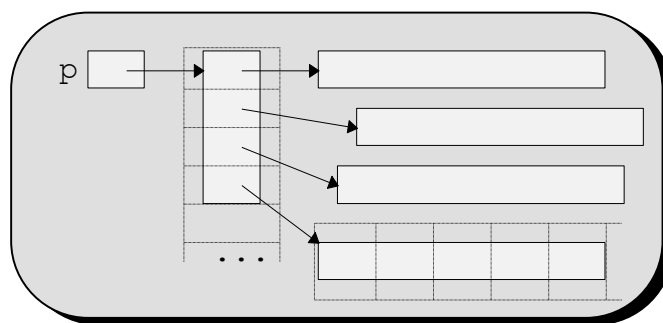


Міністерство освіти України
Львівський національний університет імені Івана Франка

Р. В. Гудзь, С. А. Ярошко

Використання динамічних структур даних у програмах на Borland Pascal

Тексти лекцій
Серія ТЛ № 10/99



Львів ЛНУ ім. Івана Франка 2000

Гудзь Р. В., Ярошко С. А. Використання динамічних структур даних у програмах на Borland Pascal: Тексти лекцій. – Львів: Ред.-вид. відділ ОЦ ЛНУ ім. І. Франка, 2000.

У текстах лекцій викладено правила оголошення та основні прийоми використання змінних вказівних типів у мові програмування Borland Pascal. Запропоновано способи реалізації таких динамічних структур даних, як список, стек, черга, дерево. Наведено приклади їх використання. Тексти лекцій проілюстровано великою кількістю програм і схем.

Для студентів факультету прикладної математики та інформатики, механіко-математичного факультету, усіх, хто цікавиться питаннями розробки алгоритмів і використання структур даних.

Рекомендовано до друку
науково-методичною радою факультету
прикладної математики та інформатики
протокол № 7 від 25.11.99

Рецензент

П. П. Вагін, канд. фіз.-мат. наук
(Львівський національний університет)

Відповідальний за випуск

В. В. Черняхівський

Редактор

Н. Й. Плиса

© Р. В. Гудзь, С. А. Ярошко, 1999

ЗМІСТ

1. Статичні та динамічні змінні
2. Вказівні типи – інструмент для використання динамічних змінних
 - 2.1. Оголошення вказівних типів
 - 2.2. Операція взяття адреси
 - 2.3. Порожній вказівник. Перетворення типів вказівників
 - 2.4. Операції над значеннями вказівних типів
 - 2.5. Приклад доступу до змінної за вказівником
 - 2.6. Створення динамічних змінних
 - 2.7. Визначення розміру вільної динамічної пам'яті
 - 2.8. Знищення динамічних змінних
3. Проблема втрачених посилань
4. Використання масивів вказівників
 - 4.1. Використання масивів вказівників на числа
 - 4.2. Обробка «довгого» тексту
5. Структура даних «список»
 - 5.1. Види списків
 - 5.2. Реалізація дій з однонапрямленим списком
 - 5.3. Зображення розріджених поліномів за допомогою списків
6. Нетипізовані вказівники
 - 6.1. Використання масивів змінної довжини
 - 6.2. Двонапрямлений лінійний список
7. Структури даних «черга» та «стек»
 - 7.1. Реалізація черги
 - 7.2. Імітаційні алгоритми
 - 7.3. Моделювання руху на регульованому перехресті
 - 7.4. Реалізація стека
8. Структура даних «дерево»
 - 8.1. Означення дерева
 - 8.2. Рекурсія
 - 8.3. Нерекурсивний алгоритм обходу дерева
 - 8.4. Дерева пошуку
 - 8.5. Пошук з включенням у дереві
 - 8.6. Вилучення з дерева пошуку
 - 8.7. Збалансовані дерева
 - 8.8. Вставка у збалансоване дерево
 - 8.9. Вилучення зі збалансованих дерев
9. Завершальний розділ
10. Список літератури

Ти, малий, скажи малому, хай малий
малому скаже, хай малий теля прив'яже
Українська народна скоромовка

Динамічними називають структури даних, які здатні змінювати свій розмір і склад під час виконання програми. Дуже часто лише з їхньою допомогою вдається адекватно відобразити дані задачі.

Сучасна об'єктно-орієнтована технологія розробки програм використовує об'єкт як базовий елемент. Кожен об'єкт має свій життєвий цикл. Він створюється програмою, взаємодіє з іншими об'єктами програми, ліквідується, коли відпала потреба у ньому. Тобто, об'єкти в програмі використовуються динамічно. У більшості випадків вони також є елементами певних динамічних структур даних.

Підтримання зручного багатовіконного інтерфейсу, ефективне розв'язання задач сортування та пошуку, гнучке пристосування програми до змінного потоку вхідних даних – можна ще і ще продовжувати перелік задач, де використовують динамічні структури даних.

Засоби динамічного створення змінних, їхнє використання у мові програмування Паскаль реалізовано за допомогою змінних вказівних типів. У цьому посібнику викладено відповідні синтаксичні правила, описано основні прийоми застосування вказівників. На різноманітних прикладах проілюстровано використання таких динамічних структур даних як списки, стеки, черги, дерева (у тому числі збалансовані дерева).

Щоб успішно засвоїти матеріал посібника, достатньо володіти основними засобами мови Паскаль: використанням процедур та функцій, структурованих типів даних, зовнішніх файлів тощо.

1. Статичні та динамічні змінні

Мабуть, кожному доводилось писати програму відшукування найбільшого елемента заданої послідовності чисел. Пригадайте, у такій програмі було оголошено кілька змінних: масив для зберігання послідовності, параметр циклу для перебирання її членів, змінну для максимального значення. Ці оголошення можуть виглядати, наприклад, так:

```
var a: array[1..20] of real; {структура даних «масив»}
    i: integer;           {індекс, параметр циклу }
    max: real; ...       {шукане значення }
```

Усі ці змінні є статичними: потребу в них можна передбачити ще на етапі проектування алгоритму і оголосити в розділі змінних програми. Під час трансляції компілятор виділить для них пам'ять у сегменті даних програми. У процесі виконання програми ні місце розташування зазначених змінних, ні їхній розмір змінюватися не будуть. Саме тому їх називають *статичними змінними*.

До статичних відносять також змінні, оголошені у процедурах і функціях, оскільки їхня кількість і тип не змінюються під час виконання програми. Ці змінні розташовуються в сегменті стека.

Для багатьох задач можна точно сказати, скільки і яких змінних потрібно оголосити в програмі. Проте характерною є ситуація, коли програміст не в змозі передбачити усі потреби у змінних. Наприклад, якщо обсяг пам'яті, необхідний для збереження даних деякої програми, змінюється в ході її виконання, причому, як у бік збільшення, так і в бік зменшення. У цих випадках використовують змінні іншої категорії – динамічні – появою і часом існування яких керує програміст, відповідно до потреб задачі. Пам'ять для динамічної змінної виділяється під час виконання програми. Як тільки потреба в ній відпадає, динамічна змінна може бути знищена, а зайнята нею пам'ять – вивільнена для повторного використання з іншою метою.

Є, принаймні, дві причини для використання динамічних змінних. По-перше, воно може бути єдиним способом ефективної реалізації алгоритму, поведінка якого суттєво залежить від вхідних даних, які постійно змінюються (наприклад, організація віконного інтерфейсу програми з користувачем). По-друге, реальні обчислювальні задачі часто використовують масиви даних, що значно перевищують за обсягом пам'яті можливості сегмента даних та стека (по 64 Кбайт кожен). Їх можна розмістити лише в іншій частині пам'яті, яка називається *кupoю* (heap), або динамічною областю пам'яті.

Зрозуміло, що динамічний розподіл сприяє гнучкому і бережливому використанню пам'яті, хоча самі операції резервування і вивільнення областей пам'яті, які відбуваються під час виконання програми, неминуче уповільнюють її роботу.

Доступ до статичних змінних виконується через їхні ідентифікатори. Цей підхід не може бути використаний для динамічних змінних, оскільки наперед невідомі їх кількість і розташування в пам'яті. Доступ до них у Паскалі реалізують за допомогою особливих змінних, які називають *вказівниками*. Призначення їх полягає у тому, щоб вказувати на місце розташування якоїсь іншої змінної визначеного типу. Вказівник є посередником між програмою і динамічною змінною. Сам вказівник, як звичайно, є статичною змінною. Його створює компілятор. Значеннями всіх вказівників є адреси пам'яті. Якщо програма містить команду створення динамічної змінної, то її виконання спричиняє виділення ділянки пам'яті для цієї змінної і присвоювання адреси виділеної ділянки вказівникові. Далі можна звертатись до новоствореної змінної через вказівник.

Звертання до динамічної змінної за допомогою вказівника можна порівняти зі звертанням до студента за номерами ряду і місця, яке він займає в аудиторії, або з проханням, переданим через «малого» іншому «малому» «прив'язати теля».

2. Вказівні типи – інструмент для використання динамічних змінних

Як уже зазначено, мова програмування Паскаль реалізує операції з динамічними змінними за допомогою змінних вказівних типів. У цьому параграфі описані правила оголошення вказівників, операції зі значеннями вказівних типів, основні способи їх використання.

2.1. Оголошення вказівних типів

Вказівні типи займають проміжне становище між простими і структурованими типами. Значення вказівного типу, як і значення простих типів (наприклад, типу `integer`, `real` чи ін.), є атомарними, нероздільними. Сам же вказівний тип, як усі структуровані, визначається через інший, базовий для нього тип.

Оголошення вказівного типу має вигляд
 $\text{^} <\text{ідентифікатор типу}>$

Наведемо кілька прикладів:

```
type Vector=array[1..10]of real;
      Pti=^integer;
      Pts=^string;
      PtV=^Vector;
```

Тип `Pti` визначає множину вказівників на значення цілого типу, `Pts` – на значення рядкового типу, а тип `PtV` – множину вказівників на значення регулярного типу `Vector`.

У мові програмування Паскаль діє правило, згідно з яким усі ідентифікатори повинні бути описані перед їхнім використанням. Для вказівних типів зроблено виняток: базовий тип може бути оголошений після оголошення вказівного, але обов'язково в тому самому розділі типів. Тобто, синтаксично правильними є оголошення, подібні до такого:

```
type PtrType =^BaseType;
      BaseType=record x,y:real end;
```

Тут для опису вказівного типу `PtrType` використано ім'я невизначеного попередньо базового типу `BaseType`. Воно визначається пізніше, але обов'язково у тому ж розділі типів, що і `PtrType`.

Запровадження такого винятку – вимушений крок, бо інакше неможливо було б описати ланку динамічної структури даних список (див. 4).

Зауважимо, що базовими для вказівних типів можуть бути будь-які, в тому числі і вказівні. Тому допустимим є використання *вказівників на вказівники*. Саме цей випадок й ілюструє скоромовка, винесена в епіграф до текстів лекцій.

2.2. Операція взяття адреси

Реально значеннями вказівних типів є адреси в пам'яті, за якими розміщені конкретні значення базового типу. Для того щоб присвоїти змінній вказівного типу певне значення, потрібно скористатись *унарною операцією взяття вказівника*, яка будується зі знака цієї операції – символу '@' і одного оператора – змінної базового типу, як зображено нижче:

$\text{@} <\text{ідентифікатор}>$

Наприклад, якщо в програмі оголошені такі змінні:

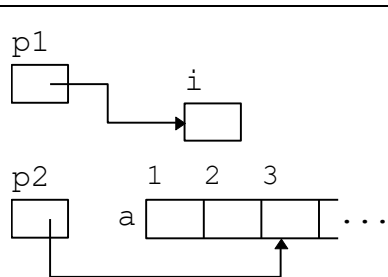


Рис.1. Операція взяття вказівника.

```
var i: integer;
    a: array[1..10] of integer;
    p1, p2: Pti; ,
```

то в результаті виконання операторів

```
p1:=@i; p2:=@a[3];
```

значенням змінної p1 стане адреса змінної i, а змінної p2 – адреса третього елемента масиву a. Схематично це зображено на рис. 1 (тип Pti оголошено раніше).

Може виникнути бажання чи потреба побачити, роздрукувати ці адреси – значення змінних p1 і p2. Тип Ptr не є стандартним, тому значення змінних цього типу не можна роздрукувати за допомогою процедури Write чи WriteLn. Що робити? По-перше, значення змінної p1 можна переглянути, занісши її до вікна Watch інтегрованого середовища Турбо Паскаль. По-друге, можна використати операцію зведення типу і перетворити p1 у змінну стандартного типу. Всі адреси в системі IBM PC мають формат

⟨сегмент⟩:⟨зміщення⟩,

де ⟨сегмент⟩ – номер сегмента оперативної пам'яті, ⟨зміщення⟩ – адреса стосовно початку сегмента. Обидві складові адреси займають по два байти, тому змінну p1 можна привести до типу LongInt:

```
WriteLn(LongInt(p1));
```

Також можна використати інший підхід. А саме, інтерпретувати складові адреси як поля змінної комбінованого типу:

```
type Adress=record Seg, Ofs: word end;
```

```
.....
begin with Adress(p1) do
  WriteLn(Seg, ':', Ofs);
  .....
```

2.3. Порожній вказівник. Перетворення типів вказівників

Серед усіх можливих вказівників у мові виділено один спеціальний, який "нікуди не вказує". Можна вважати, що він містить адресу ділянки оперативної пам'яті, в якій не може бути розташована жодна змінна. Такий порожній вказівник позначається службовим словом **nil**. Він вважається константою, сумісною з довільним вказівним типом. Тобто значення **nil** можна присвоювати змінній будь-якого вказівного типу. Наприклад:

```
var a:^byte; b:^real; c:^array[1..10] of integer;
begin ... a:=nil; ... b:=nil; ... c:=nil; ...
```

Вказівні типи, утворені від різних базових типів, відповідно до загальних правил мови Паскаль, вважаються різними. Хоча фізична природа вказівних типів у всіх випадках передбачає адреси в оперативній пам'яті, описані нижче типи PRec і PLong задають різні множини значень

```

type Rec=record Hi, Lo: word end;
    PRec=^Rec;
    PLong=^LongInt;
var p, q: PRec; r: PLong;
    b: Rec; x: LongInt;

```

Тому змінній `p` не можна присвоїти значення змінної `r` і навпаки. Для вказівних змінних одного типу (змінні `p` і `q`) такі дії допустимі:

```
p:=@b; q:=p;
```

Для вказівних типів, так само як і для простих, допускаються конструкції явного перетворення. Наприклад, використовуючи приведення типу, можна записати:

```
PRec(r) := p; PLong(q) := @x;
```

У першому операторі вказівник `r` на змінну довгого цілого типу перетворюється у вказівник на запис і отримує як значення адресу запису `b` (із вказівника `p`). У другому – вказівнику на запис присвоюється адреса довгого цілого `x`.

Використовувати перетворення типів вказівників треба досить обережно і лише тоді, коли це справді необхідно. Оскільки базові типи вказівників можуть мати різну довжину, а компілятор не перевіряє, на яку ділянку пам'яті вказує такий перетворений вказівник, то наслідки можуть бути непередбачуваними.

Крім описаних вказівних типів, Турбо Паскаль містить також і "універсальний" вказівний тип `pointer`, який не конкретизує базового типу, тобто таких значень, на які він вказує. Значення типу `pointer` називають нетипізованими вказівниками. Їхнє використання буде описане у восьмому параграфі.

2.4. Операції над значеннями вказівних типів

Над значеннями вказівних типів допускаються операції порівняння на рівність і нерівність. Вони перевіряють, чи вказують два вказівники на одне і те саме місце в пам'яті, і позначаються звичайним чином – знаками "=" і "<>". Звичайно, у порівнянні можуть брати участь два вказівники лише одного типу. У разі потреби порівняння вказівників різних типів необхідно використати явне перетворення типу, як описано в попередньому параграфі. Наведемо приклади деяких можливих порівнянь:

```

if p=q then ...; {p і q описані раніше}
Sign:=r=nil;      {змінна Sign: Boolean}
while (PRec(r)<>p) and (p<>nil) do ...

```

Нагадаємо, що порівнювати треба лише вказівники, визначені до моменту порівняння (як і змінні будь-яких інших типів). Значення вказівник може одержати одним із трьох способів:

- 1) у результаті виконання операції `@` взяття вказівника;
- 2) у результаті виконання оператора присвоювання;
- 3) у результаті виконання процедури `New`, яка буде описана в наступному параграфі.

Ще однією, не менш важливою дією над значеннями вказівників, є доступ до змінної за вказівником шляхом розіменування. Розглянемо ще раз схему, зображену на рис. 1. Після виконання оператора $p1 := @i$ для доступу до змінної i є дві можливості: через ідентифікатор i та через записану в $p1$ адресу цієї змінної. Щоб реалізувати другу з них, тобто отримати доступ до змінної через вказівник, потрібно використати *операцію розіменування*. Для того після імені змінної-вказівника треба написати символ $^$. Запис $p1^$ означає "змінна, на яку вказує $p1$ ". За означенням, розіменування має тип, що збігається з базовим типом вказівника, тобто $p1^$ є змінною цілого типу, яка збігається в пам'яті зі змінною i .

Щоб збільшити значення i на 1, можна скористатись одним з операторів:

```
i:=i+1;      {доступ до змінної через ідентифікатор}
p1^:=p1^+1; {доступ до змінної за вказівником}
p1^:=i+1;   {обидва способи доступу}
```

Доступ до третього елемента масиву a , зображеного на рис. 1, можна зробити так:

$a[3]$..., або $p2^$..., або $i:=3; a[i]$..., або $i:=3; a[p1^]$... і т. д.

Розіменування вважається **некоректним**, якщо вказівна змінна має значення **nil**. У цьому випадку не існує змінної, на яку посилається вказівник, тому оператори вигляду

```
p1:=nil; p1^:=2;
```

хоч і не приводять до аварійної зупинки виконання, проте є недопустимими, а іноді і потенційно небезпечними для подальшого ходу виконання програми.

