Зміст рекурсії

Рекурсією називається ситуація, коли підпрограма, або функція викликає сама себе. Вперше стикаючись з такою алгоритмічної конструкцією, більшість людей відчуває певні труднощі, однак трохи практики і рекурсія стане зрозумілим і дуже корисним інструментом у вашому програмістську арсеналі.

Процедура або функція може містити виклик інших процедур або функцій. У тому числі процедура може викликати сама себе. Ніякого парадоксу тут немає - комп'ютер лише послідовно виконує інструкцію і, якщо зустрічається виклик процедури, просто починає виконувати цю процедуру. Без різниці, яка процедура дала команду це робити.

Приклад рекурсивної процедури:

```
procedure Rec(a: integer);
begin
  if a>0 then
    Rec(a-1);
  writeln(a);
end;
```

Розглянемо, що відбудеться, якщо в основній програмі поставити виклик, наприклад, виду $Rec(3)$. Нижче представлена блок-схема, що показує послідовність виконання операторів.

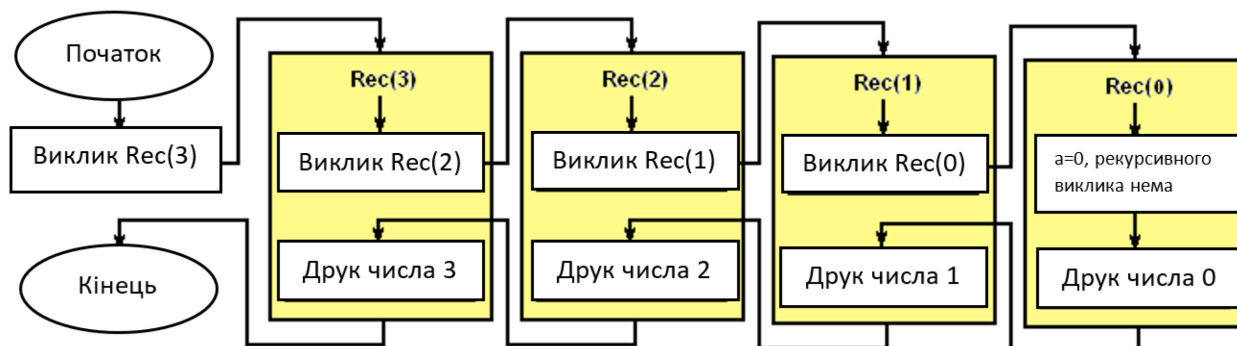

Рис. 1. Блок схема роботи рекурсивної процедури.

Процедура Rec викликається з параметром $a = 3$. У ній міститься виклик процедури Rec з параметром $a = 2$. Попередній виклик ще не завершився, тому можете уявити собі, що створюється ще одна процедура і до закінчення її роботи

першого свою роботу не закінчує. Процес виклику закінчується, коли параметр $a = 0$. У цей момент одночасно виконуються 4 примірники процедури. Кількість одночасно виконуваних процедур називають *глибиною рекурсії.*

Четверта викликана процедура ($Rec\ (0)$) надрукує число 0 і закінчить свою роботу. Після цього управління повертається до процедури, яка її викликала ($Rec\ (1)$) і друкується число 1. І так далі поки не завершаться всі процедури. Результатом вихідного виклику буде печатка чотирьох чисел: $0, 1, 2, 3$.

Ще один візуальний образ того, що відбувається представлений на *рис. 2.*

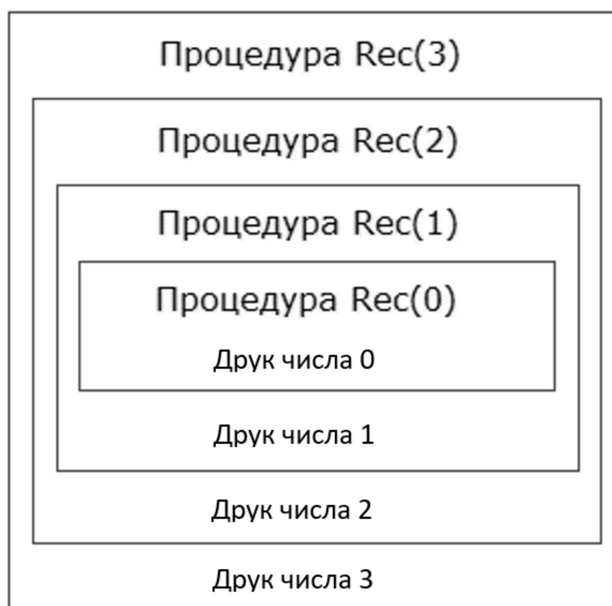

Рис. 2. Виконання процедури Rec з параметром 3 складається з виконання процедури Rec з параметром 2 і друку числа 3. У свою чергу виконання процедури Rec з параметром 2 складається з виконання процедури Rec з параметром 1 і друку числа 2. і. т. д.

В якості самостійного вправи подумайте, що вийде при виклику $Rec\ (4)$. Також подумайте, що вийде при виклику описаної нижче процедури $Rec2\ (4)$, де оператори помінялися місцями.

```
procedure Rec2(a: integer);
begin
  writeln(a);
  if a>0 then
    Rec2(a-1);
end;
```

Зверніть увагу, що в наведених прикладах рекурсивний виклик стоїть всередині умовного оператора. Це необхідна умова для того, щоб рекурсія закінчилася. Також зверніть увагу, що сама себе процедура викликає з іншим параметром, не таким, з яким була викликана вона сама. Якщо в процедурі не використовуються глобальні змінні, то це також необхідно, щоб рекурсія не продовжувались до нескінченності.

Складна рекурсія

Можлива трохи більш складна схема: функція A викликає функцію B, а та в свою чергу викликає A. Це називається *складної рекурсією.* При цьому виявляється, що описувана перший процедура повинна викликати ще не описану. Щоб це було можливо, потрібно використовувати випереджальний опис.

Приклад:

```
procedure A(n: integer); {Випереджувальний опис (заголовок) першої процедури}
procedure B(n: integer); {Випереджуваючий опис другої}
procedure A(n: integer); {Повний опис процедури A}
begin
  writeln(n);
  B(n-1);
end;
procedure B(n: integer); {Повний опис процедури B}
begin
  writeln(n);
  if n<10 then
    A(n+2);
end;
```

Випереджаючий опис процедури B дозволяє викликати її з процедури A. Випереджаючий опис процедури A в даному прикладі не потрібно і додано з естетичних міркувань.

Якщо звичайну рекурсію можна уподібнити уробороса *(рис. 3),* то образ складної рекурсії можна почерпнути з відомого дитячого вірша, де «Вовки з переляку, з'їли один одного». Уявіть собі двох вовків які з'їли один одного, і ви зрозумієте складну рекурсію.

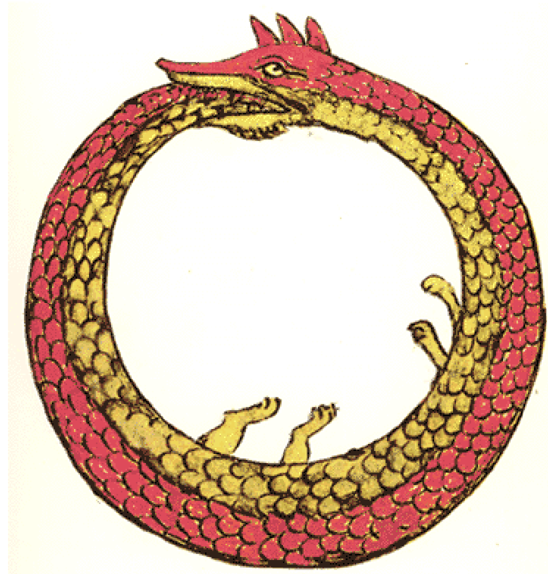

Рис. 3. Уроборос - змій, який пожирає свій хвіст. Малюнок з алхімічного трактату «Synosius» Теодора Пелеканоса (1478г).

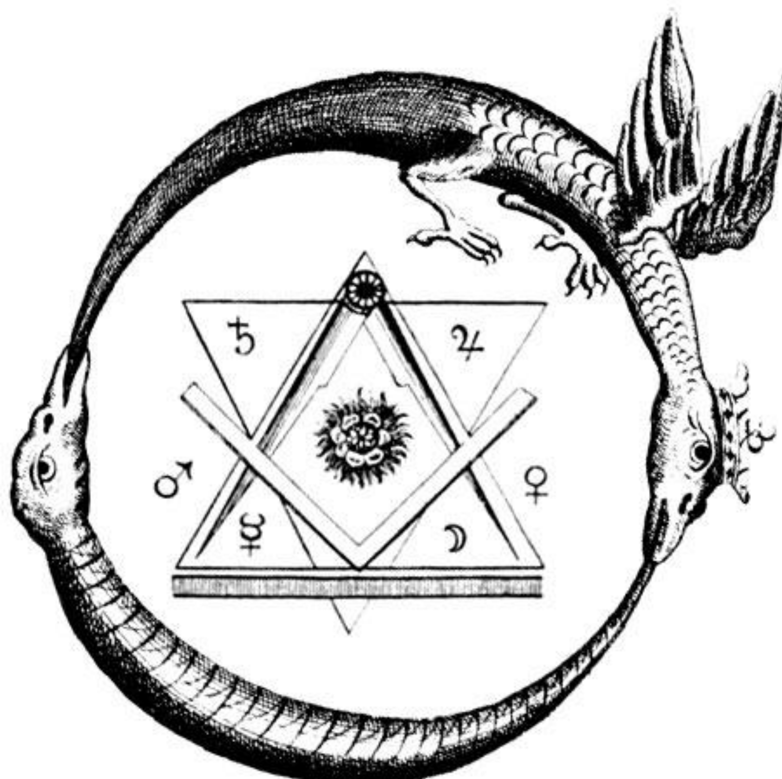

Рис 4. Складна рекурсія.

