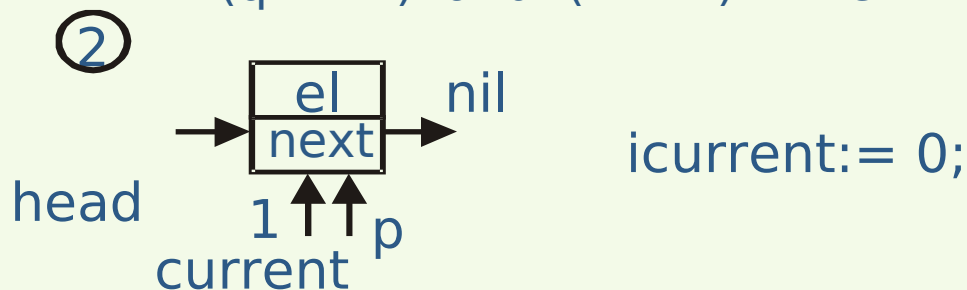
priekštecī}
 elementu}
 pirmais}
 {meklē
 {dzēš 1.
 {dzēš elementu, kas nav

Vienkāršsaistītais saraksts bez beigu rādītāja (14)

if $q = \text{nil}$ then

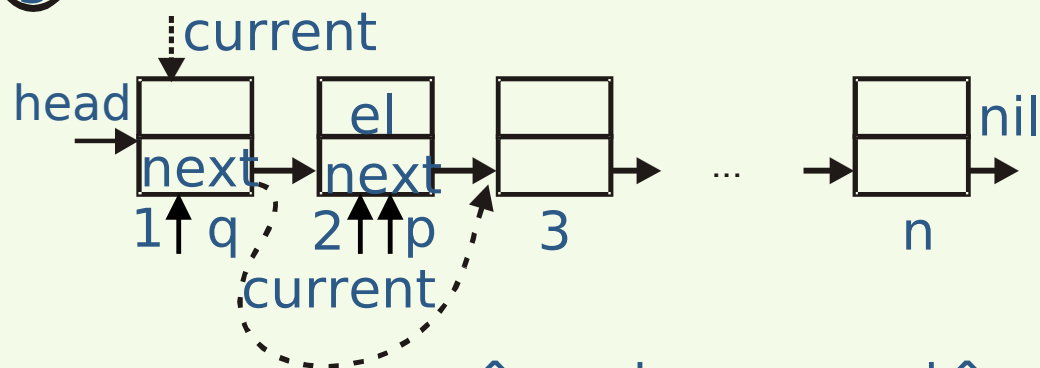


if $(q = \text{nil})$ and $(n = 1)$ then



Vienkāršsaistītais saraksts bez beigu rādītāja (15)

③ if $(n > 1)$ and $(\text{current} \neq \text{head})$ then

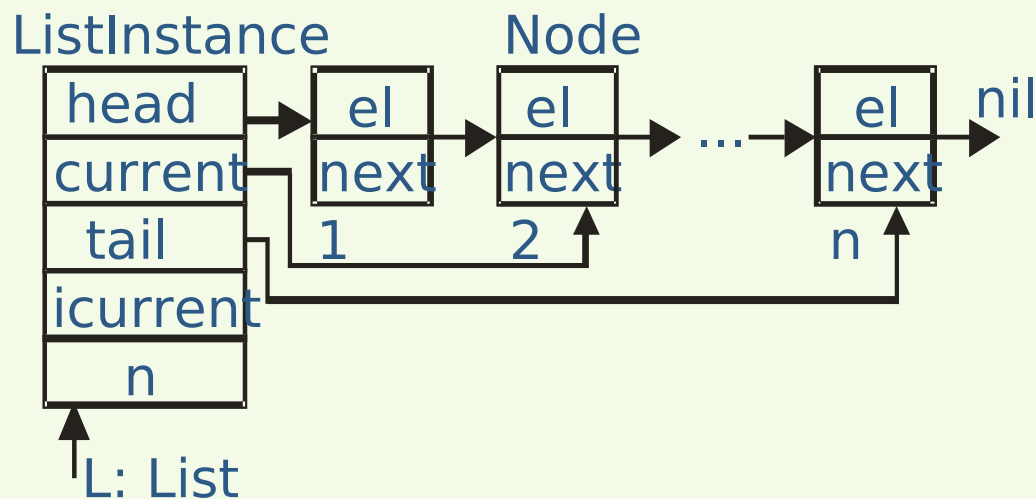


$q.^{\text{next}} := \text{current}.^{\text{next}};$

$\text{current} := q;$

$\text{icurrent} := \text{icurrent} - 1;$

Vienkāršsaistītais saraksts ar beigu rādītāju (1)



Salīdzinājumā ar vienkāršsaistītu sarakstu bez beigu rādītāja, atšķirības būs šādās operācijās:

Create → tail := nil;
Last → Last := L[^].current = L[^].tail;
FindLast → L[^].current := L[^].tail;

Insert
Delete

```

...
List =
    ^ListInstance;
ListInstance =
    record
        head:
            NodePointer;
        current:
            NodePointer;
        tail:
            NodePointer;
        icurrent: Count;
        n: Count;
    end;

```

Vienkāršsaistītais saraksts ar beigu rādītāju (2)

```
procedure Create (var L: List; var created: boolean);  
{Izveido jaunu tukšu sarakstu L^}  
begin  
    new(L);  
    with L^ do  
        begin  
            current:= nil;  
            head:= nil; tail:= nil;  
            icurrent:= 0;  
            n:= 0  
        end;  
    created:= true  
end;
```

Vienkāršsaistītais saraksts ar beigu rādītāju (3)

```
procedure FindLast (var L: List);  
{Sarakstā L^ meklē pēdējo elementu, kas kļūst par tekošo  
elementu}  
begin  
  if Size(L) <> CurPos (L) then with L^ do  
    begin  
      current:= tail;  
      icurrent:= n  
    end  
  end;  
  
function Last(L: List): boolean;  
{Pārbauda, vai pēdējais elements ir tekošais sarakstā L^}  
begin  
  Last::= L^.current = L^.tail  
end;
```

Vienkāršsaistītais saraksts ar beigu rādītāju (4)

```

procedure Insert (var L: List; e: StdElement);
{Sarakstā L^ aiz tekošā elementa pievieno jaunu elementu e, kas
klūst tekošo elementu}
var p: NodePointer;
begin
  if not Full(L) then with L^ do
    begin
      new (p);  p^.el:= e;
      if Empty(L) then
        tukšs}
        begin
          head:= p;  tail:= p;  p^.next:= nil
        end
      else
        tukšs}
        {saraksta nav
        {izkārtoto divas
        saites}
        begin
          p^.next:= current^.next;
          current^.next:= p;
          if Last(L) then tail:= p
        end;
        current:= p;  current:= current + 1;  n:= n + 1
    end
  end

```


Vienkāršsaistītais saraksts ar beigu rādītāju (5)

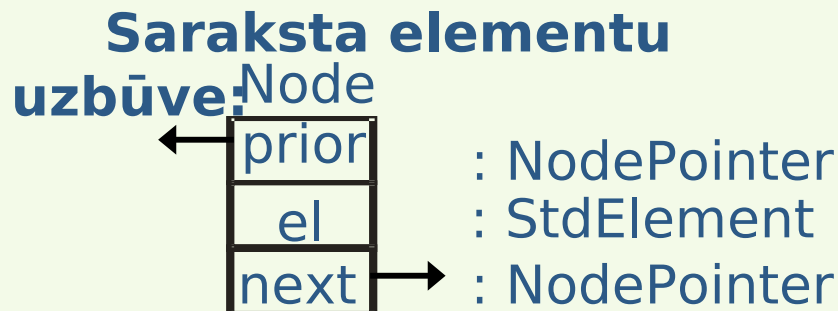
```

procedure Delete (var L: List);
{Sarakstā L^ dzēš tekošo elementu un vairumā gadījumu iepriekšējais elements
klūst tekošo elementu}
var p, q: NodePointer;
begin
  if not Empty(L) then with L^ do
    begin
      p:= head;  q:= nil;
      while p <> current do
        begin
          q:= p;  p:= p^.next
        end;
      if q = nil then
        begin
          head:= current^.next;  q:= head;
          if n = 1 then begin icurrent:= 0;  tail:= nil  end
        end
      else
        begin
          q^.next:= current^.next;
          icurrent:= icurrent - 1
        end;
      if p = current then tail:= q;
      dispose (p);
    end;
  end;
end;

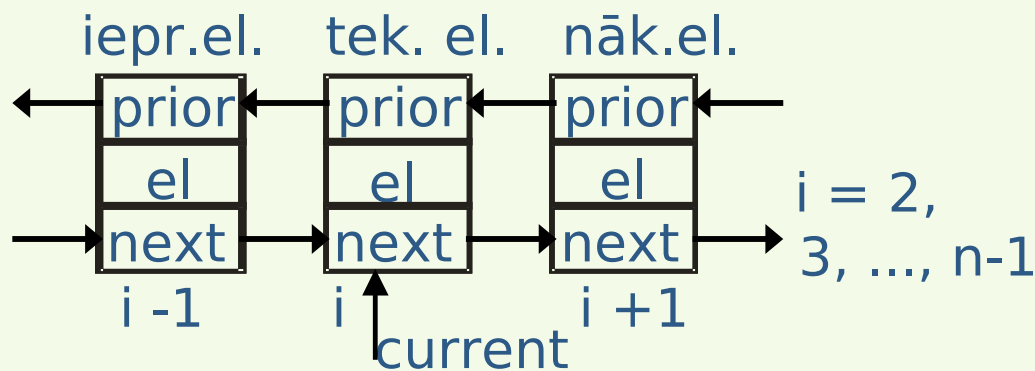
```

{meklē
 priekštecī}
 {jādzēš 1.
 elements}
 {jādzēš elements, kas nav
 pirmais}
 {pārkārto
 saiti}

Divkāršsaistītais saraksts (1)



Saraksta elementu sasaiste:



Divkāršsaistītais saraksts (2)

Rādītāji un darbības ar rādītājiem:

current - tekošā elementa rādītājs.

Rādītāju pārceļ uz saraksta nākamo elementu:

current:= current^.next; icurrent:= icurrent+1;

Rādītāju pārceļ uz saraksta nākamo elementu:

current:= current^.prior; icurrent:= icurrent-1;

Norādes:

current^.next - norāde uz nākamo elementu,

prior,
current^.next^.prior - norāde uz tekošā elementa lauku

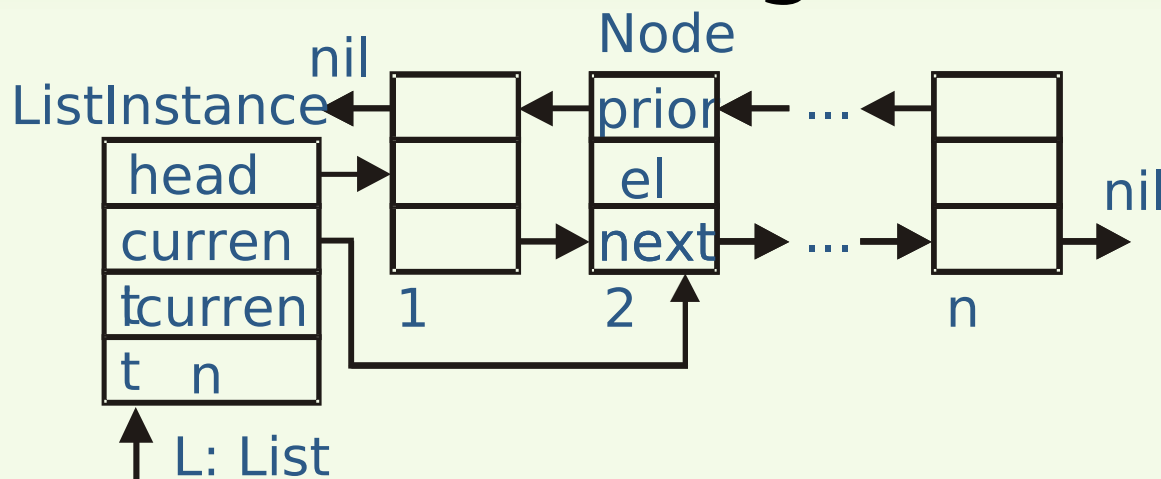
next,
current^.prior^.next - norāde uz tekošā elementa lauku

current^.prior - norāde uz iepriekšējo elementu,

current^.next^.next - norāde uz aiznākamo elementu,

current^.prior^.prior - norāde uz iepriekšējo elementu.

Divkāršsaistītais saraksts bez beigu rādītāja (1)



```

...
Node = record
  el: StdElement;
  next:
    NodePointer;
  prior:
    NodePointer
end;

```

Tekošā elementa priekšteča
meklēšana ir kļuvusi vienkāršāka:

```

FindPrior(L) => current := current^.prior;
               icurrent := icurrent - 1;

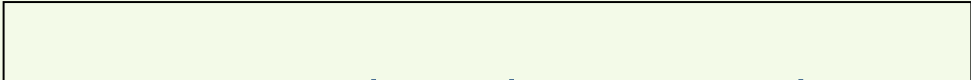
```

No vienkāršsaistīta saraksta atšķiras tikai šādas piecas operācijas:

FindPrior, Findith,
InsertAfter, InsertBefore,

Delete

Divkāršsaistītais saraksts bez beigu rādītāja (2)

```
procedure FindPrior (var L: List);  
  {Sarakstā L^ meklē tekošā elementa priekštecī, kas kļūs par  
jauno  
tekošo elementu}  
begin  
  if CurPos(L) > 1 then with L^ do  
    begin  
        
      current:= current^.prior;  icurrent:= icurrent - 1  
    end  
  end;  
end;
```

Divkāršsaistītais saraksts bez beigu rādītāja (3)

```

procedure InsertAfter (var L: List; e: StdElement);
{Sarakstā L^ jaunu elementu e pievieno aiz tekošā elementa, un tas kļūst par
jauno
tekošo elementu}
var p: NodePointer;
begin
  if not Full(L) then with L^ do
    begin
      new(p);  p^.el:= e;
      if Empty(L) then
        begin
          head:= p;  p^.next:= nil;  p^.prior:= nil
        end
      else
        begin
          p^.next:= current^.next;
          p^.prior:= current;
          if not Last(L) then current^.next^.prior:= p;
          current^.next:= p
        end
      current:= p;  current:= current + 1;  n:= n + 1
    end
  end;
end;

```

tukšs}

tukšs}

saites}

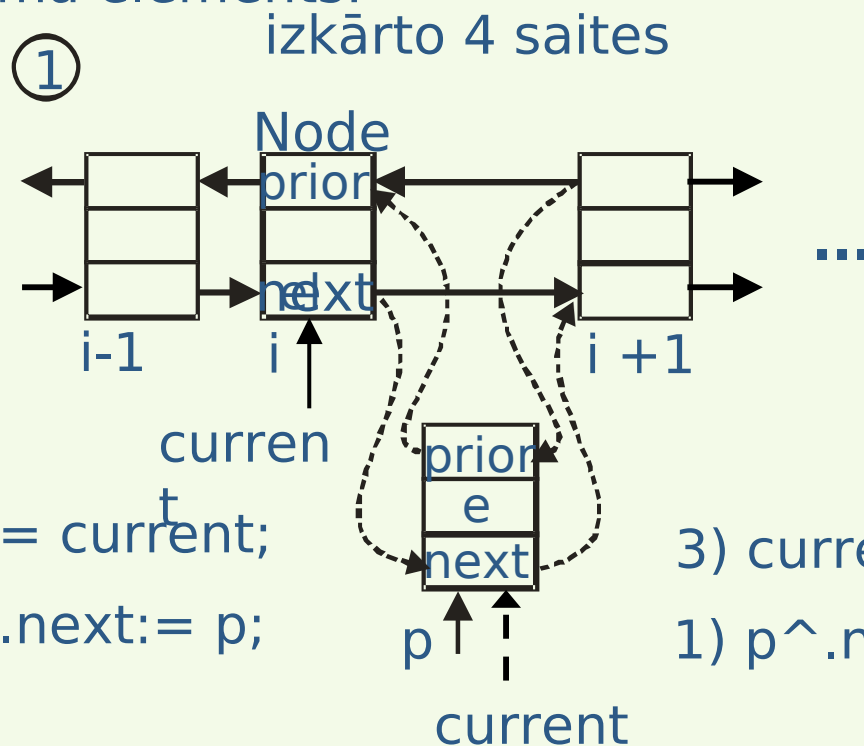
{saraksts ir

{saraksts nav

{izkārto

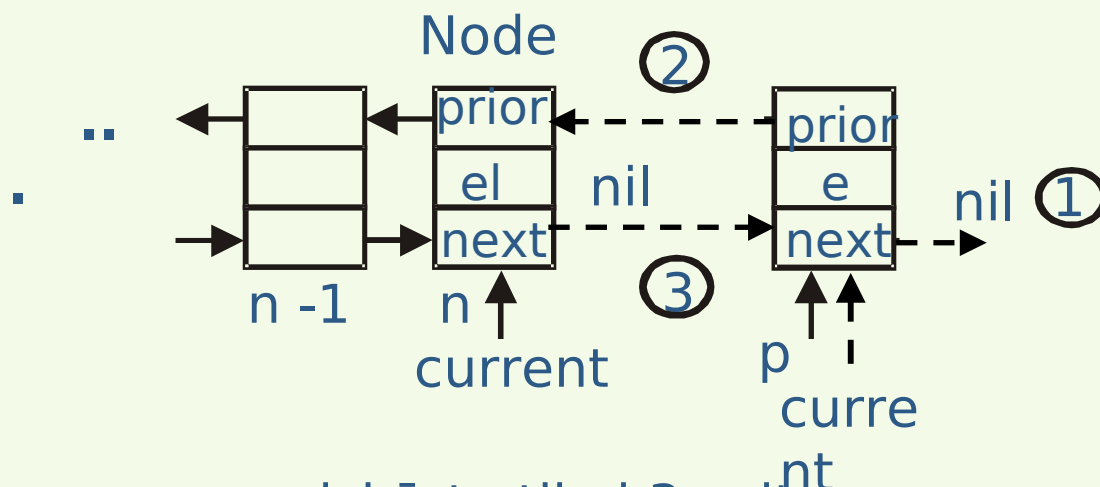
Divkāršsaistītais saraksts bez beigu rādītāja (4)

Saišu izkārtošana, ja saraksts nav tukšs un tekošais ir vidējā posma elements:



Divkāršsaistītais saraksts bez beigu rādītāja (5)

② Saišu izkārtošana, ja tekošais ir pēdējais elements:



izkārto tikai 3 saites:

- 1) $p^{\wedge}.next := current^{\wedge}.next;$
- 2) $p^{\wedge}.prior := current;$
- 3) $current^{\wedge}.next := p;$

Divkāršsaistītais saraksts bez beigu rādītāja (6)

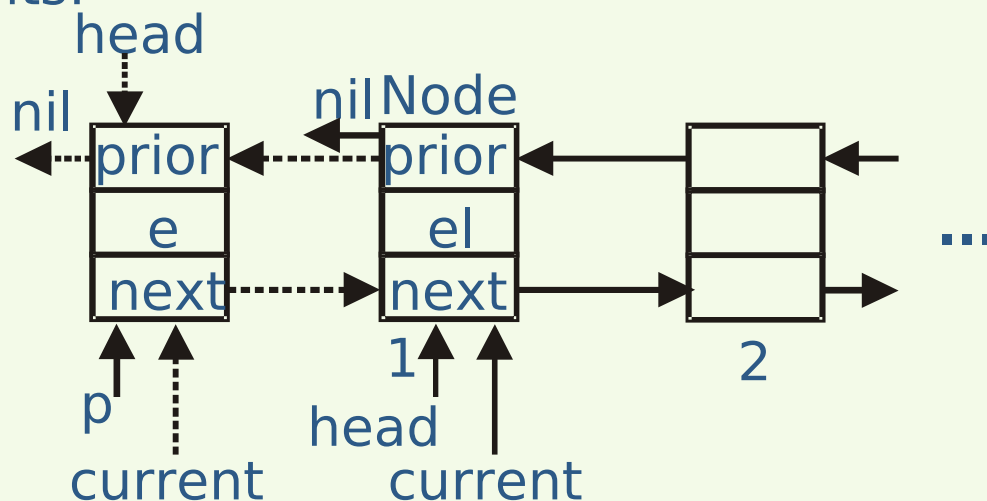
```

procedure InsertBefore (var L: List; e: StdElement);
{Sarakstā L^ pirms tekošā elementa pievieno jaunu elementu e, kas
klūst                                par
tekošo elementu}
var p: NodePointer;
begin
    if not Full(L) then with L^ do
        if not Empty(L) then
            if current = head then
                {pievieno pirms
1.elementa}
                begin
                    new(p); p^.el:= e;
                    head:= p;
                    p^.prior:= nil; p^.next:= current;
                    current^.prior:= p;
                    current:= p; n:= n + 1
                end
            else begin
                {tekošais nav 1.
elements}
                FindPrior(L); InsertAfter(L,e)
            end
        else InsertAfter(L,e)
    end
    {saraksts ir
pilns}

```

Divkāršsaistītais saraksts bez beigu rādītāja (7)

Jauna elementa pievienošana, ja tekošais ir pirmais elements:



Divkāršsaistītais saraksts bez beigu rādītāja (8)

```

procedure Delete (var L: List);
{Sarakstā L^ dzēš tekošo elementu un vairumā gadījumu nākamais
elements
  klūst par tekošo elementu}
var p: NodePointer;
begin
  if not Empty(L) then with L^ do
    begin
      p:= current;
      if current = head then {dzēš 1.
elementu}
        begin
          if n = 1 then {1. elements -
vienīgais}
            begin
              head:= nil; current:= nil; icurrent:= 0
            end
          else {1. elements sarakstā nav vienīgais}
            begin
              head:= current^.next;
              current^.next^.prior:= nil;
              current:= head
            end
          end
        end
      end
    end
  end

```

Divkāršsaistītais saraksts bez beigu rādītāja (9)

```

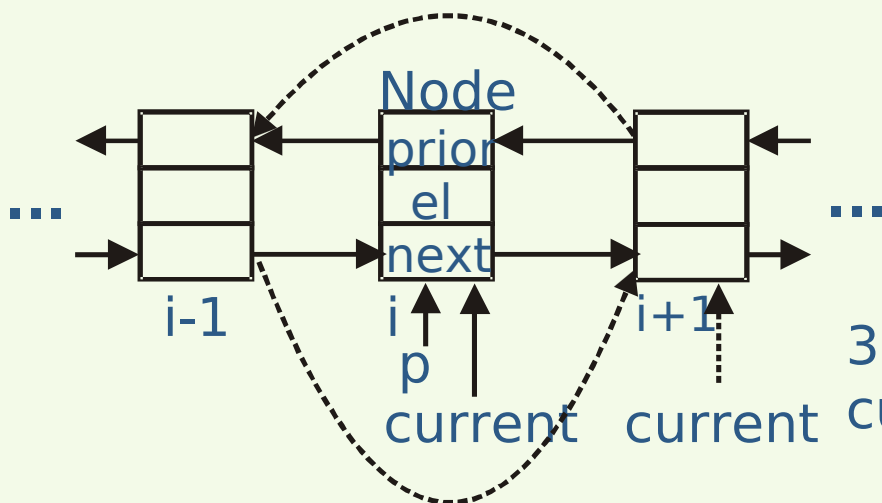
else if current^.next = nil then      {Last(L) - dzēš pēdējo
elementu}
    begin
        current^.prior^.next:= nil;
        current:= current^.prior;
        icurrent:= icurrent - 1
    end
else                                  {dzēš vidējā posma
elementu}
    begin
        current^.prior^.next:= current^.next;      {izkārto 2
saites}
        current^.next^.prior:= current^.prior;
        current:= current^.next
    end;
dispose (p);    n:= n -1              {kopējais
zars}
end
end

```

Divkāršsaistītais saraksts bez beigu rādītāja (10)

Saišu izkārtošana, ja dzēš vidējā posma elementu:

2) $\text{current}^{\wedge}.\text{next}^{\wedge}.\text{prior} := \text{current}^{\wedge}.\text{prior};$



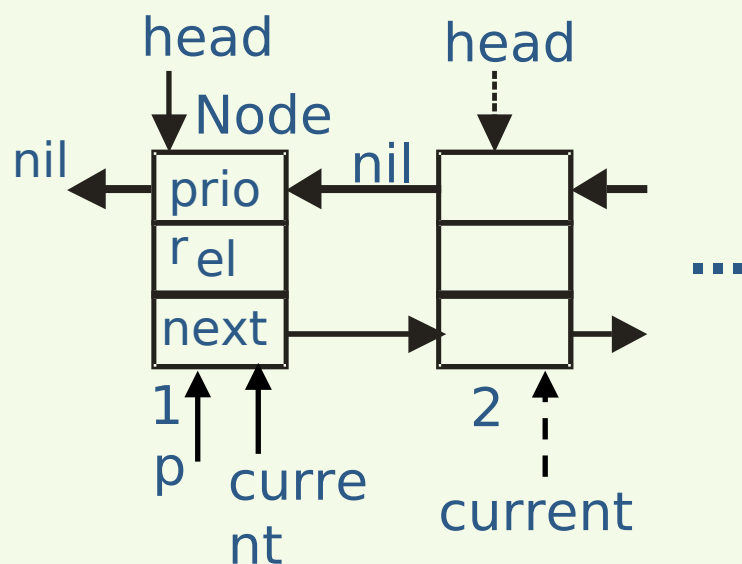
3)

$\text{current} := \text{current}^{\wedge}.\text{next};$

1) $\text{current}^{\wedge}.\text{prior}^{\wedge}.\text{next} := \text{current}^{\wedge}.\text{next};$

Divkāršsaistītais saraksts bez beigu rādītāja (11)

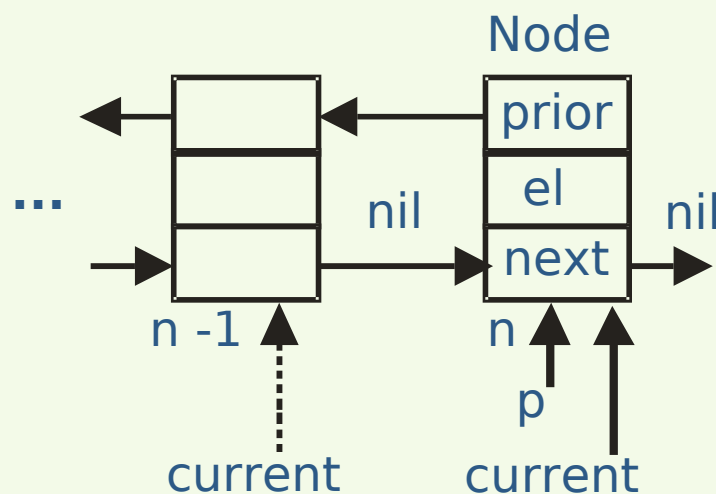
Saišu izkārtošana, ja dzēš 1. elementu, kas nav vienīgais sarakstā:



```
head:=current^.next;  
current^.next^.prior:=nil;  
current:=head;
```

Divkāršsaistītais saraksts bez beigu rādītāja (12)

Saišu izkārtošana, ja dzēš pēdējo elementu:



```
current^.prior^.next:=nil;  
current:=current^.prior;  
!current:=icurrent-1;
```

Divkāršsaistītais saraksts bez beigu rādītāja (12)

```

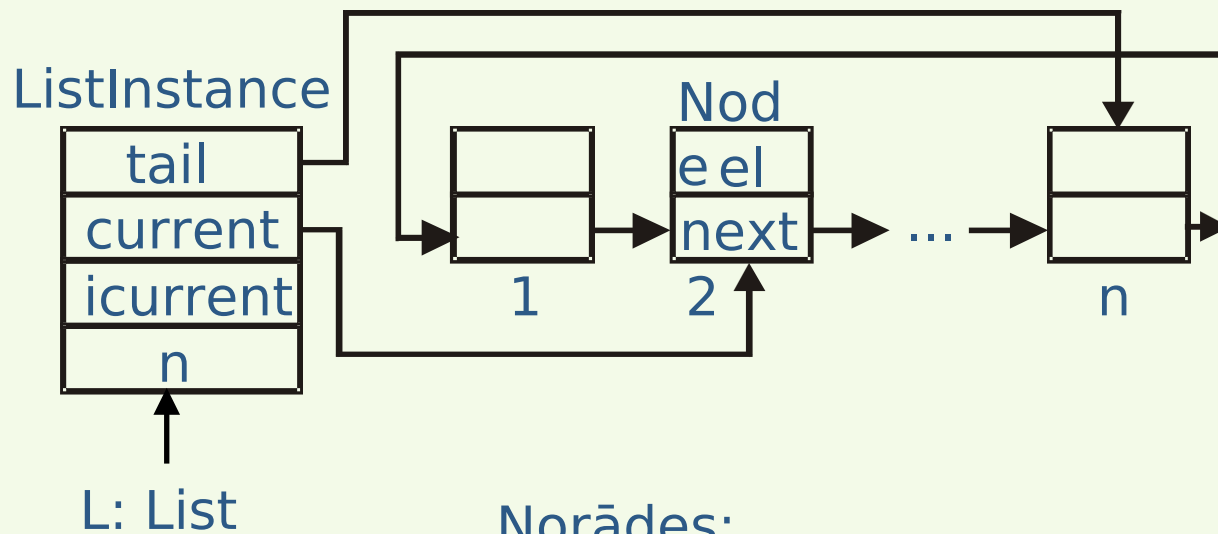
procedure Findith (var L: List; i: Count);
{Sarakstā L^ meklē elementu ar kārtas numuru i. Ja meklēšana ir
sekmīga,
sameklētais elements kļūst par tekošo elementu}
var k: Count;
begin
    if (not Empty(L)) and (i <= Size(L)) then with L^ do
        begin
            if i <= (n div 2) then
                {atrodas tuvāk
sāukumam}
                begin
                    current:= head;
                    for k:= 1 to i-1 do current:= current^.next
                end
            else
                {atrodas tuvāk
beigām}
                begin
                    FindLast(L);
                    for k:= n downto i + 1 do current:= current^.prior
                end;
            end;
        end;
    end;
end;