2.5. Приклад доступу до змінної за вказівником

Розглянемо фрагмент програми [7]

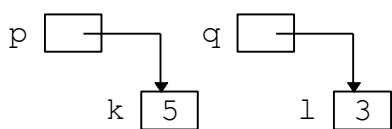


Рис. 2. Вказівники на статичні змінні.

```

.....
type ref=^integer;
var p,q: ref; k,l: integer;
begin ..... p^:=q^;           {A}
if{1} p=q then p:=nil
else if{2} p^=q^ then q:=p;   {B}
if{3} p=q then q^:=4;       {C}
.....
WriteLn(k,l); WriteLn(p^,q^);
```

Нехай змінні p , q , k і l мають значення як показано на рис. 2. Дамо відповіді на такі запитання:

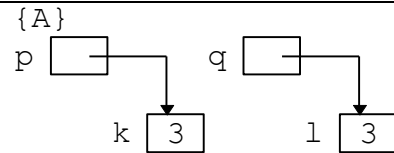
- ? що є значенням змінної p ? Що означає змінна $p^$? Які типи змінних p і $p^$?
- ? що буде надруковано в результаті виконання наведених вище операторів?

Відповідь.

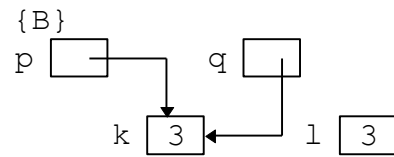
! Значенням змінної p є адреса змінної k . Змінна $p^$ означає змінну типу `integer`, її значенням є число 5. Змінна p має тип `integer`, а змінна $p^$ – "вказівник на ціле".

! Буде надруковано "4 3" в першому рядку і "4 4" – у другому. На рис. 3 схематично показано послідовність дій, спричинених операторами {A}, {B}, {C}.

Змінні $p \wedge i \wedge q$ - це цілі змінні.



Умова {1} = false, бо $p \wedge i \wedge q$ вказують на різні змінні; умова {2} = true, бо значення $p \wedge i \wedge q$ однакові після виконання {A}.



Умова {3} = true.

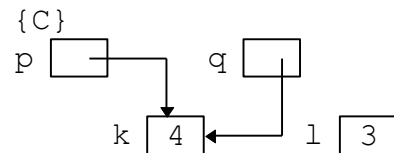


Рис. 3. Хід виконання програми.

2.6. Створення динамічних змінних

Створення і знищення динамічних змінних реалізується в Турбо Паскалі стандартними процедурами: відповідно `New` і `Dispose`.

Щоб створити нову динамічну змінну певного типу, потрібно викликати процедуру `New`, вказавши як параметр ім'я змінної-вказівника. Процедура діє так:

- у купі відводиться місце для зберігання змінної, тип якої збігається з базовим типом вказівника-параметра;
- вказівникові присвоюється адреса початку відведеної ділянки пам'яті.

Використання процедури `New` продемонструємо на прикладі виконання такого завдання: оголосити типи та змінні, записати оператори потрібні для створення вказівника і динамічних змінних, зображених на рис. 4.

Розв'язок

```

type Combi=record a: Boolean;
                    b,c: ^real end;
var r: Combi;           {A}
begin New(r);           {B}
with r^ do
begin a:=false; New(b); {C}
        b^:=2.7; c:=nil  {D}
end; {with}
  
```

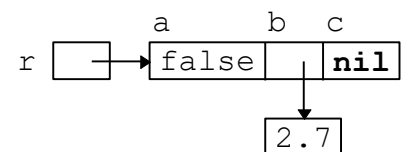
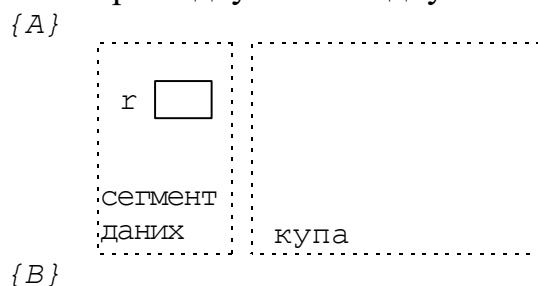


Рис. 4. Динамічна структура.

Прослідкуємо як відбувається створення змінних.



Вказівник `r` – статична змінна, пам'ять для якої відводиться компілятором в сегменті даних.

У результаті виконання `New(r)` в купі виділено пам'ять для змінної типу `Combi`, `r` містить вказівку на неї.

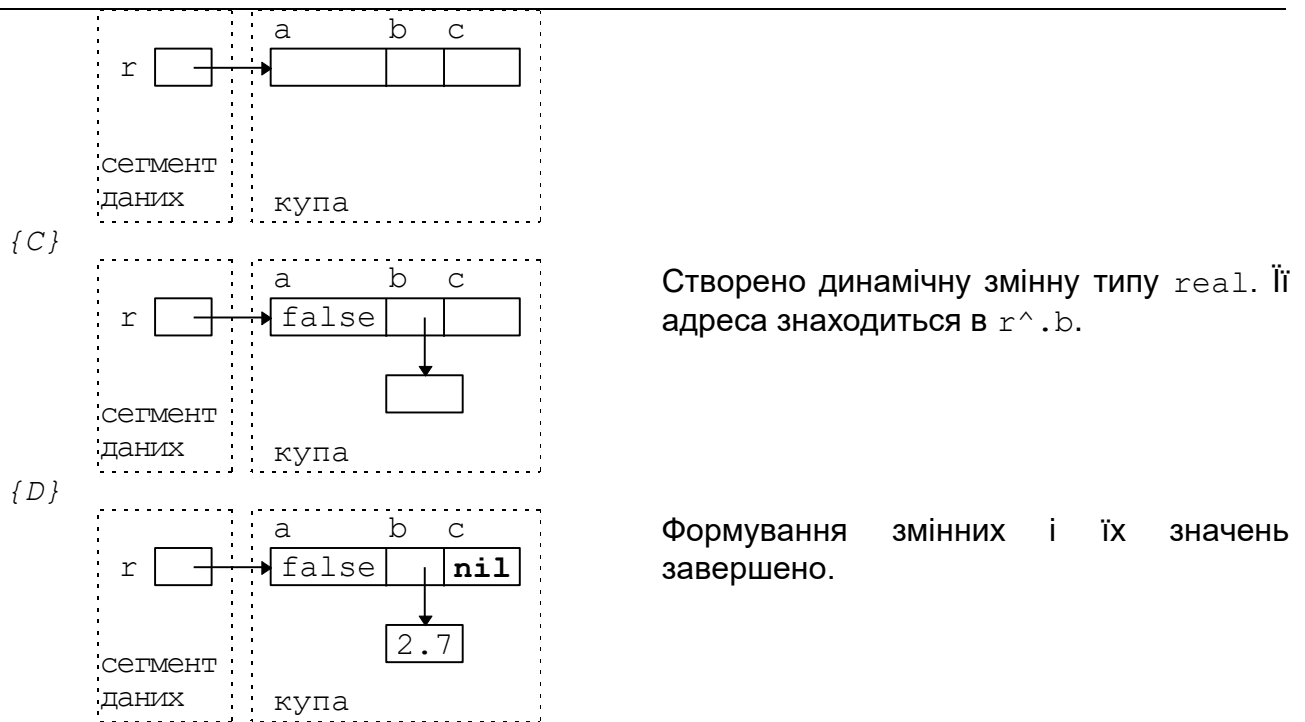


Рис. 5. Послідовність формування динамічних змінних.

Мова Турбо Паскаль дає змогу також використовувати `New` у функціональній формі. У цьому варіанті параметром функції `New` є ім'я вказівного типу, а результатом функції є вказівник цього типу, що посилається на відведену область для значення базового типу. В багатьох випадках такий спосіб використання `New` є зручнішим і наочнішим. Еквівалентом наведеного вище прикладу буде такий фрагмент:

```

type PReal=^real; PCombi=^combi;
     combi=record a: Boolean; b,c: PReal end;
var r: PCombi;
begin r:=New(PCombi); with r^ do
  begin a:=false; b:=New(PReal);
        b^:=2.7; c:=nil end; {with} .....
```

2.7. Визначення розміру вільної динамічної пам'яті

У випадку створення великої кількості динамічних змінних може постати проблема вичерпання області динамічної пам'яті. Якщо під час виконання процедури (функції) `New` виявиться, що для розміщення в купі не вистачає місця, то значення вказівника-параметра не зміниться. **Виконання програми не зупиниться, ніяких повідомлень видано не буде. Подальша робота з невстановленим вказівником може привести до непередбачуваних наслідків.** Для підвищення надійності програми перед створенням нових динамічних змінних треба перевіряти обсяг вільної динамічної пам'яті. Зробити це можна, використавши одну зі стандартних функцій: `MaxAvail` або `MemAvail`.

Функція `MaxAvail` повертає максимальний розмір у байтах найбільшої вільної ділянки, придатної для розміщення динамічних змінних.

Функція `MemAvail` повертає сумарний розмір усіх вільних ділянок динамічної пам'яті.

У загальному випадку ці дві функції дають різні результати, оскільки внаслідок довільного способу відведення і звільнення пам'яті купа може бути фрагментована на зайняті і вільні ділянки.

Щоб пересвідчитись у можливості створення динамічної змінної, потрібно знати обсяг необхідної для неї пам'яті. Для стандартних типів ці числа відомі і наведені в літературі [1, 5, 10]. Наприклад, змінна типу `integer` займає 2 байти, типу `real` – 6 байтів і так далі. Розмір змінної складного типу можна обчислити вручну, додавши розміри всіх її складових. Такий спосіб визначення розміру змінних не надто зручний і не гарантує від помилок, тому краще використати стандартну функцію `SizeOf`.

Функція `SizeOf` повертає як результат розмір у байтах змінної, яка вказана параметром цієї функції. Параметром функції `SizeOf` можна також вказати ім'я типу.

Наприклад, якщо програма містить такі оголошення

```
type Rec=record a: integer; b: real; c: Boolean;
                d: array[1..10]of char end; {record}
var x: word; y: real;
    z: record Re, Im: real end;
    Elem: Rec;
```

то для того, щоб змінній `x` присвоїти сумарний розмір усіх оголошених змінних, потрібно виконати оператор

```
x:=SizeOf(x)+SizeOf(y)+SizeOf(z)+SizeOf(Elem);
```

Змінна `x` отримає значення 39, оскільки доданки в наведеному операторі мають значення відповідно 2, 6, 12 і 19.

Щоб визначити розмір довільної змінної типу `Rec`, треба викликати функцію `SizeOf`, вказавши параметром ім'я типу: `x:=SizeOf(Rec)`. Зрозуміло, що вказувати можна імена будь-яких відомих типів у тому числі і стандартних: `SizeOf(real)`, `SizeOf(LongInt)` і т. д.

Таким чином для створення змінних попереднього прикладу можна запропонувати таку послідовність дій:

```
if MaxAvail>=SizeOf(Pcombi) then
  begin New(r); with r^ do
    begin a:=false;
      if MaxAvail>=6 then begin New(b); b^:=2.7 end
        else WriteLn('Бракує купи для поля b');
      c:=nil end {with}
    end else WriteLn('Вичерпано купи.');
```

Зауваження. Функція `SizeOf` виконується на етапі трансляції програми. Це означає, що її використання не збільшує часу виконання готової програми, а також і те, що результати, повернені функцією, залишаються незмінними протягом усього часу роботи програми.

Якщо виникла така ситуація: функція `MaxAvail` повертає розмір, менший за потрібний для нової динамічної змінної, а результат функції `MemAvail` свідчить

про те, що сумарний обсяг вільної динамічної пам'яті досить великий, то це означає, що в даний момент купа сильно фрагментована, і потрібно зайняти ділянки пам'яті ущільнити так, щоб утворилась єдина незайнята. Така задача в програмуванні має назву "збирання сміття". Опис способів її розв'язання не є предметом вивчення текстів лекцій. Його можна знайти, наприклад, в [10].

2.8. Знищення динамічних змінних

Для вивільнення пам'яті, відведеної за допомогою процедури `New` або функції `New`, використовується симетрична їм стандартна процедура `Dispose`. Її виклик у загальному випадку має вигляд:

```
Dispose(R);
```

де `R` – вказівник на динамічну змінну, тобто змінна вказівного типу, значенням якої є адреса динамічної змінної, розміщеної раніше в купі за допомогою `New`.

У інших випадках, наприклад, коли вказівник є невизначений, чи вказує на статичну змінну, або параметром є `nil` – виклик процедури `Dispose` призводить до аварійної зупинки виконання програми.

Отже, послідовність дій з деякою динамічною змінною вкладається в таку схему:

```
var P: ^aBaseType;
begin New(P);
      { використання вказівника P,
        робота з динамічною змінною }
      Dispose(P)
end.
```

3. Проблема втрачених посилань

Використання динамічних змінних потребує від програміста особливої уважності. Іноді синтаксично правильні, однак некоректні з погляду логіки програми дії з вказівниками можуть призвести до виникнення помилок, які буває важко виявити.

Розглянемо дві характерні помилкові ситуації, що можуть виникати в процесі некоректної роботи з динамічними змінними за допомогою вказівників.

```
var p, q: ^integer;
begin { фрагмент 1 }
  New(p); p^:=1;
      { створили першу змінну }
  New(q); q^:=2;   { і другу }
  p:=q;           { першу втратили ! }
  .....
  { фрагмент 2 }
  New(p); p^:=1;   { створили динамічну змінну }
  q:=p;           { до неї є два шляхи }
  Dispose(p);     { змінну знищили, а що таке q^ ? }
  .....
```

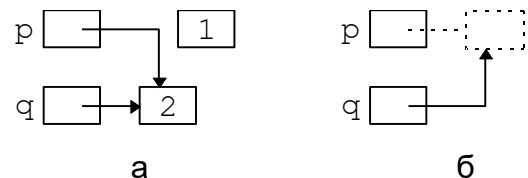


Рис. 6. Некоректне використання динамічних змінних.

Результати виконання цих фрагментів схематично зображено на рис. 6.

У першому випадку відбулася втрата адреси першої динамічної змінної. Відведена для неї ділянка пам'яті стала недоступною. Вона залишатиметься зайнятою аж до завершення виконання програми. "Засмічення" пам'яті такими недоступними динамічними змінними може призвести до завчасного вичерпання всієї купи і неможливості подальшого правильного виконання програми.

У другому випадку q вказує на неіснуючу змінну і робота з q^{\wedge} може призвести до несподіваних результатів, якщо ділянка пам'яті, на яку вказує q , буде використана для розміщення нових динамічних змінних. У такій ситуації поведінка програми залежатиме не від її семантики, а від системних особливостей функціонування, тобто від способу реалізації мови Паскаль у конкретній системі програмування.

Описані помилки можуть траплятися і в менш очевидних ситуаціях. Розглянемо, наприклад, фрагмент програми

```

type iPtr=^integer;
procedure MakeInt1(var q:iPtr; x:integer);
  begin q:=New(iPtr); q^:=x end;
procedure MakeInt2(x:integer);
  var q:iPtr;
  begin q:=New(iPtr); q^:=x end;
procedure MakeInt3(q:iPtr; x:integer);
  begin q:=New(iPtr); q^:=x end;
var p,r:iPtr;
begin MakeInt1(p,1);           {помилка нема}
  MakeInt1(2);               {динамічна змінна недоступна, бо локальна}
  {змінна q перестала існувати після завершення MakeInt2}
  MakeInt1(r,5); ... {динамічна змінна недоступна, бо r -}
  {це параметр-значення, він залишився невизначеним}

```

Процедура MakeInt1 написана без помилок. У результаті її виконання буде створено динамічну змінну цілого типу, ініціалізовану значенням 1. Параметр-змінна p отримає вказівник на неї (рис. 7, а).

Виконання MakeInt2 спричинить втрату динамічної змінної, оскільки вказівник на неї було присвоєно локальній змінній q . Як відомо, локальні змінні підпрограм розташовуються у сегменті стека лише на час їх виконання і стають недоступними після його завершення (рис. 7, б).

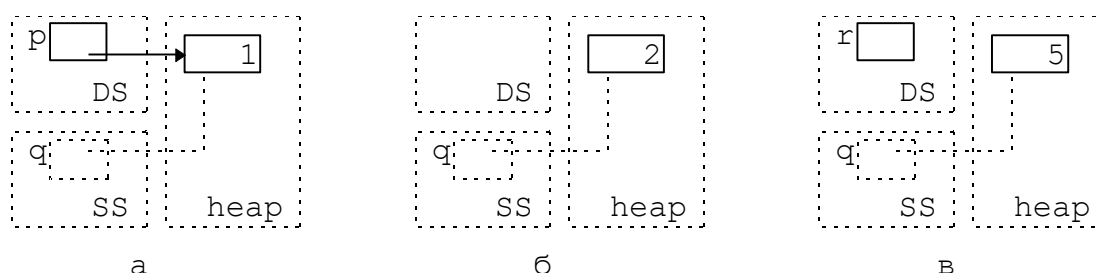


Рис. 7. Втрата динамічних змінних у підпрограмах.

Виконання MakeInt3 теж призведе до втрати динамічної змінної, бо її параметр q оголошено як параметр-значення. Присвоєння йому вказівника на

новостворенну змінну ніяк не впливає на значення фактичного параметра r (рис. 7, в).

Завершимо обговорення помилкових ситуацій ще одним прикладом

```

var p: ^integer;           {глобальна змінна}
procedure InitInteger;
  var i: integer; {локальна змінна, розташована в SS}
begin i:=0; p:=@i end;

```

Виклик процедури `InitInteger` призведе до того, що змінна `p` вказуватиме на неіснуючу змінну `i`. Поміркуйте, чому.

4. Використання масивів вказівників

Масив є тією структурою даних, яка, мабуть, найчастіше використовується в багатьох мовах програмування. Нагадаємо, що в Турбо Паскалі масиви – це змінні регулярних типів. У тих випадках, коли масив повинен містити багато елементів з невеликою кількістю можливих значень (наприклад, масив зі ста нулів і одиниць), можна запропонувати інший спосіб його реалізації. Наприклад, можна утворити масив вказівників, елементи якого містять адреси розташування в пам'яті відповідних значень.

Нижче наведено два приклади використання масивів вказівників для представлення масивів дійсних чисел з великою кількістю однакових елементів і великого за розміром тексту.