Імітація роботи циклу з допомогою рекурсії

Якщо процедура викликає сама себе, то, по суті, це призводить до повторного виконання містяться в ній інструкцій, що аналогічно роботі циклу. Деякі мови програмування не містять циклічних конструкцій зовсім, надаючи програмістам організовувати повторення за допомогою рекурсії (наприклад, Пролог[1], де рекурсія - основний прийом програмування).

Для прикладу замінемо роботу циклу for. Для цього нам буде потрібно змінна n, та лічильник кроків, яку можна реалізувати, наприклад, як параметр процедури.

Приклад 1.

```
procedure LoopImitation(i, n: integer);
{Перший параметр – лічильник кроків, другий параметр – загальна кількість кроків}
begin
  writeln('Hello N ', i);  //Тут можуть бути будь-які команди
  if i<n then            //Допоки лічильник не набуде найбільшого значення
    LoopImitation(i+1, n); //значення n, повторює інструкцію шляхом
                       //виклику нового екземпляру процедури
end;
```

Результатом виклику виду $LoopImitation(1, 10)$ стане десятикратне виконання інструкцій зі зміною лічильника від 1 до 10. У даному випадку буде надруковано:

Hello N 1

Hello N 2

…

Hello N 10

Взагалі, не важко помітити, що параметри процедури це межі зміни значень лічильника.

[1] Пролог (фр. Prolog, англ. Prolog) — мова логічного програмування загального призначення, пов'язана зі штучним інтелектом та математичною лінгвістикою

Можна поміняти місцями рекурсивний виклик і вкладене тіло інструкції повторення, як у наступному прикладі.

Приклад 2.

```
procedure LoopImitation2(i, n: integer);
begin
 if i<n then
  LoopImitation2(i+1, n);
 writeln('Hello N ', i);
end;
```

У цьому випадку, перш ніж почнуть виконуватися інструкції, відбудеться рекурсивний виклик процедури. Новий екземпляр процедури також, перш за все, викличе ще один примірник і так далі, поки не дійдемо до максимального значення лічильника. Тільки після цього остання з викликаних процедур виконає свої інструкції, потім виконає свої інструкції передостання і т.д. Результатом виклику $LoopImitation2\ (1, 10)$ буде друк вітань у зворотному порядку:

Hello N 10

…

Hello N 1

Якщо уявити собі ланцюжок з рекурсивно викликаних процедур, то в *прикладі 1* ми проходимо її від раніше викликаних процедур до пізніших. У *прикладі 2* навпаки від більш пізніх до ранніх.

В загальному, рекурсивний виклик можна розташувати між двома блоками інструкцій. *Наприклад:*

```
procedure LoopImitation3(i, n: integer);
begin
 writeln('Hello N ', i); {Може знаходитись перший блок команд}
 if i<n then
  LoopImitation3(i+1, n);
 writeln('Hello N ', i); { Може знаходитись другий блок команд }
end;
```

Тут спочатку послідовно виконаються інструкції з першого блоку потім у зворотному порядку інструкції другого блоку. При виклику $LoopImitation3\ (1,10)$ отримаємо:

Hello N 1

...

Hello N 10

Hello N 10

...

Hello N 1

Потрібно відразу два цикли, щоб зробити те ж саме без рекурсії.

Тим, що виконання частин однієї і тієї ж процедури розділено можна скористатися. Наприклад:

Приклад 3: Перетворення числа в двійкову систему.

Отримання цифр двійкового числа, як відомо, відбувається за допомогою ділення із залишком на основу системи числення 2. Якщо є число x. То його остання цифра в його двійковому поданні дорівнює

$$c_1 = x\ mod\ 2;$$

Взявши ж цілу частину від ділення на 2:

$$x_2 = x\ div\ 2;$$

Поле чергового поділу не отримаємо цілу частину рівну 0. Без рекурсії це буде виглядати так:

```
while x>0 do
begin
  c:=x mod 2;
  x:=x div 2;
  write(c);
end;
```

Проблема тут у тому, що цифри двійкового представлення обчислюються у зворотному порядку (спочатку останні). Щоб надрукувати число в нормальному вигляді доведеться запам'ятати всі цифри в елементах масиву і виводити в окремому циклі.

За допомогою рекурсії неважко домогтися виведення в правильному порядку без масиву і другого циклу. А саме:

```
procedure BinaryRepresentation(x: integer);
var
  c, x: integer;
begin
  {Перший блок. Виконується в порядку виклику процедур }
  c := x mod 2;
  x := x div 2;
  {Рекурсивный вызов}
  if x>0 then
    BinaryRepresentation(x);
  {Другий блок. У зворотному порядку}
  write(c);
end;
```

Взагалі кажучи, ніякого виграшу ми не отримали. Цифри двійкового представлення зберігаються в локальних змінних свої для кожного працюючого примірника рекурсивної процедури. Тобто, пам'ять заощадити не вдалося. Навіть навпаки, витрачаємо зайву пам'ять на зберігання багатьох локальних змінних x. Тим не менш, таке рішення здається мені красивим.

Рекурентні співвідношення. Рекурсія і ітерація

Кажуть, що послідовність векторів $\{\vec{x}_n\}$ задана рекурентним співвідношенням, якщо заданий початковий вектор $\overrightarrow{x_0} = (x_0^1, \dots, x_0^D)$ і функціональна залежність подальшого вектора від попереднього

$$\overrightarrow{x_n} = \overrightarrow{f}(\vec{x}_{n-1}) \quad (1)$$

Простим прикладом величини, що обчислюється за допомогою рекурентних співвідношень, є факторіал

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$$

Черговий факторіал $n!$ можна обчислити за попереднім як:

$$n! = (n-1)! \cdot n \quad (2)$$

Ввівши позначення $x_n = n!$, Отримаємо співвідношення:

$$x_n = x_{n-1} \cdot n, \;\; x_0 = 1 \quad (3)$$

Вектора $\vec{x}_n$ з формули (1) можна інтерпретувати як набори значень змінних. Тоді обчислення необхідного елемента послідовності складатиметься в повторюваному обчисленні їх значень. Зокрема для факторіала:

```
x := 1;
for i := 2 to n do
  x := x * i;
writeln(x);
```

Кожне таке проходження циклу ($x: = x \cdot i$) називається *ітерацією.*

Звернемо, однак, увагу, що співвідношення (1) є чисто рекурсивним визначенням послідовності і обчислення n-го елемента є насправді багаторазове взяття функції f від самої себе:

$$x_n = \underbrace{f(f(\ldots f(x_0)))} \quad (4)$$

Зокрема для факторіала можна написати:

```
function Factorial(n: integer): integer;
begin
  if n > 1 then
    Factorial := n * Factorial(n-1)
  else
    Factorial := 1;
end;
```

Слід розуміти, що виклик функцій тягне за собою деякі додаткові накладні витрати, тому перший варіант обчислення факторіала буде кілька більш швидким. Взагалі ітераційні рішення працюють швидше рекурсивних.

Перш ніж переходити до ситуацій, коли рекурсія корисна, звернемо увагу ще на один приклад, де її використовувати не слід.

Розглянемо окремий випадок рекурентних співвідношень, коли таке значення в послідовності залежить не від одного, а відразу від кількох попередніх значень. Прикладом може служити відома послідовність Фібоначчі, в якій кожен наступний елемент є сума двох попередніх:

$$x_n = x_{n-1} + x_{n-2}, \qquad x_0 = 1, \quad x_1 = 1 \quad (5)$$

При «лобовому» підході можна написати:

```
function Fib(n: integer): integer;
begin
  if n > 1 then
    Fib := Fib(n-1) + Fib(n-2)
  else
    Fib := 1;
end;
```

Кожен виклик Fib створює відразу дві копії себе, кожна з копій - ще дві і т.д. Кількість операцій зростає з номером n експоненціально, хоча при ітераційному вирішенні досить лінійного по n кількості операцій.

Насправді, наведений приклад вчить нас не **КОЛИ** рекурсію не слід використовувати, а тому **ЯК** її не слід використовувати. Зрештою, якщо існує швидке ітераційне (на базі циклів) рішення, то той же цикл можна реалізувати за допомогою рекурсивної процедури або функції. Наприклад:

```
// x1, x2 – початкові значення (1, 1)
// n – номер потрібного числа Фібоначе
function Fib(x1, x2, n: integer): integer;
var
  x3: integer;
begin
  if n > 1 then
  begin
    x3 := x2 + x1;
    x1 := x2;
    x2 := x3;
    Fib := Fib(x1, x2, n-1);
  end else
    Fib := x2;
end;
```

І все ж ітераційні рішення кращі. Питається, коли ж у такому разі, слід користуватися рекурсією?

Будь рекурсивні процедури та функції, що містять всього один рекурсивний виклик самих себе, легко замінюються ітераційними циклами. Щоб отримати щось, що не має простого нерекурсивними аналога, слід звернутися до процедур і функцій, що викликає себе два і більше разів. У цьому випадку безліч викликаються процедур утворює вже не ланцюжок, як на *рис. 1*, а ціле дерево. Існують широкі класи задач, коли обчислювальний процес повинен бути організований саме таким чином. Якраз для них рекурсія буде найбільш простим і природним способом вирішення.