```


Cirkulārie saraksti

(circular lists) (1)

Vienkāršsaistītais cirkulārais saraksts:



Norādes:

pirmais elements: $\text{tail}^{\wedge}.\text{next}$

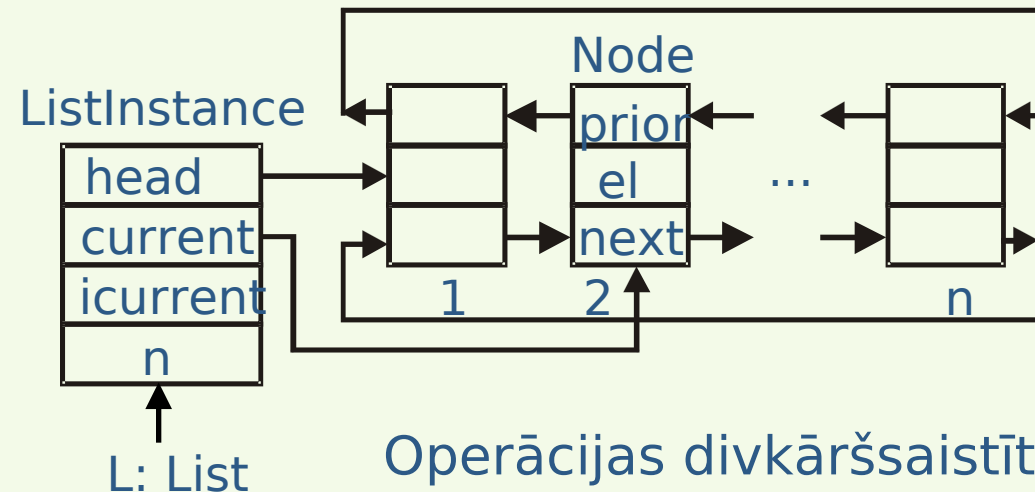
pēdējais elements: **tail**

tekošais elements: **current**

Cirkulārie saraksti

(circular lists) (2)

Divkāršsaistītais cirkulārais saraksts:



Norādes:
 pirmais elements: head
 pēdējais elements: head[^].prior
 tekošais elements: current

Operācijas divkāršsaistītā cirkulārā sarakstā:

Create,	*First,	*FindFirst,
InsertAfter,		
*Terminate,	Last,	FindLast,
InsertBefore,		
*Size,	FindNext,	Delete,
*CurPos,	FindPrior,	*Update
*Empty,	FindKey,	*Retrieve,
*Full,	Findith,	

Divkāršsaistītais cirkulārais saraksts (1)

```
procedure FindNext (var L: List);
begin
    if not Empty(L) then with L^ do
        begin
            current:= current^.next;
            if icurrent = n then icurrent:= 1
                else icurrent:= icurrent + 1;
        end
    end;
end;

procedure FindPrior (var L: List);
begin
    if not Empty(L) then with L^ do
        begin
            current:= current^.prior;
            if icurrent = 1 then icurrent:= n
                else icurrent:= icurrent - 1;
        end
    end;
end;
```

Divkāršsaistītais cirkulārais saraksts (2)

```
procedure FindLast (var L: List);  
begin  
    if CurPos(L) <> Size(L) then with L^ do  
        begin  
            current:= head^.prior;  
            icurrent:= n  
        end  
    end;  
end;
```

Divkāršsaistītais cirkulārais saraksts (3)

```

procedure FindKey (var L: List; tkey: KeyType; var found: boolean);
{Cirkulārajā sarakstā L^ meklē elementu, kura atslēgas lauka vērtība
ir tkey. Ja meklēšana ir sekmīga, sameklētais elements kļūst par tekšo
elementu}
var p: NodePointer; k: Count;
begin
    found:= false;
    if not Empty(L) then with L^ do
        begin
            k:= 1; p:= head;
            while (p^.next <> head ) and (p^.el.key <> tkey) do
                begin
                    {meklē
                    elementu}
                    p:= p^.next; k:= k+1
                end;
            if p^.el.key = tkey then
                begin
                    {sekmīga
                    meklēšana}
                    current:= p;
                    icurrent:= k;
                    found:= true;
                end
            end
        end
    end
end

```

Divkāršsaistītais cirkulārais saraksts (5)

```

procedure Findith(var L: List; i: Count);
{Cirkulārajā sarakstā L^ meklē elementu ar kārtas numuru i. Ja
meklēšana ir
sekmīga, sameklētais elements kļūst par tekošo elementu}
var k: Count;
begin
  if (not Empty(L)) and (i <= L^.n) then with L^ do
    begin
      if i <= (n div 2) then {elements atrodas tuvāk
sāukumam}
        begin
          current:= head;
          for k:= 1 to i - 1 do current:= current^.next
        end
      else {elements atrodas tuvāk saraksta
beigām}
        begin
          current:= head^.prior;
          for k:= n downto i + 1 do current:= current^.prior
        end;
      current:= i
    end
  end
end;

```

Divkāršsaistītais cirkulārais saraksts (5)

```

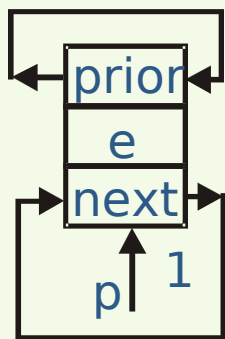
procedure InsertAfter (var L: List; e: StdElement);
{Cirkulārajā sarakstā L^ aiz tekošā elementa pievieno jaunu
elementu e, kas
klūst par tekošo elementu}
var p: NodePointer;
begin
  if not Full(L) then with L^ do
    begin
      new(p); p^.el:= e;
      if Empty(L) then {saraksts ir
tukšs}
        begin
          head:= p; p^.next:= p; p^.prior:= p
        end
      else {saraksts nav
tukšs}
        begin {izkārto 4
saites}
          current^.next^.prior:= p;
          p^.next:= current^.next;
          p^.prior:= current;
          current^.next:= p
        end
      end;
      current:= p; icurrent:= icurrent + 1; n:= n + 1
    end
  end;
end;

```

Divkāršsaistītais cirkulārais saraksts (6)

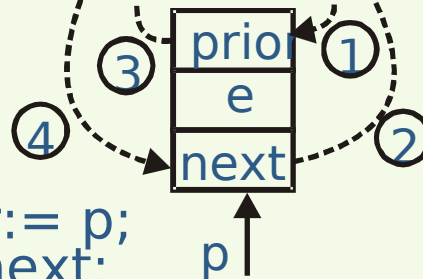
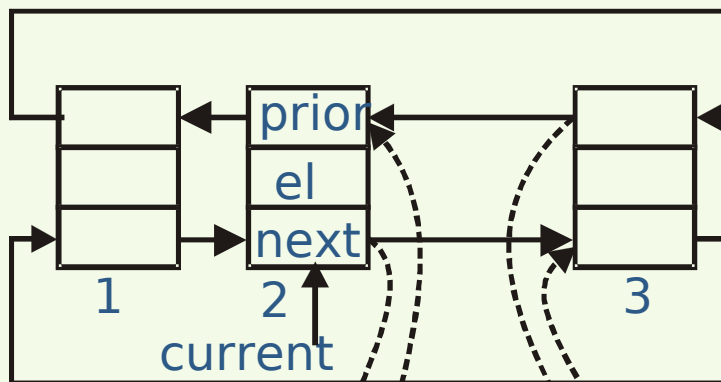
Saišu izkārtošana:

if Empty(L) then



```
p^.el := e;
p^.next := p;
p^.prior := p;
```

if not Empty(L) then



- 1) $\text{current}^{\wedge}.\text{next}^{\wedge}.\text{prior} := p;$
- 2) $p^{\wedge}.\text{next} := \text{current}^{\wedge}.\text{next};$
- 3) $p^{\wedge}.\text{prior} := \text{current};$
- 4) $\text{current}^{\wedge}.\text{next} := p;$

Divkāršsaistītais cirkulārais saraksts (7)

```

procedure InsertBefore (var L: List; e: StdElement);
{Cirkulārajā sarakstā L^ pirms tekošā elementa pievieno jaunu
elementu e, kas
kļūst par tekošo elementu}
begin
    if not Full(L) then with L^ do
        begin
            if not Empty(L) then
                begin
                    {FindPrior(L)}
                    current:= current^.prior;
                    if icurrent = 1 then icurrent:= n
                        else icurrent:= icurrent - 1
                    end;
                    InsertAfter(L, e);
                    if current^.next = head then
                        {Last(L)}
                        begin
                            {pēdējais elements kļūst par
                            pirmo}
                            head:= current;
                            icurrent:= 1
                        end
                    end
                end
            end
        end
    end
end;

```

Divkāŗšsaistītais cirkulārais saraksts (8)

```
function Last(L:List): boolean;  
  {Cirkulārajā sarakstā L^ pārbauda, vai pēdējais  
elements ir  
tekošais elements}  
begin  
  Last:= L^.current = L^.head^.prior  
end;
```



Divkāršsaistītais cirkulārais saraksts (9)

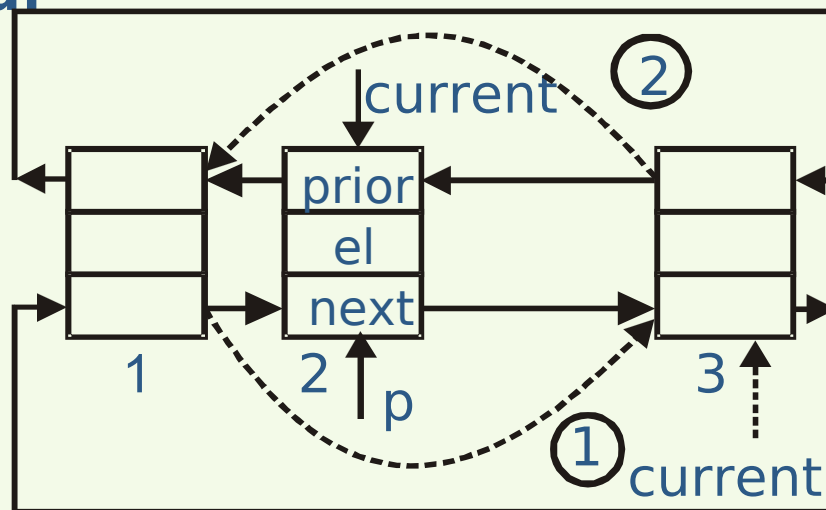
```

procedure Delete (var L: List);
{Cirkulārajā sarakstā L^ dzēš tekošo elementu, tekošā elementa
pēctecis kļūst
par tekošo elementu}
var p: NodePointer;
begin
    if not Empty (L) then with L^ do
        begin
            p:= current;
            if n = 1 then {saraksts kļūs
tukšs}
                begin
                    head:= nil; current:= nil; icurrent:= 0;
                end
            else {sarakstā ir vairāki
elementi}
                begin {izkārto 2
saites}
                    current^.prior^.next:= current^.next;
                    current^.next^.prior:= current^.prior;
                    if current^.next = head then icurrent:= 1;
                end
            end
        end
    end
{Last}

```

Divkāršsaistītais cirkulārais saraksts (10)

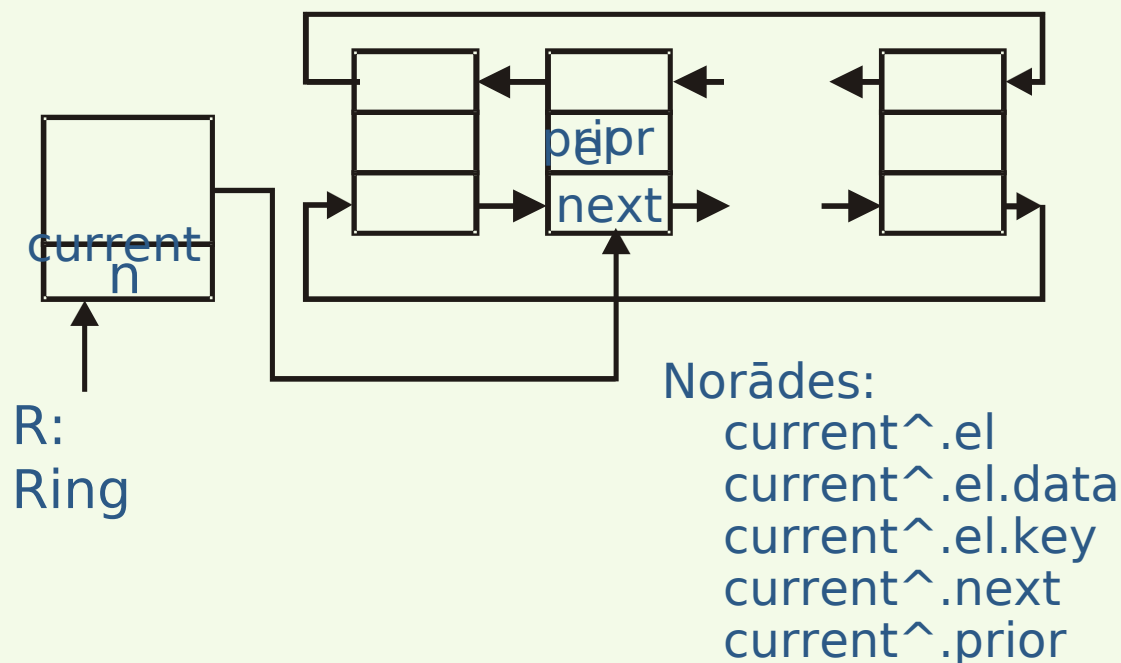
Saišu izkārtošana, ja dzēš vidējā posma elementu:



- 1) $\text{current}^{\wedge}.\text{prior}^{\wedge}.\text{next} := \text{current}^{\wedge}.\text{next};$
- 2) $\text{current}^{\wedge}.\text{next}^{\wedge}.\text{prior} := \text{current}^{\wedge}.\text{prior};$

$\text{current} := \text{current}^{\wedge}.\text{next};$

Divkāršsaistītais gredzens



Gredzens ir cirkulārs saraksts, kuram nav ne sākuma, ne beigu.

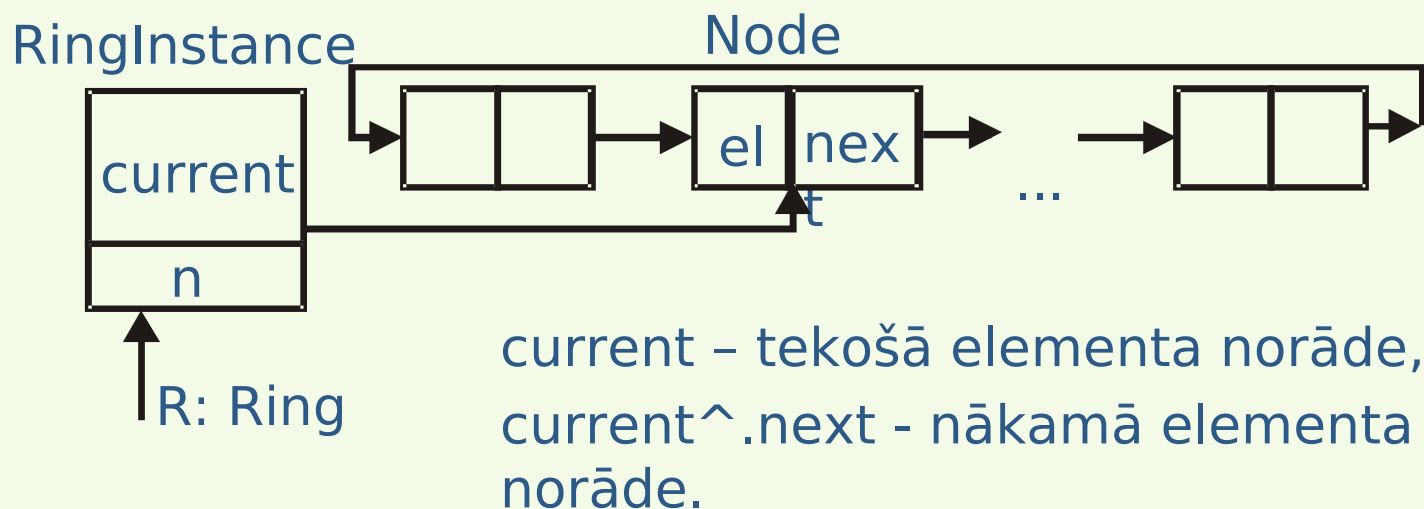
Reizēm lauku `current` vadības struktūrā tomēr lieto. Tādā gadījumā `current` vērtība atbilst jaunā elementa izvietojumasecībai, gredzenu veidojot.

Gredzena apstrādes operācijas:

Create, Terminate, Size, Full, Empty, FindNext, FindPrior,

Nav paredzētas šādas operācijas: FindKey, Insert, InsertAfter, InsertBefore, Delete, Update, Retrieve, CurPos, First, Last, FindFirst, FindLast, FindIth,

Vienkāršsaistītais gredzens (1)



Vienkāršsaistītais gredzens (2)

Dažas vienkāršsaistītā gredzena apstrādes operācijas:

```
procedure FindPrior (var R: Ring);  
  {Vienkāršsaistītajā gredzenā  $R^{\wedge}$  meklē tekošā elementa  
priekštecī, kas  
  kļūst par tekošo elementu}  
  var p, q: NodePointer;  
  begin  
    if Size(R) > 1 then with  $R^{\wedge}$  do  
      begin  
        p:= current^.next;  q:= current;  
        while p <> current do  
          begin  
            q:= p; p:= p^.next  
          end;  
        current:= q  
      end  
    end;  
end;
```

Vienkāršsaistītais gredzens (3)

```

procedure FindKey (var R:Ring; tkey: KeyType; var found:
boolean);
{Vienkāršsaistītajā gredzenā R^ meklē elementu, kura
atslēgas lauka
vērtība ir tkey. Ja meklēšana ir sekmīga, sameklētais elements
klūst par
tekošo elementu}
var p: NodePointer;
begin
    found:= false;
    if not Empty(R) then with R^ do
        begin
            p:= current;
            while (p^.next <> current) and (p^.el.key <> tkey)
do
                p:= p^.next;           {meklē
elementu}
            if p^.el.key = tkey then      {sekmīga
meklēšana}
                begin
                    found:= true;
                    current:= p;
                end
            end
        end
    end
end

```


Hronoloģiski sakārtotais saraksts (HSS)

(Chronologically ordered list)

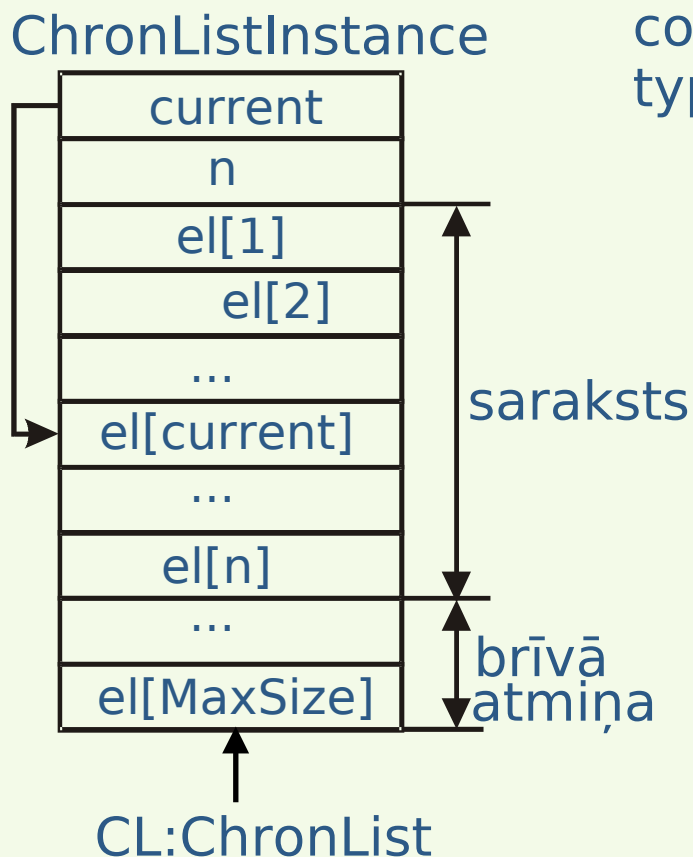
Elementu kārtojuma kritērijs sarakstā: **elementu pievienošanas laiks.**

Jaunus elementu pievieno tikai saraksta beigās.

Operācija Delete neietekmē hronoloģisko kārtību.
Ja tekošo elementu dzēš, tad operācija Insert to pievienotu saraksta beigās.

Vektoriālajā formā attēlotais HSS

(1)



```

const MaxSize = 500;
type Position = 1 .. MaxSize;
Count = 0 .. MaxSize;
DataType = string;
KeyType = integer;
Edit = 1 .. 3;
StdElement = record
    data: DataType;
    key: KeyType
end;
ChronListInstance = record
    current, n: Count;
    el: array[Position] of
        StdElement
end;
ChronList =
    ^ChronListInstance;
  
```

Atšķirīga ir tikai operācija **Insert**

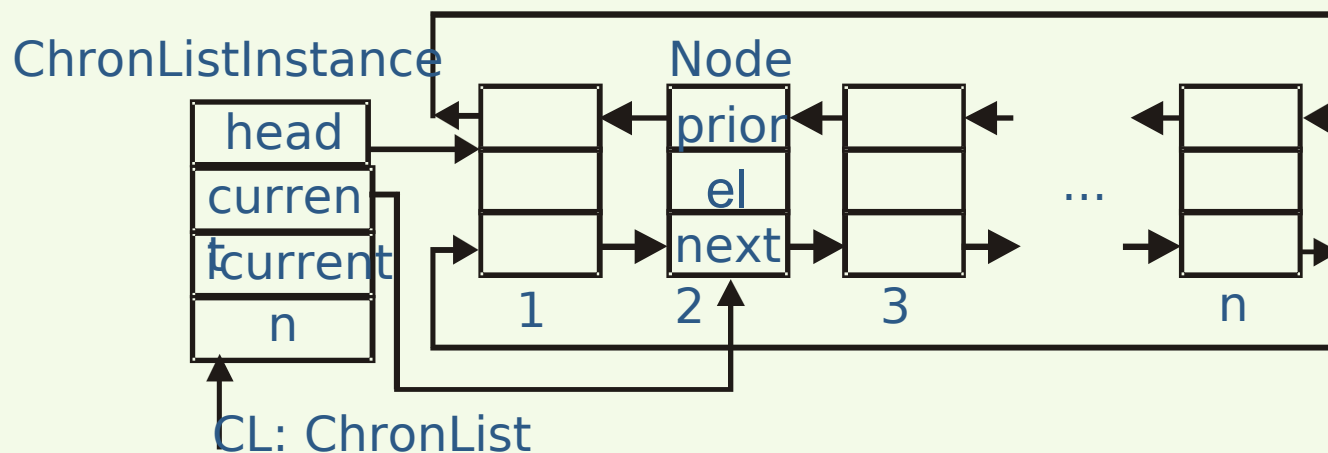
[^]ChronListInstance;

Vektoriālajā formā attēlotais HSS (2)

```
procedure Insert (CL: ChronList; e: StdElement);  
{Saraksta CL^ beigās pievieno jaunu elementu e}  
begin  
    if not Full(CL) then    with CL^ do  
        begin  
            n:= n + 1;  
            el[n]:= e;  
            current:= n  
        end  
    end;  
end;
```

Saistītajā formā attēlotais HSS (1)

Izvēlas divkāršsaistītu cirkulāru sarakstu HSS veidošanai:



```
const MaxSize = 500;
type Count = 0 .. MaxSize;
Edit = 1 .. 3;
DataType = string;
KeyType = integer;
StdElement = record
    data: DataType;
    key: KeyType
end;
```

```
Node = record
    el: StdElement;
    next, prior: NodePointer end;
NodePointer = ^Node;
ChronListInstance = record
    head, current: NodePointer;
    icurrent, n: Count
end;
ChronList = ^ChronListInstance;
```

Saistītajā formā attēlotais HSS (2)

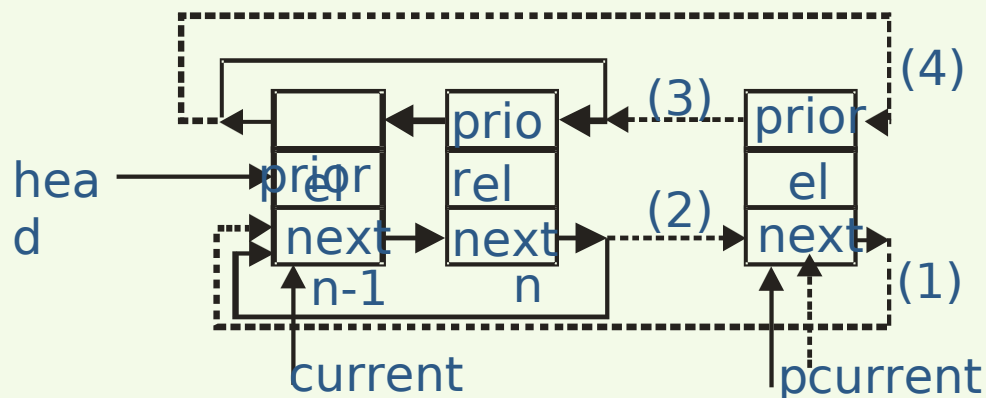
```

procedure Insert (var CL: ChronList; e: StdElement);
{Hronoloģiski sakārtotā saraksta CL^ beigās pievieno jaunu elementu
e}
var p: NodePointer;
begin
    if not Full(CL) then with CL^ do
        begin
            new (p); p^.el:= e;
            if Empty(CL) then {saraksts ir
tukšs}
                begin
                    head:= p; p^.next:= p; p^.prior:= p
                end
            else {saraksts nav
tukšs}
                begin {izkārtoto 4
saites}
                    p^.next:= head;
                    head^.prior^.next:= p;
                    p^.prior:= head^.prior;
                    head^.prior:= p
                end
            end
        end
    end
end;

```

Divkāršsaistītais cirkulārais HSS

Saišu izkārtošana, pievienojot jaunu elementu:

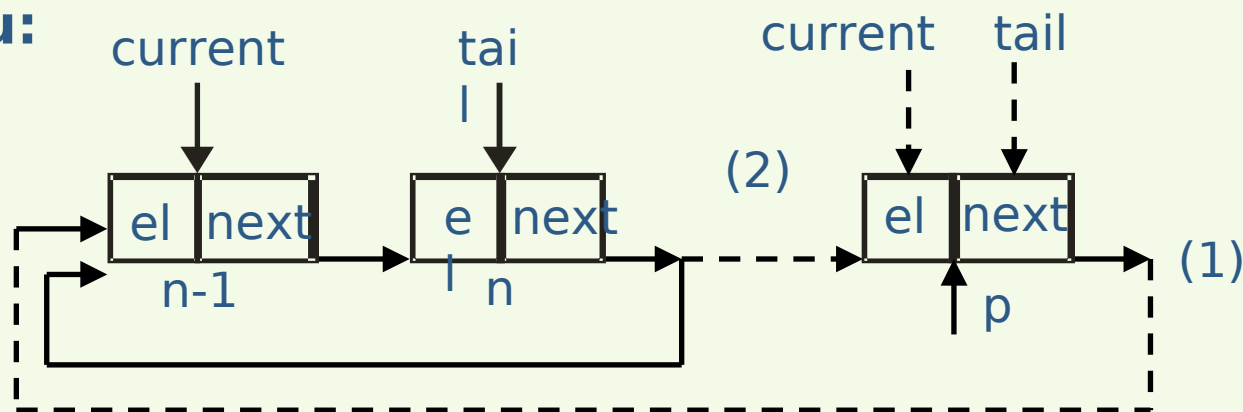


- 1) $p^{\wedge}.next := head;$
- 2) $head^{\wedge}.prior^{\wedge}.next := p;$
- 3) $p^{\wedge}.prior := head^{\wedge}.prior;$
- 4) $head^{\wedge}.prior := p;$

$current := p;$

Vienkāršsaistītais cirkulārais HSS

Saišu izkārtošana, pievienojot jaunu elementu:



1) $p^{\wedge}.next := tail^{\wedge}.next;$

2) $tail^{\wedge}.next := p;$

$current := p; \quad tail := p;$

Hronoloģiski sakārtotais saraksts (HSS)

(Chronologically ordered list)

HSS var izmantot arī kā steku vai

rindu:

Steka operācija

Push (CL, e)

Pop (CL, e)

LIFO

Rindas operācija

Enqueue (CL, e)

Serve (CL, e)

FIFO

HSS operācijas

Insert (CL, e)

FindLast (CL)

Retrieve (CL, e)

Delete (CL)

HSS operācijas

Insert (CL, e)

FindFirst (CL)

Retrieve (CL, e)

Delete (CL)

Sašķirotais saraksts (1)

(sorted list)

Elementi sašķirotajā sarakstā sakārtoti pēc atslēgas lauku vērtībām:

$key_{i-1} < key_i < key_{i+1}, \quad i = 2, 3, \dots, n-1.$

Sašķirotā saraksta veidošana:

- 1) ar operāciju **Insert** – sākot no saraksta sākuma, sameklē pirmo elementu, kura **current[^].el.key > e.key**, un jauno elementu pievieno sarakstā pirms tā;
- 2) jaunu elementu sarakstam pievieno kā parasti un laiku pa laikam ar šķirošanas operāciju **Sort** saraksta elementus sašķiro augošā secībā.