4.1. Використання масивів вказівників на числа

У деякій програмі зроблено такі оголошення:

```

const n=10;
type ref=^real; range=1..n;
vector=array[range]of ref;

```

Вважаючи, що всі елементи вектора `x` (змінної `x` типу `vector`) відмінні від `nil`, описати:

- ? функцію `Max(x)` для знаходження найбільшого з чисел, на які посилаються елементи вектора `x`;
- ? процедуру `Unique(x)`, яка у векторі `x` всі елементи, що посилаються на рівні числа, замінює на перший з цих елементів.

Розв'язок.

```

function Max(var x: vector):real;
  { Max знаходить найбільше з чисел, на які посилаються }
  { елементи вектора x }
  var m: real; {змінна поточного найбільшого значення}
      i: range; {індексна змінна для перебору елементів вектора}
begin m:=x[1]^; {початкове значення для m - число,
                на яке вказує перший елемент вектора}
  for i:=2 to n do {перебираємо решту елементів}
    if x[i]^>m then m:=x[i]^; {перевизначення m}
  max:=m

```

{після закінчення перебору m містить найбільше число}
end; {Max}

Видно, що цей алгоритм відрізняється від звичайного (відшукування найбільшого елемента масиву дійсних чисел) лише наявністю операції розіменування.



Перед виконанням Unique масив містить вказівники на різні динамічні змінні, деякі з них містять однакові числа.

Після виконання Unique вилучено динамічні змінні з однаковими числовими значеннями, відповідно змінено і вказівники.

Рис. 8. Перетворення масиву вказівників процедурою Unique.

```

procedure Unique(var x: vector);
  { Unique вивільняє зайву пам'ять, зайняту однаковими
  {   числовими значеннями
  var i, j: range; {індексні змінні для перебору елементів вектора}
  s: set of range; {множина індексів опрацьованих елементів}
  r: ref;      {робоча змінна-вказівник}
  a: real;    {робоча змінна базового типу}
begin s:=[];      {ні один елемент ще не опрацьовано}
  for i:=1 to n-1 do      {перебираємо елементи вектора}
  if not (i in s) then {i-й елемент вказує на нове число}
  begin s:=s+[i];      {опрацьовали i-й елемент}
    r:=x[i]; a:=r^; {задаємо значення робочих змінних}
    for j:=i+1 to n do      {шукаємо рівні до i-го}
    if x[j]^=a then
      begin Dispose(x[j]);      {вилучаємо знайдене}
        x[j]:=r; {x[j] вказує на перше з рівних чисел}
        s:=s+[j];      {j-й елемент вже опрацьовано}
      end {if & for j}
    end {if not & for i}
end; {Unique}

```

Зауважимо, що елемент $x[n]$ перевіряти непотрібно: якщо він вказує на число, яке дорівнює одному з попередніх, то значення $x[n]$ вже змінене в циклі "for_j", в іншому випадку $x[n]$ змінювати не треба.

Результати виконання процедури Unique показані на рис. 8 на прикладі перетворення масиву з шести елементів.

4.2. Обробка «довгого» тексту

Щоб реалізувати одне з можливих представлень "довгого" тексту, його розбивають на ділянки (рядки) однакової довжини і створюють масив вказівників на ці рядки.

```

const d=80;      {довжина рядка}

```

```

n=1000;           {максимальна кількість рядків}
type line=string[d];           {рядок}
ref=^line;           {вказівник на рядок}
range=1..n;           {діапазон}
LargeText=array[range] of ref; {текст}

```

Вважається, якщо в тексті менше за n рядків, то останні елементи дорівнюють **nil**; на початку масиву посилань **nil** не повинно бути.

Використовуючи це зображення тексту, описати:

- ? процедуру ReadText (f, T), яка зчитує з текстового файла f послідовність літер і формує з них текст T ;
- ? процедуру ReplaceLine (T, i, j), яка міняє місцями i -й і j -й рядки тексту T ;
- ? процедуру AddLine (T, i, j), яка додає після i -го рядка тексту T копію j -го;
- ? процедуру DeleteLine (T, i), яка вилучає i -й рядок з тексту T ;
- ? процедуру PrintText (T), яка друкує по рядках текст T .

Розв'язок.

```

procedure ReadText(var f: text; var T: LargeText);
  { ReadText створює текст T з літер, записаних }
  { у файлі f }
  var k,i:0..n; {індексні змінні для перебору елементів масиву T}
      j: 0..d;   {лічильник довжини рядка}
      c: char;   {для зачитування літер з файлу f}
      s: line;   {для формування рядка тексту}
begin Reset(f); k:=0;
  while not Eof(f) do
    begin j:=0; s:='';           {спочатку рядок порожній}
      while (j<d) and (not Eof(f)) do
        begin read(f,c); s:=s+c; inc(j) end; {while}
          {сформували черговий рядок тексту}
        inc(k); New(T[k]);      {утворили динамічну змінну}
        T[k]^:=s               {записали в неї значення}
      end; {While} close(f);
    for i:=k+1 to n do T[i]:=nil
      {"хвіст" тексту заповнили порожніми вказівниками}
end; {Read Text}
procedure ReplaceLine(var T: LargeText; i,j: range);
  { ReplaceLine міняє місцями i-й та j-й рядки текста T }
  var p: ref; {для тимчасового зберігання вказівника}
begin if (T[i]=nil) or (T[j]=nil) then
  WriteLn(' ReplaceLine: Вказано неіснуючий рядок.')
  else begin p:=T[i]; T[i]:=T[j]; T[j]:=p end
    {міняємо місцями вказівки на рядки}
end; {ReplaceLine}
procedure AddLine(var T: LargeText; i,j: range);
  { AddLine додає після i-го рядка текста T копію j-го }
  var k: range;      {індексна змінна}
begin if (T[i]=nil) or (T[j]=nil) then
  WriteLn(' AddLine: Вказано неіснуючий рядок.')
  else begin k:=n;
    while T[k]=nil do dec(k); {знайти останній зайнятий рядок}
    if k=n then WriteLn(' AddLine: Немає вільних рядків.')

```

```

    else begin while k>i do      {"посунути" всі рядки}
        begin T[k+1]:=T[k];      {до i+1 включно}
            dec(k) end; {while}
        New(T[i+1]); {виділити пам'ять для нового рядка}
        if j<=i then T[i+1]^:=T[j]^      {скопійовано}
            else T[i+1]^:=T[j+1]^      {j-й рядок}
        end{if} end{If}
end; {AddLine}

procedure DeleteLine(var T: LargeText; i: range);
    { DeleteLine вилучає i-й рядок з текста T }
    var k:range;          {індексна змінна}
begin if T[i]=nil then
    WriteLn(' DeleteLine: Вказано неіснуючий рядок.')
    else begin Dispose(T[i]);      {звільнити пам'ять}
        for k:=i to n-1 do
            T[k]:=T[k+1];      {"зімкнути" рядки тексту}
            T[n]:=nil end{if}
end; {DeleteLine}

procedure PrintText(var T:LargeText);
    { PrintText виводить по рядках текст T на екран }
    var k:range;          {індексна змінна}
begin if T[1]=nil then
    WriteLn(' PrintText: Текст порожній.')
    else begin k:=1;          {повторювати}
        while (T[k]<>nil)and(k<=n) do {до кінця тексту}
            begin Write(T[k]^);      {надрукувати k-й рядок}
                inc(k)          {збільшити індексну змінну}
            end; {while} WriteLn
        end{if}
end; {PrintText}

```

Наведені у частині 3 приклади демонструють ефективні методи опрацювання даних. У першому випадку використання масиву вказівників допомагає економити пам'ять. Ця економія тим суттєвіша, чим менший розмір вказівника від розміру значення базового типу. (У наведеному прикладі ці розміри відповідно 4 і 6 байтів). У другому випадку, завдяки використанню вказівників, прискорюється опрацювання рядків тексту. Адже зміна значення вказівника виконується швидше, ніж зміна значення цілого рядка, а зміна розташування рядка в тексті не потребує фізичного переміщення самого рядка – достатньо присвоїти вказівку на нього іншому елементу масиву вказівників.

5. Структура даних «список»

У цьому параграфі описано, як реалізувати за допомогою вказівників і динамічних змінних характерну для багатьох класів задач структуру даних – список.

5.1. Види списків

З математичного погляду *список* – це скінчена *послідовність пов'язаних* між собою *об'єктів* довільної природи і розміру. Ці об'єкти прийнято називати ланками. Впорядкованість ланок залежить від призначення списку, і нею може керувати програміст.

Поняття «список» природно узагальнює звичні для нашого повсякденного життя структури як, наприклад, список студентів академічної групи, список абонентів телефонної мережі тощо. За допомогою списків можна імітувати схему залізничних сполучень, роботу станції технічного обслуговування та інше.

Ланку списку умовно можна поділити на дві частини:

- 1) тіло (власне сам об'єкт);
- 2) довідкова інформація про розмір і тип конкретної ланки, взаємозв'язок з іншими ланками тощо.

У певному сенсі структуру даних "рядок" можна вважати частковим випадком списку, в якому всі ланки мають однакову жорстку структуру: тіло – це код одного символу, довідка – посилання на наступну ланку.

Зі списком можна виконувати такі *основні операції*:

- ◇ перехід до сусідньої ланки;
- ◇ включення у довільному місці нової ланки;
- ◇ вилучення зі списку довільної ланки;
- ◇ зміна порядку ланок у списку.

Зазначимо, що перелічені вище дії не вимагають фізичного переміщення ланок, а лише передбачають зміну довідкової частини окремих ланок.

Є багато різних *типів списків*. Найрозповсюдженішими з них є такі:

- лінійний однонаправлений список;
- двонаправлений список;
- кільцевий, як частковий випадок двонапрявленого;
- ієрархічний;
- асоціативний.

5.2. Реалізація дій з однонаправленим списком

Ми детальніше розглянемо реалізацію засобами Турбо Паскалю алгоритмів опрацювання однонаправлених списків, тобто таких, у яких чергова ланка має вказівку на місцезнаходження наступної ланки (рис. 9). Остання ланка не має наступника і тому містить порожній вказівник **nil**. Вказівка на першу ланку є у змінній – заголовку списку. Якщо список порожній (немає жодного елемента), то в заголовку – **nil**.

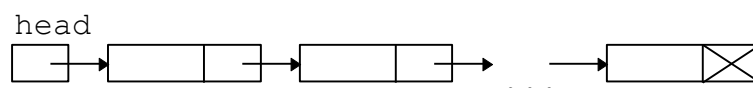


Рис. 9. Схематичне зображення лінійного однонаправленого списку.

Для реалізації однонапрявленого списку засобами мови Турбо Паскаль можна використати такі оголошення типів:

```
type ElementType=... {тип інформаційної частини ланки
    може бути довільним у залежності від потреби}
Chain=^Section; {тип "список"}
Section=record element: ElementType;
    link: Chain end; {record}
```

Роботу зі змінними типу Chain проілюструємо прикладом.

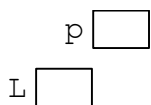
Нехай ElementType=real. Описати процедуру або функцію для виконання кожної з наступних дій:

- ? побудова списку з послідовності дійсних чисел, записаних у вхідному текстовому файлі;
- ? визначення того, чи є список L порожнім;
- ? вилучення з непорожнього списку всіх ланок, які містять задане дійсне значення;
- ? впорядкування даного списку методом простої вставки;
- ? об'єднання двох впорядкованих за зростанням непорожніх списків L1 та L2 в один впорядкований за зростанням список L1.

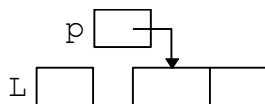
Зазначимо, що в багатьох випадках для опрацювання першої ланки списку потрібно виконувати не такі дії, як для решти ланок. Ця особливість пояснюється тим, що адреса першої ланки списку зберігається у його заголовку, а адреси всіх інших ланок – в полі link їхніх попередників. Іноді, щоб досягти одноманітності у способах опрацювання першої ланки та всіх інших, можна тимчасово вставити після заголовка списку (перед першою ланкою) «фіктивну» ланку. Після завершення опрацювання списку таку ланку вилучають. Наведені нижче тексти процедур містять відповідні коментарі.

Розв'язок.

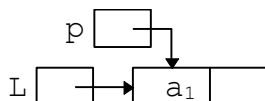
```
type ElementType=real;
Chain=^Section;
Section=record element: ElementType;
    link: Chain end; {record}
```



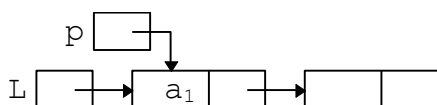
У початковий момент роботи процедури BuildChain є лише локальні змінні, розміщені в сегменті стека.



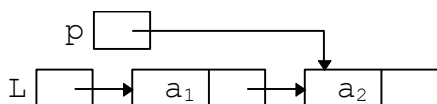
За допомогою оператора p:=new(Chain) створили першу ланку списку.



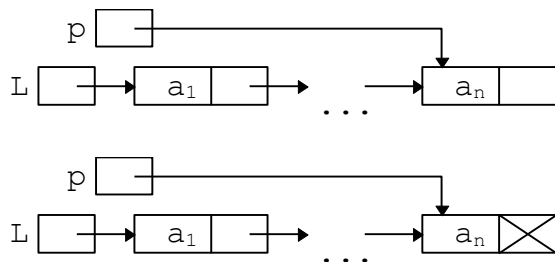
«Прив'язали» її до заголовка списку і занесли в неї числове значення.



Розпочали цикл для створення всіх наступних ланок (зображено результат виконання new(p^.link)).



Робочий вказівник p посунули на новостворену ланку, занесли в неї числове значення.



Цикл завершує роботу, коли прочитано і занесено у список останнє число з файлу.

Занесенням у поле зв'язку останньої ланки значення **nil** завершили формування списку.

Рис. 10. Процес формування лінійного однонаправленого списку.

```

procedure BuildChain (var f: text; var L: Chain);
  { BuildChain будує список L з чисел, записаних у }
  {           непорожньому файлі f           }
  var p: Chain; {робоча змінна для створення нових ланок}
begin p:=new(Chain); L:=p;           {створили ПЕРШУ ланку}
  reset(f);read(f,p^.element);      {занесли в неї число}
  while not eof(f) do {заносимо в список РЕШТУ чисел}
  begin new(p^.link);                {наступна ланка}
    p:=p^.link; read(f,p^.element)
  end;{while} p^.link:=nil          {остання ланка}
end; {Build Chain}

```

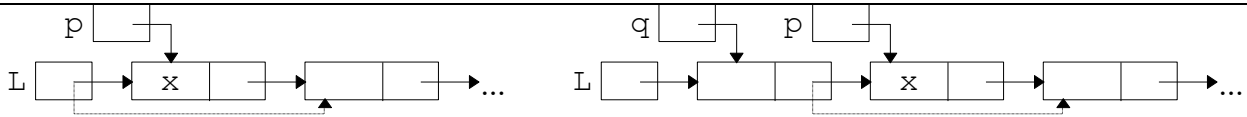
Процес формування списку процедурою BuildChain можна продемонструвати за допомогою схеми, зображеної на рис. 10.

```

function IsEmpty (var L: Chain): Boolean;
  { IsEmpty перевіряє, чи список L є порожній }
begin IsEmpty:=L=nil
  {заголовок порожнього списку містить порожню вказівку}
end; {IsEmpty}

procedure Exclude (var L: Chain; x: real);
  { Exclude вилучає всі ланки зі списку L, }
  {           які містять значення x           }
  var q,p: Chain; {змінні для перебору і вилучення ланок}
begin { *** ОПРАЦЮВАННЯ ПЕРШОЇ ЛАНКИ *** }
  {перевіримо чи L не починається з x}
  while (L<>nil)and(L^.element=x) do
  begin p:=L;                                {фіксуємо знайдену ланку}
    L:=L^.link;                               {список вже починається з наступної}
    Dispose(p)                                 {ліквідували фіксовану ланку}
  end;{while}
  If L=nil then {список містив лише x} Exit {все}
  { *** ОПРАЦЮВАННЯ РЕШТИ СПИСКУ *** }
  else begin q:=L; p:=L^.link; {розглядаємо перші дві}
  while p<>nil do
  if p^.element=x then {знайшли ланку}
  begin q^.link:=p^.link; {попередню зв'язали з наступною}
    Dispose(p);          {ланку з елементом x ліквідували}
    p:=q^.link           {далі розглянемо наступну}
  end {then}
  else begin q:=p; p:=p^.link {рухаємось за списком далі}
end {if} end {If}
end; {Exclude}

```



Перед вилученням першої ланки достатньо змінити вказівник у заголовку списку.

Для вилучення наступної ланки потрібно два допоміжні вказівники (другий потрібен для доступу до попередньої ланки).

Рис. 11. Вилучення ланок з лінійного однонапрявленого списку.

```

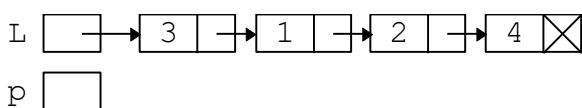
procedure SortChain(var L: Chain);
  { SortChain впорядковує список L за зростанням }
  { методом простої вставки }
  var p,q,r: Chain;           {робочі змінні}
begin New(p);                {тимчасово створюємо "фіктивну" ланку,}
  {щоб забезпечити одноманітність опрацювання всього списку}
  p^.link:=L; L:=L^.link; p^.link^.link:=nil; {приготування}
  {p - впорядкований список, вставляємо в нього ланки списку L}
  While L<>nil do
  begin q:=p;                 {шукаємо місце у впорядкованому списку p}
    while (q^.link<>nil) and (q^.link^.element<L^.element)
      do q:=q^.link;          {для чергової ланки з L}
    if q^.link<>nil then {вставка ланки всередину списку p}
      begin r:=L; L:=L^.link;
        r^.link:=q^.link; q^.link:=r end
    else                       {у кінець списку p}
      begin q^.link:=L; L:=L^.link; q^.link^.link:=nil end
    end; {While}
  L:=p^.link; Dispose(p)      {вилучили "фіктивну" ланку }
end; {SortChain}

```

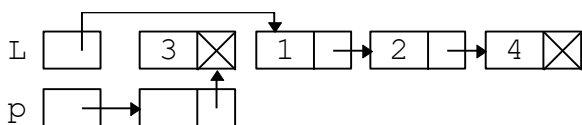
Алгоритм сортування значно складніший від алгоритму простого створення списку. Проілюструємо його прикладом впорядкування послідовності чисел {3, 1, 2, 4}. Продемонструємо на ньому три характерні ситуації, які виникають у ході його виконання:

- вставка ланки на початок відсортованого списку;
- вставка всередину списку;
- вставка останньої ланки.

