

**ФАХОВИЙ КОЛЕДЖ РАКЕТНО-КОСМІЧНОГО МАШИНОБУДУВАННЯ  
ДНІПРОВСЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ імені Олеся Гончара**

**Циклова комісія програмної інженерії**

## **НАВЧАЛЬНІ МАТЕРІАЛИ**

**до самостійного вивчення тем з дисципліни**

### **АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ**

---

спеціальність 121 Інженерія програмного забезпечення

освітня програма Розробка програмного забезпечення  
(назва освітньої програми)

відділення Комп'ютерної та програмної інженерії  
(назва відділення)

**Дніпро  
2023 рік**

## ЗМІСТ

|    |                                                                      |    |
|----|----------------------------------------------------------------------|----|
| 1. | Двозв'язні та кільцеві списки. Реалізація та основні функції обробки | 2  |
| 2. | Хеш – таблиці                                                        | 11 |
| 3. | Алгоритми уникнення колізій                                          | 23 |
| 4. | Алгоритми евристики. Співставлення зі зразком.                       | 37 |
| 5. | Динамічне програмування. Задача тріангуляції.                        | 53 |
| 6. | Алгоритми чисельної апроксимації.                                    | 65 |
| 7. | Алгоритми обробки символічної інформації                             | 78 |

### КРИТЕРІЇ ОЦІНЮВАННЯ роботи студента над темою для самостійного вивчення

В якості результату самостійної роботи над темою студент повинен надати конспект теми та відповіді на контрольні питання, приведені в плані самостійного вивчення відповідної теми.

Розподіл балів та критерії оцінювання виконаного завдання наведено в таблиці

| Опрацювання теми для самостійного вивчення  |          |                                                                                                                                                      |
|---------------------------------------------|----------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| Бали                                        | Критерій |                                                                                                                                                      |
| <b>2</b><br>Конспект теми                   | 2        | Конспект теми повний, логічно-послідовний. Студент зумів виділити та опрацювати основні положення теми, виділені в плані самостійного вивчення теми. |
|                                             | 1,5      | Конспект повний, але при цьому допущені невеликі пропуски, що не спотворили логічного і інформаційного змісту теми.                                  |
|                                             | 1        | Конспект неповно або непослідовно розкриває зміст матеріалу теми, але в конспекті виділені основні положення теми.                                   |
|                                             | 0,5      | Конспект не розкриває основний зміст навчального матеріалу, або студент не зумів виділити та опрацювати основні положення теми.                      |
| <b>3</b><br>Відповіді на контрольні питання | 3        | Студент правильно та повному об'ємі відповідає на всі контрольні питання, надані в плані самостійного вивчення теми.                                 |
|                                             | 2        | Студент правильно та повному об'ємі відповідає на 70% контрольних питань, наданих в плані самостійного вивчення теми.                                |
|                                             | 1        | Студент правильно та повному об'ємі відповідає на 50% контрольних питань, наданих в плані самостійного вивчення теми.                                |
|                                             | 0,5      | Студент правильно та повному об'ємі відповідає не більше ніж на 30% контрольних питань, наданих в плані самостійного вивчення теми.                  |
| <b>Максимальна кількість балів: 5</b>       |          |                                                                                                                                                      |

# План самостійного вивчення теми № 1

з навчальної дисципліни Алгоритми та структури даних

**Тема** Двозв'язні та кільцеві списки. Реалізація та основні функції обробки.

**Мета** З'ясувати структуру двозв'язного списку та виконання основних операцій над двозв'язним списком. З'ясувати структуру циклічного списку та виконання основних операцій над циклічним списком

*знати:*

- структуру лінійного двонаправленого списку;
- структуру лінійного циклічного списку.

*вміти:-*

- реалізовувати основні операції над двозв'язними та циклічними списками.

Час – 4 години

## Навчальні питання:

- 1 Структура двозв'язного списку.
- 2 Операції із двозв'язним списком
- 3 Структура циклічного списку
- 4 Операції над циклічними списками

## Методичні рекомендації

Самостійно вивчаючи запропоновану тему, студент повинен засвоїти такі основні положення:

*По-перше* Кожен вузол двозв'язного списку містить (крім корисних даних) також посилання на наступний за ним вузол (поле next) і попередній (поле prev). Поле next в останнього елемента й поле prev у першого містять NULL.

*По-друге* Основні операції із двозв'язним списком:

1. Додавання вузла в початок списку
2. Додавання вузла в кінець списку
3. Додавання вузла після заданого
4. Пошук вузла в списку

## 5. Видалення вузла

*По-третє*

Іноді список (однозв'язний або двусвязный) замикають у кільце, тобто покажчик next останнього елемента вказує на перший елемент, і (для двусвязних списків) покажчик prev першого елемента вказує на останній. У таких списках поняття "хвоста" списку не має змісту, для роботи з ним треба використати покажчик на "голову", причому "головою" можна вважати будь-який елемент.

## Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротиєєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац. ун-т залізн. трансп .ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

## Навчальні матеріали до самостійного вивчення теми

### 1 Структура двозв'язного списку.

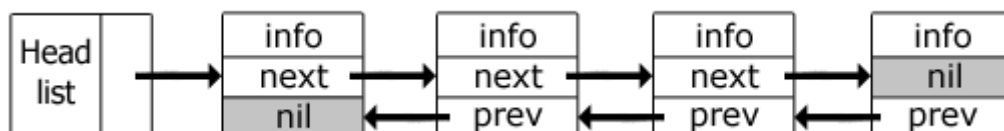
Список - абстрактний тип даних, що реалізує впорядкований набір значень. Списки відрізняються від масивів тим, що доступ до їхніх елементів здійснюється послідовно, у той час як масиви - структура даних довільного доступу. Даний абстрактний тип має кілька реалізацій у вигляді структур даних. Деякі з них будуть розглянуті тут.

Список (зв'язний список) - це структура даних, що представляє собою кінцева безліч упорядкованих елементів, зв'язаних один з одним посередством покажчиків. Кожен елемент структури містить поле з якою-небудь інформацією, а також покажчик на наступний елемент. На відміну від масиву, до елементів списку немає довільного доступу.



В однозв'язному списку, наведеному вище, початковим елементом є Head list (голова списку [довільне найменування]), а все інше називається хвостом. Хвіст списку становлять елементи, розділені на дві частини: інформаційну (поле info) і вказівну (поле next). В останньому елементі замість покажчика, утримується ознака кінця списку - nil.

Однозв'язний список не занадто зручний, тому що з однієї крапки є можливість потрапити лише в наступну крапку, рухаючись тим самим у кінець. Коли крім покажчика на наступний елемент є з і на попередній, то такий список називається двозв'язним.



Кожен вузол містить (крім корисних даних) також посилання на наступний за ним вузол (поле next) і попередній (поле prev). Поле next в останнього елемента й поле prev у першого містять NULL. Вузол оголошується так:

```
struct Node {  
    char word[40]; // область даних  
    int count;  
    Node *next, *prev; // посилання на сусідні вузли  
};  
  
typedef Node *PNode; // тип даних "показчик на вузол"
```

Надалі ми будемо вважати, що показчик Head указує на початок списку, а показчик Tail - на кінець списку:

```
PNode Head = NULL, Tail = NULL;
```

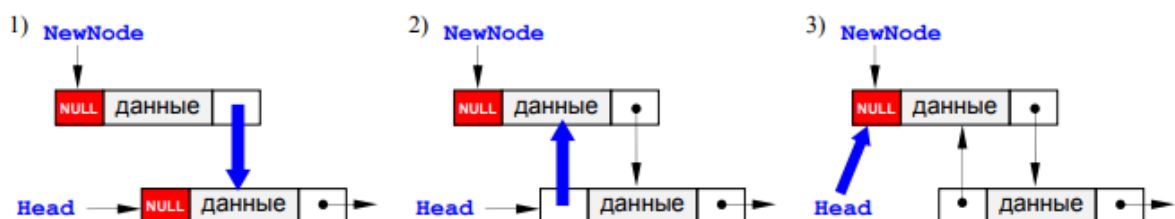
Для порожнього списку обидва показчики рівні NULL.

## 2 Операції із двозв'язним списком

### *Додавання вузла в початок списку*

При додаванні нового вузла NewNode у початок списку треба:

- 1) установити посилання next вузла NewNode на голову існуючого списку і його показчик prev в NULL;
- 2) установити посилання prev колишнього першого вузла (якщо він існував) на NewNode;
- 3) установити голову списку на новий вузол;
- 4) якщо в списку не було ні одного елемента, хвіст списку також установлюється на новий вузол.



За такою схемою працює наступна процедура:

```

void AddFirst(PNode &Head, PNode &Tail, PNode NewNode)
{
    NewNode->next = Head;
    NewNode->prev = NULL;
    if ( Head ) Head->prev = NewNode;
    Head = NewNode;
    if ( ! Tail ) Tail = Head; // цей елемент - перший
}

```

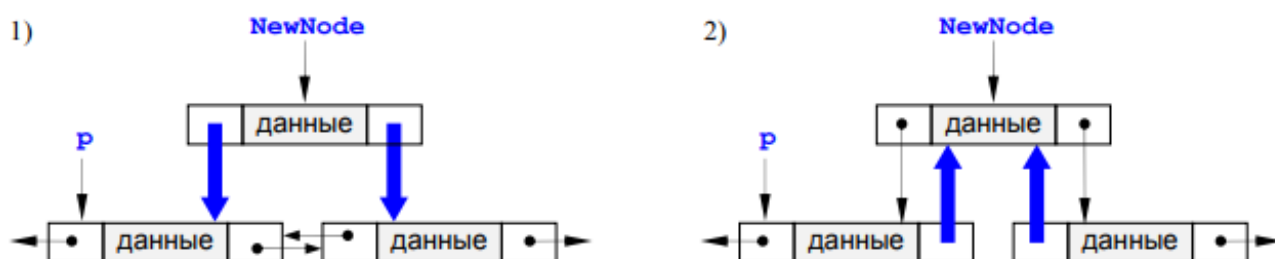
### *Додавання вузла в кінець списку*

Завдяки симетрії додавання нового вузла NewNode у кінець списку проходить аналогічно, у процедурі треба скрізь замінити Head на Tail і навпаки, а також поміняти prev й next.

### *Додавання вузла після заданого*

Дано адресу NewNode нового вузла й адреса p одного з існуючих вузлів у списку. Треба вставити в список новий вузол після p. Якщо вузол p є останнім, то операція зводиться до додавання в кінець списку (див. вище). Якщо вузол p - не останній, то операція вставки виконується у два етапи:

- 1) установити посилання нового вузла на наступний за даним (next) і попередній йому (prev);
- 2) установити посилання сусідніх вузлів так, щоб включити NewNode у список.



Такий метод реалізує наведена нижче процедура (вона враховує також можливість вставки елемента в кінець списку, саме для цього в параметрах передаються посилання на голову й хвіст списку):

```

void AddAfter (PNode &Head, PNode &Tail, PNode p, PNode NewNode)
{
if ( ! p->next )
    AddLast (Head, Tail, NewNode); // вставка в кінець списку
else {
    NewNode->next = p->next; // з посилання нового вузла
    NewNode->prev = p;
    p->next->prev = NewNode; // з посилання сусідніх вузлів
    p->next = NewNode;
}
}

```

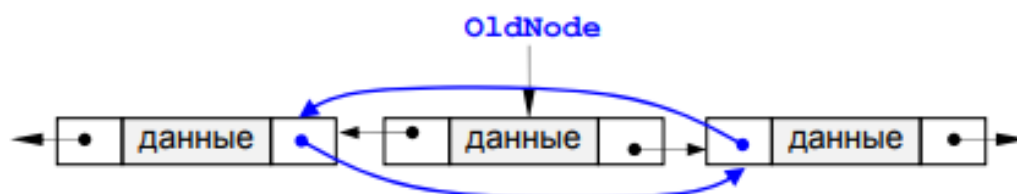
Додавання вузла перед заданим виконується аналогічно.

### *Пошук вузла в списку*

Прохід по двусвязному списку може виконуватися у двох напрямках - від голови до хвоста (як для однозв'язкового) або від хвоста до голови.

### *Видалення вузла*

Ця процедура також вимагає посилання на голову й хвіст списку, оскільки вони можуть змінитися при видаленні крайнього елемента списку. На першому етапі встановлюються посилання сусідніх вузлів (якщо вони є) так, ніби вузла, що видаляється, не було б. Потім вузол видаляється й пам'ять, що він займає, звільняється. Ці етапи показані на малюнку внизу. Окремо перевіряється, чи не є вузол, що видаляється, першим або останнім вузлом списку.



Такий метод реалізує наведена нижче процедура:

```

void Delete(PNode &Head, PNode &Tail, PNode OldNode)
{

```

```

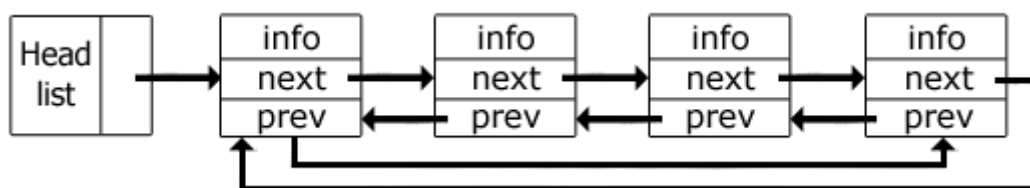
if (Head == OldNode) {
    Head = OldNode->next; // видаляємо перший елемент
    if ( Head )
        Head->prev = NULL;
    else Tail = NULL; // видалили єдиний елемент
}
else {
    OldNode->prev->next = OldNode->next;
    if ( OldNode->next )
        OldNode->next->prev = OldNode->prev;
    else Tail = NULL; // видалили останній елемент
}
delete OldNode;
}

```

### 3 Структура циклічного списку

Можливість рухатися як уперед, так і назад корисна для виконання деяких операцій, але додаткові покажчики вимагають залучення більшої кількості пам'яті, чим такий необхідно в еквівалентному однозв'язному списку.

Для двох видів списків описаних вище існує підвид, називаний кільцевим списком. Зробити з однозв'язного списку кільцевий можна додавши всього лише один покажчик в останній елемент, так щоб він посилався на перший. А для двусвязного буде потрібно два покажчики: на перший й останній елементи.



Іноді список (однозв'язний або двусвязний) замикають у кільце, тобто покажчик next останнього елемента вказує на перший елемент, і (для двусвязних списків)

показчик `prev` першого елемента вказує на останній. У таких списках поняття "хвоста" списку не має змісту, для роботи з ним треба використати показчик на "голову", причому "головою" можна вважати будь-який елемент.

#### 4 Операції над циклічними списками

Операція *включення* в кільцевий двозв'язаний список подібна до попередніх. Наприклад, нехай потрібно встановити ділянку з новим записом після ділянки з індексом  $i$ . Нова ділянка розміщується на вільному місці пам'яті і включається у список шляхом коректування вказівників в ділянці з індексом  $i$ , а також у тій ділянці, яка раніше була після неї.

Операція *виключення* ділянки із кільцевого двозв'язного списку також зводиться до коректування двох посилань - у "сусіда ліворуч" змінюється вказівник на наступну ділянку, а у "сусіда праворуч" - вказівник на попередню ділянку.

Циклічний двозв'язний список можна відобразити у одномірний масив аналогічно лінійному списку. У такому випадку процедури включення-виключення елементів зводяться до обчислення індексів вказівників, як і в лінійному списку, і заміни вмістимого елементів масиву з цими індексами. Якщо тіло елемента списку являв собою рядок символів або бітів, а вказівники - цілі числа, то не в кожній мові програмування таке відображення можливе. Тому, наприклад, простіше відображувати списки у два масиви - один для тіла елемента, другий - для довідкової частини.

Частіше, однак, елементи списку не розташовують у якому-небудь масиві, а просто розміщують кожний окремо в оперативній пам'яті, виділеній даній задачі. Звичай елементи списку розміщуються в динамічній пам'яті. У якості вказівників у цьому випадку використовують адреси елементів в оперативній пам'яті.

Голова списку зберігає вказівник на перший і останній елементи списку. Оскільки список зациклений у кільце, то наступним за головою списку буде його перший елемент, а попереднім - останній елемент. Голова списку зберігає тільки вказівник й не зберігає ніякого елемента. Це як би порожня комірка, у якій не можна нічого покласти і яка використовується тільки для того, щоб зберігати адреси наступного й

попереднього елементів списку, тобто першого й останнього. Коли список порожній, голова списку зациклена сама на себе.

Перевага вказівникової реалізації списку полягає в тому, що процедури додавання й видалення елементів не приводять до масових операцій.

### **Питання та завдання для самоконтролю знань студентів**

1. Які існують різновиду зв'язних списків?
2. У чому складається відмінність лінійного списку від кільцевого?
3. У чому полягають недоліки однозв'язного списку?
4. У чому складається відмінність однозв'язного списку від двусвязного?
5. У чому відмінність операції вставки у двусвязный список від вставки в однозв'язний список?
6. У чому відмінність операції видалення із двусвязного списку від видалення з однозв'язного списку?
7. У чому полягають особливості роботи з кільцевими списками?
8. У чому складається відмінність ланки двусвязного списку від ланки однозв'язного списку?
9. Зобразите структуру лінійного двусвязного списку.
10. Перелічіть достоїнства й недоліки лінійних двусвязных списків.
11. Зобразите можливі структури двусвязных кільцевих списків.

## План самостійного вивчення теми № 2

з навчальної дисципліни Алгоритми та структури даних

|             |                                                                                                                                                              |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Тема</b> | Хеш – таблиці                                                                                                                                                |
| <b>Мета</b> | Ознайомлення з поняттями швидкого доступу до даних та хешування даних. Ознайомлення з основними принципами побудови хеш-таблиці з використанням хеш-функції. |

*знати:*

- поняття швидкого доступу до даних та хешування даних;
- основні принципи побудови таблиці на основі хеш - функції;

Час – 4 години

### Навчальні питання:

- 1 Методи швидкого доступу до даних. Хешування даних
- 2 Хеш-таблиця
- 3 Хеш-функція

### Методичні рекомендації

Самостійно вивчаючи запропоновану тему, студент повинен засвоїти такі основні положення:

*По-перше* Для скорочення часу доступу до даних використовуються так зване випадкове впорядкування або хешування. При цьому дані організуються у вигляді таблиці за допомогою хеш-функції  $h$ , яка використовується для „обчислення” адреси за значенням ключа.

*По-друге* Хеш-таблиця - це звичайний масив з незвичайною адресацією, що задається хеш-функцією. Доступ до елементів здійснюється по ключі (key). Ключі можуть бути рядками, числами, покажчиками. Хеш-таблиці дозволяють у середньому за час  $O(1)$  виконувати додавання, пошуки й видалення елементів.

*По-третє* Хеш-функція - легко обчислювальна функція, що перетворює вихідне повідомлення довільної довжини (прообраз)

у повідомлення фіксоване довжини (хеш-образ), для якої не існує ефективного алгоритму пошуку колізій.

*По-четверте*

В результаті, для реалізації хешування необхідні три речі:

1. Структура даних (хеш-таблиця) для зберігання даних;
2. Функція хешування, яка встановлює відповідність між значеннями ключа і розміщенням в таблиці;
3. Алгоритм вирішення конфліктів, який визначає послідовність дій, якщо декілька ключів відповідають одній комірці таблиці.

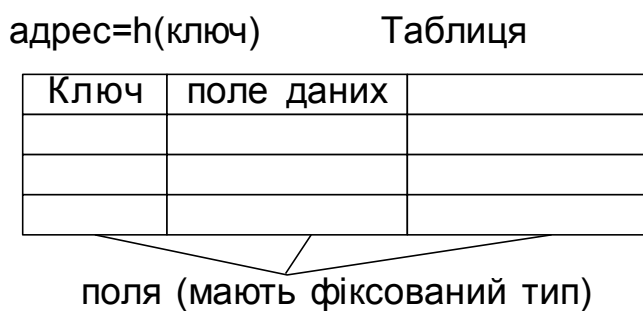
## **Література**

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротиєєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп .ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

## Навчальні матеріали до самостійного вивчення теми

### 1 Методи швидкого доступу до даних. Хешування даних

Для прискорення доступу до даних можна використовувати попереднє їх впорядкування у відповідності зі значеннями ключів. При цьому можуть використовуватися методи пошуку в упорядкованих структурах даних, наприклад, метод дихотомічного пошуку, що суттєво скорочує час пошуку даних за значенням ключа. Проте при добавленні нового запису потрібно дані знову впорядкувати. Втрати часу на повторне впорядкування можуть значно перевищувати виграш від скорочення часу пошуку. Тому для скорочення часу доступу до даних використовується так зване випадкове впорядкування або хешування. При цьому дані організуються у вигляді таблиці за допомогою хеш-функції  $h$ , яка використовується для „обчислення” адреси за значенням ключа.



Ми розглядали алгоритм інтерполяційного пошуку, який використовує інтерполяцію для пришвидшення пошуку. Порівнюючи шукане значення зі значеннями елементів у відомих точках, цей алгоритм може визначити імовірне розміщення шуканого елемента. По суті, він створює функцію, яка встановлює відповідність між шуканим значенням і індексом позиції, в якій він повинен знаходитися. Якщо перше передбачення помилкове, то алгоритм знову використовує цю функцію, передбачаючи нове розміщення, і так далі, до тих пір, поки шуканий елемент не буде знайдено.