Meklēšanas operācijai **FindKey** ir divi algoritmi atbilstoši diviem saraksta attēlojuma modeļiem:

1) vektoriālā formā attēlotais modelis – operācijas FindKey algoritmi:

- a) lineārā meklēšana, efektivitāte $O(n)$;
- b) binārā meklēšana, efektivitāte $O(\log_2 n)$;
- c) interpolatīvā meklēšana, efektivitāte $O(\log_2 n)$.

2) saistītā formā attēlotais modelis - operācijas FindKey algoritms: tikai lineārā meklēšana, efektivitāte $O(n)$.

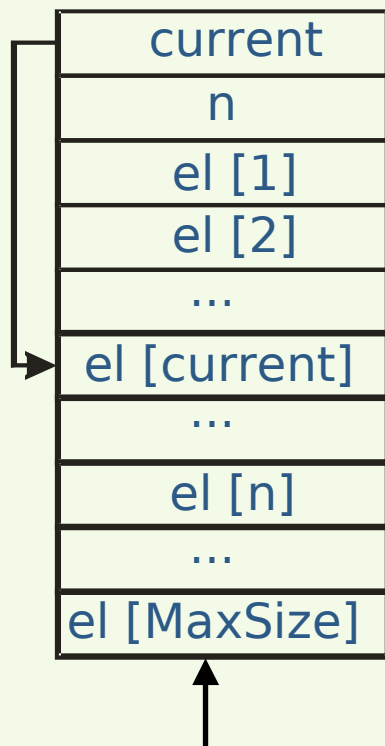
Atskirīgas operācijas: **BinSearch, Findkey, Insert, Update.**

Lietojuma priekšrocības vektoriālā formā attēlotajam sašķirotajam

Sašķirotais saraksts (2)

(sorted list)

SortedListInstance



SL:SortedList

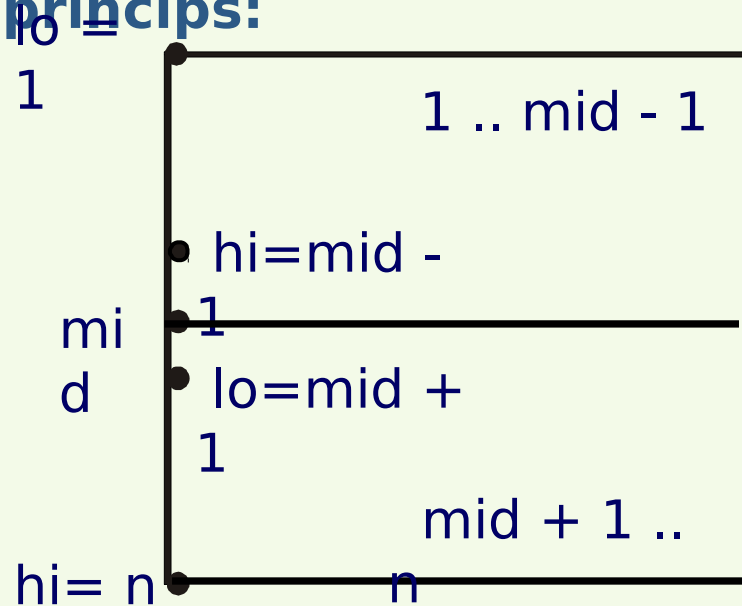
```

const MaxSize = 500;
type Position = 1 .. MaxSize;
Count = 0 .. MaxSize;
DataType = string;
KeyType = integer;
StdElement = record
    data: DataType;
    key: KeyType
end;
Edit = 1 .. 3;
SortedListInstance = record
    current: Count;
    n: Count;
    el: array [Position] of StdElement
end;
SortedList = ^SortedListInstance;
  
```

Sašķirotais saraksts(3) (sorted list)

Binārās meklēšanas algoritma darbības

principi:



if $el[mid].key > tkey$ then $hi := mid - 1$;

$mid := (lo + hi) \div 2$;

if $el[mid].key < tkey$ then $lo := mid + 1$;

if $el[mid].key = tkey$ then found := true;

if $mid = 1$ then found := false;

Sašķirotai saraksts (4)

(sorted list)

Uzlabotais (Bottenbruch) binārās meklēšanas algoritms:

```
procedure BinSearch (SL: SortedList; tkey: KeyType; var lo, hi:
Count);
var mid: Count;
begin
    with SL^ do
        while lo < hi do
            begin
                mid:= (lo + hi + 1) div 2;
                if el[mid].key > tkey then hi:= mid - 1
                    else lo:= mid
            end
        end
    end;
```

Sašķirotais saraksts (5)

(sorted list)

```

procedure FindKey (SL: SortedList; tkey: KeyType; var
found:boolean);
{Binārās meklēšanas algoritms. Sašķirotajā sarakstā SL^ meklē
elementu,
kura atslēgas lauka vērtība ir tkey. Ja meklēšana ir sekmīga,
sameklētais
elements kļūst par tekošo elementu.}
var lo, hi: Count;
begin
    found:= false;
    if not Empty(SL) then with SL^ do
        begin
            lo:= 0;    hi:= n;                {meklēšanas
diapazons}
            BinSearch (SL, tkey, lo, hi);      {elementa
meklēšana}
            if (hi <> 0) and (el[hi].key = tkey) then
                begin                          {sekmīga
meklēšana}
                    current:= hi;

```

Sašķīrotais saraksts (6)

(sorted list)

```

procedure Insert (var SL: SortedList; e: StdElement);
{Sašķīrotajā sarakstā SL^ jaunajam elementam e sameklē vietu un to
izvieto
pozīcijā hi+1}
var lo, hi, k: Count;
begin
    if not Full(SL) then with SL^ do
        begin
            if Empty(SL) then current:= 1                {saraksts ir
tukšs}
            else                                           {meklē
vietu}
                begin
                    lo:= 0;    hi:= n;
                    BinSearch (SL, e.key, lo, hi);
                                {sameklēto pozīciju hi+1
                                atbrīvo}
                    for k:= n downto hi + 1 do el[k+1]:= el[k];
                    current:= hi + 1
                end
            el[current]:= e
        end
    end
{elementu izvieto pozīcijā
hi+1}

```

Sašķirotais saraksts (7)

(sorted list)

```
procedure Update (var SL: SortedList; e: StdElement; k: Edit);
{Sašķirotā sarakstā SL^ labo tekošā elementa saturu atbilstoši
labošanās
variantam k}
var temp: StdElement;
begin
  if not Empty(SL) then with SL^ do
    begin
      case k of
        1: el[current].data:= e.data;
        2: el[current].key:= e.key;
        3: el[current]:= e;
      end;
      if k>1 then
        begin
          temp:= el[current];
          Delete(SL);
          Insert(SL, temp);
        end;
    end;
  end;
end;
```

saturs}

saglabā}

dzēš}

{labots atslēgas lauka

{izlaboto elementu

{un tad sarakstā

{un no jauna izvietots saraksts}

Sašķirotais saraksts (8)

(sorted list)

Šis operācijas Update algoritms ir īss un viegli saprotams. Taču tam piemīt arī trūkums. Sašķīrotā sarakstā elementa dzēšana tiek izpildīta kā elementu pārsūtīšanas operācija, kuras izpildes efektivitātes kārtā ir $O(n)$.

Jauna elementa pievienošana sašķīrotajam sarakstam ir saistīta ar vietas meklēšanu jaunajam elementam un, lai šo sameklēto vietu pēc tam atbrīvotu, atkal nepieciešama elementu pārsūtīšana, kuras izpildes efektivitātes kārtā atkal ir $O(n)$.

Var izveidot arī tādu operācijas Update algoritmu, kas tiks izpildīts divreiz ātrāk.

Sašķirotais saraksts (9)

(sorted lists)

```
procedure Update (var SL: SortedList; e: StdElement; k: Edit);
{Sašķirotā sarakstā SL^ labo tekošā elementa saturu atbilstoši
labošanas variantam k}
var lo, hi, i: Count;
    temp: StdElement;
    oldkey: KeyType;
begin
    if not Empty(SL) then with SL^ do
        begin
            oldkey:= el[current].key; {fiksē veco atslēgas lauka
vērtību}

            case k of
            1: el[current].data:= e.data;
            2: el[current].key:= e.key;
            3: el[current]:= e;
            end;
```

Sašķirotais saraksts (10)

(sorted list)

```

if k>1 then                                {labots atslēgas lauka
saturš}
    begin
        temp:= el[current];                {saglabā izlaboto
elementu}
        {uzdod meklēšanas diapazonu un meklē
elementa vietu}
        if el[current].key < oldkey then
            begin lo:=0;  hi:= current end
        else begin lo:= current; hi:= n end;
        BinarySearch(SL, e.key, lo, hi);
        {elementam atbrīvo vietu un izvieta attiecīgajā
pozīcijā}
        if el[current].key < oldkey then begin
            for i:= current-1 downto hi+1 do el[i+1]:=
el[i];
            el[hi+1]:= x end
        else begin
            for i:= current+1 to hi do el[i-1]:= el[i];
            el[hi]= x end
    end

```

Rekursīvais binārās meklēšanas algoritms

```
procedure BinSearch (SL: SortedList; tkey: KeyType; var lo, hi:
Count);
var mid: Count;
begin
  if lo < hi then
    begin
      mid:= (lo + hi + 1) div 2;
      if el[mid].key > tkey then
        BinSearch(SL, tkey, lo, mid-1)
      else
        BinSearch(SL, tkey, mid, hi)
    end
  end;
end;
```

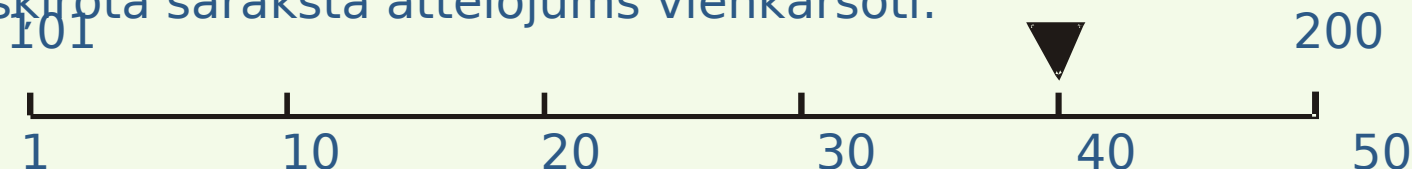
Interpolatīvā meklēšana

Meklēšanas procesā sarakstu dala noteiktā proporcijā, bet nevis uz pusēm.

Piemērs:

$n = 50$; $el[lo].key = 101$; $el[hi].key = 200$; $tkey = 180$.

Saskirotā saraksta attēlojums vienkāršoti:



Attālums no saraksta sakuma: $\frac{180 - 101 + 1}{200 - 101 + 1} = 0.8$;

Sameklētā elementa pozīcija: $0.8 * 50 =$

40;

Dalījuma punkta pos noteikšana:

$$\text{fract} = \frac{tkey - el[lo].key + 1}{el[hi].key - el[lo].key + 1}, \quad \text{pos} = \text{fract} * \underbrace{(hi - lo + 1)}_n$$

Multiplikatīvā meklēšana (Standish, 1980)

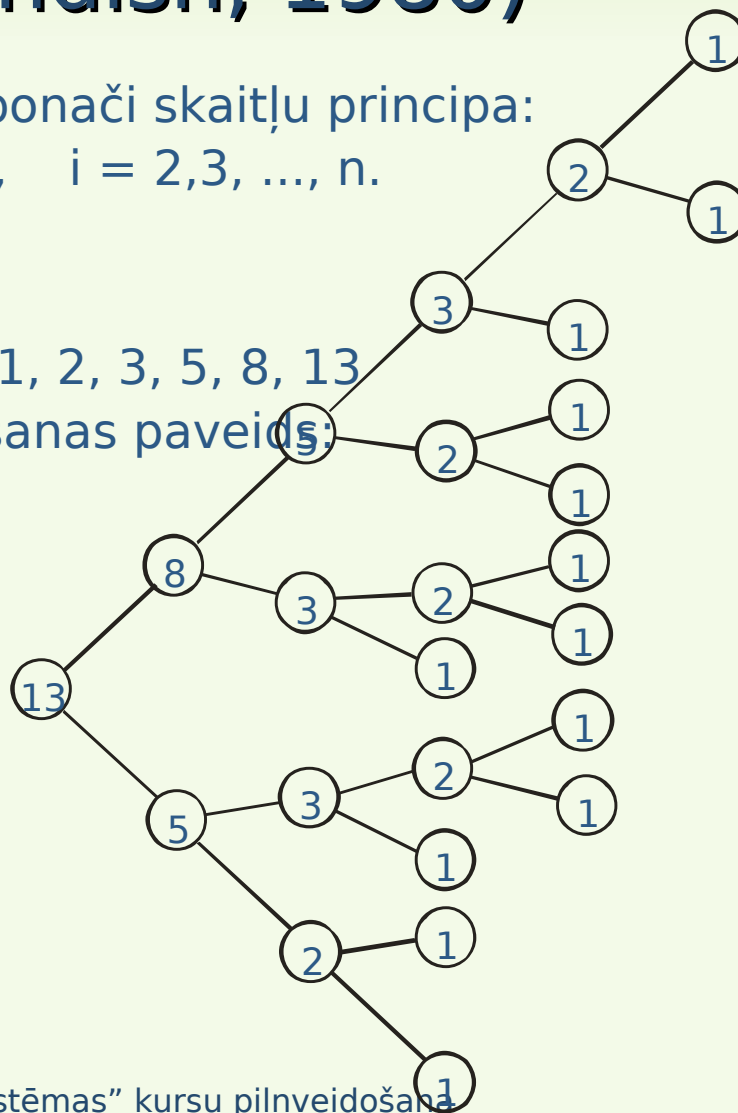
Sarakstu dala 2 daļās pēc Fibonači skaitļu principa:

$$F_0 = 0; \quad F_1 = 1; \quad F_i = F_{i-1} + F_{i-2}, \quad i = 2, 3, \dots, n.$$

Piemērs: $n = 13$;

Fibonači skaitļu virkne: 0, 1, 1, 2, 3, 5, 8, 13

Pamatā tas ir binārās meklēšanas paveids:



Saistītā formā attēlotais sašķirotais saraksts (1)

Operācijā **FindKey** lieto tikai lineārās meklēšanas algoritmu.

Ja lietotu binārās meklēšanas algoritmu, tad saraksta viduspunkts `mid` būtu jāmeklē, tā meklēšana ir $O(n)$ kārtas operācija, kurā tiek izpildītas darbības ar rādītājiem, visā meklēšanas procesā būtu nepieciešamas n šādas manipulācijas. Šī iemesla dēļ no binārās meklēšanas algoritma izmantošanas atsakās.

Operācija **Insert** – sākot no saraksta sākuma, meklē vietu jaunajam elementam.

Meklēšanas procesa darbības:

```
current:=head;   icurrent:=1;           {sāk vietas  
meklēšanu}  
while current^.el.key < e.key do         {meklēšana  
nosacījums}  
begin  
    current:=current^.next;              {iestata nākamo  
elementu}  
    icurrent:=icurrent+1
```

Saistītā formā attēlotais sašķirotais saraksts (2)

Jauno elementu izvieto pirms tā elementa, uz ko norāda rādītājs *current*. Kad vietas sameklēta un jaunais elements sarakstā izvietots, tas kļūst par tekošo elementu sašķirotajā sarakstā.

Operācija **Update** – ja tiek labots atslēgas lauks *key*, tad elementam sarakstā sameklē jaunu vietu līdzīgi tam, kā tas tika darīts operācijā *Insert*. Vecajā vietā izlaboto elementu dzēš un izvieto jaunajā vietā pirms tā elementa, uz ko norāda rādītājs *current*. Vietas meklēšanu pabeidzot, attiecīgi izkārtoti arī elementu saites un precizē rādītājus.

Pēc lietojuma biežuma sakārtotais saraksts (1)

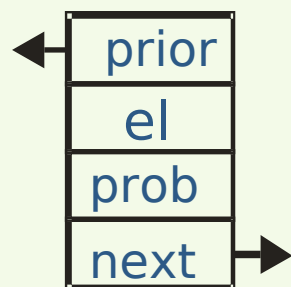
(Frequency ordered list)

Saraksta elementu kārtojuma kritērijs: to lietojuma biežums.

Visbiežāk lietotais elements atrodas saraksta sākumā, bet visretāk lietotais elements – saraksta beigās.

Katram elementam sarakstā būtu jāpievieno tā lietojuma varbūtība (probability). Praktiski tā nav zināma. Reizēm lietojuma varbūtību cenšas iepriekš prognozēt vai noteikt citādi (piemēram, pēc 80-20 likuma – kā to izmanto finansu istēmā).

Pēc lietojuma biežuma sakārtota saraksta elementa uzbūve:



Lauka *prob* varbūtība grūti nosakāma

vai arī nav zināma,

Lauka *prob* varbūtība ar laiku mainās.

Tāpēc izmanto pēc lietojuma biežuma sakārtota saraksta paveidu: pašorganizēto sarakstu (self organizing list).

Pēc lietojuma biežuma sakārtotais saraksts (2)

(Frequency ordered lists)

Ir 3 paņēmieni pašorganizētā saraksta veidošanai. Katram pašorganizēta saraksta veidošanas paņēmienam ir sava būtība un tās realizācijas metode.

1.paņēmiena būtība: ja ar kādu pašorganizētā saraksta elementu strādā, tad tas strauji pārvietojas virzienā uz saraksta sākumu, apejot visus elementus, kuriem ir mazāks lietojuma biežums.

2.paņēmiena būtība: ja ar kādu pašorganizētā saraksta elementu strādā, tad tas lēni (par vienu vietu) pārvietojas virzienā uz saraksta sākumu.

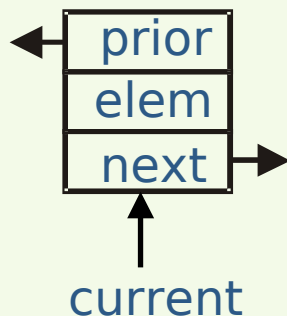
3.paņēmiena būtība: ja kāds pašorganizētā saraksta elements netiek lietots, tad tas pārvietojas virzienā uz saraksta beigām.

Pēc lietojuma biežuma sakārtotais saraksts (3)

(Frequency ordered lists)

1. metode pašorganizētā saraksta veidošanai:

1) katram elementam saistītajā sarakstā pievieno papildus lauku *freq*, kurā ieraksta elementa lietojuma biežumu:



Elementa un tā lauku piekļuve un norādes:

- `current^.elem`
- `current^.elem.el`
- `current^.elem.el.data`
- `current^.elem.el.key`
- `current^.elem.freq`
- `current^.next`
- `current^.prior`

Pēc lietojuma biežuma sakārtotais saraksts (4)

(Frequency ordered lists)

Dažas atšķirības, definējot elementu divkāršsaistītā sarakstā:

```
type FreqType = integer;           {parasti vesels pozitīvs  
skaitlis}  
  
StdElement = record  
  data: DataType;  
  key: KeyType  
end;  
AugElement = record  
  el: StdElement;  
  freq: FreqType  
end;                               {papildus lietojuma biežuma  
lauks}  
Node = record  
  elem: AugElement;  
  next, prior: NodePointer  
end;  
ieraksts}
```

Pēc lietojuma biežuma sakārtotais saraksts (5)

(Frequency ordered lists)

2) izpildot operācijas **FindKey**, **Findith**, **Retrieve**, **Update**, izmainās elementa lietojuma biežums:
 $\text{inc}(\text{elem}[\text{current}].\text{freq})$ – vektorālā formā attēlotajā sarakstā un
 $\text{inc}(\text{current}^\wedge.\text{elem}.\text{freq})$ – saistītajā sarakstā. Tekošais elements tiek pārvietots tuvāk saraksta sākumam, apejot visus elementus, kuriem mazāka *freq* vērtība, t.i., virzienā uz saraksta sākumu tiek sameklēts pirmais elements, kuram ir vienāda vai lielāka lietojuma biežuma *freq* vērtība. Tas elements, kurš atrodas aiz sameklētā elementa, tiek apmainīts vietām ar tekošo elementu, tādējādi tekošais elements tiek pārvietots tuvāk saraksta sākumam;

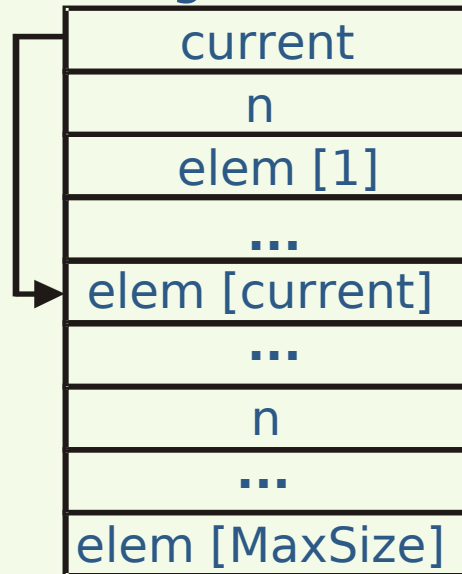
3) ar operāciju **Insert** jaunu elementu vienmēr pievieno saraksta beigās. Jaunam elementam lietojuma biežums

Pēc lietojuma biežuma sakārtotais saraksts (5)

(Frequency ordered lists)

Atšķirības, definējot vektoriālā formā atēloto pašorganizēto sarakstu:

SelfOrgListInstance



SOL: SelfOrgList

```
type AugElement = record
  el: StdElement;
  freq: FreqType;
end;
```

```
SelfOrgListInstance = record
  n: Count;
  elem: array{Position} of
    AugElement
end;
```

```
SelfOrgList =
  ^SelfOrgListInstance;
```

piekļuve:

```
elem [current].el
elem [current].el.data
elem [current].el.key
elem [current].freq
```

Pēc lietojuma biežuma sakārtotais saraksts (7)

(Frequency ordered lists)

2. metodes realizācija, veidojot pašorganizēto sarakstu;

- 1) elementam sarakstā papildus lauku freq nepievieno;
- 2) izpildot operācijas **FindKey**, **Findith**, **Retrieve**, **Update**, elements sarakstā tiek apmainīts vietām ar iepriekšējo elementu;
- 3) izpildot operāciju **Insert**, jaunu elementu pievieno saraksta beigās.

2. metode saistītajā formā attēlotā pašorganizētā saraksta veidošanai:

```
procedure Swap(var SOL: SelfOrgList);  
{Saistītajā sarakstā SOL^ tekošo un iepriekšējo elementu apmaina vietām}  
var temp: StdElement;  
begin  
  if (Size(SOL)>1) and (not First(SOL)) then with SOL^ do  
    begin  
      temp:= current^.el;  
      current^.el:= current^.prior^.el;  
      current^.prior^.el:= temp;  
      current:= current^.prior;  
      icurrent:= icurrent - 1;  
    end  
  end
```

Pēc lietojuma biežuma sakārtotais saraksts (8)

(Frequency ordered lists)

2. metode vektoriālajā formā attēlotā pašorganizētā saraksta veidošanai:

```
procedure Swap (var SOL: SefOrgList);  
{Vektoriālā formā attēlotā sarakstā SOL^ tekošo un iepriekšējo  
elementu apmaina vietām}  
var temp: StdElement;  
begin  
    if (Size (SOL)>1) and (not First(SOL)) then  with SOL^ do  
        begin  
            temp:= el[current];  
            el[current]:= el[current - 1];  
            el[current - 1]:= temp;  
            current:= current - 1  
        end  
    end;  
end;
```

Pēc lietojuma biežuma sakārtotais saraksts (9)

(Frequency ordered lists)

3. metode vektoriālajā formā attēlotā pašorganizētā saraksta veidošanai:

```
procedure Swap (var SOL: SelfOrgList);  
  {Tekošajā pozīcijā current esošo elementu dzēš un izvieto pirmajā  
pozīcijā}  
  var k: Position;  
      temp: StdElement;  
begin  
  if (Size(SOL)>1) and (not First(SOL)) then with SOL^ do  
    begin  
      temp:= el[current];  
      for k:= current-1 downto 1 do  
        el[k]:= el [k+1];  
        {elementu sarakstā  
dzēš}  
      el[1]:= temp;  
      {un tad izvieto  
1.pozīcijā}  
      current:= 1  
    end  
  end
```


Pēc lietojuma biežuma sakārtoti saraksti (10)

(Frequency ordered lists)

3. metode divkāršsaistīta cirkulāra pašorganizētā saraksta veidošanai:

```
procedure Swap(var SOL: SelfOrgList);
var p: NodePointer;
    temp: StdElement;
begin
    if (Size(SOL)>1) and (not First(SOL)) then with SOL^
do
    begin
        p:= current; temp:= current^.el;
{elementu dzēš}
        current^.prior^.next:= current^.next;
        current^.next^.prior:= curent^.prior; dispose (p);
        new(p); {un tad izvieto saraksta
sākumā}
        p^.el:= temp; p^.next:= head; p^.prior:=
head^prior;
        head^prior^.next:= p;
```

Pēc lietojuma biežuma sakārtotai saraksts (11)

(Frequency ordered lists)

1.metode vektoriālajā formā attēlotā pašorganizētā saraksta veidošanai:

```
procedure Swap(var SOL: SelfOrgList);
{Pašorganizētajā sarakstā SOL^, sākot ar tekošo pozīciju
current,
virzienā uz saraksta sākumu meklē elementu, kuram ir mazāks
lietojuma biežums freq, un apmaino to vietām ar tekošo
elementu}
var k: Position;
    temp: AugElement;
begin
    with SOL^ do
        begin
            {palielina lietojuma biežumu
par 1}
            elem[current].freq:= elem[current].freq+1;
            if (Size(SOL)>1) and (not First(SOL)) then
                begin
                    {meklē apmaināmo elementu -
```

Pēc lietojuma biežuma sakārtotai saraksts (12)

(Frequency ordered lists)

```
if elem[current].freq > elem[1].freq then k:= 1
else
```

```
begin
```

```
  k:= current-1;
```

```
  while elem[current].freq > elem[k].freq do
```

```
k:= k-1;
```

```
  k:= k+1
```

```
end;
```

```
temp:= elem[current];           {elementu mainīšana
```

```
vietām}
```

```
elem[current]:= elem[k];
```

```
elem[k]:= temp;
```

```
current:= k
```

```
end
```

```
end
```

```
end;
```

Pēc lietojuma biežuma sakārtotais saraksts (13)

(Frequency ordered lists)

1.metode divkāršsaistīta cirkulāra pašorganizētā saraksta veidošanai:

```
procedure Swap(var SOL: SelfOrgList);
{Pašorganizētajā sarakstā SOL^, sākot ar tekošo elementu
current, virzienā uz saraksta sākumu meklē elementu, kuram ir
mazāks lietojuma biežums freq, un apmaino to vietām ar tekošo
elementu}
var p: NodePointer;
    k: Count;
    temp: AugElement;
begin
    with SOL^ do
        begin
            {palielina lietojuma biežumu
par 1}
            current^.elem.freq:= current^.elem.freq+1;
            if (Size(SOL)>1) and (not First(SOL)) then
                begin
                    if current^.elem.freq <=
                    current^.prior^.elem.freq
```

Pēc lietojuma biežuma sakārtoti saraksti (14)

(Frequency ordered lists)

```
if current^.elem.freq > head^.elem.freq then
  begin
    p:= head; k:= 1
  end
else
  begin
    {meklē samaināmo
elementu}
    p:= current^.prior; k:= icurrent -1;
    while (current^.elem.freq > p^.elem.freq) do
      begin p:= p^.prior; k:= k - 1 end;
      p:= p^.next; k:= k+1
    end;
    {elementu apmaiņšana
vietām}
    temp:= current^.elem;
    current^.elem:= p^.elem;
    p^.elem:= temp;
    current:= p; icurrent:= k
  end
end
end
end
```

Pēc lietojuma biežuma sakārtotais saraksts (15)

(Frequency ordered lists)

Lineārās meklēšanas operācija divkāršsaistītā cirkulārā pašorganizētajā sarakstā:

```
procedure FindKey (var SOL: SelfOrgList; tkey: KeyType;
                  var found: boolean);
{Pašorganizētā sarakstā SOL^ meklē elementu, kura atslēgas
lauka
vērtība ir tkey. Ja meklēšana ir sekmīga, sameklētais elements
pavirzās
uz saraksta sākumu un kļūst par tekošo elementu}
var p: NodePointer;
    k: Count;
begin
    found:= false
    if not Empty(SOL) then with SOL^ do
        begin
            k:= 1; p:= head;
```

Pēc lietojuma biežuma sakārtoti saraksti (16)

(Frequency ordered lists)

```
while (p^.next <> head) and (p^.elem.el.key <> tkey) do  
  begin
```

```
    p:= p^.next;  k:= k+1
```

```
  end;
```

```
if p^.elem.el.key = tkey then
```

```
  begin
```

```
    found:= true;
```

```
    curent:= p;
```

```
    icurrent:= k;
```

```
    Swap(SOL)
```

{izsauc apmaiņas

```
operāciju}
```

```
  end
```

```
end
```

```
end;
```

Daudzkāršsaistītais saraksts (1)

(multilinked list)

Prasība, ka divkāršsaistītā sarakstā elementu otrā sasaiste ar rādītājiem *prior* ir loģiski pretēja elementu pirmajai sasaistei ar rādītājiem *next*, nav obligāta. Elementus sarakstā var sasaistīt ar rādītājiem, izmantojot jebkuru kārtošanas vai sasaistes principu. Pat tad, ja pēc būtības saraksts ir divkāršsaistīts, bet elementu divkāršā sasaiste nav loģiski pretēja, šādu sarakstu sauc par otrās kārtas daudzkāršsaistītu sarakstu (multilinked list of order two).