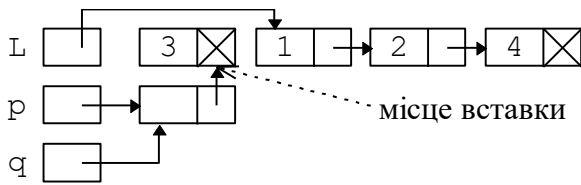
Зауважимо, що перші дві з цих ситуацій алгоритм опрацьовує однаково завдяки тимчасовому включенню у список фіктивної початкової ланки (рис. 12).



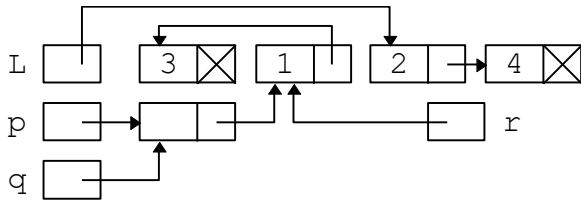
Спочатку L вказує на невпорядкований список; для простоти на рисунку показано тільки одну з робочих змінних – вказівник p.



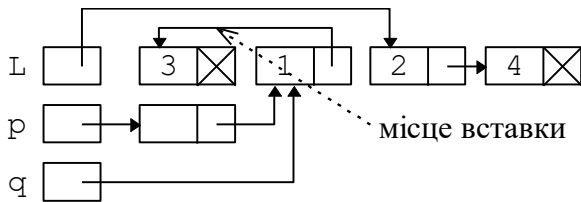
Завершено підготовку до сортування: створено тимчасову ланку, першу ланку зі списку L перенесено у список p.



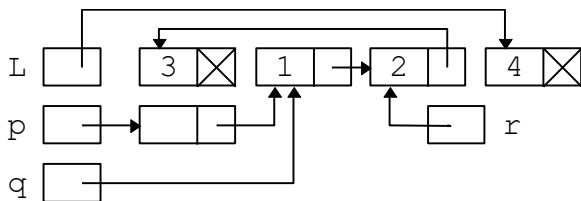
Вказівник q використовується для перебору ланок впорядкованого списку p ; знайдено місце на початку списку, куди треба вставити ланку з одиницею.



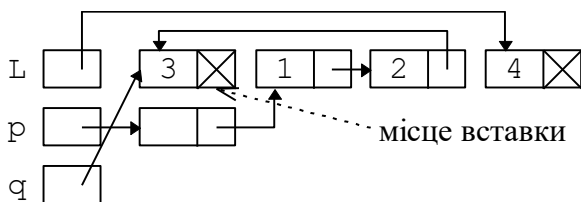
Для виконання вставки використано робочу змінну-вказівник r , вказівник L переміщено на наступну ланку неупорядкованого списку.



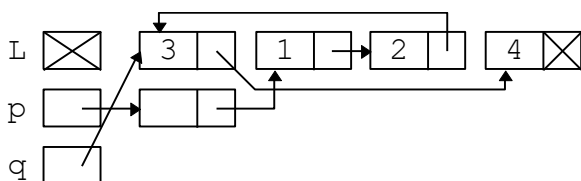
Як і в попередньому випадку, вказівник q використано для перебору ланок впорядкованого списку p ; знайдено місце всередині списку, куди треба вставити ланку з двійкою.



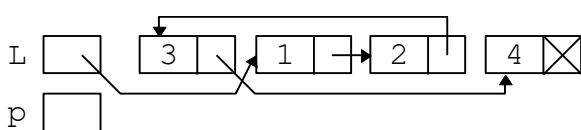
Вставка ланки у впорядкований список p відбулась так само, як у попередньому випадку.



Після перебору всіх ланок списку p , виявлено, що чергову ланку з L потрібно вставляти в кінець списку.



Алгоритм такої вставки простіший: її можна виконати без використання додаткових змінних; вказівник L отримав значення *nil*, отже, впорядкування завершено.



Виконання оператора $L := p^{\wedge}.link$ «прив'яже» відсортований список до L ; після того додаткову ланку можна знищити.

Рис. 12. Приклад впорядкування лінійного списку.

```

procedure Unite (var L1, L2: Chain);
  { Unite об'єднує впорядковані списки L1 і L2      }
  {          в список L1, непорушуючи впорядкування }
  var p, q, r: Chain;           {робочі змінні}
begin {перевіримо чи потрібно вставити ланку L2 перед
                                             першою ланкою L1}
  if L1^.element > L2^.element then
    begin p := L2; L2 := L2^.link; {вилучили ланку з L2}
      p^.link := L1; L1 := p {вставили її на початок L1}
    end else p := L1;           {почнемо з початку L1}
  r := p^.link;                 {друга ланка списку L1}
  while (L2 <> nil)            {переберемо всі ланки L2}

```

```

    and(r<>nil) {вставлятимемо їх всередину L1}
do if L2^.element<=r^.element
    then begin q:=L2;L2:=L2^.link; {вилучили з L2}
        q^.link:=r;p^.link:=q;p:=q {вставили в L1}
        end
    else begin p:=r;r:=r^.link{просунулись по L1}
        end; {while}
    if r=nil then begin p^.link:=L2; L2:=nil end
end; {Unite}

```

5.3. Зображення розріджених поліномів за допомогою списків

Поліном $P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ з дійсними коефіцієнтами a_i ($i = 0, \dots, n$) можна зобразити у вигляді списку, якщо $a_i = 0$, то відповідна ланка у список не включається. Наприклад, на рис. 13 схематично зображено поліном $S_{40}(x) = 52x^{40} - 3,5x^8 + 1$.

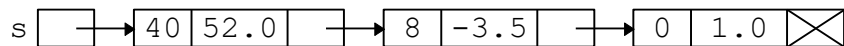


Рис. 13. Список – поліном.

Завдання: описати тип `ElementType` і процедури та функції для обробки списків-поліномів:

- ? логічну функцію `Equal(p, q)`, яка перевіряє на рівність поліноми p і q ;
- ? процедуру `Derivate(p, q)`, яка будує p – похідну полінома q ;
- ? процедуру `Print(p, v)`, яка друкує поліном p від змінної, ім'я якої передається їй параметром v . Наприклад, `Print(S, 'y')` має надрукувати:

$$52 y^{40} - 3.5 y^8 + 1$$

Розв'язок.

```

type ElementType=record power:byte;           {показник степеня}
                        coef:real end; {числовий коефіцієнт}
Chain=^Section;
Section=record element: ElementType;
                link: Chain end;

```

Інформаційна частина кожної ланки списку містить значення комбінованого типу, яке зображає показник степеня і ненульовий коефіцієнт; оголошення самого списку таке саме, як і раніше.

```

function Equal(p, q: Chain): Boolean;
{ Equal порівнює поліноми p і q }
var b: Boolean;           {робоча змінна}
begin b:=true;           {припускаємо, що p=q}
    while b and(p<>nil) and(q<>nil) do
        {поліноми рівні, якщо рівні всі ланки}
    begin with p^.element do
        b:=(power=q^.element.power) and(coef=q^.element.coef);
        p:=p^.link; q:=q^.link; {розглянемо наступні ланки}
    end; {while}
equal:=b and (p=q)           {чи списки однієї довжини?}
    { (поля link останніх ланок повинні містити nil) }

```

```

end; {Equal}

procedure Derivate(var p: Chain; q: Chain);
  { Derivate обчислює поліном p - похідну }
  {      непорожнього полінома q      }
var r: Chain;      {робоча змінна}
begin with q^.element do
  If power=0 then      {поліном - константа}
    begin p:=nil; Exit end      {похідна від неї рівнює 0}
  else begin New(p);      {створили першу ланку}
    p^.element.power:=power-1;      {понижили степінь}
    p^.element.coef:=power*coef; {i збільшили коефіцієнт}
    r:=p end; {If & with}
  while q^.link<>nil do      {рахуємо, поки є продовження}
    begin q:=q^.link;      {наступна ланка}
      with r^,q^.element do
        if power>0 then      {не константа}
          begin New(link); r:=link; {створили нову ланку}
            r^.element.power:=power-1;
            r^.element.coef:=power*coef {i обчислили ii}
          end{if}
        end; {while} r^.link:=nil      {кінець похідної}
    end; {Derivate}

procedure Print(p: Chain; v: char);
  { Print друкує непорожній поліном p(v) }
  {   коефіцієнти - з точністю до сотих   }
var sign: Boolean;      {чи треба друкувати "+"}
begin sign:=false;      {перед поліномом - ні}
  while p<>nil do      {переглянемо весь список-поліном}
    with p^.element do      {для зручності звертання}
      begin if sign and (coef>0) then write('+');
        sign:=true; {всередині p-ма "+" друкувати треба}
        write(trunc(coef), '.', round(frac(abs(coef))*100));
          {дійсне число друкуємо у зручному вигляді}
        if power>0 then
          begin write(v);      {друкуємо ім'я змінної}
            if power>1 then write('^',power)
          end; {if}      {i показник степеня}
        p:=p^.link end; {while & with}
      writeln      {рядок друку закінчився}
    end; {Print}

```

Зауваження. Оголошення параметрів функції Equal, параметра q у Derivate чи параметра p у Print як параметрів-змінних було б помилковим, бо виконання таких підпрограм призвело б до повної втрати переданих їм фактичних параметрів-списків. Поміркуйте, чому.

6. Нетипізовані вказівники

Крім описаних раніше вказівникових типів, у мові Турбо Паскаль передбачено особливий стандартний вказівний тип, для якого базовий тип не

конкретизується. Він позначається ідентифікатором `pointer` і вважається сумісним з усіма вказівними типами. Наприклад, описи

```
var P1: ^integer; P2: ^real; P: pointer;
```

допускають присвоювання вигляду `P:=P1` або `P:=P2` (але не `P1:=P` і не `P2:=P`). Нагадаємо також, що оператор `P1:=P2` є недопустимим.

Значення типу `pointer` називають нетипізованими вказівниками.

Змінна типу `pointer`, як і будь-який інший вказівник, може містити довільну адресу пам'яті, але тепер уже програмістові доводиться вирішувати, як інтерпретувати вміст пам'яті за вказаною адресою. Тобто програміст повинен знати, на дані якого типу вказує ця змінна. Робота з нетипізованим вказівником, звичайно, передбачає перетворення його у вказівник на дані конкретного базового типу (використовуючи конструкцію приведення типу). Наприклад, під час виконання фрагмента програми

```
type PInteger = ^integer;
      PReal = ^real;
var p: pointer;
begin    ... WriteLn(PInteger(p)^) {A};
          ... WriteLn(PReal(p)^)   {B}; ...
```

оператор `{A}` надрукує як ціле число два байти пам'яті, починаючи з занесеної в `p` адреси. Оператор `{B}` надрукує 6 байтів пам'яті, починаючи з цієї ж адреси. Вони будуть проінтерпретовані як значення типу `real`.

Для виділення та звільнення пам'яті під час роботи з нетипізованими вказівниками в Турбо Паскалі використовують відповідно процедури `GetMem` і `FreeMem`. Їх виклик у загальному випадку має вигляд:

```
GetMem(<нетипізований_вказівник>, <кількість_байтів>),
FreeMem(<нетипізований_вказівник>, <кількість_байтів>),
```

де перший параметр – змінна типу `pointer`, а другий – константа цілого типу, що задає розмір пам'яті в байтах, яка виділяється чи звільняється. Максимальне значення цього параметра – 65521 (байт). Наприклад, виклик процедури `GetMem(p, 60)` виділяє область розміром 60 байтів і записує в `p` адресу її початку. Щоб звільнити цю область, треба зробити виклик `FreeMem(p, 60)`. Звільняти потрібно ту ж кількість байтів, яка була виділена, бо інакше поведінка програми може стати непередбачуваною. Найлегше дотриматись цієї вимоги, використовуючи для вказання розміру функцію `SizeOf`, про яку вже йшлося раніше.

У [5] можна знайти приклад використання нетипізованого вказівника для побудови рухомих графічних зображень:

```
uses Graph;
var p: pointer; l: longint;
    i: byte; x1, y1, x2, y2: integer;
begin    .....
        l:=ImageSize(x1, y1, x2, y2);
        {визначаємо розмір потрібної ділянки динамічної пам'яті}
        GetMem(p^, l);
```

```

    {резервуємо буфер для зберігання графічного образу}
    GetImage(x1,y1,x2,y2,p^);           {запам'ятовуємо образ}
    ClearDevice; {очистили екран, приготувались до мультика}
    ReadLn;
    for i:=0 to 20 do {образ виводимо щоразу в нове місце}
        PutImage(x1+i*5,y1,p^,normalPut);
    ReadLn; FreeMem(p,l); ... {звільнили пам'ять}

```

Проілюструємо використання змінних типу `pointer` ще двома прикладами.

6.1. Використання масивів змінної довжини

Багато задач починаються зі слів «задано натуральне n і послідовність дійсних чисел a_1, a_2, \dots, a_n ». Далі потрібно виконати певні обчислення, перетворення, використовуючи a_i . З такої умови випливає, що в програмі треба оголосити масив для зберігання значень цієї послідовності. Але як оголосити масив довжиною n , якщо це число стає відоме лише на етапі виконання програми? Зауважимо, що такі ситуації виникають і під час розв'язування реальних обчислювальних задач.

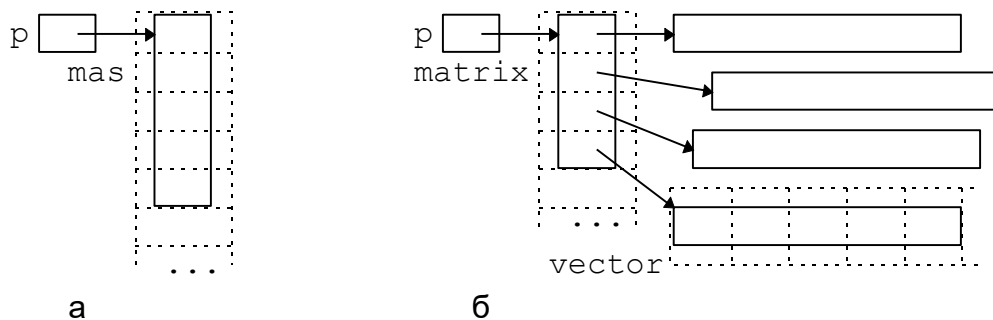


Рис. 14. Масиви змінних розмірів.

Щоб розв'язати поставлене завдання, програміст часто виділяє пам'ять для масиву з запасом. Наприклад, якщо відомо, що $n \leq 100$, можна використати оголошення:

```

type mas=array[1..100]of array;
var a: mas;

```

Проте витрати пам'яті будуть недопустимо великими, якщо реальні значення n у більшості випадків використання програми набагато менші.

У цій ситуації не допоможе і така змінна вказівного типу:

```

var d:^mas;

```

Адже виконання `New (d)` призведе до створення 100-елементного масиву.

Щоб створити масив точно з n елементів, використаємо нетипізований вказівник і процедуру виділення пам'яті `GetMem`. Щоб інтерпретувати виділену ділянку пам'яті як масив, використаємо явне перетворення типу змінної.

```

type mas=array[1..100]of array;
var p:pointer; i:integer;
begin write('Введіть n: '); readln(n);
      GetMem(p,n*SizeOf(real)); {виділили потрібну кількість байт}
      writeln('Введіть послідовність з ',n,' чисел');
      for i:=1 to n do {введемо n чисел}
          read(mas(p^)[i]); {інтерпретуємо p^ як масив}

```

.....

Схематично масив p^{\wedge} зображено на рис. 14, а. «Накладання» типу на рисунку позначено пунктирними лініями.

Складніше виглядає створення та використання матриці змінних розмірів:

```

type vector=array[1..maxint] of real;
      matrix=array[1..maxint] of pointer;
var m:pointer; i,j:integer;
begin write('Введіть n,m: '); readln(n,m);
      GetMem(p,n*SizeOf(real));           {p^ будемо інтерпретувати}
                                          {як масив вказівників на рядки матриці}
for i:=1 to n do GetMem(matrix(p^)[i],m*SizeOf(real));
      {попередньо виділили пам'ять для кожного рядка матриці}
      .....

```

Звертання до елемента такої матриці матиме вигляд
 vector(matrix(p^)[i]^)[j]

Зауважимо, що процедури New і GetMem дають змогу виділити не більше, ніж 64 Кбайт. Описана вище матриця у реальній ситуації може мати набагато більший розмір (якщо тільки це потрібно для задачі). Схематично вона зображена на рис. 14, б.

Описаний спосіб створення масивів є дуже гнучким. Але його використання ставить підвищені вимоги до відповідальності програміста за коректне використання індексних змінних. Компілятор не зможе виявити вихід індекса за верхню межу, наприклад, масиву mas(p^).

6.2. Двонаправлений лінійний список

У цьому пункті описано приклад створення двонапрявленого списку. Тобто такого, в якому кожна ланка містить вказівку не лише на наступну ланку (як в однонапрявленому списку), а й на попередню.

Наведено дві процедури: перша з них зчитує з диска текстовий файл і завантажує його в динамічну пам'ять як двонаправлений список. Тіло однієї ланки списку повинно містити один абзац вихідного тексту. Абзац може складатися з довільної кількості літер, але не більшої ніж 2000. Ознакою кінця абзацу у файлі є символ "#". Оскільки розміри різних абзаців можуть бути різними, пам'ять для них виділятимемо за допомогою GetMem.

