Міністерство освіти і науки України
Львівський національний університет імені Івана Франка

О. П. Завада

АЛГОРИТМІЗАЦІЯ І ПРОГРАМУВАННЯ

Тексти лекцій

Львів
Видавничий центр ЛНУ імені Івана Франка
2004

Завада О. П. Алгоритмізація і програмування: Тексти лекцій. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2004. - 76 с.

У текстах лекцій розглянуто основи алгоритмізації процедур обробки економічної інформації та питання програмування таких процедур на мовах Pascal, VBA, C++. Курс має на меті сформувати науково обґрунтовані погляди на сучасні технології в інформатиці, виробити практичні навички з програмування економічних задач.

Для студентів спеціальності вищих навчальних закладів, що навчаються за спеціальністю ”Економічна кібернетика”.

ББК 32.97

Відповідальний за випуск: В. Є. Юринець

Комп’ютерний набір: О. П. Завада

Рекомендовано до друку кафедрою
інформаційних систем у менеджменті
Протокол № 19 від 25 червня 2004 р.

Вступ

Курс ”Алгоритмізація і програмування” згідно програми вивчається студентами спеціальності ”Економічна кібернетика” на другому курсі. Кількість лекційних занять становить 17 год. Дисципліна ”Алгоритмізація і програмування” опирається на знання, набуті в курсі „Інформатика та комп’ютерна техніка” і передує предметам „Технологія програмування”, „Об’єктно-орієнтоване програмування”.

Метою вивчення дисципліни є отримання студентами фундаментальних знань з теорії алгоритмів; з алгоритмізації процедур обробки економічної інформації; набуття практичних навичок із програмування таких процедур.

Кваліфіковане володіння сучасними програмними пакетами (бухгалтерськими, статистичними системами фінансових розрахунків тощо) вимагає вміння формувати конкретні конфігурації пакетів, програмуючи конкретні умови. При використанні навіть таких стандартних систем як Excel та Access постійно виникає потреба в написанні макросів, котрі є програмами на алгоритмічній мові. Практичне використання економіко-математичних моделей також неможливе без алгоритмізації та наступного програмування.

Формування пакетів прикладних програм, розробка макросів, комп’ютерна реалізація економіко-математичних методів – це сфера діяльності спеціалістів з економічної кібернетики (у першу чергу, фахівців спеціалізації „Інформаційні системи в менеджменті”).

Центральна увага в даних лекціях присвячена основним поняттям теорії алгоритмів та алгоритмізації процедур обробки економічної інформації.

В текстах лекцій автор намагається переконати читача в тому, що різні мови програмування мають між собою більше спільного, ніж відмінного. З цієї позиції розглядаються лише основні конструкції мов Pascal, C++ та VBA і виявляється спільне між ними. З цієї ж точки зору в даних лекціях не розглядаються такі системи як Delphi, Borland C++ Builder тощо. Вивчення цих систем є метою інших дисциплін.

У лекціях наводиться широкий набір типових алгоритмів обробки економічної інформації, а також набір учбових програм по вивченню основних конструкцій мов Pascal, C++ та VBA.

Матеріал лекцій може бути корисний також студентам інших економічних спеціальностей та студентам спеціальності „Прикладна математика”.

Розділ 1. Основи програмування на алгоритмічних мовах

1.1. Теоретичні основи алгоритмізації

1.1.1. Принципові можливості алгоритмізації та комп'ютерної техніки

Перша у світі електронно-обчислювальна машина (ЕОМ) була створена в 1946 році в США групою вчених під керівництвом Дж. фон Неймана. Її робота базувалася на чотирьох основних принципах:

- принцип двійковості. Як інформаційна частина (числа, текст), так і керуюча (команди) представляються на машинному рівні у двійковій системі;
- принцип адресності. Усі дії виконуються над вмістом певних адрес. Команди також розміщені за адресами;
- принцип програмного управління. Від початку до кінця процес обчислень здійснюється під управлінням програми (набору команд);
- принцип переадресації. Над командами виконуються арифметичні дії, як і над даними. Такі сформовані команди мають змогу виконувати операції над послідовністю даних.

Ці принципи дотепер називають принципами Неймана. Хоча основні ідеї побудови й функціонування ЕОМ були висунуті ще в 1890-х роках англійським ученим Ч. Беббіджем. В 1930-х роках були сформульовані (А.Тьюрінг та ін.) принципи інформатики, які чітко окреслювали можливості майбутніх ЕОМ. Отже, приступаючи в кінці другої світової війни до створення першої ЕОМ, учені чітко усвідомлювали можливості цієї майбутньої для них техніки. Зокрема, один з основних принципів інформатики стверджує:

Будь-яка ЕОМ, побудована на принципах фон Неймана, яка має в складі своїх команд арифметичні дії, пересилку (присвоєння) та розгалуження є універсальною (алгоритмічно повною) у тому сенсі, що за допомогою цих команд можна в принципі виконати довільний алгоритм.

Звичайно, для виконання конкретного алгоритму на конкретній техніці може не вистарчити машинних ресурсів (пам'яті чи часу). Проте алгоритмічна повнота має місце принципово. Сучасні комп'ютери теж функціонують на принципах фон Неймана, отже поняття універсальності відноситься і до них.

Проте, звичайно, далеко не всі задачі можна в принципі розв'язати на комп'ютері. З науково-технічним прогресом усе нові й нові задачі підключаються до розв'язання на комп'ютерах. Проте існують і такі задачі, які в принципі ніколи на комп'ютерах розв'язані не будуть (як не буде знайдено

універсального розчинника, вічного двигуна, ліки від усіх хвороб,...). Уперше ряд чітко сформульованих математичних задач, для яких у принципі не існує алгоритму їхнього розв'язування (а, отже, і розв'язування за допомогою комп'ютера), було виявлено в 1930-х роках. При цьому неіснування таких алгоритмів було строго математично доведено (доведення у вигляді теорем). Наведемо дві з таких теорем.

Теорема Райса (Rice). Не існує алгоритму (в принципі), який по тексту програми та по вхідних даних зміг би установити, чи відбудеться зациклення.

Звичайно, ця теорема нами сформульована в перефразованому вигляді, оскільки у 1930-х роках ще не існувало термінів ”програма”, ”вхідні дані”, ”зациклення”.

Теорема Черча (Church). Не існує алгоритму, який би по заданих аксіомах, правилах виведення та деякому твердженню зміг би встановити, чи це твердження виводиться з аксіом, чи ні.

Алгоритм, його властивості

Під алгоритмом розуміють скінчену систему правил для розв'язування певного класу задач, яка задовольняє таким властивостям:

- масовість. Алгоритм – це не знаходження розв'язку однієї конкретної задачі, а єдиний спосіб для розв'язання класу задач. Як правило, множина вхідних даних алгоритму є нескінченною;
- результативність. Після скінченої кількості кроків алгоритм повинен видати певний результат (навіть такий результат, як ”я не можу знайти розв'язку”);
- детермінованість. Хто б не виконував алгоритм і коли б його не виконував, він при одних і тих же вхідних даних завжди повинен отримувати той самий результат.

Остання властивість виражає основну суть алгоритму: виконавець може не розуміти змісту. Він повинен лише вірно виконувати запропоновану йому послідовність дій.

Розглядають три основні способи задання алгоритмів:

- вербальний (словесний);
- за допомогою блок-схем;
- за допомогою програми на алгоритмічній мові.

При цьому словесний спосіб аж ніяк не є легшим, ніж написання програми. Справді, алгоритм (при будь-якому заданні) повинен бути настільки точно описаний, щоб кожен виконавець однозначно розумів усі інструкції алгоритму (без присутності автора алгоритму як консультанта).

Як твердить принцип інформатики, для запису будь-якого алгоритму достатньо наявності арифметичних команд, операції розгалуження та операції пересилки. Усі алгоритмічні мови (BASIC, PASCAL, C++ тощо) ці операції включають. Отже, із позиції універсальності усі мови програмування рівнопотужні. Проте з позиції реального написання й налагодження програм дуже суттєвою є і методика написання цих програм.

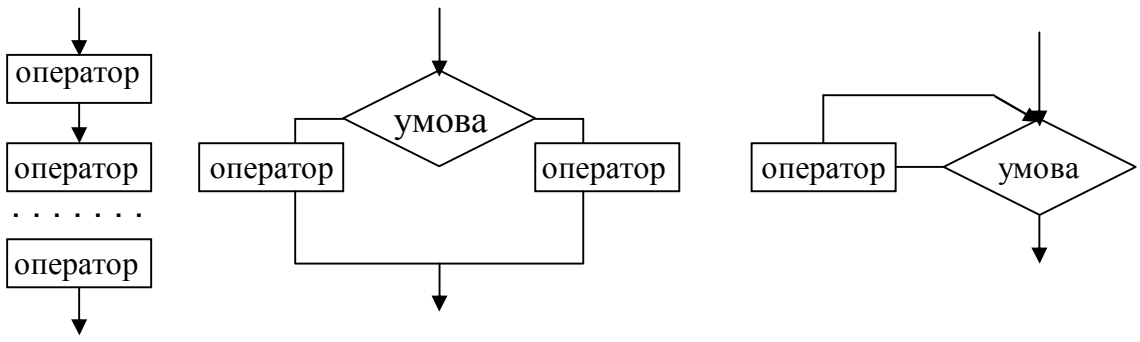
Основна трудність при розробці великих програм (як і будь-якої інтелектуальної діяльності) полягає в тому, що людина не може одночасно тримати в голові більше, ніж 7±2 об'єкти. Отже, блок-схему програми з 10 і більше блоками уже неможливо зрозуміти. Майстерність програміста (як і письменника, науковця) полягає у вмінні розбити одну велику задачу на ряд простіших, які можна розв'язувати незалежно в часі (або одночасно різними людьми). Зокрема, бажано так будувати блок-схему (чи текст програми), щоб у кожен момент часу концентрувати свою увагу на невеликому фрагменті блок-схеми (не більше 7 блоків) чи програми.

Однією з методик розробки програм, яка дозволяє поетапно будувати програму, є структурне програмування.

Теоретичною основою застосування структурного програмування є

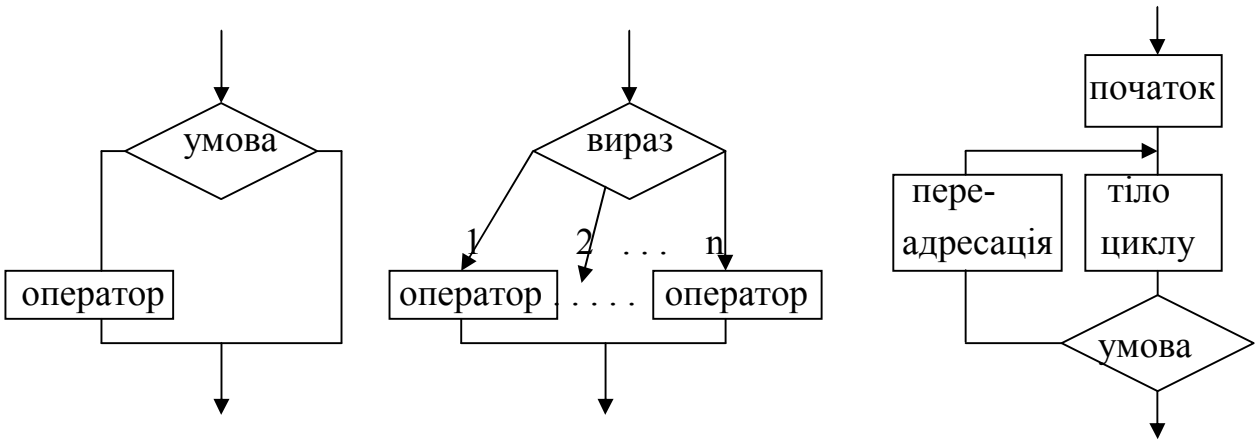
Теорема Бома-Джакопіні.

Для того щоб скласти блок-схему довільного алгоритму, достатньо трьох конструкцій: складеного оператора типу BEGIN_END, оператора розгалуження типу IF_THEN_ELSE та оператора циклу типу WHILE_DO:



begin оператор;... if умова then оператор while умова
...;оператор end else оператор do оператор

Для зручності на практиці використовують ще три такі конструкції:



If умова
Then оператор

Case вираз Of
1: оператор;
. . . . . .
n: оператор
end

FOR змінна=початкове
значення
To кінцеве значення
Do оператор

Структурне програмування рекомендує:

1. Використовувати всі шість конструкцій природнім способом (не програмувати циклічний процес оператором розгалуження тощо);
2. Не використовувати жодних інших керуючих конструкцій (зокрема, оператора безумовного переходу GOTO).

Ряд систем програмування (PASCAL, C++, шкільна Алгоритмічна Мова) мають оператори, що відповідають усім шести конструкціям. Отже, в цих системах природно використовувати структурне програмування. Структурні програми є набагато зрозуміліші і наглядніші, ніж неструктурні. Великі програмні розробки можна виконати тільки за допомогою технології структурного програмування.

Деякі системи, наприклад, FOXPRO взагалі не мають оператора GOTO. На них, отже, можна писати тільки структурні програми.

Проте існують системи (мова BASIC ранніх версій, зокрема, GW-BASIC, які іноді ще досі викладається в середніх школах), що не містять складеного оператора або інших конструкцій зі структурного набору. При використанні таких систем програміст змушений систематично звертатися до оператора GOTO. Дуже швидко програма стає зовсім незрозумілою. Розробити таким чином програму хоча б середньої складності неможливо. Неструктурні мови програмування тому є лише учбовими.

1.1.2. Основи програмування на алгоритмічних мовах

Основні конструкції алгоритмічних мов (описи змінних, оператори розгалуження та циклу, процедури обробки файлів, схема заміни формальних параметрів на фактичні тощо) відрізняються лише синтаксисом, а за змістом є рівносильними.

Таким чином програми одного й того ж алгоритму будуть подібними на себе незалежно від мови програмування.

Для того, щоб отримати правильну і зрозумілу програму рекомендується не використовувати без потреби специфічних засобів програмування та різних програмістських ”трюків”.

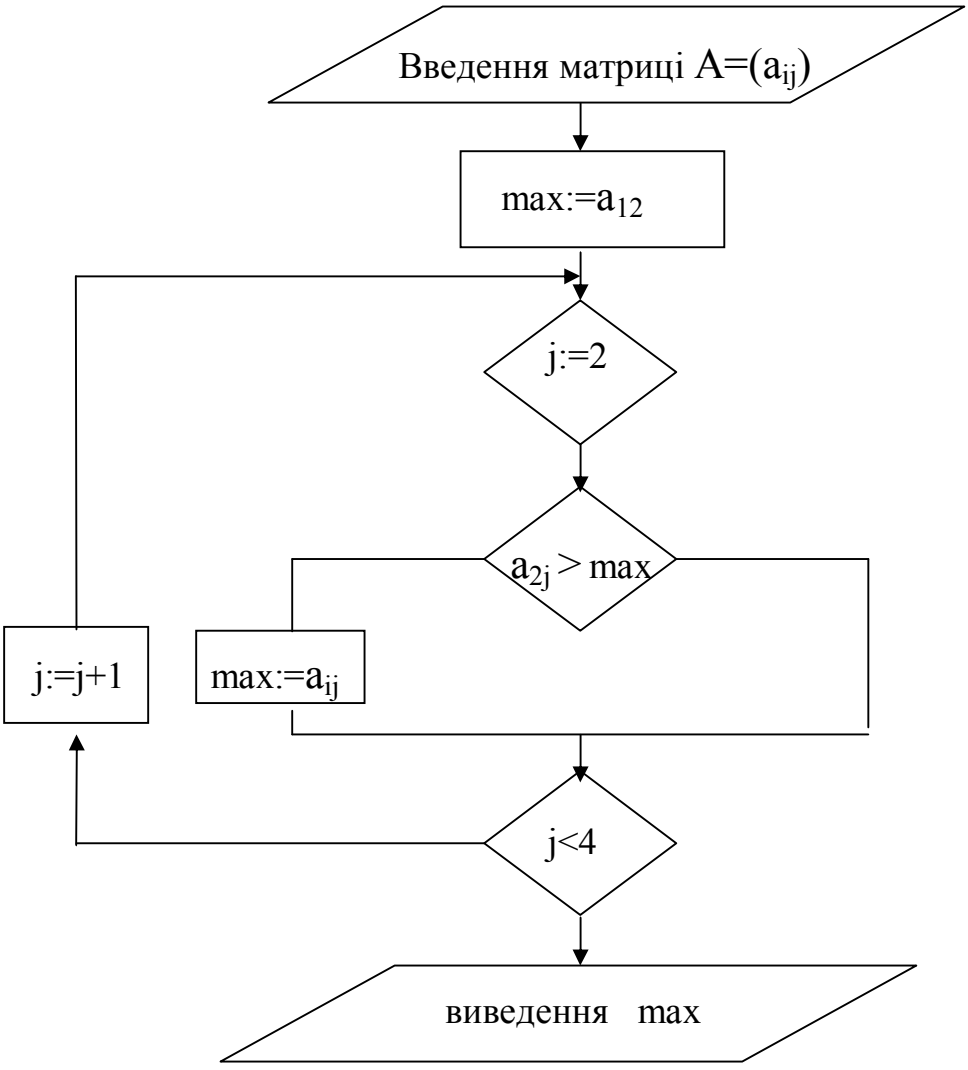
Приклад. Задана матриця $A=(a_{ij})_{i=1,...,3;j=1,...,4}$. На мові PASCAL скласти програму, яка знаходить найбільший елемент у другому рядку цієї матриці.

```
Program P;
Uses Crt;
Const  m=3; n=4;
Var
   A: array[1..n,1..m] of real;
    max: real;  i,j: integer;
Begin
   Clrscr;
   {ввід вхідних даних}
    For i:=1 to n do
    For j:=1 to m do
      Begin
        GoToXY(2+i,j*20-18);  Write('a(',i:1,',',j:1,') =');  Read(a[I,j])
      End;
max:=a[2,1];
For j:=2 to n do
   If A[2,j]>max then max:=A[2,j];
Writeln;
Writeln('найбільший елемент другого рядка дорівнює ', max:6:2);
Repeat Until KeyPressed
End.
```

У програмі P реалізовано виведення результату на екран та введення вхідних даних із клавіатури за допомогою модуля Crt. Конструкція GoToXY(i,j) встановлює курсор на i-у позицію j-ого рядка. Тіло циклу Repeat Until KeyPressed складається лише з пустого оператора. Таким чином цей цикл виконує зупинку роботи програми до тих пір, поки не буде натиснена довільна клавіша.