Дерева

Теоретичною базою для рекурсивних функцій, що викликають себе більше одного разу, служить розділ дискретної математики, що вивчає дерева.

1. Основні визначення. Способи зображення дерев

Визначення: *Деревом* будемо називати кінцеве безліч T, що складається з одного або більше вузлів, таких що:
а) Є один спеціальний вузол, званий коренем даного дерева.
б) Інші вузли (виключаючи корінь) містяться в $m \geq 0$ попарно непересічних подмножествах $T_1, T_2, \dots, T_m$, Кожне з яких в свою чергу є деревом. Дерева $T_1, T_2, \dots, T_m$ називаються *піддеревами* даного дерева.

Це визначення є рекурсивним. Якщо коротко, то дерево це безліч, що складається з кореня і приєднаних до нього піддерев, які теж є деревами. Дерево визначається через себе. Однак дане означення визначене, так як рекурсія кінцева. Кожне піддерево містить менше вузлів, ніж містить його дерево. Зрештою, ми приходимо до піддерев, що містить всього один вузол, а це вже зрозуміло, що таке.

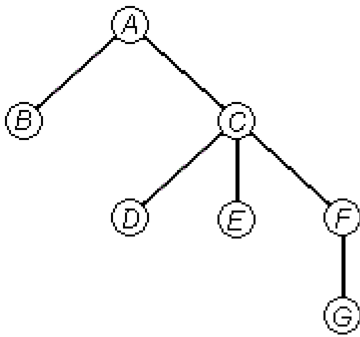

Рис. 5. Дерево.

На *рис. 5* показано дерево з сімома вузлами. Хоча звичайні дерева ростуть знизу вгору, малювати їх прийнято навпаки. При малюванні схеми від руки такий спосіб, очевидно, зручніше. Через даної неузгодженості іноді виникає плутанина, коли говорять про те, що один з вузлів знаходиться над або під іншим. З цієї причини зручніше користуватися термінологією, що вживається при описі генеалогічних дерев, називаючи ближчі до кореня вузли предками, а більш далекі нащадками.

Вузли, що не містять піддерев, *називаються* кінцевими *вузлами або листям.* Безліч не перетинаються дерев називається *лісом.* Наприклад, ліс утворюють піддерева, що виходять з одного вузла.

Графічно дерево можна зобразити і деякими іншими способами. Деякі з них представлені на *рис. 6*. Згідно з визначенням дерево являє собою систему вкладених множин, де ці множини або не перетинаються або повністю утримуються одне в одному. Такі безлічі можна зобразити як області на площині *(рис. 6а).* На *рис. 6б* вкладені безлічі розташовуються не на площині, а витягнуті в одну лінію. *Рис. 6б* також можна розглядати як схему деякої алгебраїчної формули, що містить вкладені дужки. *Рис.6в* дає ще один популярний спосіб зображення деревовидної структури у вигляді вкладеного списку.

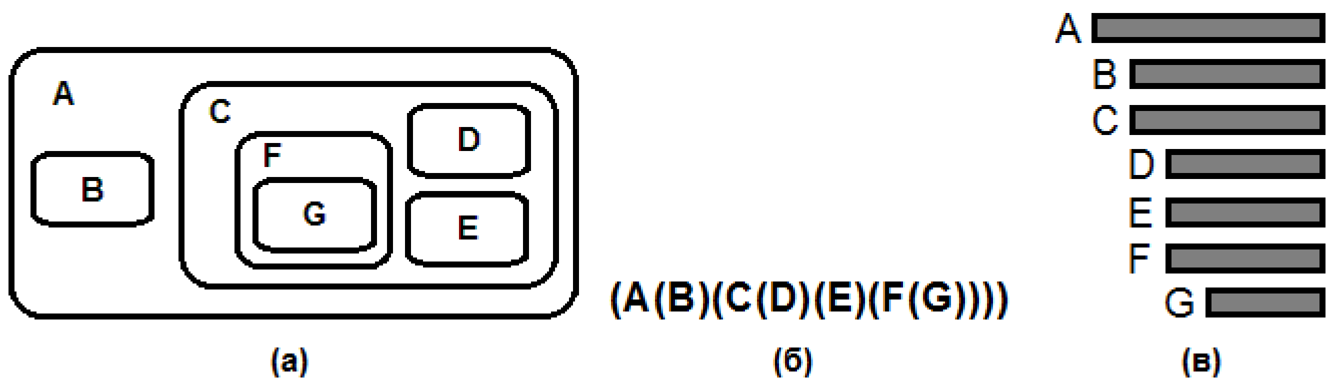

Рис. 4. Інші способи зображення деревовидних структур: (а) вкладені множини; (б) вкладені дужки; (в) уступчастий список.

Уступчастий список має очевидну подібність зі способом форматування програмного коду. Дійсно, програма, написана в рамках парадигми структурного програмування, може бути представлена як дерево, що складається з вкладених один в одного конструкцій.

Також можна провести аналогію між вкладеним списком і зовнішнім виглядом змісту в книгах, де розділи містять підрозділи, ті в свою чергу свої підпункти і т.д. Традиційний спосіб нумерації таких розділів (розділ 1, підрозділи 1.1 і 1.2, підрозділ 1.1.2 і т.п.) називається *десятковою системою Дьюї.* У застосуванні до дерева на *рис. 5 і 6* ця система виглядатиме:

$$1.\,A;\ 1.1\ B;\ 1.2\ C;\ 1.2.1\ D;\ 1.2.2\ E;\ 1.2.3\ F;\ 1.2.3.1\ G;$$

2. *Проходження дерев*

У всіх алгоритмах, пов'язаних з деревовидними структурами незмінно зустрічається одна і та ж ідея, а саме ідея *проходження* або *обходу дерева.* Це - такий спосіб відвідування вузлів дерева, при якому кожен вузол проходиться точно один раз. При цьому виходить лінійна розстановка вузлів дерева. Зокрема існує три способи: можна проходити вузли в прямому, зворотному і кінцевому порядку.

Алгоритм обходу в прямому порядку:

- Потрапити в корінь,
- Пройти всі піддерева зліва на право в прямому порядку.

Даний алгоритм рекурсивний, так як проходження дерева містить проходження піддерев, а вони в свою чергу проходяться за тим же алгоритмом.

Зокрема для дерева на *рис. 5 і 6* прямий обхід дає послідовність вузлів: A, B, C, D, E, G, H.

Одержаний список відповідає послідовному зліва направо перерахуванню вузлів при поданні дерева за допомогою вкладених дужок і в десятковій системі Дьюї, а також проходу зверху вниз при поданні у вигляді вкладеного списку.

При реалізації цього алгоритму на мові програмування попадання в корінь відповідає виконання процедурою або функцією деяких дій, а проходження піддерев - рекурсивним викликам самої себе. Зокрема для бінарного дерева (де з кожного вузла виходить не більше двох піддерев) відповідна процедура буде виглядати так:

```
// Preorder Traversal – англійська назва для прямого проходження
procedure PreorderTraversal({Аргументи});
begin
  //Перегляд корня
  DoSomething({Аргументи});

  //Перегляд лівого піддерева
  if {Існує ліве піддерево} then
    PreorderTransversal({Аргументи 2});

  //Проходження правого піддерева
  if {Існує праворуч піддерево} then
    PreorderTransversal({Аргументи 3});
end;
```

Тобто спочатку процедура виробляє всі дії, а тільки потім відбуваються всі рекурсивні виклики.

Алгоритм обходу у зворотному порядку:

- Пройти ліве піддерево,
- Потрапити в корінь,
- Пройти наступне за лівим поддерево.
- Потрапити в корінь, і т.д поки не буде пройдено крайнє праве піддерево.

Тобто проходяться всі піддерева зліва на право, а повернення в корінь розташовується між цими проходженнями. Для дерева на *рис. 3 і 4* це дає послідовність вузлів: B, A, D, C, E, G, F.

У відповідній рекурсивної процедурі дії будуть розташовуватися в проміжках між рекурсивними викликами. Зокрема для бінарного дерева:

```
// Inorder Traversal – англійська назва для зворотнього проходження дерева
procedure InorderTraversal({Аргументи});
begin
  //Проходження лівого піддерева
  if {Існує ліве піддерева} then
    InorderTraversal({Аргументи 2});
  //Проходженні корня
  DoSomething({Аргументи});
 //Проходження правого піддерева
  if {Існує правопіддерево} then
    InorderTraversal({Аргументи 3});
end;
```

Алгоритм обходу з кінцевих вузлів:

- Пройти всі піддерева зліва на право,
- Потрапити в корінь.

Для дерева на *рис. 5 і 6* це дасть послідовність вузлів: B, D, E, G, F, C, A.