Друга процедура виводить на екран вміст n -ї ланки списку (n -й абзац завантаженого тексту).

```

type string12=string[12];
      tbuf= array[1..2000] of char;
      TwoDirectChain=^part;           {двонаправлений список}
      part=record previous, next: TwoDirectChain;
      {зв'язок з попередньою і наступною ланками}
      size_of_paragraph: word;         {розмір абзаца}
      paragraph:pointer{вказівник на абзац}
      end; {TwoDirectChain}

procedure LoadText(var T: TwoDirectChain;
                   var loaded: Boolean; filename: string12);

```

```

{ LoadText формує двонаправлений список T з абзаців }
{
    тексту, записаного у файлі filename
}
var p: TwoDirectChain; {робоча змінна для формування ланок}
    f: text;           {змінна для приєднання файлу}
    size: word; c: char; buffer: tbuf;
    {робочі змінні і буфер для накопичення абзацу}
begin Assign(f,filename); {$I-}    {контроль берем на себе}
    Reset(f); {$I+}                {спроба приєднати файл}
    If Ioresult<>0 then            {дія не відбулась}
begin loaded:=false; t:=nil;
    WriteLn('LoadFile: Немає файлу ',filename)
end else                            {завантажуємо текст}
begin loaded:=true;
    New(p); t:=p;                  {тимчасово створимо зайву ланку}
    While not Eof(f) and loaded do
{працюватимемо до закінчення файлу чи вичерпання купи}
begin size:=0; Read(f,c);          {читаємо абзац}
    while c<>'#' do                {до його кінця}
begin inc(size); buffer[size]:=c;  {в буфер}
    read(f,c) end; {while}
    loaded:=MemAvail>=size;        {чи вистачить купи?}
    if not loaded then
        writeln('LoadFile: Бракує динамічної пам'яті.')
    else                            {пам'яті вистачило}
begin New(p^.next);                {створили нову ланку}
    p^.next^.previous:=p;
    {прив'язали її до попередньої}
    p:=p^.next; with p^ do        {працюємо з новою}
begin size_of_paragraph:=size;
    GetMem(paragraph,size);
    {резервуємо пам'ять для абзацу}
    Move(buffer,paragraph^,size)
    {завантажуємо абзац у пам'ять}
end {with}
end {if}                            {завантаження абзацу завершено}
end; {While} Close(f); {текст завантажили закрили файл}
    p^.next:=nil;                  {кінець списку}
    p:=T; T:=T^.next; Dispose(P); {вилучили зайву ланку}
    if t<>nil then t^.previous:=nil
    {початок непорожнього списку}
end {If}
end; {LoagText}

procedure ShowCadr(T: TwoDirectChain; n: integer);
{ ShowCadr виводить на екран вмістиме n-ї ланки списку T }
var i: integer;
begin for i:=2 to n do T:=T^.next; {знаходимо n-ну ланку}
    with T^ do                    {друкуємо її вміст}
    for i:=1 to size_of_paragraph do write(tbuf(paragraph^)[i]);
    writeln
end; {ShowCadr}

```

На завершення цього прикладу зобразимо схематично структуру даних, яку створюватиме процедура LoadText.

Змінна `T` містить вказівку на першу ланку двонапрявленого списку. Кожна ланка складається з чотирьох полів:

- `previous` містить вказівку на попередню ланку (це поле першої ланки списку має значення `nil`);
- `next` містить вказівку на наступну ланку (в останній ланці це поле містить `nil`);
- `size_of_paragraph` містить розмір у байтах відповідного абзацу вихідного тексту (адже абзаци можуть бути різного розміру);
- `paragraph` містить вказівку на ділянку динамічної області пам'яті, де розташований цей абзац. Використання типу `pointer` для опису даного поля дає змогу відводити для різних абзацив ділянки пам'яті різного розміру – саме того, який потрібен для завантаження кожного конкретного абзацу.

На рис. 15 схематично зображено описану вище структуру.

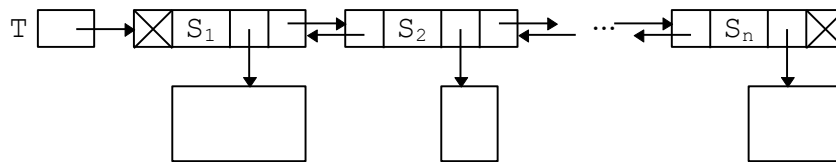


Рис.15. Двонапрявлений список.

7. Структури даних «черга» та «стек»

Одними з найчастіше вживаних динамічних структур даних є черга і стек. Це окремі випадки лінійного однонапрявленого списку. Спочатку дамо їхнє формальне означення та спосіб реалізації засобами мови програмування Паскаль, а потім наведемо приклади використання.

7.1. Реалізація черги

Черга – це одна з найпростіших динамічних структур даних, яка використовується для тимчасового зберігання даних і працює за правилом «першим прийшов – першим пішов». Реалізувати чергу можна за допомогою однонапрявленого списку, над яким дозволено виконувати операції лише двох видів:

- 1) додавання нової ланки лише на початку списку;
- 2) вилучення лише кінцевої ланки списку.

Для доступу до останньої ланки черги зручно використати додатковий вказівник як показано на рис. 16. Зауважимо, що стрілки на рисунку зображують зв'язки між ланками списку, а рух ланок у черзі відбувається у протилежному напрямі.

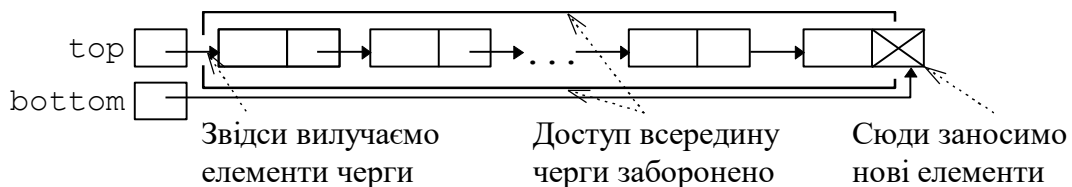


Рис. 16. Структура даних «черга».

Нижче наведено необхідні оголошення типів та процедури занесення елемента в чергу та вилучення його з черги

```

type ElementType=... {тип інформаційної частини ланки
                        може бути довільним залежно від потреби}
Chain=^Section; {тип "список"}
Section=record element:ElementType; link:Chain end;

procedure InQueue(var top,bottom: Chain; x: ElementType);
  { InQueue заносить елемент x в top - початок черги }
begin If MaxAvail<SizeOf(x) then
  writeln('InQueue: Вичерпана динамічна пам'ять')
  else begin {вільної пам'яті є досить}
  if bottom=nil then {створюємо першу ланку черги}
  begin New(bottom); top:=bottom;
  end else {продовжуємо чергу}
  begin New(bottom^.link); bottom:=bottom^.link
  end; {if} {задаємо поля нової ланки}
  with bottom^.do
  begin element:=x; bottom^.link:=nil end{with}
  end{If}
end; {InQueue}

```

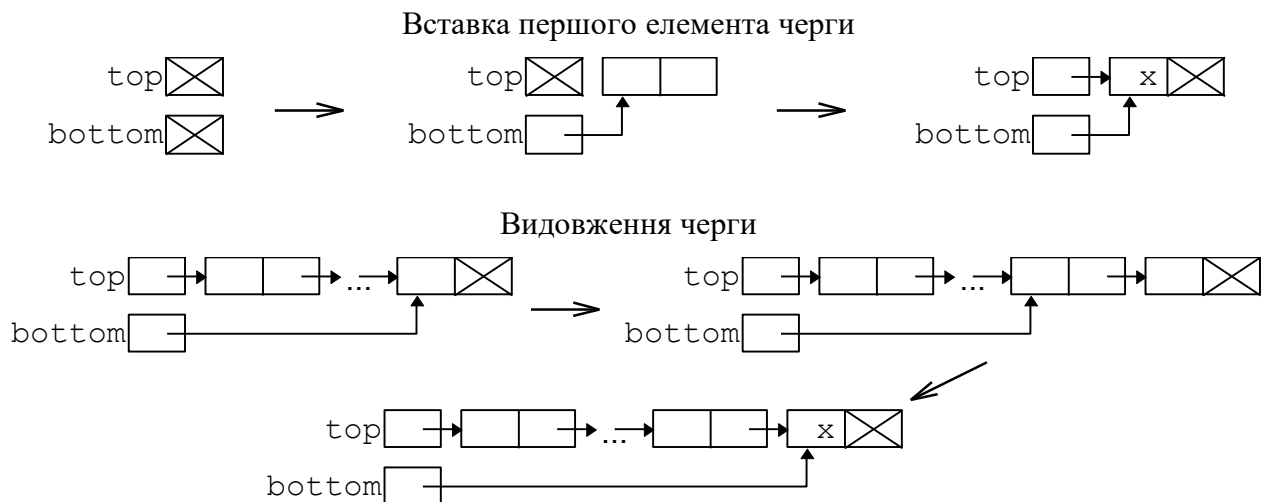


Рис. 17. Занесення елемента в чергу.

```

procedure OutQueue(var top,bottom: Chain; var x: ElementType);
  { OutQueue вилучає значення з bottom - кінця черги }
  { i записує його в x }
  var p: Chain; {робоча змінна}
begin If top=nil then writeln('OutQueue: Черга порожня')
  else begin if top=bottom then {черга стане порожня}
  begin p:=top; top:=nil; bottom:=nil end
  else {вкорочуємо чергу}
  begin p:=top; top:=top^.link end; {if}
  x:=p^.element; Dispose(p)
  end{If}
end; {OutQueue}

```

Вилучення єдиного елемента з черги

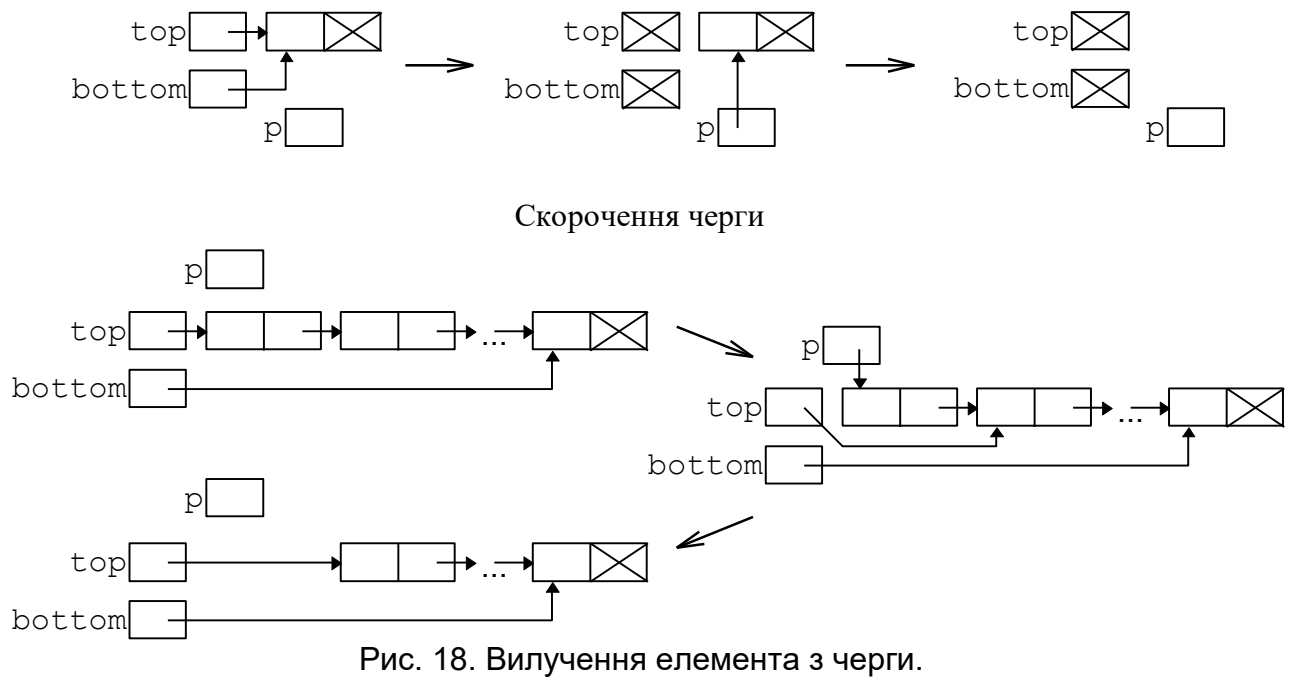


Рис. 18. Вилучення елемента з черги.

Виконання цих процедур схематично зображено на рис. 17, 18.

7.2. Імітаційні алгоритми

У практиці трапляється низка задач, розв'язання яких вимагає опису великих систем. Наприклад, обслуговування комп'ютерних мереж, розрахунків космічних польотів, прийняття адміністративних рішень, керування виробництвом тощо. Такі системи вивчають за допомогою *машинного моделювання*, тобто у процесі експериментування на ЕОМ над моделлю реальної динамічної системи. Безпосередня мета експериментів – спостереження за поведінкою системи у заданих умовах і значеннях параметрів. Кінцевою метою експериментів може бути формулювання стратегії керування, визначення оптимальної конфігурації системи тощо.

Можна назвати такі переваги моделювання:

- є змога вивчати всі частини системи, об'єднані в одне ціле;
- будь-які параметри можна змінювати;
- інформацію про систему можна отримати, не затрачаючи засобів на побудову реального прототипу;
- моделювання можна проводити в різних масштабах часу (прискореному чи сповільненому).

Одними з найпростіших і найпоширеніших імітаційних алгоритмів є алгоритми, які моделюють роботу лінії обслуговування. Найпростіша лінія обслуговування без відмов складається з черги клієнтів та одного пристрою обслуговування. Її можна описати так.

1. Момент прибуття клієнта – випадкова величина: $t_{k+1} = t_k + T$, де t_k – момент прибуття k -го клієнта, T – випадкова змінна, $1 \leq T \leq T_{\max}$.
2. Тривалість обслуговування k -го клієнта – випадкова змінна s_k , $1 \leq s_k \leq s_{\max}$.

3. Черга працює за принципом «першим прийшов – першим обслужили». Клієнт, що прийшов, одразу потрапляє в чергу і залишається в ній, поки його не обслужать. Після обслуговування клієнт одразу покидає систему.
4. Конфігурація системи змінюється в результаті настання подій. Подія є незалежною (первинною), якщо вона не спричинена іншими подіями. Первинними подіями є прибуття клієнта, завершення обслуговування. Залежною подією є перехід на обслуговування наступного клієнта.
5. У початковий момент система порожня.
6. Поток подій керує годинник з дискретним приростом часу.
7. Для моделювання випадкових величин використовується давач випадкових чисел.

У результаті експериментів бажано одержати таку інформацію:

- a) число надходжень клієнтів за весь час імітації;
- b) середня довжина черги;
- c) середній час очікування в черзі;
- d) максимальна довжина черги;
- e) ефективність використання пристрою обслуговування;
- f) середній, найменший і найбільший час обслуговування;
- g) функція розподілу довжини черги.

Для реалізації в програмі черги клієнтів лінії обслуговування зручно використати динамічну структуру даних «черга», описану у попередньому параграфі.

7.3. Моделювання руху на регульованому перехресті

Нижче наведено приклад імітаційного алгоритму. Він моделює рух автомобілів однією з вулиць перехрестя, регульованого світлофором. Програма розроблена студентами механіко-математичного факультету ЛНУ ім. І. Франка під час проходження обчислювальної практики.

З а в д а н н я . Створити модуль, що містить засоби для роботи зі структурою даних черга: оголошення типів, процедур (функцій) для перевірки пустоти черги, можливості збільшення черги, для додавання елемента в чергу, вилучення елемента з черги.

За допомогою ресурсів такого модуля, а також стандартних модулів CRT і DOS змодельовати в режимі реального часу проїзд автомобілів вулицею з одностороннім рухом і одним світлофором. Вважати, що k -й автомобіль під'їжджає до світлофора через p_k секунд після попереднього автомобіля і рушає з затримкою q_k секунд після зупинки на червоне світло. Світлофор перемикає червоне і зелене світло за однакові проміжки часу. Числа p_k і q_k ($k=1, 2, \dots$) отримати за допомогою давача випадкових чисел.

Можна вважати, що у цій задачі моделюється робота лінії обслуговування. Пристроєм обслуговування є світлофор, а клієнтами – автомобілі. Коли світиться зелений сигнал, обслуговування відбувається миттєво, усі автомобілі проїжджають перехрестя. У іншому випадку (світиться червоний сигнал) обслуговування не

відбувається, усі клієнти стають у чергу, доки знову не засвітиться зелений сигнал. Попадання автомобіля в чергу відбувається також і тоді, коли рух через перехрестя вже почався, але ще не всі автомобілі з черги рушили.

Текст модуля містить створені студентами засоби, які реалізують структуру даних «черга» і операції з нею. Зауважимо, що вони (засоби) дещо відрізняються від описаних раніше.

```

unit Queue;                                { модуль містить ресурси, }
                                           { що реалізують структуру даних ЧЕРГА }

  interface
type   typeEl=integer;                      {тип елемента}
        tQueue=^queue;                       {черга}
        queue=record el:typeEl; link:tqueue end;
procedure newQueue(var d:tQueue);          {створює нову чергу}
function isEmpty(d:tQueue):Boolean;
                                           {перевіряє, чи черга порожня}
procedure intoQueue(var d:tQueue;a:typeEl);
                                           {заносить елемент у чергу}
procedure outQueue(var d:tQueue;var a:typeEl);
                                           {вилучає елемент з черги}
procedure showQueue(var d:tQueue);
                                           {роздруковує вміст черги,}
                                           {використовується для налагодження}
procedure clearQueue(var d:tQueue);        {ліквідує чергу}

  implementation
procedure newQueue(var d:tQueue);
  begin d:=nil end;
function isEmpty(d:tQueue):Boolean;
  begin isEmpty:=(d=nil); end;
procedure intoQueue(var d:tQueue;a:typeEl);
  var t:tQueue;
begin if isEmpty(d) then
  begin new(d);d^.el:=a;d^.link:=nil;
  end; new(t);t^.el:=a;t^.link:=d;d:=t;
end;
procedure outQueue(var d:tQueue;var a:typeEl);
var t,q:tQueue;
begin if not isEmpty(d) then
  if not isEmpty(d^.link) then
  begin t:=d;
  while t^.link^.link<>nil do t:=t^.link;
  a:=t^.link^.el;q:=t^.link;
  t^.link:=nil;dispose(q);
  end else
  begin a:=d^.el;t:=d;d:=nil;dispose(t); end
  else begin
  writeln('Error:Queue is empty! Impossible reading!');
  halt(1);end;
end;
procedure showQueue(var d:tQueue);
var t:tQueue;
begin t:=d;

```

```

if t=nil then writeln('Queue is empty!')
else if t^.link=nil then writeln('Queue is empty!')
  else
    begin write('The begining of the queue ');
      repeat write('=> ',t^.el); t:=t^.link;
      until t^.link=nil;
    end;
end;
procedure clearQueue(var d:tQueue);
var t:tQueue;
begin t:=d; if t=nil then writeln('Queue is cleared!')
  else while t^.link <>nil do
    begin d:=t^.link; dispose(t); t:=d;
    end;
end;
end{unit Queue}.