Кожен програміст завжди повинен могти виконати за текстом програми (своєї чи чужої) її граматичний аналіз, прокрутку на конкретних вхідних даних, а також побудувати блок-схему алгоритму. Тільки вміння виконувати (можливо, усно) вказані три процедури дозволяє самостійно виявляти та усувати помилки та описки в програмі і досягати написання абсолютно правильної програми.

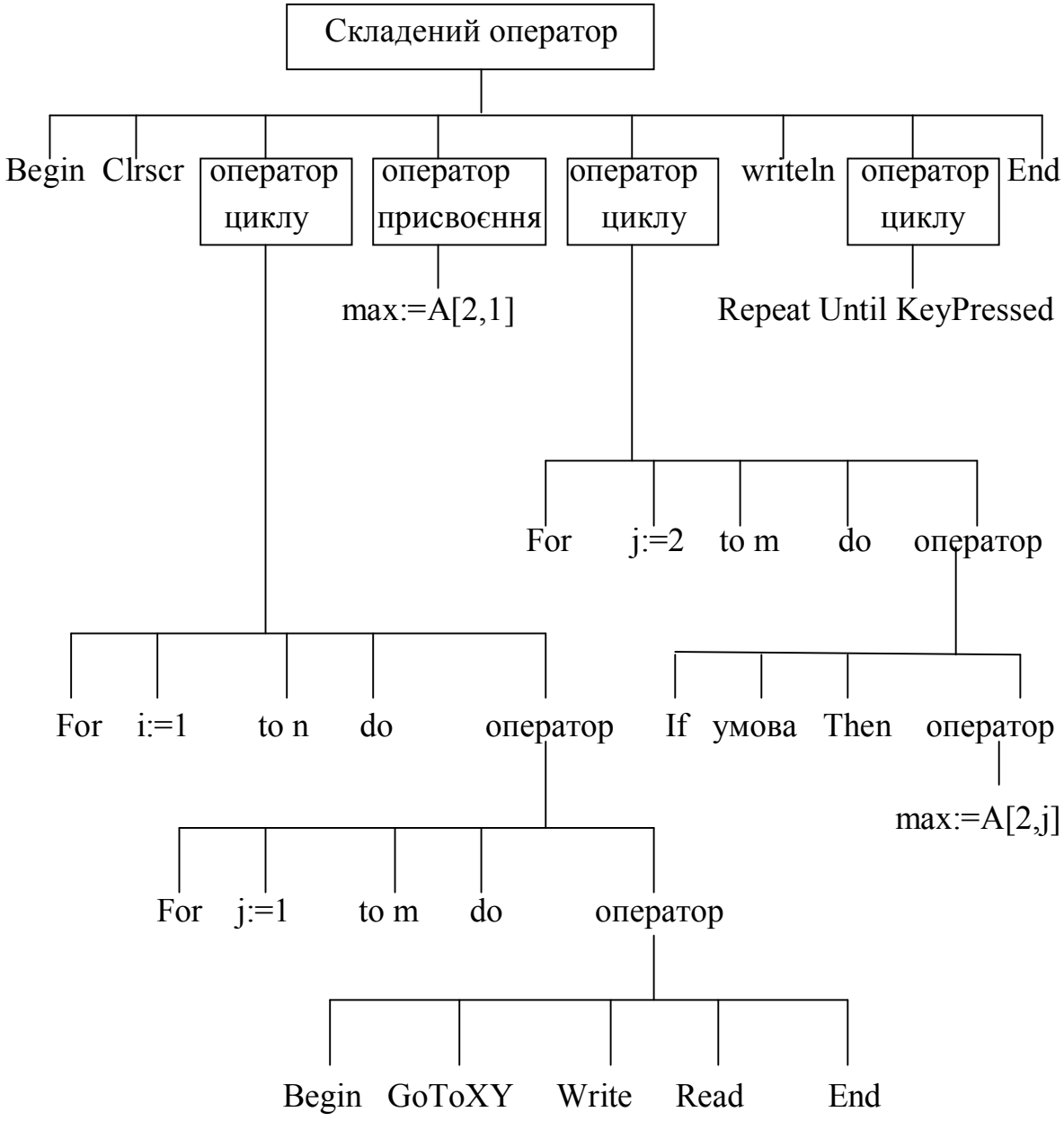
Блок-схема алгоритму (незалежно від того, чи цей алгоритм було реалізовано засобами мови PASCAL, C++ чи VBA) є такою:



Варто нагадати, що блок-схему довільного алгоритму завжди можна побудувати лише з невеликої кількості стандартних керуючих конструкцій (конструкцій розгалуження, циклу тощо) .

Граматичний аналіз, на противагу блок-схемі, суттєво залежить від алгоритмічної мови. Для здійснення граматичного аналізу необхідно детально знати конкретний вигляд операторів конкретної мови програмування і навіть її конкретної версії.

Граматичний аналіз PASCAL–програми P є таким:



І, нарешті, виконаємо прокрутку цієї програми для таких вхідних даних:

A[1,1]:=4	A[1,2]:=2	A[1,3]:=2	A[1,4]:=1
A[2,1]:=1	A[2,2]:=3	A[2,3]:=4	A[2,4]:=3
A[3,1]:=2	A[3,2]:=2	A[3,3]:=5	A[3,4]:=1

З метою налагодження програми блок введення даних з клавіатури тимчасово можна замінити набором таких операторів присвоєння.

Маємо (згідно блок-схеми) таку послідовність дій:

max:=1
j:=2
A_{22}>max? (3>1?) Так
max:=3
j<4? Так
j:=3
A_{23}>max? (4>3?) Так
max:=4
j<4? Так
j:=4
A_{24}> max? (3>4?) Ні
j<4? Ні
Виведення результату max=4

Звичайно, в теперішній час програміст не будує блок-схем перед написанням тексту програми. Він відразу пише структурні програми.

Програміст також не виконує на папері граматичний аналіз та прокрутку. Він виконує всі ці дії „в умі”, часто на рівні підсвідомості.

Кожна людина у своїй діяльності допускає описи. Так, друкарка високої кваліфікації має право допустити дві описки на сторінку. Кваліфікований програміст робить в середньому одну описку на 25 – 50 операторів. Кінцевий програмний продукт, проте, не має права містити цих описок.

При потребі виконати аналіз фрагменту програми, у якому потрібно знайти і виправити помилку ми завжди мусимо (усно чи письмово) виконати як граматичний аналіз операторів, так і прокрутку.

Ще однієї вимогою до програмування є використання коментарів. При відсутності коментарів сам автор програми дуже швидко перестає її розуміти.

1.2. Основні конструкції мови PASCAL

1.2.1. Типи даних в мові PASCAL

Загальна структура PASCAL-програми є такою:
Заголовок (Program ім‘я програми)
Розділ модулів (Uses)
Розділ міток (Label)
Розділ констант (Const)
Розділ типів (Type)
Розділ описів змінних (Var)
Розділ процедур та функцій (Procedure, Function)
Розділ операторів (Begin оператори, які розділяються символом ”;” End.)
Деякі з цих розділів можна переставляти місцями.

З кожною змінною програми пов’язаний певний тип. Тип змінної задає
- розмір пам’яті, що займає ця змінна;
- спосіб машинного представлення цієї змінної;
- множину допустимих значень цієї змінної;
- набір допустимих операцій над змінною.

В мові Pascal всі змінні повинні бути описані у розділі Var.
Основними арифметичними типами даних у мові Pascal є:

Тип змінної	Ключове слово	Діапазон значень	Обсяг пам’яті, байт
Довге ціле число	longint	-2147483648 .. 2147483647	4
Ціле число	integer	-32768 .. 32767	2
Коротке ціле число	shortint	-128 .. 127	1
Дійсне число	real	$-2{,}9 * 10^{-39}$ .. $1{,}7 * 10^{38}$	6
Скорочене дійсне число	single	$-1{,}5 * 10^{-45}$ .. $3{,}4 * 10^{308}$	4

Над цілими та дійсними числами виконують арифметичні операції та операції порівняння <, >, <=, >=, <>, =. Функція round виконує заокруглення дійсного числа до найближчого цілого. Функція trunc здійснює відкидання дробової частини числа.

Опис змінних логічного типу має вигляд

```
Var
   змінна, змінна: Boolean;
```

Логічні змінні (висловлення) можуть приймати одне з двох значень: True (істинне) або False (хибне). Над логічними змінними в мові Pascal виконуються три операції:

- And (кон'юнкція, логічне множення, логічне "і");
- Or (диз'юнкція, логічне додавання, логічне "або");
- Not (заперечення).

Значеннями символьного (літерного) типу є елементи скінченої множини символів (наприклад, набору символів кодової таблиці RUSCII).

Опис рядкового (стрінгового, текстового) типу задаються так:

```
Var
    змінна : String[довжина рядка];
```

Значення символьних та рядкових змінних беруться в одинарні лапки: 'a', '+', 'інформатика'. Над рядковими змінними можна виконувати операції присвоєння, введення та виведення (R1:=R2; read(R1); write(R2)). Доступ до окремого символу рядка R здійснюється за допомогою квадратних дужок: R[i].

Над змінним рядкового типу можна також виконувати операцію приписування (R:='Екі'+'21').

Приклад.

```
Const
     N=10;
     Pi=3,141592; {коментар: тип константи визначається за її виглядом}
Var
    a,b,c: Real; x1,x1:Real; k:integer;
    S, S1, S2: string;
    C: char;
    b1, b2:Boolean;
Begin
   . . . . . .
c:=a+k; S:=S1+'_'+S2; C:=S[3];
b1:= (a<b) and (b<c)
End.
```

1.2.2. Основні керуючі оператори в мові PASCAL

Оператор присвоєння в мові PASCAL має вигляд:

ім’я змінної := вираз

При цьому тип змінної має співпадати з типом виразу, або бути сумісним із ним (якщо змінна має дійсний тип, а значення виразу є цілим числом, то такий оператор присвоєння є допустимим; у протилежному випадку – недопустимим).

Схеми операторів введення з клавіатури та виведення на екран є такими:

write(змінна, . . . , змінна)

Між операторами, а також між довільними двома конструкціями мови PASCAL ставиться крапка з комою.

Приклад. Обчислення площі трикутника за формулою Герона.

```
Program P1;
Var
  a,b,c,p,s: real;
  {обчислення площі трикутника за трьома сторонами}
Begin
  Writeln;
  Writeln (‘введіть довжини сторін трикутника:’);
  read(a,b,c);
  p:=(a+b+c)/2; s:=Sqrt(p*(p-a)*(p-b)*(p-c));
  write(‘площа трикутника із сторонами’);
  writeln(a:5;2,b:5:2,c:5:2,’ дорівнює ’,s:6:2)
End.
```

Загальний вигляд оператора виведення Write має вигляд:

Write(x:n) або Write(x:n:m)

Цей оператор дуже просто реалізує форматоване виведення. Параметр n вказує на кількість позицій на екрані, які будуть відведені під змінну *x*, а параметр m – на кількість дробових цифр в дійсному числі. При потребі текст (рядкова змінна) доповнюється пробілами справа, а число - пробілами зліва.

Оператор Writeln здійснює перехід на новий рядок, а оператор Writeln(змінні) спершу здійснює виведення вказаних змінних і потім виконує перехід на новий рядок.

Основними керуючими операторами є:

- складений оператор (begin_end);
- оператори розгалуження (if_then та if_then_else);
- оператор варіанта (множинного вибору, case_of);
- оператори циклу (For , While та Repeat);

Група операторів всередині операторних дужок begin_end розглядається мовою PASCAL як один оператор – складений оператор.

Оператори розгалуження (умовні оператори) мають такий вигляд:

If умова Then оператор

If умова Then оператор_1 Else оператор_2

Приклад. При 0<a<5 обчислити x=1 та y=2, інакше обчислити z=3 та t=4.

```
if a>0 and a<5 then
            begin
                x:=1;
                y:=2
            end
       else
             begin  z:=3; t:=4 end
```

Синтаксис оператора варіанта є таким:

```
Case   вираз   of
     варіант_1:  оператор_1;
     варіант_2:  оператор_2;
 . . . . . .
     варіант_n:  оператор_n
      else          оператор_n+1
end
```

Приклад.

```
Case  k  of
      1,2: write('не атестований');
      3,4,5: write('атестований')
      else
         write('невірна оцінка')
end
```

Конструкція else в операторі Case є необов'язковою.

Розрізняють оператори циклу трьох типів: For, While та Repeat.

Схема оператора While така:

While умова (логічний вираз) Do оператор

Оператор в тілі циклу виконується до тих пір, поки умова є істинною (поки значенням логічного виразу є True).

Приклад. Обчислити із заданою точністю є значення y=sin(x) за формулою

$$Y = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \ldots$$

. . . . . .

```
read(x,eps);
y:=x; u:=x; r:=-x*x; i:=2;
while abs(u)>eps do
begin
   u:=(u*r)/(i*(i+1));
   y:=y+u;
   i:=i+2
end;
```

Для кращого розуміння рекомендується виконати прокрутку цієї програми.

Синтаксис оператора циклу Repeat є таким:

Repeat оператор ;…; оператор Until умова (логічний вираз)

Група операторів у тілі циклу виконується до тих пір, поки умова ще не стала істиною.

Приклад. Обчислити за формулою у попередньому прикладі значення y=sin(x). Використати оператор Repeat.

. . . . . .

```
read(x,eps);
y:=x; u:=x; r:=-x*x; i:=2;
Repeat
   u:=(u*r)/(i*(i+1));
   y:=y+u;
   i:=i+2
Until abs(u)<eps;
```

Означення оператора For (циклу з параметром, циклу з наперед відомою кількістю повторень) є таким:

For змінна := початкове значення To кінцеве значення Do оператор

Приклад. Обчислити $Y = \sum_{i=1}^{n-1} i^2$.

```
…; y:=0;  For i:=1 to n-1 do y:=y+i*i; …
```

Довільний алгоритм завжди можна реалізувати при допомозі лише розглянутих вище керуючих конструкцій. При цьому рекомендується використовувати структурне програмування [7]. З цих міркувань оператор GOTO <мітка>та розділ Label в даних вказівках не розглядаються.

Загальна схема опису змінних регулярного типу (масивів, змінних типу array) у мові PASCAL є такою:

Type ім’я типу = Array [тип індексів] of тип компонент

Наприклад, вектор (одновимірний масив) $A=(a_1,\ldots,a_n)$ можна описати одним із двох способів:

```
  Type
     T1=array [1..n] of real;
  Var A:T1;
```

або

```
  Var   A: array [1..n] of real;
```

Тут типом індексів є множина цілих чисел у діапазоні від 1 до n.

Окремі елементи масиву A іменуються так: A[1], … ,A[i-2], …,A[n]. Дії є можливими лише з окремими елементами масиву.

Приклад. Знайти кількість та суму додатних елементів вектора $A=(a_1,\ldots,a_{100})$.

```
Var
  A: array[1..10] of real; s:real;  k, i: integer;
Begin
 . . . . . .
            k:=0; s:=0;
       For i:=1 to 100 do
             If a[i]>0 then
        begin  k:=k+1; s:=s+a[i] end
End.
```

1.2.3. Процедури та функції в мові PASCAL

Розробка великих програм є неможливою без розбиття її на окремі процедури, які створюються окремо (різними виконавцями в різний час).

Опис процедури інформує комп'ютерну систему про те, які дії потрібно буде виконати в момент виклику цієї процедури. Наказом на виконання цих дій є оператор процедури.

В мові PASCAL опис процедури має такий вигляд:

```
Procedure  ім'я_процедури (описи формальних параметрів);
    Var   описи локальних змінних;
    Begin
      оператори (тіло процедури)
    End;
```

Оператор процедури є таким:

```
ім'я_процедури(перелік фактичних параметрів);
```

Кількість фактичних параметрів кожної процедури в мові PASCAL обов'язково повинна дорівнювати кількості формальних. Повинні співпадати також порядок розташування формальних і фактичних параметрів та їхні типи. Проте символічні імена відповідних параметрів зовсім не мусять співпадати. Цей факт дає змогу використовувати процедури, які були написані раніше в інших програмістських колективах. На практиці формують, а потім широко використовують так звані бібліотеки стандартних процедур.