У відповідній рекурсивної процедурі дії будуть розташовуватися після рекурсивних викликів. Зокрема для бінарного дерева:

```
// Postorder Traversal – англійська назва для обходу з кінців вузлів
procedure PostorderTraversal({Аргументи});
begin
  //Проходження лівого піддерева
  if {Існує ліве піддерево} then
    PostorderTraversal({Аргументи 2});

  //Проходження правого піддерева
  if {Існує праве піддерево} then
    PostorderTraversal({Аргументи 3});
```

```
  //Проходження корня
  DoSomething({Аргументы});
 end;
```

3. *Подання дерева в пам'яті комп'ютера*

Якщо деяка інформація розташовується у вузлах дерева, то для її зберігання можна використовувати відповідну динамічну структуру даних. На Паскалі це робиться за допомогою змінної типу запис ($record$), що містить покажчики на піддерева того ж типу. Наприклад, бінарне дерево, де в кожному вузлі міститься ціле число можна зберегти за допомогою змінної типу $PTree$, який описаний нижче:

```
 type
  PTree = ^TTree;
  TTree = record
   Inf: integer;
   LeftSubTree, RightSubTree: PTree;
  end;
```

Кожен вузол має тип $PTree$. Це покажчик, тобто кожен вузол необхідно створювати, викликаючи для нього процедуру New. Якщо вузол є кінцевим, то його полях $LeftSubTree$ і $RightSubTree$ присвоюється значення nil. В іншому випадку вузли $LeftSubTree$ і $RightSubTree$ також створюються процедурою New.

Схематично одна така запис зображена на *рис. 7*.

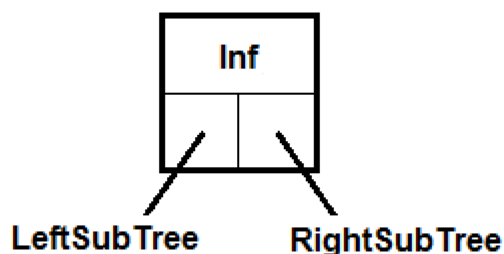

Рис. 7. Схематичне зображення записи типу $TTree$. Запис має три поля: Inf - деяке число, $LeftSubTree$ і $RightSubTree$ - покажчики на записи того ж типу $TTree$.

Приклад дерева, складеного з таких записів, показаний на *рис. 8*.

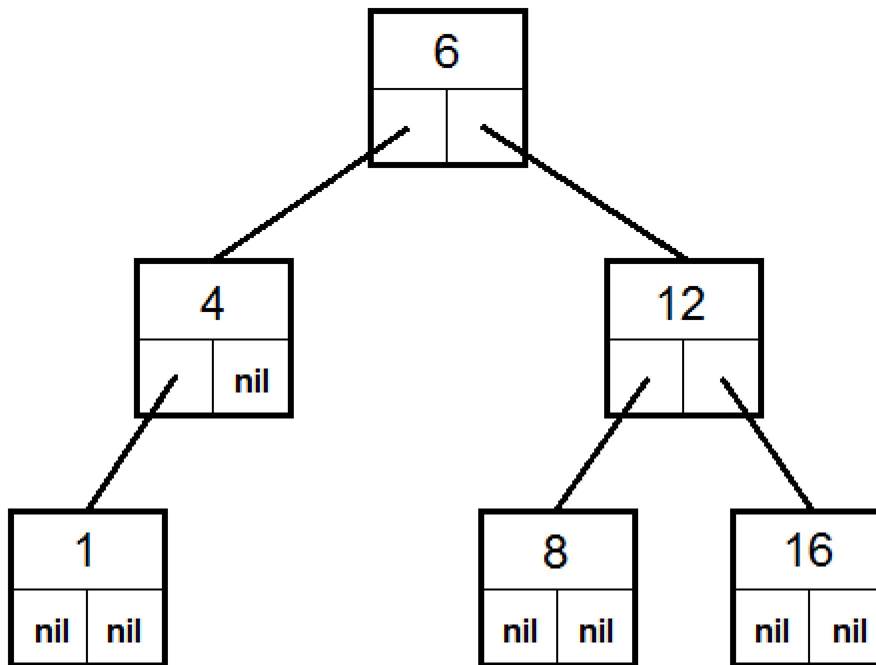

Рис. 8. Дерево, складене з записів типу $TTree$. Кожен запис зберігає число і два покажчика, які можуть містити або $\boldsymbol{nil}$, або адреси інших записів того ж типу.

Якщо ви раніше не працювали зі структурами складаються з записів, які містять посилання на записи того ж типу, то рекомендуємо ознайомитися з матеріалом про рекурсивні структури даних.

Приклади рекурсивних алгоритмів

1. *Ханойські вежі*

Згідно з легендою у Великому храмі міста Бенарас, під собором, що відзначає середину світу, знаходиться бронзовий диск, на якому укріплені 3 алмазних стрижня, висотою в один лікоть і товщиною з бджолу. Давним-давно, на самому початку часів ченці цього монастиря завинили перед богом Брамою. Розгніваний, Брама спорудив три високі стрижня і на один з них помістив 64 диска з чистого золота, причому так, що кожен менший диск лежить на більшому. Як тільки всі 64 диска будуть перекладені зі стрижня, на який Бог Брама склав їх при створенні світу, на інший стрижень, вежа разом з храмом звернуться до пил і під громові гуркіт загине світ. У процесі потрібно, щоб більший диск жодного разу не опинявся над

меншим. Ченці в скруті, в якій же послідовності варто робити перекладання? Потрібно забезпечити їх софтом для розрахунку цієї послідовності.

Незалежно від Брами дану головоломку в кінці 19 століття запропонував французький математик Едуард Люка. У продаваемом варіанті зазвичай використовувалося 7-8 дисків *(рис. 8)*.

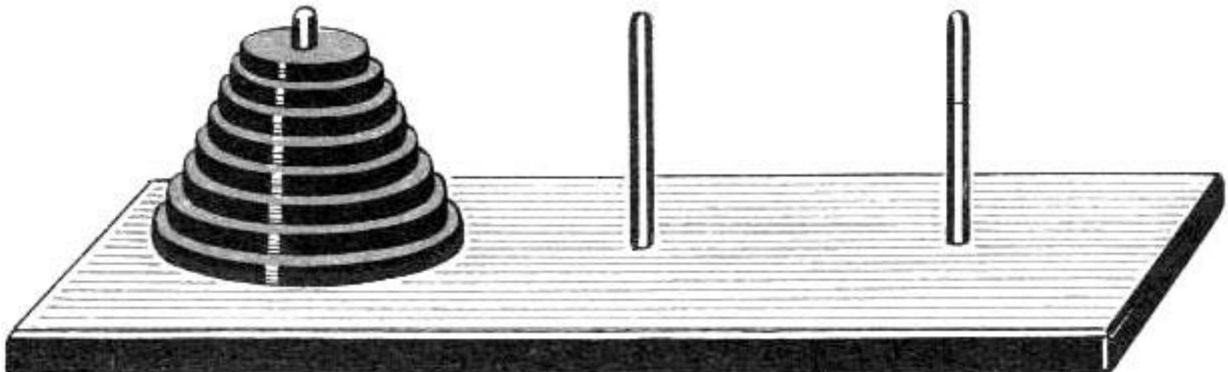

Рис. 8. Головоломка «Ханойські вежі».

Припустимо, що існує рішення для $n - 1$ диска. Тоді для перекладання n дисків треба діяти наступним чином:

1) Перекладаємо $n\ - 1$ диск.

2) Перекладаємо n -й диск на що залишився вільним штир.

3) Перекладаємо стопку з $n\ - 1$ диска, отриману в пункті (1) поверх n -го диска.

Оскільки для випадку $n = 1$ алгоритм перекладання очевидний, то по індукції за допомогою виконання дій (1) - (3) можемо перекласти довільну кількість дисків.

Створимо рекурсивну процедуру, що друкує всю послідовність перекладань для заданої кількості дисків. Така процедура при кожному своєму виклик повинна друкувати інформацію про один перекладанні (з пункту 2 алгоритму). Для перекладань з пунктів (1) і (3) процедура викличе сама себе зі зменшеним на одиницю кількістю дисків.

```
//n – кількість дисків
//a, b, c – номер вежі. Перекладання відбувається з вежі a на вежу b при допомозі вежі c,
```

```
procedure Hanoi(n, a, b, c: integer);
begin
 if n > 1 then
 begin
  Hanoi(n-1, a, c, b);
  writeln(a, ' -> ', b);
  Hanoi(n-1, c, b, a);
 end else
  writeln(a, ' -> ', b);
 end;
```

Зауважимо, що безліч рекурсивно викликаних процедур в даному випадку утворює дерево, прохідне в зворотному порядку.

2. *Синтаксичний аналіз арифметичних виразів*

Завдання синтаксичного аналізу полягає в тому, щоб за наявною рядку, що містить арифметичний вираз, і відомим значенням, що входять до неї змінних, обчислити значення виразу.

Процес обчислення арифметичних виразів можна представити у вигляді бінарного дерева. Дійсно, кожен з арифметичних операторів $(+, -, *, /)$ вимагає двох операндів, які також будуть і арифметичними виразами і, відповідно можуть розглядатися як піддерева. *Рис. 9* показує приклад дерева, відповідного висловом:

$$x - 2 \cdot \left(\frac{1}{x} + \frac{x}{3}\right) \qquad (6)$$

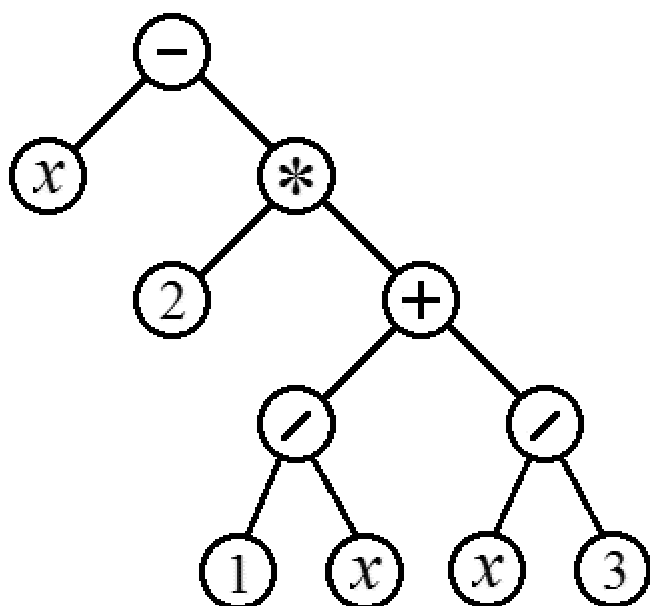

Рис. 9. Синтаксичне дерево, відповідне арифметичному виразженню (6).