```

Текст програми

```

program pr2;
uses dos,crt,Queue;
const c=#220+#219+#219+#220; {автомобіль}
      s=' '+#219+#219; {сигнал світлофора}
var x:tQueue; a,b:typeEl;
      i,j,j1:integer;
      p:Boolean;
procedure move_c(var c1:integer);
{ пересуває вказаний автомобіль на одну позицію вперед }
begin c1:=c1+1;
  if c1>=1 then begin textcolor(0);
    gotoXY(c1-1,20);writeln(c);
    textcolor(5);
    gotoXY(c1,20);writeln(c); end;
end;
procedure take_of(c1:integer);
{ стирає автомобіль, який досяг краю екрана }
begin c1:=c1+1; textcolor(0);
  gotoXY(c1-1,20);writeln(c);
end;
procedure move_all(f1:Boolean;var y:tQueue);
var i1,i2:integer;
      Hour, Minute, Second, Sec100:word;
      f2,f3:Boolean;
begin f3:=true; outQueue(y,i1);
  repeat GetTime( Hour, Minute, Second, Sec100);
    f2:=((second div 7)mod 2 =0);
    if f2 then {зелений сигнал}
      begin textcolor(yellow); gotoXY(55,18); write(s);
        textcolor(green); gotoXY(55,19); write(s);
      end else {червоний сигнал}
      begin textcolor(yellow); gotoXY(55,19); write(s);
        textcolor(red);gotoXY(55,18); write(s);
      end;
    if f1 then begin
      if i1>75 then
        begin take_of(i1); j1:=j1-random(75)-9;

```

```

        intoQueue(y,j1)
    end else
    begin move_c(i1); intoQueue(y,i1);
    end;
end else intoQueue(y,i1); i2:=i1;
outQueue(y,i1); f3:=f1;
f1:=(abs(i2-i1)>5) and not ((not f2 and (i1=50)));
if f3 and (abs(i2-i1)<9) then f1:=false;
delay(2);
until keyPressed;
end;
Begin {main program} newQueue(x); randomize;
j1:=0; for i:=1 to 60 do
begin clrscr; j:=random(75); j1:=j1-i-9;
intoQueue(x,j1);
end; move_all(true,x);
End.

```

Вигляд фрагмента екрана під час роботи програми зображено на рис. 19.

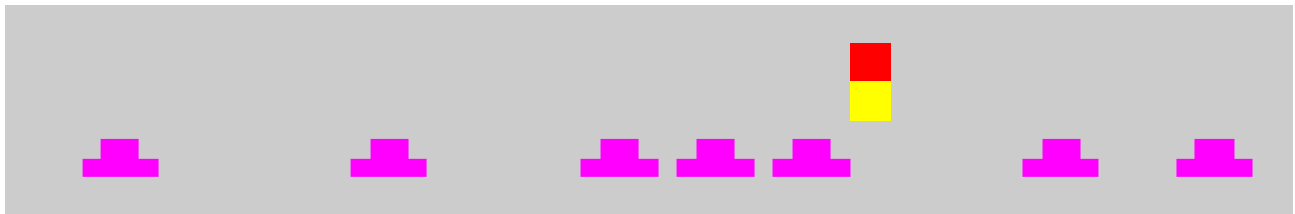


Рис. 19. Моделювання руху автомобілів на перехресті.

Пропонуємо читачам самостійно вдосконалити модуль `Queue`, перевірити правильність процедури `Move_All` і доповнити програму новими можливостями.

7.4. Реалізація стека

У програмуванні часто використовують особливий тип черги, в якій замовлення виконуються не в порядку надходження, а завжди перевага в обслуговуванні надається замовленню, яке надійшло останнім. Таку структуру даних називають *стеком* або *магазин*. Дані заносять у стек і вилучають лише з одного кінця – *вершини стека*. Стек працює за принципом «першим прийшов – останнім пішов». Деякий механізм за цим принципом працюватиме так: замовлення, яке надійшло першим, заносять у стек і одразу ж починають виконувати. Якщо під час виконання чергового замовлення надходить наступне, то робота механізму переривається, нове замовлення заносять у стек і механізм розпочинає роботу з цим новим замовленням. Попереднє ж залишається в стеку, очікуючи завершення опрацювання нового замовлення. Як тільки опрацювання замовлення завершується, воно вилучається зі стека, і механізм повертається до обслуговування попереднього. Залежно від часу надходження замовлень і потрібної тривалості їхньої обробки стек може видовжуватись, скорочуватись, залишатись деякий час порожнім.

Практична потреба в організації стека виникає, наприклад, під час перетворення транслятором арифметичних виразів програми з інфіксної форми запису

до постфіксної (до так званого польського запису), відслідковування послідовностей викликів процедур тощо.

Стек можна реалізувати за допомогою однонапрявленого списку, над яким можна виконувати такі операції:

- 1) додавання нової ланки лише на початок списку;
- 2) вилучення лише першої ланки списку.

Занесення і вилучення даних зі стека реалізують відповідно такі дві процедури:

```

function Push(var stack: Chain; x: ElementType):Boolean;
  { Push заносить елемент x у вершину стека stack }
var p: Chain;
begin if Max Avail>=SizeOf(x) then {вільної пам'яті є досить}
  begin New(p); with p^ do {створили нову ланку}
    begin element:=x; {занесли x в стек}
      link:=stack end; {with}
    stack:=p; Push:=true {вставили ланку}
  end else Push:=false {Вичерпана динамічна пам'ять}
end; {Push}

function Pop(var stack: Chain; var x: ElementType):Boolean;
  { Pop вилучає значення з вершини стека stack }
  { і записує його в x }
var p: Chain;
begin if stack=nil then Pop:=false {Стек порожній}
  else begin p:=stack; with stack^ do
    begin x:=element; stack:=link end; {вилучили значення}
    Dispose(p); Pop:=true end{if} {звільнили пам'ять}
end; {Pop}

```

Приклад використання стека наведемо в параграфі 8.3 у нерекурсивній процедурі обходу дерева.

8. Структура даних «дерево»

У попередніх параграфах описано способи реалізації та опрацювання лінійних списків: одно- та двонапрявлених. Зупинимось тепер на розгляді ієрархічних списків – ще однієї структури даних, характерної для багатьох класів задач – деревовидної.

8.1. Означення дерева

Деревовидною структурою (*деревом*) називають множину взаємопов'язаних об'єктів (які називають також вершинами або *вузлами*), розташованих по рівнях за таким правилом:

- на першому рівні – один вузол так званий *корінь* дерева;
- будь-який вузол X наступного, i -го ($i \neq 0$) рівня пов'язаний лише з одним вузлом Y попереднього, $(i-1)$ -го рівня.

У такому випадку Y називають безпосереднім предком вузла X , а X – безпосереднім нащадком Y . Якщо вузол немає потомків, то він називається

листом. Всі вузли, крім кореневого, які мають потомків, називаються внутрішніми вузлами, або *вершинами*.

Дерева застосовують для представлення об'єктів, які мають ієрархічну внутрішню будову. Наприклад, адміністративну структуру університету можна зобразити у вигляді дерева, коренем якого є ректор, а листками – зокрема, студенти; математичну формулу – деревом, листками якого є операнди, а вершинами – знаки операцій і так далі.

Є різні способи зображення дерев: за допомогою графа (рис. 20, а), вкладених множин (рис. 20, б) тощо. Ми інтерпретуватимемо дерева як списки з ланками, що можуть мати по декілька наступників (рис.20, в). Розглянуті вище однонапрямлені списки є, по суті, "виродженими" деревами, вершини якого мають тільки одного наступника.

Максимальна кількість безпосередніх нащадків того самого предка називається *степенем дерева*. Наприклад, на рис. 20 зображено дерево степеня два. Такі дерева називають *двійковими (бінарними) деревами*.

Усі вузли дерева, пов'язані з вершиною *X* безпосередньо або через інші вузли, називаються її *потомками*. Довільна вершина *X* дерева разом з усіма своїми потомками називається *піддеревом* цього дерева.

Як видно з рис. 20, в, кожна вершина дерева складається з полів зв'язку з потомками та інформативної частини, яка, залежно від потреби, містить інформацію довільного типу. Інформативну частину вершини дерева називатимемо *елементом* дерева. Щоб реалізувати таку структуру даних, можна використати, наприклад, такі оголошення:

```

type TreeElementType=... {тип інформаційної частини
    вершини може бути довільним залежно від потреби}
    Tree=^Node; {дерево}
    Node=record elem: TreeElementType; {елемент}
        left, right: Tree {зв'язок з нащадками}
    end; {Node}

```

Над деревом виконують такі основні операції:

- ◇ обхід дерева;
- ◇ вставка нової вершини (піддерева);
- ◇ вилучення вершини (піддерева).

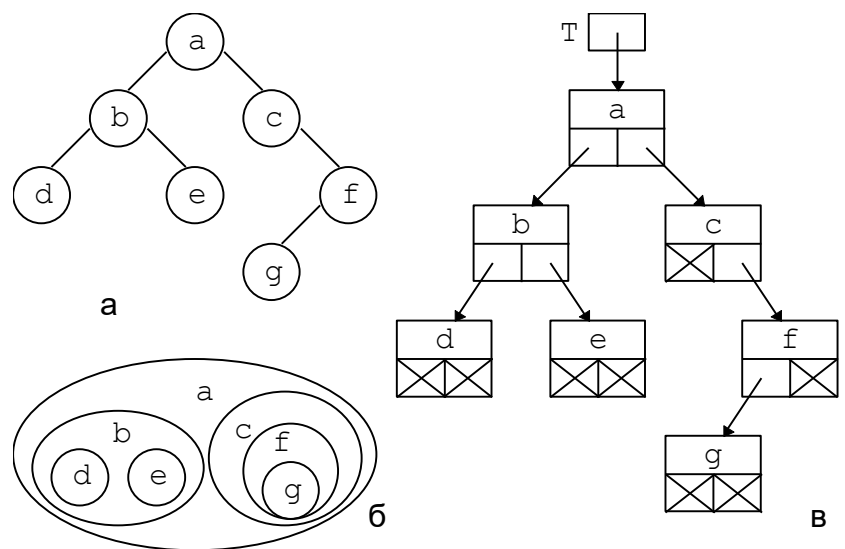


Рис. 20. Способи зображення дерева.

Найчастіше виконують обхід дерева. Під час обходу алгоритм повинен опрацювати всі вершини дерева, кожна – один раз. Дерево обходять, щоб відшукати певний елемент, роздрукувати всі елементи тощо. Є різні способи обходу, наприклад, лівосторонній, правосторонній, за рівнями. Під час лівостороннього обходу першим опрацюють крайній лівий листок, а останнім – крайній правий.

8.2. Рекурсія

Рекурсія – це метод визначення поняття, функції, розв’язку задачі через те саме поняття, функцію, розв’язок.

Дерево за своєю природою є рекурсивною структурою даних. Адже його означення можна сформулювати так: дерево з базовим типом Node – це

- 1) або порожнє дерево,
- 2) або деяка вершина типу Node зі скінченим числом пов’язаних з нею окремих дерев з базовим типом Node, які називаються піддеревами.

Ніклаус Вірт зазначає [2], що кожній структурі даних відповідає певний алгоритм обробки і відповідна структура керування, яка його реалізує. Масиву відповідає цикл з параметром, файлу – ітераційний цикл. Деревам відповідають рекурсивні алгоритми.

Рекурсивний розв’язок задачі з розміром вхідних даних n складається з двох частин:

- 1) визначення шуканого розв’язку через розв’язок задачі з меншим розміром вхідних даних;
- 2) розв’язок задачі найменшого розміру n_0 .

Проілюструємо сказане кількома прикладами.

1. Найбільший член заданої послідовності можна знайти за таким правилом:

$$\begin{aligned} \max(x_1, x_2, \dots, x_n) &= \text{більше_з_чисел}(x_1, \max(x_2, \dots, x_n)); & \max(x_1, x_2) \\ &= \text{більше_з_чисел}(x_1, x_2). \end{aligned}$$

2. Числа Фібоначчі: $f_0 = f_1 = 1$; $f_n = f_{n-1} + f_{n-2}$, $n = 3, 4, \dots$.

3. Нехай КД – функція обчислення кількості додатних членів заданої послідовності.

Тоді її можна визначити так:

$$\text{КД}(x_1, x_2, \dots, x_n) = \begin{cases} \text{КД}(x_2, \dots, x_n), & x_1 \leq 0 \\ \text{КД}(x_2, \dots, x_n) + 1, & x_1 > 0; \end{cases} \quad \text{КД}(x) = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0. \end{cases}$$

Рекурсія тут очевидна, але використання її було б неефективним, особливо у другому з них. Подумайте, чому.

Проте дуже корисним є механізм рекурсії під час роботи з деревовидними структурами. Адже *<переглянути_двійкове_дерево>* означає *<переглянути_ліве_піддерево>*, *<переглянути_вершину>*, *<переглянути_праве_піддерево>*. Використання рекурсії дає змогу програмі "пам’ятати" про всі ще не переглянуті піддерева. Зауважимо, що листок піддерев немає, і для нього перегляд зводиться до *<переглянути_вершину>* (розв’язок найпростішої задачі).

Рекурсивні алгоритми легко реалізувати за допомогою підпрограм мови Паскаль, оскільки будь-яка процедура чи функція може бути рекурсивною. Нагадаємо, що рекурсивною називається процедура (функція), в тілі якої є виклик самої себе. (Буває також непряма рекурсія, наприклад, у випадку, коли дві процедури містять виклики одна одної, але ми її не розглядатимемо.)

Розглянемо таку задачу: нехай елементами дерева є дійсні числа, тобто `TreeElementType = real`. Описати рекурсивну функцію, яка:

? визначає, чи входить елемент `E` в дерево `T`;

? обчислює середнє арифметичне елементів непорожнього дерева `T`.

Розв'язок.

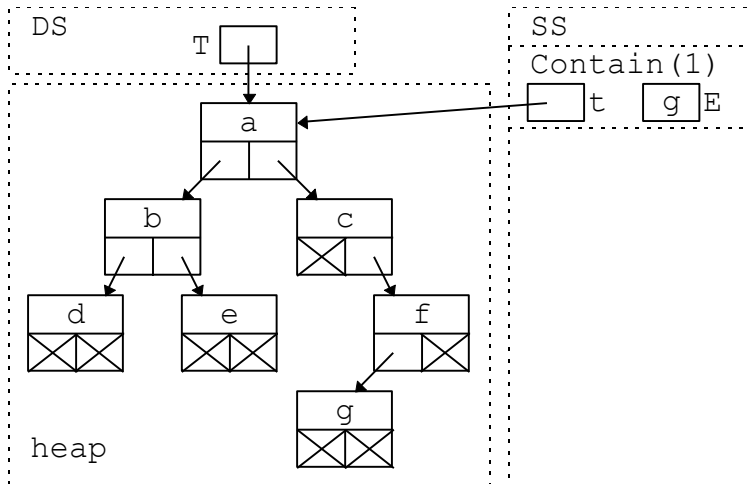
```

type TreeElementType=real;
      Tree=^Node;                               {дерево}
      Node=record elem: TreeElementType; {елемент}
            left, right: Tree {зв'язок з нащадками}
            end; {Node}
function Contain(E: TreeElementType; t: Tree): Boolean;
  { рекурсивна функція Contain перевіряє, }
  {   чи елемент E належить дереву t   }
  var b: Boolean;
begin b:=t<>nil; if b then                       {дерево непорожнє}
  with t^ do
    begin b:=b and (E=elem);      {i містить у вершині E}
          b:=b or Contain(E,Left) {або ліворуч}
          or Contain(E,Right)    {чи праворуч}
    end; {with & if} Contain:=b
end; {Contain}

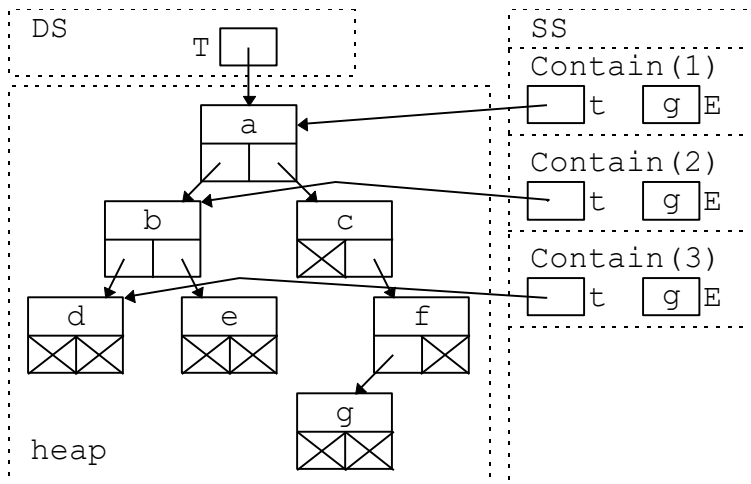
function Average(T: tree): real;
  { Average обчислює середнє арифметичне дійсних чисел, }
  {   що є елементами дерева T   }
  var s: real;      {сума}
      k: word;      {кількість}
procedure SumAndCount(var s: real; var k: word; t: tree);
  { службова рекурсивна процедура для обчислення суми }
  { i кількості елементів дерева T }
  var s1: real; k1: word; {робочі змінні}
begin with T^ do
  begin s:=elem; k:=1;
    if Left<>nil then      {переглянемо ліве піддерево}
    begin SumAndCount(s1,k1,Left);
          s:=s+s1; inc(k,k1) end; {if}
    if Right<>nil then    {переглянемо праве піддерево}
    begin SumAndCount(s1,k1,Right);
          s:=s+s1; inc(k,k1) end {if}
    end {with}
end; {SumAndCount}
begin {Average}
  SumAndCount(s,k,t); Average:=s/k
end; {Average}

```

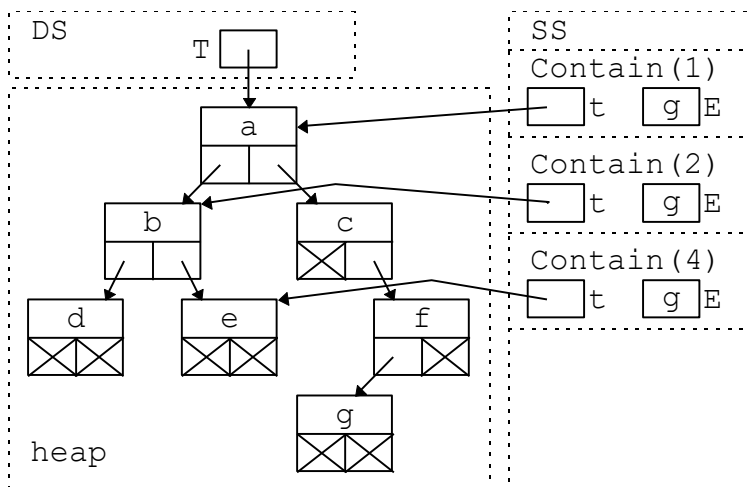
Послідовність дій за алгоритмом рекурсивної функції `Contain` проілюструємо прикладом відшукування елемента `g` у дереві `T`, яке було зображено на рис. 20, в. Вміст вершин цього дерева умовно позначено літерами, бо тип елемента для ілюстрації немає принципового значення. У конкретній ситуації він може бути, наприклад, літерний, числовий, рядковий, чи будь-який інший.