Ще більш потужнім засобом, ніж механізм заміни формальних параметрів на фактичні є механізм реалізації локальних змінних.

Змінна, описана всередині процедури, є локальною в цій процедурі. ЇЇ позначення може співпадати з позначенням якоїсь іншої змінної в головній програмі, або в іншій процедурі. З машинної точки зору локальна змінна – це нова змінна, яка існує лише на час роботи процедури.

Приклад. Розробити процедуру m2 знаходження більшого числа серед двох заданих. Написати програму m3 знаходження найбільшого числа з-поміж трьох заданих чисел, використовуючи процедуру m2.

Введемо позначення для змінних. Аргументами програми m3 будуть три числа, для яких вибираємо символічні імена x, y та z. Результатом програми m3 буде змінна m. Програма буде також використовувати допоміжну змінну r.

Аргументи процедури m2 позначимо через a та b, а результат через c. Процедура також буде використовувати локальну (внутрішню) змінну x (колізії із змінною *x* в програмі m3 не виникне).

```
Program m3;
Var
    x,y,z:real; m:real; {аргументи та результат}
    r:real; {допоміжна змінна}
Procedure  m2(a,b:real; var c:real);
      var x:real;     {локальна для процедури змінна}
      begin
          if a> b then x:=a
                   else x:=b;
          c:=x
     end;
Begin
   Writeln('Введіть три числа');
   Read(x,y,z);
   m2(x,y,r);
   m2(r,z,m);
   writeln('найбільше число = ', m:8:2);
End.
```

Описи формальних параметрів *a,b:real; var c:real* виконані інакше для параметрів a та b та інакше для параметра c. Це пов'язано з тим, що змінні a та b – це параметри-значення (заміна цих формальних параметрів на фактичні відбувається за допомогою присвоєння значень). В той же час c – параметр-змінна (заміна цього параметру на відповідний фактичний відбувається заміною імені змінної).

Під час виконання (під час виклику) першого з операторів процедури m2(x,y,r) відбудуться такі дії:

a:=x; b:=y (відбувається присвоєння значень)

з'являється локальна для процедури змінна *x*

виконуються оператори

```
        if a> b then x:=a
                 else x:=b;
        r:=x    (відбувається заміна імені c на r )
```

локальна змінна *x* припиняє існування

У тому випадку, коли результатом роботи процедури є лише одна змінна, то замість конструкції Procedure часто використовують конструкцію Function. Опис функції має вигляд:

```
Function  ім‘я_функції(описи формальних параметрів): тип результату;
    Var   описи локальних змінних;
    Begin
      оператори (тіло функції)
    End;
```

Для імені функції та для імені її результату використовується те саме позначення. Покажчик функції має вигляд

ім‘я_функції(перелік фактичних параметрів)

Покажчик функції використовується у виразах на однакових правах з іншими змінними.

Приклад. Розробити функцію f2 знаходження більшого числа серед двох заданих. Написати програму mm3 знаходження найбільшого числа з-поміж трьох заданих чисел, використовуючи функцію f2.

```
Program mm3;
Var
   x,y,z:real; m:real; {аргументи та результат}
Function  f2(a,b:real):real;
     var x:real;     {локальна змінна}
     begin
        if a> b then x:=a  else x:=b;
        f2:=x
    end;
Begin
  Writeln(‘Введіть три числа’) Read(x,y,z);
  m:=f2(f2(x,y),z);
  writeln(‘найбільше число = ’, m:8:2);
End.
```

Набори процедур та функцій у мові Pascal об’єднуються в модулі. Наприклад, під'єднавши в розділі Uses модуль Crt, програміст отримує змогу використовувати такі стандартні процедури, як ClrScr – очистка екрану та GoToXY(x,y) – встановлення курсору на позицію x у рядку y.

Розглянемо ще один тип даних – комбінований тип, (запис, тип record).

Опис змінної типу record є таким:

ім‘я: Record змінна: тип; … ; змінна: тип End

Приклад.

```
x1,x2: record
         Name: string;
         Price: real;
         Qty: integer
       end
```

Доступ до окремих елементів запису здійснюється через символ “.”:

x1.name:=’телевізор’; x2.Qty:=x1.Qty+12;

1.2.4. Файли в мові PASCAL

Набори записів однакової структури утворюють послідовні файли (файли типу record):

```
Type
   Zap= record  Name: string;  Price: real;  Qty: integer end;
Var
    F: file of Zap;
```

або

```
Var
    F: file of record  Name: string;  Price: real;  Qty: integer end;
```

Над файлами типу record виконуються такі операції (процедури):

Assign(F,’шлях до набору даних’) – зв'язує символьне ім’я F у програмі з конкретним набором даних на зовнішньому носії;

Reset(F) та Rewrite(F) - підготовлюють файл до читання (до запису) з його початку; покажчик встановлюється на початок файлу;

Read(F,*x*) - зчитує в оперативну пам’ять *x* той запис файлу F, на який встановлено покажчик; покажчик при цьому пересувається на один запис далі;

Write(F,*x*) - записує один запис з оперативної пам’яті за адресою *x* в кінець файлу F;

Функція EOF(F) розпізнає, чи покажчик досяг кінця файлу.

Процедура Seek(F,k) встановлює покажчик на k-ий запис файлу F.

Приклад. Задано файл, кожен запис якого містить назву товару, його ціну та кількість. Підрахувати загальну вартість усіх товарів.

```
Program Pr3;
Type  Z= record  Name: string;  Price: real;  Qty: integer end;
Var  F: file of z; x:z;  .S;real;
Begin
Assign(f,'C:\f.dat'); reset (F);
          S:=0;
while not eof(F) do
    begin
       read(F,x);
       S:=S+x.Price*x.Qty
    end;
Writeln('Вартість = ', s:8:2)
End.
```

Файлові структури даних дуже широко застосовуються при обробці економічної інформації. По-перше, в одному записі змінні можуть мати різний тип. По-друге, кількість записів у файлі не мусить бути відомою.

В мові PASCAL існують файли, які не поділяються на записи: текстові файли (опис: file of text або text). Усі змінні (навіть числа) текстового файлу представлені в символьному вигляді. Тому оператори зчитування та запису в такий файл перераховують усі необхідні змінні. Перевага текстового файлу полягає в тому, що його можна переглядати найпростішими редакторами.

```
Program Pr4;
Var F: file of text; Name: string;  Price: real;  Qty: integer; S:real;
Begin
Assign(f,'C:\f.dat'); reset (F);
              S:=0;
while not eof(F) do
    begin
       read(F, Name, Price, Qty);
       S:=S+Price*Qty
    end;
Writeln('Вартість = ', s:8:2)
End.
```

1.3. Основні конструкції мови VBA

1.3.1. Типи даних в мові VBA

Мова Visual Basic for Application (VBA) широко застосовується в середовищі Microsoft Office.

Арифметичні типи даних мови VBA подамо таблицею:

Тип змінної	Ключове слово	Діапазон значень	Обсяг пам'яті, байт
Довге ціле число	long	-2147483648 .. 2147483647	4
Ціле число	integer	-32768 .. 32767	2
Дійсне число	single	$-1{,}4 * 10^{-45}$ .. $3{,}4 * 10^{38}$	4
Дійсне число подвійної точності	double	$-4{,}9 * 10^{-324}$ .. $1{,}7 * 10^{308}$	8

Текстові (символьні) рядки задаються, як і в мові PASCAL, конструкцією string.

Опис типу даних у VBA є таким:

DIM ім'я змінної As тип змінної

Якщо деяке значення не повинно в процесі виконання програми змінюватися, то його оголошують як константу оператором Const:

Приклад.

```
Const n=10
Const pi=3.141592
DIM a1, a2 as single
DIM i,j as integer,
      z,x,y as double
 DIM S,S1,S2 as string
```

Над цілими та дійсними типами мова VBA дозволяє виконувати арифметичні дії. Над змінним типу string можна виконувати операцію & (зчеплення, конкатенації, приписування). Нехай S1="бан" та S2="ка" Тоді результатом операції S1&S2 буде рядок (послідовність символів) "банка", а результатом операції S2&S1 - "кабан".

В мові VBA існує два способи завдання масивів:

DIM ім'я змінної (Nmax) As тип елементів

та

DIM ім'я змінної (Nmin to Nmax) As тип елементів

Приклад.

```
Dim a(5) as single
Dim m(1 to 3, 1 to 4) as single
```

У першому випадку задано вектор (одновимірний масив) з елементами a(o), a(1), a(2), a(3), a(4).

У другому випадку описано двовимірний масив (матрицю) з елементами m(1,1),…,m(3,4).

Мова VBA дозволяє використовувати також динамічні масиви. Кількість елементів такого масиву наперед невідома (наприклад, кількість відмінників на факультеті):

```
DIM A() as string
```

Після того, як програмним шляхом ця кількість відмінників знайдена (наприклад, в змінній n), то далі за допомогою оператора ReDim можна задати конкретні розміри цього масиву:

```
ReDim A(n)
```

1.3.2. Основні керуючі оператори в мові VBA

Замість символу ":=" в мові PASCAL оператор присвоєння в мові VBA використовує символ "=".

Найпростіші оператори введення та виведення (в текстовому режимі) мають такий вигляд:

Cls - очистка екрану;

Locate i,j - встановити курсор на j-у позицію і-ого рядка;

Input змінна - введення конкретного значення змінної з клавіатури;

Print змінна ; змінна ; змінна - виведення значень змінних на екран.

Як і в усіх інших мовах, мова VBA має два типи оператора розгалуження (умовного оператора):

```
If умова Then
    оператори
End if
```

та

```
If умова Then
    Оператори_1
 Else
    Оператори_2
End if
```

Приклад. Якщо x>3 , то обчислити a=a+1 та b=2 , інакше обчислити a=a+3 та c=4.

```
If x>3 then
   a=a+1
   b=2
Else
    a=a+3
    c=4
Endif
```

Оператор циклу типу For має такий синтаксис:

```
For лічильник=початкове_значення To кінцеве_значення [Step крок]
        оператори
Next лічильник
```

Конструкція [Step крок] при цьому є необов'язковою.

Приклад. Знайти суму a(n+3)+a(n+5)+…+a(n+13)

```
Dim A (1 to n+20) as single
Dim i as integer
. . . . . .
    S=0
For i=n+3 to n+13 step 2
   S=s+a(i)
Next i
Print "s= "; s
```

Приклад. Знайти кількість елементів вектора $A=(a_1,\ldots,a_4)$, які перевищують значення 2,5.

```
REM ‘Це коментар’
Const n=4
Dim A (1 to n) as single
Dim i as integer, k as integer
Cls
Locate 2,8
Print “Програма на VBA”
For i=1 to n
          Locate 4, i*15
          Print “a(“; i ;”)= ”
          Locate 4, i*15+8
          Input a(i)
   Next i
k=0
For i=1 to n step 1
        if a(i) >2,5 then
             k=k+1
        End if
Locate 6,8
Print “кількість елементів, які є більшими, ніж 2,5  дорівнює  “; k
```

Один з операторів циклу за умовою має такий синтаксис:

```
While  умова
    оператори
Wend
```

Приклад. Розв’язати попередню задачу за допомогою оператора While:

```
  . . . . . .
k=0
i=1
While i<= n
        if a(i) >2,5 then
             k=k+1
        end if
            i=i+1
 Wend
 . . . . . .
```

1.3.3. Використання VBA в системі EXCEL

Економіст у своїй практиці постійно працює зі стандартними додатками системи Windows – Word, Excel, Access. Останні дві системи мають досить потужні засоби для обробки економічної інформації. Проте завжди виникають ситуації, коли самими тільки вбудованими функціями системи Excel чи Access розв'язати задачу неможливо. У таких випадках користувач (самостійно або з допомогою програміста) повинен створити макрос (програму на мові VBA).

Приклад. В системі Excel задана таблиця

	A(1)	B(2)	C(3)	D(4)	E(5)	F(6)
1		№ залікової	Прізвище	Група	Оцінка з математики	Оцінка з макроекономіки
2		1404123c	Баран Б.Б.	Екі-11	3	4
. . .		. . . . .	. . . . .	. . . .	. . . . .	. . . . .
28		1404c098	Їжак І.І.	Екф-12	4	5
29						
30						

Записати в комірку С30 середній бал з макроекономіки серед тих студентів, які отримали „3" із математики.

Відповідь на цей запит можна отримати, записавши в С30 досить складну формулу: =Sumif(E2:E28,3,F2:F28)/countif(E2:E28,3) .

Проте є також змога побудувати макрос (Tools, Macro, Macros, ім'я макросу, Step Into і записати VBA-процедуру) із використанням операторів For та if і конструкції Cells(*i*,*j*) – комірки на перетині *i*-ого рядка та *j*-ого стовпця:

```
Sub mm1()
  S=0
  k=0
For i=3 to 28
    if Cells(i,5) =3 then
        S=S+Cells(i,6)
   End if
Next i
  Cells(30,3)=S/k
End sub
```

Зазначимо, що складніші запити (студенти, які здавали тільки макроекономіку; групи, де немає „двійок" з математики тощо) можна виконати в Excel (а також в Access) лише за допомогою мови VBA.

1.4. Основні конструкції мови С++

1.4.1. Поняття про мову С++

На даний час мова С++ є найпоширенішою мовою програмування. Більша частина системного та прикладного програмного забезпечення розробляється цією мовою. Мова С++ на противагу мові PASCAL дозволяє писати дуже ефективні програми з токи зору використання машинних ресурсів. Проте дана мова є значно складнішою для вивчення, ніж такі мови, як Pascal та VBA. Крім того, термінологія мови С++ часто є досить специфічною.

Програми на мові С++ оформляються у вигляді функцій. Програма користувача, яка виконує введення даних та виведення результатів, як правило, оформляється у вигляді головної функції без параметрів:

```
void main()
{
//тіло програми
} /* кінець програми */
```

Ця головна функція повинна називатися main. Оскільки функція не має формальних параметрів, то в дужках після main нічого не записується.

Тіло програми береться в операторні (фігурні) дужки.

Коментарі або поміщаються в дужки типу /* */, або розпочинаються символами // .

Термін void означає той факт, що тип результату роботи функції є невизначеним, “порожнім”. Зазначимо, що, незважаючи на відсутність усіх формальних параметрів (як аргументів, так і результатів), програміст усе ж таки зобов’язаний записати як термін void так і дужки, всередині яких немає жодних параметрів.

Описи змінних та оператори записуються в довільному порядку.

Для того щоб мати змогу використати функції введення та виведення, перед початком програми потрібно підключити файл, який містить ці функції.

Зокрема, підключивши (директивою #include) файл iostream.h, ми отримуємо змогу використати функцію введення з клавіатури

cin>>змінна…>>змінна

та функцію виведення на екран

cout<<вираз…<<вираз

Приклад. Написати програму, яка вводить із клавіатури одне дійсне та одне ціле число, додає їх і виводить на дисплей результат.

```
# include < iostream.h>
void main()
{int a;
  float b,c;
cout<<endl<<"введіть два числа"
cin>>a>>b;
c=a+b;
cout<<endl<<"Сума чисел дорівнює "<<c
int r;          /* для затримки */
cin>>r;         /*    екрану       */
}
```

Параметр endl в операторі cout виконує перехід на новий рядок.

1.4.2. Типи даних в мові С++

Як і в інших мовах, тип змінної задає її машинне представлення, діапазон допустимих значень на набір операцій, які можна виконувати над цією змінною. Тип змінної у мові С++ оголошують.

Розглянемо описи деяких типів арифметичних змінних:

Тип даних	Ключове слово	Діапазон значень	Обсяг пам'яті, байт
Знаковий довгий цілий	Int	-2147483648 .. 2147483647	4
Знаковий короткий цілий	Signed char int	-32768 .. 32767	2
Дійсний (із плаваючою крапкою)	Float	$-3,4 *10^{-38}$ .. $3,4*10^{38}$	4
З плаваючою крапкою подвійної точності	Double	$-1,7*10^{-308}$ .. $1,7*10^{308}$	8

Загальний вигляд оголошення змінних є таким:

<тип> <ім'я змінної>{,<ім'я змінної>};

Змінні, значення яких не можна змінити в процесі роботи програми, називаються константами. Вони задаються за допомогою ключового слова const.

Приклад.

```
const int n=10;
float x,y;
int i;
```

Над арифметичними змінними виконуються операції +, - , * , / , а також операція інкрементації (збільшення на одиницю) ++ (замість пари операторів a=a+1; b=a можна записати один оператор b=a++).

Над числами виконуються такі операції порівняння: <, >, <=, >=, ==(дорівнює), !=(не дорівнює).

Розглянемо оголошення символьного та рядкового (послідовність символів) типів:

```
char <змінна>{,<змінна>};  /* опис символьного типу */
```

Фігурні дужки означають той факт, що конструкція, розміщена у цих дужках, може бути відсутньою, а може бути присутньою довільну кількість разів:

```
char x,y,z;
char a;
char s1,s2;
```

Символьна константа береться в поодинокі лапки: ‘є’, ‘+’, ‘z’.

Послідовність символів представляє собою рядок. Символьний рядок береться в подвійні лапки: “група Екк-21”, “a+b=c”.

Рядковий тип можна задати як масив символів: char a[6]. Тепер змінній *a* можна, наприклад, присвоїти рядок символів:

a=”Ekf-11”

Як і в мові Pascal, в мові C++ є наявною конструкція регулярного типу даних (масиву).