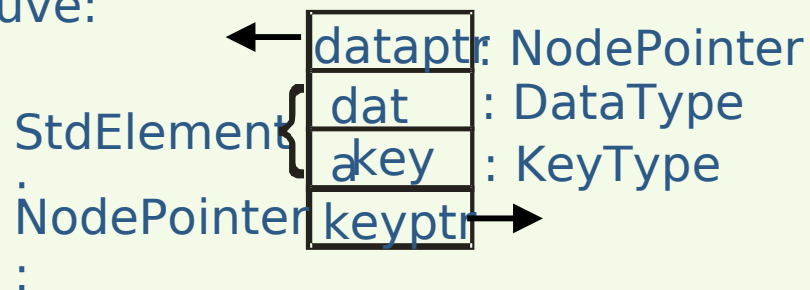
Apskatīsim sarakstu, kurā elementu datu laukā *data* izvietota kāda rakstzīme, bet atslēgas laukā *key* uzdoti veseli pozitīvi skaitļi. Katram elementam papildus pievienots rādītāja lauks *dataptr*, kas paredzēts, lai datu laukus sasaistītu alfabētiskā secībā, un rādītāja lauks *keyptr*, lai atslēgas laukus sasaistītu augošā secībā. Vienkāršības labad rādītāju lauki *next* un *prior* netiek apskatīti, bet tie ir iespējami.

Daudzkāršsaistītais saraksts (2)

(multilinked list)

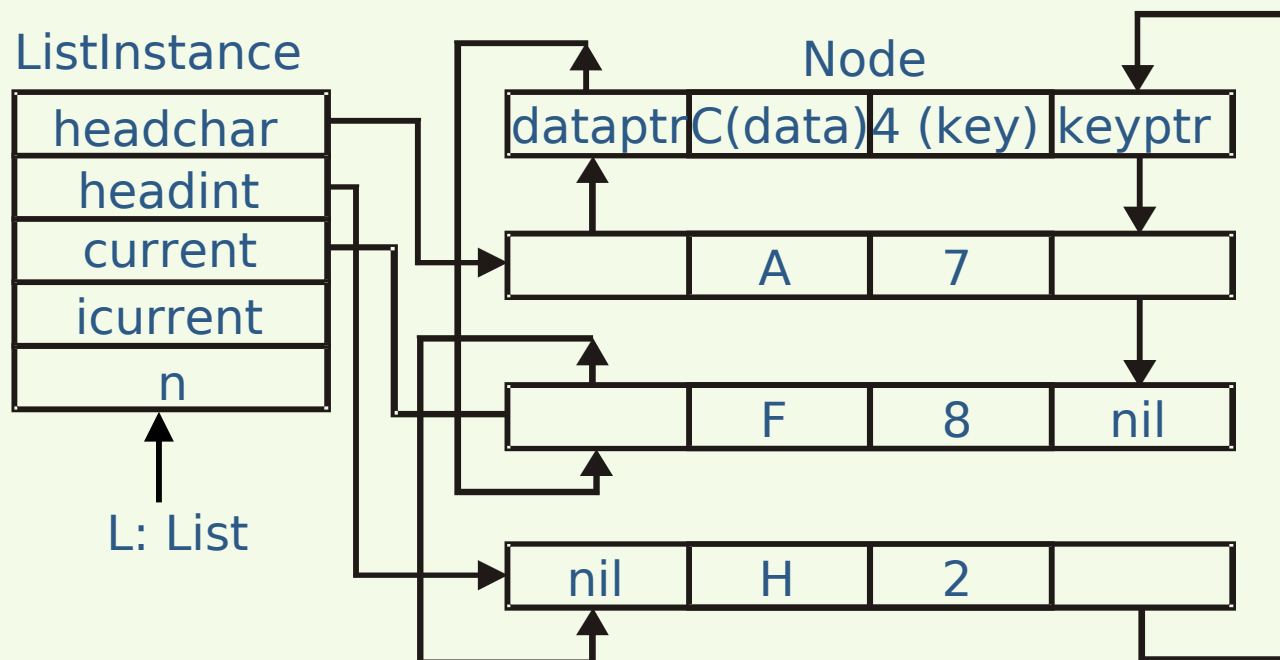
Daudzkāršsaistīta saraksta elementa

uzbūve:



Katram elementam varētu pievienot vēl divus rādītāju laukus: **next** un **prior**.

Otrās kārtas divkāršsaistītā saraksta piemērs:



Daudzkāršsaistītais saraksts (3)

(multilinked list)

Piemērā 3. elements ir tekošais, tātad $icurrent=3$. Sarakstā ir tikai 4 elementi. Jebkurā brīdī sarakstam var pievienot jaunus elementus vai tekošo elementu dzēst, attiecīgi pārkārtojot elementu sasaisti abos rādītāju kontūros un precizējot norādes. Saraksts nodrošina plašas elementu meklēšanas iespējas.

Otrās kārtas daudzkāršsaistītā saraksta elementu un attiecīgo lauku piekļuves nodrošināšanai paredzēti divi sākumrādītāji: *headint* un *headchar*. Rādītājs *headchar* norāda uz 1.elementu datu lauku sasaistē, bet rādītājs *headint* uz 1.elementu atslēgas lauku sasaistē. Abās sasaistēs pēdējā atslēgas lauka vērtība ir *nil*. Var paredzēt arī vēl attiecīgi divus beigu rādītājus *tailint* un *tailchar*. Tātad katram sasaistes kontūram obligāti jāparedz savs sākumrādītājs *head* un ieteicams arī attiecīga rādītāja *tail* lietojums. Var veidot arī cirkulāru sarakstu.

Strādājot ar daudzkāršsaistīto sarakstu, kāds no saraksta elementiem vienmēr ir tekošais elements un uz to norāda rādītājs *current*. Piemērā tekošais sarakstā ir trešais elements.

Steks (stack)

Steks ir lineāra datu struktūra, kas darbojas pēc principa **LIFO** (Last-in / First-out – Pēdējais iekšā / Pirmais ārā).

Steka virsotnē *top*, izpildot operāciju *Push*, nepārtraukti var pievienot jaunus elementus.

No steka var nolasīt tikai to elementu, kas pēdējais pievienots stekam. Šim nolūkam paredzēta operācija *Pop*. Pēc nolasīšanas elements stekā tiek dzēsts un iepriekšējais elements nokļūst steka virsotnē *top*.

Pamatoperācijas: Push, Pop.

Servisa operācijas: Size, Empty, Full, Create, Terminate

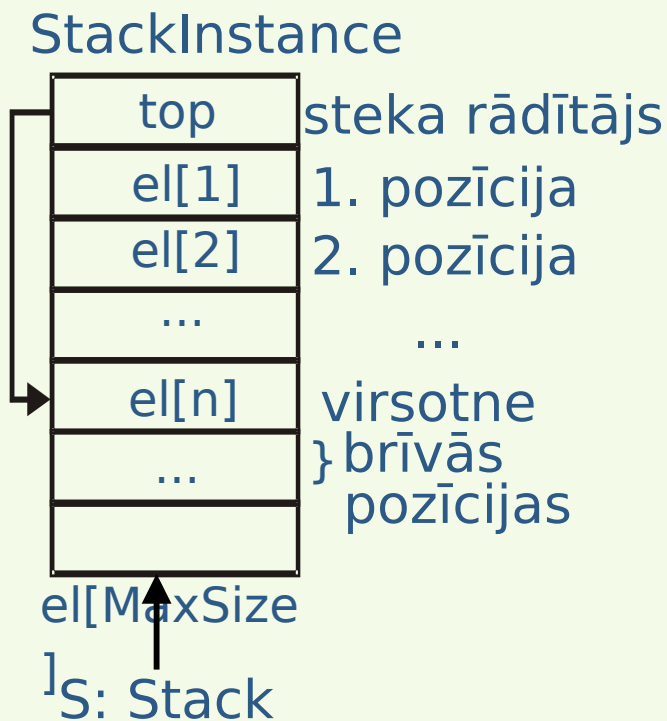
Papildoperācijas: Retrieve, Top, u.c.

Steka elementu tips: StdElement.

Steka tips: Stack.

Vektoriālajā formā attēlotais steks

(1)



```
const MaxSize = 500;    {uzdod lietotājs}
type Position = 1 .. MaxSize;
    Count = 0 .. MaxSize;
    DataType = string;  {uzdod lietotājs}
    KeyType = integer;
    StdElement = record
        data: DataType;
        key: KeyType
    end;
StackInstance = record
    top: Count;
    el: array [Position] of StdElement
end;
Stack = ^StackInstance;
```

Vektoriālajā formā attēlotais steks (2)

```
procedure Create (var S: Stack; var created: boolean);  
{Izveido jaunu tukšu steku S^}  
begin
```

```
    new(S);  
    S^.top:= 0;  
    created:= true;
```

```
end;
```

```
procedure Terminate (var S: Stack; var created: boolean);  
{Likvidē steku S^}  
begin
```

```
    if created then  
        begin  
            dispose (S);  
            created:= false  
        end
```

```
end;
```

Vektoriālajā formā attēlotais steks (3)

```
function Size (var S: Stack): Count;  
{Nosaka elementu skaitu stekā S^}  
begin  
    Size:= S^.top  
end;  
  
function Empty (S: Stack): boolean;  
{Pārbauda, vai steks S^ ir tukšs}  
begin  
    Empty:= S^.top = 0;  
end;  
  
function Full (S: Stack): boolean;  
{Pārbauda, vai steks S^ ir pilns}  
begin  
    Full:= S^.top = MaxSize  
end;
```

Vektoriālajā formā attēlotais steks

(4)

```
procedure Push (var S: Stack, e: StdElement);  
{Stekam S^ pievieno elementu e}  
begin  
    if not Full(S) then with S^ do  
        begin  
            top:= top + 1;  
            el[top]:= e  
        end  
    end;  
  
procedure Pop (var S: Stack; var e: StdElement);  
{No steka S^ nolasa elementu e}  
begin  
    if not Empty (S) then with S^ do  
        begin  
            e:= el[top];  
            top:= top - 1  
        end  
    end;  
  
end;
```

Vektoriālajā formā attēlotais steks (5)

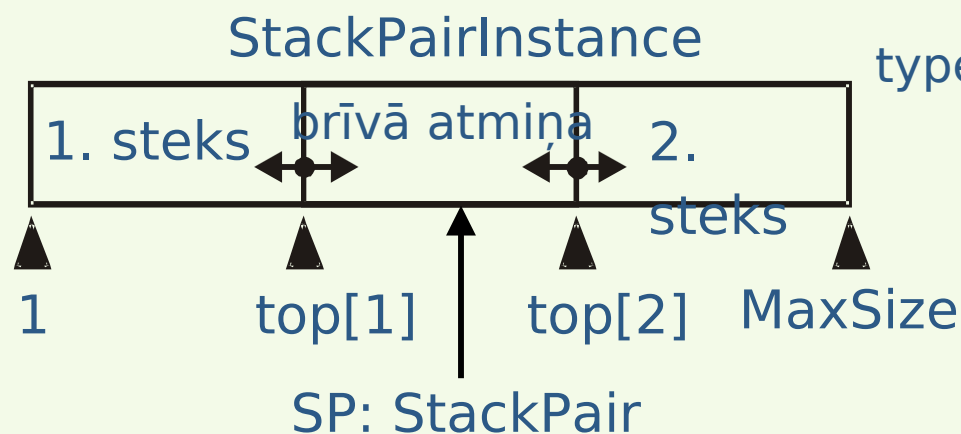
Vektoriālajā formā attēlotā steka liela priekšrocība ir tā, ka steka apstrādes operāciju algoritmi ir īsi un vienkārši.

Vektoriālajā formā attēlotā steka galvenais trūkums - visai neefektīva pamatatmiņas izmantošana. Maksimālais elementu skaits stekā iepriekš ir grūti prognozējams. Steka laukā ir jāparedz pietiekami liels brīvs atmiņas apgabals, lai nepieļautu steka pārpildi. Šo trūkumu var novērst, steka brīvajā atmiņas apgabalā organizējot vēl vienu steku, t.i., parastā steka vietā izveidojot steka pāri. Vienlaicīgi var strādāt ar abiem stekiem vai arī

tikai ar vienu no tiem.

Steka pāris (1)

(stack pair)



```

const MaxSize = 500;
      MaxPlusOne = MaxSize + 1;
type StackTop = 0.. MaxPlusOne;
      StackNo = 1.. 2; {steka numurs}
      Position = 1 .. MaxSize;
      DataType = string;
      KeyType = integer;
      StdElement = record
        data: DataType;
        key: KeyType
      end;
      StackPairInstance = record
        top: array[StackNo] of StackTop;
        el: array[Position] of StdElement
      end;
      StackPair = ^StackPairInstance;
  
```

Steka pāris (2)

(stack pair)

```
procedure Create (var SP: StackPair; var created: boolean);  
{Izveido jaunu tukšu steka pāri SP^}  
begin  
    new(SP);  
    SP^.top[1]:= 0;  
    SP^.top[2]:= MaxPlusOne;  
    created:= true  
end;  
  
procedure Terminate (var SP: StackPair; var created: boolean);  
{Likvidē steka pāri SP^}  
begin  
    if created then  
        begin  
            dispose (SP);  
            created:= false  
        end  
    end;  
end;
```

Steka pāris (3)

(stack pair)

```
function Size (SP: StackPair; SNo: StackNo): StackTop;  
  {Nosaka elementu skaitu steka pāra SP^ pirmajā vai otrajā  
stekā SNo}  
begin  
  case SNo of  
    1: Size:= SP^.top[1];  
    2: Size:= MaxSize - SP^.top[2] + 1  
  end  
end;  
  
function Full (SP: StackPair): boolean;  
  {Pārbauda, vai steka pāris SP^ ir pilns}  
begin  
  Full:= SP^.top[1] + 1 = SP^.top[2]  
end;
```

Steka pāris (4)

(stack pair)

```
function Empty(SP: StackPair; SNo: StackNo): boolean;  
{Pārbauda, vai steka pāri SP^ steks SNo ir tukšs}  
begin  
    case SNo of  
        1: Empty:= SP^.top[1] = 0;  
        2: Empty:= SP^.top[2] = MaxPlusOne  
    end  
end;  
  
procedure Push (var SP: StackPair; SNo: StackNo; e:  
StackElement);  
{Steka pāra SP^ stekā SNo izvieta jaunu elementu e}  
begin  
    if not Full(SP) then with SP^ do  
        begin  
            case SNo of  
                1: top[1]:= top[1] + 1;  
                2: top[2]:= top[2] - 1  
            end;  
            el[top[SNo]]:= e  
        end  
    end;  
end;
```

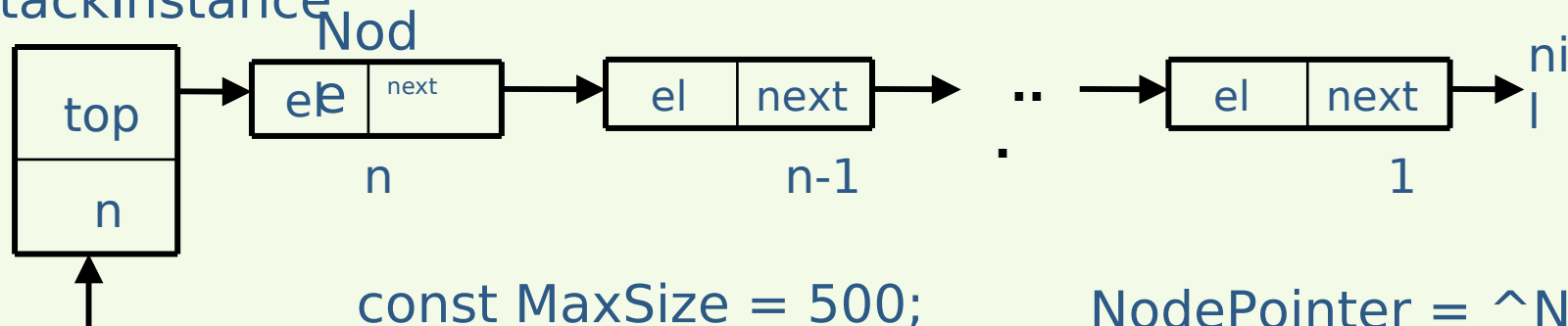
Steka pāris (5)

(stack pair)

```
procedure Pop (SP: StackPair; SNo: StackNo; var e:
StdElement);
{No steka pāra SP^ steka SNo nolasa elementu e}
begin
  if not Empty (SP, SNo) then with SP^ do
    begin
      e:= el[top[SNo]];
      case SNo of
        1: top[1]:= top[1] -1
        2: top[2]:= top[2] +1
      end
    end
  end
end;
```

Saistītajā formā attēlotais steks (1)

StackInstance



S: Stack

```

const MaxSize = 500;
type Count = 0 .. MaxSize;
DataType = string;
KeyType = integer;
StdElement = record
    data: DataType;
    key: KeyType
end;
Node = record
    el: StdElement;
    next: NodePointer
end;

```

```

NodePointer = ^Node;
StackInstance = record
    top: NodePointer;
    n: Count
end;
Stack = ^StackInstance;

```

Saistītajā formā attēlotais steks (2)

```
procedure Create (var S: Stack; var created: boolean);  
{Izveido jaunu tukšu steku S^}  
begin  
    new(S);  
    S^.top:= nil;  
    S^.n:= 0;  
    created:= true  
end;
```

```
function Size (S: Stack): Count;  
{Nosaka elementu skaitu stekā S^}  
begin  
    Size:= S^.n  
end;
```

Saistītajā formā attēlotas steks (3)

```
procedure Terminate (var S: Stack; var created: boolean);  
{Likvidē steku S^}  
var p: NodePointer;  
begin  
    if created then with S^ do  
        begin  
            if not Empty(S) then  
                while top <> nil do  
                    begin  
                        p:= top; top:= top^.next;  
                        dispose (p)  
                    end;  
            dispose (S);  
            created:= false  
        end  
    end;  
end;
```


Saistītajā formā attēlotais steks (4)

```
function Empty (S: Stack): boolean;  
  {Pārbauda, vai steks S^ ir tukšs}  
begin  
  Empty:= S^.top = nil  
= 0} {Empty:= S^.n  
end;
```

```
function Full (S: Stack): boolean;  
  {Pārbauda, vai steks S^ ir pilns}  
begin  
  Full:= S^.n = MaxSize  
end;
```

Saistītajā formā attēlotais steks (5)

```

procedure Push (var S: Stack; e: StdElement);
{Stekā S^ izvieta jaunu elementu e}
var p: NodePointer;
begin
    if not Full(S) then with S^ do
        begin
            new(p);  p^.el:= e;
            if Empty(S) then {steks ir
tukšs}
                begin
                    top:= p;  p^.next:= nil
                end
            else {steks nav
tukšs}
                begin {izkārtoti
saītes}
                    p^.next:= top;  top:= p
                end;
            n:= n + 1;
        end
    end

```

Saistītā formā attēlots steks (6)

```
procedure Pop (var S:Stack; var e: StdElement);  
{No steka S^ nolasa elementu e}  
var p: NodePointer;  
begin  
    if not Empty(S) then with S^ do  
        begin  
            e:= top^.el;  
            p:= top;  
            top:= top^.next;  
            dispose (p);  
            n:= n -1  
        end  
    end  
end
```

Rinda (Queue)

Rinda ir lineāra datu struktūra, kas strādā pēc principa:

FIFO – First-In / First-Out (Pirmais iekšā / Pirmais ārā) vai

HPIFO – High Priority-In / First-Out.

Parastajā rindā elementu apstrāde atkarīga no pievienošanas laika, prioritātes rindā – no elementa prioritātes lieluma.

Rindu nepārtraukti var papildināt ar jauniem elementiem. Jaunu elementu pievieno rindas beigās (vai meklē elementam vietu prioritātes rindas gadījumā).

No rindas nolasa to elementu, kas atrodas rindas sākumā. Pēc nolasīšanas elements rindā tiek dzēsts.

Servisa operācijas: Create, Terminate, Size, Empty, Full.

Pamatoperācijas: Enqueue, Serve (Dequeue).

Rindas elementa tips: StdElement.

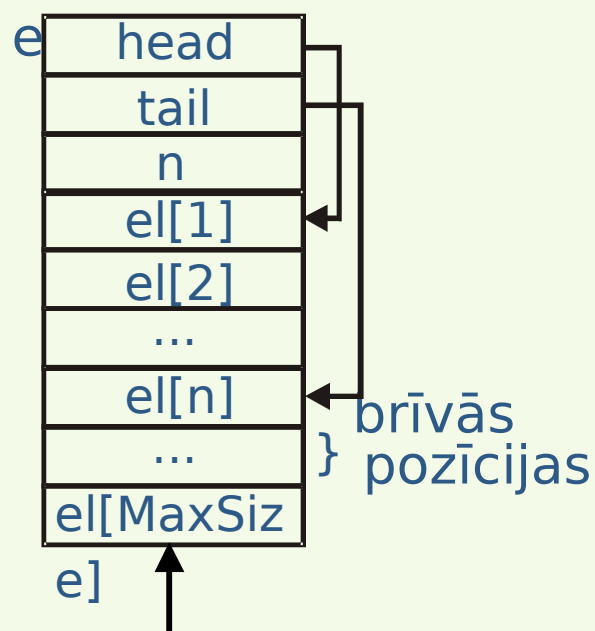
Rindas tips: Queue.

Rādītāji: head – norāda uz rindas sākumu (uz 1. elementu),
tail – norāda uz rindas beigām (uz pēdējo elementu).

Katram rindas elementam, izņemot pēdējo, ir pēctecis, kas kļūst par
apkalpojamo rindā, ja tiek izpildīta operācija Serve.

Vektoriālajā formā attēlotā rinda (1)

QueueInstance



Q: Queue

```

const MaxSize = 500;      {uzdod
lietotājs}
type Position = 1..MaxSize;
    Count = 0..MaxSize;
    DataType = string;    {uzdod
lietotājs}
    KeyType = integer;
    StdElement = record
        data: DataType;
        key: KeyType
    end;
QueueInstance = record
    head, tail, n: Count;
    el: array [Position] of
        StdElement
end;
Queue = ^QueueInstance;
  
```

Vektoriālajā formā attēlotā rinda (2)

```
procedure Create (var Q: Queue; var created: boolean);  
{Izveido jaunu tukšu rindu Q^}  
begin  
    new (Q);  
    Q^.head:= 0;    Q^.tail:= 0;  
    Q^.n:= 0;  
    created:= true;  
end;
```

```
procedure Terminate (var Q: Queue; var created:boolean);  
{Likvidē rindu Q^}  
begin  
    if created then  
        begin  
            dispose (Q);  
            created:= false  
        end  
    end;  
end;
```

Vektoriālajā formā attēlotā rinda (3)

```
function Size (Q: Queue): Count;  
{Nosaka elementu skaitu rindā Q^}  
begin  
    Size:= Q^.n  
end;  
  
function Empty (Q: Queue): boolean;  
{Pārbauda, vai rinda Q^ ir tukša}  
begin  
    Empty:= Q^.n = 0  
= 0} {Q^.head  
end;  
  
function Full (Q: Queue): boolean;  
{Pārbauda, vai rinda Q^ ir pilna}  
begin  
    Full:= Q^.n = MaxSize  
end;
```

Vektoriālajā formā attēlotā rinda

(4)

```

procedure Enqueue (var Q: Queue; e: StdElement);
{Rindā Q^ izvieta jaunu elementu e}
begin
    if not Full(Q) then with Q^ do
        begin
            tail:= tail+1;           {jaunā elementa
pozīcija tail}
            el[tail]:= e;           {elementu e izvieta
rindā}

            if Empty(Q) then head:= 1;
            n:= n+1
        end
    end;
procedure Serve (var Q: Queue; var e: StdElement);
{No rindas Q^ nolasa elementu e}
begin
    if not Empty(Q) then with Q^ do
        begin
            e:= el[head];    n:= n-1;           {nolasa
elementu}
            if n > 0 then head:= head+1;       {nolasīto
elementu dzēš}
        end
    end;

```


Vektoriālā formā attēlota rinda (5)

Operācijas $\text{Serve}(Q, e)$ algoritma priekšrocība ir tā vienkāršība un efektivitāte. Būtisks trūkums ir tas, ka vektorā veidojas brīvas pozīcijas rindas sākumā. Ir divi paņēmieni šī trūkuma novēršanai:

1) nolasīto elementu rindā dzēst, visus elementus, sākot ar otro, pārsūtot

pa vienu pozīciju virzienā un rindas sākumu:

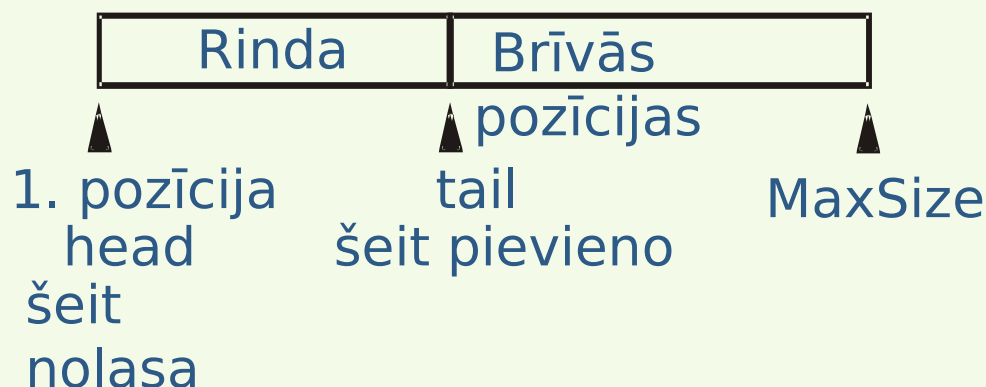
```

procedure Serve1 (var Q: Queue; var e: StdElement);
{No rindas Q^ nolasa elementu e}
var i: Position;
begin
    if not Empty(Q) then with Q^ do
        begin
            e := el[head];                                {nolasa
elementu}
            for i := 2 to n do                             {nolasīto elementu
dzēš}
                el[i-1] := el[i];
            n := n - 1;   tail := n;
            if tail = 0 then head := 0                    {ja rinda kļuvusi
tukša}
        end
    end

```

Cirkulārā rinda (1)

Rinda pēc tās izveidošanas:



Var atzīmēt, ka

1) ja strādā ar rindu, tā vektorā virzās no zemākas pozīcijas

uz augstāku pozīciju;

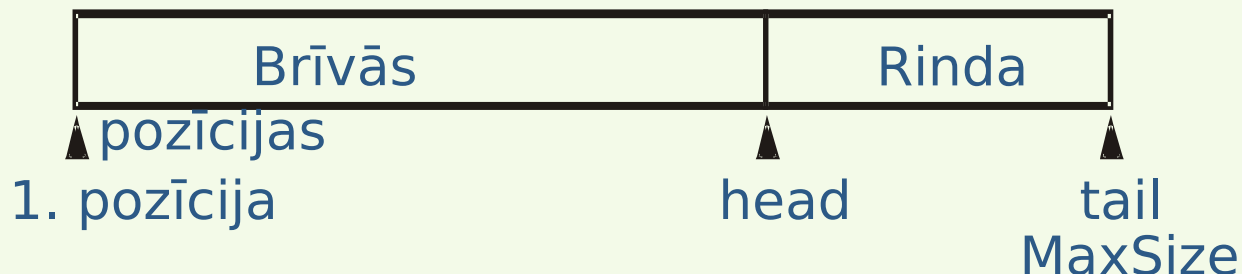
2) rindai virzoties uz priekšu, tās paplašināšanās vai samazināšanās ir atkarīga no operāciju Enqueue un Serve izpildes skaita un secības.