Хешування використовує аналогічний підхід, відображаючи елементи в хеш-таблиці. Алгоритм хешування використовує деяку функцію, яка визначає імовірне розміщення елемента в таблиці на основі значення шуканого елемента.

Ідеальною хеш-функцією є така хеш-функція, яка для будь-яких двох неоднакових ключів дає неоднакові адреси. Підібрати таку функцію можна у випадку, якщо всі можливі значення ключів відомі наперед. Така організація даних носить назву „досконале хешування”.

Наприклад, потрібно запам'ятати декілька записів, кожен з яких має унікальний ключ зі значенням від 1 до 100. Для цього можна створити масив з 100 комітками і присвоїти кожній комітці нульовий ключ. Щоб додати в масив новий запис, дані з нього просто копіюються у відповідну комітку масиву. Щоб додати запис з ключем 37, дані з нього копіюються в 37 позицію в масиві. Щоб знайти запис з певним ключем – вибирається відповідна комітка масиву. Для вилучення запису ключу відповідної комітки масиву просто присвоюється нульове значення. Використовуючи цю схему, можна додати, знайти і вилучити елемент із масиву за один крок.

У випадку наперед невизначеної множини значень ключів і обмеженні розміру таблиці підбір досконалої функції складний. Тому часто використовують хеш-функції, які не гарантують виконання умови.

Наприклад, база даних співробітників може використовувати в якості ключа ідентифікаційний номер. Теоретично можна було б створити масив, кожна комітка якого відповідала б одному з можливих чисел; але на практиці для цього не вистарчить пам'яті або дискового простору. Якщо для зберігання одного запису потрібно 1 КБ пам'яті, то такий масив зайняв би 1 ТБ (мільйон МБ) пам'яті. Навіть якщо можна було б виділити такий об'єм пам'яті, така схема була б дуже неекономною. Якщо штат фірми менше 10 мільйонів співробітників, то більше 99 процентів масиву буде пустою.

Щоб вирішити цю проблему, схеми хешування відображають потенційно велику кількість можливих ключів на достатньо компактну хеш-таблицю. Якщо на фірмі працює 700 співробітників, можна створити хеш-таблицю з 1000 коміток. Схема хешування встановлює відповідність між 700 записами про співробітників і 1000 позиціями в таблиці. Наприклад, можна розміщати записи в таблиці у відповідності з трьома першими цифрами ідентифікаційного номеру. При цьому запис про співробітника з номером 123456789 буде знаходитися в 123 комітці таблиці.

Очевидно, що оскільки існує більше можливих значень ключа, ніж комірок в таблиці, то деякі значення ключів можуть відповідати одним і тим коміркам таблиці. Даний випадок носить назву „колізія”, а такі ключі називаються „ключі-синоніми”.

Щоб уникнути цієї потенційної проблеми, схема хешування повинна включати в себе алгоритм вирішення конфліктів, який визначає послідовність дій у випадку, якщо ключ відповідає позиції в таблиці, яка вже зайнята іншим записом.

Усі методи використовують для вирішення конфліктів приблизно однаковий підхід. Вони спочатку встановлюють відповідність між ключем запису і розміщенням в хеш-таблиці. Якщо ця комірка вже зайнята, вони відображають ключ на іншу комірку таблиці. Якщо вона також вже зайнята, то процес повторюється знову до тих пір, поки нарешті алгоритм не знайде пусту комірку в таблиці. Послідовність позицій, які перевіряються при пошуку або вставці елемента в хеш-таблицю, називається тестовою послідовністю.

В результаті, для реалізації хешування необхідні три речі:

- Структура даних (хеш-таблиця) для зберігання даних;
- Функція хешування, яка встановлює відповідність між значеннями ключа і розміщенням в таблиці;
- Алгоритм вирішення конфліктів, який визначає послідовність дій, якщо декілька ключів відповідають одній комірці таблиці.

## 2 Хеш-таблиця

Хеш-таблиця - це звичайний масив з незвичайною адресацією, що задається хеш-функцією. Доступ до елементів здійснюється по ключі (key). Ключі можуть бути рядками, числами, покажчиками. Хеш-таблиці дозволяють у середньому за час  $O(1)$  виконувати додавання, пошуки й видалення елементів.

Наприклад, на `hashTable` (рисунок 1) – це масив з 8 елементів. Кожен елемент являє собою покажчик на лінійний список, що зберігає числа. Хеш-функція в цьому прикладі просто ділить ключ на 8 і використовує залишок як індекс у таблиці. Це дає нам числа від 0 до 7. Оскільки для адресації в `hashTable` нам і потрібні числам від 0 до 7, алгоритм гарантує припустимі значення індексів.

Хешування корисно, коли широкий діапазон можливих значень повинен бути збережений у малому обсязі пам'яті, і потрібний спосіб швидкого, практично довільного доступу. Хеш-таблиці часто застосовуються в базах даних, і, особливо, у мовних процесорах типу компіляторів й асемблерів, де вони добірно обслуговують таблиці ідентифікаторів. У таких додатках, таблиця – найкраща структура даних.

Тому що всякий доступ до таблиці повинен бути зроблений через хеш-функцію, функція повинна задовольняти двом вимогам: Вона повинна бути швидкої, і вона повинна породжувати гарні ключі для розподілу елементів по таблиці. Остання вимога мінімізує колізії (випадки, коли два різних елементи мають однакове значення хеш-функції) і запобігає випадку, коли елементи даних із близькими значеннями попадають тільки в одну частину таблиці.

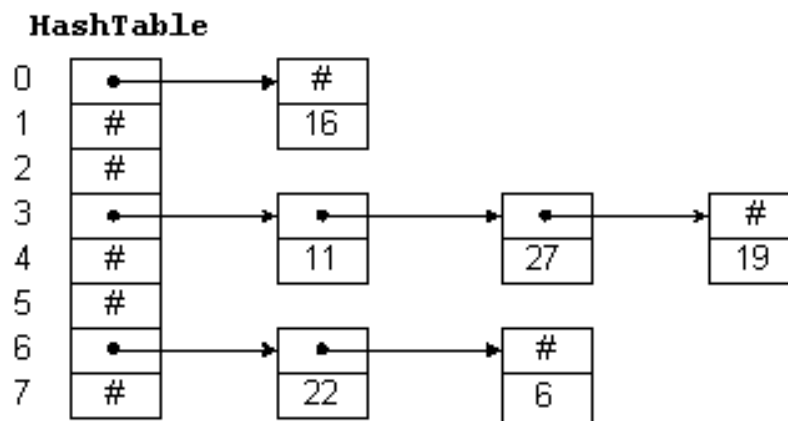


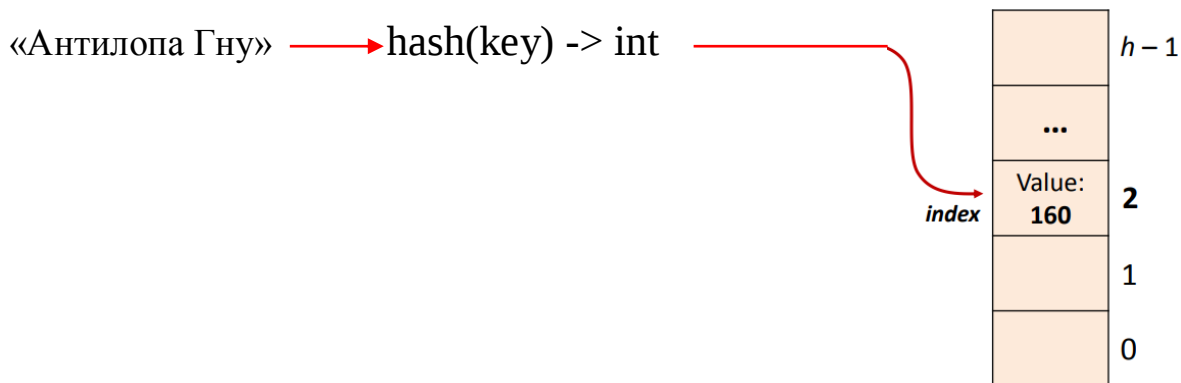
Рис. 1. Хеш-таблиця

Щоб вставити в таблицю новий елемент, ми хешуємо ключ, щоб визначити список, у який його потрібно додати, потім вставляємо елемент у початок цього списку. Наприклад, щоб додати 11, ми ділимо 11 на 8 й одержуємо залишок 3. Таким чином, 11 варто розмістити в списку, на початок якого вказує `hashTable[3]`. Щоб знайти число, ми його хешуємо і проходимо по відповідному списку. Щоб видалити число, ми знаходимо його й видаляємо елемент списку, що його утримує.

При додаванні елементів у хеш-таблицю виділяються шматки динамічної пам'яті, які організуються у вигляді зв'язаних списків, кожний з яких відповідає входу хеш-таблиці. Цей метод називається зв'язуванням.

Основна ідея побудови хеш-таблиці (Hash tables)

|                   |                                    |                                 |
|-------------------|------------------------------------|---------------------------------|
| <b>Ключ (Key)</b> | <b>Хеш-функція (Hash function)</b> | <b>Хеш-таблиця (Hash table)</b> |
|-------------------|------------------------------------|---------------------------------|



- Хеш-функція - відображає (перетворює) ключ (key) у номер елемента (index) масиву (у ціле число від 0 до  $h - 1$ ).
- Час обчислення хеш-функції залежить від довжини ключа й не залежить від кількості елементів у масиві.
- Комірки масиву називаються buckets, slots.
- На практиці, звичайно відома інформація про діапазон значень ключів.
- На основі цього вибирається розмір  $h$  хеш-таблиці й вибирається хеш-функція
- Коефіцієнт  $\alpha$  заповнення хеш-таблиці (load factor, fill factor) - відношення числа  $n$  збережених елементів у хеш-таблиці до розміру  $h$  масиву (середнє число елементів на один осередок)  $\alpha = n/h$ .
- Приклад:  $h = 128$ , у хеш-таблицю додали 50 елементів, тоді  $\alpha = 50 / 128 \approx 0.39$
- Від цього коефіцієнта залежить середній час виконання операцій додавання, пошуку й видалення елементів.

### 3 Хеш-функція

Хешування (hashing) - перетворення вхідного масиву даних довільної довжини у вихідний рядок фіксованої довжини. Такі перетворення також називаються хеш-фу-

нкціями або функціями згортки, вхідний масив - прообразом, а результати перетворення - хешем, хеш-кодом, хеш-образом, цифровим відбитком або дайджестом повідомлення (message digest).

Хеш-функція - легко обчислювальна функція, що перетворить вихідне повідомлення довільної довжини (прообраз) у повідомлення фіксоване довжини (хеш-образ), для якої не існує ефективного алгоритму пошуку колізій.

Колізією для функції  $h$  називається пара значень  $x, y, x \neq y$ , така, що  $h(x) = h(y)$ . Таким чином хеш-функція повинна мати наступні властивості:

- для даного значення  $h(x)$  неможливо знайти значення аргументу  $x$ . Такі хеш-функції називають стійкими в змісті обігу або стійкими в сильному змісті;
- для даного аргументу  $x$  неможливо знайти інший аргумент  $y$  такий, що  $h(x) = h(y)$ . Такі хеш-функції називають стійкими в змісті обчислення колізій або стійкими в слабкому змісті.

У випадку, коли значення хеш-функції залежить не тільки від прообразу, але й закритого ключа, те це значення називають кодом перевірки дійсності повідомлень (Message Authentication Code, MAC), кодом перевірки дійсності даних (Data Authentication Code, DAC) або имитовставкой.

На практиці хеш-функції використовують у наступних цілях:

- для прискорення пошуку даних у БД;
- для перевірки цілісності й дійсності повідомлень;
- для створення стислого образу, застосовуваного в процедурах ЕЦП;
- для захисту пароля в процедурах аутентифікації.

Прискорення пошуку даних. Наприклад, при записі текстових полів у базі даних може розраховуватися їхній хеш-код і дані можуть міститися в розділ, що відповідає цьому хеш-коду. Тоді при пошуку даних треба буде спочатку обчислити хеш-код тексту й відразу стане відомо, у якому розділі їх треба шукати, тобто шукати треба буде не по всій базі, а тільки по одному її розділі (це сильно прискорює пошук).

Побутовим аналогом хешування в цьому випадку може служити розміщення слів у словнику за алфавітом. Перша буква слова є його хеш-кодом, і при пошуку ми переглядаємо не весь словник, а тільки розділ з потрібною буквою.

Процедура обчислення (стандартна схема алгоритму) хеш-функції представлена на рисунку 2.

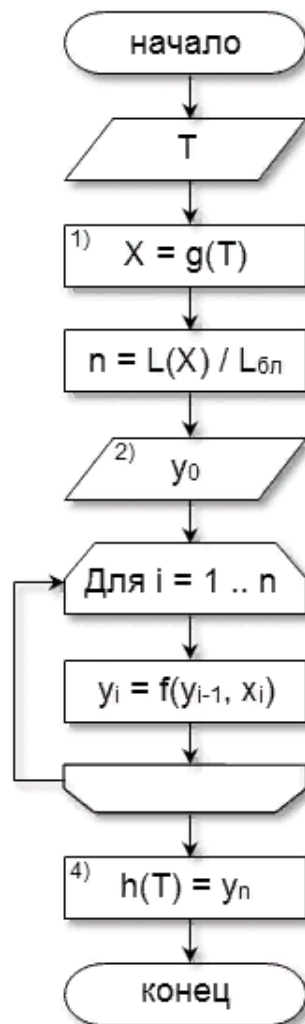


Рис. 2. Процедура обчислення значення хеш-функції.

Навіщо потрібний хеш

Розглянемо ще приклад. У вас є текст у файлі. Але насправді це текст, а масив цифрових символів (по суті число). Як ви знаєте, у комп'ютерній логіці використовуються двійкові числа (нуль й одиниця). Вони запросто можуть бути перетворені в шестнадцятирічні цифри, над якими можна проводити математичні операції. Застосувавши до них хеш-функцію ми одержимо на виході (після ряду ітерацій) число заданої довжини (хеш-сумму).

Якщо ми потім у вихідному текстовому файлі поміняємо хоча б одну букву або додамо зайвий пробіл, то повторно розрахований для нього хеш уже буде відрізнятися від споконвічного (взагалі інше число буде). Доходить, навіщо все це потрібно? Ну,

звичайно ж, для того, щоб зрозуміти, що файл саме той, що й повинен бути. Це можна використати в цілому ряді аспектів роботи в Інтернеті й без цього взагалі складно уявити собі роботу мережі.

Наприклад, прості хеш-функції (не надійні, але розраховує швидко) застосовуються при перевірці цілісності передачі пакетів по протоколі TCP/IP (і ряду інших протоколів й алгоритмів, для виявлення апаратних помилок і збоїв - так називане надлишкове кодування). Якщо розраховане значення хеша збігається з відправленим разом з пакетом (так називаною контрольною сумою), то значить втрат по шляху не було (можна переходити до наступного пакета).

А це, адже на хвилинку, основний протокол передачі даних у мережі Інтернет. Без нього нікуди. Так, є ймовірність, що відбудеться накладка - їх називають колізіями. Адже для різних споконвічних даних може вийти той самий хеш. Ніж простіше використається функція, тим вище така ймовірність. Але отут потрібно просто вибрати тим часом, що важливіше в цей момент - надійність ідентифікації або швидкість роботи. У випадку TCP/IP важлива саме швидкість. Але є й інші області, де важливіше саме надійність.

Схожа схема використається й у технології блокчейн, де хеш виступає гарантією цілісності ланцюжка транзакцій (платежів) і захищає неї від несанкціонованих змін. Завдяки йому й розподіленим обчисленням зламати блокчен дуже складно й на його основі благополучно існує безліч криптовалют, включаючи саму популярну з них - це биткоин. Останній існує вже з 2009 рік і дотепер не був зламаний.

Більше складні хеш-функції використаються в криптографії. Головна умова для них - неможливість за кінцевим результатом (хешу) обчислити початковий (масив даних, що обробили даною хеш-функцією). Друга головна умова - стійкість до колізіями, тобто низька ймовірність одержання двох однакових хеш-сумм із двох різних масивів даних при обробці їхньою цією функцією. Розрахунки по таких алгоритмах більше складні, але отут уже головне не швидкість, а надійність.

Так само хеширование використається в технології електронного цифрового підпису. За допомогою хеша отут знову ж засвідчують, що підписують саме той документ, що потрібно. Саме він (хеш) передається в токен, що і формує електронний

цифровий підпис. Але про це, я сподіваюся, ще буде окрема стаття, тому що тема цікава, але у двох абзацах її не розкриєш.

Для доступу до сайтів і серверів по логину й паролі теж часто використовують хеширование. Погодитися, що зберігати паролі у відкритому виді (для їхнього звірення із уводять користувачами, що) досить ненадійно (можуть їх викрасти). Тому зберігають хеши всіх паролів. Користувач уводить символи свого пароля, миттєво розраховується його хеш-сумма й звіряється з тим, що є в базі. Надійно й дуже просто. Звичайно для такого типу хеширования використовують складні функції з дуже високої криптостойкістю, щоб по хешу не можна було б відновити пароль.

Якими властивостями повинна володіти хеш-функція:

- Як уже було сказано, функція ця повинна вміти приводити будь-який обсяг даних (а всі вони цифрові, тобто двійкові, як ви розумієте) до числа заданої довжини (по суті цей стиск до бітової послідовності заданої довжини хитрим способом).
- При цьому найменша зміна (хоч на один біт) вхідних даних повинне приводити до повної зміни хеша.
- Вона повинна бути стійкою у зворотній операції, тобто ймовірність відновлення вихідних даних по хешу повинна бути досить низкою (хоча останнє сильно залежить від задіяних потужностей)
- В ідеалі вона повинна мати як можна більше низьку ймовірність виникнення колізій. Погодитися, що не айс буде, якщо з різних масивів даних будуть часто виходити ті самі значення хеша.
- Гарна хеш-функція не повинна сильно навантажувати залізо при своєму виконанні. Від цього сильно залежить швидкість роботи системи на ній побудованої. Як я вже говорив вище, завжди є компроміс між швидкістю роботи і якістю одержуваного результату.
- Алгоритм роботи функції повинен бути відкритим, щоб будь-який бажаючий міг би оцінити її криптостойкість, тобто ймовірність відновлення початкових даних по видаваному хешу.

## **Питання та завдання для самоконтролю знань студентів**

- 1 Сформулювати основну ціль хешування даних.
- 2 В чому полягає процес хешування даних?
- 3 Призначення хеш-таблиці.
- 4 Навести приклад побудови хеш-таблиці.
- 5 Призначення хеш-функції.
- 6 Сформулювати основний алгоритм побудови хеш-функції.

# План самостійного вивчення теми № 3

з навчальної дисципліни Алгоритми та структури даних

**Тема** Алгоритми уникнення колізій.

**Мета** Ознайомитись з основними методами уникнення колізій при хешуванні.

*знати:*

- основні принципи побудови хеш-таблиці на основі хеш - функції;
- основні методи уникнення колізій;
- основні методи організації даних для прискорення пошуку за вторинними ключами.

*вміти:-*

- використовувати метод ланцюжків та метод відкритої адресації;
- проводити оцінку якості хеш - функції;

Час – 4 години

## Навчальні питання:

- 1 Методи розв'язання колізій
  - 1.1 Метод ланцюжків
  - 1.2 Метод відкритої адресації
- 2 Переповнення таблиці і повторне хешування
- 3 Оцінка якості хеш-функції
- 4 Організація даних для прискорення пошуку за вторинними ключами

## Методичні рекомендації

Самостійно вивчаючи запропоновану тему, студент повинен засвоїти такі основні положення:

*По-перше* Методом ланцюжків називається метод, в якому для розв'язання колізій у всі записи вводяться покажчики, які використовуються для організації списків – „ланцюжків перепов-

внення”. У випадку виникнення колізій при заповненні таблиці в список для потрібної адреси хеш-таблиці додається ще один елемент.

*По-друге*

Метод відкритої адресації полягає в тому, щоб, користуючись якимось алгоритмом, який забезпечує перебір елементів таблиці, переглядати їх в пошуках вільного місця для нового запису.

*По-третє*

При великій наповненості таблиці виникають часті колізії і циклічні переходи на початок таблиці. При невдалому виборі хеш-функції відбуваються аналогічні явища. В найгіршому випадку при повному заповненні таблиці алгоритми циклічного пошуку вільного місця приведуть до зациклювання. Тому при використанні хеш-таблиць необхідно старатися уникати дуже густого заповнення таблиць. Звичайно довжину таблиці вибирають із розрахунку дворазового перевищення передбачуваної максимальної кількості записів. Не завжди при організації хешування можна правильно оцінити потрібну довжину таблиці, тому у випадку великої наповненості таблиці може знадобитися рехешування. У цьому випадку збільшують довжину таблиці, змінюють хеш-функцію і впорядковують дані.

*По-четверте*

Як вже було відмічено, дуже важливий правильний вибір хеш-функції. При вдалій побудові хеш-функції таблиця заповнюється більш рівномірно, зменшується кількість колізій і зменшується час виконання операцій пошуку, вставки і вилучення. Для того щоб попередньо оцінити якість хеш-функції можна провести імітаційне моделювання.