Формат оголошення масиву є таким:

<тип> <ім’я масиву>[розмірність]

Приклад.

```
float a[10][10];
int b[5];
```

Перше оголошення задає матрицю розміром 10*10. Елементи цієї матриці мають імена від a[0][0] до a[9][9]. Друге оголошення задає вектор (одновимірний масив) з елементами b[0], b[1], b[2], b[3], b[4].

1.4.3. Основні керуючі оператори в мові C++

Як і в мові Qbasic, так і в мові C++ оператор присвоєння використовує символ “=” замість символу “:=”.

Складений оператор – це послідовність операторів, яка знаходиться посередині фігурних дужок (у мові PASCAL - посередині слів begin та end).

Формат умовного оператора (оператора розгалуження) є одним із таких:

```
if (<вираз>)
   <оператор1> ;
else
   <оператор2>;
```

```
if (<вираз>)
   <оператор1>;
```

Умовний вираз береться в круглі дужки. Конструкція THEN не записується. При потребі як після if, так і після else використовують складений оператор. Перед словом else записується крапка з комою.

В умові (логічному виразі) використовують такі логічні операції:

! - заперечення, логічне “ні”;

&& - кон’юнкція, логічне “і”;

|| - диз’юнкція, логічне “або”.

Приклад. Якщо a>b і b=c , то обчислити s1=a+3 та s2=a+4 , інакше обчислити s3=a+5.

```
if (a>b && b==c)
   {s1=a+3;
    s2=a+4
   }
else s3=a+5;
```

Як і в інших алгоритмічних мовах, у мові С++ існують оператор циклу типу FOR та оператор циклу типу WHILE.

Синтаксис оператора типу FOR є таким:

for(вираз1; вираз2; вираз3)
<оператор (тіло циклу)>;

Тут перший вираз задає початкове значення параметру циклу, другий вираз – кінцеве значення, а третій – умову переадресації. Наприклад, заголовок циклу, котрий повинен виконуватися для i=1,2,…,n, повинен мати вигляд

```
for(i=1;i<n+1;i++)
```

Приклад. Знайти суму $S = \sum_{i=1}^{10} \frac{1}{i}$ з використанням оператора for.

```
float s,a;
int i;
s=0;
for(i=1;i<11;i++)
  {a=1/i;
    s=s+a
  }
```

Синтаксис оператора while у мові С++ є таким:

```
while (<вираз>)
    <оператор>;
```

Приклад. Знайти суму $S = \sum_{i=1}^{10} \frac{1}{i}$ з використанням оператора while.

```
float s,a;
int i;
s=0; i=1;
while(i<11)
{
  a=1/i; s=s+a;
   i=i+1;
}
```

Розділ 2. Типові алгоритми обробки економічної інформації

У різних предметних областях інформація має різну структуру. Алгоритми обробки інформації з різних галузей також суттєво відрізняються. Так, в математичних задачах (саме такі задачі розглядалися в середній школі) обсяг вхідних даних є невеликим, а алгоритми є складними і полягають у виконанні великої кількості повторень. Реальні задачі економіки полягають в опрацюванні дуже великої кількості інформації, яка при цьому постійно оновлюється. Алгоритми аналізу та корегування економічної інформації є набагато простішими, ніж алгоритми в математиці. Проте ці алгоритми повинні бути досить раціональними з точки зору швидкодії та затрат машинної пам'яті. У цьому розділі будуть розглянуті алгоритми обробки економічної інформації, які є типовими для задач обліку, маркетингу, економічного аналізу, відбору даних та інформаційної підтримки прийняття економічних рішень.

2.1. Алгоритми пошуку

Сучасні бази даних в економіці містять мегабайти, а іноді й терабайти інформації. Внаслідок цього розробка оптимальних алгоритмів сортування (упорядкування) та пошуку має величезне практичне значення. При цьому для різних ситуацій найбільш ефективними виявляються різні алгоритми.

Розглянемо декілька класичних алгоритмів пошуку даних (інформації), які розміщені в одновимірній таблиці.

2.1.1. Лінійний пошук

Приклад. Вияснити, чи знаходиться елемент b в одновимірному цілочисельному масиві (одновимірній таблиці) $a = (a_1,\ldots,a_n)$.

```
Program P1;
Const n=100;
Var
      A:array[1..n] of integer; b:integer;
       i:byte; R: Boolean;
Begin
      . . . . . .
R:=false;
```

```
For i:=1 to n do
   if b=a[i] then R;=true;
if b then write (‘шуканий елемент знаходиться в таблиці’)
      else write (‘шуканий елемент не знаходиться в таблиці’);
  . . . . . .
```

Цей алгоритм завжди виконує n порівнянь. Середню швидкість алгоритму можна збільшити майже удвічі, завершуючи цикл відразу після знаходження елементу b :

```
 . . . . . .
R:=false; i:=1;
While R and i<=n do
  Begin  if b=a[i] then r:=true; i:=i+1  end;
if b then write (‘шуканий елемент знаходиться в таблиці’)
      else write (‘шуканий елемент не знаходиться в таблиці’);
  . . . . . .
```

Приклад. Задано впорядкований масив цілих чисел $a_1 \le a_2 \le \ldots \le a_n$ і деяке ціле число b . Потрібно знайти таке місце i , щоб

$$a_i \le b \le a_{i+1}$$

```
Program P2;
Var
          A:array[1..n] of integer; b:integer;
     . . . . . .
      i:=1;
      while (a[i]<b) and (i<n) do i:=i+1; {знайдено місце вставки}
```

Приклад. Вставити елемент b у впорядкований масив (цілих чисел, дійсних чисел, символів) $a_1 \le a_2 \le \ldots \le a_{n-1}$ так, щоб впорядкування масиву не порушилося.

```
Program P3;
Var
A:array[1..n] of integer; b:integer;
     . . . . . .
i:=n-1;
while (a[i]<b) and (i>=1) do
    begin a[i+1]:=a[i]; i:=i-1 end;   {пошук та пересування}
     a[i+1]:=b; {вставка елементу у масив}
```

2.1.2. Пошук методом половинного ділення

У випадку пошуку елемента у впорядкованому масиві набагато швидшим є алгоритм послідовних поділів цього масиву пополам (метод половинного ділення).

Приклад. Вставити елемент *b* у впорядкований масив $a_1 \le a_2 \le \ldots \le a_{n-1}$.Використати метод половинного ділення.

```
Program P4;
Const n=100;
Type vector=array[1..n] of integer;
Var a:vector;
     b;integer;
     I,p,q,s:byte;
Begin
  . . . . . .
p:=1; q:=n; {початкові межі пошуку}
while (p<>q) do
   begin
        s:=(p+q) div 2; {середина}
        if a[s]<b then p:=s+1 else q:=s
   end;  {елемент  потрібно вставити після ap= aq }
for i:=n-1 downto p do a[i+1]:=a[i];
a[p]:=b {вставка}
  . . . . . .
```

Кількість порівнянь для пошуку місця вставки у методі половинного ділення становить $[\log_2(n+1)]$ замість n чи $n/2$ у методі лінійного пошуку. Проте власне вставка (а також виключення) елементу в таблицю все ж таки вимагає $n/2$ кроків.

2.1.3. Пошук у двійковому дереві

Задання великих масивів даних у вигляді двійкового дерева є одним з найпоширеніших представлень у базах даних. Швидкість пошуку у двійкових деревах (як і швидкість пошуку методом половинного ділення) вимірюється величиною $\log_2(n)$. Крім того, алгоритм вставки нового елементу та алгоритм виключення елементу з двійкового дерева мають ту ж швидкодію $\log_2(n)$.

Нехай, наприклад, задана така послідовність : $a_1, a_2,\ldots,a_n,\ldots$ = 215, 192, 250, 220, 100, 260, 231,…

Перше з чисел (a_1) поміщається в корінь дерева. Кожне наступне число a_i порівнюється з тим, що знаходиться в корені. Коли $a_i < a_1$, то відбувається рух (переміщення) по дереву вліво, якщо ж $a_i > a_1$, то відбувається рух вправо. Якщо після такого руху виявляється вільне місце, то число a_i записується на це вільне місце. Якщо ж у результаті одного переміщення вільного місця не виявляється, то виконуються нові порівняння.

Для нашої послідовності отримується таке двійкове дерево (рис. 1):

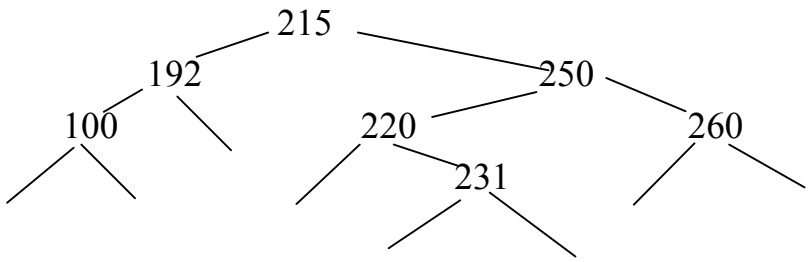


Рис. 1.

Інформація, що знаходиться в кожній вершині дерева, реально розміщується в певній адресі у пам'яті комп’ютера. Дуги (покажчики з однієї вершини на іншу) – це значення адрес вершин. Таким чином, кожній вершині дерева на рисунку в пам'яті комп’ютера відповідає три елементи: значення (a_i), дуга (адреса) вліво та дуга (адреса) вправо. Якщо дуга на ліву чи праву вершину відсутня, то адресою вважається спеціальний символ, наприклад, число нуль (рис. 2.):

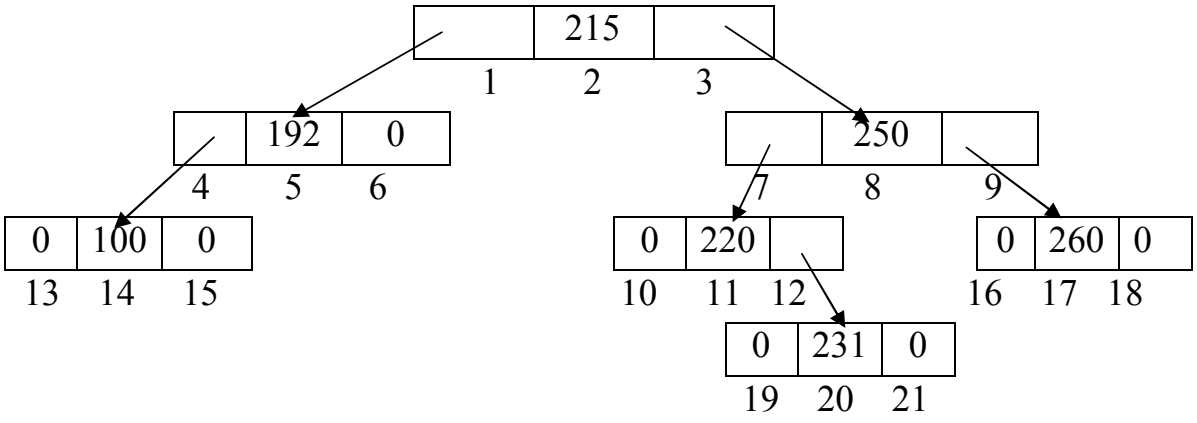


Рис. 2.

Однією з переваг задання масиву (одновимірної таблиці) у вигляді дерева є той факт, що вершини не мусять бути розташовані в пам'яті комп'ютера підряд.

У випадку великих масивів дуги задаються за допомогою конструкції "покажчик (pointer)". Ця конструкція буде розглянута нами в пункті 2.5. Зараз із метою розуміння алгоритму пошуку в дереві застосуємо конструкцію "array":

DER : array[1..n] of integer

5	215	8	14	192	0	11	250	17	0	220	20	0	100	0	0	260	0	0	231	0	…
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Напишемо програму, яка знаходить місце (адресу) елементу k у послідовності $a_1, a_2,\ldots,a_n$, заданій масивом DER (в припущенні, що елемент k в цій послідовності існує).

```
Program P5;
Const n=50;
Var     DER : array[1..n] of integer; k: integer;
          i: integer;
Begin
  . . . . . .
write ('введіть елемент k'); readln (k);
          i:=2;    {корінь дерева}
while DER[i]<>k do
     if k< DER[i] then i:=DER[i-1]
                  else  i:=DER[i+1];
writeln('елемент',k:5,'знаходиться на', i:3,'-ому місці');
  . . . . . .
```

Доведено, що при рівновипадковому поступленні елементів a_i в таблицю середній час пошуку у двійковому дереві становить $1{,}44\log_2 n$ кроків. Для двійкових дерев спеціального вигляду (так званих збалансованих дерев) середній час є ще меншим: $1{,}04\log_2 n$.

Робота алгоритмів вставки та виключення елементів із дерева полягають у пошуку місця вставки (виключення) та у виконанні потім лише декількох дій по перебудові дерева. Таким чином швидкодія останніх алгоритмів також має порядок $\log_2 n$.

Зараз в усіх потужних системах управління базами даних (СУБД) пошук даних у таблицях реалізовано за допомогою двійкових дерев. Двійково-пошукове дерево в термінології баз даних називається індексом, а організація дерева – індексуванням (по аналогії з пошуком за індексом у бібліотеках).

2.2. Алгоритми сортування

Як ми бачили в попередньому розділі, інформація у великих таблицях повинна бути впорядкованою (посортованою). Це сортування повинно підтримуватися при кожній зміні даних у таблиці. Слід враховувати, що в системах обробки економічної інформації вхідні дані поступають невпорядкованими, а виконувати сортування вручну зараз уже є недопустимим. Отже, вибір найкращого алгоритму сортування в кожній конкретній ситуації має важливе значення.

2.2.1. Сортування вставками

Спочатку вважається, що впорядкована частина таблиці складається тільки з елементу a_1 . Розглядається почергово кожен j-ий елемент таблиці (j=2,3,…,n) і вставляється на потрібне місце у вже впорядкований масив a_1, a_2,…,a_{j-1} . Для вставки a_j у масив a_1,a_2,…,a_{j-1} використовується алгоритм програми Р3 .

```
Program P6;
  . . . . . .
for j=2 to n do
   begin
        b:=a[j]; i:=j-1;
        while (i>=1) and (a[i]>b) do
              begin a[i+1]:=a[i]; i:=i-1 end;  {пошук місця}
       a[i+1]:=b   {вставка}
   end;
. . . . . .
```

Цей алгоритм ще називають сортуванням простими вставками, сортуванням за допомогою занурення.

Очевидно, що алгоритм сортування вставками має оцінку $O(n^2)$, тобто виконує $C \cdot n^2$ порівнянь. Тому цей метод є сенс застосовувати тільки для невеликих масивів.

2.2.2. Сортування вибором

Ідея цього алгоритму є такою: вибирають найбільший елемент і записують його на останнє j=n місце. Далі вибирають найбільший серед тих, що залишилися і записують на передостаннє j=n-1 місце. Таку дію виконують n-1 разів.

Для знаходження місця (номеру) *i* максимального елементу в невпорядкованій частині використовуються послідовний перегляд.

```
Program P7;
  . . . . . .
for i:=n downto 2 do {вставка на місце j=n, n-1,…,2}
  begin
     b:=a[1]; k:=1;
     for i:=2 to j do {вибрано найбільший}
          if a[i]>b then {елемент в послідовності a1,…,aj}
                    begin b:=a[i]; k:=i end;
    a[k]:=a[j]; a[j]:=b  {обмін місцями}
  end;
  . . . . . .
```

Оцінка швидкості цього алгоритму також становить $O(n^2)$. При сортуванні вибором виконується дещо більше порівнянь, ніж при сортуванні вставками, проте переміщень здійснюється менше.

2.2.3. Сортування обмінами

Кожна пара сусідніх елементів порівнюється між собою і, при потребі, переставляється. В результаті першого перегляду масиву (таблиці) найбільший елемент займе останнє місце. Після другого перегляду передостаннє місце займе другий за величиною елемент. Алгоритм обміну виконує n-1 кроків. Найбільші елементи при цьому кожен раз виплавають нагору. Тому даний алгоритм ще називають методом бульбашки.

```
Program P8;
  . . . . . .
for i:=1 to n do
   for j:=1 to n-1 do
      if a[j]>a[j+1] then
          begin  R:=a[j+1]; a[j+1]:=a[j]; a[j]:=R end
```

Оцінка швидкості алгоритму також дорівнює $O(n^2)$. Недоліком алгоритму є досить велика кількість переміщень. Проте для частково впорядкованих масивів цей метод є дуже зручним. У цьому випадку бажано, щоб внутрішній цикл виробляв ознаку ”відбувалися переміщення, чи ні”. Тоді кількість повторень зовнішнього циклу суттєво зменшується.

2.2.4. Адресне сортування

Нехай відомо, що всі елементи цілочисельного масиву a_1, a_2,…,a_n не перевищують величини M. тоді виникає можливість виконати сортування цього масиву за n+M кроків. Для цього будується додатковий масив b = (b_1, b_2,…,b_n),

де $b_j = \begin{cases} 1, & існує\ i, \quad що\ a_i = j \\ 0, & інакше \end{cases}$.

Далі переглядається масив b і ті значення j , де b_j=1 утворюють впорядований масив.

```
Program P9;
Const n=10; M=1000;
Var   a:array[1..n] of 1..M;
      b:array[1..M] of 0..1;
      i,j : integer;
Begin
  . . . . . .
  for j:=1 to M do b[j] :=0;  {побудова}
  for i:=1 to n do b[a[i]]:=1;   {масиву b}
   i:=0;
  for j:=1 to M do
        if b[j]=1 then
             begin  i:=i+1; a[i]:=j  end;
         . . . . . .
```

Недоліком адресного сортування є великі витрати машинної пам'яті.