Cirkulārā rinda (2)

Rinda pēc kāda laika, kad operācija Serve izpildīta daudz biežāk nekā operācija Enqueue:



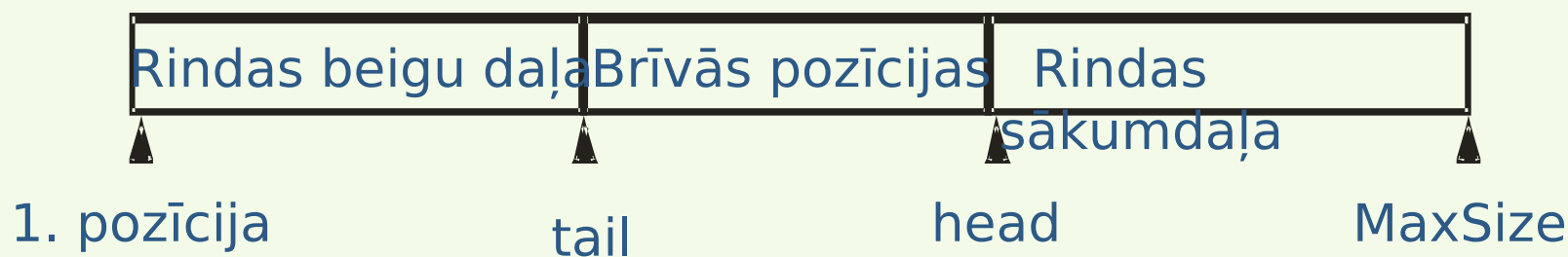
Situācija, kad rinda aizvirzījies līdz vektora beigām – rinda ir pilna, bet faktiski brīvā atmiņa izveidojusies vektora sākumā:



Cirkulārā rinda (3)

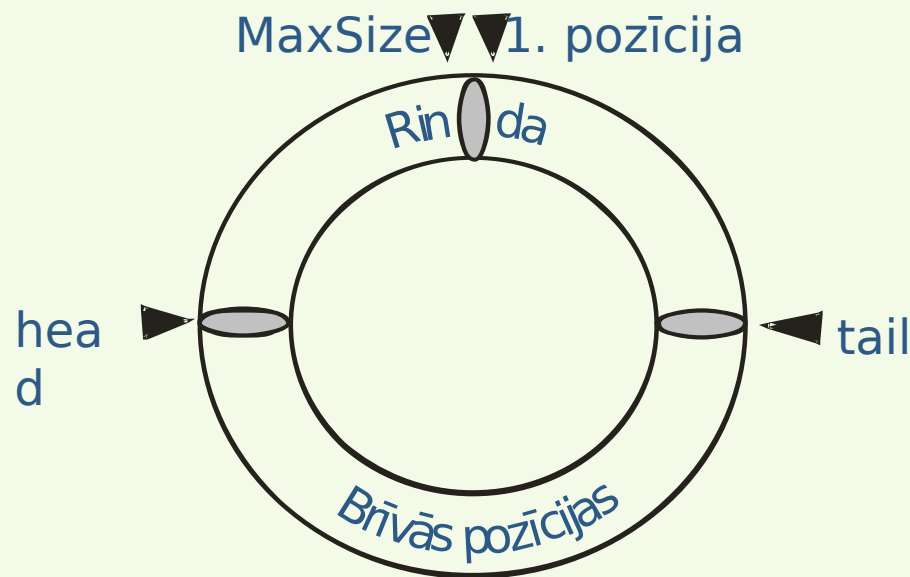
Rindas pārpildes novēršana:

- 1) rindas pārbīde uz vektora sākumu. Tā ir $O(n)$ kārtas operācija. Neefektīvs paņēmieni, pārbīde var būt bieži nepieciešama;
- 2) veidot cirkulāru rindu, kurā rindas gals pārvietojas uz vektora sākumu (tail atrodas zemāko pozīciju rajonā):



Cirkulārā rinda (4)

Šai gadījumā rindai iedalītais atmiņas apgabals kļūst cirkulārs – rinda pārvietojas pa noslēgtu cirkulāru atmiņas apgabalu:



Cirkulārai rindai ir tāds pats modelis un modeļa apraksts kā parastai vektoriālā formā attēlotajai rindai.

Rindas cirkulārumu realizē nevis ar rindas aprakstu, bet gan ar operācijām Create, Enqueue, Serve.

Pārējās operācijas ir tādas pašas kā parastai vektoriālā formā attēlotajai rindai.

Cirkulāra rinda (5)

```
procedure Create (var Q: Queue; var created: boolean);  
{izveido jaunu tukšu cirkulāru rindu Q^}
```

```
begin  
  new (Q);  
  with Q^ do  
    begin  
      head:= 1; tail:= MaxSize; n:= 0  
    end;  
  created:= true  
end;
```

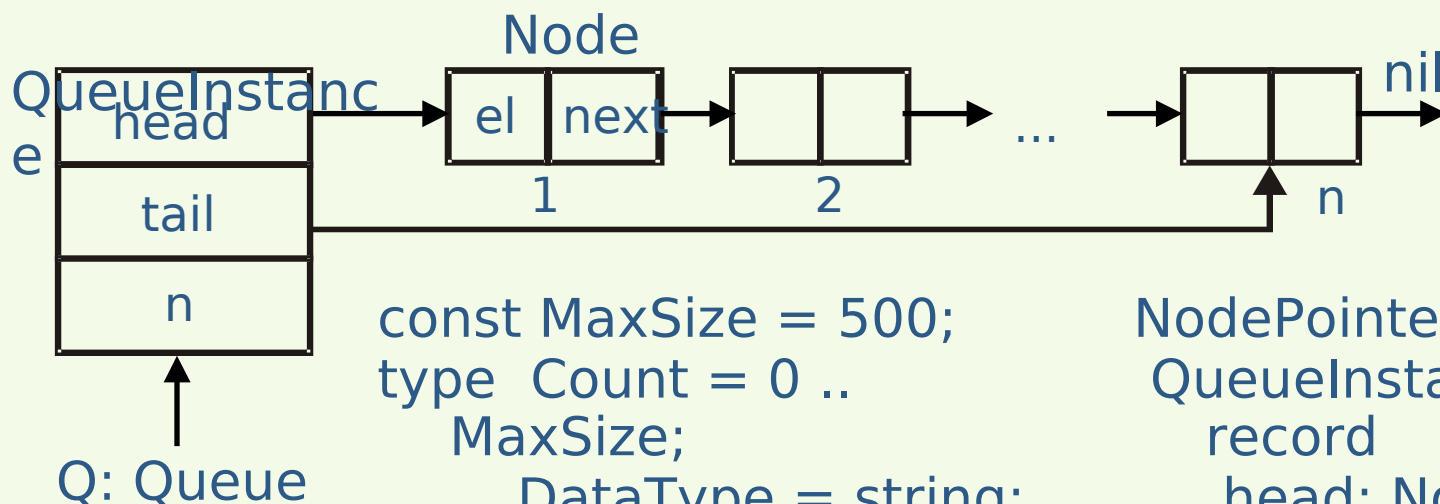
```
procedure Enqueue (var Q: Queue; e: StdElement);  
{Cirkulārā rindā Q^ izvieta jaunu elementu e}
```

```
begin  
  if not Full(Q) then with Q^ do  
    begin  
      tail:= (tail mod MaxSize) + 1;  
      {pozīcija tail}  
      el[tail]:= e; {izvieta elementu  
      pozīcijā tail}  
      if Empty(Q) then head:= 1;  
      n:= n + 1  
    end  
  end;  
end;
```

Cirkulāra rinda (6)

```
procedure Serve (var Q: Queue; var e: StdElement);  
{No cirkulāras rindas Q^ nolasa elementu e}  
begin  
    if not Empty(Q) then with Q^ do  
        begin  
            e:= el[head];                {nolasa  
elementu}  
            head:= (head mod MaxSize) +1;    {pozīcija  
head}  
            n:= n -1;  
            if Empty(Q) then tail:= MaxSize  
        end  
    end;  
end;
```

Saistītajā formā attēlotā rinda (1)



```

const MaxSize = 500;
type Count = 0 ..
    MaxSize;
    DataType = string;
    KeyType = integer;
    StdElement = record
        data: DataType;
        key: KeyType
    end;
    Node = record
        el: StdElement;
        next: NodePointer
    end;
end;

```

```

NodePointer = ^Node;
QueueInstance =
    record
        head: NodePointer;
        tail: NodePointer;
        n: Count
    end;
Queue =
    ^QueueInstance;
end;

```

end;

Saistītajā formā attēlotā rinda (2)

```
procedure Create (var Q: Queue; var created: boolean);  
{Izveido jaunu tukšu rindu Q^}  
begin  
    new(Q);  
    with Q^ do  
        begin  
            head:= nil;    tail:= nil;    n:=0  
        end;  
    created:= true  
end;  
  
function Size (var Q: Queue): Count;  
{Nosaka elementu skaitu rindā Q^}  
begin  
    Size:= Q^.n  
end;
```

Saistītajā formā attēlotā rinda (3)

```
procedure Terminate (var Q: Queue; var created: boolean);
{Likvidē rindu Q^}
var p: NodePointer;
begin
    if created then
        begin
            if not Empty(Q) then with Q^ do
                while head <> nil do
                    begin
                        p:= head;
                        head:= head.^next;
                        dispose (p)
                    end;
                dispose(Q); created:= false
            end
        end
    end;
```

Saistītā formā attēlota rinda (4)

```
function Empty (Q: Queue): boolean;  
  {Pārbauda, vai rinda Q^ ir tukša}  
begin  
  Empty:= Q^.n = 0;                                {Q^.head =  
nil}  
end;  
  
function Full (Q: Queue): boolean;  
  {Pārbauda, vai rinda Q^ ir pilna}  
begin  
  Full:= Q^.n = MaxSize  
end;
```

Saistītajā formā attēlotā rinda (5)

```

procedure Enqueue (var Q: Queue; e: StdElement);
{Rindā Q^ izvieta jaunu elementu e}
var p: NodePointer;
begin
    if not Full(Q) then with Q^ do
        begin
            new(p);  p^.el:= e;
            if Empty(Q) then                                {rinda ir
tukša}
                begin
                    head:= p;  tail:= p;  p^.next:= nil
                end
            else                                            {rinda nav
tukša}
                begin                                     {izkārto 2
saites}
                    p^.next:= tail^.next;
                    tail^.next:= p;  tail:=p
                end;
            end;
        end;
    end;
end;

```

Saistītajā formā attēlotā rinda (6)

```

procedure Serve (var Q: Queue; var e: StdElement);
{No rindas Q^ nolasa elementu e}
var p: NodePointer;
begin
    if not Empty(Q) then with Q^ do
        begin
            p:= head;
            e:= head^.el;           {nolasa
elementu}
            head:= head^.next;     {pārstata radītāju
head}
            if n = 1 then tail:= nil;
            dispose (p);           {dzēš nolasīto
elementu}
            n:= n - 1;
        end
    end
end

```

Prioritātes rinda

Darbības princips: **HPIFO** - High Priority In / First Out (ar Augstu prioritāti iekšā / Pirmais ārā).

Prioritātes rindas galvenokārt izmanto notikumu reģistrēšanas un apkalpošanas sistēmās.

Vektoriālajā formā attēlotā prioritātes rinda (1)

Elementa uzbūve:

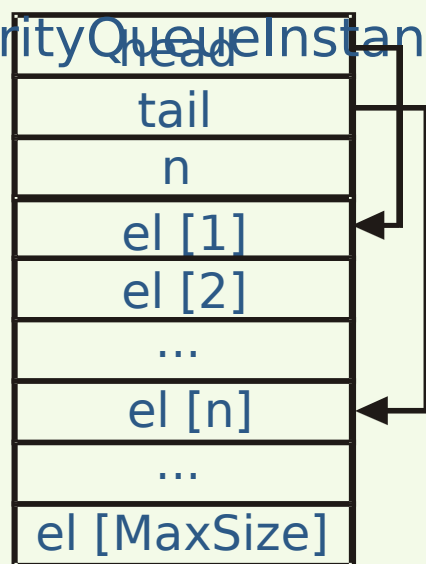
$el[i] \Rightarrow$
 $i = 1, 2, \dots, n$

data	: DataType
ke	: KeyType
prty	: Priority

```
const MaxSize = 500;
type Position = 1 .. MaxSize;
Count = 0 .. MaxSize;
DataType = string;
KeyType = integer;
Priority: integer;

StdElement = record
  data: DataType;
  key: KeyType;
  prty: Priority
end;
```

PriorityQueueInstance



```
PriorityQueueInstance = record
  head, tail, n: Count;
  el: array[Position] of
    StdElement
end;
```

PQ: PriorityQueueInstance

Vektoriālajā formā attēlotā prioritātes rinda (2)

```

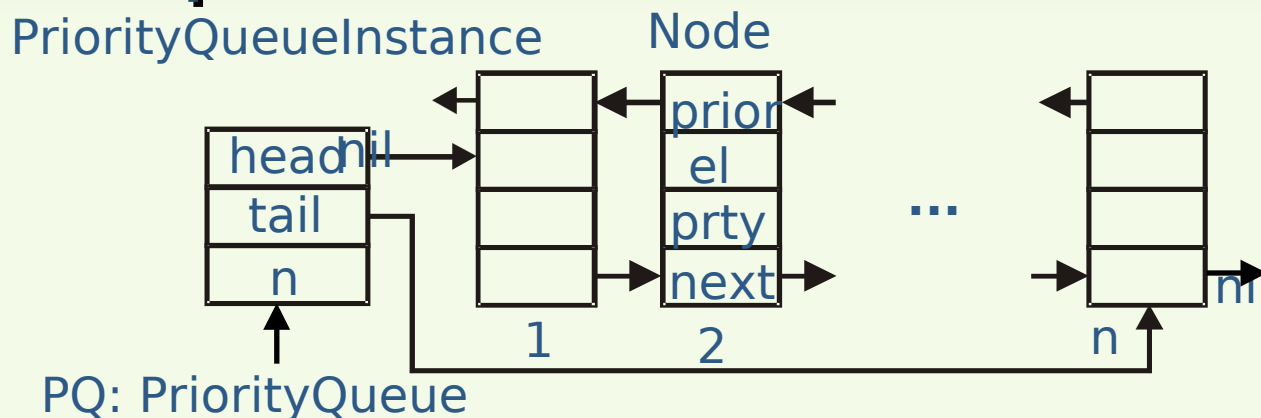
procedure Enqueue (var PQ: PriorityQueue; e: StdElement);
{Prioritātes rindā PQ^ izvieto jaunu elementu e}
var i, k: Position;
begin
    if not Full(PQ) then with PQ^ do
        begin
            if Empty(PQ) then {prioritātes rinda ir
tukša}
                begin
                    head:= 1; el[head]:= e; tail:= head
                end
            else {elementam meklē
vietu}
                for i:= n downto 1 do
                    if el[i].prty > e.prty then
                        begin {izvieto aiz i-tā
elementa}
                            for k:= n downto i + 1 do
                                el[k+1]:= el[k];
                            el[i+1]:= e; break
                        end;
                    if i = n then tail:= tail + 1;
                    n:= n + 1;
                end
            end
        end
    end
end

```


Vektoriālā formā attēlota prioritātes rinda (3)

```
procedure Serve (var PQ: PriorityQueue; var e: StdElement);  
{No prioritātes rindas PQ^ nolasa elementu e}  
var i: Position;  
begin  
    if not Empty(PQ) then with PQ^ do  
        begin  
            e:= el[head];           {nolasa  
elementu}  
            for i:= 2 to n do el[i-1]:= el[i]; {nolasīto elementu  
dzēš}  
            n:= n -1;  
            tail:= n;  
            if tail = 0 then head:= 0  
        end  
    end;  
end;
```

Saistītajā formā attēlotā prioritātes rinda (1)



```

const MaxSize = 500;
type Position = 1 .. MaxSize;
Count = 0 .. MaxSize;
DataType = string;
KeyType = integer;
Priority: integer;
StdElement = record
  data: DataType;
  key: KeyType;
end;

```

```

Node = record
  el: StdElement;
  next: NodePointer;
  prior: NodePointer;
  prty: Priority;
end;
NodePointer = ^Node;
PriorityQueueInstance = record
  head, tail: NodePointer;
  n: Count
end;
PriorityQueue =
  PriorityQueueInstance;

```

Saistītajā formā attēlotā prioritātes rinda (2)

```

procedure Enqueue(var PQ: PriorityQueue; pr: Priority; e: StdElement);
{Prioritātes rindā PQ^ izvieta jaunu elementu e}
var p, q: NodePointer;
begin

```

```

    if not Full(PQ) then with PQ^ do
    begin

```

```

        new(p); p^.el:= e; p^.prty:= pr;
        if Empty(PQ) then

```

{prioritātes rinda ir

tukša}

```

            begin

```

```

                head:= p; tail:=p;

```

```

                p^.next:= nil; p^.prior:= nil

```

```

            end

```

```

        else

```

{elementam meklē

vietu}

```

            if head^.prty < pr then

```

{izvieta pirms

1.elementa}

```

                begin

```

```

                    p^.next:= head; p^.prior:= nil;

```

```

                    head^.prior:= p; head:= p

```

Saistītā formā attēlota prioritātes rinda (3)

```
else
  begin
    elementa} {izvieto vidusposmā aiz pirmā
    q:= tail;
    while (q^.prty < pr) and (q^.prior <> nil) do
      q:= q^.prior;
    p^.next:= q^.next;
    p^.prior:= q;
    if q <> tail then q^.next^.prior:= p
      else tail:= p;
    q^.next:= p
  end;
  n:= n +1
end
end;
```

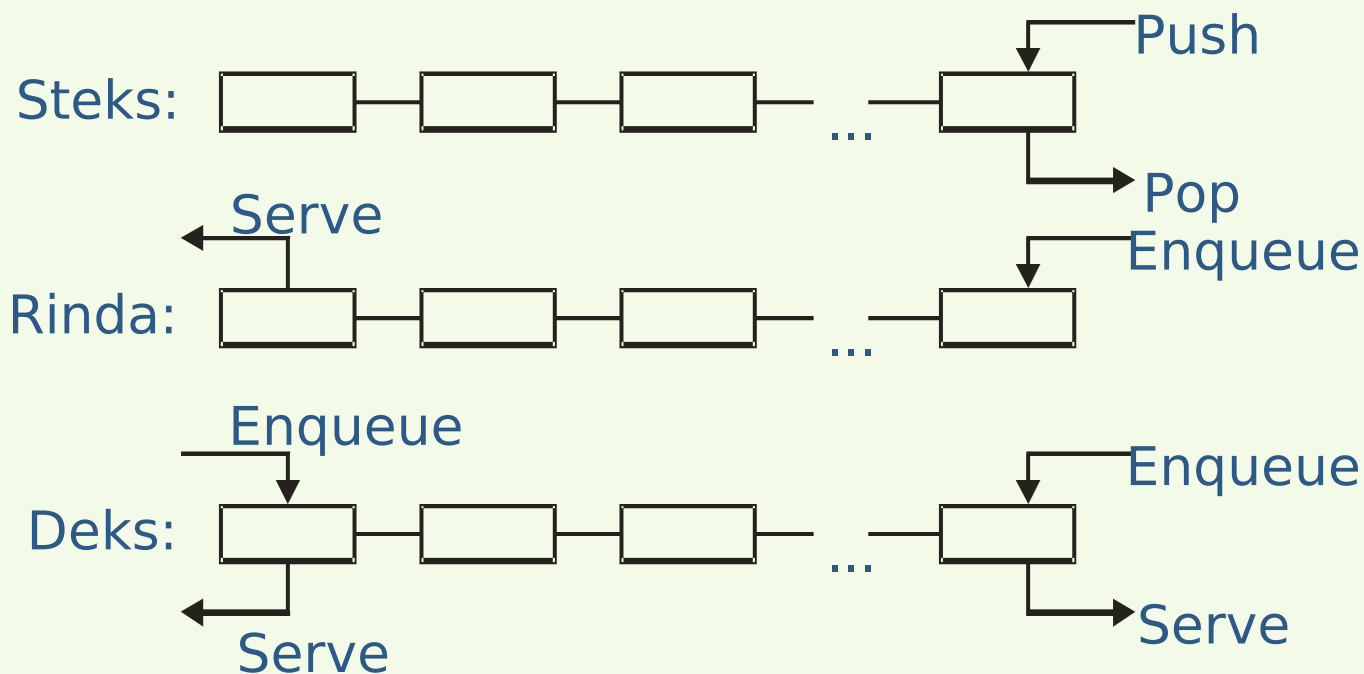
Saistītajā formā attēlotā prioritātes rinda (4)

```

procedure Serve (var PQ: PriorityQueue; var pr: Priority;
                 var e: StdElement);
{No prioritātes rindas PQ^ nolasa elementu e}
var p: NodePointer;
begin
  if not Empty(PQ) then with PQ^ do
    begin
      p:= head;
      e:= head^.el;  pr:= head^.prty;           {nolasa
elementu}
      if n = 1 then                               {prioritātes rinda kļūs
tukša}
        begin
          head:= nil;  tail:= nil
        end
      else                                           {prioritātes rindā ir vairāki
elementi}
        begin                                       {nolasīto elementu
dzēš}
          head^.next^.prior:= nil;
          head:= head^.next
        end;
      dispose(p);
      n:= n-1;
    end
  end

```

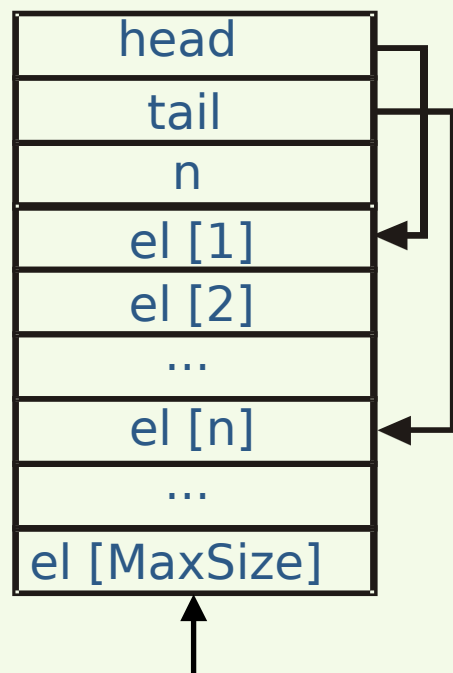
Deks – rinda ar diviem galiem



Svarīgs parametrs - deka gala numurs

Vektoriālajā formā attēlotais deks (1)

DequeueInstance



D: Dequeue

```
const MaxSize = 500;
type Position = 1 .. MaxSize;
Count = 0 .. MaxSize;
DataType = string;
KeyType = integer;
```

```
No= 1 .. 2;
```

```
StdElement = record
  data: DataType;
  key: KeyType;
  prty: Priority
end;
```

```
DequeueInstance = record
  head, tail: Count;
  n: Count
end;
```

```
Dequeue= ^ DequeueInstance
```

Vektoriālajā formā attēlotais deks

(2)

```

procedure Enqueue (var D: Dequeue; e: StdElement; DNo: No);
{Dekā D^ izvieta jaunu elementu e}
var i: Position;
begin
    if not Full(D) then with D^ do
        if Empty(D) then
            {vienalga, kurā galā
            izvietot}
            begin
                head:= 1;  tail:= 1;
                el[head]:= e;  n:= n+1
            end
        else case DNo of
            1: begin
                {elementu izvieta deka
                sākumā}
                for i:= tail downto head do el[i+1]:= el[i];
                el[head]:= e;  tail:= n;
                n:= n + 1;
                end;
            2: begin
                { elementu izvieta deka
                galā}
                n:= n + 1;
                tail:= n;  el[tail]:= e
            end
        end
    end
end

```


Vektoriālajā formā attēlotais deks (3)

```

procedure Serve (var D: Dequeue; var e: StdElement; DNo: No);
{No deka D^ nolasa elementu e}
var i: Position;
begin
    if not Empty(D) then with D^ do
        if n = 1 then
            elements}
            begin
                e:= el[head];
            elementu}
            head:= 0; tail:= 0; n:=0
            tukšs}
        end
    else case DNo of
        1: begin
            sākumā}
            e:= el[head];
            for i:= 2 to n do el[i-1]: = el[i];
            elementu dzēs}
            n:= n -1; tail:= n
        end;
        2: begin
            gala}
            e:= el[tail];
            for i:= 1 to n-1 do el[i]: = el[i+1];
            elementu dzēs}
            n:= n -1; head:= n
        end;
    end;
end;

```

{ja dekā tikai 1 elementu nolasa}

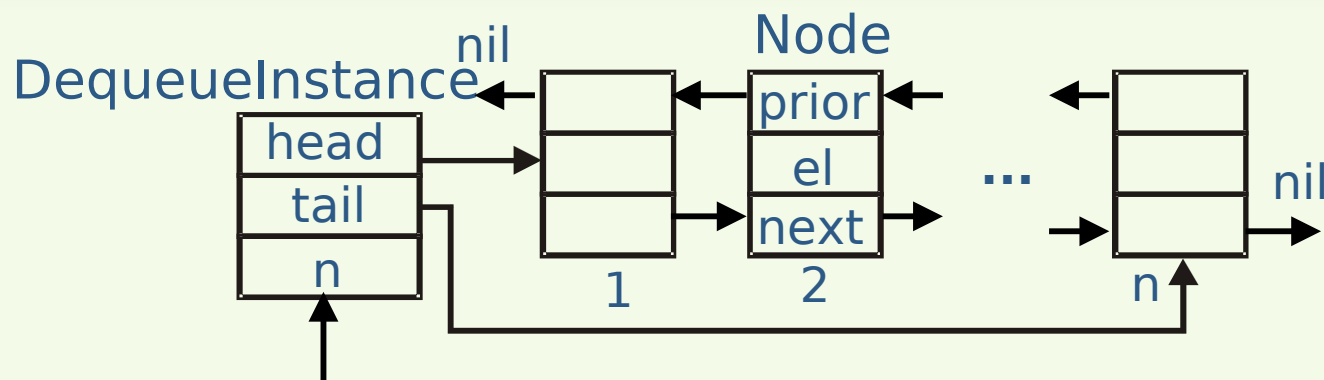
{deks kļūst tukšs}

{elementu nolasa deka sākumā}

{nolasīto elementu dzēs}

{elementu nolasa deka galā}

Saistītajā formā attēlotais deks (1)



D: Dequeue

```

const MaxSize = 500;
type Count = 0 .. MaxSize;
DataType = string;
KeyType = integer;
No = 1 .. 2;
StdElement = record
  data: DataType;
  key: KeyType;
end;

```

```

Node = record
  el: StdElement;
  next: NodePointer;
  prior: NodePointer
end;
NodePointer = ^Node;
DequeueInstance = record
  head: NodePointer;
  tail: NodePointer;
  n: Count
end;
Dequeue =
  ^DequeueInstance;

```

Saistītā formā attēlots deks (2)

```

procedure Enqueue (var D: Dequeue; e: StdElement; DNo: No);
{Dekā D^ izvieta jaunu elementu e}
var p: NodePointer;
begin
  if not Full(D) then with D^ do
    begin
      new (p);  p^.el:= e;
      if Empty(D) then                                {vienalga, kurā galā
        begin
          p^.next:= nil;  p^.prior:= nil;  head:= p;  tail:=p
        end
      else
        begin
          case DNo of
            1: begin                                     {elementu izvieta deka
                sākumā}
                  p^.prior:= nil;  p^.next:= head;
                  head^.prior:= p;  head:=p
                end;
            2: begin                                     {elementu izvieta deka
                galā}
                  p^.next:= nil;  p^.prior:= tail;
                  tail^.next:= p;  tail:= p
                end
            end
          end
        end
      end
    end
  end
end

```

Saistītā formā attēlots deks (3)

```

procedure Serve (var D: Deque; var e: StdElement; DNo: No);
{No deka D^ nolasa elementu e}
var p: NodePointer;
begin
  if not Empty(D) then with D^ do
    begin
      if n = 1 then                                     {deks kļūs
tukšs}
        begin
          p:= head; head:= nil; tail:= nil
        end
      else case DNo of
        1: begin                                         {elements tiks nolasīts deka
sākumā}
          p:= head;                                     {izkārto
saites}
          head^.next^.prior:= nil; head:= head^.next
        end;
        2: begin                                         {elements tiks nolasīts deka
galā}
          p:= tail;                                     {izkārto saites}
          tail^.prior^.next:= nil; tail:= tail^.prior
        end end;
      e:= p.prior;
    end
  end
  {nolasa
elementu}

```

Koks (tree) jeb hierarhiskā datu struktūra (1)

Koks ir nelineāra struktūra, kurā elementu sasaistes raksturs ir **viens-ar-vairākiem** (one-to-many).