## **Література**

1. Крєневич А.П. Алгоритми і структури даних. Підручник. [Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротисєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп. ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

# Навчальні матеріали до самостійного вивчення теми

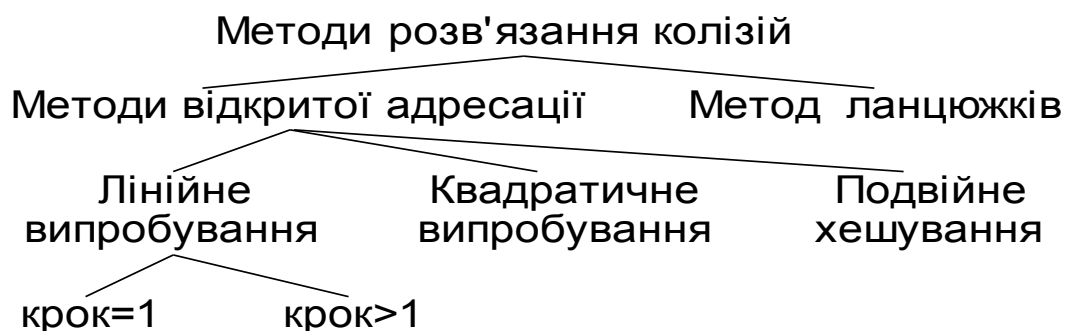
## Вступ

При одержанні таблиці за допомогою перетворення ключів має місце один недолік. Припустимо, що існують два різних ключі  $k_1$  й  $k_2$  ( $k_1 \neq k_2$ ) такі, що  $h(k_1) = h(k_2)$ . Коли запис із ключем  $k_1$  уводиться в таблицю, вона уставляється в позицію з індексом  $h(k_1)$ . Але коли хеширується ключ  $k_2$ , одержувана позиція є тією же позицією, у якій зберігається запис із ключем  $k_1$ . Ясно, що два записи не можуть займати ту саму позицію. Така ситуація називається колізією (collision) при хешуванні или столкновением. Іноді колізію називають конфліктом.

Алгоритм, що дозволяє розподіляти в таблиці запису, що конкурують із іншими записами в один осередок хеш-таблиці, називається методом розв'язання колізій (collision resolution).

## 1 Методи розв'язання колізій

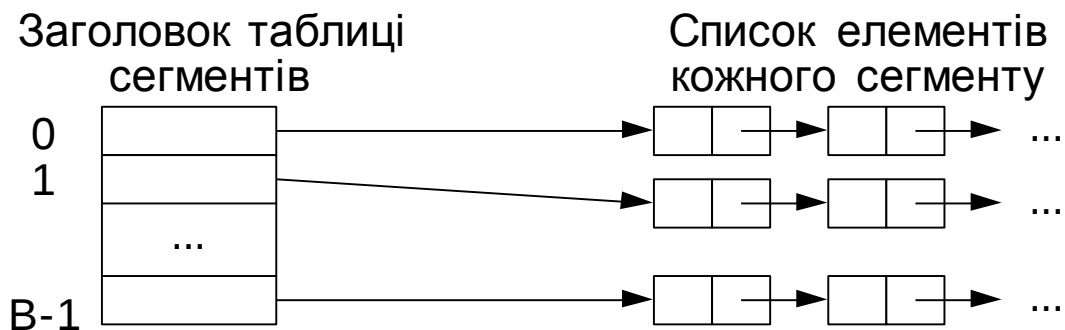
Для розв'язання колізій використовуються різноманітні методи, які в основному зводяться до методів „ланцюжків” і „відкритої адресації”.



### 1.1 Метод ланцюжків

Методом ланцюжків називається метод, в якому для розв'язання колізій у всі записи вводяться покажчики, які використовуються для організації списків – „ланцюжків переповнення”. У випадку виникнення колізій при заповненні таблиці в список для потрібної адреси хеш-таблиці добавляється ще один елемент.

Пошук в хеш-таблиці з ланцюжками переповнення здійснюється наступним чином. Спочатку обчислюється адреса за значенням ключа. Потім здійснюється послідовний пошук в списку, який зв'язаний з обчисленим адресом. Процедура вилучення з таблиці зводиться до пошуку елемента в його вилучення з ланцюжка переповнення. Схематичне зображення хеш-таблиці при такому методі розв'язання колізій приведене на наступному рисунку.



Якщо сегменти приблизно однакові за розміром, то у цьому випадку списки усіх сегментів повинні бути найбільш короткими при даній кількості сегментів. Якщо вихідна множина складається з  $N$  елементів, тоді середня довжина списків буде рівна  $N/B$  елементів. Якщо можна оцінити  $N$  і вибрати  $B$  якомога ближчим до цієї величини, то в списку буде один-два елементи. Тоді час доступу до елемента множини буде малою постійною величиною, яка залежить від  $N$ .

Одна з переваг цього методу хешування полягає в тому, що при його використанні хеш-таблиці ніколи не переповнюються. При цьому вставка і пошук елементів завжди виконується дуже просто, навіть якщо елементів в таблиці дуже багато. Із хеш-таблиці, яка використовує зв'язування, також просто вилучати елементи, при цьому елемент просто вилучається з відповідного зв'язного списку.

Один із недоліків зв'язування полягає в тому, що якщо кількість зв'язних списків недостатньо велика, то розмір списків може стати великим, при цьому для вставки чи пошуку елемента необхідно буде перевірити велику кількість елементів списку.

## 1.2 Метод відкритої адресації

Метод відкритої адресації полягає в тому, щоб, користуючись якимось алгоритмом, який забезпечує перебір елементів таблиці, переглядати їх в пошуках вільного місця для нового запису.

Лінійне випробування зводиться до послідовного перебору елементів таблиці з деяким фіксованим кроком

$$a = h(\text{key}) + c * i,$$

де  $i$  – номер спроби розв'язати колізію. При кроці рівному одиниці відбувається послідовний перебір усіх елементів після поточного.

Квадратичне випробування відрізняється від лінійного тим, що крок перебору елементів нелінійно залежить від номеру спроби знайти вільний елемент

$$a = h(\text{key}^2) + c * i + d * i^2$$

Завдяки нелінійності такої адресації зменшується кількість спроб при великій кількості ключів-синонімів. Проте навіть відносно невелика кількість спроб може швидко привести до виходу за адресний простір невеликої таблиці внаслідок квадратичної залежності адреси від номеру спроби.

Ще один різновид методу відкритої адресації, яка називається подвійним хешуванням, базується на нелінійній адресації, яка досягається за рахунок сумування значень основної і додаткової хеш-функцій.

$$a = h_1(\text{key}) + i * h_2(\text{key}).$$

Розглянемо алгоритми вставки і пошуку для методу лінійного випробування.

*Вставка:*

1.  $i = 0$
2.  $a = h(\text{key}) + i * c$
3. Якщо  $t(a) = \text{вільно}$ , то  $t(a) = \text{key}$ , записати елемент і зупинитися
4.  $i = i + 1$ , перейти до кроку 2

*Пошук:*

1.  $i = 0$
2.  $a = h(\text{key}) + i * c$

3. Якщо  $t(a) = \text{key}$ , то зупинитися – елемент знайдено
4. Якщо  $t(a) = \text{вільно}$ , то зупинитися – елемент не знайдено
5.  $i = i + 1$ , перейти до кроку 2

Аналогічно можна було б сформулювати алгоритми додавання і пошуку елементів для будь-якої схеми відкритої адресації. Відмінності будуть лише у виразі, який використовується для обчислення адреси (крок 2).

З процедурою вилучення справа складається не так просто, так як вона в даному випадку не буде оберненою до процедури вставки. Справа в тому, що елементи таблиці знаходяться в двох станах: вільно і зайнято. Якщо вилучити елемент, перевівши його в стан вільно, то після такого вилучення алгоритм пошуку буде працювати некоректно. Нехай ключ елемента, який вилучається, має в таблиці ключі синоніми. У цьому випадку, якщо за ним в результаті розв'язання колізій були розміщені елементи з іншими ключами, то пошук цих елементів після вилучення завжди буде давати негативний результат, так як алгоритм пошуку зупиняється на першому елементі, який знаходиться в стані вільно.

Скоректувати цю ситуацію можна різними способами. Самий простий із них полягає в тому, щоб проводити пошук елемента не до першого вільного місця, а до кінця таблиці. Проте така модифікація алгоритму зведе нанівець весь виграш в прискоренні доступу до даних, який досягається в результаті хешування.

Інший спосіб зводиться до того, щоб відслідкувати адреси всіх ключів-синонімів для ключа елемента, що вилучається, і при необхідності розмістити відповідні записи в таблиці. Швидкість пошуку після такої операції не зменшиться, але затрати часу на саме розміщення елементів можуть виявитися значними.

Існує підхід, який не має перерахованих недоліків. Його суть полягає в тому, що для елементів хеш-таблиці додається стан „вилучено”. Даний стан в процесі пошуку інтерпретується, як зайнято, а в процесі запису як вільно.

Тепер можна сформулювати алгоритми вставки, пошуку і вилучення для хеш-таблиці, яка має три стани елементів.

*Вставка:*

1.  $i = 0$
2.  $a = h(\text{key}) + i * c$

3. Якщо  $t(a) = \text{вільно}$  або  $t(a) = \text{вилучено}$ , то  $t(a) = \text{key}$ , записати елемент і стоп
4.  $i = i + 1$ , перейти до кроку 2

*Видалення:*

1.  $i = 0$
2.  $a = h(\text{key}) + i * c$
3. Якщо  $t(a) = \text{key}$ , то  $t(a) = \text{вилучено}$ , стоп елемент вилучений
4. Якщо  $t(a) = \text{вільно}$ , то стоп елемент не знайдено
5.  $i = i + 1$ , перейти до кроку 2

*Пошук:*

1.  $i = 0$
2.  $a = h(\text{key}) + i * c$
3. Якщо  $t(a) = \text{key}$ , то стоп – елемент знайдено
4. Якщо  $t(a) = \text{свободно}$ , то стоп – елемент не знайдено
5.  $i = i + 1$ , перейти до кроку 2

Алгоритм пошуку для хеш-таблиці, яка має три стани, практично не відрізняється від алгоритму пошуку без врахування вилучення. Різниця полягає в тому, що при організації самої таблиці необхідно відмічати вільні і вилучені елементи. Це можна зробити, зарезервувавши два значення ключового поля. Інший варіант реалізації може передбачати введення додаткового поля, в якому фіксується стан елемента. Довжина такого поля може складати всього два біти, що достатньо для фіксації одного з трьох станів.

## 2 Переповнення таблиці і повторне хешування

Очевидно, що в міру заповнення хеш-таблиці будуть відбуватися колізії і в результаті їх розв'язання методами відкритої адресації чергова адреса може вийти за межі адресного простору таблиці. Щоб це явище відбувалося рідше, можна піти на збільшення розмірів таблиці у порівнянні з діапазоном адрес, які обчислюються хеш-функцією.

З однієї сторони це приведе до скорочення кількості колізій і прискоренню роботи з хеш-таблицею, а з іншої – до нераціональних витрат адресного простору. Навіть при збільшенні таблиці в два рази у порівнянні з областю значень хеш-функції нема гарантій того, що в результаті колізій адреса не перевищить розмір таблиці. При цьому в початковій частині таблиця може залишатися достатньо вільних елементів. Тому на практиці використовують циклічний перехід на початок таблиці.

Розглянемо даний спосіб на прикладі методу лінійного випробування. При обчисленні адреси чергового елемента можна обмежити адресу, взявши в якості такої остачу від цілочисельного ділення адреси на довжину таблиці  $N$ .

*Вставка:*

1.  $i = 0$
2.  $a = (h(\text{key}) + c*i) \% N$
3. Якщо  $t(a) = \text{вільно}$  або  $t(a) = \text{вилучено}$  то  $t(a) = \text{key}$ , записати елемент і стоп
4.  $i = i + 1$ , перейти до кроку 2

В даному алгоритмі не враховується можливість багатократного перевищення адресного простору. Більш коректним буде алгоритм, який використовує зсув адреси на 1 елемент у випадку кожного повторного перевищення адресного простору. Це підвищує імовірність знаходження вільного елемента у випадку повторних циклічних переходів до початку таблиці.

*Вставка:*

1.  $i = 0$
2.  $a = ((h(\text{key}) + c*i) / n + (h(\text{key}) + c*i) \% n) \% n$
3. Якщо  $t(a) = \text{вільно}$  або  $t(a) = \text{вилучено}$ , то  $t(a) = \text{key}$ , записати елемент і стоп
4.  $i = i + 1$ , перейти до кроку 2

Розглядаючи можливість виходу за межі адресного простору таблиці, ми не враховували фактори наповненості таблиці й вдалого вибору хеш-функції. При великій наповненості таблиці виникають часті колізії і циклічні переходи на початок таблиці. При невдалому виборі хеш-функції відбуваються аналогічні явища. В найгіршому випадку при повному заповненні таблиці алгоритми циклічного пошуку вільного місця

приведуть до зациклювання. Тому при використанні хеш-таблиць необхідно старатися уникати дуже густого заповнення таблиць. Звичайно довжину таблиці вибирають із розрахунку дворазового перевищення передбачуваної максимальної кількості записів. Не завжди при організації хешування можна правильно оцінити потрібну довжину таблиці, тому у випадку великої наповненості таблиці може знадобитися рехешування. У цьому випадку збільшують довжину таблиці, змінюють хеш-функцію і впорядковують дані.

Проводити окрему оцінку густини заповнення таблиці після кожної операції вставки недоцільно, тому можна проводити таку оцінку непрямым способом – за кількістю колізій під час однієї вставки. Достатньо визначити деякий поріг кількості колізій, при перевищенні якого потрібно провести рехешування. Крім того, така перевірка гарантує неможливість зациклювання алгоритму у випадку повторного перегляду елементів таблиці. Розглянемо алгоритм вставки, який реалізує описаний підхід.

*Вставка:*

1.  $i = 0$
2.  $a = ((h(\text{key}) + c*i) / n + (h(\text{key}) + c*i) \% n) \% n$
3. Якщо  $t(a) = \text{вільно}$  або  $t(a) = \text{вилучено}$ , то  $t(a) = \text{key}$ , записати елемент і стоп
4. Якщо  $i > m$ , то стоп – потрібно рехешування
5.  $i = i + 1$ , перейти до кроку 2

В даному алгоритмі номер ітерації порівнюється з пороговим числом  $m$ . Варто зауважити, що алгоритми вставки, пошуку і вилучення повинні використовувати ідентичне утворення адреси чергового запису.

*Видалення:*

1.  $i = 0$
2.  $a = ((h(\text{key}) + c*i) / n + (h(\text{key}) + c*i) \% n) \% n$
3. Якщо  $t(a) = \text{key}$ , то  $t(a) = \text{вилучено}$  і стоп елемент вилучено
4. Якщо  $t(a) = \text{вільно}$  або  $i > m$ , то стоп – елемент не знайдено
5.  $i = i + 1$ , перейти до кроку 2

*Пошук:*

1.  $i = 0$
2.  $a = ((h(\text{key}) + c*i) / n + (h(\text{key}) + c*i) \% n) \% n$
3. Якщо  $t(a) = \text{key}$ , то стоп – елемент знайдено
4. Якщо  $t(a) = \text{вільно}$  або  $i > m$ , то стоп – елемент не знайдено
5.  $i = i + 1$ , перейти до кроку 2

### 3 Оцінка якості хеш-функції

Як вже було відмічено, дуже важливий правильний вибір хеш-функції. При вда-  
лій побудові хеш-функції таблиця заповнюється більш рівномірно, зменшується кіль-  
кість колізій і зменшується час виконання операцій пошуку, вставки і видалення. Для  
того щоб попередньо оцінити якість хеш-функції можна провести імітаційне моделю-  
вання.

Моделювання проводиться наступним чином. Формується вектор цілих чисел,  
довжина якого співпадає з довжиною хеш-таблиці. Випадково генерується достатньо  
велика кількість ключів, для кожного ключа обчислюється хеш-функція. В елементах  
вектора підраховується кількість генерацій даної адреси. За результатами такого мо-  
делювання можна побудувати графік розподілу значень хеш-функції. Для отримання  
коректних оцінок кількість генерованих ключів повинна в декілька разів перевищу-  
вати довжину таблиці.

Якщо кількість елементів таблиці достатньо велика, то графік будується не для  
окремих адрес, а для груп адрес. Великі нерівномірності засвідчують високу імовір-  
ність колізій в окремих місцях таблиці. Зрозуміло, така оцінка є наближеною, але вона  
дозволяє попередньо оцінити якість хеш-функції і уникнути грубих помилок при її  
побудові.

Оцінка буде більше точною, якщо генеровані ключі будуть більш близькими до  
реальних ключів, які використовуються при заповненні хеш-таблиці. Для символних  
ключів дуже важливо добитися відповідності генерованих кодів символів тим кодам  
символів, які є в реальному ключі. Для цього потрібно проаналізувати, які символи  
можуть бути використані в ключі.

Наприклад, якщо ключ представляє собою прізвище українською мовою, то будуть використані українські букви. Причому перший символ може бути великою буквою, а інші – малими. Якщо ключ представляє собою номерний знак автомобіля, то також нескладно визначити допустимі коди символів в певних позиціях ключа.

Розглянемо більш загальний випадок. Нехай необхідно генерувати ключ із  $m$  символів з кодами в неперервному діапазоні від  $n1$  до  $n2$ .

```
for (i=0; i<m; i++)  
    str[i] := (char) (rand() % (n2-n1) + n1);
```

На практиці можливі варіанти, коли символи в одних позиціях ключа можуть належати до різних діапазонів кодів, причому між цими діапазонами може існувати розрив. Наприклад генерація ключа з  $m$  символів з кодами в діапазоні від  $n1$  до  $n4$  (діапазон має розрив від  $n2$  до  $n3$ ).

```
for (int i=0; i<m; i++)  
{  
    x = rand() % ((n4-n3) + (n2-n1));  
    if ( x <= (n2-n1) )  
        str[i] = (char) (x + n1);  
    else  
        str[i] = (char) (x + n1 + n3 - n2);  
}
```

Розглянемо ще один конкретний приклад. Нехай відомо, що ключ складається з 7 символів. Із них три перші символи – великі латинські букви, далі йдуть дві цифри, інші – малі латинські.

Приклад: довжина ключа 7 символів;  
3 великі латинські (коди 65-90);  
2 цифри (коди 48-57);  
2 малі латинські (коди 97-122).

```
char key[7];  
for (i=0; i<3; i++) key[i] = (char) (rand()%(90-65)+65);  
for (i=3; i<5; i++) key[i] = (char) (rand()%(57-48)+57);  
for (i=5; i<7; i++) key[i] = (char) (rand()%(122-97)+97);
```

#### 4 Організація даних для прискорення пошуку за вторинними ключами

До тепер розглядалися способи пошуку в таблиці за ключами, які дозволяють однозначно ідентифікувати запис. Такі ключі називають первинними ключами. Можливий варіант організації таблиці, при якому окремий ключ не дозволяє однозначно ідентифікувати запис. Така ситуація часто зустрічається в базах даних. Ідентифікація запису здійснюється за деякою сукупністю ключів. Ключі, які не дозволяють однозначно ідентифікувати запис в таблиці, називаються вторинними ключами.

Навіть при наявності первинного ключа, для пошуку запису можуть використовуватися вторинні. Наприклад, пошукові системи Internet часто організовані як набори записів, які відповідають Web-сторінкам. В якості вторинних ключів для пошуку виступають ключові слова, а сама задача пошуку зводиться до вибірки з таблиці деякої множини записів, які містять потрібні вторинні ключі.

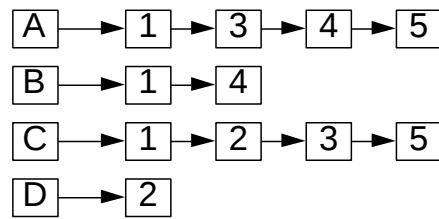
##### ***Інвертовані індекси***

Розглянемо метод організації таблиці з інвертованими індексами. Для таблиці будується окремий набір даних, який містить так звані інвертовані індекси. Допоміжний набір містить для кожного значення вторинного ключа відсортований список адрес записів таблиці, які містять даний ключ.

Пошук здійснюється у допоміжній структурі достатньо швидко, так як фактично відсутня необхідність звернення до основної структури даних. Ділянка пам'яті, яка використовується для індексів, є відносно невеликою у порівнянні з іншими методами організації таблиць.

Таблиця (пряма адресація)

|   |   |   |   |  |
|---|---|---|---|--|
| 1 | A | B | C |  |
| 2 | C | D |   |  |
| 3 | A | C |   |  |
| 4 | A | B |   |  |
| 5 | A | C |   |  |

Структура інвертованих індексів  
(спискова структура)

Недоліками даної системи є великі затрати часу на складання допоміжної структури даних і її поновлення. Причому ці затрати зростають зі збільшенням об'єму бази даних.

Система інвертованих індексів є досить зручною й ефективною при організації пошуку в великих таблицях.

### **Бітові карти**

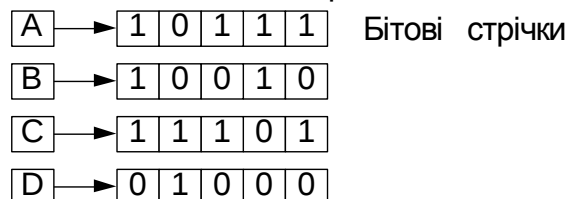
Для таблиць невеликого об'єму використовують організацію допоміжної структури даних у вигляді бітових карт. Для кожного значення вторинного ключа записів основного набору даних записується послідовність бітів. Довжина послідовності бітів рівна кількості записів. Кожен біт в бітовій карті відповідає одному значенню вторинного ключа і одному запису. Одиниця означає наявність ключа в запису, а нуль – відсутність.