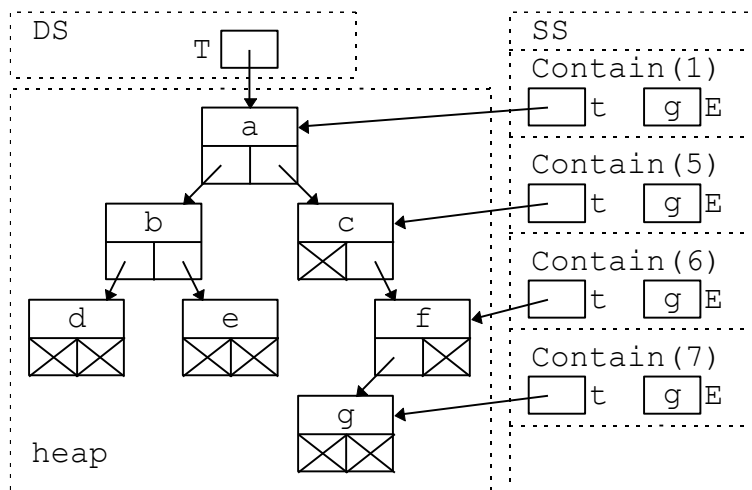
У момент виклику функції `Contain` її параметр `t` отримав вказівник на дерево `T`, яке було створене раніше.



Відбулося два рекурсивних виклики `Contain(E, t^.left)`. Тепер системний стек містить змінні трьох екземплярів функції.



У другому екземплярі функції відбувся виклик `Contain(E, t^.right)`.



Так схематично можна зобразити зв'язки між змінними після останнього рекурсивного виклику функції `Contain`. У наступний момент всі її екземпляри, починаючи з останнього, завершать роботу і будуть вилучені зі стека.

Рис. 21. Лівосторонній обхід дерева рекурсивною підпрограмою.

Видно, що час пошуку елемента функцією `Contain` є величиною порядку $O(n)$, де n – загальна кількість вершин дерева. У найгіршому випадку вона мусить обійти всі вершини даного дерева, як зображено на рис. 21.

Порівняйте цю оцінку швидкодії з оцінкою часу відшукування елемента у збалансованому дереві пошуку, яка буде наведена у параграфі 8.7.

8.3. Нерекурсивний алгоритм обходу дерева

Алгоритм, що виконує обхід дерева повинен зберігати інформацію про ті вказівники на піддерева, які він «проминув». Рекурсивні процедури з цією метою використовують системний стек. Він містить дані усіх рекурсивно викликаних екземплярів такої процедури. Нерекурсивна процедура обходу дерева може використовувати власний стек, щоб заносити в нього інформацію про ті піддерева, до перегляду яких потрібно ще повернутися. Алгоритм завершується після вичерпання такого стека.

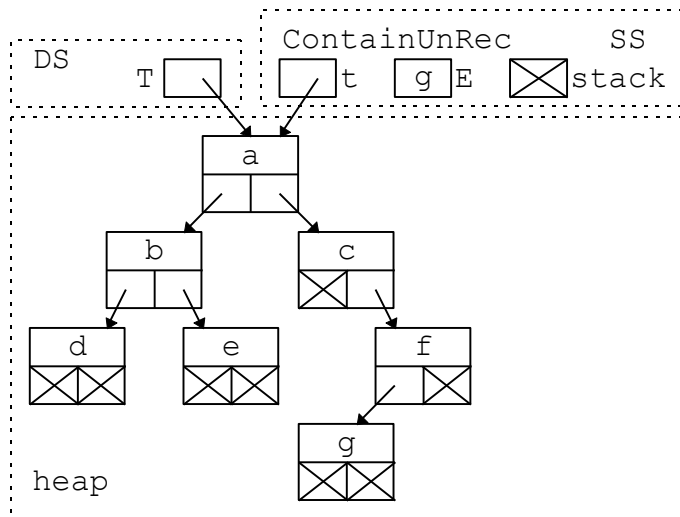
Нижче наведено текст нерекурсивної функції `ContainUnRec` (аналога описаної раніше `Contain`), яка перевіряє, чи входить елемент `E` в задане дерево `T`. Вона використовує власний стек – змінну `stack` типу `Chain`, і функції `Push` і `Pop` – відповідно занесення і вилучення зі стека. Ці функції і тип `Chain` описані у попередніх параграфах. Тип ланки стека задано так:

```

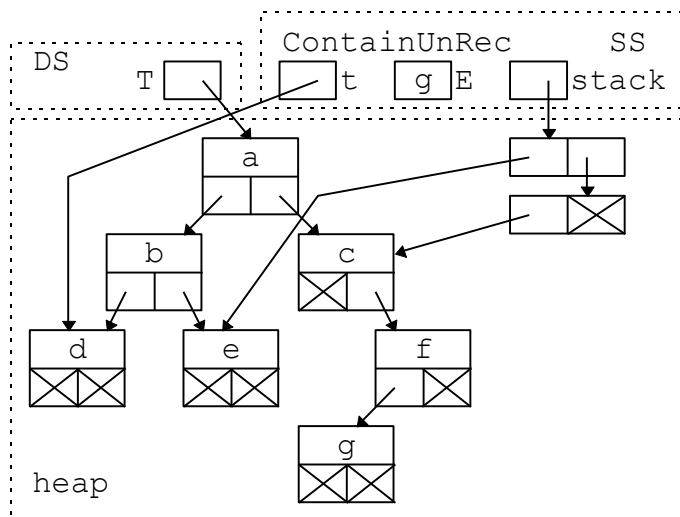
type ElementType=Tree;
function ContainUnRec (E: TreeElementType; T: Tree): Boolean;
{ нерекурсивна функція ContainUnRec перевіряє, чи елемент E }
{ належить дереву T. Для запам'ятовування неопрацьованих }
{ віток дерева вона використовує власний стек }
var b: Boolean; stack: Chain;
begin if T=nil then {порожнє дерево нічого не містить}
    ContainUnRec:=false else
    begin b:=true; {надіємось, що елемент знайдеться}
        repeat with T^ do
            if elem=E then {знайшли E у корені дерева}

```

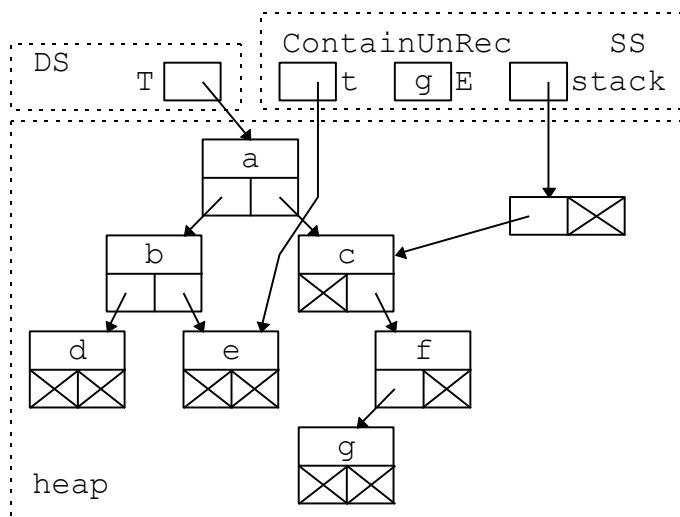
```
begin ContainUnRec:=true; Exit end
```



Функція ContainUnRec містить додаткову змінну stack для запам'ятовування тих віток дерева, які тимчасово проминули. Спочатку стек порожній.



Відбувся спуск лівою віткою дерева. Стек містить вказівники на праві вітки тих вершин, пошук у яких уже відбувся. Зауважимо, що першим у стек занесли вказівник на вершину «c».



Після перегляду вершини «d» функція переглядає вершину «e», вилучивши вказівник на неї зі стека.

Далі функція вилучить зі стека вказівник на вершину «c» і завершить пошук, спустившись до вершини «g».

Рис. 22. Нерекурсивний обхід дерева.

```
else begin stack:=nil;      {шукатимемо в піддеревах}
                {потрібно запам'ятати шлях до правого}
if not Push(stack,right) then
    begin writeln('Вичерпана купа');
          ContainUnRec:=false; Exit end;
```

```

    if left<>nil then T:=left      {шукатимемо ліворуч}
    else b:=Pop(stack,T)          {повернемось до правого}
    end {else&if&with}
  until not b;
  ContainUnRec:=false           {оскільки переглянули все дерево}
end                               {і не знайшли}
end; {ContainUnRec}

```

Послідовність виконання функції `ContainUnRec` зображено на рис. 22. Як і в попередньому випадку час її роботи є величиною порядку $O(n)$.

8.4. Дерева пошуку

Вузли дерева можуть містити спеціальне поле для їхнього імені – так званого *ключа*. Ключами можуть бути довільні коди (числові, літерні), що допускають порівняння. Всі ключі в дереві повинні бути різними. Вони служать для ідентифікації вузлів дерева. Завдяки використанню ключів алгоритми обробки деревовидних структур стають простішими, адже опрацьовують не власне елементи, а їхні ключі.

Якщо для кожної вершини правильним є твердження про те, що ліве її піддерево містить лише вершини з меншими ключами, а праве – з більшими, то таке дерево називають *деревом пошуку*. Дерева пошуку зручно використовувати для організації швидкого доступу до потрібних частин великих масивів даних. Для цього даним приписують певні ключі і розташовують їх у вузлах відповідного дерева. Потрібного ключа можна знайти, скориставшись алгоритмом бінарного пошуку. Час його виконання є величиною порядку $O(\log_2 n)$, де n – кількість ключів у дереві.

Найефективнішим для пошуку є використання *ідеально збалансованих* дерев. Дерево пошуку називається ідеально збалансованим, якщо кількість вершин у всіх його відповідних лівих і правих піддеревах відрізняється не більше, ніж на одиницю.

8.5. Пошук з включенням у дереві

Розглянемо спочатку випадок, коли дерево пошуку тільки росте. Для цього розв'яжемо таку задачу: визначити частоту входження кожного слова у задану послідовність слів, яка міститься у заданому текстовому файлі по одному слову в записі файла.

Щоб розв'язати цю задачу, побудуємо двійкове дерево пошуку. Кожна його вершина міститиме саме слово (ключ), кількість його входжень у послідовність (корисна інформація) і вказівники на праве та ліве піддерева. Спочатку дерево порожнє. Кожне прочитане з файла слово потрібно шукати в дереві. Якщо слово знайдено, його лічильник збільшуємо на одиницю. У іншому випадку слово включаємо в дерево з одиничним значенням лічильника.

Таку задачу називають *пошуком по дереву з включенням*. Її реалізує наведена нижче програма.

```

program TreeSearch;

```

```

type str=string[40];           {вважається, що довжина слова < 41}
    tree=^node;                {дерево}
    node=^record key:str; {вершина дерева містить слово-ключ,}
        count:word;          {лічильник слова,}
        left,right:tree end; {поля зв'язку з піддеревами.}
procedure PrintTree(T:tree; shift:byte);
{ процедура PrintTree роздруковує дерево T, виконуючи }
{ лівосторонній обхід; вершини нижчих рівнів друкуються }
{ з відступом shift стосовно вершин вищих рівнів }
    var i:byte;
begin if T<>nil then           {опрацьовуємо непорожнє дерево}
    with T^ do
        begin PrintTree(left,shift+1);    {спочатку друкуємо ліве}
            {піддерево, потім - слово і його лічильник,}
            for i:=1 to shift do write(' '); writeln(count,' ',key);
            PrintTree(right,shift+1)      {а тепер - праве.}
        end{with}
    end; {PrintTree}
procedure Search(s:str; var T:tree);
{ процедура Search виконує пошук з включенням }
{ слова s у дереві T }
begin if T=nil then {слово не знайшли, включаємо його в дерево}
    begin T:=New(tree); with T^ do
        begin key:=s; count:=1;
            left:=nil; right:=nil end
        end else with T^ do           {шукаємо в непорожньому дереві:}
        if s<key then Search(s,left) else      {менший ключ - ліворуч,}
        if s>key then Search(s,right) else     {більший - праворуч.}
            inc(count)           {знайшли ключ! Збільшуємо його лічильник}
        end; {Search}
var f:text; fileName:string; {змінні для приєднання до}
        {заданого зовнішнього текстового файлу}
    s:str; T:tree;              {будуватимемо дерево T}
Begin write('Введіть ім'я файла: '); readln(fileName);
    assign(f,filename); reset(f);           {відкрили файл}
    T:=nil                                  {дерево спочатку порожнє}
    while not EoF(f) do                   {читаємо і вставляємо всі слова}
        begin readln(f,s); Search(s,T)
        end; close(f);
    PrintTree(T,0);           {тепер можна побачити дерево на екрані}
    readln {для завершення роботи програми натисніть <Enter>}
End.

```

Збалансування дерева T, побудованого цією програмою, залежить від порядку слів у заданій послідовності. Не можна передбачити, як воно буде рости і якої набуде форми. Напевне, це дерево не буде ідеально збалансованим. Перебудова дерева після кожного включення для досягнення ідеальної збалансованості є занадто дорогою (довготривалою) процедурою. Її виконують для дерев, які довго залишаються без змін і використовують переважно для пошуку.

Вправа. Простежити, як відбуватиметься ріст дерева, якщо файл містить слова 'h', 'i', 'c', 'f', 'j', 'd', 'b', 'g', 'e'.

8.6. Вилучення з дерева пошуку

Вилучення з дерева є задачею оберненою до попередньої. На жаль, вона є складнішою, особливо, для дерев пошуку. Простою ця задача є лише тоді, коли потрібно вилучити листок або внутрішню вершину з одним нащадком. Якщо ж вершина, що вилучається, має двох нащадків, її треба замінити або на крайній правий елемент лівого піддерева, або на крайній лівий правого. Ці елементи можуть мати тільки одного нащадка.

Наведена нижче процедура `Withdraw` розрізняє такі три випадки:

- 1) компоненти з заданим ключем немає;
- 2) компонента з заданим ключем має не більше, ніж одного нащадка;
- 3) компонента з заданим ключем має двох нащадків.

```

procedure Withdraw(s:str; var T:tree);
var q:tree;
    procedure del(var r:tree);
    begin if r^.right<>nil then del(r^.right,h)
        else begin q^.key:=r^.key; q^.count:=r^.count;
            q:=r; r:=r^.left end
    end; {del}
Begin {Withdraw}
    if T=nil then {ключа в дереві нема, do nothing}
    else if s<T^.key then Withdraw(s,T^.left)
    else if s>T^.key then Withdraw(s,T^.right)
    else begin q:=T;
        if q^.right=nil then t:=q^.left
        else if q^.right=nil then t:=q^.right
        else del(t^.left);
            Dispose(q) end
end; {Withdraw}

```

Вправа. Простежити як буде змінюватися збудоване раніше дерево під час вилучення слів 'i', 'b', 'd'.

8.7. Збалансовані дерева

Ідеальне збалансування дерева пошуку виконати складно і дорого, тому частіше використовують просте збалансування.

Дерево називається *збалансованим* тоді і лише тоді, коли висоти двох піддерев кожної з його вершин відрізняються не більше, ніж на одиницю.

Ця вимога є суттєво слабшою порівняно з умовами ідеальної збалансованості. Дерево, яке має однакову *кількість* вершин у піддеревих, має піддерева однакової *висоти*, але не навпаки.

Збалансовані дерева називають ще AVL-деревими за іменами їхніх винахідників Г. М. Адельсон-Вельського та Є. М. Ландіса (1962). Для AVL-дерев час пошуку теж є величина порядку $O(\log_2 n)$, а довжина AVL-дерева в найгіршому випадку не перевищує 1,45 довжини ідеально збалансованого дерева.

Включення та вилучення елемента збалансованого дерева виконується дещо складніше.

8.8. Вставка у збалансоване дерево

Розглянемо як зміниться збалансоване дерево після включення у нього нової вершини. Нехай дерево містить корінь T , ліве піддерево L з висотою h_L і праве піддерево R з висотою h_R . Припустимо, що нову вершину включили в ліве піддерево. Це збільшить його висоту на одиницю. Тоді можливі три випадки.