2.2.5. Сортування за допомогою перемішаних таблиць

Нехай можливі значення елементів (ключів) таблиці утворюють деяку множину потужності M. Нехай також у кожен момент часу таблиця може містити не більше, ніж n елементів, де n є набагато меншим, ніж M.

Приклад. В кожен момент часу масив даних може містити не більше, ніж n=10 елементів, кожен із яких є цілим числом від 1 до M = 999. Потрібно так розмістити ці елементи, щоб пошук кожного з них полягав у виконанні лише кількох кроків.

Пропонується такий алгоритм сортування (алгоритм перемішування, алгоритм розсіювання):

1. Утворюється пустий (наприклад, заповнений нулями) масив із n =10 комірок;

2. Кожен елемент, який є числом, що містить k сотень, записується в комірку k ; якщо ця комірка вже зайнята, то цей елемент записується в наступну вільну комірку.

Нехай масив будується з таких чисел:

215, 910, 305, 250, 826, 827.

Після розсіювання перших трьох чисел маємо:

Номер комірки	Ключ	Дані, пов’язані з ключем
0	0	
1	0	
2	215	дані, що відносяться до ключа 215
3	305	дані, що відносяться до ключа 305
4	0	
5	0	
6	0	
7	0	
8	0	
9	910	дані, що відносяться до ключа 910

Комірка з номером 2 для ключа 250 вже є зайнятою, тому записуємо цей ключ та пов'язані з ним дані у першу наступну вільну комірку – комірку 4. Для останнього елемента (елемента 827) вільною коміркою буду комірка з номером 0 (як наступна за коміркою 9).

Отримана таблиця є, з однієї сторони, майже впорядкованою, з іншої – перемішаною. Такі таблиці ще називають hash-таблицями від англійського слова hash (м'ясорубка, що перемішує):

0	827
1	0
2	215
3	305
4	250
5	0
6	0
7	0
8	826
9	910

Має місце така теорема: Якщо

1. Кожній комірці таблиці відповідає однакова кількість елементів множини можливих значень ключів;

2. ключі поступають у таблицю рівновипадково;

3. таблиця заповнена не більше, ніж на 75%,

то середній час пошуку конкретного ключа у цій таблиці становить 1,5 кроків.

Складемо програму запису нового елементу *b* у таблицю $T = t_0,\ldots t_9$. .

```
Program P10;
  . . . . . .
k:=trunc(b/100);  {ціла частина від b/100}
while T[k}>0 do
    begin       {пошук комірки для  b}
        if k=9 then  k:=0
                else k:=k+1
    end;
T[k]:=b;  {запис елемента b в таблицю}
```

Алгоритм виключення елемента з перемішаної таблиці полягає спершу в його пошуку, а потім в записуванні на його місце спеціального символу, наприклад, числа ”-1”.

Алгоритм пошуку розпочинає пошук із комірки trunc(*b*/100) і закінчує пошук або знаходженням елементу *b*, або вийшовши на нульову комірку.

Таким чином усі три алгоритми над перемішаними таблицями виконують лише декілька порівнянь. Швидкодія цих алгоритмів залежить лише від коефіцієнта заповнення, а не від довжини таблиці.

2.3. Алгоритми над файлами

Файли, записи яких містять реквізити різних типів, є основними структурами даних в економічних задачах. Розглянемо деякі класичні алгоритми над файлами.

2.3.1. Знаходження часткових підсумків

Нехай задано файл такої структури:

Група	Предмет	Кількість незадовільних оцінок
К11	Р1	2
К11	Р4	1
С21	Р1	4
К21	Р1	3
К21	Р3	0
К21	Р4	1
. . . . .	. . . . .	. . . . .

Цей файл є впорядкованим за першим стовпцем. Отже, всі записи, що стосуються однієї групи, у цьому файлі розташовані підряд. Зазначимо, що конкретні назви груп та кількість рядків, які відповідають кожній з груп, є наперед невідомими.

Потрібно знайти часткові підсумки незадовільних оцінок по кожній із груп зокрема. Тобто потрібно отримати файл такого вигляду:

Група	Кількість незадовільних оцінок
К11	3
С21	4
К21	4
. . . . .	. . . . .

Основна ідея алгоритму є такою: в пам'ять зчитується перший запис файлу F1. Далі в циклі зчитуються всі наступні записи цього файлу. Якщо наступний запис відноситься до цієї ж групи, що й попередній, то відбувається сумування. Якщо ні – то формується один запис файлу F2.

```
Program P11;
Type
   T1=record Gr: string[3]; Pr: string[2]; M2:integer end;
    T2= record Gr: string[3]; M2:integer end;
Var
   x1:T1; x2:T2;
   F1: file of T1; F2: file of T2;
    G:string[3]; M:integer;
Begin
   Reset(F1); Rewrite(F2);
   Read(F1,X1);       {перший запис}
  G:= x1.Gr; M:=x1.m2;
While not EOF(F1) do
     Begin
        Read(F1,x1);
        if x1.Gr=G then M:=M+x1.m2 {група не змінилася}
                   else                    {група змінилася}
                      begin
                      x2.Gr:=x1.Gr; x2.M2:=M;
                     write(F2,x2);
                     G:=x1.Gr; M:=x1.M2
                      end
     End;
      x2.Gr:=x1.Gr; x2.M2:=M;  {остання група}
       write(F2,x2);
```

Програма P11 дуже просто перетворюється в програму проекції файлу F1 за першим стовпцем, тобто у програму, яка знаходить перелік груп без повторень:

К11
С21
К21
. . . .

Для цього потрібно лише видалити оператори, які підраховують величину M - кількість незадовільних оцінок.

2.3.2. Знаходження перетину та об'єднання файлів

Нехай задано два файли F1 та F2 однакової структури. Файл F1 містить номери залікових та прізвища тих студентів, які отримали оцінку «незадовільно» з математики, а файл F2 - номери залікових та прізвища студентів, які отримали оцінку «незадовільно» з права. Файли F1 та F2 не обов'язково є посортованими.

Об'єднанням файлів F1 та F2 (аналогічно до операції об'єднання множин в математиці) є файл F3, записи якого відповідають тим студентам (без повторень), які отримали хоча б одну незадовільну оцінку. Перетином файлів F1 та F2 є файл F4 – файл студентів, які отримали незадовільні оцінки з обох предметів.

```
Program P12;              { перетин }
Type
    T=record N: string[8]; F: string[15] end;
Var
 x1,x2,x4:T;
 F1,F2,F4: file of T;
Begin
 . . . . ; Reset(F1); Rewrite(F4);
while not eof(F1) do
   begin
       read(F1,x1);
       reset(F2);
       while not eof (F2) do
           begin
             read(f2,x2);
             if x1.n=x2.n then
                  begin x4.n:=x1.n; x4.F:=x1.F; write(F4,x4) end
           end
   end;
 . . . . . .
```

У тому випадку, коли файли F1 та F2 є впорядкованими, програму можна зробити значно швидшою, перемістивши оператор reset(F2) із тіла зовнішнього циклу на початок програми поруч з оператором reset(F1).

```
Program P13;   {об'єднання}
Var    R:boolean;
. . . . . .
Begin
    Reset(F1); Rewrite((F3);
   While not eof(F1) do
      begin
           read(F1,x1);x3:=x1; write(F3,x3)
      end;  {весь файл F1 переписуємо у F3}
 Reset(F2);
While not eof(F2) do
     begin
          read(F2,x2);
          R:= false;   {в F1 не знайдено запису x2}
         Reset (F1);
         While not eof (F1) do
           begin read(F1,x1); if x2.n=x1.n then R:=true end
         if R=false then
              begin x3:=x2; write(F3,x2) end {дописуємо запис x2 з F2}
      end;
```

2.3.3. Побудова сполучення двох файлів

Операція сполучення двох файлів за ключем на практиці зустрічається, мабуть, найчастіше серед усіх операцій над таблицями (файлами).

Нехай задано два файли:

F1

Код товару	Кількість товару
C	Q

F2

Код товару	Ціна за одиницю товару
C	P

Припустимо, що кожен код товару з файлу F1 знайдеться у файлі F2. Проте кількість записів файлу F2 може бути більшою, ніж у файлі F1. Потрібно побудувати файл F3, записи якого мають таку структуру:

F3

Код товару	Кількість товару	Ціна за одиницю товару
C	Q	P

Розглянемо випадок, коли файл F1 не є впорядкованим за кодами товарів (оскільки файл F2, як правило, на практиці є впорядкованим). Тоді алгоритм побудови F3 полягає в послідовному перегляді файлу F1 (цикл), де для кожного запису x1 із F1 переглядаються всі записи з F2 (цикл у циклі).

```
Program P14;
Type
  T1=record C: string; Q: integer; end;
  T2= record C: string; P: real; end;
  T3= record C: string; Q: integer; P: real end;
Var
  X1:T1; x2:T2; x3:T3;
  F1:file of T1; F2:file of T2; F3:file of T3;
 . . . . . .
reset(F1); rewrite(F3);
repeat
      Read(F1,x1);
      Reset(F2); {встановлення покажчика на початок файлу F2}
      Repeat
          Read(F2,x2);
          if x1.C=x2.c then  {коди співпали}
               begin
                    x3.C:=x1.C; x3.Q:=x1.Q; x3.P:=x2.P
               end;  {формування запису файлу F3}
      Until eof(F2);
      Write(F3,x3)
until eof(F1);
close(F1); close(F2); close(F3);
```

В іншому випадку, коли як F1, так і F2 є однаково впорядкованими, швидкодія алгоритму може стати значно кращою. Для цього в зовнішньому циклі не потрібно кожен раз встановлювати покажчик файлу F2 на початок (команду Reset(F2) слід перенести поруч з командою Reset(F1)).

Нехай файл F1 містив n записів, а файл F2 - m записів. Тоді у випадку неоднаково впорядкованих файлів сполучення за ключем вимагає $n \cdot m$ зчитувань, а у випадку однаково впорядкованих – лише $n+m$.

2.3.4. Алгоритм злиття двох впорядкованих файлів

Одним із способів впорядкування великих файлів є розбиття їх на окремі частини, сортування кожної з них окремо і далі злиття цих частин в один файл. Крім того, задача злиття впорядкованих файлів має і самостійне значення.

Нехай файли F1 та F2 мають однакову структуру:

F1, F2

Ключ (Key)	Дані (D)

Ці файли можуть мати різну довжину, проте повинні бути однаково впорядкованими. Крім того, жоден із ключів файлу F1 не повинен зустрітися у файлі F2 і навпаки. Алгоритм злиття полягає в тому, щоб одночасно послідовно переглядати обидва вхідних файли.

```
Program P15;
Type
     Z=record key: string; D: string end;
Var
     x1,x2,x3:z; F1,F2,F3:file of z;
   . . . . . .
reset(F1); reset(F2); rewrite(F3);
While not eof(F1) do
 Begin
       Read(F1,x1);
       While (not eof(F2)) and (x2.key<x1.key) do
          begin
              read(F2,x2) x3:=x2; write(F3,x3)
          end; {переписати на F3 з F2 всі записи, менші від x1}
       x3:=x1;
       write (F3,x3)  {переписати на F3 запис x1}
 End;
While not eof(F2) do {цей цикл інколи не працює жодного разу}
  Begin
      read(F2,x2); x3:=x2; write(F3,x3) {переписати „хвіст” файлу F2}
  End;
  . . . . . .
```

2.4. Алгоритмізація наближених обчислень

Класичний курс вищої математики досліджує загальні властивості рівнянь, нерівностей, систем рівнянь та нерівностей, інтегралів, диференціальних рівнянь тощо. Існують точні методи (алгоритми) для обчислення деяких типів інтегралів, деяких типів звичайних та диференціальних рівнянь. Проте для більшості математичних задач, які виникають із реальної практики, точних алгоритмів їх розв'язку немає. В цій ситуації дуже важливе значення набувають так звані методи наближених обчислень (числові методи), які дозволяють практично знайти розв'язок конкретного рівняння, обчислити конкретний інтеграл тощо. Звичайно, наближені обчислення знаходять не загальну формулу, а лише конкретні числа (з потрібною точністю).

2.4.1. Обчислення числових рядів

Нехай потрібно у деякій точці x обчислити значення функції $y=e^{-x}$ з точністю $\varepsilon=10^{-6}$.

З курсу вищої математики відомо, що функція розкладається в такий ряд Тейлора:

$$e^{-x}=\sum_{n=0}^{\infty}a_n=\sum_{n=0}^{\infty}(-1)^n\frac{x^n}{n!}=1-\frac{x}{1!}+\frac{x^2}{2!}-\frac{x^3}{3!}+\ldots$$

Цей ряд є знакозмінним, тому ознакою закінчення обчислень може бути умова $\frac{x^n}{n!}<\varepsilon$.

Зробимо два зауваження.

1. Незважаючи на те, що ми повинні обчислити ряд різних значень $a_0, a_1, \ldots, a_n, \ldots$, програмістський підхід дозволяє розраховувати усі ці значення в тій самій комірці пам'яті a.

2. Для великих значень n як чисельник x^n, так і знаменник $n!$ виразу a_n стають настільки великими, що починають перевищувати максимально допустиме число в пам'яті комп'ютера. Тому є сенс використовувати рекурентну формулу:

$$\begin{cases} a_0=1 \\ a_n=a_{n-1}\cdot\dfrac{-x}{n!} \qquad (n=1,\,2,\,\ldots) \end{cases}.$$

```
Program P16;
var
   x: real;  y: extended; {максимальний порядок числа = 4932} eps: real;
   n integer; a: extended;
  . . . . . .
n:=1; a:=1; y:=a;
Repeat
     a:=a*(-x/n);  y:=y+a
     n:=n+1
Until abs(a)<eps;
Writeln( ‘x= ‘, x:10:6, ‘y= ‘,y:10:6)
  . . . . . .
```

Як видно з наведеного вище прикладу, програмування математичних розрахунків має деякі специфічні особливості по відношенню до розрахунків у класичній математиці.

2.4.2. Розв'язування рівнянь методом половинного ділення

Знаходження коренів рівняння f(x)=0 є складною математичною задачею вже у тому випадку, коли f(x) не є лінійною чи квадратичною функцією.

Наближене розв'язування таких рівнянь полягає у виконанні двох етапів.

1. Відокремлення коренів. За допомогою дослідження функції y=f(*x*) та з використанням її графіка виявляють інтервал [a;b], на якому рівняння f(*x*)=0 має рівно один корінь.

2. Уточнення коренів. Якщо функція f(*x*) є неперервною на відрізку [a;b] і має на [a;b] рівно один корінь, то знаки чисел f(a) та f(b) мусять бути різними. Поділивши відрізок [a;b] пополам (на відрізки [a;c] та [c;b]), виявляємо той з них, де функція f(*x*) на його кінцях знову має різні знаки. Таким чином інтервал, де знаходиться шуканий корінь, зменшується вдвоє. Процес поділу відрізка пополам продовжується, поки довжина відрізка, де знаходиться корінь, не стане меншою від заданого числа ε.

```
Program P17;
Var
  a,b,c: real; eps:real;
function f(x:real):real; <тіло функції>;
```

```
Begin
While b-a>eps do
   begin
        c:=(a+b)/2;
        if  f(a)*f(b)>0 then a:=c else b:=c
   end;
  writeln((b+a)/2:10:6);
  . . . . . .
```

2.4.3. Наближене знаходження визначених інтегралів

Визначеним інтегралом $\int_a^b f(x)dx$ на відрізку [a;b] від неперервної на цьому відрізку функції f(x) називається границя інтегральних сум

$$\lim_{n\to\infty} S_n = \lim_{n\to\infty} \frac{b-a}{n} \cdot \sum_{i=1}^{n} f(x_i) = \lim_{n\to\infty} \frac{b-a}{n} \cdot \sum_{i=1}^{n} f(a + i \cdot \frac{b-a}{n}) \quad .$$

Зрозуміло, що для достатньо великих значень n інтегральна сума S_n стає близькою до точного значення інтегралу:

$$\int_a^b f(x)dx \approx \frac{b-a}{n} \cdot \sum_{i=1}^{n} f(a + i \cdot \frac{b-a}{n}) \quad .$$

Отже, програма знаходження визначених інтегралів полягає у знаходженні інтегральних сум. А програма знаходження суми - це звичайний цикл типу FOR.

```
Program P18;
Var S,a,b: real; x: real; i: integer;
function f(x:real):real; <тіло функції>;
 . . . . . .
S:=0; x:=a;
For i:=1 to n do
  begin
      x:=a+i*((b-a)/n)
     S:=S+f(x)
  end;  S:=S*((b-a)/n);
write(‘інтеграл = ‘, S)
 .. . . . . .
```

Курс вищої математики дає методи для знаходження величини n, яка забезпечує задану точність ε.

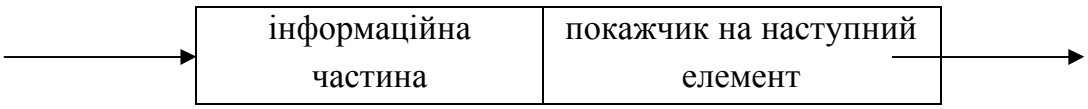
2.5. Спискове програмування

2.5.1. Поняття про динамічне програмування

Більшість типів даних (цілі та дійсні числа, масиви типу array, записи типу record тощо) є статичними. Вони з'являються в момент виконання описів програми і протягом всього виконання програми займають у пам'яті комп'ютера фіксовану кількість комірок, розміщених підряд.