Основною перевагою такої організації є дуже проста й ефективна організація обробки складних запитів, які можуть об'єднувати значення ключів різними логічними предикатами. У цьому випадку пошук зводиться до виконання логічних операцій запиту безпосередньо над бітовими стрічками й інтерпретації результуючої бітової стрічки. Іншою перевагою є простота поновлення карти при добавленні записів.

Таблиця (пряма адресація)

|   |   |   |   |  |
|---|---|---|---|--|
| 1 | A | B | C |  |
| 2 | C | D |   |  |
| 3 | A | C |   |  |
| 4 | A | B |   |  |
| 5 | A | C |   |  |

Бітові карти



До недоліків бітових карт варто віднести збільшення довжини стрічки пропорційно довжині файлу. При цьому наповненість карти одиницями зменшується зі збільшенням довжини файлу. Для великої довжини таблиці і ключів, які рідко зустрічаються, бітова карта перетворюється в велику розріджену матрицю, яка складається, в основному, з одних нулів.

### **Питання та завдання для самоконтролю знань студентів**

Використати хеш-таблиці із застосуванням ланцюжків для дозволу колізій (на базі однозв'язних списків).

1. Використовувана хеш-функція - розподіл із залишком. Побудувати статистичний розподіл імовірності  $P(n)$  формування ланцюжків довжини  $n$  для кожного з текстів при  $m = 512, 701, 1024, 1579, 2048$ .

2. Використовувана хеш-функція - множення. Побудувати статистичний розподіл імовірності  $P(n)$  формування ланцюжків довжини  $n$  для кожного з текстів при  $A = 0.23, 0.517, 0.618, 0.85$  й  $m = 2048$ .

3. Використовувана хеш-функція - множення. Побудувати статистичний розподіл імовірності  $P(n)$  формування ланцюжків довжини  $n$  для кожного з текстів при  $A = 0.618$  й  $m = 512, 701, 1024, 1579, 2048$ .

4. Реалізувати дві хеш-функції: розподіл із залишком і множення. Побудувати й зрівняти статистичні розподіли ймовірності  $P(n)$  формування ланцюжків довжини  $n$  для кожного з текстів. Параметри хеш-функцій:

- розподіл із залишком:  $m = 1579$ ;
- множення:  $m = 1579, A = 0.618$ .

# План самостійного вивчення теми № 4

з навчальної дисципліни Алгоритми та структури даних

**Тема** Алгоритми евристики. Співставлення зі зразком.

**Мета** Засвоїти поняття евристичного алгоритму. Ознайомитись з алгоритмічними стратегіями: метод спроб та помилок, пошук з поверненнями, співставлення зі зразком.

*знати:*

- ідею алгоритмічних стратегій: метод спроб та помилок, пошук з поверненнями, співставлення зі зразком.

*вміти:-*

- реалізувати програми рішення задач на основі алгоритмічних стратегій: метод спроб та помилок, пошук з поверненнями, співставлення зі зразком.

Час – 4 години

## Навчальні питання:

- 1 Алгоритми евристики
- 2 Програмування з поверненнями назад
  - 2.1 Метод спроб та помилок
  - 2.2 Реалізація пошуку з поверненням. Задача пошуку шляха в лабіринті
- 3 Співставлення зі зразком

## Методичні рекомендації

Самостійно вивчаючи запропоновану тему, студент повинен засвоїти такі основні положення:

*По-перше* Евристичні алгоритми – це алгоритми, які мають такі властивості:

- 1 Вони дозволяють знайти добрі, хоча і не завжди найкращі розв'язки з усіх, що існують.
- 2 Метод пошуку або побудови розв'язку звичайно значно простіший, ніж той що гарантує оптимальність розв'язку.

*По-друге*

Пошук з поверненням (англ. Backtracking) - загальний метод знаходження рішень задачі, в якій потрібно повний перебір усіх можливих варіантів в деякій множині.

Рішення задачі методом пошуку з поверненням зводиться до послідовного розширення часткового рішення. Якщо на черговому кроці таке розширення провести не вдається, то повертаються до попереднього часткового рішення і продовжують пошук далі. Цей алгоритм дозволяє знайти усі рішення поставленої задачі, якщо вони існують. Для прискорення методу стараються обчислення організувати так, щоб якомога раніше виявляти свідомо непідходящі варіанти

*По-третьє*

Базові концепції алгоритмів співставлення зі зразком здебільшого запропоновані в 1990-х роках. Згодом до них вносилися лише модифікації. Алгоритми співставлення зі зразком поділяють на 2 основні класи в залежності від того, чи зберігають вони відомості про попередню конфліктну множину або обчислюють її на кожному циклі співставлення [10]. Проте даної класифікації недостатньо для формування рекомендацій вибору одного з них.

## **Література**

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротисєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац. ун-т залізн. трансп. ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

## Навчальні матеріали до самостійного вивчення теми

### 1 Алгоритми евристики

У складніших іграх практично неможливо провести пошук навіть в невеликому фрагменту дерева. У цих випадках, можна використовувати різні евристики. Евристикою називається алгоритм або емпіричне правило, яке ймовірно, але не обов'язково дасть добрий результат.

Наприклад, в шахах звичайною евристикою є „посилення переваги”. Якщо у супротивника менше сильних фігур і однакова кількість інших, то слід йти на розмін при кожній нагоді. Зменшення кількості фігур робить дерево рішень коротшим і може збільшити відносну перевагу. Ця стратегія не гарантує виграшу, але підвищує його вірогідність.

Інша евристика, що часто використовується, полягає в присвоєнні різних ваг різним клітинкам. У шахах вага кліток в центрі дошки вища, оскільки фігури, що знаходяться на цих позиціях, можуть атакувати більшу частину дошки. Коли обчислюється вага поточної позиції на дошці, вона може присвоюватися більшою фігурам, які займають клітки в центрі дошки.

Якщо якість рішення не так важлива, то прийнятним може бути результат, отриманий за допомогою евристики. В деяких випадках точність вхідних даних може бути недостатньою. Тоді хороше евристичне рішення може бути таким же правильним, як і теоретично „якнайкраще рішення”.

У попередньому прикладі метод гілок і границь використовувався для вибору інвестиційних можливостей. Проте, вкладення можуть бути ризикованими, і точні результати часто наперед невідомі. Можливо, що наперед буде невідомий точний дохід або навіть вартість деяких інвестицій. В цьому випадку, ефективне евристичне рішення може бути таким же надійним, як і якнайкраще рішення, яке ви може обчислити точно.

Отже, евристичні алгоритми – це алгоритми, які мають такі властивості:

1 Вони дозволяють знайти добрі, хоча і не завжди найкращі розв'язки з усіх, що існують.

2      Метод пошуку або побудови розв'язку звичайно значно простіший, ніж той що гарантує оптимальність розв'язку.

Поняття „добрий розв'язок” змінюється від задачі до задачі, тому його важко визначити точно. Припустимість використання евристики залежить від співвідношення часу та складності пошуку розв'язку обома способами та співвідношення якості обох розв'язків.

У цьому розумінні усі умови, що висуваються до розв'язку, звичайно ділять на дві групи щодо витрат праці:

- ті, які легко задовольнити;
- ті, що вимагають великої роботи.

З іншої сторони, вони поділяються на такі групи щодо їхньої важливості для кінцевої якості:

- ті, які обов'язково слід задовольнити;
- ті, що можуть бути послаблені або змінені.

Цю ситуацію образно показано в таблиці:

|   | 1                 | 2                 |
|---|-------------------|-------------------|
| a | Слід задовольнити | ?                 |
| b | ?                 | Варто відмовитися |

Повернемося до нашого прикладу – задачі про мандрівного крамаря. Якщо міст, про які йдеться – багато, то пошук точно мінімального за довжиною циклу може вимагати дуже багато часу. З іншої сторони, об'їзд слід зробити лише один раз. Якщо витрати на пошук оптимального маршруту можуть виявитися більшими або еквівалентними до витрат на пальне під час подорожі, то, можливо, варто просто рушати в путь, прямуючи до найближчого міста кожного разу.

Якщо ж слід відшукати найкращий маршрут для регулярного об'їзду (наприклад, вибирання пошти зі скриньок, розвезення хлібу з заводу по магазинах, маршрут для руки-маніпулятора робота під час монтування друкованих плат) , то все ж варто відшукати оптимальний маршрут, бо витрати на програмування є разовими, а витрати на зайвий рух є регулярними.

Візьмемо за приклад сортування масиву. Якщо потрібно відсортувати один раз масив зі 100 записів, то можна використати будь-який з простих методів сортування – на весь процес буде витрачено щонайбільше 2-3 секунди машинного часу. Якщо ж розробляють пакет, що буде регулярно сортувати значні масиви інформації, то варто написати сучасну програму.

Частіше за все евристичні алгоритми базуються на методі сходження або на методі частинних цілей.

## 2 Програмування з поверненнями назад

Іноді доводиться мати справу з задачами пошуку оптимального розв'язку, коли неможливо застосувати жоден з відомих алгоритмів, які здатні допомогти відшукати оптимальний варіант розв'язку, і залишається застосувати останній засіб – повний перебір.

Пошук з поверненням (англ. Backtracking) - загальний метод знаходження рішень задач, в якій потрібно повний перебір усіх можливих варіантів в деякій множині. Як правило дозволяє вирішувати завдання, в яких ставляться питання типу, : "Перерахуєте усі можливі варіанти ", "Скільки існує способів ", чи "Є спосіб ", чи "Існує об'єкт". і т. п.

Термін backtrack був введений в 1950 році американським математиком Дерриком Генрі Лемером.

Незначні модифікації методу пошуку з поверненням, пов'язані з представленням даних або особливостями реалізації, мають і інші назви: метод гілок і меж, пошук в глибину, метод проб і помилок і т. д. Пошук з поверненням практично одночасно і незалежно був винайдений багатьма дослідниками ще до його формального опису.

### 2.1 Метод спроб та помилок

Багато задач не допускають аналітичного розв'язку, а тому їх доводиться вирішувати методом спроб та помилок, тобто перебираючи усі можливі варіанти та

відкидаючи їх у випадку невдачі. У разі, якщо побудова розв'язку є складною процедурою, то фактично під час роботи будується дерево можливих кроків алгоритму, а потім – у випадку невдачі – відрізаються відповідні гілки дерева, доки не буде побудовано той шлях, який веде до успіху. Проходження вздовж гілок дерева та відхід у разі невдачі є алгоритм з поверненнями.

Розглянемо гру, в яку грають два гравці, наприклад, шахи, шашки чи „хрестики-нулики”. Гравці по чергово роблять ходи, і стан гри відображається відповідним станом на дошці. Будемо вважати, що є скінчена кількість позицій на дошці і в грі передбачене певне „правило завершення”. З кожною такою грою можна асоціювати дерево, яке називається деревом гри. Кожний вузол такого дерева представляє певну позицію на дошці. Початкова позиція відповідає кореню дерева. Якщо позиція  $x$  асоціюється з вузлом  $n$ , тоді нащадки вузла  $n$  відповідають сукупності допустимих ходів із позиції  $x$ , і з кожним нащадком асоціюється відповідна результуюча позиція на дошці.

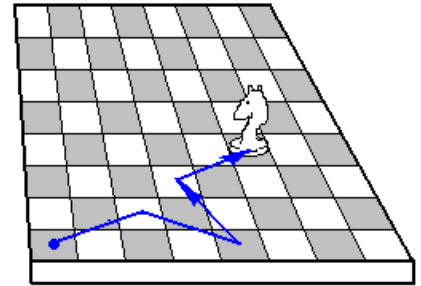
Листки цього дерева відповідають таким позиціям на дошці, з яких неможливо виконати хід, – або тому, що хтось з гравців вже отримав перемогу, або тому, що усі можливі ходи вже вичерпані. Кожному вузлу дерева відповідає певна ціна. На початку назначають ціни листкам. Нехай мова йде про гру „хрестики-нулики”. В такому випадку листкам назначається ціна  $-1$ ,  $0$  і  $1$  в залежності від того, чи відповідає даній позиції програш, нічия або виграш.

Ці ціни розповсюджуються вгору по дереві у відповідності з наступним правилом. Якщо вузол відповідає такій позиції, з якої повинен виконати хід гравець 1, тоді відповідна ціна є максимальним значенням цін нащадків даного вузла. При цьому передбачається, що цей гравець зробить самий вигідний для себе хід, тобто такий, який принесе йому самий цінний результат. Якщо ж вузол відповідає ходу гравця 2, тоді відповідна ціна є мінімальним значенням цін нащадків. При цьому вважається, що гравець два виконає самий вигідний для себе хід, тобто такий, який при самих сприятливих умовах приведе до програшу гравця 1, або до нічиї.

Цей алгоритм покажемо на прикладі задачі, яка має назву „хід коня”.

На шахівниці  $n \times n$  стоїть на полі  $(x, y)$  шаховий кінь. Слід знайти такий маршрут коня (який ходить згідно шахових правил), коли він обходить усю шахівницю, побувавши у кожній клітинці рівно один раз.

Загальна структура алгоритму буде така: на кожному кроці буде аналізуватися, чи можна ще зробити хід куди-небудь. Якщо так, то робимо хід, якщо ні, то повертаємося на один хід назад. Так робимо до тих пір, поки довжина ланцюга ходів не стане рівною  $n-1$ . Тоді це повний „хід коня”.



Загальний вигляд основної функції такого алгоритму:

```
// спроба ще одного ходу
{
    // початкові установки
    do
    {
        // вибір чергового ходу зі списку можливих ходів
        if (хід годиться)
        {
            // запис ходу
            if (дошку не заповнено)
            {
                // спроба ще одного ходу
                if (невдача)
                    // витирання ходу
            }
        }
    }
    while (успішний хід) || (інших ходів немає)
}
```

Ця процедура є рекурсивною: зробивши свій хід, вона звертається до самої себе, передаючи сама собі, природно, нову позицію. Остаточна програма:

```
const n = 8; // Розмір шахівниці
// Масив допустимих ходів
int step[2][n] = {{2,1,-1,-2,-2,-1,1,2},{1,2,2,1,-1,-2,-2,-1}};
// Дошка
int h[n][n] = {{0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0},
               {0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0},
               {0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0}};
bool Try(int, int, int);
void main(void)
{
    bool Success; // Успішний хід
```

```

// Початкові координати коня
int x0 = 0;
int y0 = 0;
h[x0][y0] = 1;
Success = Try(2, x0, y0);
if (!Success)
    cout << " Не вдалося ";
else
    // Вивід результатів
}
bool Try(int i, int x, int y)
{
    bool NextSuccess;
    int u, v;
    int k = 0; // Поточний можливий хід
    do
    {
        NextSuccess = false;
        // Визначення нового кроку за шаховими правилами
        u = x + step[0][k];
        v = y + step[1][k];
        if ( (u>=0) && (u<n) && (v>=0) && (v<n) ) // Хід допустимий?
            if (h[u][v]==0) // Поле не зайняте?
            {
                h[u][v] = i; // запис ходу
                if (i<n*n)
                {
                    NextSuccess = Try(i+1, u, v); // Наступний хід
                    if (!NextSuccess) // Хід неуспішний?
                        h[u][v] = 0; // Витирання ходу
                }
            }
            else
                NextSuccess = true;
        }
        k++;
    }
    while ( !NextSuccess && (k<8) );
    return NextSuccess;
}

```

Характерним для цього прикладу є те, що під час пошуку розв'язку поступово то збільшують  $i$  – номер ходу, записуючи свій маршрут, то зменшують, витираючи ті гілки дерева можливих маршрутів, які не ведуть до успіху, після їх повного обстеження. Ця дія називається поверненням.

## 2.2 Реалізація пошуку з поверненням. Задача про лабіринт.

Рішення задачі методом пошуку з поверненням зводиться до послідовного розширення часткового рішення. Якщо на черговому кроці таке розширення провести не вдається, то повертаються до попереднього часткового рішення і продовжують пошук далі. Цей алгоритм дозволяє знайти усі рішення поставленої задачі, якщо вони існують. Для прискорення методу стараються обчислення організувати так, щоб якомога раніше виявляти свідомо непідходящі варіанти.

Приведемо алгоритм побудови загального алгоритму пошуку з поверненням. Нехай задані правила деякої гри, тобто допустимі ходи і правила її завершення. Потрібно побудувати дерево цієї гри і оцінити його корінь. Це дерево можна побудувати звичним чином, а потім виконати обхід його вузлів у зворотному порядку. Такий обхід у зворотному порядку гарантує, що алгоритм попадає у внутрішній вузол лише після обходу всіх його нащадків, в результаті чого можна оцінити цей вузол, знайшовши максимум або мінімум значень його усіх нащадків.

Об'єм пам'яті для зберігання такого дерева може виявитися достатньо великим, але, якщо дотримуватися мір безпеки, можна обійтися зберіганням в пам'яті в будь-який заданий момент часу лише одного шляху – від кореня до того чи іншого вузла. Приведемо алгоритм пошуку, який виконує обхід дерева за допомогою послідовності рекурсивних викликів. Цей алгоритм передбачає виконання наступних умов:

- 1 Виграші є дійсними числами з обмеженого інтервалу, наприклад  $[-1, 1]$ .
- 2 Константа  $\infty$  більша, ніж будь-який додатній виграш, а  $-\infty$  менша, ніж будь-який від'ємний виграш.
- 3 Дані типу *modetype* (тип режиму) можуть приймати два фіксовані значення – MIN, або MAX.
- 4 Передбачений тип даних *boardtype* (тип ігрової дошки), яка визначається способом представлення позицій на дошці.
- 5 Передбачена функція *payoff* (виграш), яка обчислює виграш для будь-якої позиції, яка є листком дерева.

```
float Search(boardtype B, modetype mode)
```

```

// Оцінює і повертає виграш для позиції В з передбачення,
// що наступним повинен ходити гравець 1 (mode=MAX) або гравець 2
(mode=MIN)
{
    boardtype C; // нащадок позиції В
    float Value; // тимчасове значення виграшу
    if (В є листком) // 1
        return payoff(В); // 2
    else
    {
        if (mode==MAX) // 3
            Value =  $-\infty$ ; // 4
        Else
            Value =  $\infty$ ; // 5
        For (для кожного сина С позиції В) // 6
        {
            if (mode==MAX) // 7
                Value = max(Value, Search(C, MIN)); // 8
            Else
                Value = min(Value, Search(C, MAX)); // 9
            return Value; // 10
        }
    }
}

```

### *Задача пошуку шляху в лабіринті*

Завдання: знайти шлях у лабіринті з початкової крапки (вхід) в кінцеву (вихід).

Нехай лабіринт заданий у вигляді матриці  $A(n \times m)$ , кожен елемент якої говорить, чи є стіна в цій клітці, чи ні. Нехай задані дві крапки: початкова (вхід) і кінцева (вихід) такі, що можна перейти з початкової крапки в кінцеву.

Матрицю  $A(n \times m)$ , що задає лабіринт заповнюємо нулями та одиницями так, що 1 – це стіна, 0 – це прохід. Клітина з координатами  $(i1, j1)$  буде означати вхід до лабіринту, клітина з координатами  $(i2, j2)$  буде означати вихід з лабіринту. Шлях проходу відмічатимемо цифрами, починаючи з 2 (2, 3, 4 і так далі).

Будемо використовувати рекурсивну процедуру move переміщення по лабіринту. Нехай на  $k$ -том кроці ми потрапили на клітину з координатами  $(i, j)$ . Якщо це та клітина, шлях до якої необхідно було відшукати (з координатами  $(i2, j2)$ ), то завдання вирішене. Необхідно роздрукувати матрицю з відміченим шляхом і припинити вико-

нання програми. Якщо ні, то необхідно перевірити, чи можна перейти на сусідню клітину (1 – справа, 2 – знизу, 3 – ліворуч, 4 – згори), і якщо можливо, то викликати процедуру переміщення вже з цієї клітини. Якщо переходу на сусідню клітину немає, то необхідно очистити початкову клітину, щоб її можна було використовувати в інших варіантах при пошуку шляху. Позначимо:

`move(i, j, k)` - процедуру рекурсивного переміщення (`i, j` - номер клітини, `k` - номер кроку);

`writematr` - процедуру, що роздруковує матрицю `A`

Тоді програма, що реалізовує запропонований алгоритм, може бути записана таким чином:

```
const n=4; m=5;
type matr=array [1..n, 1..m] of integer;
{Матриця, задаюча варіанти проходу. 1-стена; 0-проход.}
const labir: matr=((1,0,0,0,1)
                    (0,0,1,0,1),
                    (1,0,0,0,0),
                    (0,0,1,0,0));

var a: matr;
    i1, j1, i2, j2, f: byte;
    k: integer;
procedure writematr;
var i, j: byte;
begin
  for i:=1 to n do
    begin
      for j:=1 to m do write(a[i, j]:3' ');
      writeln;
    end
  end;
procedure move(i, j: byte; k: integer);
begin
  a[i, j]:=k; k:=k+1;
  if (i=i2) and(j=j2) then begin writematr; f:=1; halt end
  else
  begin
```

```

        if (i+1<=n) and(a[i+1, j]=0) then move(i+1, j, k);
        if (j+1<=m) and(a[i, j+1]=0) then move(i, j+1, k);
        if (j - 1>0) and(a[i, j - 1]=0) then move(i, j - 1, k);
        if (i - 1>0) and(a[i - 1, j]=0) then move(i - 1, j, k);
    end;
    a[i, j]:=0; k:=k - 1;
end;
begin
    a:=labir;
    writeln('Координати i1, j1'); readln(i1, j1);
    writeln('Координати i2, j2'); readln(i2, j2);
    f:=0; k:=2;
    if(a[i1, j1]=1) or(a[i2, j2]=1) then
        writeln ('Невірні координати')
    else move(i1, j1, k);
    if f=0 then writeln('Проходу немає');
end.