Elementu tips: StdElement

Termini:

kreisais bērns – left child

labais bērns – right child

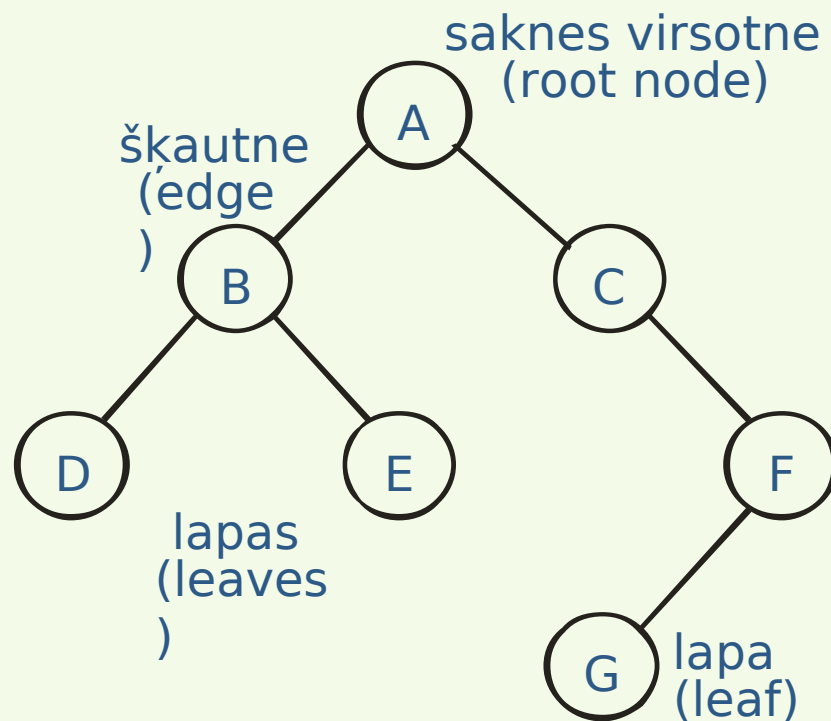
vecāks – parent

brāļi – siblings

sencis – ancestor

pēcnācējs – descendant

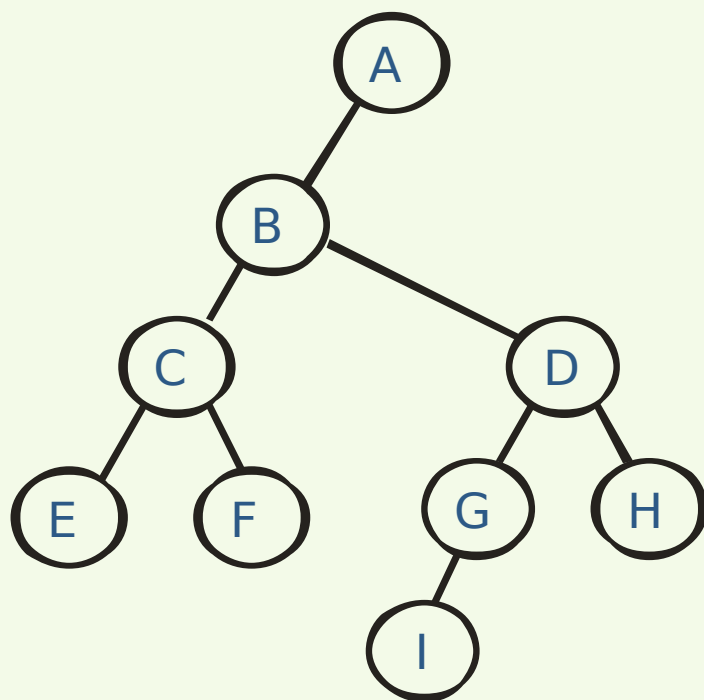
vienkāršais ceļš – simple path



Koks (tree) jeb hierarhiskā datu struktūra (2)

Ja virsotņu skaits ir n , tad šķautņu skaits $q = n - 1$.

Ceļa garums (path length) katram vienkāršajam ceļam ir vienāds ar virsotņu skaitu šai ceļā.



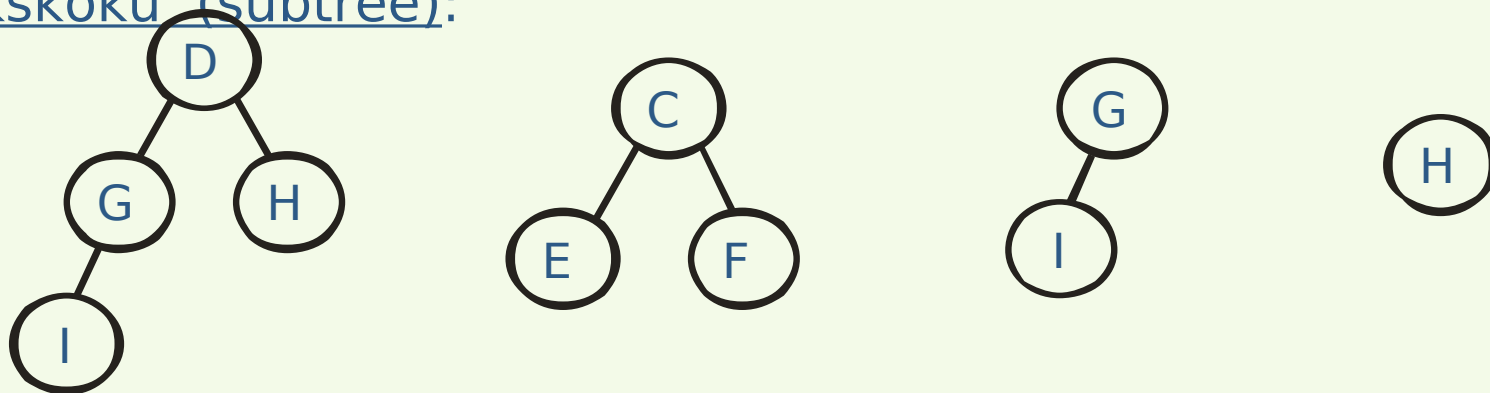
vienkāršais ceļš: A-B-C-E,
tā garums ir 4;

vienkāršais ceļš: A-B-D-G-I,
tā garums ir 5;

vienkāršais ceļš: G-I,
tā garums ir 2.

Koks (tree) jeb hierarhiskā datu struktūra (3)

Kāda virsotne kopā ar visiem tās pēcnācējiem veido apakškoku (subtree):



Viena virsotne kokā vienmēr ir tekošā (current).

Koku klasifikācija:

1) binārie koki:

a) binārās meklēšanas koki;

b) sabalansētie koki (AVL koki, sarkanmelnie koki

u.c.);

c) kaudzes (heap).

2) B koki

Koks (tree) jeb hierarhiskā datu struktūra (4)

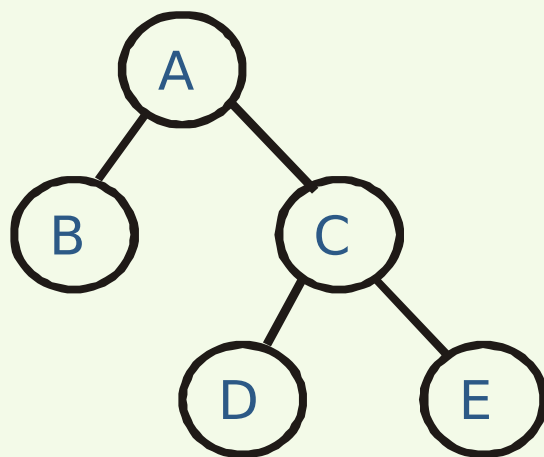
Koka raksturlielumi:

- 1) virsoņu skaits (size);
- 2) virsošnes pakāpe (height) ir tās pēcteču skaits;
- 3) virsošnes līmenis jeb dziļums (level). Saknes līmenis

ir 1,

katra bērna līmenis ir par 1 lielāks kā vecāka līmenis;

- 4) vidējā ceļa garums (avePath).

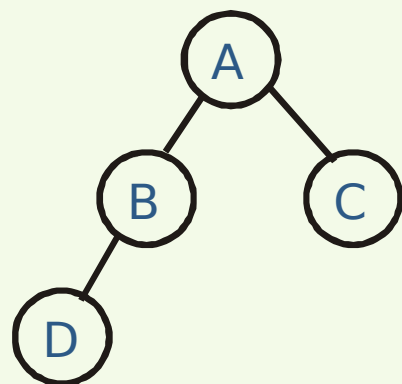


Virsošnes A pakāpe $\Rightarrow 3+1=4$

Virsošnes B pakāpe $\Rightarrow 0$

Virsošnes C pakāpe $\Rightarrow 2$

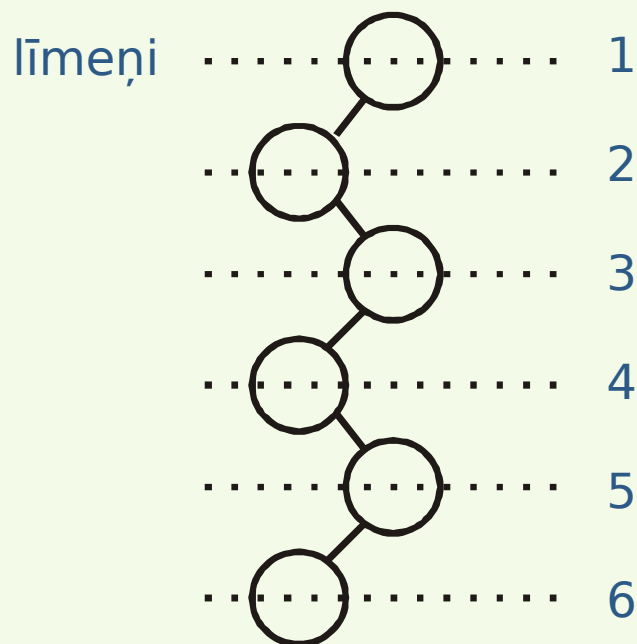
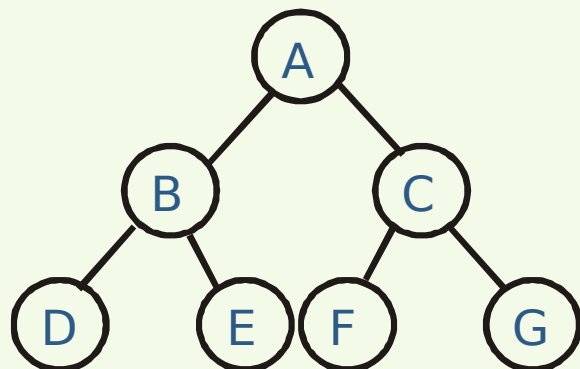
Koks (tree) jeb hierarhiskā datu struktūra (5)



līmenis - 1

līmenis - 2

līmenis - 3



ja A - tekošā virsotne, tad

$$\text{avePath} = \frac{(3+3+3+3+2+2+1)}{7} = 17 / 7 = 2,429$$

Koks (tree) jeb hierarhiskā datu struktūra (6)

Operācijas:

1) apkalpošanas operācijas:

Create,
Terminate, TreeDispose,
Characteristics,
Empty, Full;

2) meklēšanas operācijas:

Find, FindParient,
FindKey;

3) pamatoperācijas:

Traverse,
Insert,
Delete, DeleteSub,
Retrieve, Update.

Koks (tree) jeb hierarhiskā datu struktūra (7)

Binārais koks:

- 1) katrai virsotnei nevar būt vairāk kā 2 apakškoki un tādējādi – ne vairāk kā 2 pēcnācēji;
- 2) katrs apakškoks ir identificējams kā vecāka virsotnes kreisais apakškoks vai kā labais apakškoks;
- 3) koks var būt tukšs.

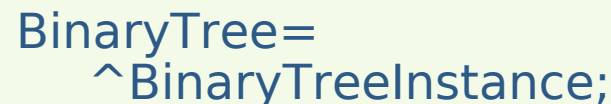
Rekursīvā binārā koka definīcija:

Binārais koks ir vai nu tukšs, vai arī to veido saknes virsotne un divi neatkarīgi binārie koki. Tie ir savā savā nesaistīti, un tos sauc par kreiso un labo apakškoku.

Operācijā Characteristics tiek noteikti 3 binārā koka raksturlielumi, izmantojot inorderālo apgaitu:

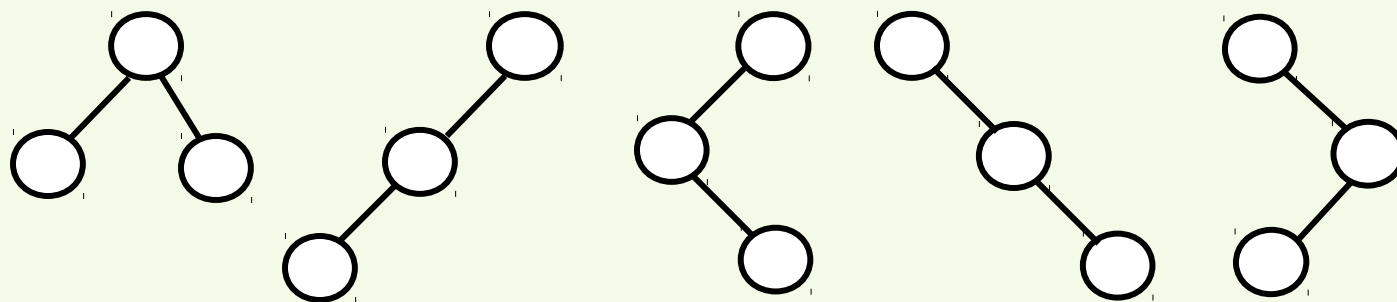
- 1) virsotņu skaits binārajā kokā (size);
- 2) binārā koka pakāpe (height);
- 3) vidējā ceļa garums (avePath) – visu ceļu summa no saknes virsotnes līdz citai virsotnei, dalīta ar šādu ceļu kopskaitu

Binārā koka attēlojums saistītā formā



Bināro koku konfigurācijas

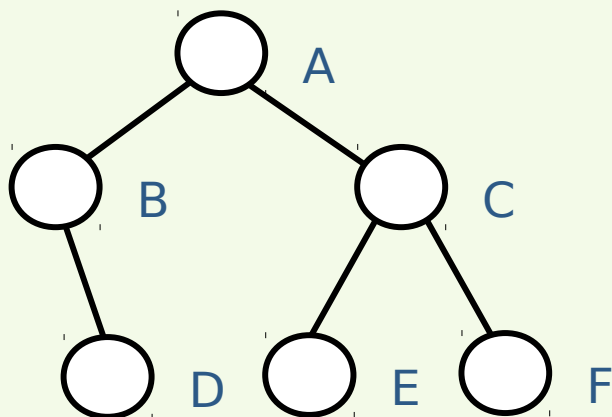
Binārie koki ar 3 virsotnēm: 5 dažādas konfigurācijas



Virsoņu skaits	1	2	3	4
Bināro koku skaits	1	2	5	14
Virsoņu skaits	5	6	15	16
Bināro koku skaits	42	132	9 694 845	35 357 670

Binārā koka apgaita (1)

Binārā koka apgaita (traverse – apgaita, traverss) ir tāda binārā koka virsotņu secība, kurā katra virsotne sastopama tikai vienu reizi. Ja binārajā kokā ir n virsotnes, tad iespējamo apgaitu skaits ir $n!$. Visbiežāk lieto tikai 3 dabiskāko apgaitu algoritmus: preorderālo, postorderālo un inorderālo binārā koka virsotņu apgaitu.



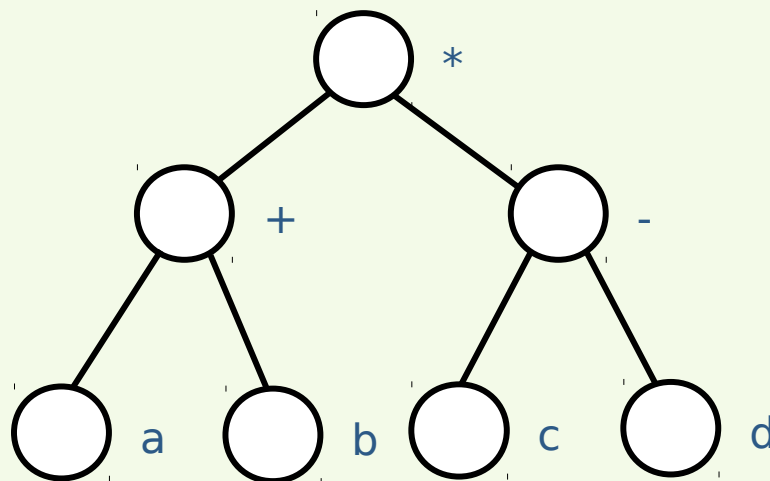
Preorderālā apgaita: A, B, D, C, E, F.

Inorderālā apgaita: B, D, A, E, C, F.

Postorderālā apgaita: D, B, E, F, C, A.

Binārā koka apgaita (2)

Apskatīsim, kāds būtu izteiksmes **(a+b)*(c-d)** attēlojums binārā koka veidā un kāda virsotņu apgaita būtu jāizvēlas, izteiksmi izskaitļojot.



Preorderālā apgaita: *, +, a, b, -, c, d.

Inorderālā apgaita: a, +, b, *, c, -, d.

Postorderālā apgaita: a, b, +, c, d, -, *.

Šāda veida bināros kokus sauc par izteiksmju kokiem. Kompilatori generē sintaktiskās analīzes kokus izteiksmju vērtību aprēķināšanai, izmantojot postorderālo apgaitu un stekus.

Binārā koka apgaita (3)

Binārā koka virsotņu apgaitas operāciju *Traverse* visērtāk realizēt rekursīvi, izmantojot iekšējās procedūras *PreOrd*, *InOrd* un *PostOrd* un ārējas procedūras *Proc* izsaukumu. Procedūra *Proc* paredzēta viena virsotnes elementa apstrādei, piemēram, lauku *data* un *key* izvadei ekrānā koka apgaitas procesā.

```
procedure Traverse ( BT: BinaryTree; ord:Order);  
{Binārā koka virsotņu apgaita, izvēloties apgaitas paņēmienu}  
procedure PreOrd (p: NodePointer; level: Count);  
begin  
    if p <> nil then begin  
        Proc(p^.el, level);  
        PreOrd(p^.left, level+1);  
        PreOrd(p^.right, level+1) end  
    end;  
end;
```


Binārā koka apgaita (4)

```
procedure InOrd (p: NodePointer; level: Count);
begin
    if p <> nil then begin
        InOrd(p^.left, level+1);
        Proc(p^.el, level);
        InOrd(p^.right, level+1) end
    end;
procedure PostOrd (p: NodePointer; level: Count);
begin
    if p <> nil then begin
        PostOrd(p^.left, level+1);
        PostOrd(p^.right, level+1);
        Proc(p^.el, level) end
    end;
begin
    sfēra}                                     {darbības
    case ord of
    preOrd: PreOrd(BT^.treeRoot, 1);
    inOrd:  InOrd(BT^.treeRoot, 1);
    postOrd: PostOrd(BT^.treeRoot, 1)
    end
end;
```

Binārais koks (1)

```
procedure Create (var BT: BinaryTree; var created: boolean);
{Izveido jaunu un tukšu bināro koku BT^}
begin
    new(BT);
    with BT^ do
        begin
            treeRoot:= nil;    current:=nil;
            St.size:= 0;    St.height:= 0;    St.avePath:= 0
        end;
    created:= true
end;
procedure Terminate (var BT: BinaryTree; var created: boolean);
{Likvidē bināro koku BT^}
begin
    if created then
        begin
            TreeDispose(BT, BT^.treeRoot);
            dispose(BT);
            created:= false
        end
    end;
end;
```

Binārais koks (2)

```
function Empty (BT: BinaryTree): boolean;  
{Pārbauda, vai binārais koks BT^ ir tukšs}  
begin  
    Empty:= BT^.treeRoot=nil  
end;
```

```
functionFull (BT: BinaryTree): boolean;  
{Pārbauda, vai binārais koks BT^ ir pilns}  
begin  
    Full:= BT^.ST.size=MaxSize  
end;
```

Binārais koks (3)

```
procedure Characteristics (BT: BinaryTree);  
{Nosaka binārā koka BT^ raksturlielumus}  
var ht, sz: Count;  
    ap, tpl: real;  
procedure InOrder (p: NodePointer; level: Count);  
begin  
    if p <> nil then  
        begin  
            InOrder(p^.left, level+1);  
            sz:= sz+1;    tpl:= tpl+level;  
            if ht < level then ht:= level;  
            InOrder(p^.right, level+1)  
        end  
    end;  
begin  
    sz:= 0;    ht:=0;    tpl:= 0;  
    with BT^ do  
        begin  
            if not Empty(BT) then  
                begin  
                    InOrd(treeRoot, 1);    ap:= tpl / sz  
                end  
            else ap:= 0;  
            St.size:= sz;    St.height:= ht;    St.avePath := ap  
        end  
    end;  
end;
```

Binārais koks (4)

```
procedure FindParent (BT: BinaryTree);  
  {Binārajā kokā BT^ sameklē tekošās virsotnes vecāku, kas kļūst par tekošo  
  virsotni}  
  var p: NodePointer;  
      S: Stack; ok: boolean;  
begin  
  if (not Empty(BT)) and (BT^.current <> BT^.treeRoot) then  
    begin  
      StackCreate(S, ok); {izveido steku nerekursīvai apgaitai}  
      with BT^ do {nerekursīva apgaita}  
        begin  
          p:= treeRoot;  
          while (p^.left <. current) and (p^.right <. current) do  
            begin  
              if p^.right <> nil then Push(S, p^.right);  
              if p^.left <> nil then p:= p^.left else Pop(S, p)  
            end;  
          current:= p  
        end;  
      StackTerminate(S, ok)  
    end  
  end;  
end;
```

Binārie koki (5)

```

procedure FindKey (var BT: BinaryTree; tkey: KeyType;
                  var found: boolean);
{Binārajā kokā BT^ meklē virsotni, kuras atslēgas lauka vērtība ir tkey.
Ja meklēšana ir sekmīga, sameklētā virsotne kļūst par tekošo}
var p: NodePointer;
procedure PreOrd (p: NodePointer; level: Count);
begin
  if (p <> nil) and (not found) then
  begin
    if p^.el.key = tkey then
    begin
      current:= p;  found:= true
    end
    PreOrd(p^.left, level+1);
    PreOrd(p^.right, level+1) end
  end
end;
begin
  found:=false;
  if not Empty(BT) then with BT^ do
  begin
    p := treeRoot;
    PreOrd(p,1)
  end
end;
end;

```

{darbības}

{uzdod virsotņu}

apgaitu}

Binārie koki (6)

```
procedure Find (var BT: BinaryTree; rel: Relative; var found:
boolean);
{Binārajā kokā BT^ meklē tekošās virsotnes radnieku rel. Ja
meklēšana ir
sekmīga sameklētā virsotne kļūst par tekošo}
begin
    found:= true;
    if not Empty(BT^) then with BT^ do
        begin
            case rel of
            root: current:= treeRoot;
            leftChild: if current^.left <> nil then
                current:= current^.left else found:= false;
            rightChild: if current^.right <> nil then
                current:= current^.right else found:= false;
            parent: if current <> treeRoot then FindParent(BT)
                else found:= false
            end
        end
    end
```

Binārie koki (7)

```
procedure TreeDispose(var BT: BinaryTree; p: NodePointer);
{Apakškokā ar sakni p^ dzēš visas virsotnes, izmantojot
postorderālo apgaitu}
begin
    if p <> nil then {ja p norāda uz kādu
virsočni}
        begin
            TreeDispose(BT, p^.left); {dzēš kreiso
apakškoku}
            TreeDispose(BT, p^.right); {dzēš labo
apakškoku}
            dispose(p) {dzēš pašu
virsočni p^}
        end
    end;

procedure Retrieve (BT: BinaryTree; var e: Sdelement);
{Binārajā kokā BT^ nolasa tekošās virsotnes saturu}
```


Binārie koki (8)

```

procedure Insert (var BT: BinaryTree; e: StdElement; rel: Relative;
                  var inserted: boolean);
{Binārajā kokā BT^ aiz tekošās virsotnes radnieka izvieta jaunu
 virsotni e, kas kļūst par tekošo virsotni}
var p: NodePointer;
begin
  if not Full(BT) then with BT^ do
    begin
      if (rel = leftChild) and current^.left <> nil) then
        inserted:= false
      else
        begin
          new(p);  inserted:= true;  p^.el:= e;
          p^.left:= nil;  p^.right:= nil;
          case rel of
            root: if Empty(BT) then treeRoot:= p  else inserted:=
false;
            leftChild: current^.left:= p;
            rightChild: current^.right:= p
          end ;
          current:= p;
          Characteristics(BT)
        end
      end;
end;
end;

```

Binārie koki (9)

```

procedure DeleteSub (var BT: BinaryTree);
{Bināraja kokā BT^ dzēš tekošās virsotnes apakškoku un saknes
virsotne kļūst
par tekošo virsotni}
var p: NodePointer; ok: boolean;
begin
  if not Empty(BT) then with BT^ do
    begin
      if current = treeRoot then                                {koks kļūs
tukšs}
        begin
          TreeDispose(BT, current);
          treeRoot:= nil;    current:= nil;
          St.size:=0;    St.height:=0;    St.avePath:=0
        end
      else                                                        {saknes virsotne nav
tekošā}
        begin
          p:= current;
          Find(BT, parent, ok);                                {sameklē
vecāku}
          if current^.left = p then current^.left:= nil
          else current^.right:= nil
        end;
      TreeDispose(BT, p);                                       {dzēš
apakškoku p}
    end
  end;

```

Binārie koki (10)

```
procedure Update (var BT: BinaryTree; e: Sdelement;  
k:Edit);  
  {Binārajā kokā BT^ labo tekošās virsotnes elementa  
saturu  
saskaņā ar labošanas variantu k}  
begin  
  if not Empty(BT) then with BT^ do  
    case k of  
      1: current^.el.data:= e.data;  
      2: current^.el.key:= e.key;  
      3: current^.el:= e  
    end  
  end;  
end;
```

Binārā koka attēlojums vektoriālajā formā (1)

Attēlojums vektoriālajā formā ir saistīts ar vairākiem trūkumiem, kas piemīt šim attēlojuma paņēmienam:

1) tā ka iepriekš grūti paredzēt virsotņu skaitu binārajā kokā, strādājot

ar to, tad jārezervē pietiekami liels brīvās atmiņas apgabals vektora

beigās jaunu virsotņu pievienošanai, tādējādi pamatatmiņas izmantošana nav ekonomiska un efektīva;

2) operāciju *Insert* un *Delete* izpilde saistīta ar elementu pārvietošanu

vektorā, kas operācijas izpildes laiku paildzina un samazina veiktspēju.

Šo iemeslu dēļ parasti bināros kokus attēlo saistītajā formā.

Attēlojot bināros kokus vektoriālā formā, skautnes attēlo ar

indeksu vertiāli nevis ar rādītājiem.

Binārā koka attēlojums vektoriālajā formā (2)

```

const MaxSize = 500;
type  Position = 1 ..
MaxSize;
      Count = 0 .. MaxSize;
      Edge = Count;
      DataType = string;
      KeyType = integer;
      StdElement = record
        data: DataType;
        key: KeyType
      end;
      Status = record
        size: Count;
        height: Count;
        avePath: real
      end;

```

```

Edit = 1 .. 3;
Order = (preOrder, inOrder,
        postOrder);
Relative = (leftChild, rightChild,
            root,
            parient);
Node = record
  el: StdElement;
  left, right: Edge
end;
BinaryTreeInstance = record
  el: array [Position] of
Node;
  current: Edge;
  St: Status
end;

```

BinaryTree =

^BinaryTreeInstance;

Binārā koka attēlojums vektoriālajā formā (3)

Piemēram:

Binārā koka vektors `BinaryTree^.el:`

Indekss	Virsozne	Kreisā	Labā
1	A	2	3
2	B	4	5
3	C	0	0
4	D	0	0
5	E	6	0
6	F	7	0
7	G	0	0

Binārā koka attēlojums vektoriālajā formā (4)

Otrā metode paredz, ka binārā koka virsotnes vektorā tiek izvietotas atbilstoši kādai virsotņu apgaitas secībai. Lai to ilustrētu, visvienkāršāk izvēlēties preorderālo apgaitu, kura sākas ar saknes virsotni. Katram virsotnes elementam papildus tiek pievienoti tikai 2 baiti loģiskās informācijas: vērtība *true* uzdod, ka kārtējā virsotne ir kāda vecāka kreisais vai labais bērns, bet vērtība *false*, ka virsotnei bērnu nav.

Binārā koka attēlojuma modeļa apraksts tādā pašā kā pirmajā paņēmienā, izņemot virsotnes aprakstu `BinaryTree^ el;`:

...	Indeks	Virsotn	Kreisā	Labā
Node = record	5	e		
el: StdElement;	1	A	true	true
left: boolean;	2	B	true	true
right: boolean	3	D	false	false
end;	4	E	true	false
NodePointer = ^Node;	5	F	true	false
...	6	G	false	false
	7	C	false	false

Binārā koka attēlojums vektoriālajā formā (5)

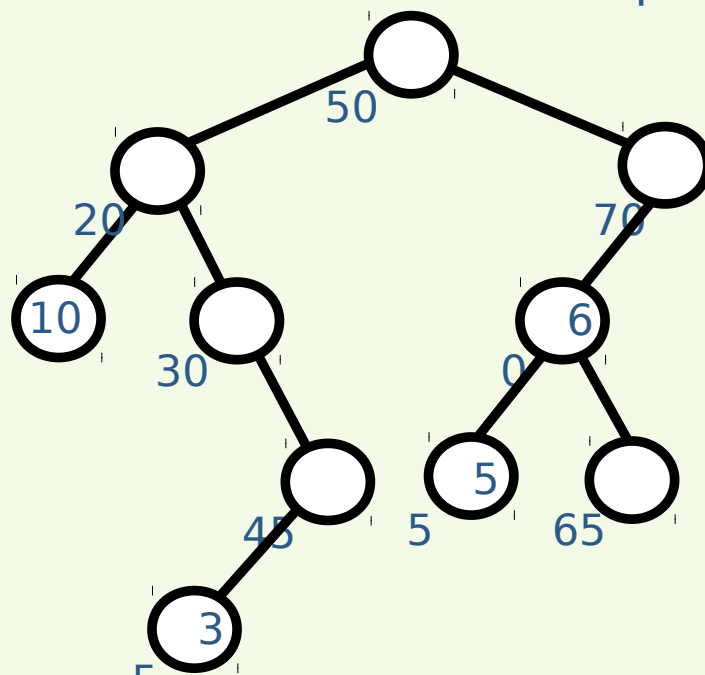
Trešā metode paredz, ka binārā kokā šķautnes tiek attēlotas netieši. Katrai virsotnei tiek piešķirta pozīcija pēc tā paša paņēmiena, kā veidojot kaudzi. Binārā koka i -tās virsotnes kreisā bērna pozīcijas numurs ir **$2i$** , bet labā bērna pozīcijas numurs ir **$2i+1$** . Ja kāds no šiem bērniem attiecīgajā līmenī kokā nav, vieta vektorā paliek tukša.