Динамічні структури даних можуть з'являтися і зникати в процесі виконання операторів. Їхній розмір також може змінюватися.

До динамічних структур належать списки. Списком в програмуванні називають впорядковану структуру, кожен елемент якої складається з інформаційної частини та одного чи декількох покажчиків на інші елементи списку:



На рис. 3. представлено звичайний однонапрямлений список:

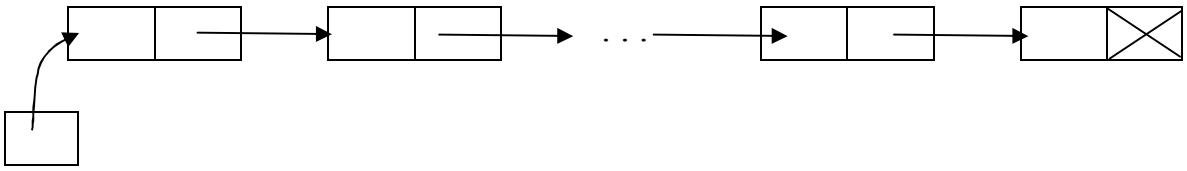


Рис. 3.

Кількість елементів списку можна змінювати в процесі роботи програми. Дуже суттєвою перевагою списків є також те, що їхні елементи не мусять бути розміщеними в пам'яті комп'ютера підряд.

Проілюструємо ідею використання покажчиків спершу на прикладі конструкції array.

Приклад. Нехай елементами списку є цілі числа. Потрібно підрахувати кількість елементів, що перевищують число 4. Ознакою кінця списку вважатимемо число 0 (рис. 4.).

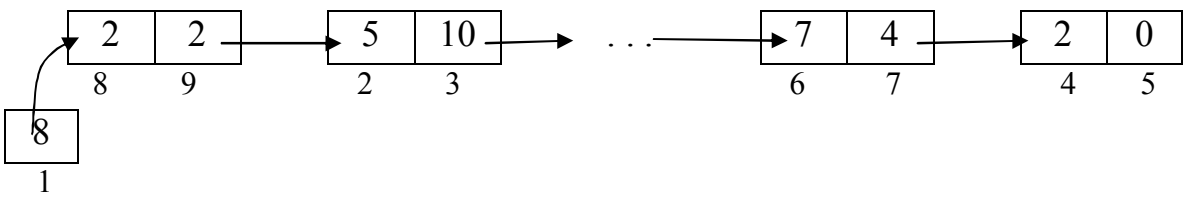


Рис. 4.

У випадку конструкції array маємо такий вектор:

A:	8	5	10	2	0	7	4	2	2	…	…
	1	2	3	4	5	6	7	8	9	10	11

Алгоритм полягає в організації такого циклу, який розпочинається з елементу A[1] і далі згідно покажчиків здійснює переходи від A[i] до A[i+1], поки значення A[i+1] не стане нулем.

```
Program P19;
Var  A: array[1..n] of integer;
       i,k: integrer
 . . . . . .
  k:=0;  i:=0;
While A[i+1] <> 0 do
begin
    i:=A[i+1];
   if A[i]>4 then k:=k+1;
end;
write(‘кількість шуканих елементів дорівнює ’, k);
 . . . . . .
```

Приклад. Нехай інформаційна частина кожного елементу списку є покажчиком на конкретний запис файлу F (рис. 5.):

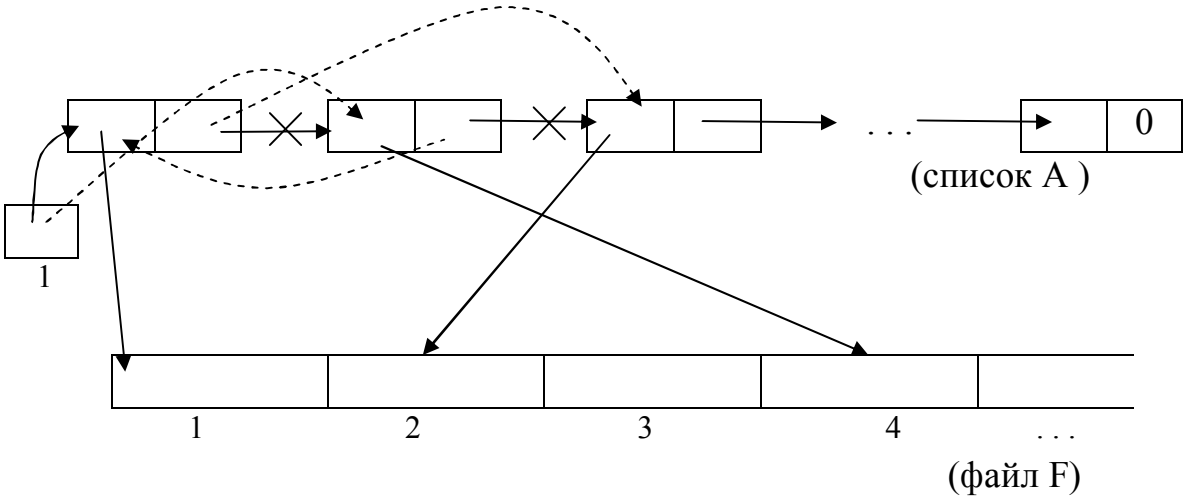


Рис. 5.

Потрібно переставити місцями записи файлу F, які є першим та другим згідно порядку, заданого списком (тобто переставити перший та другий елементи списку, що відповідає записам номер 1 та номер 4 у файлі F).

Адресою першого елементу списку є A[1],

другого елементу – A[A[1]+1],

третього елементу – A[A[A[1]+1]+1].

Отже, алгоритм полягає в заміні трьох покажчиків (нові покажчики зображені пунктиром на рис. 5).

```
Program P20;
R1:= A[1];  R2:= A[A[1]+1]; R3:= A[A[A[1]+1]+1]; {куди стрілки}
L1:=1; L2:=A[1]+1; l3:= A[A[1]+1]+1;   {звідки стрілки}
A[L1]:=R2; A[L2]:= R3; A[L3]:=R2;
```

Програма змінила послідовність перегляду записів файлу F, не переставляючи фізично ці записи.

Приклад. Нехай кожен елемент списку включає три складові: змістовний ключ (код виробу, номер залікової тощо), покажчик на запис файлу (де міститься детальна інформація про виріб, про студента,…) та покажчик на наступний елемент списку (рис. 6):

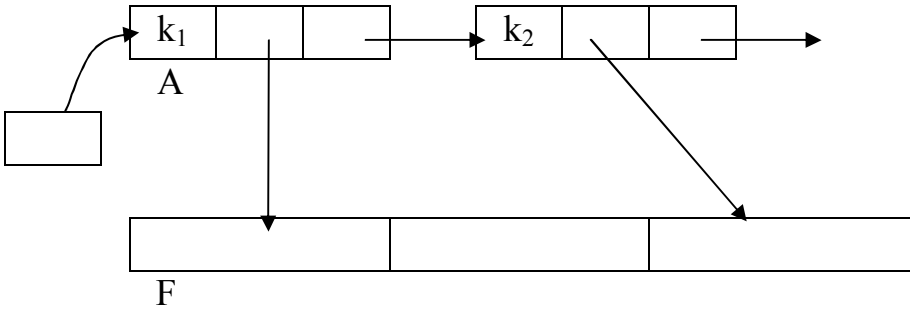


Рис. 6.

Скласти програму, яка зчитує в оперативну пам'ять для аналізу той запис файлу F, який відповідає заданому ключу *k*.

```
Program P21;
Var
A:array[1..100] of integer;
F:file of record … end; x: record … end;
 . . . . . .
read(k);  {k - змістовний ключ}
i:=A[1];
while (A[i]<>k) do i:=A[i+2];
    j:=A[i+1]; seek(F,j); {j – машинний ключ}
    read(F,x);  {зчитування потрібного запису}
```

В програмі P21 використана конструкція seek(F,j) – встановлення покажчика файлу F на запис з номером j. Програма фактично реалізувала індексний доступ до файлу (перехід від змістовного з точки зору людини ключа *k* до машинного ключа *j*).

2.5.2. Змінні типу “покажчик”

Змінні типу array не дозволяють створювати довгих списків. Крім того, всі елементи конструкції *array* мусять мати однаковий тип, що робить цю конструкцію дуже обмеженою для використання.

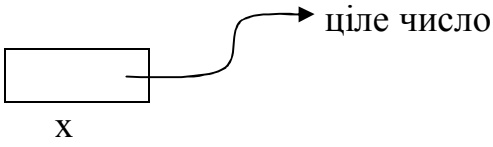
З метою організації списків в сучасних мовах програмування вводяться змінні типу покажчик (pointer). Синтаксис опису покажчика є таким:

Type

<тип покажчика> = ^<тип> ,

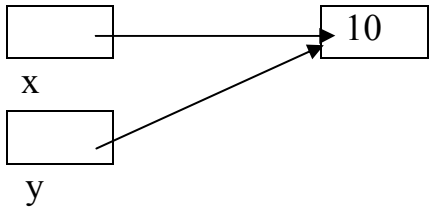
наприклад,

```
Type
  p=^integer;
 Var
   x,y: p; {покажчики на змінну типу integer}
   z ^char;  {покажчик на змінну типу char}
```

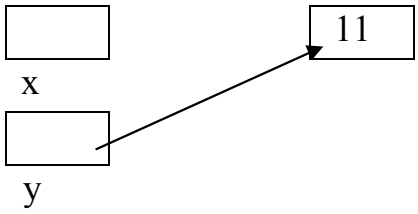


Далі в програмі можливі такі оператори:

x^:=10; y:=x;



x^:=x^+1; x:= nil;

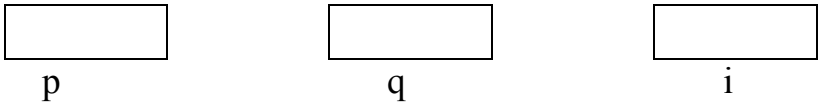


Для того щоб виділити в процесі роботи програми місце пам'яті, в мові Pascal використовують оператор *new*; щоб звільнити місце – оператор *dispose*.

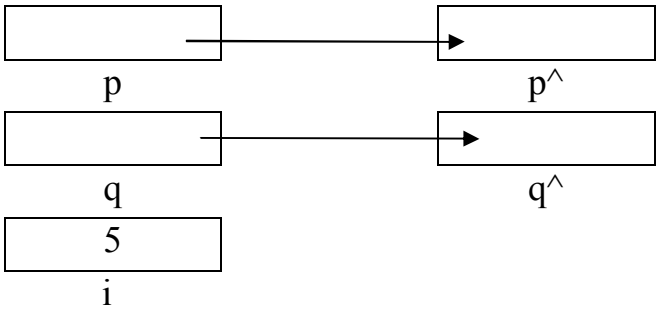
Приклад. Скласти програму з виростанням операторів *new* та *dispose*.

```
Program P22;
Var i:integer;
    p,g: ^integer;
Begin
   {перший рядок} new(p); new(q); i:=5;
   {другий рядок} p^:=7; q^:=p^-i; i:=i+1;
   {третій рядок} p^:=q^+i;
      q:=p;
      write(q^)
End.
```

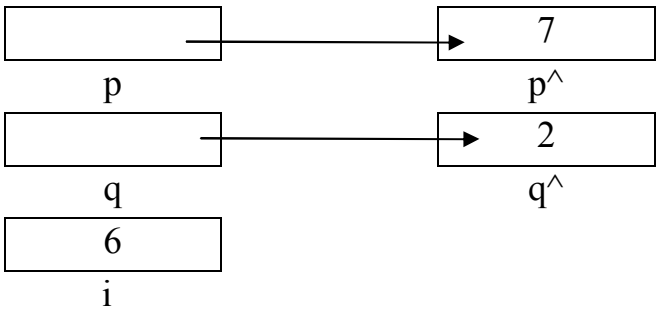
Після описів маємо:



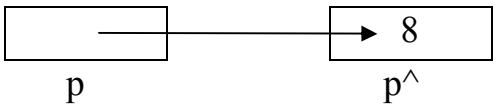
Після виконання операторів першого рядка:



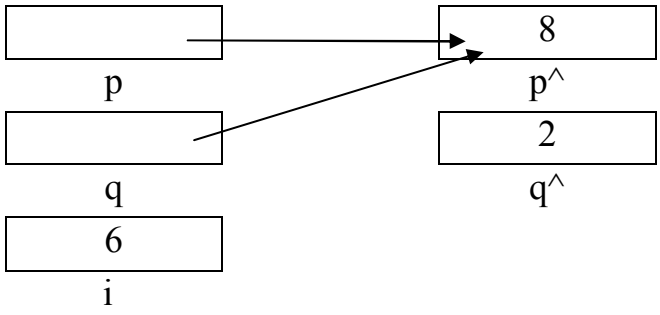
Після наступних трьох операторів (другий рядок):



Після оператора p^:=q^+i (в третьому рядку) маємо:

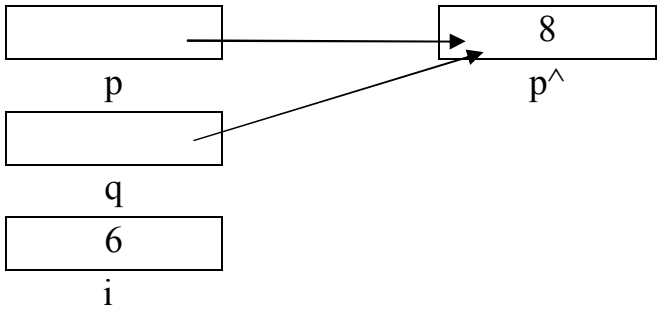


Після виконання присвоєння q:=p отримуємо:



Оператор write(q^) виводить число 8.

Зазначимо, що місце пам'яті під число 2 вже не можна звільнити. Цю пам'ять можна було звільнити, виконавши оператор dispose(q) перед виконанням q:=p :

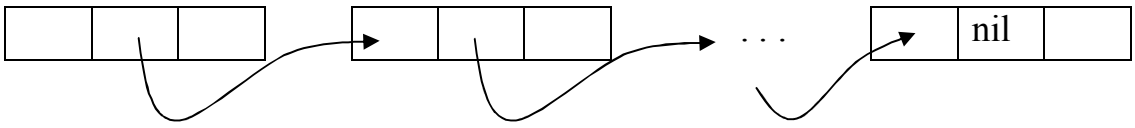


2.5.3. Спискові структури

Розглянемо означення (рекурсивне):

```
Type    d=^t;
        t= record
             . . . . .
           end;
```

Змінними типу t є записи. Одне з полів кожного запису є або nil, або посилання (покажчик):

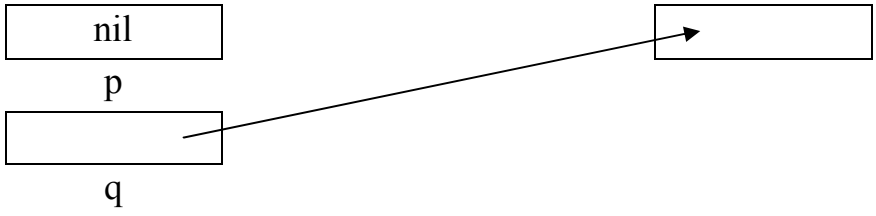


Приклад. Побудова спискової структури.

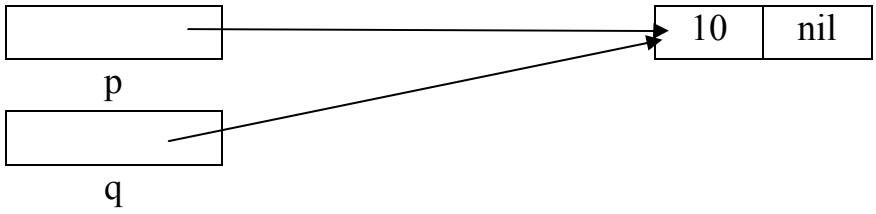
```
Type    pointer  {покажчик} =^elem  {елемент}
        elem = record
                 inf: integer  {інформаційна частина}
                 next: pointer {покажчик}
                 end;
```

```
var
   p,q: integer;
Begin
  . . . . . .
  p:= nil;
  For i:=10 downto 1 do
       begin
         new(q); q^.inf:=i;
        q^.next:=p; p:=q
       end;
```

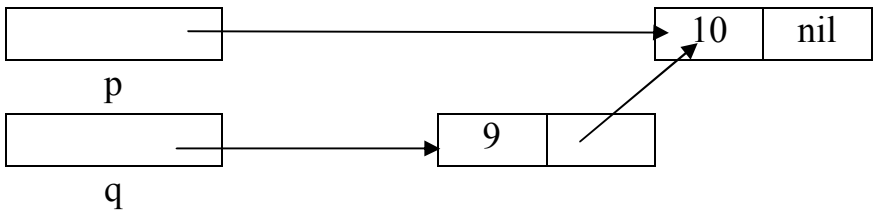
Після операторів p:=nil та new(q) маємо:



Після q^.inf:=i; q^.next; p:=q:



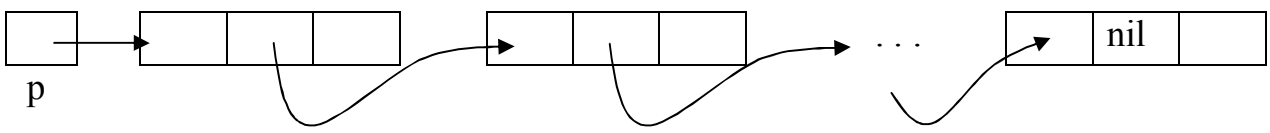
Далі, після виконання групи операторів new(q); q^.inf:=i; q^.next:=p; p:=q, отримуємо:



Наступні кроки виконання циклу будуть будувати нові елементи спискової структури (від останнього до першого).