```

Тут - labir - матриця-константа 4x5, задаюча приклад лабіринту, f - змінна, через яку відстежується, чи є прохід з початкової точки в кінцеву або ні, процедура halt - стандартна процедура, яка припиняє виконання програми.

Якщо взяти за початкову точку точку з координатами (4, 1), як кінцева - точку з координатами (1, 4), то розглянута програма видасть наступний варіант проходу :

1 0 0 8 1

0 0 1 7 1

1 4 5 6 0

2 3 1 0 0

Постановка завдання "Співставлення зі зразком" може бути сформульована на прикладі обробки рядків у такий спосіб:

Дано зразок (рядок) і вхідний рядок . Потрібно визначити індекс, починаючи з якого зразок утримується в рядку . Якщо зразок не втримується - повернути індекс, що не може бути інтерпретований як позиція в рядку (наприклад, негативне число).

Пошук підстроки у довгому фрагменті тексту - важливий елемент текстових редакторів. Однак ту ж саму техніку можна використати для пошуку бітових або байтових рядків у двійковому файлі. Так, наприклад, здійснюється пошук вірусів у пам'яті комп'ютера. У програмах обробки текстів звичайно є функція перевірки синтаксису, що не тільки виявляє неправильно написані слова, але й пропонує варіанти їхнього правильного написання. Один з підходів до перевірки складається в складанні відсортованого списку слів документа. Потім цей список рівняється зі словами, записаними в системному словнику й словнику користувача; слова, відсутні в словниках, позначаються як можливо невірно написані.

За умовою завдання потрібно знайти тільки перше входження деякої підстроки в довгому тексті. Пошук наступних входжень заснований на тім же підході (має сенс завести додаткову функцію, викликувану при кожному виявленні зразка).

Стандартний алгоритм починає з порівняння першого символу тексту з першим символом підстроки. Якщо вони збігаються, то відбувається перехід до другого символу тексту й підстроки. При збігу рівняються наступні символи. Так триває доти, поки не виявиться, що підстрока цілком збіглася з відрізком тексту, або поки не зустрінуться незбіжні символи. У першому випадку завдання вирішене, у другому ми зрушуємо покажчик поточного положення в тексті на один символ і заново починаємо порівняння з підстрокою.

Процес використання стандартного алгоритму на базі пошуку зразка "they" у тексті "there they are" наведений у наступній таблиці:

|             |                |                                                                    |
|-------------|----------------|--------------------------------------------------------------------|
| 1<br>проход | THERE THEY ARE | Три первых символа совпадают, а четвертый<br>- нет                 |
|             | THEY           |                                                                    |
| 2<br>проход | THERE THEY ARE | Сдвиг указателя в тексте<br>Указатель подстроки – в начале         |
|             | THEY           |                                                                    |
| 3<br>проход | THERE THEY ARE | Сдвиг указателя в тексте<br>Указатель подстроки – в начале         |
|             | THEY           |                                                                    |
| 4<br>проход | THERE THEY ARE | Сдвиг указателя в тексте<br>Указатель подстроки – в начале         |
|             | THEY           |                                                                    |
| 5<br>проход | THERE THEY ARE | Сдвиг указателя в тексте<br>Указатель подстроки – в начале         |
|             | THEY           |                                                                    |
| 6<br>проход | THERE THEY ARE | Сдвиг указателя в тексте<br>Указатель подстроки – в начале         |
|             | THEY           |                                                                    |
| 7<br>проход | THERE THEY ARE | Сдвиг указателя в тексте и сдвиг в тексте<br>до полного совпадения |
|             | THEY           |                                                                    |

При першому проході три перших символи підстроки збігаються із символами тексту. Однак тільки сьомий прохід дає повний збіг. З алгоритму ясно, що основна операція - порівняння символів, і саме число порівнянь варто підраховувати. Якщо підрахувати кількість порівнянь, то їх буде 13. У найгіршому випадку при кожному проході збігаються всі символи за винятком останнього.

Опис алгоритму можна представити у вигляді:

Позначення даних:

text - вихідний рядок

substring - задана підстрока

textLoc - покажчик поточного порівнюваного символу в тексті

subLoc покажчик поточного порівнюваного символу в підстроке

textStart покажчик на початок порівняння в тексті

Наступний алгоритм здійснює стандартне порівняння рядків:

```

subLoc=1 //покажчик текущ. порівнюваного символу в підстроке
textLoc=1 //покажчик поточного порівнюваного символу в тексті
textStart=1 //покажчик на початок порівняння в тексті
//поки текст не закінчений і не закінчені символи в підстроке
while TextLoc<=length(text) and subLoc<=length(substring) do
  if text[textLoc]=substring[subLoc] then //якщо збіг символів
    begin
      textLoc=textLoc+1 //покажчик у тексті збільшують
      subLoc=subLoc+1 //покажчик у підстроке збільшують
    end
  end

```

```

else // почати порівняння заново з наступного символу
begin
textStart=textStart+1 // покажчик на початок порівняння в
//тексті збільшують
textLoc= textLoc+1//покажчик у тексті збільшують
subLoc= 1// покажчик у підстроке встановлюють у початок
end
end if
end while
if (subLoc > length(substring)) then
return textStart // збіг знайдений
else
return 0 // збіг не знайдений
end if

```

### *Ідея алгоритму Бойера-Мура*

Проблема стандартного алгоритму полягає в тому, що він затрачає багато зусиль (багато порівнянь) впусту. Зменшити цей недолік допомагає алгоритм Бойера-Мура. Він здійснює порівняння зі зразком праворуч ліворуч, а не ліворуч праворуч. Досліджуючи шуканий зразок, можна здійснювати більше ефективні стрибки в тексті при виявленні розбіжності.

У прикладі ми спочатку порівнюємо в с г і виявляємо розбіжність. Оскільки ми знаємо, що буква г взагалі не входить у зразок, ми можемо зрушитися в тексті на цілих чотири букви (тобто на довжину зразка) вправо. Потім ми порівнюємо букву в с h і знову виявляємо розбіжність. Однак оскільки цього разу h входить у зразок, ми можемо зрушитися вправо тільки на дві букви так, щоб букви h збіглися. Потім ми починаємо порівняння праворуч і виявляємо повний збіг шматочка тексту зі зразком. В алгоритмі Бойера-Мура робиться 6 порівнянь замість 13 порівнянь у вихідному простому алгоритмі.

Крім описаних алгоритмів поширення одержали такі, як: алгоритм Кнута-Морриса-Пратта; статистичні методи.

## **Питання та завдання для самоконтролю знань студентів**

1. Що таке евристичні методи та в чому їх перевага?
2. В чому полягає метод спроб та помилок?
3. В чому полягає підхід до розв'язання задачі „хід коня”?
4. Дайте характеристику алгоритмічної стратегії перебір з поверненнями.
5. В чому полягає ідея алгоритму рішення задачі пошуку шляху в лабіринті?
6. Сформулюйте ідею методу співставлення зі зразком.
7. Опишіть алгоритм пошуку рязка (зразка) в тексті.

# План самостійного вивчення теми № 5

з навчальної дисципліни Алгоритми та структури даних

**Тема** Динамічне програмування. Задача триангуляції.

**Мета** Ознайомитись з основними ідеями алгоритмічних стратегій динамічного програмування.

*знати:-*

- ідею та особливості методу динамічного програмування;
- постановку та алгоритмічний метод розв'язання задачі триангуляції багатокутника.

*вміти:-*

- застосовувати методи динамічного програмування при розробці програм.

Час – 4 години

## Навчальні питання:

- 1 Ідея методу динамічного програмування
- 2 Використання рекурентних співвідношень
- 3 Оптимальна триангуляція багатокутника

## Методичні рекомендації

Самостійно вивчаючи запропоновану тему, студент повинен засвоїти такі основні положення:

*По-перше*

Динамічне програмування є одним з методів рішення завдань, у яких завдання великої розмірності можна вирішувати, опираючись на вже вирішені завдання меншого розміру.

Динамічне програмування може бути застосоване при виконанні наступних умов:

1) Існує спосіб виразити рішення завдання через рішення підзадач меншої розмірності. Цей спосіб задається рекуррентними співвідношеннями.

2) Існують відомі рішення для завдання малої розмірності (завдання малої розмірності вирішується просто).

- По-друге* Рекурентні співвідношення при використанні динамічного програмування зв'язують рішення завдання з рішеннями підзадач меншої розмірності.
- По-третє* Тріангуляція багатокутника – це набір діагоналей, що розріжуть багатокутник на трикутники; сторонами цих трикутників є сторони вихідного багатокутника й діагоналі тріангуляції. Завдання про оптимальну тріангуляцію Даний багатокутник  $P=(v_0, \dots, v_{n-1})$  і вагова функція  $w$ , певна на множині трикутників. Потрібно знайти тріангуляцію, для якої сума ваг трикутників буде найменшою  $w(v_i, v_j, v_k)=|v_i v_j|+|v_j v_k|+|v_k v_i|$

## Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротиєєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп .ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

## Навчальні матеріали до самостійного вивчення теми

### 1 Ідея методу динамічного програмування

Динамічне програмування є одним з методів рішення завдань, у яких завдання великої розмірності можна вирішувати, опираючись на вже вирішені завдання меншого розміру.

Словосполучення "динамічне програмування" уперше було використано в 1940-х роках Р. Беллманом для опису процесу знаходження рішення завдання, де відповідь на одне завдання може бути отриманий тільки після рішення завдання, "попередньої" їй.

Динамічне програмування може бути застосоване при виконанні наступних умов:

1) Існує спосіб виразити рішення завдання через рішення підзадач меншої розмірності. Цей спосіб задається рекуррентними співвідношеннями.

2) Існують відомі рішення для завдання малої розмірності (завдання малої розмірності вирішується просто).

Динамічне програмування обчислює рішення для всіх підзадач. Обчислення йде від малих підзадач до більшого. Рішення підзадач запам'ятовуються в таблиці. Ми заповнюємо цю таблицю незалежно від того, чи потрібна нам насправді конкретна підзадача для одержання загального рішення.

Перевага методу полягає в тому, що раз вже завдання вирішене, її відповідь зберігається й ніколи не обчислюється заново.

Динамічне програмування звичайно дотримується двох підходів до рішення завдань:

— спадне динамічне програмування: завдання розбивається на підзадачі меншого розміру, вони вирішуються й потім комбінуються для рішення вихідного завдання. Використається запам'ятовування для рішень підзадач, які часту зустрічаються.

– висхідне динамічне програмування: всі підзадачі, які згодом знадобляться для рішення вихідного завдання прораховуються раніше й потім використовуються для побудови рішення вихідного завдання.

Основні властивості завдань динамічного програмування:

1. Завдання має властивість оптимальності.
2. Невелике число різних підзадач.

Процес розробки алгоритмів динамічного програмування можна розбити на чотири етапи.

1. Опис структури оптимального рішення.
2. Рекурсивне визначення значення, що відповідає оптимальному рішення.
3. Обчислення значення, що відповідає оптимальному рішення, з помістю методу висхідного аналізу.
4. Складання оптимального рішення на основі інформації, отриманий<sup>^</sup>-ний на попередніх етапах.

Можна виділити два класи завдань, розв'язуваних методом динамічного програмування.

Завдання з першого класу пов'язані з обчисленням сум або добутків залежно від постановки завдання.

До другого класу ставляться оптимизационные завдання, для яких може бути сформульований принцип оптимальності; суть принципу оптимальності полягає в тому, що рішення підзадач, використовувані для знаходження оптимального рішення завдання, також повинні бути оптимальними.

Ми розглянемо на прикладах використання методу динамічного програмування для завдань із обох класів.

## 2 Використання рекурентних співвідношень

Рекурентні співвідношення при використанні динамічного програмування зв'язують рішення завдання з рішеннями підзадач меншої розмірності. Приведемо два приклади завдань, для яких можна записати такі рекурентные співвідношення.

*Задача про число сполучень.*

Для n-елементної множини сполученням з n елементів по k називається довільна підмножина з k елементів. Наприклад, для множини  $V=\{a,b,c,d\}$  його підмножина  $\{b,d\}$  - це сполучення з 4-х елементів по 2.

Позначення застосовується для числа всіх сполучень із n елементів по k. Так, для множини V множина всіх двоелементних підмножин дорівнює  $\{\{a,b\}, \{a,c\}, \{a,d\}, \{b,c\}, \{b,d\}, \{c,d\}\}$ .

Кількість сполучень із n елементів по k обчислюється за формулою:

$$C_n^m = \frac{n!}{(n-m)! m!}.$$

Помітимо, що це - швидко зростаюча функція, тому для обчислень по цій формулі потрібна арифметика "великих чисел".

*Числа Фібоначчі.*

Послідовність Фібоначчі  $F_n$  задається формулами:

$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$  при  $n > 1$ .

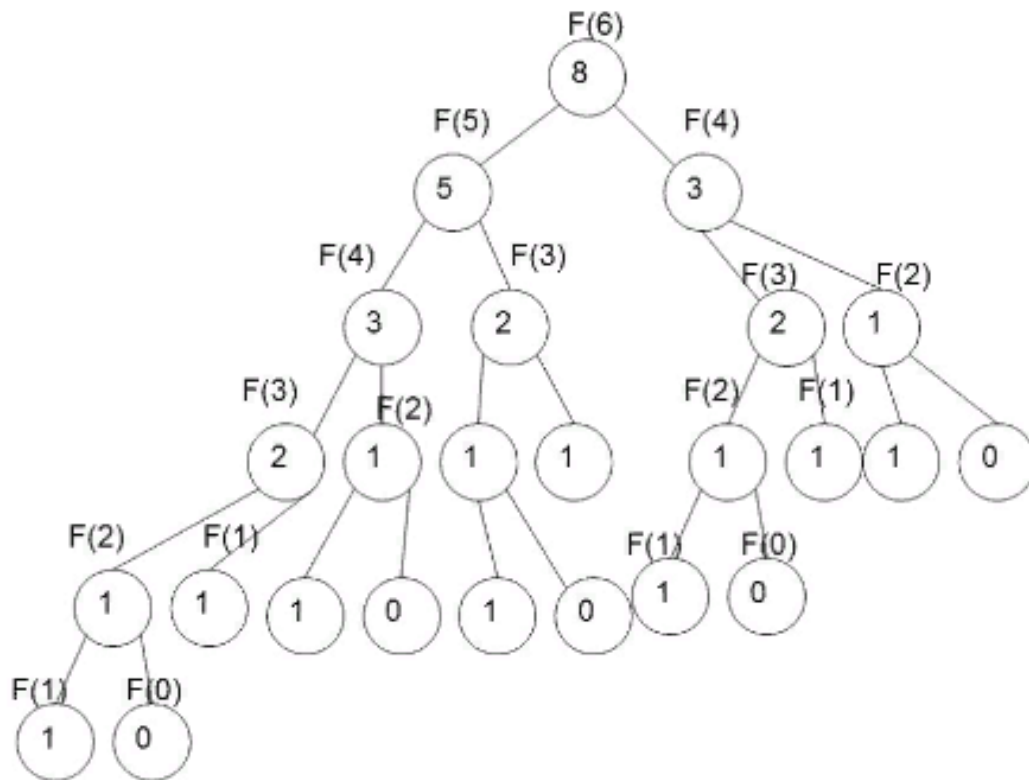
Необхідно знайти  $F_n$  по номері n.

– Рекурсивний метод рішення:

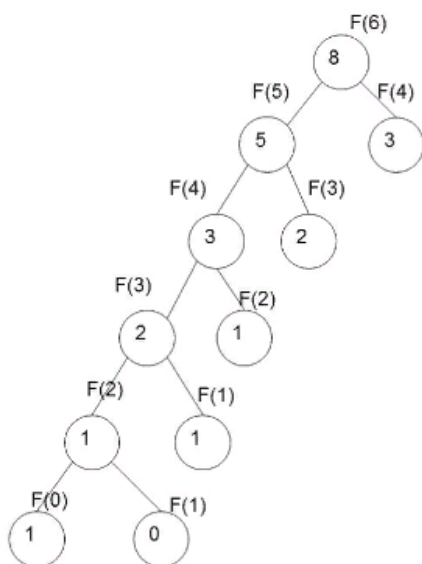
```
int F(int n)
{
    if (n == 0) return 0;
    if (n == 1) return 1;
    else return F(n - 1) + F(n - 2);
}
```

– Нисхідне динамічне програмування

```
int F(int n)
{
    if (A[n] != -1) return A[n];
    if (n == 0) return 0;
    if (n == 1) return 1;
    else { A[n] = F(n - 1) + F(n - 2); return A[n]; }
}
```



– Висхідне динамічне програмування



$F[0] = 0;$

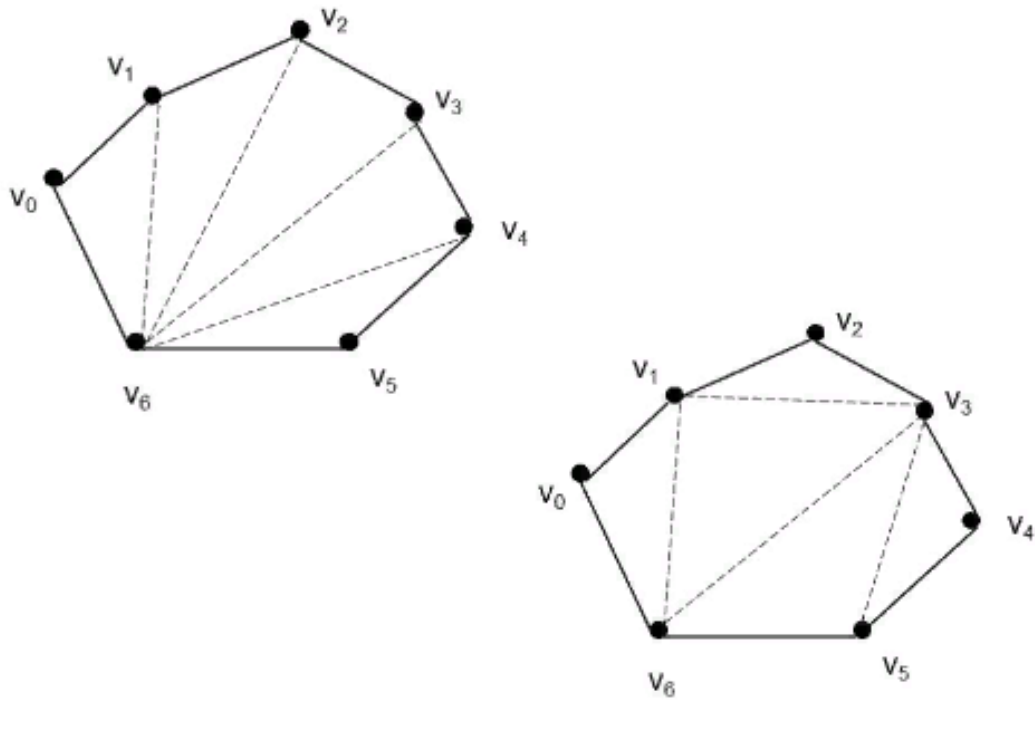
$F[1] = 1;$

$\text{for } (i=2; i < n; i++) F[i] = F[i-1] + F[i-2];$

### 3 Оптимальна триангуляція багатокутника

Триангуляція багатокутника – це набір діагоналей, що розріжуть багатокутник на трикутники; сторонами цих трикутників є сторони вихідного багатокутника й діагоналі триангуляції. Завдання про оптимальну триангуляцію Даний багатокутник

$P=(v_0, \dots, v_{n-1})$  і вагова функція  $w$ , певна на множині трикутників. Потрібно знайти триангуляцію, для якої сума ваг трикутників буде найменшою  $w(v_i, v_j, v_k)=|v_i v_j|+|v_j v_k|+|v_k v_i|$



Нехай  $T$  - оптимальна триангуляція  $n+1$ -кутника  $P=(v_0, \dots, v_n)$ . Ребро  $v_0 v_n$  входить в один із трикутників триангуляції. Нехай це трикутник  $(v_0 v_k v_n)$ , де  $1 \leq k \leq n-1$ . Тоді вага триангуляції  $T$  дорівнює ваги трикутника  $(v_0 v_k v_n)$  + сума ваг триангуляцій багатокутників  $(v_0, v_1 \dots v_k)$  і  $(v_k, v_{k+1}, \dots, v_n)$ . Триангуляції зазначених багатокутників повинні бути оптимальними. Якщо оптимальні вартості для таких багатокутників (при різних  $k$ ) уже обчислені, то залишається вибрати оптимальне  $k$ .