Binārās meklēšanas koks (1)

(binary search tree)

Par binārās meklēšanas koku sauc tādu bināro koku, kurā katrai virsotnei N ir spēkā šādi nosacījumi:

- 1) ja virsotne L ir virsotnes N kreisā apakškoka virsotne, tad $L < N$;
- 2) ja virsotne R ir virsotnes N labā apakškoka virsotne, tad $R > N$.



Ja N ir binārās meklēšanas koka virsotne, tad visas virsotnes tā kreisajā apakškokā ir mazākas par N , bet visas virsotnes tā labajā apakškokā ir lielākas par N .

Binārās meklēšanas koks (2)

(binary search tree)

Virsotnes binārajā meklēšanas kokā ir sakārtotas pēc virsotnes atslēgas lauka **key** vērtībām. Viena no virsotnēm binārajā meklēšanas kokā, ja tas nav tukšs, vienmēr ir tekošā virsotne. Tekošo virsotni iestata meklēšanas operācijas **FindKey**, **Find** un **FindParent**, kā arī pamatoperācijas **Insert** un **Delete**.

Bināro meklēšanas koku parasti attēlo saistītā formā. Binārās meklēšanas koka modeļa apraksts ir tāds pats kā parastajā binārajā kokā, mainīts tiek tikai binārā meklēšanas koka tipa nosaukums: **Binary SearchTree**:

```
...
BinarySearchTreeInstance = record
    treeRoot: NodePointer;
    current: NodePointer;
    St: Status
end;
BinarySearchTree = ^BinarySearchTreeInstance;
```

Salīdzinājumā ar parasto bināro koku, atšķirīgi būs tikai operāciju **FindKey**, **Insert**, **Delete** un **Update** algoritmi.

Binārā meklēšanas koka konfigurācija ir atkarīga no tā, kādā secībā un ar kādām jaunās virsotnes atslēgas lauka vērtībām tiek izpildīta operācija

Binārās meklēšanas koks (3)

(binary search tree)

Piemēram:

```
Create(BST, created);  
Insert(BST, 10,  
inserted);  
Insert(BST, 20,  
inserted);  
Insert(BST, 30,  
inserted);  
Insert(BST, 40,  
inserted);
```

```
Create(BST, created);  
Insert(BST, 40, inserted);  
Insert(BST, 20, inserted);  
Insert(BST, 50, inserted);  
Insert(BST, 10, inserted);  
Insert(BST, 30, inserted);  
Insert(BST, 60, inserted);
```

Binārās meklēšanas koks (4)

(binary search tree)

```

procedure FindKey (var BST: Binary SearchTree; tkey:KeyType;
                  var found: boolean);
{Binārās meklēšanas algoritms. Binārajā kokā BST^ meklē virsotni, kuras
atslēgas lauka vērtība ir tkey. Ja meklēšana ir sekmīga, sameklētā virsotne kļūst par tekošo. Ja
meklēšana ir nesekmīga, par tekošo virsotni kļūst tā vecāka virsotne, aiz kuras
būtu bijis jāatrodas virsotnei ar atslēgas lauka vērtību tkey}
var p, prior: NodePointer;
begin
    found:= false;
    if not Empty(BST) then with BST^ do
        begin
            p:= treeRoot;                                {meklēšanu sāk no
saknes}
            repeat                                       {virsotnes
meklēšana}
                prior:= p;                                {saglabā
vecāku}
                if p^.el.key = tkey then                 {sekmīga
meklēšana}
                    begin
                        current = p; found:= true
                    end
                else if tkey < p^.el.key > then p:= p^.left
                else p:= p^.right
            until found;
        end
    end
end

```

Binārās meklēšanas koks (5)

(binary search tree)

Operācijas FindKey izpildes efektivitāte ir $O(\log_2(n))$, kur n – virsotņu skaits binārās meklēšanas kokā. Virsotnes vienkāršajā ceļā veido lineāru datu struktūru, kuras elementu skaits ir vienāds ar binārās meklēšanas koka pakāpi, bet koka pakāpe vienmēr ir mazāka par virsotņu skaitu kokā.

```

procedure Insert (var BST: BinarySearchTree; e: Sdelement; var inserted:
boolean);
{Binārās meklēšanas kokā BST^ vecāka virsotnei, ja tā ir tekošā, pievieno
jaunu
virsotni e, kas kļūst par tekošo virsotni}
var p, psave: NodePointer;
    found: boolean;
begin
    inserted:= false;
    if not Full(BST) then with BST^ do
        begin
            psave:=current;                                {saglabā iepriekšējo
stāvvokli}
            FindKey(BST, e.key, found);                      {pārbaude
kokā}
            if found then current:=psave
            else                                              {kokā nav virsotnes ar atslēgu
e.key}
                begin
                    new(p);  p^.el:=e;                        {veido lapas
virsotni}
                    p^.left:= nil;  p^.right:= nil;
                    if not Full(BST) then current:=p
                    else if e.key < current^.el.key then current^.left:= p

```

Binārās meklēšanas koks (6)

(binary search tree)

Operācija **Delete** ir paredzēta, lai binārās meklēšanas kokā dzēstu tekošo virsotni. Ir svarīgi noskaidrot atšķirību starp jēdzieniem dzēst (delete) un aizvākt (remove). Aizvākt nozīmē likvidēt virsotni binārās meklēšanas kokā. Dzēst nozīmē likvidēt virsotnes elementu *e*. Aizvākšana ir dzēšanas sastāvdaļa. Rezultātā vai nu virsotne, vai virsotnes elements *e* binārās meklēšanas kokā vairs nav, tiek izmainīta arī binārās meklēšanas koka uzbūve. Atkarībā no tā, kāda virsotne binārās meklēšanas kokā ir jādzēš, reizēm dzēšana ir tekošās virsotnes aizvākšana, citos gadījumos dzēšamās virsotnes elements tiek aizstāts ar citas virsotnes elementu un tad šī cita virsotne ir jāaizvāc.

Binārās meklēšanas koks (7)

(binary search tree)

```

procedure Delete (var BST: BinarySearchTree; tkey: KeyType;
                  var deleted: boolean);
{Binārās meklēšanas kokā BST^ dzēš virsotni, kuras atslēgas lauka
vērtība ir tkey. Saknes virsotne kļūst par tekošo virsotni}
var remove: NodePointer;
procedure SubDel ( var q: NodePointer);
{Virsotnes dzēšana vai aizvākšana}
begin
    if q^.right <> nil then {labā vistālākā elementa
rekursīva}
        SubDel(q^.right); {meklēšana kreisajā
apakškokā}
    else {dzēšamā elementa aizstāšana
ar citu}
        begin
            remove^.el:= q^.el;
            remove:= q;
            q:= q^.left;
            q^.left:= remove;
        end
    end
end

```

Binārās meklēšanas koks (8)

(binary search tree)

```

procedure Del (tkey: KeyType; var p: NodePointer; var found:
boolean);
begin
  if Empty(BST) then found:= false
  else with BST^ do
    if tkey < el.key then
      Del(tkey, left, found)           {rekursīva meklēšana pa
kreisi}
    else if tkey > el.key then
      Del(tkey, right, found)          {rekursīva meklēšana pa labi}
    else
      {atslēga tkey ir atrasta kokā}
      begin
        remove:= p;
        if p^.left = nil then
          p:= right                    {virsošnei p^ nav kreisā
bērna}
        else if p^.right = nil then
          p:= left                     {virsošnei p^ nav labā
bērna}
        else SubDel(left);             {virsošnei p^ ir abi bērni}
        dispose(remove); found:= true
      end
    end;
  begin
    Del(tkey, BST^.left, found);
    Del(tkey, BST^.right, found);
  end;
end;

```

RTU akadēmiskās studiju programmas "Datortehniskās inženierzinātnes" kurss "Datoru sistēmas"
 2005/0125/VPD/ESF/PIAA/04/01/3.2.p.2/063/0007

Binārās meklēšanas koks (9)

(binary search tree)

Ja operācijā **Update** tiek labota tekošās virsotnes atslēgas lauka *key* vērtība, tad jaunās atslēgas vērtībai *e.key* jābūt lielākai par atslēgas vērtību tekošās virsotnes kreisajā apakškokā un mazākai par atslēgas vērtību tekošās virsotnes labajā apakškokā. Ja šādu prasību nevar nodrošināt, tad jāatsakās no labošanas operācijas vai arī pēc tās izpildes binārās meklēšanas koks jārestrukturē. Viens no paņēmieniem, kā to izdarīt, ir vispirms labojamo virsotni binārās meklēšanas kokā dzēst un pēc tam ar izlabotās atslēgas vērtību to pievienot no jauna.

Binārās meklēšanas koks (10)

(binary search tree)

```
procedure Update (var BST: Binary SearchTree; e: Stdelement;  
k: Edit);  
  {Binārās meklēšanas kokā BST^ labo tekošās virsotnes  
elementu  
atbilstoši labošanas variantam k}  
  var ok: boolean;  
  begin  
    if not Empty(BST) then with BST^ do  
      if k = 1 then current^.el.data:= e.data  
      else {jālabo datu  
lauks}  
      begin  
        Delete(BST, current^.key, ok);  
        Insert(BST, e, ok);  
      end  
    end  
  end;
```

AVL koki (1)

AVL (Adelson – Velsky – Landis) koki ir speciāli pēc pakāpes sabalansētu binārās meklēšanas koku paveids, kurā jebkuras virsotnes kreisā apakškoka pakāpe atšķiras no labā apakškoka pakāpes ne vairāk kā par vienu. Piemēram:

Vispārējā gadījumā katrai virsotnei divu apakškoku pakāpju starpība ir p ($p > 0$). Virsotni pievienojot vai dzēšot, AVL koks kļūst nesabalansēts, un, lai to novērstu, AVL koks tiek restrukturēts, lai iegūtu sabalansētību. Šādu restrukturizāciju AVL kokā sauc par rotāciju. Izpildot operācijas Insert vai Delete, iespējamās trīs dažādas rotācijas atkarībā no AVL koka konfigurācijas.

AVL koki (2)

Katram AVL kokam ir minimāla pakāpe, un, kā pierādīja Knuts, AVL koka pakāpe nekad nepārsniedz $1,45\log_2(n)$, aptuveni vērtējot $\log_2(n)$, kur n – virsotņu skaits.

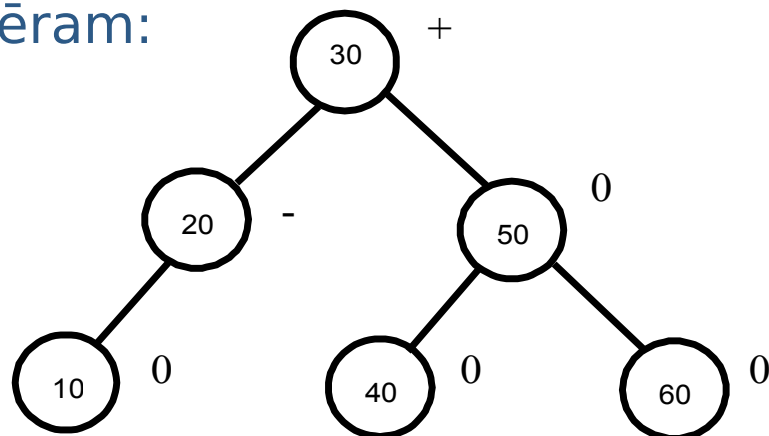
AVL koki tiek attēloti saistītā formā, lietojot modeli, kas līdzīgs tam, kādu lieto, attēlojot parastos bināros kokus. Atšķirība ir tai ziņā, ka katrai virsotnei pievieno papildus lauku *bal*, kurā attēlo attiecīgās virsotnes divu apakškoku sabalansētību.

Ja kādas virsotnes kreisā apakškoka pakāpe ir par vienu lielāka nekā labā apakškoka pakāpe, tad lauka *bal* vērtība ir *tall.left*. Ja kādas virsotnes labā apakškoka pakāpe ir par vienu lielāka nekā kreisā apakškoka pakāpe, tad lauka *bal* vērtība ir *tall.right*. Ja abu apakškoku pakāpes ir vienādas, tad virsotne ir sabalansēta un lauka *bal* vērtība ir *equalHt*. AVL koku attēlos šīs vērtības tiks

parādītas ar rakstzīmēm $-$, 0 un $+$.

AVL koki (3)

Piemēram:



AVL koka modeļa apraksts visumā ir tāds pats, ka parastam binārajam kokam, ar to atšķirību, ka papildus tiek definēts salansētības tips *Balance* un lauks *bal* koka virsotnē.

```

const MaxSize = 500;
type Count = 0 .. MaxSize;

Balance = (tallLeft, equalHt,
tallRight);

Node = record
  el: StdElement;
  left, right: NodePointer;
  bal: Balance
end;
  
```

Kaudze (1)

(heap)

Kaudze ir speciālā veidā organizēts binārais koks ar secīgu virsotņu izvietojumu, to parasti attēlo vektoriālā formā. Kaudze ir efektīvi izmantojama prioritātes rindas veidošanai.

Ja $r[1], r[2], \dots, r[n]$ ir elementu secība, tad tā būs kaudze, ja $r[i] < r[2i]$ un $r[i] < r[2i+1]$ visām i vērtībām. Šīs abas nevienādības sauc par kaudzes nosacījumiem.

Piemērā dotas trīs kaudzes, ievērojot kaudzes nosacījumus:

[1]	10	10	10
[2]	20	30	40
[3]	25	20	20
[4]	30	40	50
[5]	40	50	42
[6]	42	25	30
[7]	50	55	25
[8]	52	52	52
[9]	55	42	55

Kaudze (2) (heap)

Sašķirotu elementu secība vienmēr ir kaudze.

Kaudzi var definēt arī kā pabeigto bināro koku, kurā katra vecāka virsotnes prioritāte ir lielāka, lielāka vai vienāda, mazāka, mazāka vai vienāda par katra bērna prioritāti. Ja vecāka virsotne ir mazāka kā tās bērni, tad tas nozīmē, ka vecāka virsotnes prioritāte ir mazāka par katra bērna virsotnes prioritāti.

Virsošnes kaudzes binārajā kokā tiek pievienotas saskaņā ar kaudzes elementu secību, sākot ar sakni un aizpildot līmeni pēc līmeņa. Katrā līmenī virsošnes tiek izvietotas no kreisās uz labo pusi. Kaudzes binārā koka konfigurācija ir pabeigtais (complete) binārais koks.

Kaudze (3) (heap)

Attēlā katrai kaudzes virsotnei pievienoti arī virsotņu secības numuri. Šajā kaudzes binārajā kokā katra vecāka virsotne ir mazāka par tās bērnu virsotnēm.

Katrā vienkāršajā ceļā no saknes līdz kādai no lapām visas virsotnes veido augošu virsotnes elementu secību. Saknes elements ir vismazākais.

Ja kaudzei grib pievienot jaunu virsotni vai kāda virsotne jādzēš vai jālabo, tad kaudze tiek rekonfigurēta tā, lai atkal visas virsotnes atbilstu kaudzes nosacījumiem. Šim nolūkam paredzētas speciālas operācijas *SiftUp* un *SiftDown*, ar kurām virsotņu elementus izvieto modeļa vektora vajadzīgajās pozīcijās.

Kaudze (4)

(heap)

```
const MaxSize = 500;
type Count = 0 .. MaxSize;
Edge = Count;
DataType = string;
KeyType = integer;
Priority = integer;
StdElement = record
    data: DataType;
    key: KeyType;
end;
Edit = 1 .. 3;
Relative = (leftChild, rightChild,
            root, parent);

Node = record
    el: StdElement;
    pty: Priority;
end;
NodePointer = ^Node;
HeapInstance = record
    el: array [Count] of
        Node;
    n: count;
end;
Heap = ^HeapInstance;
```

Kaudze (5) (heap)

```
procedure SiftUp (var H: Heap);  
  {Kaudzē  $H^{\wedge} n$ -to elementu pārvieto virzienā uz sākumu,  
  nodrošinot  
  kaudzes nosacījumu izpildi}  
  var child, parent: Count;  
  begin  
    if not Empty(H) then with  $H^{\wedge}$  do  
      begin  
        el[0]:= el[n];  
        child:= n;    parent:= n div 2;  
        while el[parent].pty > el[0].pty do  
          begin  
            el[child]:= parent;  
            parent:= parent div 2  
          end;  
        el[chil]:= el[0]  
      end  
    end  
  end;
```

Kaudze (6)

(heap)

```

procedure SiftDown (var H:Heap; k: Count);
{Kaudzē H^ k-to elementu izsijā lejupvirzienā, nodrošinot kaudzes
nosacījumu
izpildi}
var child, parent: Count;
begin
  if not Empty(H) then with H^ do
    begin
      el[0]:= el[k];           {izsijāmo elementu
saglabā}
      parent:= k;   child:= k+k;
      while child <= n do      {kamēr ir vismaz viens
bērns}
        begin
          if child < n then    {ja ir divi
bērni}
            if el[child].pty > el[child+1].pty {izvēlas
vienu}
              then child:= child+1;          {ar mazāko
prioritāti}
          if el[0].pty > el[child].pty then    {meklē
vietu}
            begin
              el[parent]:= el[child];
              parent:= child;   child:= parent+parent

```

Kaudze (7) (heap)

```
procedure HeapCreate (H: Heap);  
{Rekonfigurē kaudzi H^}  
var k: Count;  
begin  
    k := (n div 2) + 1;  
    while k > 1 do  
        begin  
            k := k - 1;  
            SiftDown(H, k)  
        end  
    end;  
end;
```

B-koks (1)

B-koks ir speciāls koka veids apjomīgas informācijas glabāšanai un efektīvai sameklēšanai. Koka nosaukuma rašanās hipotēze – firmā Boeing izstrādātā datu struktūra.

Katra B-koka virsotne satur vairākus elementus un tai var būt daudz bērnu. Šo īpašību dēļ un arī tāpēc, ka B-koki ir labi sabalansēti (līdzīgi kā AVL-koki), tie nodrošina īsu vienkāršo ceļu, kurā pieejams visai liels virsotņu sakopojums. Operācijās **Insert** un **Delete** ir garantēts, ka garākais ceļš no saknes uz lapu ir $O(\log_2 n)$.

B-koks (2)

Par n -tās kārtas B-koku sauc tādu stipri sazarotu koku ar pakāpi $2n+1$, kuram piemīt šādas īpašības:

- 1) katra virsotne, izņemot sakni, satur ne mazāk kā n un ne vairāk kā $2n$ atslēgas (n ir B-koka kārtā);
- 2) sakne satur vismaz 1 un ne vairāk kā $2n$ atslēgas;
- 3) visas lapas izvietotas vienā līmenī;
- 4) katrā virsotnē rādītāju skaits ir par 1 lielāks nekā atslēgu skaits;
- 5) katra virsotne satur 2 sarakstus: augošā secībā pēc atslēgām sakārtotu elementu *el* sarakstu un atslēgas lauku vērtībām *key* atbilstošu rādītāju *ptr* sarakstu. Lapām rādītāju saraksta nav.

Stipri sazarota koka pakāpe ir vismaz 3.

No virsotnes izejošo šķautņu skaits ir mazāks vai vienāds ar n , kur n ir B-koka kārtā.

Jebkurā momentā, strādājot ar B-koku, attālums no saknes līdz jebkurai lapai ir vienāds ar fiksētu lielumu d .

B-koks (3)

Kādas virsotnes bērnu skaits ir 0 vai par vienu lielāks kā elementu skaits šai virsotnē.

Apskatīsim B-koku ar kārtu $n=2$. To sauc arī par otrās kārtas B-koku. Praksē visai parasti ir B-koki ar kārtu $n=100$ vai lielāku.

Ja $n=2$, tad

- 1) atslēgu skaits saknē: 1 – 4;
- 2) atslēgu skaits pārējās virsotnēs: 2 – 4;
- 3) iespējamo bērnu skaits: 0 – 5;
- 4) pieņemsim, ka atslēgu vērtību diapazons: 1 – 31.

B-koks (4)

Kreisajā apakškokā atrodas visas virsotnes ar atslēgām, kuras mazākas par saknes atslēgas vērtību 20, bet labajā apakškokā – visas virsotnes ar atslēgām, kuras lielākas par saknes atslēgas vērtību 20.

Operāciju **Insert** un **Delete** algoritmi ir sarežģīti ar vairākiem alternatīviem risinājumiem.

```

const MaxSize = 500;           {uzdod B-koka
lietotājs}
    Order = 2;                 {B-koka
kārta}
    NodeSize = 4;              {max elementu skaits
virsoņē}

    ...
type Nodepointer = ^ Node;
Node = record
    parent: Nodepointer;
    el_array[1..NodeSize] of StdElement;

```


B-koks (4)

Tabulā sniegta informācija par sakarībām starp virsotņu līmeni un virsotņu un elementa skaitu B-kokā.

Līmenis	Minimālais virsotņu skaits	Minimālais elementu skaits	Maksimālais virsotņu skaits	Maksimālais elementu skaits
1	1	1	1	$2n$
2	2	$2n$	$2n+1$	$2n(2n+1)$
3	$2(n+1)$	$2n(2n+1)$	$(2n+1)^2$	$2n(2n+1)^2$
4	$2(n+1)^2$	$2n(2n+1)^2$	$(2n+1)^3$	$2n(2n+1)^3$
...		...		
k	$2(n+1)^{k-2}$	$2n(2n+1)^{k-2}$	$(2n+1)^{k-1}$	$2n(2n+1)^{k-1}$

Šķirošanas (sorting) jēdziens (1)

Datu apstrāde paredz arī tādas operācijas kā lietotājiem paredzēto datu sakārtošana pēc kāda kritērija, piemēram, alfabētiskā secībā. Datu šķirošana ir to kārtošana pēc atslēgām augošā vai dilstošā secībā. Sašķirotiem datiem ir liela nozīme datu meklēšanas procesā (operācija **FindKey**), jo tad ir iespējams izmantot binārās meklēšanas algoritmu.

Sašķirojamie dati parasti izvietoti vektoriālajā formā attēlotajā sarakstā jeb tabulā. Tikai daži meklēšanas algoritmi ir izmantojami (radix sort), ja saraksts attēlots saistītajā formā, pie kam šai gadījumā būtu nepieciešama divkāršā elementu sasaiste.

Ir daudz un dažādi šķirošanas algoritmi. Visi šķirošanas algoritmi, vadoties no datu apjoma, iedalāmi divās atšķirīgās grupās:

1) iekšējās (internal) šķirošanas algoritmi;

2) ārējās (external) šķirošanas algoritmi

Šķirošanas (sorting) jēdziens (2)

Iekšējās šķirošanas gadījumā datu apjoms nav liels, tāpēc tie izvietojami datora pamatatmiņā (RAM). Tos lieto visbiežāk. Ir daudz un dažādi iekšējās šķirošanas algoritmi. Svarīga problēma ir šķirošanas algoritma izvēle un tā efektivitātes novērtējums. Šķirošanas procesā notiek 2 elementu izvēle, to salīdzināšana un pārsūtīšana no vienas vieta uz citu, ja tas nepieciešams. Algoritma veikspēju (performance) parasti izsaka matemātiskās kārtas veidā, piemēram: $O(n^2)$.

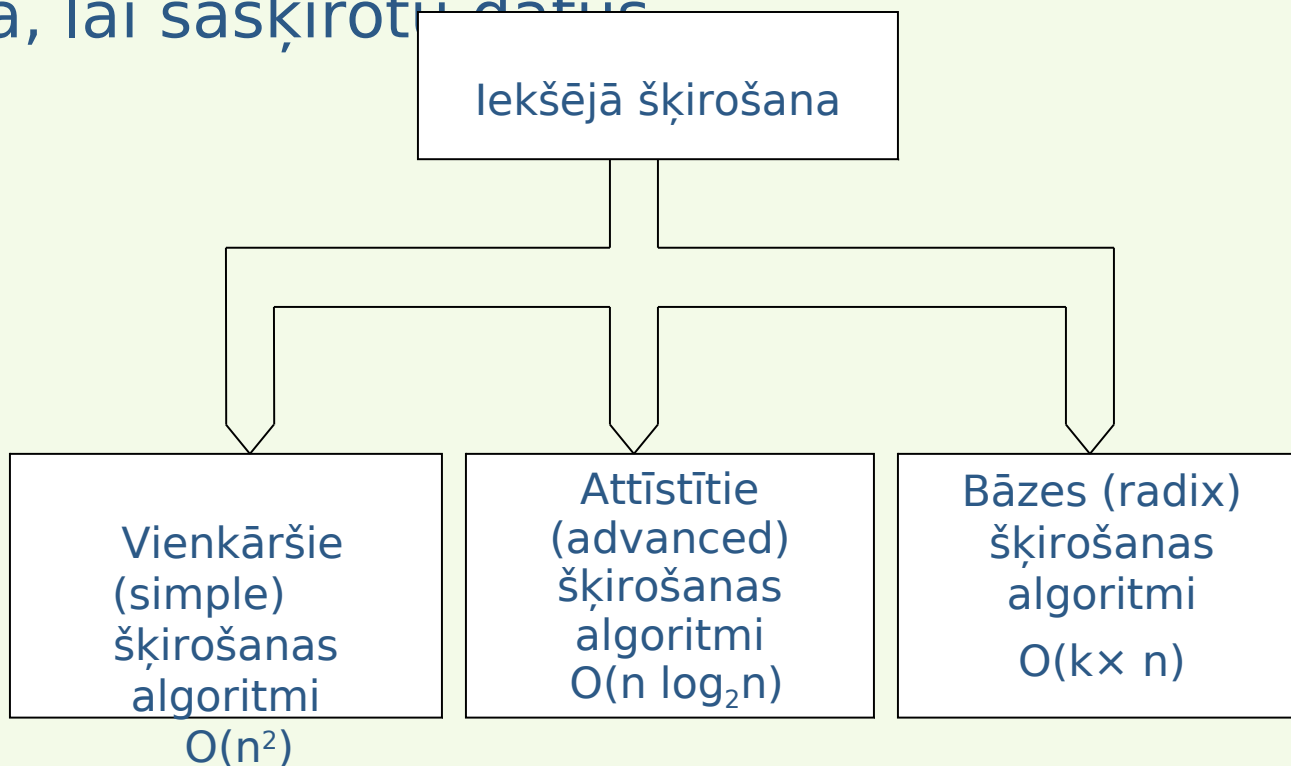
Ārējās šķirošanas gadījumā datu apjoms ir liels un tie tiek glabāti ārējā (diska) atmiņā.

Lai apmainītu divus elementus vietām, tiks izmantota operācija *Swap*:

```
procedure Swap (x,y: StdElement);  
var temp: StdElement;  
begin  
    temp := x;  
    x := y;  
    y := temp;  
end;
```

Šķirošanas (sorting) jēdziens (3)

Visi iekšējās šķirošanas algoritmi ir klasificējami pēc to izpildes efektivitātes, t.i., cik daudz darbību (salīdzināšanas un/vai pārsūtīšanas operāciju) jāizpilda, lai sašķirotu datus.



Šķirošanas (sorting) jēdziens (4)

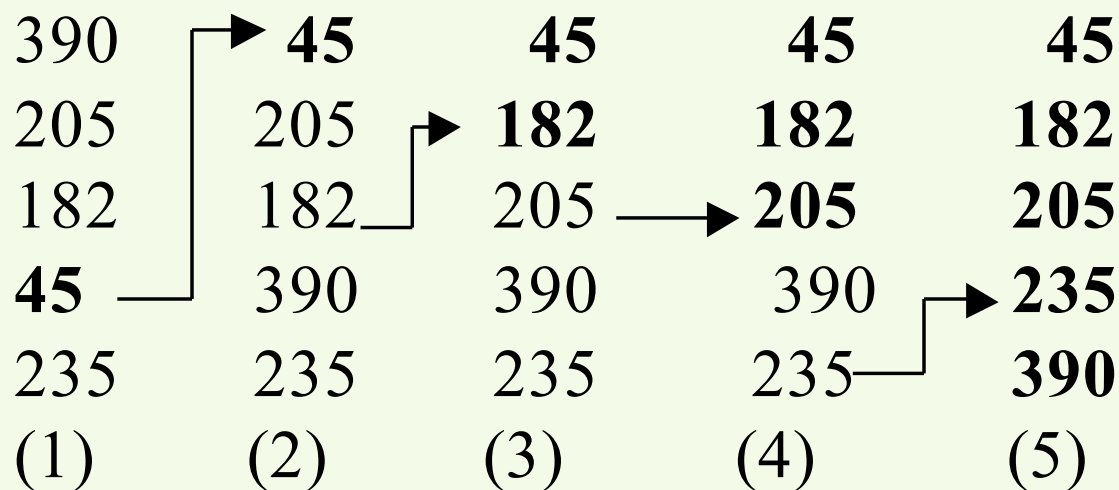
Dinamiski veidota vektoriālajā formā attēlotā saraksta

```
apraksts: const MaxSize = 100;                                {uzdod  
    lietotājs}  
    type Position = 1 .. MaxSize;  
    {indeksa tips}  
        Count = 0 .. MaxSize;                                {elementu  
skaits tips}  
        DataType = string;                                    {jebkurš  
datu tips}  
        KeyType = integer;    {skalārs ordināls datu tips  
vai string}  
        StdElement = record                                    {saraksta  
elementu tips}  
            data: DataType;  
            key: KeyType  
        end;  
        ListInstance = record
```

Vienkāršie šķirošanas algoritmi (1)

Izvēles šķirošana (Selection Sort)

Izvēles šķirošanas pamatoperācija ir mazākā (lielākā) elementa izvēle elementu secībā:



Vienkāršie šķirošanas algoritmi (2)

```
procedure SelectSort (var L: List);
var i, j, small: Position;
begin
  with L^ do
    begin
      for i:= 1 to n-1 do
        begin
          small:= i;
          for j:= i+1 to n do
            if el[j].key < el[small].key then small:= j;
          Swap(el[i], el[small])
        end
      end
    end
end;
```

Iekšējā cikla izpildes skaits: $(n-1)+(n-2)+ \dots +1 = 0.5n(n-1) \Rightarrow \mathbf{O(n^2)}$.
Tajā tiek izpildīta viena salīdzināšanas operācija un vairākas piešķires operācijas.