У такому дереві кінцевими вузлами завжди будуть змінні (тут *x) або числові константи, а всі внутрішні вузли будуть містити арифметичні оператори.* Щоб виконати оператор, треба спочатку обчислити його операнди. Таким чином, дерево на малюнку слід обходити з кінця. Відповідна послідовність вузлів

$$x\ 2\ 1\ x\ |\ x\ \ 3\ |+*\ - \qquad (7)$$

називається *зворотної польської записом* арифметичного виразу.

При побудові синтаксичного дерева слід звернути увагу на таку особливість. Якщо є, наприклад, вираз

$$a - b + c \qquad (8)$$

і операції додавання і віднімання ми будемо зчитувати зліва на право, то правильне синтаксичне дерево буде містити мінус замість плюса *(рис. 10а)*. По суті, це дерево відповідає виразу $a - (b - c)$. Полегшити складання дерева можна, якщо аналізувати вираз (8) навпаки, справа наліво. У цьому випадку виходить дерево з *рис. 10б*, еквівалентне дереву 8а, але не вимагає заміни знаків.

Аналогічно справа наліво потрібно аналізувати вирази, що містять оператори множення і ділення.

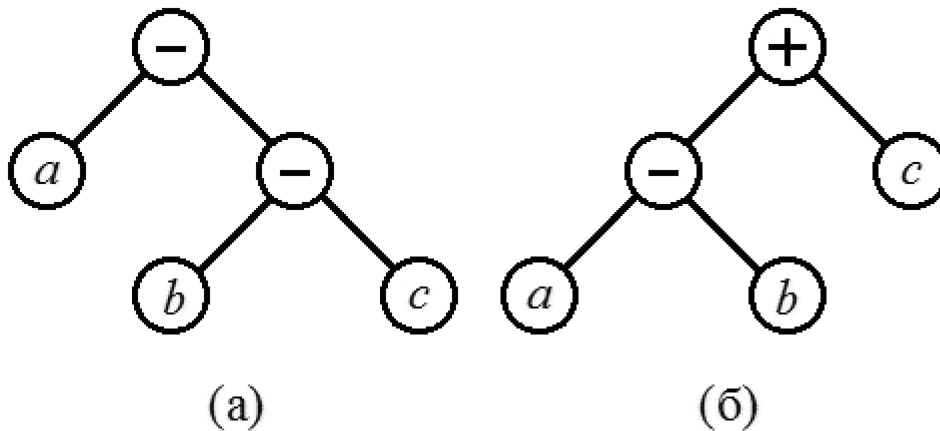

Рис. 10. Синтаксичні дерева для вираження a - b + c при читанні зліва направо (а) і справа наліво (б).

Результатом аналізу рядка має бути послідовність вузлів дерева в кінцевому порядку. Кожний вузол має зберігати інформацію про підвузли і про операції, які в ньому відбувається. Наприклад, вузли можна реалізувати у вигляді записів, одне з полів який має процедурний тип. Інший варіант - кожен вузол це об'єкт, де операція реалізована як віртуальний метод.

3. *Швидкі сортування*

Прості методи сортування начебто методу вибору або методу бульбашки сортують масив з n елементів за $O\ (n^2)$ операцій. Однак за допомогою принципу «розділяй і володарюй» вдається побудувати більш швидкі, що працюють за $O\ (n\ log_2 n)$ *алгоритми.* Суть цього принципу в тому, що рішення виходить шляхом рекурсивного поділу задачі на трохи простіші того ж типу до тих пір, поки вони не стануть елементарними. Наведемо як приклади кілька швидких алгоритмів такого роду.

Алгоритм 1: «Швидка» сортування (quicksort).

1. Вибирається опорний елемент (наприклад, перший або випадковий).

2. Реорганізовуючи масив так, щоб спочатку йшли елементи менші опорного, потім рівні йому, потім великі. Для цього досить пам'ятати, скільки було знайдено менших (m_1) і великих (m_2), ніж опорний і ставити черговий

елемент на місце з індексом (m_1), а черговий більший на місце з індексом $n - 1 - (m_2)$.

Після виконання такої операції опорний елемент і рівні йому стоять на своєму місці, їх переставляти більше не доведеться. Між «менший» і «більшої» частина масиву перестановок також бути не може. Тобто ці частини можна сортувати незалежно один від одного.

3. Якщо "менша» або «велика» частина складається з одного елемента, то вона вже відсортована і робити нічого не треба. Інакше сортуємо ці частини за допомогою алгоритму швидкого сортування (тобто, виконуємо для неї кроки 1-3).

Як бачите, швидке сортування складається з виконання кроків 1 і 2 та рекурсивного виклику алгоритму для одержані частин масиву.

Алгоритм 2: Сортування злиттям (merge sort).

Ділимо масив на дві частини приблизно однакового розміру і, якщо вийде половина масиву містить більше одного елемента, то сортуємо її за допомогою сортування злиттям. Як бачите, цей пункт містить рекурсивне звернення до всього алгоритму в цілому.

З'єднує дві відсортовані половини так, щоб вийшов одна відсортований масив. Для цього поміщаємо у допоміжний масив елементи з першої половини, поки вони не перевершують чергового елементу з другої половини. Потім починаємо поміщати туди елементи другої половини, поки вони не перевершують чергового елементу з першої половини. Потім знову беремо елементи першої половини і т.д. Ця операція називається злиттям і вимагає стільки кроків, скільки елементів у обох з'єднуються масивах.

Алгоритм 3: Сортування деревом (tree sort).

Перш ніж переходити до пояснення суті алгоритму введемо одне поняття. *Двійкового дерева пошуку* називається бінарне дерево, у вузлах якого розташовуються числа таким чином, що в лівому піддереві кожного вузла знаходяться числа менші, ніж в цьому вузлі, а в правому піддереві більше або

рівні того, що в цьому вузлі. На *рис. 11* показано два приклади дерев пошуку, складених з одних і тих же чисел.

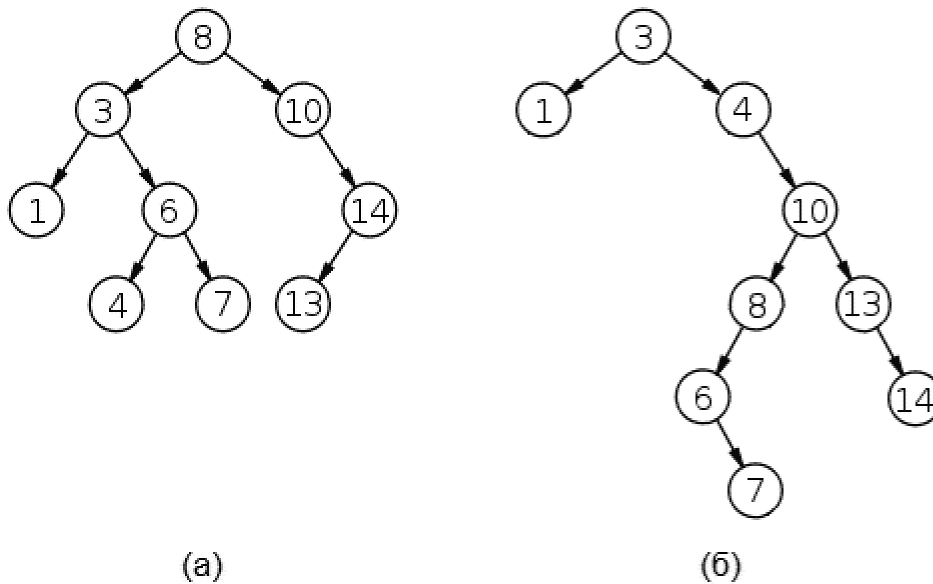

Рис. 11. Двійкові дерева пошуку, складені з чисел 1, 3, 4, 6, 7, 8, 10, 13, 14.

Якщо для кожної вершини висота піддерев розрізняється не більше ніж на одиницю, то дерево називається збалансованим. Збалансовані дерева пошуку також називаються *АВЛ-деревами* (за першими літерами прізвищ винахідників Г. М. Адельсона-Бєльського і Е. М. Ландіса). Як видно на *рис. 11а* показано збалансоване дерево, на *рис. 11б* незбалансоване.

Зауважимо, що розташування чисел за збільшенням вийде, якщо обходити ці дерева в зворотному прядки.

Сортування деревом вийде, якщо ми спочатку послідовно будемо додавати числа з масиву в двійкове дерево пошуку, а потім обійдемо його у зворотному порядку.

Якщо дерево буде близько до збалансованого, то сортування потребують приблизно $(n\, log_2 n)$ операцій. Якщо не пощастить і дерево виявиться максимально незбалансованим, то сортування займе n^2 операцій.