$m[i, j]$  - вага оптимальної триангуляції багатокутника  $(v_{i-1}, v_i \dots v_j)$  Вага оптимальної триангуляції всього багатокутника дорівнює  $m[1, n]$ .

Рекуррентная формула:

$$m[i, j] = \begin{cases} 0, & i = j \\ \min_{i \leq k < j} (m[i, k] + m[k+1, j] + w(v_{i-1} v_k v_j)), & i < j \end{cases}$$

Реалізація алгоритму оптимальної триангуляції багатокутника.

Що потрібно:

Клас, щось на зразок списку, де можна рухатися назад і кінець з'єднаний з початком. Тобто замкнуте коло, елементами якого будуть об'єкти описані пунктом нижче.

Клас для подання крапки. У ньому, як покладається, повинні бути координати  $x$  і  $y$ . Так само ще одне поле в якому записане значення кута відповідній цій крапці в багатокутнику

Функція, на вхід якої йдуть два вектори, а на вихід - кут між ними

Функція, на вхід якої йде крапка й трикутник, на вихід - ознака, чи лежить крапка усередині трикутника.

Тепер сам алгоритм.

Підготовка робочих об'єктів.

Результатом роботи повинен бути список трикутників (`result`), тому створюємо порожній список - двунаправлений замкнутий список (`points`), що представляє багатокутник.

Перед стартом прораховуємо кути для всіх крапок багатокутника.

Вибираємо будь-яку крапку багатокутника як "робочу" ( $p(i)$ ).

Створюємо порожній список для зберігання тимчасових трикутників.

Якщо крапка ліворуч від "робітник" ( $p(i) \rightarrow \text{left}$ ) має кут менше ніж 180 градусів і трикутник ( $p(i), p(i) \rightarrow \text{left}, p(i) \rightarrow \text{left} \rightarrow \text{left}$ ) не містить усередині себе інших крапок багатокутника - заносимо цей трикутник у наш тимчасовий список.

Якщо крапка праворуч від "робітник" ( $p(i) \rightarrow \text{right}$ ) має кут менше ніж 180 градусів і трикутник ( $p(i), p(i) \rightarrow \text{right}, p(i) \rightarrow \text{right} \rightarrow \text{right}$ ) не містить усередині себе інших крапок багатокутника - заносимо цей трикутник у наш тимчасовий список.

Якщо "робоча" крапка ( $p(i)$ ) має кут менше ніж 180 градусів і трикутник ( $p(i) \rightarrow \text{left}, p(i), p(i) \rightarrow \text{right}$ ) не містить усередині себе інших крапок багатокутника - заносимо цей трикутник у наш тимчасовий список.

Якщо тимчасовий список не містить трикутників - вибираємо замість "робітник", крапку ліворуч від її й повертаємося до першого пункту.

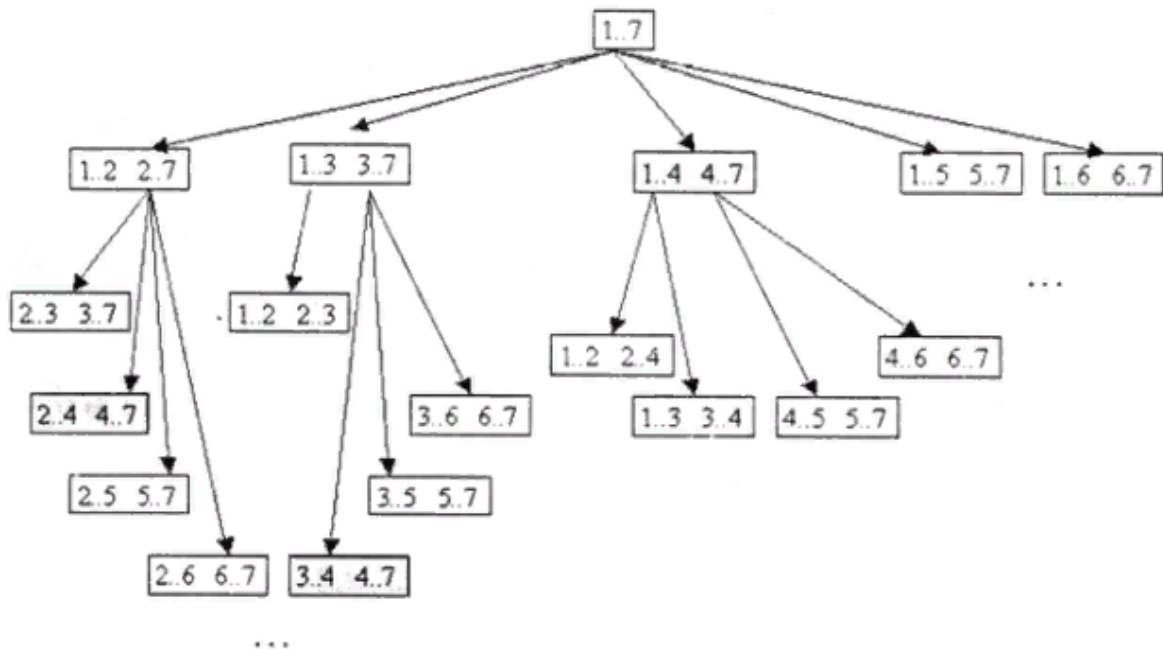
Якщо містить - вибираємо трикутник з мінімальною різницею між мінімальним і максимальним кутом (потрібно перерахувати значення кутів), заносимо його в список `result`, видаляємо з `points` середню крапку із трикутника який вибрали а сусіднім крапкам від її (в `points`) перераховуємо значення кутів, першу крапку вибираємо в якості "робочої" ( $p(i)$ ). Якщо в `points` залишилося тільки дві крапки - припиняємо роботу, список трикутників утримується в `res`, інакше повертаємося до першого пункту.

Тепер пари слів про оптимізацію алгоритму.

На другому етапі вибирається трикутник з мінімальною різницею між мінімальним і максимальним кутом для того, щоб трикутник був максимально подібний до правильного, іноді це важливо. Якщо ж вам немає різниці як виглядає трикутник - тоді можна не створювати тимчасовий список трикутників, а вибрати перший із трьох можливих трикутників який не містить усередині себе іншу крапку багатокутника й кут, що утворить середня крапка трикутника в багатокутнику менше 180 градусів. Таке спрощення значно знизить обчислювальні витрати.

Ще, якщо впевнені, що багатокутник опуклий - тоді не потрібно перевіряти чи містить трикутник інші крапки багатокутника.

Покажемо за допомогою дерева, як будемо збирати цілісне рішення з рішень підзадач для семикутника. Завдання знайти  $f(1..7)$  розпалося на 5 пар завдань  $(f(1..k), f(k..7))$  для  $k=2..5$ . Далі кожна з підзадач рекурсивно розбивається на більше дрібні, поки останні не стануть тривіальними (виду  $f(i..i+1)$  або  $f(i..i)$ ). Якщо придивитися до цього дерева, видно, що ті самі підзадачі на різних гілках дерева повторюються. Більше того, повторюються не тільки тривіальні підзадачі (наприклад,  $f(2..3)$  на блакитних вузлах), але й не- тривіальні (завдання  $f(4..7)$  на зелених вузлах), які породжують цілі дерева рекурсивних викликів.



Динамічне програмування полягає саме в тім, що підзадачі потрібно вирішувати тільки один раз, а при кожному наступному виклику рекурсивної процедури для рішення тієї ж підзадачі потрібно просто прочитати дані з таблиці. Так ми й надійдемо при написанні коду.

```

const
InFileName = 'in.txt'; MaxN = 50; NotCalculated = -1; MaxReal = 1e30;
var
N: integer; {кількість вершин}
d: array[1..MaxN, 1..MaxN] of real; {відстані між вершинами}
f: array[1..MaxN, 1..MaxN] of real; {динамічна таблиця рішень}
Cut: array[1..MaxN, 1..MaxN] of integer; {таблиця для запам'ятовування
вершини розрізу}
{ функція знаходження відстані між двома крапками }
function distance(x1,y1,x2,y2: real):real;
begin
distance := sqrt(sqr(x1-x2)+sqr(y1-y2));
end;
{процедура зчитування вхідних даних і знаходження відстаней між верши-
нами}
procedure ReadData; var
ft: text; x,y: array[1..MaxN] of real;
i,j: integer;
begin

```

```

assign(ft,InFileName); reset(ft); read(ft,N); for i:= 1 to N do
readln(ft, x[1],y[i]);
close(ft);
for i:= 1 to N do
for j:= 1 to д do
begin
if abs(i-j) <= 1 then
{якщо це сторона, то} d[i,j]:=0 {у розріз вона не ввійде,значить нехай
її довжина=0}
else d[i,j]:=Distance(x[i],y[1],x[2],y[2]);
f [i,j]:= NotCalculated;
{ ставимо оцінку, що підзадача f(i,.j) не решена }
end;
end;
{рекурсивна процедура пошука оптимального рішення}
procedure Divide(i,j:integer); var k, minK: integer;
L, minL: real;
begin
if f[i,j] <> NotCalculated then
{ якщо підзадача вже вирішувалася, виходимо } exit;
if i+2 >= j then begin
{якщо завдання тривіальне (трикутник, відрізок або крапка)}
Cut[i,j] := 0; {тої розбивок немає}
f [1,2] := 0; {і довжина такої неіснуючої розбивки дорівнює нулю}
exit;
end;
minL := MaxReal;
{ шукаємо мінімум по k (рекурентна формула)}
for k := 1<1 to j-1 do begin
Divide(i,k); {вирішуємо підзадачу f(i..k)}
Divide(k,j); {вирішуємо підзадачу f(k..2)}
L := P[i,k] + F[k,j] + D[i,k] + D[k,j];
if L < MinL then begin
MinL := L; {запам'ятовуємо мінімальну довжину }
MinK := k; { і крапку розбивки }
end;
end;
end;

```

```

F[i,j]:= MinL; { запам'ятовуємо оптимум у динамічній таблиці }
Cut[i,j] :=MinK;
end;
{процедура рекурсивно виводить розбивку }
procedure PrintCut(i,j integer) begin
if Cut[i,j] <> 0 then begin
if i+1 < Cut[i,j] then
WriteLn(i,'-',Cut[i,j]);
if Cut[i,j]+1 < j then
WriteLn(Cut[i,j],'-',3);
PrintCut(i,Cut[1 3]);
PrintCut(Cut[i,j],j);
end;
end;
begin
ReadData; { читаємо вхідні дані }
Divide(1,N); { виконуємо розбивку } { виводимо результати }
  wreteln('Min(mal cut length:f[1,n]);
writeln('Cut the polygon along these diagonals:');
PrintCut(1,N);
end.

```

## **Питання та завдання для самоконтролю знань студентів**

1. Сформулюйте завдання динамічного програмування та їх особливості.
2. Які підходи застосовні до рішення завдань динамічного програмування?
3. Опишіть етапи процесу розробки алгоритмів динамічного програмування.
4. В чому полягає ідея застосування рекурентних співвідношень при розв'язанні задач динамічного програмування?
5. Сформулюйте постановку задачі триангуляції багаторуктника.
6. Опишіть алгоритм розв'язання задачі триангуляції багатокутника.

# План самостійного вивчення теми № 6

з навчальної дисципліни Алгоритми та структури даних

**Тема** Алгоритми чисельної апроксимації.

**Мета** Сформулювати постановку задачі чисельної апроксимації функцій та освітити ідею алгоритмічного методу найменших квадратів.

*знати:*

- знати постановку задачі чисельної апроксимації функцій;
- ідею алгоритмічного методу найменших квадратів.

*вміти:*

- використовуючи метод найменших квадратів, знаходити емпіричні функції лінійної та квадратичної апроксимації;
- писати програмний код, що реалізує метод найменших квадратів для лінійної моделі.

Час – 4 години

## Навчальні питання:

1 Постановка задачі чисельної апроксимації

2 Метод найменших квадратів

2.1 Лінійна апроксимація

2.1.Квадратична апроксимація

## Методичні рекомендації

Самостійно вивчаючи запропоновану тему, студент повинен засвоїти такі основні положення:

*По-перше* Завдання про наближення (апроксимацію) функції полягають у тому, щоб дану функцію  $f(x)$  приблизно замінити (апроксимувати) деякою функцією  $\varphi(x)$ , значення якої в заданій області незначним чином відмінні від дослідних даних -  $f(x) \approx \varphi(x)$ . Методи рішення такого завдання відносяться до категорії *чисельних методів або методів обчислювальної математики*.

*По-друге* При інтерполяції від наближення потрібно, щоб вона мала ту ж таблицю значень, що й наближувана функція:  
$$\varphi(x_i) = f(x_i), i = 0, 1, 2, \dots, n.$$

Ця умова називається умовою *інтерполяції*. Функція, що задовольняє умовам інтерполяції, називається інтерполяційною, а крапки  $x_0, x_1, x_2, \dots, x_n$  *вузлами інтерполяції*.

Найчастіше в якості інтерполяційної функції вибирають алгебраїчні багаточлени, тому що їхні значення обчислюються простіше всього. Таким чином, вирішується наступне завдання – визначається алгебраїчний багаточлен  $n$ -го ступеня:

$$P_n(x_i) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n, \quad (1)$$

що задовольняє умовам інтерполяції:

$$P_n(x_i) = f(x_i), \quad i = 0, 1, 2, \dots, n.$$

*По-третє*

Мірою відхилення багаточлена  $\varphi(x)$  від заданої функції  $f(x)$  на множині крапок  $(x_i, y_i)$  є величина  $S$ , рівна сумі квадратів різниці між значеннями багаточлена й функції для всіх крапок  $x_1, x_2, \dots, x_n$ :

$$S = \sum_{i=1}^n \varepsilon_i^2 = \sum_{i=1}^n [\varphi(x_i) - y_i]^2 \rightarrow \min.$$

## Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротисєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп .ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

## Навчальні матеріали до самостійного вивчення теми

### 1 Постановка задачі чисельної апроксимації

У практиці відомі 3 способи завдання функції: аналітичний, графічний, табличний. В інженерній практиці найпоширенішим є випадок, коли вид зв'язку між параметрами  $X$  й  $Y$  невідомий, тобто неможливо записати цей зв'язок у вигляді деякої залежності  $y = f(x)$ . Як правило, навіть при відомій залежності  $y = f(x)$ , вона настільки громіздка, що її використання в практичних розрахунках недоречно. Найчастіше цей зв'язок представлений у вигляді таблиці, тобто дискретній множині значень аргумента  $\{x_i\}$  поставлений у відповідність множині значень функції  $\{y_i\} (i = 1, 2, \dots, n)$ . Ці значення - або результати розрахунків, або експериментальні дані. На практиці нам можуть знадобитися значення величини  $y$  й в інших крапках, відмінних від вузлів  $x_i$ . Часто ці значення можна одержати лише шляхом складних розрахунків або проведенням дорогих експериментів. Таким чином, необхідно використати наявні табличні дані для наближеного обчислення шуканого параметра  $y$  при будь-якому значенні (з деякої області) визначального параметра  $x$ , оскільки точний зв'язок  $y = f(x)$  невідомий. Завдання дослідження в більшості випадків вимагають установити певний вид функціональної залежності між характеристиками досліджуваного явища. Цієї мети й служить завдання про наближення функції. Тобто завдання про наближення (апроксимацію) функції полягають у тому, щоб дану функцію  $f(x)$  приблизно замінити (апроксимувати) деякою функцією  $\varphi(x)$ , значення якої в заданій області незначним чином відмінні від дослідних даних -  $f(x) \approx \varphi(x)$ . Методи рішення такого завдання відносяться до категорії *чисельних методів або методів обчислювальної математики*. Один зі способів апроксимації функцій - *інтерполяція*. Він використовується в тих випадках, коли основна інформація про наближення функції дається у вигляді таблиці її значень. У результаті рішення завдання інтерполяції лінія, що відповідає функції, яка інтерполюється, буде обов'язково проходити через всі крапки вихідних даних. У цьому випадку крапки є *узлами інтерполяції*.

При інтерполяції від наближення потрібно, щоб вона мала ту ж таблицю значень, що й наближувана функція:

$$\varphi(x_i) = f(x_i), i = 0, 1, 2, \dots, n.$$

Ця умова називається умовою *інтерполяції*. Функція, що задовольняє умовам інтерполяції, називається інтерполяційною, а крапки  $x_0, x_1, x_2, \dots, x_n$  *вузлами інтерполяції*.

Найчастіше в якості інтерполяційної функції вибирають алгебраїчні багаточлени, тому що їхні значення обчислюються простіше всього. Таким чином, вирішується наступне завдання – визначається алгебраїчний багаточлен  $n$ -го ступеня:

$$P_n(x_i) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n, \quad (1)$$

що задовольняє умовам інтерполяції:

$$P_n(x_i) = f(x_i), i = 0, 1, 2, \dots, n.$$

Алгебраїчний багаточлен, що задовольняє цим умовам, називається *інтерполяційним багаточленом*. Геометричний зміст інтерполяції полягає в тому, що графіки функції  $y = f(x)$  і інтерполяційного багаточлена  $y = P_n(x)$  повинні проходити через всі табличні крапки  $(x_i, y_i)$ ,  $i = 0, 1, 2, \dots, n$ . На рисунку 1(а) ці крапки виділені.

Саме ця умова повинне забезпечити близькість графіків цих функцій на розглянутому відрізку, щоб можна було використовувати інтерполяційний багаточлен  $P_n(x)$  як наближення для функції  $f(x)$ . Існують різні форми запису інтерполяційного багаточлена: традиційна форма (1), багаточлен Лагранжа й інтерполяційна формула Ньютона.

Крім побудови інтерполяційних залежностей, можна використати більше загальний варіант наближення функції - побудову апроксимуючих залежностей на основі різних функціональних взаємозв'язків між двома розглянутими величинами.

Наближена функціональна залежність, отримана на підставі експериментальних даних, називається *апроксимуючою функцією або емпіричною формулою*.

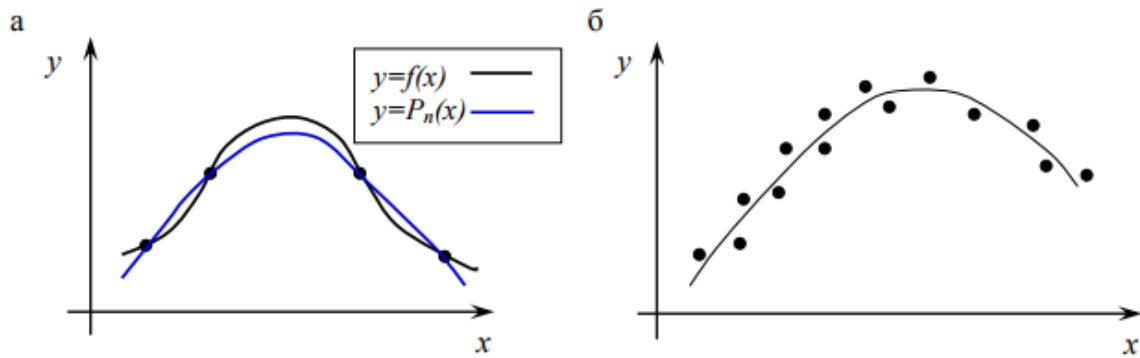


Рис.1. Графічна інтерпретація принципу побудови інтерполяційного полінома (а) й апроксимуючої лінії для функції, що задана таблично.

Побудова емпіричної формули складається з 2 етапів:

I. Підбір загального виду формули. Іноді він відомий з фізичних міркувань.

Якщо характер залежності невідомий, то початково його вибирають геометрично: експериментальні крапки наносять на графік і приблизно вгадується загальний вид залежності шляхом порівняння отриманої кривої із графіками відомих функцій (багаточлена, логарифмічних, показової функцій і т.п.). Вибір виду емпіричної залежності - найбільш складна частина рішення задачі, тому що клас відомих аналітичних залежностей неосяжний. Практика, однак, показує, що при виборі аналітичної залежності досить обмежитися досить вузьким колом функцій: лінійною, степеневою показниковою.

II. Визначення значень параметрів апроксимуючої функції.

## 2 Метод найменших квадратів

Будемо вважати, що вид апроксимуючої функції або емпіричної формули обраний і представлений у вигляді:

$$y = \varphi(x, a_0, a_1, a_2, \dots, a_m),$$

де  $\varphi(x)$  – невідома функція,  $a_0, a_1, a_2, \dots, a_m$  - невідомі параметри.

Потрібно визначити такі параметри, при яких значення апроксимуючої функції приблизно збігалися зі значеннями досліджуваної функції в крапках  $x_i$ , тобто  $y_i \approx \varphi(x_i)$ . Різницю між цими значеннями (відхилення) позначимо через  $\varepsilon_i$ .