1. Було $h_L = h_R$. Після вставки L і R матимуть різну висоту, але критерій збалансованості не порушиться.
2. Було $h_L < h_R$. Після вставки L і R матимуть однакову висоту, збалансованість стала кращою.
3. Було $h_L > h_R$. Після вставки критерій збалансованості буде порушено, і дерево потрібно перебудувати.

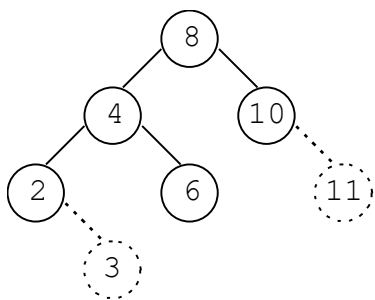


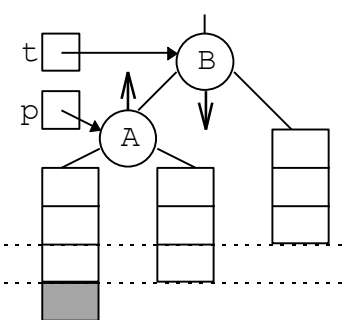
Рис. 23. Збалансоване дерево.

Розглянемо приклад, зображений на рис. 23. Включення вершини 11 (або 9) зробить дерево 10 одностороннім без порушення збалансованості (випадок 1), а дерево 8 – краще збалансованим (випадок 2). Включення вершини 3 (або будь-якої з 1, 5, 7) вимагатиме подальшого балансування дерева.

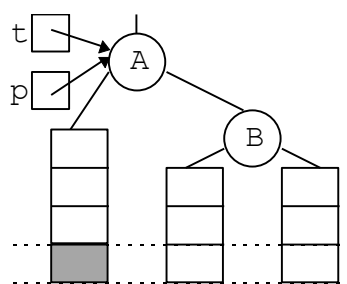
Суттєво різними є лише дві ситуації: включення 1 (або 3) і включення 5 (або 7). Схеми на рис. 24 демонструють як відновити балансування. Вершини дерева переміщуються лише у вертикальному напрямі з рівня на рівень, не порушуючи відношення порядку «ліворуч – праворуч» між вершинами дерева. Впорядкування ключів не змінюється під час переміщень.

з рівня на рівень, не порушуючи відношення порядку «ліворуч – праворуч» між вершинами дерева. Впорядкування ключів не змінюється під час переміщень.

Перед балансуванням

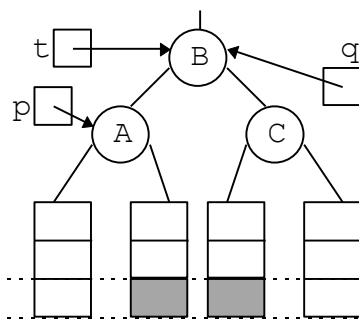
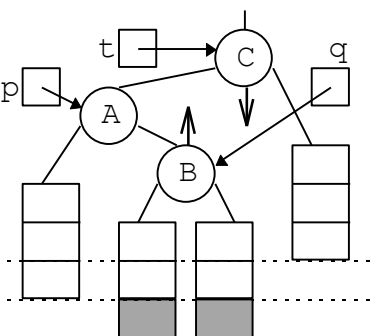


Після балансування



Один LL обмін можна виконати за допомогою таких операторів

```
{t=@B}
p:=t^.left; {p=@A}
t^.left:=p^.right;
p^.right:=t; t:=p;
```



Подвійний LR обмін

```
{t=@C}
p:=t^.left; {p=@A}
q:=p^.right; {q=@B}
p^.right:=q^.left;
q^.left:=p; {обмін A-B}
t^.left:=q^.right;
q^.right:=t; {обмін C-B}
t:=q;
```

Рис. 24. Характерні ситуації відновлення збалансованості.

Алгоритм впорядкування суттєво залежить від того, як зберігається інформація про збалансування. Дерево містить її уже в своїй структурі, тому можна взагалі нічого не дописувати ні в одну вершину. Тоді, щоб перевірити збалансування, довелося б щоразу переглядати піддерева вершини, до якої долучили нову. Це дуже великі затрати. Ми розширимо структуру вершини, додавши в неї поле `bal`, яке міститиме різницю висот правого і лівого піддерев $h_R - h_L$.

Тепер процес включення можна виконати так:

- 1) хід вперед шляхом пошуку, доки не переконаємось, що ключа в дереві немає;
- 2) включення вершини і визначення результуючого показника збалансованості;
- 3) обернений хід шляхом пошуку і перевірка в кожній вершині показника збалансованості, виконання у разі потреби перебудови дерева.

Із зробленими припущеннями можна буде легко вдосконалити вже описану раніше процедуру `Search` для виконання додаткових дій під час повернення шляхом пошуку. З цією метою додамо в заголовок процедури параметр `rise`, який має значення «висота дерева збільшилась».

Розглянемо детальніше, які перевірки і зміни потрібно виконати у вершині дерева після повернення з її лівого піддерева. (Повернення з правого піддерева обробляється аналогічно з відповідною заміною імен полів зв'язку вершини дерева.) Нехай процес повернувся у вершину p^{\wedge} . Залежно від висот її піддерев перед включенням потрібно розрізнити такі ситуації.

1. Було $h_L = h_R$, $h_R - h_L = p^{\wedge}.bal = 0$. Стало $h_L := h_L + 1$, $p^{\wedge}.bal = -1$. Ліве піддерево переважає на 1, але критерій збалансованості не порушено.
2. Було $h_L < h_R$, $h_R - h_L = p^{\wedge}.bal = 1$. Стало $h_L := h_L + 1$, $p^{\wedge}.bal = 0$. Досягли рівноваги.
3. Було $h_L > h_R$, $h_R - h_L = p^{\wedge}.bal = -1$. Стало $h_L := h_L + 1$, $p^{\wedge}.bal = -2$. Дерево потребує перебудови. Щоб її виконати, потрібно в'яснити, котра зі згаданих раніше ситуацій балансування має місце:
 - a) якщо $p^{\wedge}.left^{\wedge}.bal = -1$, то застосуємо перший спосіб балансування;
 - b) якщо $p^{\wedge}.left^{\wedge}.bal = 1$, то – другий.

Балансування полягає у циклічному переприсвоюванні вказівників, що призводить до одно- чи двократного обміну двох чи трьох вершин, які беруть участь у процесі. Крім обертання вказівників потрібно, також належно виправити показники збалансованості.

```

type str=string[40];           {вважається, що довжина слова < 41}
                                {показник збалансованості може набувати}
                                {лише трьох значень: -1, 0, 1}
  balance=-1..1;                {лише трьох значень: -1, 0, 1}
  tree=^node;                    {дерево}
  node=^record key:str; {вершина дерева містить слово-ключ,}
                                count:word;           {лічильник слова,}
                                bal:balance;           {показник збалансованості,}
                                left,right:tree end;   {поля зв'язку з піддеревими.}

```

```

procedure SearchAndBalance(s:str; var T:tree; var rise:Boolean);

```

```

{ процедура SearchAndBalance виконує пошук з включенням }
{   слова s у збалансованому дереві T   }
var p,q:tree;           {робочі змінні для обміну вершин}
begin {спочатку вважається, що rise=false}
  IF T=nil THEN {слово не знайшли, включаємо його в дерево}
  begin T:=New(tree); rise:=true;           {дерево збільшилось}
    with T^ do begin key:=s; count:=1;
      left:=nil; right:=nil end
    end ELSE           {шукаємо в непорожньому дереві:}
  IF s<T^.key THEN
  begin SearchAndBalance(s,T^.left,rise); {менший ключ - ліворуч}
    if rise then           {ліва гілка виросла}
      case T^.bal of      {як змінилось балансування ?}
      1: begin T^.bal:=0; rise:=false end;
      0: T^.bal:=-1;
      -1: begin p:= T^.left;           {потрібна перебудова !}
        if p^.bal=-1 then           {один LL обмін}
          begin T^.left:=p^.right; p^.right:=T;
            T^.bal:=0; T:=p end
          else           {подвійний LR обмін}
          begin q:=p^.right; p^.right:=q^.left;
            q^.left:=p; T^.left:=q^.right; q^.right:=T;
            if q^.bal=-1 then T^.bal:=1 else T^.bal:=0;
            if q^.bal=1 then p^.bal:=-1 else p^.bal:=0;
            T:=q
          end; {else & if}
          T^.bal:=0; rise:=false end {перебудови}
        end {case & if raise}
      end ELSE
  IF s>T^.key THEN
  begin SearchAndBalance(s,T^.right,rise); {більший - праворуч}
    if rise then           {права гілка виросла}
      case T^.bal of      {як змінилось балансування ?}
      -1: begin T^.bal:=0; rise:=false end;
      0: T^.bal:=1;
      1: begin p:= T^.right;           {потрібна перебудова !}
        if p^.bal=1 then           {один RR обмін}
          begin T^.right:=p^.left; p^.left:=T;
            T^.bal:=0; T:=p end
          else           {подвійний RL обмін}
          begin q:=p^.left; p^.left:=q^.right;
            q^.right:=p; T^.right:=q^.left; q^.left:=T;
            if q^.bal=1 then T^.bal:=-1 else T^.bal:=0;
            if q^.bal=-1 then p^.bal:=1 else p^.bal:=0;
            T:=q
          end; {else & if}
          T^.bal:=0; rise:=false end {перебудови}
        end {case & if raise}
      end ELSE
    inc(T^.count) {знайшли ключ! Збільшуємо його лічильник}
  end; {SearchAndBalance}

```

Складність операції балансування передбачає, що збалансовані дерева треба використовувати лише тоді, коли пошук інформації відбувається набагато частіше, ніж її включення.

Вправа. Нехай ключами дерева пошуку є натуральні числа. Простежити, як відбувається його ріст і балансування, якщо дерево будується з послідовності чисел 4, 5, 7, 2, 1, 3, 6.

8.9. Вилучення зі збалансованих дерев

Для звичайних дерев вилучення вершини було більш трудомістким, ніж включення. Така ж ситуація спостерігається і для збалансованих дерев.

Принципова схема алгоритму залишається такою самою: простими є випадки вилучення листка чи вершини з одним нащадком. Якщо ж вершина має два піддерева, то її замінюємо на крайню праву вершину її лівого піддерева. Для контролю за необхідністю балансування використовується логічна змінна *diminish*. Вона вказує, чи зменшилась висота дерева. Перебудови дерева з метою збалансування (як і під час включення) базуються на одно- або двократних обмінах вершин.

```

procedure WithdrawAndBalance
    (s:str; var T:tree; var diminish:Boolean);
var q:tree;
    procedure del(var r:tree; var h:Boolean);
    begin {h=false} if r^.right<>nil
        then begin del(r^.right,h); if h then balanceR(r,h) end {then}
        else begin q^.key:=r^.key; q^.count:=r^.count;
            q:=r; r:=r^.left; h:=true end
    end; {del}
    procedure balanceL(var t:tree; var diminish:Boolean);
    var p,q:tree; bp,bq:Balance;
    begin {diminish=true ліва гілка стала коротшою}
        case t^.bal of
        -1: t^.bal:=0;
        0: begin t^.bal:=1; diminish:=false end;
        1: {перебудова} begin p:=t^.right; bp:=p^.bal;
            if bp>=0 then {один RR обмін}
            begin t^.right:=p^.left; p^.left:=t;
                if bp=0 then begin t^.bal:=1; p^.bal:=-1;
                    diminish:=false end
                else begin t^.bal:=0; p^.bal:=0 end; t:=p end {then}
            else {подвійний RL обмін} begin q:=p^.left;
                bq:=q^.bal; p^.left:=q^.right; q^.right:=p;
                t^.right:=q^.left; q^.left:=t;
                if bq=1 then t^.bal:=-1 else t^.bal:=0;
                if bq=-1 then p^.bal:=1 else p^.bal:=0;
                t:=q; q^.bal:=0
            end {else&if} end {перебудова}
        end {case}
    end; {balanceL}
procedure balanceR(var t:tree; var diminish:Boolean);
    var p,q:tree; bp,bq:Balance;

```

```

begin {diminish=true права гілка стала коротшою}
  case t^.bal of
    1: t^.bal:=0;
    0: begin t^.bal:=-1; diminish:=false end;
   -1: {перебудова} begin p:=t^.left; bp:=p^.bal;
      if bp<=0 then {один LL обмін}
        begin t^.left:=p^.right; p^.right:=t;
          if bp=0 then begin t^.bal:=-1; p^.bal:=1;
            diminish:=false end
          else begin t^.bal:=0; p^.bal:=0 end; t:=p end {then}
        else {подвійний LR обмін} begin q:=p^.right;
          bq:=q^.bal; p^.right:=q^.left; q^.left:=p;
          t^.left:=q^.right; q^.right:=t;
          if bq=-1 then t^.bal:=1 else t^.bal:=0;
          if bq=1 then p^.bal:=-1 else p^.bal:=0;
          t:=q; q^.bal:=0
        end {else&if} end {перебудова}
      end {case}
  end; {balanceR}
begin {WithdrawAndBalance, diminish=false}
  if t=nil then {ключа в дереві немає, do nothing}
  else if t^.key>s then
    begin WithdrawAndBalance(s,t^.left,diminish);
      if diminish then balanceL(t,diminish)
    end else if t^.key<s then
    begin WithdrawAndBalance(s,t^.right,diminish);
      if diminish then balanceR(t,diminish)
    end else {виключення t^} begin q:=t;
      if q^.right=nil then begin t:=q^.left; diminish:=true end
      else if q^.right=nil then begin t:=q^.right; diminish:=true
    end
  else begin del(t^.left,diminish);
      if diminish then balanceL(t,diminish) end; Dispose(q)
    end
  end; {WithdrawAndBalance}

```

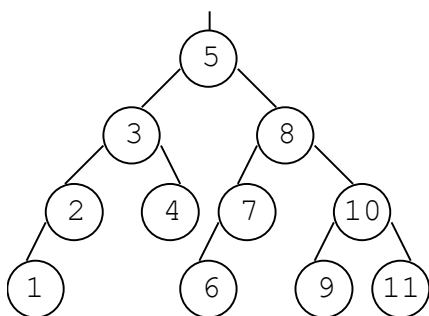


Рис. 25

Балансування виконується, якщо змінна `diminish` набуває значення «істина». Це трапляється тоді, коли з дерева вилучено знайдену вершину, або коли висота дерева змінилась у ході балансування.

У поведінці процедур `SearchAndBalance` і `WithdrawAndBalance` є суттєва різниця. Якщо включення одного елемента може призвести щонайбільше до одного обміну (двох чи трьох вершин), то вилучення може потребувати обмінів у всіх вершинах шляху пошуку. Проте ймовірність балансування дерева після вставки є вищою, ніж після вилучення. Експериментально встановлено, що один обмін трапляється на кожні дві вставки і на кожні п'ять вилучень, тому затрати на вставку і вилучення є приблизно однаковими.

Вправа. Простежити, як буде змінюватися збалансоване дерево, зображене на рис. 25 під час вилучення ключів 4, 8, 6, 5, 2, 1, 7.

9. Завершальний розділ

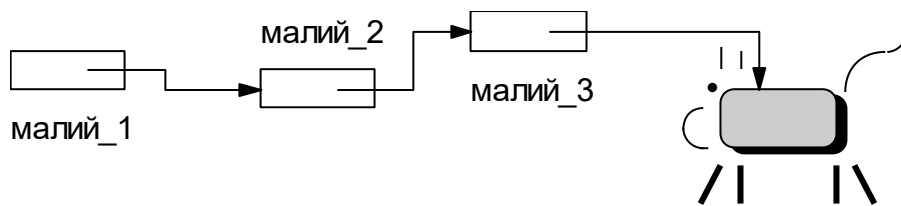
Автори радіють з того, що цей розділ не є першим, який Ви читаете.

Бажаємо всяких успіхів і поприв'язувати всіх телят за допомогою чудових «малих» (це жартома).

Важко переоцінити значення вказівників для сучасних технологій програмування. Мабуть, найширше їх використовують в об'єктно-орієнтованих програмах. Тому будемо раді, якщо цей посібник допоможе Вам у вивченні такого важливого розділу програмування.

У текстах лекцій ми свідомо зробили наголос на динамічних структурах даних. Фундаментом будь-якого алгоритму є відповідно підібрані структури даних. Дуже часто їх правильний вибір є вирішальним для успішного створення програми.

Посібник ні якою мірою не претендує на повноту. Надіємося, що він заохотить читачів до вивчення спеціальної літератури, присвяченої цій тематиці.



10. Список літератури

1. *Абрамов В. Г., Трифонов Н. П., Трифонова Г. Н.* Введение в язык Паскаль. – М.: Наука, 1988.
2. *Вирт Н.* Алгоритмы + структуры данных = программы. – М.: Мир, 1985.
3. *Гудман С., Хидетниеми С.* Введение в разработку и анализ алгоритмов. – М.: Мир, 1981.
4. *Дуго С. М., Клешко Г. Н., Мишенин А. И., Петров Е. А.* Сборник задач по курсу "Информационные системы и структуры данных". – М.: Статистика, 1981.
5. *Довгаль С. И., Литвинов Б. Ю., Сбитнев А. И.* Персональные ЭВМ: Турбо Паскаль V 6.0. Объектное программирование. Локальные сети. – К.: Информсистема сервис, 1993.
6. *Зуев Е. А.* Язык программирования Turbo Pascal 6.0. – М.: Унитех, 1992.
7. *Лэнгсам Й., Огенстайн М., Тененбаум А.* Структуры данных для ПЭВМ. – М.: Мир, 1989.
8. *Пильщиков В. Н.* Сборник упражнений по языку Паскаль. – М.: Наука, 1989.
9. *Семашко Г. Л., Салтыков А. И.* Программирование на языке Паскаль. – М.: Наука, 1988.
10. *Страуструп Б.* Язык программирования C++. Второе издание. К.: Диалектика, 1993.
11. *Фаронов В. В.* Турбо Паскаль 7.0. Начальный курс. Учебное пособие. – "Нолидж", 1997.
12. Turbo Pascal Version 7.0. Programmer's Guide. Borland International Inc. 1995.