4. Довільна кількість вкладених циклів

Розмістивши рекурсивні виклики всередині циклу, по суті, отримаємо вкладені цикли, де рівень вкладеності дорівнює глибині рекурсії.

Для прикладу напишемо процедуру, що друкує всі можливі поєднання з k чисел від 1 до n (C_n^k).Числа, що входять в кожне поєднання, будемо друкувати в порядку зростання. Сполучення з двох чисел ($k = 2$) друкуються так:

```
for i1 := 1 to n do
  for i2 := i1 + 1 to n do
    writeln(i1, ' ', i2);
```

Сполучення з трьох чисел (k = 3) так:

```
for i1 := 1 to n do
  for i2 := i1 + 1 to n do
    for i3 := i2 + 1 to n do
      writeln(i1, ' ', i2, ' ', i3);
```

Однак, якщо кількість чисел у поєднанні задається змінною, то доведеться вдатися до рекурсії.

```
procedure Combinations(
  n, k: integer;
  //Масив у якому будуть утворюватися комбінації
  var Indexes: array of integer;
  //Лічильник глибини рекурсії
  d: integer);
var
  i, i_min: integer;
  s: string;
begin
  if d < k then
  begin
    if d = 0 then
      i_min := 1
    else
      i_min := Indexes[d-1] + 1;
    for i := i_min to n do
    begin
      Indexes[d] := i;
```

```
    Combinations(n, k, Indexes, d+1);
   end;
  end
  else
  begin
   for i := 0 to k-1 do
    write(Indexes[i], ' ');
   writeln;
  end;
 end;
```

5. *Задачі на графах*

Графом називають графічне зображення, що складається з *вершин* (вузлів) і з'єднують деякі пари вершин *ребер (рис. 12а)*.

Більш строго: граф - сукупність безлічі вершин і безлічі ребер. Безліч ребер - підмножина евклідового квадрата безлічі вершин (тобто ребро з'єднує рівно дві вершини).

Ребрах можна також присвоїти напрямок. Граф в цьому випадку називається *орієнтованим (рис. 12б)*.

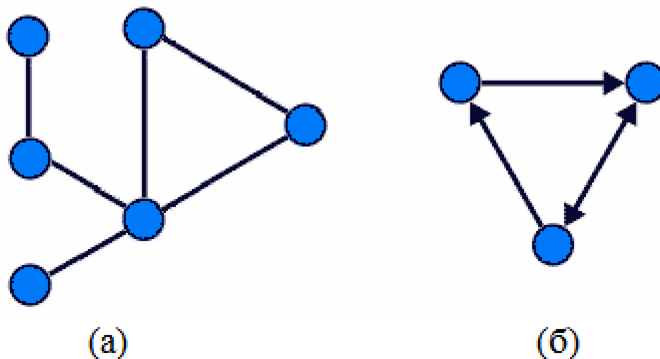

Рис.12. (а) Граф. (б) Орієнтований граф.

Теорія графів знаходить застосування в самих різних областях. Кілька прикладів:

- Логістика та транспортні системи. Вершинами будуть склади з товарами або пункти призначення, а ребра - дороги, їх з'єднують.

- Маршрутизація мереж. Вершини - комп'ютери, з'єднані в мережу, ребра - зв'язки між ними. Вирішується завдання про шляхи передачі даних з одного комп'ютера на інший.
- Комп'ютерна хімія. Моделі у вигляді графів використовуються для опису шляхів протікання складних реакцій. Вершини - беруть участь в реакціях речовини, ребра - шляхи перетворень речовин. Також графом є зображення структур молекул: вершини - атоми, ребра - хімічні зв'язки.
- Електричні сітки.
- Сайти в Інтернеті можна вважати вузлами орієнтованого графа, ребрами якого будуть гіперпосилання. І т.д.

Сучасна теорія графів являє собою потужну формальну систему, що має неозоре безліч застосувань.

Шляхом або *ланцюгом* в графі називається послідовність вершин, в якій кожна вершина з'єднана ребром з наступною. Шляху, в яких початкова та кінцева вершина збігаються, називають циклами. Якщо для кожної пари вершин існує шлях їх з'єднують, то такий граф називають зв'язковим.

У програмуванні використовуються три способи зберігання в пам'яті інформації про структуру графів.

1) *Матриці суміжності*

Квадратна матриця М, де як рядки, так і стовпці відповідають вершинам графа. Якщо вершини з номерами і та j з'єднані ребром, то $M_{ij} = 1$, інакше $M_{ij} = 0$. Для неорієнтованого графа матриця, очевидно, симетрична. Орієнтований граф задається антисиметричною матрицею. Якщо ребро виходить з вузла *i* і приходить у вузол *j, то* $M_{ij} = 1$, а симетричний елемент $M_{ji} = -1$.

2) *Матриця інцидентності*

Стовпці матриці відповідають вершинам, а рядки ребрах. Якщо ребро з номером *i* з'єднує вершини з номерами *j* і *k, то елементи матриці* $I_{ij} = I_{ik} = 1$. Інші елементи *i* -й рядки рівні 0.

3) *Список ребер*

Просто набір пар номерів вершин, з'єднаних ребрами.

Розглянуті вище дерева є окремим випадком графів. Деревом буде будь зв'язний граф, що не містить циклів.

Завдання, що виникають в теорії графів численні і різноманітні. Про них пишуться товсті книги, і немає ніякої можливості скільки-небудь повно їх тут оглянути. Тому ми обмежимося зауваженням, що багато з цих завдань вимагають систематичного перебору вершин. Якщо перебирати вершини, пов'язані ребрами і при цьому відвідувати кожну вершину тільки один раз, то безліч відвідуваних алгоритмом вершин буде утворювати дерево, а сам алгоритм природно зробити рекурсивним.

Наприклад, класичною задачею є пошук шляху з однієї вершини в іншу. Алгоритм пошуку повинен буде побудувати дерево можливих шляхів з початкової вершини, кінцевими вузлами якого будуть вершини, з яких не можна потрапити ні в яку вершину, що не належить раніше побудованої гілки (Не позначену як уже відвідану). Завдання буде вирішено, коли один з кінцевих вузлів співпаде з кінцевою вершиною, шлях в яку потрібно знайти.

6. *Фрактали*

Фракталами називають геометричні фігури, що володіють властивістю успадкування, тобто складаються з частин, подібних до всієї фігури.

Класичним прикладом є крива Коха, побудова якої показано на *рис. 13.* Спочатку береться відрізок прямої *(рис. 13а).* Він ділиться на три частини, середня частина вилучається і замість неї будується кут *(рис. 13б),* сторони якого рівні довжині вилученого відрізка (тобто 1/3 від довжини вихідного відрізка). Така операція повторюється з кожною з одержані 4-х відрізків *(рис. 13в).* І так далі *(рис. 13г).* Крива Коха виходить після нескінченного числа таких ітерацій. На практиці побудова можна припинити, коли розмір деталей виявиться менше дозволу екрану *(рис. 13д).*