Приклад. Перегляд списку.

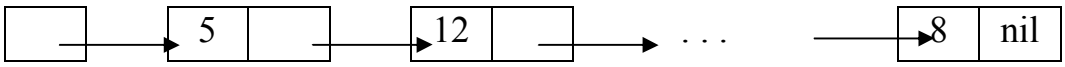
Нехай значенням змінної *p* є покажчик на перший елемент списку:



Написати програму, яка переглядає всі елементи цього списку і друкує інформаційну частину кожного елементу.

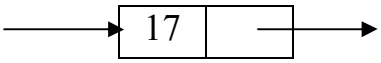
```
. . . . . .
q:=p;
while q<>nil do
begin
    write(q^.inf);
    q:=q^.next {аналог оператора i:=A[i+1]}
end;
```

Приклад. Задано список вигляду



p

Потрібно вставити елемент

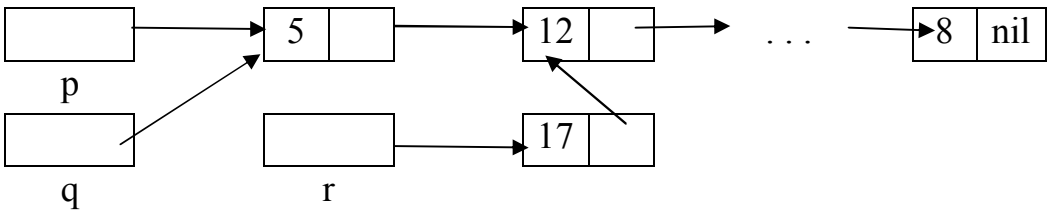


між другим та третім елементами списку.

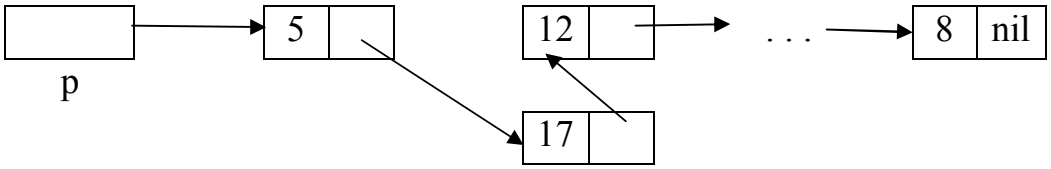
Виконуємо оператори:

q:=p; {для перегляду}

new(r); r^.inf:=17; r^.next:=q^next;



Виконуємо оператори: q^.next:=r; dispose(q); dispose(r):



Новий елемент вставлено на потрібне місце.

Приклад. Перенести із списку *p* другий елемент в список *r*, а потім підрахувати кількість таких елементів, де інформаційна частина є буквою „н”.

```
Program P23;
Type   pointer = ^elem;
       elem = record inf:char; next:pointer end;
Var
    p,q,r: pointer; k :integer;
Begin
r:=p^next;
p^.next:=p^.next^.next;
r^.next:=nil;  {елемент перенесено з одного списку в інший}
k:=0;
while q<>nil do
   begin
       if q^.inf = ‘н’ then k:=k+1
       q:=q^.next
  end
End.
```

Списокове програмування є досить складним для початківців. Проте таке програмування є основним у роботі професійного програміста.

2.6. Алгоритми на графах та деревах

2.6.1. Дерева та алгоритми над ними

Деревовидною структурою (деревом) називають множину взаємозв’язаних об’єктів (вершин, вузлів), розташованих за рівнями за таким правилом:

- на першому рівні один вузол (корінь дерева);
- будь-який вузол і-ого рівня (і ≠ 1) пов'язаний лише з одним вузлом (і-1)-ого рівня (рис. 7.).

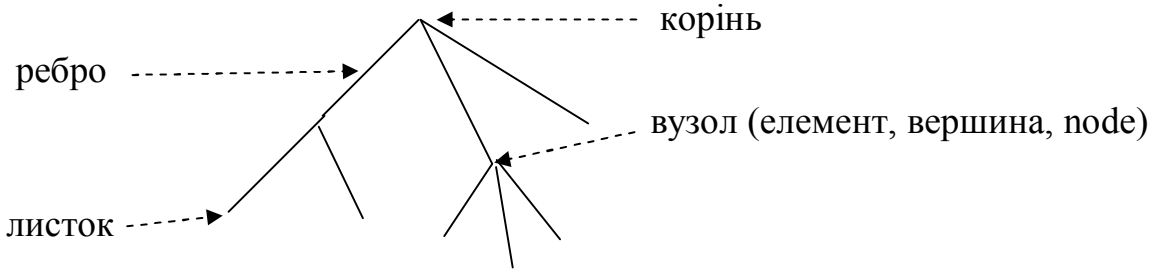


Рис. 7.

Дерево за своєю природою є рекурсивною структурою даних. Тому можливе і таке означення дерева:

Дерево – це або порожнє дерево,
або вершина, якій підпорядковується скінчена кількість дерев (піддерев).

Тому для роботи з деревами природнім є використовувати рекурсивні алгоритми.

Розглянемо дерева спеціального вигляду – двійкові дерева, в яких кожна вершина має не більше, ніж два піддерева (рис. 8.):

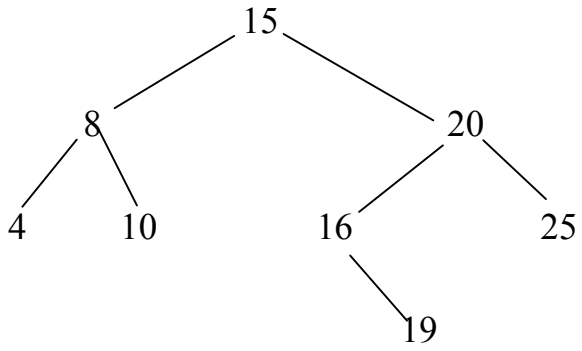


Рис. 8.

Приклад. Елементами двійкового дерева є цілі числа. Підрахувати суму цих чисел.

```
Type   tree=^node;
       node=record elem: integer; left, right: tree end;
Function Sum(t: tree): integer;
  begin
    if tree = nil then sum:=0; {рекурсивне звертання до функції sum}
          else sum:=sum(t^.left)+t^.elem+ sum(t^.right)
  end; {опису функції}
```

Рекурсивні алгоритми мають дуже красивий математичний вигляд, проте вони вимагають значних затрат машинної пам'яті і часу (для створення копій процедур та функцій). Крім того, не всі системи програмування мають апарат рекурсивних процедур та функцій. Тому частіше обробку деревовидних структур програмують, використовуючи алгоритм повернення назад. Програмування з поверненням назад (backtracing) реалізоване в багатьох експертних системах.

Приклад. Задано двійкове дерево. Знайти суму всіх значень його вершин, використовуючи механізм повернення назад. Для дерева, представленого на рис. 8., порядок проходження вершин буде таким:

15, 8 (запам’ятати 20), 4 (запам’ятати 10), 10, 20, 16 (запам’ятати 25), 19, 25

Для запам'ятовування відкладених ”на потім” вершин будується магазин (стек, stack, структура типу LIFO) у вигляді одновимірного масиву (рис. 9):

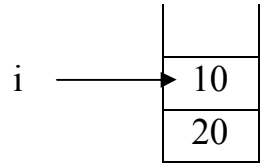


Рис. 9.

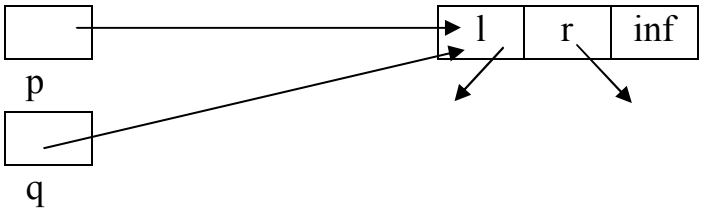
Stack: array[1..50] of integer; i{поточна кількість елементів стеку}:integer;

Основні дії над стеком виглядають так:

{записати елемент у стек:} i:=i+1; stack[i]:=k;

{зчитати один елемент із стека:} k:=stack[i]; i:=i-1;

Двійкове дерево представимо в пам'яті за допомогою спискової структури:



Тепер програма знаходження суми значень вершин дерева з використанням механізму backtracing буде такою:

```
Program P24;
Type rebro=^node
        node= record
                l, r: rebro;
                inf: integer
                end;
Var     p,q: rebro; S:integer; i;integer;
        Stack: array[1..50] of integer;
  . . . . . .
    q:=p; S:=0;
while q<>nil or i>0 {стек не пустий}do
```

```
begin  {*}
     S:=S+q^.inf;
     if q^.l<> nil  then
     begin {**}
          if q^.r<>nil then {записати праву вершину в стек}
                        begin
                           i:=i+1;  Stack[i]:=q^.r
                        end;
           q:=q^.l  {рух вліво}
      end {**}
  else  {вліво вершин немає}
     begin q:=Stack[i]; i:=i-1 end
end {*}
```

. . . . . .

Приклад. В пам'яті комп'ютера знаходиться файл F, кожен запис якого містить ключ та інформаційну частину. Індекс для цього файлу має вигляд двійкового дерева (рис. 10):

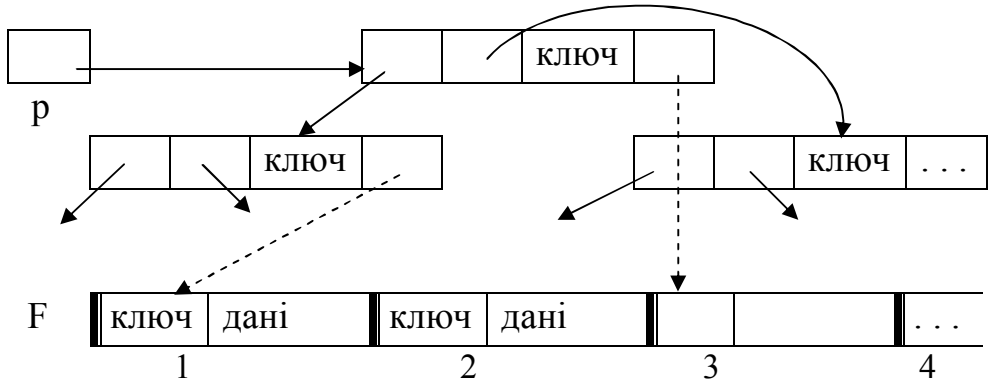


Рис. 10.

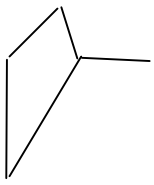
З клавіатури вводиться значення ключа *k*. Скласти програму, яка організує індексний доступ до файлу F, тобто знаходить запис файлу, пов'язаний з цим ключем (в припущенні, що такий запис існує). Індексом є спискова структура з використанням конструкції „покажчик”.

```
Program P25;
Type
        z=record  k: integer;    end;
     rebro=^node;
      node=record  l,r:rebro; k{ключ},n{номер запису в F}:integer end;
```

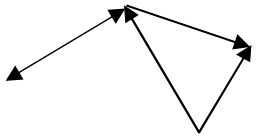
```
Var
p,q: rebro;
F: file of z;  x:z;
k, n: integer;
  . . . . . .
       Read(k);
         q:=p;
while k<>q^.k do
     if k<q^.k then  q:=q^.l
                 else  q:=q^.r
   n:=q^.n;
   seek(F,n);
    read(F,x);
  . . . . . .
```

2.6.2. Графи та алгоритми над ними

Графом (сітьовою структурою) називають множину взаємозв'язаних об'єктів (вершин), які сполучені напрямленими або ненапрямленими дугами (рис. 11.).



ненапрямлені дуги,
неорієнтований граф



напрямлені дуги,
орієнтований граф

Рис. 11.

Існує ряд економічних задач (наприклад, задача комівояжера, задача знаходження критичного шляху тощо), які зводяться до алгоритмів на графах.

Приклад. Для виконання деякої основної роботи (будівництва будинку тощо) потрібно виконати ряд переходів з одного стану в інший. Кожен новий стан отримується в результаті деяких підробіт. Відомий час виконання кожної з цих підробіт. Потрібно розрахувати час виконання основної роботи. Потрібно також знайти так званий критичний шлях. Критичний шлях – це така послідовність робіт, що затримка виконання будь-якої з них призводить до затримки виконання усієї роботи (рис. 12.):

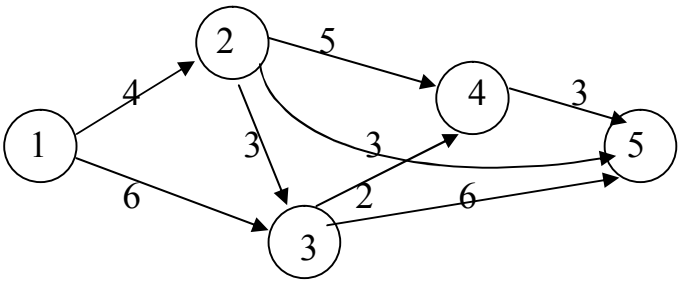


Рис. 12.

Час роботи 1→2 становить 4 дні, роботи 1→3 дорівнює 6 днів, а роботи 2→3 - 3 дні. Стан № 2 досягається виконанням лише однієї роботи 1→2, тобто за 4 дні. Стан № 3 може бути досягнений лише після того, як із стану № 1 буде виконана робота 1→3 (6 днів) та із стану № 2 (4 дні) робота 2→3 (3 дні). Тому стан № 3 досягається за max{0+6;4+3} = 7 (днів). Отже, при переході з початкового стану № 1 до стану № 3 критичним шляхом є шлях <1;2;3>. Розрахувавши час досягнення кожного із станів <R1=0; R2=4; R3=7; R4=9; R5=13>, виявляємо критичний шлях у заданому графі: <1;2;3;5>.

Побудуємо загальний алгоритм знаходження критичного шляху.

Вхідні дані алгоритму – це граф із п'ятьма вершинами {1, 2, 3, 4, 5} та восьми дугами {<1,2>; <1,3>; <2,3>; <2,4>; <2,5>;<3,4>; <3,6.>; <4,5>}.

Кожній дузі <i;j> відповідає час t_{ij}.

Кожній вершині j відповідає час завершення всіх попередніх робіт R_j.

Алгоритм складається з двох частин:

А. Для всіх вершин j, починаючи з першої, знаходимо величини R_j за формулами $R_j = \max\limits_{i:\exists<i,j>} \{R_i + t_{ij}\}$;

В. Для кожної вершини j, починаючи з останньої, знаходимо попередню їй вершину *i* таку, що $R_j = \max\limits_{<i,j>} \{R_i + t_{ij}\}$.

Програмна реалізація цього алгоритму дуже суттєво залежить від способу представлення графу в пам'яті комп'ютера.

Можливі різні способи таких представлень, зокрема:

А. Матриця . Кожна дуга <i,j> характеризується одним з елементів матриці (елементом на перетині *i*-ого рядка та *j*-ого стовпця). Матричне представлення є зручним для програмування, проте вимагає значних затрат машинної пам'яті:

$$\begin{pmatrix} 0 & t_{12}=4 & t_{13}=6 & 0 & 0 \\ 0 & 0 & t_{23}=3 & t_{24}=5 & 0 \\ 0 & 0 & 0 & t_{34}=2 & t_{35}=6 \\ 0 & 0 & 0 & 0 & t_{45}=3 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

В. Набір дуг та вершин.

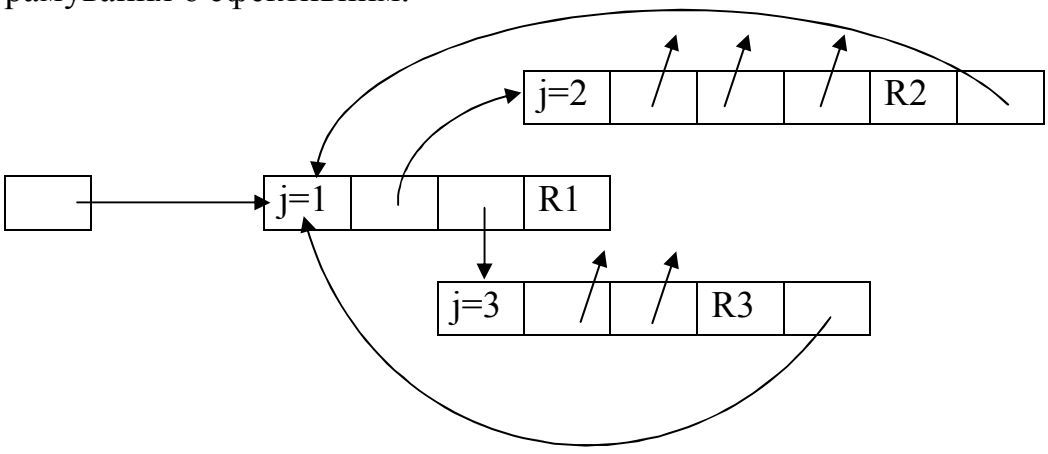
Дуги

1	2	$t_{12}=4$
1	3	$t_{13}=6$
2	3	$t_{23}=3$
2	4	$t_{24}=5$
2	5	$t_{25}=3$
3	4	$t_{34}=2$
3	5	$t_{35}=6$
4	5	$t_{45}=3$

Вершини

1	$R_1=0$
2	$R_2=4$
3	$R_3=7$
4	$R_4=9$
5	$R_5=13$

С. Спискова структура з оберненими зв'язками. Програмування на основі такої структури вимагає високої кваліфікації, проте лише таке програмування є ефективним.