Тоді:

$$\varepsilon_i = \varphi(x, a_0, a_1, a_2, \dots, a_m) - y_i, i = 1, 2, \dots, n$$

Мірою відхилення багаточлена  $\varphi(x)$  від заданої функції  $f(x)$  на множині крапок  $(x_i, y_i)$  є величина  $S$ , рівна сумі квадратів різниці між значеннями багаточлена й функції для всіх крапок  $x_1, x_2, \dots, x_n$ :

$$S = \sum_{i=1}^n \varepsilon_i^2 = \sum_{i=1}^n [\varphi(x_i) - y_i]^2 \rightarrow \min.$$

Задача пошуку кращих значень параметрів  $a_0, a_1, a_2, \dots, a_m$  зводиться до деякої мінімізації відхилень  $\varepsilon_i$ . Існує декілька способів рішення цієї задачі. Розглянемо найбільш відомий – метод найменших квадратів. Параметри  $a_0, a_1, a_2, \dots, a_m$  емпіричної формули знаходяться із умови мінімуму функції

$S = S(a_0, a_1, a_2, \dots, a_m)$ . Так як параметри виступають у ролі незалежних змінних функції  $S$ , то її мінімум знайдемо дорівнюючи до нуля приватні похідні по цим змінним.

$$\frac{\partial S}{\partial a_0} = 0; \quad \frac{\partial S}{\partial a_1} = 0; \quad \frac{\partial S}{\partial a_2} = 0; \quad \dots; \quad \frac{\partial S}{\partial a_m} = 0.$$

Отримані співвідношення - система рівнянь для визначення  $a_0, a_1, a_2, \dots, a_m$

## 2.1 Лінійна апроксимація

Розглянемо як емпірична формула лінійну функцію:

$$\varphi(x, a, b) = ax + b.$$

Сума квадратів відхилень запишеться в такий спосіб:

$$S = S(a, b) = \sum_{i=1}^n \varepsilon_i^2 = \sum_{i=1}^n [\varphi(x_i) - y_i]^2 = \sum_{i=1}^n (ax_i + b - y_i)^2 \rightarrow \min.$$

Для знаходження  $a$  й  $b$  необхідно знайти мінімум функції  $S(a, b)$ . Необхідна умова існування мінімуму для функції  $S$ :

$$\begin{cases} \frac{\partial S}{\partial a} = 0 \\ \frac{\partial S}{\partial b} = 0 \end{cases} \quad \text{или} \quad \begin{cases} 2 \sum_{i=1}^n (ax_i + b - y_i)x_i = 0 \\ 2 \sum_{i=1}^n (ax_i + b - y_i) = 0. \end{cases}$$

Спростимо отриману систему:

$$\begin{cases} a \sum_{i=1}^n x_i^2 + b \sum_{i=1}^n x_i = \sum_{i=1}^n x_i y_i \\ a \sum_{i=1}^n x_i + bn = \sum_{i=1}^n y_i. \end{cases}$$

Введемо позначення:

$$SX = \sum_{i=1}^n x_i, \quad SXX = \sum_{i=1}^n x_i^2, \quad SY = \sum_{i=1}^n y_i, \quad SXY = \sum_{i=1}^n x_i y_i.$$

Одержимо систему рівнянь для знаходження параметрів  $a$  й  $b$ :

$$\begin{cases} aSXX + bSX = SXY \\ aSX + bn = SY, \end{cases} \quad (2)$$

з якої знаходимо:

$$a = \frac{SXY \cdot n - SX \cdot SY}{SXX \cdot n - SX \cdot SX}, \quad b = \frac{SXX \cdot SY - SX \cdot SXY}{SXX \cdot n - SX \cdot SX}.$$

*Приклад 1.* Нехай, вивчаючи невідому функціональну залежність між  $x$  й  $y$ , у результаті серії експериментів, була отримана таблиця значень (табл. 1). Необхідно знайти наближену функціональну залежність і визначити значення параметрів апроксимуючої функції.

Таблиця 1. Дані експерименту

|     |     |     |      |      |      |      |      |      |
|-----|-----|-----|------|------|------|------|------|------|
| $X$ | 1,2 | 2,9 | 4,1  | 5,5  | 6,7  | 7,8  | 9,2  | 10,3 |
| $Y$ | 7,4 | 9,5 | 11,1 | 12,9 | 14,6 | 17,3 | 18,2 | 20,7 |

Для визначення виду залежності нанесемо експериментальні крапки на графік (мал. 2).

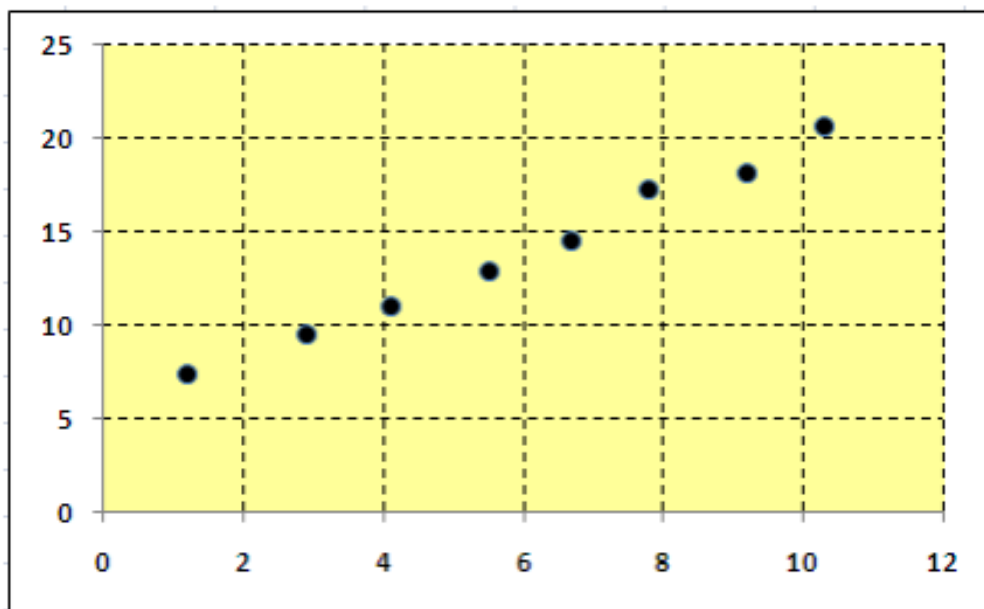


Рис.2. Експериментальні крапки

Із графіка видно, що в якості апроксимуючої функції можна вибрати багаточлен першого ступеня. Тоді необхідно побудувати лінійну модель  $f = ax + b$ , що щонайкраще буде описувати спостережувані значення. Далі, використовуючи метод найменших квадратів, знайдемо значення коефіцієнтів апроксимуючої функції:  $a$  й  $b$ . Для цього обчислимо:

$$SX = \sum_{i=1}^n x_i = 1,2 + 2,9 + 4,1 + 5,5 + 6,7 + 7,8 + 9,2 + 10,3 = 47,7;$$

$$SXX = \sum_{i=1}^n x_i^2 = 1,2^2 + 2,9^2 + 4,1^2 + 5,5^2 + 6,7^2 + 7,8^2 + 9,2^2 + 10,3^2 = 353,37;$$

$$SY = \sum_{i=1}^n y_i = 7,4 + 9,5 + 11,1 + 12,9 + 14,6 + 17,3 + 18,2 + 20,7 = 111,7;$$

$$SXY = \sum_{i=1}^n x_i y_i = 1,2 \cdot 7,4 + 2,9 \cdot 9,5 + 4,1 \cdot 11,1 + 5,5 \cdot 12,9 + 6,7 \cdot 14,6 + 7,8 \cdot 17,3 + 9,2 \cdot 18,2 + 10,3 \cdot 20,7 = 766,3.$$

Система рівнянь (2) для знаходження параметрів  $a$  й  $b$  буде мати вигляд:

$$\begin{cases} 353,37a + 47,7b = 766,3 \\ 47,7a + 8b = 111,7. \end{cases}$$

Вирішуючи систему, одержимо значення коефіцієнтів:  $a = 1,4543$  й  $b = 5,2911$ . Перевіримо правильність вибору лінійної моделі. Для цього обчислимо значення апроксимуючої функції  $f = 1,4543x + 5,2911$  і внесемо отримані значення в таблицю 2.

Таблиця 2. Результати обчислень

| № пп.        | 1       | 2      | 3       | 4       | 5       | 6       | 7       | 8       |
|--------------|---------|--------|---------|---------|---------|---------|---------|---------|
| $X$          | 1,2     | 2,9    | 4,1     | 5,5     | 6,7     | 7,8     | 9,2     | 10,3    |
| $Y$          | 7,4     | 9,5    | 11,1    | 12,9    | 14,6    | 17,3    | 18,2    | 20,7    |
| $F = ax + b$ | 7,0363  | 9,5086 | 11,2538 | 13,2899 | 15,0351 | 16,6348 | 18,6709 | 20,2707 |
| $\epsilon_i$ | -0,3637 | 0,0086 | 0,1538  | 0,3899  | 0,4351  | -0,6652 | 0,4709  | -0,4293 |

З таблиці видно, що значення апроксимуючої функції приблизно збігаються з  $Y$  для всіх крапок  $X$ . Отже, справедливий висновок: досліджувана функціональна залежність може бути приблизно описана лінійною моделлю

$$f = 1,4543x + 5,2911.$$

Визначимо міру відхилення  $S$ :

$$S = \sum_{i=1}^n \epsilon_i^2 = 1,3459.$$

Обчислене значення  $S$  (невелике  $\rightarrow \min$ ), що ще раз підтверджує правильність вибору моделі.

**Завдання:** написати й налагодити програмний код, що реалізує метод найменших квадратів для лінійної моделі, блок-схема до якого наведена на рисунку 3.

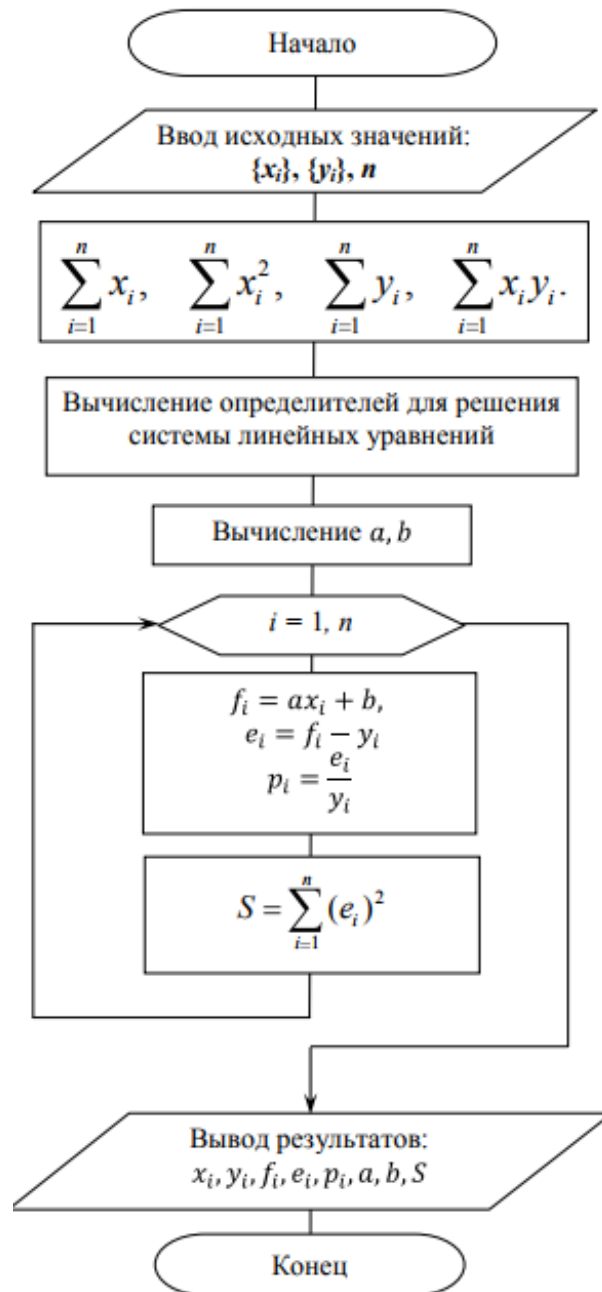


Рис.3. Укрупнена блок-схема методу найменших квадратів

## 2.1. Квадратична апроксимація

Розглянемо як емпіричну формулу квадратичну функцію:

$$\varphi(x, a_0, a_1, a_2) = a_2 x^2 + a_1 x + a_0.$$

Сума квадратів відхилень запишеться в такий спосіб:

$$S = S(a_0, a_1, a_2) = \sum_{i=1}^n (a_2 x_i^2 + a_1 x_i + a_0 - y_i)^2 \rightarrow \min.$$

Дорівнюємо до нуля приватні похідні  $S$  по невідомих параметрах:

$$\begin{cases} \frac{\partial S}{\partial a_0} = 2 \sum_{i=1}^n a_2 x_i^2 + a_1 x_i + a_0 - y_i = 0 \\ \frac{\partial S}{\partial a_1} = 2 \sum_{i=1}^n (a_2 x_i^2 + a_1 x_i + a_0 - y_i) x_i = 0 \\ \frac{\partial S}{\partial a_2} = 2 \sum_{i=1}^n (a_2 x_i^2 + a_1 x_i + a_0 - y_i) x_i^2 = 0. \end{cases}$$

Введемо позначення:

$$X_1 = \sum_{i=1}^n x_i, \quad X_2 = \sum_{i=1}^n x_i^2, \quad X_3 = \sum_{i=1}^n x_i^3, \quad X_4 = \sum_{i=1}^n x_i^4, \\ Z_1 = \sum_{i=1}^n y_i, \quad Z_2 = \sum_{i=1}^n x_i y_i, \quad Z_3 = \sum_{i=1}^n x_i^2 y_i.$$

Одержимо систему рівнянь для знаходження параметрів  $a_0, a_1, a_2$ :

$$\begin{cases} a_0 n + a_1 X_1 + a_2 X_2 = Z_1 \\ a_0 X_1 + a_1 X_2 + a_2 X_3 = Z_2 \\ a_0 X_2 + a_1 X_3 + a_2 X_4 = Z_3. \end{cases} \quad (3)$$

Використовуючи правило Крамера, знаходимо:

$$a_0 = \frac{\Delta_0}{\Delta}, \quad a_1 = \frac{\Delta_1}{\Delta}, \quad a_2 = \frac{\Delta_2}{\Delta},$$

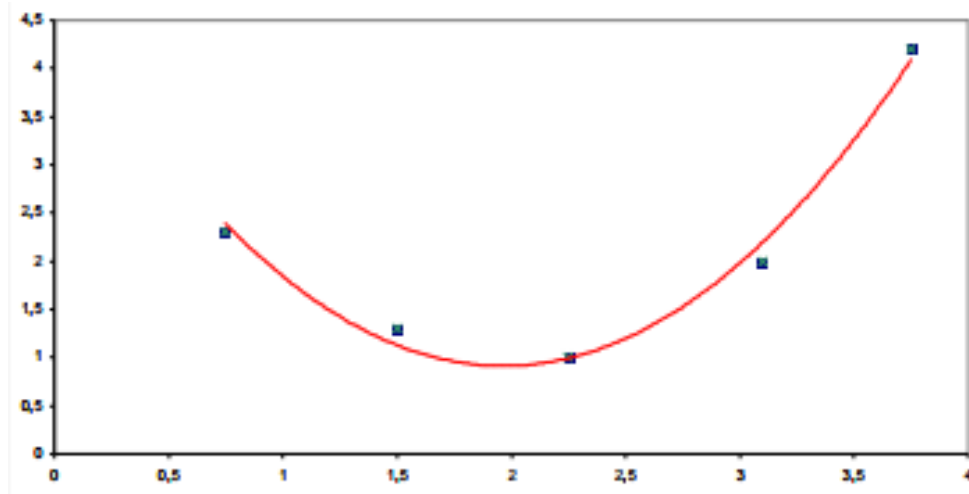
де

$$\Delta_0 = \det \begin{pmatrix} Z_1 & X_1 & X_2 \\ Z_2 & X_2 & X_3 \\ Z_3 & X_3 & X_4 \end{pmatrix} \quad \Delta_1 = \det \begin{pmatrix} n & Z_1 & X_2 \\ X_1 & Z_2 & X_3 \\ X_2 & Z_3 & X_4 \end{pmatrix} \\ \Delta_2 = \det \begin{pmatrix} n & X_1 & Z_1 \\ X_1 & X_2 & Z_2 \\ X_2 & X_3 & Z_3 \end{pmatrix} \quad \Delta = \det \begin{pmatrix} n & X_1 & X_2 \\ X_1 & X_2 & X_3 \\ X_2 & X_3 & X_4 \end{pmatrix}.$$

*Приклад.* Одержати емпіричну формулу для функції  $f(x)$ , заданою таблицею, використовуючи метод найменших квадратів.

|        |      |     |      |     |      |
|--------|------|-----|------|-----|------|
| $x$    | 0,75 | 1,5 | 2,25 | 3   | 3,75 |
| $f(x)$ | 2,3  | 1,3 | 1,0  | 2,2 | 4,2  |

Табличні дані зобразимо на графіку, з якого видно, що в якості емпіричної функції можна взяти параболу.



Знайдемо коефіцієнти системи рівнянь із таблиці.

| $n$            | $x$   | $y$ | $x^2$   | $x^3$    | $x^4$    | $xy$   | $x^2y$   |
|----------------|-------|-----|---------|----------|----------|--------|----------|
| 0              | 0,75  | 2,3 | 0,5625  | 0,421875 | 0,316406 | 1,725  | 1,29375  |
| 1              | 1,5   | 1,3 | 2,25    | 3,375    | 5,0625   | 1,95   | 2,925    |
| 2              | 2,25  | 1   | 5,0625  | 11,39063 | 25,62891 | 2,25   | 5,0625   |
| 3              | 3     | 2,2 | 9       | 27       | 81       | 6,6    | 19,8     |
| 4              | 3,75  | 4,2 | 14,0625 | 52,73438 | 197,7539 | 15,75  | 59,0625  |
| $\sum_{i=0}^n$ | 11,25 | 11  | 30,9375 | 94,92188 | 309,7617 | 28,275 | 88,14375 |

$$\begin{cases} 5a_0 + 11,25a_1 + 30,94a_2 = 11 \\ 11,25a_0 + 30,94a_1 + 94,92a_2 = 28,27 \\ 30,94a_0 + 94,92a_1 + 309,76a_2 = 88,14 \end{cases}$$

Рішення цієї системи  $a_0 = 4,54$ ;  $a_1 = -3,66$ ;  $a_2 = 0,95$ , і залежність описується формулою  $y = 4,54 - 3,66x + 0,95x^2$

## Питання та завдання для самоконтролю знань студентів

1. Визначити координати чотирьох крапок з випадковим відхиленням від експонентної залежності в інтервалі  $X=0,1\dots4$ . Провести інтерполяцію поліномом для чотирьох варіантів з різної внесеної для експоненти погрішністю ( 1%, 2%, 5%, 10% ).
2. Виконати апроксимацію для 40 крапок, отриманих аналогічно п.5.1 для варіантів з тією же погрішністю. Використати як апроксимуюча залежність лінійну комбінацію  $X_2$  й  $X_4$ .
3. Скласти процедуру, що проводить порівняльну апроксимацію МНК й інтерполяцію кубічним поліномом. Провести розрахунки за умовами п.5.1.
4. Скласти процедуру, що реалізує лінійний варіант МНК, при якому коефіцієнти апроксимуючої залежності  $P(x)=a+bx$  визначаються без рішення системи рівнянь як

$$b = \frac{\sum_{i=1}^N (x_i - x_s)(F_i - F_s)}{\sum_{i=1}^N (x_i - x_s)^2}, \quad a = F_s - bx_s, \quad x_s = \sum_{i=1}^N x_i / N, \quad F_s = \sum_{i=1}^N F_i / N.$$

5. Провести лінійну апроксимацію для чотирьох варіантів випадкового відхилення крапок від горизонтальної прямої. Варіювати кількість крапок (  $N = 10, 20, 50, 100$  ) при відхиленні не більше 10%.

# План самостійного вивчення теми № 7

з навчальної дисципліни Алгоритми та структури даних

**Тема** Алгоритми обробки символьної інформації

**Мета** Ознайомитись з основними методами обробки символьної інформації.

*знати:*

- подання символьної інформації;
- функції та методи обробки символьної інформації в класах `char`, `char[]` та `String`.

*вміти:*

- застосовувати функції та методи обробки символьної інформації в класах `char`, `char[]` та `String` при реалізації програм роботи з текстами.

Час – 4 години

## Навчальні питання:

- 1 Подання символьної інформації
- 2 Клас `char`
- 3 Клас `char[]` - масив символів
- 4 Клас `String`
  - 4.1 Конструктори класу `string`. Операції над рядками. Стрічкові константи
  - 4.2 Незмінний клас `string`. Статичні та динамічні властивості й методи класу `String`

## Методичні рекомендації

Самостійно вивчаючи запропоновану тему, студент повинен засвоїти такі основні положення:

*По-перше* Символьна інформація в алгоритмах і програмах описується даними двох типів: символьним і літерним. Вони відрізняються один від одного тим, що значенням символьної змінної є один символ, а літерною — рядок символів.

*По-друге* В `C#` є символьний клас `Char`, заснований на класі `System.Char` і що використовує двобайтне кодування `Unicode`

подання символів. Для цього типу в мові визначені символні константи - символні літерали.

*По-третьє*

У мові C# визначений клас Char[], і його можна використати для подання рядків постійної довжини. Більше того, оскільки масиви в C# динамічні, то розширюється клас завдань, у яких можна використати масиви символів для подання рядків. Так що є сенс розібратися, наскільки добре C# підтримує роботу з таким поданням рядків.