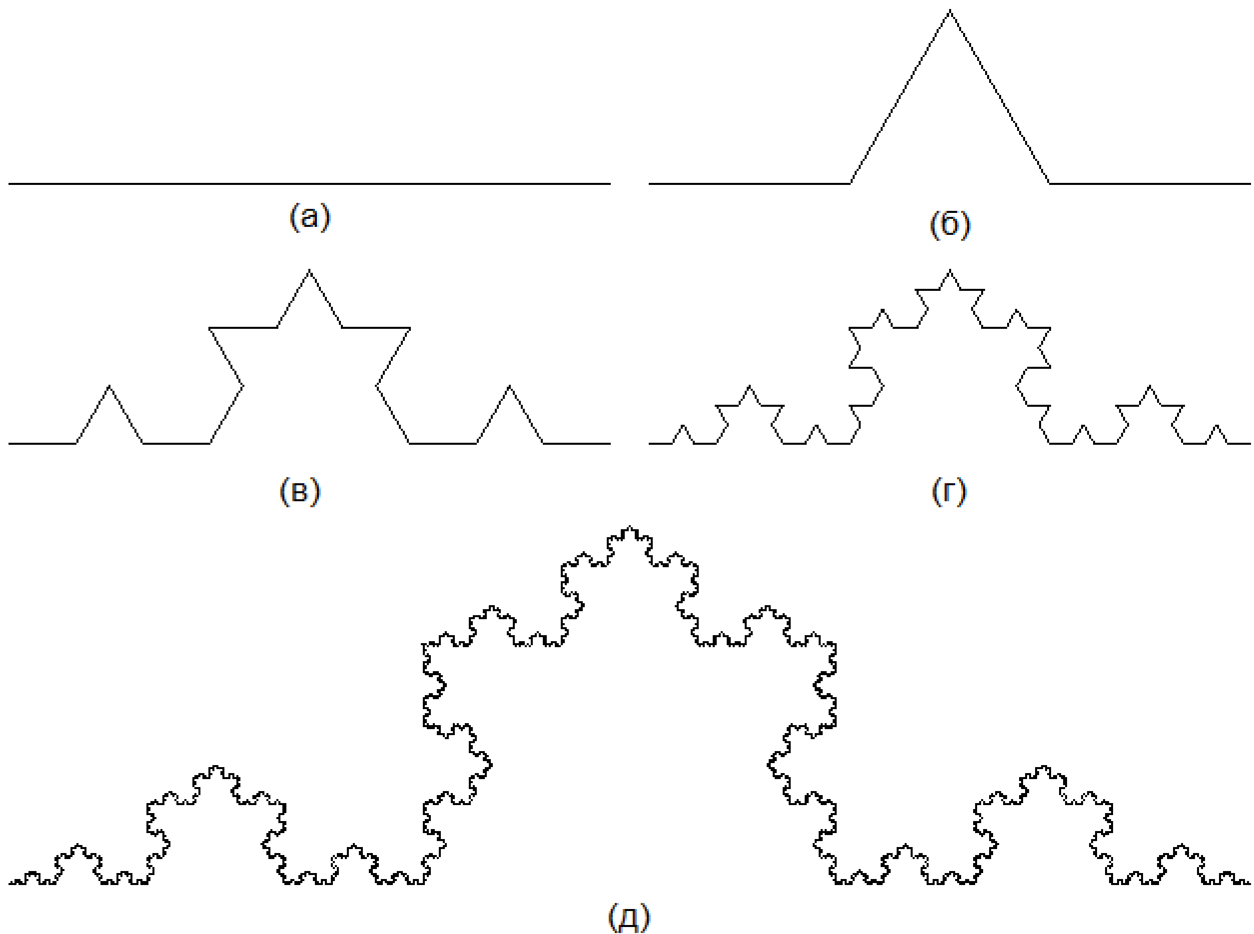

Рис. 13. Процес побудови кривої Коха.

Ще одним прикладом може служити деревце на рис. 6. Воно також містить частини, подібні всьому дереву в цілому, що робить його фракталом.

Фрактали, по суті, рекурсивні структури і їх побудова природно проводити за допомогою рекурсивних процедур.

Позбавлення від рекурсії

Будь рекурсивний алгоритм може бути переписаний без використання рекурсії. Зауважимо, що швидкодія алгоритмів при позбавленні від рекурсії, як правило, підвищується. Ще однією причиною щоб позбутися від рекурсії є обмеження на обсяг збережених програмою локальних змінних і значень параметрів одночасно виконуються процедур. При дуже глибокою рекурсії цей обсяг зростає, і програма перестає працювати, видаючи помилку «Stack overflow» (переповнення стека).

Так чому ж люди продовжують користуватися рекурсивними алгоритмами? Очевидно, тому що це простіше і природніше, ніж відповідні нерекурсівние рішення. Тим не менш, знання про способи обійтися без рекурсії необхідно.

Нижче представлено кілька варіантів того, як це можна зробити.

1. ***Явне використання стека***

Стеком називається структура даних, в якій додавання та вилучення даних відбувається з одного кінця, званого вершиною стека *(рис. 14).* Наочним чином стека може служити стос тарілок - додавати або забрати тарілки можна тільки зверху. Кожна тарілка відповідає елементу даних.

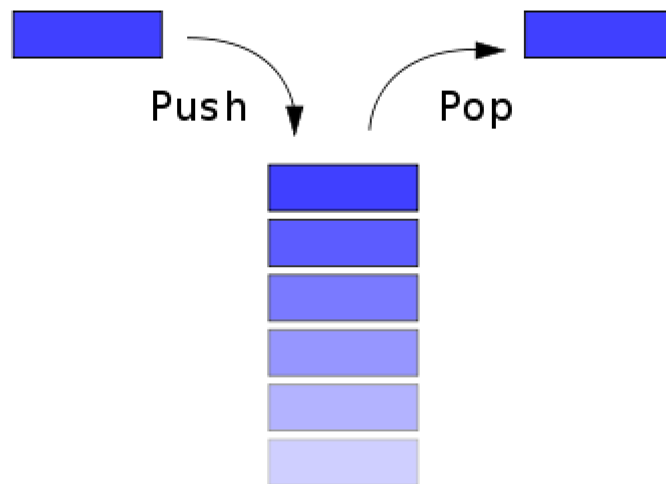

Рис.14. Наочне уявлення стека. Push (проштовхування) - традиційна назва для операції додавання даних в стек, Pop (виштовхування) - традиційна назва для операції вилучення даних з стека.

Коли одна процедура або функція викликає іншу, то параметри першої процедури, а також місце, з якого її виконання має продовжитися після того як відпрацює викликана процедура (точка повернення), запам'ятовуються в так званому *стеку* викликів. Якщо викликана процедура в свою чергу чогось викликає, то її параметри і точка повернення також додаються в стек.

При рекурсивних викликах стек викликів зберігає ланцюжок з даних про одночасно працюючі процедури. У всіх просунутих середовищах розробки цей ланцюжок разом з параметрами процедур можна переглянути під час налагодження. Відповідна команда зазвичай називається "Call Stack".

Універсальний спосіб позбутися рекурсії - це самостійно запрограмувати ті дії зі стеком, які фактично відбуваються, коли ви використовуєте рекурсивні виклики. Покажемо, як це можна зробити, на прикладі процедури яка викликає себе два рази.

Для початку реалізуємо у вигляді класу стек, який зберігає параметри процедури:

```
type
  //Запис для збереження параметрів процедури
  Parameters = record
    //Список параметрів
  end;

  //Стек зручно представити у вигляді зв’язаних списків
  PList = ^List;
  List = record
    Data: Parameters;
    Next: PList;
  end;

  //Опис однонаправленого списка пов’язаного з методами
  //додаванння та видалення елементів, отримаємо стек.
  Stack = class
  private
    StackTop: PList;
  public
    //Внесення даних
    procedure Push(NewData: Parameters);
    //Видобуток даних
    function Pop: Parameters;
    //Перевірка існування даних
    function Empty: boolean;
  end;

implementation
```

```
//Додавання нових даних
procedure Stack.Push(NewData: Parameters);
var
  NewElement: PList;
begin
  New(NewElement);
  NewElement^.Data := NewData;
  NewElement^.Next := StackTop;
  StackTop := NewElement;
end;

//Видобуток даних
function Stack.Pop: Parameters;
var
  PopedElement: PList;
begin
  PopedElement := StackTop;
  StackTop := StackTop^.Next;
  Pop := PopedElement^.Data;
  Dispose(PopedElement);
end;

//Перевірка існування даних
function Stack.Empty: boolean;
begin
  Empty := StackTop = nil;
type
  //Записи для збереження параметрів процедури
  Parameters = record
    //Список параметрів
  end;
  //Стек зручно представити у вигляді зв’язаних списків
  PList = ^List;
  List = record
    Data: Parameters;
    Next: PList;
```

```
  end;

  Stack = class
  private
    StackTop: PList;
  public
    //Додавання даних
    procedure Push(NewData: Parameters);
    //Видобуток даних
    function Pop: Parameters;
    //Перевірка наявності даних
    function Empty: boolean;
  end;

implementation

// Додавання даних
procedure Stack.Push(NewData: Parameters);
var
  NewElement: PList;
begin
  New(NewElement);
  NewElement^.Data := NewData;
  NewElement^.Next := StackTop;
  StackTop := NewElement;
end;

// Видобуток даних
function Stack.Pop: Parameters;
var
  PopedElement: PList;
begin
  PopedElement := StackTop;
  StackTop := StackTop^.Next;
  Pop := PopedElement^.Data;
  Dispose(PopedElement);
```

```
end;

// Перевірка наявності даних
function Stack.Empty: boolean;
begin
  Empty := StackTop = nil;
end;
```

Розглянемо узагальнену рекурсивну процедуру з двома викликами самої себе.

```
procedure Recurs(P1: Parameters);
begin
  DoSomething(P1);
  if <умова> then
  begin
    P2 := F(P1);
    Recurs(P2);
    P3 := G(P1);
    Recurs(P3);
  end;
end;
```

В цій процедурі деякі дії ($DoSomething$) виконуються багато разів при різних значеннях параметрів. Нерекурсивний аналог повинен зберігати ці параметри в стек. Кожен рекурсивний виклик буде відповідати додаванню чергових параметрів в стек. Замість рекурсії з'являється цикл, який виконується, поки в стеку є необроблені параметри.

```
procedure NonRecurs(P1: Parameters);
var
  S: Stack;
  P: Parameters;
begin
  S := Stack.Create;
  S.Push(P1);
  while not S.Empty do
  begin
```

```
    P1 := S.Pop;
    DoSomething(P1);
    if <умова> then
    begin
      P3 := G(P1);
      S.Push(P3);
      P2 := F(P1);
      S.Push(P2);
    end;
  end;
end;
```

Зверніть увагу, що рекурсивні виклики йшли спочатку для параметрів $P2$, потім для $P3$. У нерекурсивною процедурі в стек відправляються спочатку параметри $P3$, а тільки потім $P2$. Це пов'язано з тим, що при рекурсивних викликах в стек, по суті, відправляється Недовиконання частина процедури, яка в нашому випадку містить виклик $Recurs\ (P3)$.

Згаданої вище перестановки можна уникнути, якщо замість стека використовувати *чергу* - структуру даних, де додавання і витяг елементів відбувається з різних кінців. Це буде деяким відступом від точної імітації процесів при рекурсивних викликах. Проте в даному прикладі це здається зручнішим: кожен рекурсивний виклик буде прямо замінюватися додаванням параметрів в чергу.