Ārējais cikls tiek izpildīts $n-1$ reizi, un katrā reizē tiek izpildīta viena apmaiņas operācija, tātad pavisam $\mathbf{O(n)}$ pārvietošanas operācijas, kas ir šīs metodes priekšrocība.

Vienkāršie šķirošanas algoritmi (3)

Apmaiņas šķirošana (Exchange Sort)

Pirmais šķirošanas solis:

390	205	205	205	205
205	390	182	182	182
182	182	390	45	45
45	45	45	390	235
235	235	235	235	390
(1)	(2)	(3)	(4)	(5)

Apmaiņas šķirošanas pamatoperācija ir divu blakus esošu elementu salīdzināšana un, ja tie neatbilst kārtojumam, tad tie tiek apmainīti vietām. Viss šķirošanas process sastāv no vairākiem posmiem. Pirmajā šķirošanas posmā vislielākais elements tiek pārvietots uz pozīciju n . Otrajā posmā operē tikai ar $n-1$ elementu, un nākamais vislielākais elements tiek pārvietots uz pozīciju $n-1$.

Vispārīgā gadījumā pēc i -tā posma izpildes i -tais lielākais elements tiks pārvietots uz pozīciju $i-1$, pārbaudot tikai $i-1$ elementu.

Tātad mazie elementi lēnām pārvietojas uz saraksta sākumu. Tāpēc šo šķirošanas algoritmu sauc arī par **burbulmetodi (bubble sort)**.

Vienkāršie šķirošanas algoritmi (4)

```
procedure ExchangeSort (var L: List);
var i, j: Position;
    sorted: boolean;
begin
    with L^ do
        begin
            i:= n;    sorted:= false;
            while (i > 0) and (not sorted) do
                begin
                    sorted:= true;
                    for j:= 1 to i-1 do
                        if el[j].key > el[j+1] then
                            begin
                                Swap(el[j], el[j+1]); sorted:= false
                            end;
                    i:= i-1;
                end
            end
        end
    end;
```

Vienkāršie šķirošanas algoritmi (5)

Iekšējā cikla atkārtojuma skaits:

$$(n-1) + (n-2) + \dots + (n-i_{\text{sorted}}) = 0.5(2n-i_{\text{sorted}}-1)i_{\text{sorted}},$$

kur i_{sorted} ir ārējā cikla izpildes skaits, kamēr sasniegts solis, kurā nebija nevienas apmaiņas operācijas.

Vissliktākajā gadījumā $i_{\text{sorted}} = n-1$, un šai gadījumā salīdzinājumu skaits būs $0.5n(n-1) \Rightarrow O(n^2)$.

Apmaiņas šķirošanai ir divi galvenie trūkumi:

1) iekšējā ciklā ir apmaiņas operācija ar 3 datu pārvietojumiem.

Jāatzīmē, ka izvēles šķirošanas algoritma iekšējā ciklā datu

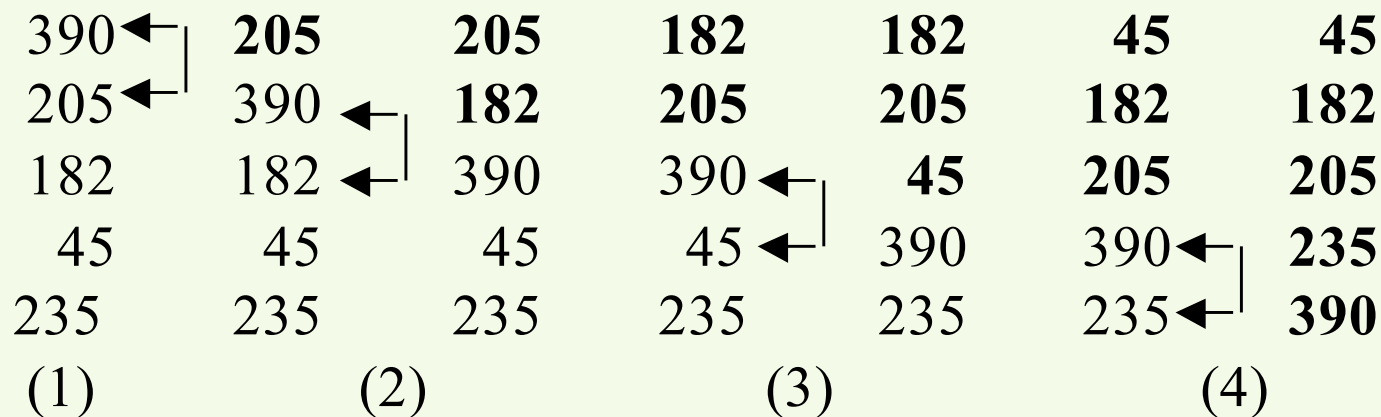
pārsūtīšanas operāciju vispār nav;

2) ja elements tiek pārvietots, tad to pārsūta uz blakus esošo

Vienkāršie šķirošanas algoritmi (6)

Iestarpinājuma šķirošana (Insertion Sort)

Izvietojuma šķirošanas pamatoperācija ir attiecīgā saraksta elementa izvietošana elementu secībā tā, lai veidotos sašķirotu elementu secība.



Vienkāršie šķirošanas algoritmi (7)

```
procedure InsertSort (var L: List);
var i, j: Count;
    temp: StdElement;
begin
    with L^ do
        begin
            for i:= 1 to n-1 do
                begin
                    if el[i].key > el[i-1].key then continue;
                    temp:= el[i];
                    j:= i-1;
                    while (j >= i) and (el[j].key > temp.key) do
                        begin
                            el[j+1]:= el[j];
                            j:= j-1
                        end;
                    el[j]:= temp
                end
            end
        end
    end;
```

Vienkāršie šķirošanas algoritmi (8)

Ārējais cikls vienmēr tiks izpildīts $n-1$ reizi. Iekšējā cikla atkārtojumu skaits ir atkarīgs no cikla parametra i vērtības, gan arī no saraksta elementu secības. Vissliktākās elementu secības gadījumā iekšējā cikla atkārtojumu skaits būs šāds:

$$(n-1)+(n-2)+ \dots +1 = 0.5n(n-1) \Rightarrow \mathbf{O(n^2)}.$$

Tomēr caurmērā šī algoritma veikspēja varētu būt divreiz lielāka, jo vidēji būtu jāpārbauda tikai puse no elementu skaita.

Ja salīdzina iestarpinājuma un izvēles šķirošanas algoritmus, tad var secināt, ka iestarpinājuma šķirošanas algoritma iekšējā ciklā ir vairāk izpildāmu darbību. Toties izpildāmo darbību apjoms šai algoritmā ir mazāks kā apmaiņas šķirošanas algoritmā.

Iestarpinājuma šķirošanas algoritms ir no retajiem, kas izmantojams, lai sašķirotu saistītā saraksta elementus augošā secībā. Par saraksta modeli jāizvēlas divkāršsaistītais saraksts.

Vienkāršie šķirošanas algoritmi (9)

Šella šķirošanas algoritms uzskatāms par uzlabotu izvietojuma šķirošanas algoritmu, kurā salīdzina saraksta elementus, kuri atrodas noteikta pozīciju skaita (soļa h) attālumā viens no otra. Elementu pēdējā salīdzināšanas iterācijā šis solis ir viens, un Šella šķirošana tiek izpildīta tāpat kā iestarpinājuma šķirošana. Šī algoritma autors ir Donalds Šells, kas to publicēja 1959.gadā. Par šo algoritmu ir plašs publikāciju klāsts un tam ir daudzveidīgas realizācijas.

Šella šķirošanas algoritma veikspēja ir diapazonā no $O(n \log_2 n)$ līdz $O(n^{1.5})$ atkarībā no saraksta elementu secības un algoritma realizācijas.

Tiek piedāvāti vairāki paņēmieni, kā izvēlēties šķirošanas soļa h vērtības:

1) sākotnējā soļa h vērtība ir $n/2$, un katrā nākamajā kārtošanas reizē iepriekšējā

soļa vērtība tiek dalīta uz pusēm $h = h/2$ vai arī dalīta ar 2.2: $h = h/2.2$;

2) šķirošanas soļa vērtība tiek aprēķināta pirms šķirošanas:

$h := 1$;

repeat $h := 3 * h + 1$ until $h > n$;

$h := h \text{ div } 3$;

{uzsākot

šķirošanu}

Ar šo paņēmieni iegūst šādas soļa vērtības:

$h = 1, 4, 13, 40, 121, \dots$;

3) šķirošanas soļa vērtības tiek ierakstītas speciālā vektorā izguvei, kad attiecīgā

soļa vērtība būs nepieciešama kārtējā šķirošanas iterācijā:

type Arr = [1..16] of integer;

const h: Arr = (1391376, 463792, 198768, 86961, 33936, 342

105776, 4592, 1968, 861, 336, 112, 48, 21, 3, 1);

Vienkāršie šķirošanas algoritmi (10)

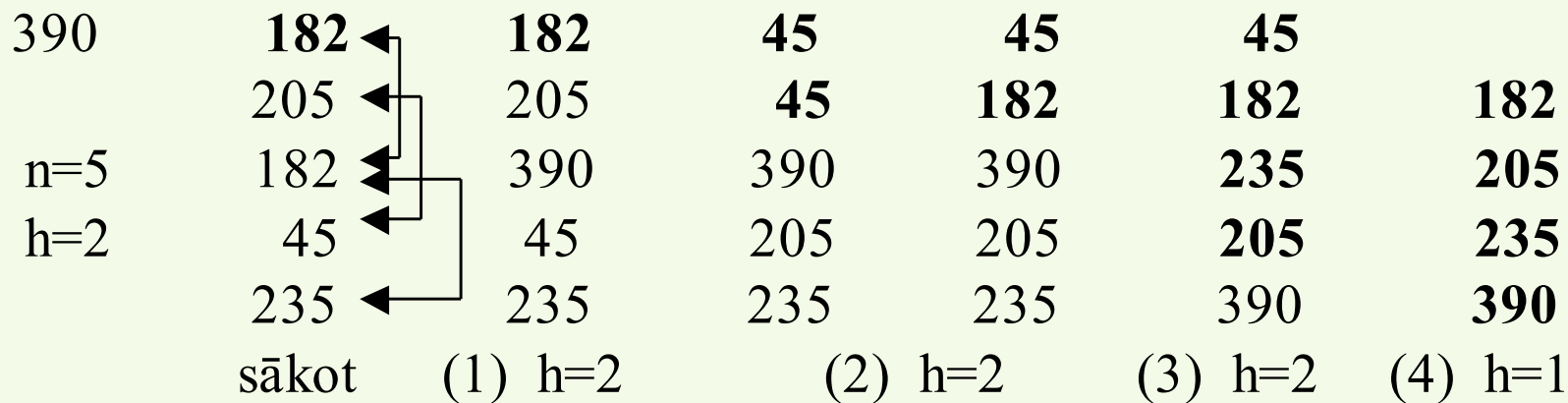
```

procedure ShellSort (var L: List);
var i, j, h: Position;
    temp: StdElement;
begin
    with L^ do
        begin
            h:= n div 2;
            while h >= 1 do
                begin
                    for i:= 1 to n-h do
                        if el[i].key > el[i+h].key then
                            begin
                                temp:= el[i+h]; j:= i;
                                while (j >= 1) and (el[j].key >
temp.key) do
                                    begin
                                        el[j+h]:= el[j]; j:= j-h
                                    end;
                                el[j+h]:= temp
                            end;
                    h:= h div 2
                end
            end
        end
    end;
end;

```

Vienkāršie šķirošanas algoritmi (11)

Šella šķirošana:



Uzlabotie šķirošanas algoritmi (1)

Ātrā šķirošana (Quicksort)

Ātrās šķirošanas algoritms ir vispopulārākais un visbiežāk lietotais no attīstītajiem šķirošanas algoritmiem, tā autors ir Hors (C. A. R. Hoare). Parasti to lieto, lai sašķirotu sarakstus, kurā daudz elementu.

Lai sašķirotu saraksta elementus, caurmērā vidēji nepieciešams izpildīt

$O(n \log_2 n)$ salīdzināšanas operācijas, tātad, ja elementu skaits ir liels, tas ir ievērojami ātrdarbīgāks kā citi šķirošanas algoritmi.

Ātrā šķirošana izmanto stratēģiju "skaldi un valdi", lai sašķirojamo sarakstu sadalītu divos apakšsarakstos. Šķirošanas procesā tiek izpildītas šādas darbības:

1) sarakstā izvēlas elementu, ko sauc par centru jeb asi (pivot);

2) sarakstu pārkārto tā, ka visi elementi, kas mazāki par centru, tiek

pārvietoti sarakstā pa kreisi no tā (pirms tā), bet visi elementi, kas

lielāki vai vienādi ar centru – tiek izvietoti aiz tā. To sauc par sadalīšanas operāciju (partition). Centra elements jau atrodas

sašķirotā pozīcijā:

3) pēc tam rekursīvi šķiro mazāko elementu apakšsarakstu un

Attīstītie šķirošanas algoritmi (2)

Svarīga problēma ir šķirošanas centra izvēle. Ir vairāki paņēmieni tā noteikšanai, bez tam ir iespējami arī daudz variantu, kā pārkārtot elementus, formējot divus apakšsarakstus. Tātad iespējamās vairākas ātrā šķirošanas algoritma modifikācijas un uzlabojumi.

```
procedure QuickSort1 (var L: List);  
  procedure Quick (left, right: Position);  
    var j: Position;  
    begin  
      if left < right then  
        izvēlas centra elementu el[j] un pārkārto sarakstu tā lai  
        elementi el[left], ..., el[j-1] ir mazāki par el[j],  
        elementi el[j+1], ..., el[right] ir lielāki par el[j]  
        Quick(left, j-1); Quick(j+1, right)  
      end;  
    begin  
      Quick(1, L^.n)  
    end.
```

Attīstītie šķirošanas algoritmi (3)

```

procedure QuickSort2 (var L: List);
  procedure Quick2 (left, right: Position);
    var j, k: Position;
    begin
      with L^ do
        begin
          if left < right then
            begin
              if el[left].key > el[right].key then Swap(el[left], el[right]);

              j:= left;  k:= right;
              repeat
                repeat  j:= j+1  until  el[j].key >= el[left].key;
                repeat  k:= k-1  until  el[k].key <= el[left].key;
                if j < k then Swap(el[j], el[k])

              until j > k;
              Swap(el[left], el[k]);
              Quick2(left, k-1);
              Quick2(k+1, right)
            end
          end
        end
      end
    end
  end
end

```

{1}

{2}

{3}

Attīstītie šķirošanas algoritmi (4)

Pieņemsim, ka sarakstā ℓ ir elementi ar šādām atslēgām:

70 40 15 30 25 60 10 75 45 65 35 50 20
55
left = 1, n=14, right = 14.

Operators {1}, sākot ar pozīciju left+1, pa kreisi sameklē elementu, kas lielāks par el[left]. Meklēšana sākas ar elementu **15** un beidzas ar **60**.

Operators {2}, sākot ar pozīciju right-1 virzienā pa labi sameklē elementu, kas kas mazāks par el[right], tas ir elements **20**.

70 40 15 30 25 60 10 75 45 65 35 50 20 55

Ja šie divi meklēšanas apgabali nepārklājas, t.i., ja $j < k$, tad operators $\{3\}$ šos sameklētos elementus apmaina vietām:

70 55 40 15 30 25 20 10 75 45 65 35 50 60

Kamēr vien $j < k$, operatori $\{1\}$, $\{2\}$ un $\{3\}$ tiek atkārtoti. Kad $j > k$, ir iegūts šāds elementu kārtojums:

70 55 35 15 30 25 20 10 40 45 65 75 50 60 24

Attīstītie šķirošanas algoritmi (5)

Kaudzes šķirošana (HeapSort)

Tas ir divposmu šķirošanas process. Vispirms sašķirojamie elementi tiek izvietoti kaudzē, bet otrajā posmā tie tik izgūti no kaudzes sašķirotā secībā. Darbības principa ziņā kaudzes šķirošana ir līdzīga izvēles šķirošanai, jo abas metodes izvēlas pēc kārtas lielākos elementus un tos apmaina vietām, veidojot sašķirotu secību. Ja saraksts ir mazāks, izvēles šķirošana strādās ātrāk.

```

const MaxSize = 500;
type Count = 0 ..
    MaxSize;
    DataType = string;
    KeyType =
        integer;
    StdElement =
        record
            data: DataType;
            key: KeyType;
        end;
    ListInstance = record
        el: array [Count] of
            StdElement;
        n: Count
    end;
    List = ^ListInstance;

```

Kaudzes šķirošanas algoritma veikspēja, ja arī elementu secība ir visneoptimālāka ir $O(n \log_2 n)$.

Attīstītie šķirošanas algoritmi (6)

```

procedure SiftDown (var L:List; k: Count);
{Kaudzē  $L^k$  elementu izsijā lejupvirzienā, nodrošinot kaudzes
nosacījumu izpildi}
var child, parent: Count;
begin
  if not Empty(L) then with  $L^k$  do
    begin
      el[0]:= el[k];                                {izsijājamo elementu
saglabā}
      parent:= k;  child:= k+k;
      while child <= n do                             {kamēr ir vismaz viens
bērns}
        begin
          if child < n then                               {ja ir divi
bērni}
            if el[child].key > el[child+1].key then
              {izvēlas vienu}
              child:= child+1;                          {ar mazāko
atslēgu}
            if el[0].key > el[child].key then              {meklē
vietu}
              begin
                el[parent]:= el[child];
                parent:= child; child:= parent+parent
              end
            end
          end
        end
      end
    end
  end

```

Attīstītie šķirošanas algoritmi (7)

```

procedure HeapCreate(var L: List);
{Izveido kaudzi L^}
var k: Count;
begin
    k:= (L^.n div 2) +1;
    while k >1 do                                {pārkārto elementus atbilstoši kaudzes
nosacījumiem}
        begin
            k:= k-1; SiftDown(L,k)
        end
    end;
end;
procedure HeapSort (var L: List);
var k: Position;
begin
    HeapCreate (L);                                {izveido
kaudzi}
    with L^ do
        begin
            while k > 1 do
                begin
                    k:= n; Swap(el[1], el[k]);
                    k:= k-1; SiftDown(L,k)
                end
            end
        end
    end;
end;
```

Attīstītie šķirošanas algoritmi (8)

Kaudzes šķirošanas algoritma veikspēja, ka elementu secība ir visneoptimālākā, ir **$O(n \log_2 n)$** .

Veidojot kaudzi, elements ar vismazāko atslēgu tiks izvietots tajā kā $el[1]$. Problēma ir sameklēt nākamo elementu ar otro mazāko atslēgu un izvietot to kaudzē kā $el[2]$.

390	205	182	45	235	- sašķirojamie
elementi					
45	205	182	390	235	- kaudze

Pirmajā posmā apmaina vietām $el[1]$ un $el[n]$ un tad elementu $el[1]$ pārbīda lejup (SiftDown) secībā $el[1], \dots, el[n-1]$ tā, lai $el[1], \dots, el[n-1]$ veidotu kaudzi.

182	205	235	390	45	- sašķirots 1
elements					

Otrajā posmā apmaina vietām elementus $el[1]$ un $el[n-1]$ un tad $el[1]$ pārbīda lejup secībā $el[1], \dots, el[n-2]$, izveidojot kaudzi ar $n-2$ elementiem. Šai gadījumā elementi $el[n-1]$ un $el[n]$ jau veido sašķirotu sarakstu ar garumu 2, bet $el[1]$ ir trešais lielākais elements.

205	390	235	182	45	- sašķiroti 2
elementi					

Šādā veidā i -ajā posmā apmaina vietām $el[1]$ un $el[n-(i-1)]$ un $el[1]$ pārbīda lejup secībā $el[1], \dots, el[n-i]$, izveidojot kaudzi ar $n-i$ elementiem. Elementu secība $el[n-(i-1)], \dots, el[n]$ ir sašķirotā un $el[1]$ ir i -

Knuta-Morisa-Prata (Knuth-Morris-Pratt) algoritms (1977) (1)

Tas ir virknes meklēšanas algoritms (string search algorithm), kas paredzēts, lai rakstzīmju virknē **S** (tekstā) sameklētu apakšvirkni **W** (šablonu, vārdu). To var uzskatīt par lineārās meklēšanas algoritma uzlabotu modifikāciju. Meklēšanas procesā tiek analizētas situācijas, lai noteiktu, ar kuru pozīciju sākas šablona teksts un par cik pozīcijām šablons tekstā jāpārbīda pa labi, lai sakritības pārbaude (match) būtu sekmīga.

Katrā meklēšanas iterācijā tiek operēts ar 2 veseliem skaitļiem: **m** un **i**, kur **m** ir pozīcija tekstā **S**, sākot ar kuru notiek sakritības pārbaude, bet **i** – pozīcija šablonā **W**, kurā pārbauda rakstzīmju pāra sakritību:

m: 12345678901234567890123 (teksta garums 23 rakstzīmes)

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

Knuta-Morisa-Prata (Knuth-Morris-Pratt) algoritms (1977) (2)

Sākot no 1.pozīcijas, notiek teksta un šablona sakritības pārbaude katram rakstzīmju pārim. Pirmajām 3 rakstzīmēm tā ir sekmīga, bet ceturtās rakstzīmes pārbaude ir nesekmīga ($S[4]$ ir tukšumzīme, bet $W[4]='D'$).

Var pamanīt, ka burts 'A' tekstā vēlreiz sastopams 5.pozīcijā, tāpēc uzdod, ka $m=5$, bet $i=1$:

m: 12345678901234567890123

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: 1234567

Otrajā meklēšanas iterācijā nesakritība tiks konstatēta rakstzīmju pārim $S[11]$ un $W[7]$. Taču šai meklēšanas iterācijā var konstatēt arī, ka pirms nesakritības fragments 'AB' bija gan tekstā, gan šablonā, bez tam ar šo fragmentu sākas nākamās iespējamās pārbaudes pozīcija 12. Tātad šo divu rakstzīmju pāru sakritību vairs var nepārbaudīt un nākamo iterāciju sākt situācijā $m=12$, $i=3$:

Knuta-Morisa-Prata (Knuth-Morris-Pratt) algoritms (1977) (3)

m: 12345678901234567890123
S: ABC ABCDAB ABCDABCDABDE
W: ABCDABD
i: 1234567

Šī salīdzināšana uzreiz ir nesekmīga ($S[14]$ ir tukšumzīme, bet $W[3] = 'C'$), tāpēc nākamo meklēšanas iterāciju sāk situācijā $m=15$, $i=1$:

m: 12345678901234567890123
S: ABC ABCDAB ABCDABCDABDE
W: ABCDABD
i: 1234567

Ceturtajā meklēšanas iterācijā tekstā un šablonā ir vienādi fragmenti 'ABCDAB ', taču nākamais rakstzīmju pāris 'C' un 'D' nesakrīt. Vadoties no tiem pašiem apsvērumiem kā iepriekš, uzdod, ka nākamajā meklēšanas iterācijā $m=16$, bet $i=3$:

m: 12345678901234567890123
S: ABC ABCDAB ABCDABCDABDE
W: ABCDABD
i: 1234567

Piektā meklēšanas iterācijā beidzas sekmīgi, meklēšanas rezultāts ir **pozīcija 16**.

Knuta-Morisa-Prata (Knuth-Morris-Pratt) algoritms (1977) (4)

```

function KMP (S,W: String; var pos:StringPos): StringLen;
{Rakstzīmju virknē S^, sākot ar pozīciju pos, meklē apakšvirkni W. Meklēšanas
rezultāts ir
tā pozīcija, sākot ar kuru apakšvirkne sameklēta, vai arī 0, ja meklēšana
beigusies}
nesekmīgi}
var m, i: StringPos;
    found: boolean;
    T: array[Position] of StringLen;
procedure KMP_Table (W: String);
{Nosaka šablonā W pārbīdes attālumu T[i] dažādās meklēšanas situācijās}
var i: StringPos;
    {aprēķināmā vektora T elementa
    pozīcija}
    j: StringPos;
    {šablona W elementa
    pozīcija}
begin
    j:=1;
    {sākumvērtības}
    i:=3; T[1]:= 0; T[2]:= 1;
    while i <= Length(W) do
    begin
        if W[i - 1] = W[j] then
            begin
                {šablons
                turpinās}
                T[i]:= j + 1; i:= i + 1; j:= j + 1
            end
        else if j > 1 then j:= T[j]
        {šablons neturpinās, nākas
        atkārties}
    end
end

```

Knuta-Morisa-Prata (Knuth-Morris-Pratt) algoritms (1977) (5)

```

begin                                     {procedūras KMP darbības sfēras
sākums}
    m:= pos;    i:= 1;
    found:= false;    KMP:= 0;
    KMP_Table(W);                                {izveido pirmos vektora T
elementus}
    while (not found) and (m+i <= Length(S)) do
        begin
            if W[i] = S[m+i] then i:= i+1
            else if i = Length(W) then
                begin
                    RMT:= m;    found:= true
                end
            else begin
                m:= m+i-T[i];
                if i>1 then i:= T[i]
            end
        end
    end;
end;
```

Vektoru T sauc arī par neveiksmes funkciju (failure function). Vektora T izveidošanas algoritma veikspēja ir $O(n)$, kur n - šablona W garums.

Knuta-Morisa-Prata meklēšanas algoritma veikspēja ir $O(k+n)$, kur k - meklēšanas iterāciju skaits.

Boijera-Mūra (Boyer-Moore) algoritms (1977) (1)

Tas ir rakstzīmju virknes meklēšanas algoritms, kas balstās uz atjautīgu secinājumu, ka daudz efektīvāk ir virknes sakritības pārbaudi sākt no šablona labējā gala pozīcijas un šai procesā virzīties uz pozīcijām šablona sākumā. Ja virknes S teksta un šablona W sakritības pārbaude ir nesekmīga, tad tiek noteikts, par cik pozīcijām šablons jāpārvieta pa labi, lai nākamajā pārbaudes iterācijā meklēšana būtu sekmīga, ja vien virknes teksts satur šablonu kā apakšvirkni. Piemēram:

m: 12345678901234567890123 (teksta garums 23 rakstzīmes)

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

Virknes meklēšanu uzsākot, tiek veidots vektors D (jump table), kurā fiksē katrai šablona rakstzīmei atbilstošu meklēšanas soļa vērtību.

Pirmajā pārbaudes iterācijā pārbauda divu rakstzīmju sakritību 7.pozīcijā, un šī pārbaude ir nesekmīga. Vektorā D izvēlas tieši tādu meklēšanas soli, vienādu ar 8, lai nākamā meklēšanas iterācija beigtos sekmīgi:

m: 12345678901234567890123

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

Otrajā meklēšanas iterācijā tiek konstatēta virknes teksta un šablona sakritība, meklēšanas rezultāts būs pozīcija 16.

Sakritības pārbaudei tiek izmantota speciāla funkcija Match.

Boijera-Mūra algoritma veikspēja ir $O(1)$.


```

begin                                     {darbības sfēras
sākums}

    found := false;                       {uzdod
sācumvērtības}

    len1 := Length(S);
    len2 := Length(W);
    kbegin := pos+len2-1;
    Preprocess(W, len2);                 {vektorā D ieraksta meklēšanas soļa
vērtības}

    while (not found) and (kbegin <= len1) do
    {meklēšana}
        if Match(S, W, kbegin-len2+1) then found := true
    {atrasta}
    else kbegin := kbegin+D[S^.data[kbegin]]

```


Boijera-Mūra (Boyer-Moore) algoritms (1977) (3)

```

function Match (S, W: String; pos: StringPos): boolean;
{Pārbauda, sākot ar pozīciju pos, vai rakstzīmju virkne S^ satur
apakšvirkni W^}
var i, last: StringPos;
    continue: boolean;
begin
    i:= 1;  last := Length(W);                                {uzdod pārbaudes
diapazonu}
    continue := true;    Match := false;
{sākumvērtības}
    if (not Empty(W)) and (Length(S) >= Length(W)+pos-1) then
        while continue and (S^.data[i] = W^.data[pos+len2-1]) do
            {sakritības
pārbaude}
            if i = last then
                begin                                           {pārbaude beidzas
sekmīgi}
                    continue :=false;    Match :=true
                end
            else
                begin
                    {pārbaude
                    turpināšana}
                    i:= i + Length(W) - W^.data[pos+len2-1];
                    continue := true;
                end
            end
        end
    end
    Match
end

```