2.7. Евристичні та генетичні алгоритми

2.7.1. Евристичні алгоритми

Розглянемо одну з класичних економічних задач – задачу комівояжера (salesman).

Комівояжер повинен відвідати клієнтів у *n* містах і повернутися назад. Кожне місто потрібно відвідати один раз. Задача полягає у знаходженні найкоротшого (найшвидшого, найдешевшого) туру.

Кількість можливих маршрутів (турів) у випадку 100 міст оцінюється числом 10^{155}. Отже, розв'язати задачу методом повного перебору неможливо. Іншого ж методу знаходження оптимального розв'язку цієї задачі на даний час не знайдено.

У таких ситуаціях задовольняються так званими евристичними алгоритмами. Ці алгоритми переважно знаходять хороші, хоча не завжди оптимальні розв'язки. Евристичні алгоритми будуються на методах часткових цілей (методах підйому).

Пронумеруємо міста числами 1, 2,…, N. Позначимо через $c_{ij}=c_{ji}$ вартість переїзду з *i*-ого міста в *j*-е.

Введемо такі змінні:

```
Var     k, i, j, u, v : 1..N; {номери міст}
        tour:array[0..n] of 1..N;  {тур, послідовність номерів міст}
        min, cost: real;   { вартість туру}
        visited: set of 1..N;  {множина вже відвіданих міст}
```

Запишемо евристичний алгоритм комівояжера на мові PASCAL:

```
Begin
    Readln(u);   {задаємо початкове місто}
   tour[0]:=u; visited:=u;  {тур розпочинається в місті u}
   v:=u;  cost:=0;   {v - поточне місто}
For k:=1 to N-1 do
  Begin  {*}
        Min:=65535;  {число, яке перевищує всі можливі значення cij}
        For i:=1 to n do
             if not (i in visited) then
                 begin  {**}
                       if c[v,i]<min then
                             begin min:=c[v,i]; j:=i end
                 end;   {**} {найменша віддаль серед невідвіданих міст}
        tour[k];=j; cost:=cost+c[v,u];
        visited:=visited+[j]; v:=j  {здійснюється переїзд в інше місто}
  End;   {*}
```

```
tour[u]:=u; {останній переїзд - переїзд в початкове місто}
cost:=cost+c[v,u];
. . . . . .
```

На останньому кроці віддаль може виявитися дуже великою. Програма знаходить хороший, хоча й не оптимальний розв'язок. На кожному кроці (в кожнім місті) алгоритм знаходив часткову ціль - знайти наступне невідвідане місто так, щоб переїзд до нього був найдешевшим.

В останній програмі була використана конструкція множини в мові Pascal. Описи множин (змінних множинного типу) у цій мові є такими:

```
Var
x: 1..3; {змінна x приймає одне із значень 1, 2, 3}
N, M: set of 1..3; {змінні N та M є множинами, що складаються із елементів 1, 2, 3}
```

Значеннями змінних n та m можуть бути такі множини:

[], [1], [2], [3], [1,2]=[2,1], [1,3], [2,3], [1,2,3].

Оператори над множинами в мові Pascal подамо таблицею:

Операції над множинами	Оператори в мові Pascal
$x \in M$	x in M
$N \cap M$	N*M
$N \cup M$	N+M
$N \subseteq M$	$N \leq M$

Приклади операторів над множинами:

M:=[2,3]; if (3 in M) then …; if $N \leq M$ then …

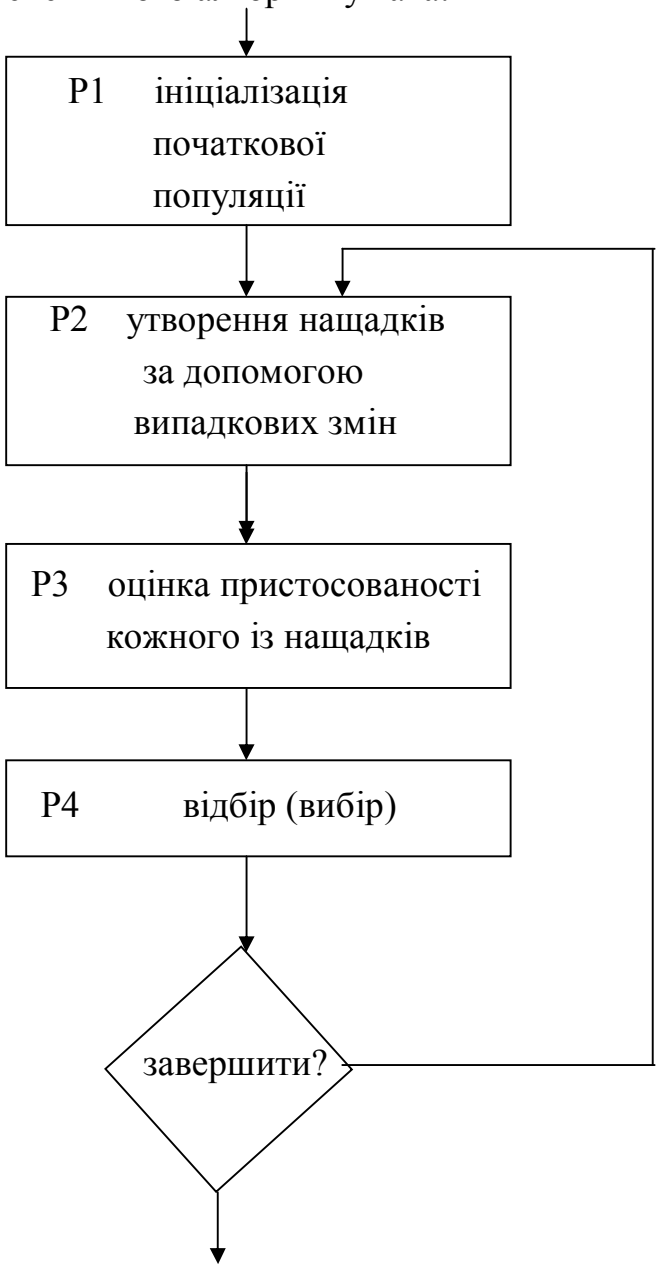
N:=M+[1,2]; {Результатом роботи оператора буде множина N=[1,2,3]}.

2.7.2. Генетичні алгоритми

Генетичні алгоритми генерують ряд кандидатів на розв'язок задачі; відбраковують погані варіанти і генерують нові варіанти, враховуючи досвід роботи алгоритму.

Таким чином кандидати на розв'язок поступово покращують свої характеристики – еволюціонують. Тому розв’язування задач за допомогою генетичних алгоритмів ще називають еволюційним численням (evolutionary computing).

Загальна схема генетичного алгоритму така:



Як видно із схеми, при розробці генетичного алгоритму потрібно прийняти конкретні рішення по кожній із чотирьох проблем P1, P2, P3, P4.

Розглянемо ідею генетичного алгоритму стосовно задачі комівояжера.

Розв'язком задачі та кандидатами на розв'язок є тури (послідовності міст).

Конкретизуємо проблеми P1, P2, P3, P4 для нашої задачі.

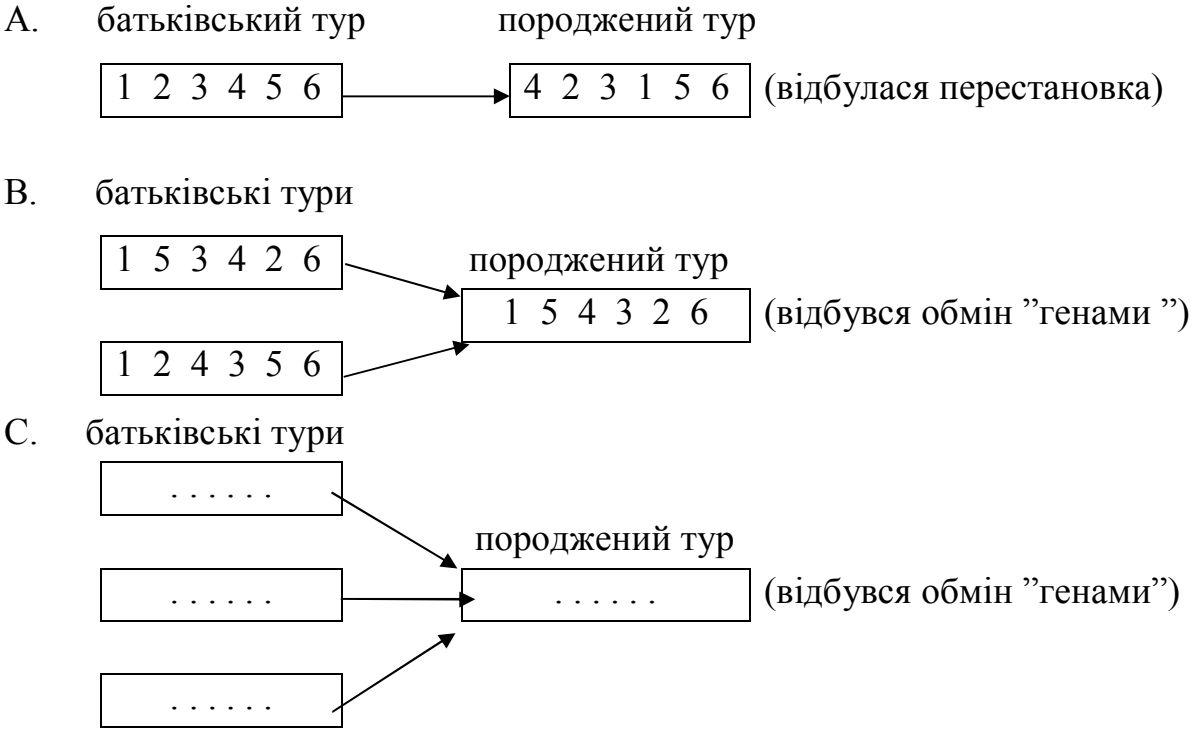
Проблема P3. Оцінку пристосованості здійснюють за допомогою цільової функції – загальної довжини туру (чи загальної вартості туру).

Проблема Р4. Правила відбору полягають у тому, щоб на кожному циклі залишати невелику кількість найкращих турів. Більш складні правила відбору можуть полягати в тому, що „слабші” розв'язки мають право існувати (жити) ще декілька поколінь (циклів). Розв’язання цієї проблеми є непростим як з точки зору програмування, так і з позицій розумного відбору даних.

Проблема Р1. Можливе або випадкове генерування початкового туру або інтуїтивний вибір більш-менш хорошого)одного чи декількох). Початкові розв’язки отримуються спеціалістами в конкретній предметній області.

Проблема Р2. Оператор випадкових змін, який на основі батьків генерує їхніх нащадків, полягає у випадковій перестановці міст (таким чином, щоб після перестановки знову утворювався тур).

Породження нового туру можливе на основі одного, двох чи навіть більшої кількості батьків:



Кожна з проблем має свою специфіку для кожної конкретної задачі. На даний час розробка хорошого генетичного алгоритму – це серйозна наукова задача.

Для зберігання інформації про популяції найчастіше використовують спискові структури.

Підсумкові запитання курсу

1. Вміти писати програми, аналогічні, до наведених у розділі 1.
2. Вміти будувати блок-схеми; виконувати граматичний аналіз та прокрутку таких програм.
3. Алгоритм, його властивості, способи задання.
4. Принципи фон Неймана функціонування комп'ютерів.
5. Алгоритмічна повнота (універсальність) ЕОМ (комп'ютерів) та алгоритмічних мов.
6. Задачі, що не розв'язуються алгоритмічно.
7. Теоретичні основи структурного програмування.
8. Основні ідеї структурного програмування.
9. Арифметичні типи даних в мовах програмування PASCAL, C++ та VBA.
10. Символьні типи даних в мовах програмування PASCAL, C++ , VBA.
11. Масиви (array, dim) в мовах програмування PASCAL, C++ , VBA.
12. Оператори розгалуження в мовах програмування PASCAL, C++ та VBA.
13. Оператори циклу (For та While) в мовах програмування PASCAL, C++ та VBA.
14. Опис процедури та оператор процедури в мові PASCAL.
15. Опис функції та покажчик функції мові PASCAL.
16. Записи та файли мові PASCAL.
17. Операції над послідовними файлами мові PASCAL.
18. Локалізація змінних в мовах програмування.
19. Алгоритм лінійного пошуку у впорядкованому та у невпорядкованому масиві.
20. Алгоритм пошуку методом половинного ділення, його характеристика.
21. Алгоритм пошуку у двійковому дереві, його характеристика.
22. Алгоритм сортування вставками, його характеристика.
23. Алгоритм сортування вибором, його характеристика.
24. Алгоритм сортування обмінами, його характеристика.

25. Алгоритм адресного сортування.
26. Алгоритм сортування за допомогою перемішаних таблиць, його характеристика.
27. Алгоритм знаходження часткових підсумків, його характеристика.
28. Алгоритм знаходження перетину двох таблиць (файлів).
29. Алгоритм знаходження об'єднання двох таблиць (файлів).
30 Алгоритм знаходження сполучення двох таблиць (файлів).
31. Алгоритм злиття двох впорядкованих файлів, його характеристика.
32. Метод половинного ділення розв'язування рівнянь f(x)=0.
33. Наближене обчислення визначених інтегралів.
34. Статичні та динамічні структури даних в мовах програмування.
35. Програма перегляду однонапрямленого списку, заданого конструкцією array.
36. Організація індексного доступу до файлу (індекс задано конструкцією array).
37. Змінні типу „покажчик (посилання)". Дії над ними.
38. Визначення (рекурсивне) спискової структури в мові Pascal.
39. Програма перегляду спискової структури (із використанням змінної типу „покажчик").
40. Алгоритм повернення назад (backtracing). Стек.
41. Організація індексного доступу до файлу (індекс задано списковою структурою з використанням змінної типу „покажчик").
42. Алгоритм побудови критичного шляху в графі.
43. Евристичні алгоритми.
44. Генетичні алгоритми. Основні проблеми їх побудови.

Література

1. Щедріна О.І. Алгоритмізація та програмування процедур обробки інформації: Навч. посібник – К.: КНЕУ, 2001. – 240 с.

2. Єжова Л.Ф. Алгоритмізація і програмування процедур обробки інформації: Навч.-метод. посібник для самост. вивч. дисц. – К.: КНЕУ, 2000. - 152 с.

3. Семотюк В. Програмування в середовищі Турбо Паскаль. Львів: Бак, 2000. – 248 с.

4. Костів О.В., Ярошко С.А. Методи розробки алгоритмів: Тексти лекцій. – Львів: ЛНУ, 2002. – 98 с.

5. Костів О. Структури даних. – Львів, ЛНУ, 2000. – 56 с.

6. Лук'янова В.В. Комп'ютерний аналіз даних. – К.: Академія, 2003. – 342 с.

7. Інформатика. Комп'ютерна техніка. Комп'ютерні технології: Підручник /За ред. О. І. Пушкаря. – К.: „Академія", 2002. - 704 с.

8. Завада О.П. Методичні вказівки до виконання контрольних робіт з курсу „Інформатика і комп'ютерна техніка". – Львів: ЛНУ, 2003. - 24 с.

9. Завада О.П. Методичні вказівки до виконання контрольних робіт з курсу „Алгоритмізація і програмування". – Львів: ЛНУ, 2004. - 68 с.

10. Кнут Д. Искусство программирования для ЭВМ. Т. 3. Сортировка и поиск. – М.: Мир, 1977 (переклад з англійської на російську).

11. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы : Построение и анализ. – М.: МЦНМО, 1999 (переклад з англійської на російську).

12. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. – М.: Мир, 1979 (переклад з англійської на російську).

13. Дал У., Дейкстра Э., Хоор К. Структурное программирование. – М.: Мир, 1975 (переклад з англійської на російську).

14. Евстигнеев В.А. Применение теории графов в программировании. – М.: Наука, 1985.

15. Кристофидес Н. Теория графов. Алгоритмический поход. – М.: Мир, 1978 (переклад з англійської на російську).

16. Майника Э. Алгоритмы оптимизации на сетях и графах. М.: Мир, 1981 (переклад з англійської на російську).

Зміст

Н а в ч а л ь н е в и д а н н я

Завада Олександр Петрович

АЛГОРИТМІЗАЦІЯ І ПРОГРАМУВАННЯ

Тексти лекцій

Підписано до друку 23.07.2004. Формат 60x84/16. Папір друк. № 3.
Різогр. друк. Умовн. друк. арк. 4,75. Наклад 75 прим. Зам. № 253.

Видавничий центр Львівського національного університету імені Івана Франка
79000, Львів, вул. Дорошенка, 41