2. Запам'ятовування послідовності рекурсивних викликів

Як говорилося вище, рекурсивні виклики утворюють дерево, де кожен вузол відповідає викликом однієї процедури. Послідовність виконання цих процедур відповідає тому або іншому алгоритму обходу вузлів. Якщо потрібно багато разів обійти вузли одного і того ж дерева, то можна один раз обійти їх рекурсивно, запам'ятати кількість і послідовність вузлів, а потім, користуючись цією інформацією, обходити вузли вже нерекурсивними.

Наприклад, задача обчислення арифметичних виразів, заданих рядком. Може виникнути ситуація, коли один і той же вираз потрібно обчислити багато разів при різних значеннях змінної x. Синтаксичне дерево, яке потрібно

обходити при таких обчисленнях, не залежить від x. Можна обійти його один раз, побудувавши при цьому масив, де кожен елемент буде відповідати вузлу дерева, а їх послідовність - порядку обходу. Повторні обчислення при новому *x* зажадають тільки нерекурсивними перебору елементів масиву.

Такий підхід не позбавляє нас від рекурсії повністю. Однак він дозволяє обмежитися тільки одним зверненням до рекурсивної процедурою, що може бути достатньо, якщо мотивом є турбота про максимальної продуктивності.

3. Визначення вузла дерева за його номером

Ідея даного підходу в тому, щоб замінити рекурсивні виклики простим циклом, що виконається стільки разів, скільки вузлів в дереві, утвореному рекурсивними процедурами. Що саме буде робитися на кожному кроці, слід визначити за номером кроку. Зіставити номер кроку і необхідні дії - завдання не тривіальна і в кожному випадку її доведеться вирішувати окремо.

Наприклад, нехай потрібно виконати *k* вкладених циклів по *n* кроків в кожному:

```
for i1 := 0 to n-1 do
  for i2 := 0 to n-1 do
    for i3 := 0 to n-1 do
      ...
```

Якщо *k* заздалегідь невідомо, то написати їх явним чином, як показано вище неможливо. Використовуючи прийом, роботи з циклами можна отримати необхідну кількість вкладених циклів за допомогою рекурсивної процедури:

```
procedure NestedCycles(Indexes: array of integer; n, k, depth: integer);
var
  i: integer;
begin
  if depth <= k then
    for i:=0 to n-1 do
    begin
      Indexes[depth] := i;
      NestedCycles(Indexes, n, k, depth + 1);
    end
```

```
  else
    DoSomething(Indexes);
end;
```

Щоб позбутися рекурсії і звести все до одному циклу, звернемо увагу, що якщо нумерувати кроки в системі числення з основою n, то кожен крок має номер, що складається з цифр $i1, i2, i3, \ldots$ або відповідних значень з масиву Indexes. Тобто цифри відповідають значенням лічильників циклів. Номер кроку в звичайній десятковій системі числення:

Всього кроків буде $n\ k$. Перебравши їх номери в десятковій системі числення і перевівши кожен з них в систему з основою n, отримаємо значення індексів:

```
M := round(IntPower(n, k));
for i := 0 to M-1 do
begin
  Number := i;
  for p := 0 to k-1 do
  begin
    Indexes[k – p] := Number mod n;
    Number := Number div n;
  end;
  DoSomething(Indexes);
end;
```

Рекурсивні структури даних

Нехай описаний тип-запис, і одним з полів цього запису є покажчик. У цей покажчик можна записати адресу, за якою дана запис розташовується, або, що більш цікаво адресу іншої записи того ж типу. Це дозволяє за допомогою покажчиків створити структуру даних, звану пов'язаним списком.

Приклад опису такої структури даних:

```
  type
    PListElement = ^TListElement;
```

```
TListElement = record
  <Произвольные поля записи>
  NextElement: PListElement;
end;
```

Тут спочатку описується тип - покажчик $PListElement$ на запис типу $TListElement$. Потім йде опис самого типу запису, одне з полів якої має тип $PListElement$. Виходить, що ми використовуємо ідентифікатор $TListElement$ ще до того, як його описали. Зазвичай такі речі заборонено робити, але в даному випадку, коли описується контрольний тип, з цього правила зроблено виняток.

Якщо є запис цього типу, то можна з допомогою покажчика динамічно створити ще один елемент цього типу. По покажчику в цьому новому елементі ще один елемент і т.д., скільки буде потрібно. Така структура даних називається односпрямованим пов'язаним списком. Її схематичне зображення показано на *рис. 15.*

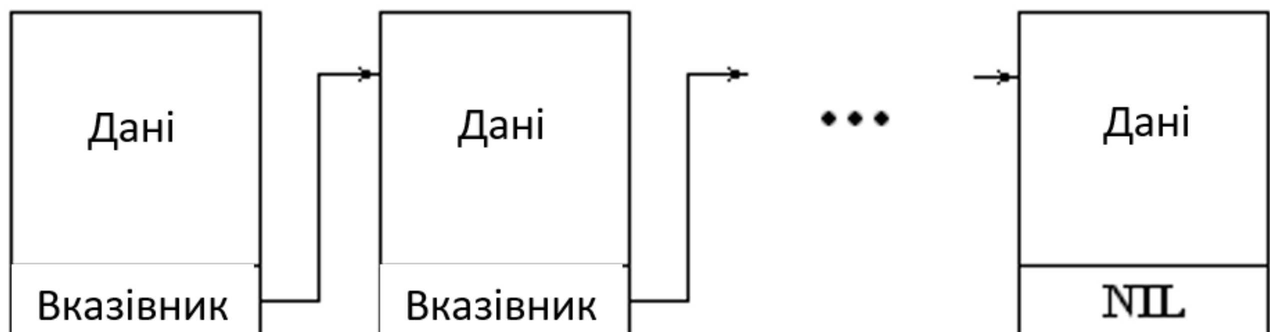

Рис. 15. Зв'язаний список.

Приклад: Нехай у вигляді односпрямованого пов'язаного списку зберігаються цілі числа. Створити процедури, які:

а) Створюють список, динамічно виділяючи з купи пам'ять під задане число елементів $\boldsymbol{N}$.

б) Заповнюють інформаційну частину елементів списку числами $\mathbf{1}, \mathbf{2}, \ldots, \boldsymbol{N}$.

в) Друкують в стовпчик інформаційну частину списку.

г) Знищують список, повертаючи в купу виділену під нього пам'ять.

Рішення:

```
type
  PList = ^TList;
  TList = record
    x: integer; {Інформаційне поле}
    Next: PList; {Посилання на наступний елемент}
  end;

procedure CreateList(var Head: PList; Num: integer);
{Процедура створює список з Len елементів,
Починаючи з Head}
var
  n: integer;
  H: PList;
begin
  if Num > 1 then
  begin
    New(Head);
    H := Head;
    {При роботі з одно направленим списком головне не забути посилання на
перший елемент. Тому надалі будемо працювати не з Head, а з допоміжною
змінною H}
    for n := 2 to Num do
    begin
      New(H^.Next);
      H := H^.Next;
    end;
    H^.Next := nil;
  end else
    Head := nil;
end;

procedure FreeList(var Head: PList);
{Процедура яка звільнює пам'ять, виділену для списку з Head}
var
  H: PList;
begin
  {Список знищується тому можемо змінити і елемент Head}
```

```
  while  Head <> nil  do
  begin
    H := Head;
    Head := Head^.Next;
    Dispose(H);
  end;
end;

procedure  FillList(Head: PList);
{Процедура заповнює інформаційну частину списка x числами 1, 2, 3, ...}
var
  n: integer;
begin
  n:=1;
  while  Head <> nil  do
  begin
    Head^.x := n;
    n := n+1;
    Head:=Head^.Next;
  end;
end;

procedure  PrintList(Head: PList);
{Процедура друкує інформаційну частину списка}
begin
  while  Head <> nil  do
  begin
    writeln(Head^.x);
    Head := Head^.Next;
  end;
end;
```

Програма, що працює з цими процедурами, може виглядати, наприклад, так:

```
var
  List: PList;
begin
```

```
  CreateList(List, 5); {Створюємо список з 5 елементів}
  FillList(List); {Заносимо до інформаційної частини 1, 2, 3, 4, 5}
  PrintList(List); {Друкуємо список інформаційної частини}
  ClearList(List); {Звільнюємо пам’ять}
  readln;
 end.
```

Пов'язані списки також як і масиви дозволяють зберігати набір однотипних даних і можуть розглядатися як альтернатива масивам. Яку структуру віддати перевагу, масиви або зв'язані списки, залежить від тих операцій, які найчастіше будуть проводитися з даними.

Наприклад, у пов'язаних списків легше виробляти операцію вставки і видалення елементів усередині структури. У масиву треба зміщувати всі наступні елементи, у списках досить поміняти покажчик одного тільки попереднього елемента.

З іншого боку, якщо треба отримати доступ до елементу за його номером, в масиві це робиться відразу, в той час як у списку доведеться пройти всі ланцюжок, починаючи з головного елемента.

У розглянутих односпрямованих списках, знаючи елемент, можна було знайти всі наступні елементи. Однак, якщо відома посилання на один з середніх елементів, то йдуть перед ним знайти не вдасться. Якщо в знаходженні попередніх елементів є потреба, то створюються двонаправлені списки, кожен елемент яких містить два покажчика - на попередній і наступний елементи.

Взагалі кажучи, кожен елемент подібної структури може містити скільки завгодно покажчиків на елементи того ж типу, що дозволяє формувати не тільки списки, але також дерева і графи.

Література

1. Д. Кнут. Искусство программирования на ЭВМ. т. 1. (раздел 2.3. «Деревья»).

2. Н. Вирт. Алгоритмы и структуры данных.