*По-четверте*

Основним типом при роботі з рядками є тип string, що задає рядка змінної довжини. Клас String у мові C# ставиться до посилальних типів. Над рядками - об'єктами цього класу - визначений широкий набір операцій, що відповідає сучасному поданню про те, як повинен бути влаштований стрічковий тип.

## Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротиєєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп .ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

# Навчальні матеріали до самостійного вивчення теми

## 1 Подання символної інформації

Символьна інформація — це інформація, що відображається за допомогою символів (букв, цифр, знаків операцій і ін.).

IBM-сумісні комп'ютери обробляють 256 різних символів, кожен з яких кодується одним байтом. Відповідність символів і байтів задається таблицею кодування, в якому для кожного символу вказується відповідний байт.

Символи з кодами від 0 до 127 побудовані за стандартом ASCII (American Standard Code for Information Interchange — Американський стандартний код обміну інформацією, читається "аски"). Друга половина таблиці (коди 128 ... 255) в нашій країні містить російські букви (кирилицю) і символи псевдографіки.

Коди 0...127

(кодування

ASCII)

|    | 000 | 016 | 032 | 048 | 064 | 080 | 096 | 112 |    |
|----|-----|-----|-----|-----|-----|-----|-----|-----|----|
| 00 |     | ◆   |     | 0   | @   | P   | `   | p   | 00 |
| 01 |     | ◆   | !   | 1   | A   | Q   | a   | q   | 01 |
| 02 |     | ↕   | "   | 2   | B   | R   | b   | r   | 02 |
| 03 | ♥   |     | #   | 3   | C   | S   | c   | s   | 03 |
| 04 | ◆   | ¶   | \$  | 4   | D   | T   | d   | t   | 04 |
| 05 | ♣   | §   | %   | 5   | E   | U   | e   | u   | 05 |
| 06 | ♠   |     | &   | 6   | F   | V   | f   | v   | 06 |
| 07 | ·   |     | '   | 7   | G   | W   | g   | w   | 07 |
| 08 |     | ↑   | (   | 8   | H   | X   | h   | x   | 08 |
| 09 | °   | ↓   | )   | 9   | I   | Y   | i   | y   | 09 |
| 10 |     | →   | *   | :   | J   | Z   | j   | z   | 10 |
| 11 |     | ←   | +   | ;   | K   | [   | k   | {   | 11 |
| 12 |     | └   | ,   | <   | L   | \   | l   |     | 12 |
| 13 |     | •   | -   | =   | M   | ]   | m   | }   | 13 |
| 14 |     | ·   | .   | >   | N   | ^   | n   | ~   | 14 |
| 15 | Ц   | ◆   | /   | ?   | О   | _   | о   | \$  | 15 |

Коди 128...255

(модифікований альтернатив-  
ний варіант)

|    | 128 | 144 | 160 | 176 | 192 | 208 | 224 | 240 |    |
|----|-----|-----|-----|-----|-----|-----|-----|-----|----|
| 00 | А   | Р   | а   | ■   | └   | ┘   | р   | ■   | 00 |
| 01 | Б   | С   | б   | ■   | └   | ┘   | с   | ┘   | 01 |
| 02 | В   | Т   | в   | ■   | └   | ┘   | т   | ┘   | 02 |
| 03 | Г   | У   | г   |     | └   | ┘   | у   | ┘   | 03 |
| 04 | Д   | Ф   | д   | └   | -   | ┘   | ф   | ┘   | 04 |
| 05 | Е   | Х   | е   | └   | +   | ┘   | х   | ┘   | 05 |
| 06 | Ж   | Ц   | ж   | └   | ┘   | ┘   | ц   | ÷   | 06 |
| 07 | З   | Ч   | з   | └   | ┘   | ┘   | ч   | ≈   | 07 |
| 08 | И   | Ш   | и   | └   | ┘   | ┘   | ш   | °   | 08 |
| 09 | Й   | Щ   | й   | └   | ┘   | ┘   | щ   | ·   | 09 |
| 10 | К   | Ъ   | к   |     | └   | ┘   | ъ   | ·   | 10 |
| 11 | Л   | Ы   | л   | └   | ┘   | ■   | ы   | √   | 11 |
| 12 | М   | Ь   | м   | └   | ┘   | ■   | ь   | ²   | 12 |
| 13 | Н   | Э   | н   | └   | ┘   | ■   | э   | z   | 13 |
| 14 | О   | Ю   | о   | └   | ┘   | ■   | ю   | ■   | 14 |
| 15 | П   | Я   | п   | └   | ┘   | ■   | я   |     | 15 |

Для того, щоб визначити по цих таблицях код того,або іншого символу, потрібно скласти номер рядка з номером стовпця, в яких він розташований. Так, код цифри 5 рівний  $05+048 = 053$ .

Символьна інформація в алгоритмах і програмах описується даними двох типів: символьним і літерним. Вони відрізняються один від одного тим, що значенням символьної змінної є один символ, а літерною — рядок символів.

Для даних символьного і літерного типів застосовані операції зчеплення (з'єднання, конкатенації) і порівняння (<, >, <=, >=, <>) порівнювати можна рядки різної довжини. Порівняння здійснюється зліва направо відповідно до ASCII-кодів відповідних символів. Так, рядок "стіл" менше рядка "стілець", рядок "teacher" більше рядка "people", а рядок "пар" менше рядка "парад".

По зрозумілих причинах у перших мовах програмування строковому типу приділялося набагато менше уваги, ніж арифметичному типу, або масивам. Тому в різних мовах рядка представлені по-різному й стандарт на строковий тип склався відносно недавно. Коли говорять про строковий тип, то звичайно розрізняють тип, що представляє:

- окремі символи, найчастіше, його називають типом `char`;
- рядка постійної довжини, часто вони представляються масивом символів;
- рядка змінної довжини - це, як правило, тип `string`, що відповідає сучасному поданню про строковий тип.

Символьний тип `char`, що представляє окремий випадок рядків - довжиною 1, корисний у багатьох завданнях. Основні операції над рядками - це розбір і збірка. При їхньому виконанні доводиться, найчастіше, доходити до кожного символу рядка. Ефективно реалізуються звичайні операції над рядками - визначення входження одного рядка в іншу, виділення підрядка, заміна символів рядка. Однак помітьте, подання рядка масивом символів добре тільки для рядків постійної довжини. Масив не пристосований до зміни його розмірів, вставки або видалення символів (підрядків).

Найбільше часто використовуваним стрічковим типом є тип, звичайно називаний `string`, що задає рядок змінної довжини. Над цим типом допускаються операції пошуку входження одного рядка в інший, операції вставки, заміни й видалення підрядків.

## 2 Клас `char`

В C# є символьний клас `Char`, заснований на класі `System.Char` і що використовує двобайтне кодування `Unicode` подання символів. Для цього типу в мові визначені символьні константи - символьні літерали. Константу можна задавати:

- символом, укладеним в одинарні лапки;
- `escape`-послідовністю, що задає код символу;
- `Unicode`-послідовністю, що задає `Unicode`-код символу.

От кілька прикладів оголошення символьних змінних і роботи з ними:

```
public void TestChar()
{
    char ch1='A', ch2 ='\x5A', ch3='\u0058';
    char ch = new Char();
    int code; string s;
    ch = ch1;
    //перетворення символьного типу в тип int
    code = ch; ch1=(char) (code +1);
    //перетворення символьного типу в рядок
    //s = ch;
    s = ch1.ToString()+ch2.ToString()+ch3.ToString();
    Console.WriteLine("s= {0}, ch= {1}, code = {2}",
        s, ch, code);
} //TestChar
```

Три символьні змінні ініціалізовані константи, значення яких задані трьома різними способами. Змінна `ch` оголошується в об'єктному стилі, використовуючи `new` і виклик конструктора класу. Тип `char`, як і всі типи `C#`, є класом. Цей клас успадковує властивості й методи класу `Object` і має велику кількість власних методів.

Чи існують перетворення між класом `char` й іншими класами? Явні або неявні перетворення між класами `char` й `string` відсутні, але, завдяки методу `ToString`, змінні типу `char` стандартним чином перетворюються в тип `string`. Існують неявні перетворення типу `char` у цілочисельні типи, починаючи з типу `ushort`. Зворотні перетворення цілочисельних типів у тип `char` також існують, але вони вже явні.

В результаті роботи процедури `TestChar` рядок `s`, отримана злиттям трьох символів, перетворених у рядки, має значення `BZX`, змінна `ch` дорівнює `A`, а її код - змінна `code` - `65`.

Не раз відзначалося, що семантика присвоювання справедлива при виклику методів і заміні формальних аргументів на фактичні. Приведемо дві процедури, що виконують взаємо-зворотні операції - одержання за кодом символу й одержання символу за його кодом:

```
public int SayCode(char sym)
{
    return (sym);
} //SayCode

public char SaySym(object code)
{
    return ((char)((int)code));
} // SaySym
```

Як бачите, у першій процедурі перетворення до цілого типу виконується неявно. У другий - перетворення явне. Заради універсальності вона злегка ускладнена. Формальний параметр має тип Object, що дозволяє передавати їй як аргумент код, заданий будь-яким цілочисельним типом. Платою за це є необхідність виконувати два явних перетворення.

Таблиця 1. Статичні методи й властивості класу Char

| Метод              | Опис                                                                                                                                                                |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| GetNumericValue    | Повертає чисельне значення символу, якщо він є цифрою, і (-1) у протилежному випадку                                                                                |
| GetUnicodeCategory | Всі символи розділені на категорії. Метод повертає Unicode категорію символу. Нижче наведений приклад                                                               |
| IsControl          | Повертає true, якщо символ є керуючим                                                                                                                               |
| IsDigit            | Повертає true, якщо символ є десятковою цифрою                                                                                                                      |
| IsLetter           | Повертає true, якщо символ є буквою                                                                                                                                 |
| IsLetterOrDigit    | Повертає true, якщо символ є буквою або цифрою                                                                                                                      |
| IsLower            | Повертає true, якщо символ заданий у нижньому регістрі                                                                                                              |
| IsNumber           | Повертає true, якщо символ є числом (десятьовою або шістнадцятьовою цифрою)                                                                                         |
| IsPunctuation      | Повертає true, якщо символ є розділовим знаком                                                                                                                      |
| IsSeparator        | Повертає true, якщо символ є роздільником                                                                                                                           |
| IsSurrogate        | Деякі символи Unicode з кодом в інтервалі [0x1000, 0x10FFF] представляються двома 16-бітними "сурогатними" символами. Метод повертає true, якщо символ є сурогатним |
| IsUpper            | Повертає true, якщо символ заданий у верхньому регістрі                                                                                                             |
| IsWhiteSpace       | Повертає true, якщо символ є "білим пробілом". До білих пробілів, крім пробілу, ставляться й інші символи, наприклад, символ кінця рядка й символ перекладу каретки |
| Parse              | Перетворить рядок у символ. Природно, рядок повинен складатися з одного символу, інакше виникне помилка                                                             |
| ToLower            | Приводить символ до нижнього регістра                                                                                                                               |
| ToUpper            | Приводить символ до верхнього регістра                                                                                                                              |
| MaxValue, MinValue | Властивості, що повертають символи з максимальним і мінімальним кодом. Символи, що повертають, не мають видимого образу                                             |

Клас Char, як і всі класи в C#, успадковує властивості й методи батьківського класу Object. Але в нього є й власні методи й властивості, і їх чимало. Зведення цих методів наведені в таблиці 1.

Більшість статичних методів перевантажені. Вони можуть застосовуватися як до окремого символу, так і до рядка, для якого вказується номер символу для застосування методу. Основну групу становлять методи Is, у край корисні при розборі рядка. Приведемо приклади, у яких використовуються багато хто з перерахованих методів:

```
public void TestCharMethods()
```

```

{
    Console.WriteLine("Статичні методи класу char:");
    char ch='a', ch1='1', lim=';', chc='\x';
    double d1, d2;
    d1=char.GetNumericValue(ch); d2=char.GetNumericValue(ch1);
    Console.WriteLine("Метод GetNumericValue:");
    Console.WriteLine("sym 'a' - value {0}", d1);
    Console.WriteLine("sym '1' - value {0}", d2);
    System.Globalization.UnicodeCategory cat1, cat2;
    cat1 =char.GetUnicodeCategory(ch1);
    cat2 =char.GetUnicodeCategory(lim);
    Console.WriteLine("Метод GetUnicodeCategory:");
    Console.WriteLine("sym '1' - category {0}", cat1);
    Console.WriteLine("sym ';' - category {0}", cat2);
    Console.WriteLine("Метод IsControl:");
    Console.WriteLine("sym '\x' - IsControl - {0}",
        char.IsControl(chc));
    Console.WriteLine("sym ';' - IsControl - {0}",
        char.IsControl(lim));
    Console.WriteLine("Метод IsSeparator:");
    Console.WriteLine("sym ' ' - IsSeparator - {0}",
        char.IsSeparator(' '));
    Console.WriteLine("sym ';' - IsSeparator - {0}",
        char.IsSeparator(lim));
    Console.WriteLine("Метод IsSurrogate:");
    Console.WriteLine("sym '\u10FF' - IsSurrogate - {0}",
        char.IsSurrogate('\u10FF'));
    Console.WriteLine("sym '\\\'' - IsSurrogate - {0}",
        char.IsSurrogate('\''));
    string str = "\U00010F00";
    //Символи Unicode в інтервалі [0x10000,0x10FFF]
    //представляються двома 16-бітними сурогатними символами
    Console.WriteLine("str = {0}, str[0] = {1}", str, str[0]);
    Console.WriteLine("str[0] IsSurrogate - {0}",
        char.IsSurrogate(str, 0));
    Console.WriteLine("Метод IsWhiteSpace:");
    str="пробіли, пробіли!" + "\x" + "\x" + "Усюди пробіли!";
    Console.WriteLine("sym '\x ' - IsWhiteSpace - {0}",
        char.IsWhiteSpace('\x'));
    Console.WriteLine("str: {0}", str);
    Console.WriteLine("й іі пробіли - символ 8 {0}, символ 17 {1}",
        char.IsWhiteSpace(str,8), char.IsWhiteSpace(str,17));
    Console.WriteLine("Метод Parse:");
    str="A";
    ch = char.Parse(str);
    Console.WriteLine("str:{0} char: {1}",str, ch);
    Console.WriteLine("Мінімальне й максимальне значення:{0}, {1}",
        char.MinValue.ToString(), char.MaxValue.ToString());
}

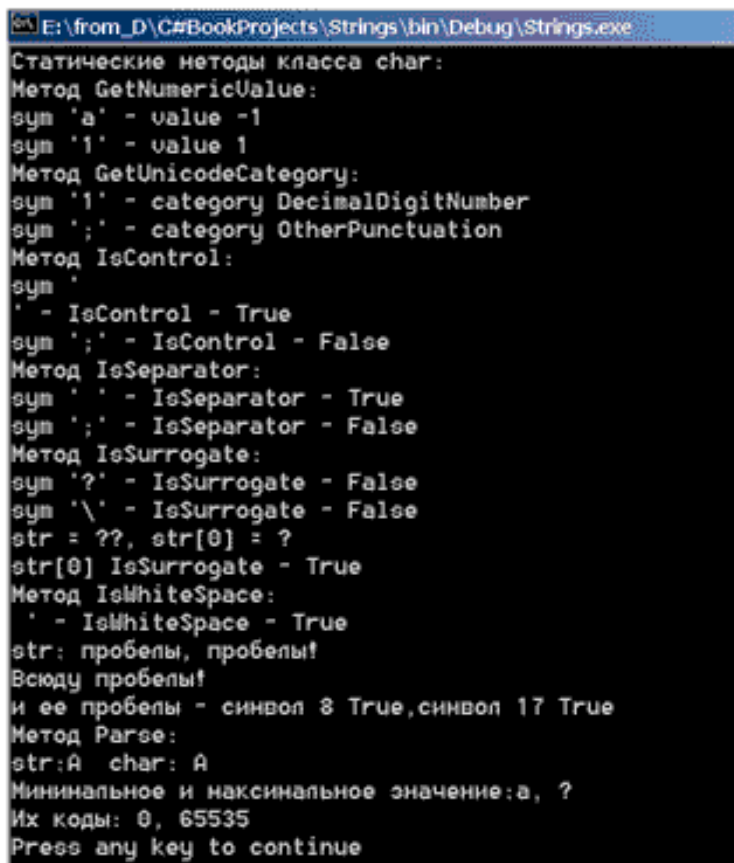
```

```

        Console.WriteLine("Їхні коди: {0}, {1}",
            SayCode(char.MinValue), SayCode(char.MaxValue));
    } //TestCharMethods

```

Результати консольного виводу, породженого виконанням методу, зображені на рисунку 1.



```

E:\from_D\C#BookProjects\Strings\bin\Debug\Strings.exe
Статические методы класса char:
Метод GetNumericValue:
enum 'a' - value -1
enum '1' - value 1
Метод GetUnicodeCategory:
enum '1' - category DecimalDigitNumber
enum ';' - category OtherPunctuation
Метод IsControl:
enum ' ' - IsControl - True
enum ';' - IsControl - False
Метод IsSeparator:
enum ' ' - IsSeparator - True
enum ';' - IsSeparator - False
Метод IsSurrogate:
enum '?' - IsSurrogate - False
enum '\' - IsSurrogate - False
str = ??, str[0] = ?
str[0] IsSurrogate - True
Метод IsWhiteSpace:
enum ' ' - IsWhiteSpace - True
str: пробелы, пробелы!
Всюду пробелы!
и ее пробелы - символ 8 True, символ 17 True
Метод Parse:
str:A char: A
Минимальное и максимальное значение:a, ?
Их коды: 0, 65535
Press any key to continue

```

Рис. 1 Виклики статичних методів класу char

Крім статичних методів, у класу Char є й динамічні. Більшість із них - це методи батьківського класу Object, успадковані й перепевні в класі Char. Із власних динамічних методів варто відзначити метод CompareTo, що дозволяє проводити порівняння символів. Він відрізняється від методу Equal тим, що для незбіжних символів видає "відстань" між символами відповідно до їх упорядкованості в кодуванні Unicode. Приведу приклад:

```

public void testCompareChars()
{
    char ch1, ch2;
    int dif;
    Console.WriteLine("Метод CompareTo");
    ch1='A'; ch2= 'Z';
    dif = ch1.CompareTo(ch2);
    Console.WriteLine("Відстань між символами {0},
        {1} = {2}", ch1, ch2, dif);
    ch1='a'; ch2= 'A';
    dif = ch1.CompareTo(ch2);
}

```

```

Console.WriteLine("Відстань між символами {0},
    {1} = {2}", ch1, ch2, dif);
ch1='Я'; ch2= 'А';
dif = ch1.CompareTo(ch2);
Console.WriteLine("Відстань між символами {0},
    {1} = {2}", ch1, ch2, dif);
ch1='А'; ch2= 'А';
dif = ch1.CompareTo(ch2);
Console.WriteLine("Відстань між символами {0},
    {1} = {2}", ch1, ch2, dif);
ch1='А'; ch2= 'А';
dif = ch1.CompareTo(ch2);
Console.WriteLine("Відстань між символами {0},
    {1} = {2}", ch1, ch2, dif);
ch1='Е'; ch2= 'А';
dif = ch1.CompareTo(ch2);
Console.WriteLine("Відстань між символами {0},
    {1} = {2}", ch1, ch2, dif);
} //TestCompareChars

```

Результати порівняння зображені на рисунку 2.

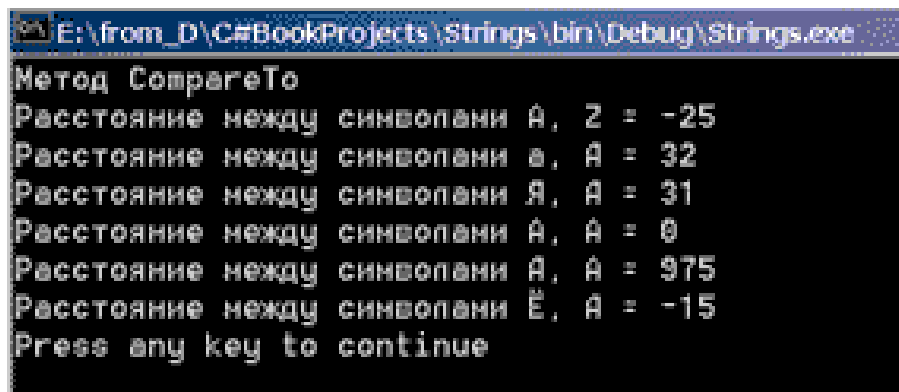


Рис. 2. Порівняння символів

### 3 Клас char[] - масив символів

У мові C# визначений клас Char[], і його можна використати для подання рядків постійної довжини. Більше того, оскільки масиви в C# динамічні, то розширюється клас завдань, у яких можна використати масиви символів для подання рядків. Так що є сенс розібратися, наскільки добре C# підтримує роботу з таким поданням рядків.

Насамперед, відповімо на запитання, чи задає масив символів C# рядок, що закінчується нулем? Відповідь: ні, не задає. Масив char[] - це звичайний масив. Константа, що задає рядок символів, належить класу String, а в C# не визначені взаємні перетворення між класами String й Char[], навіть явні. У класу String є, щоправда, динамічний метод ToCharArray, що задає подібне перетворення. Можливо також посимвольно передати вміст змінної string у масив символів. Приведемо приклад:

```

public void TestCharArAndString()
{
    //масиви символів
    //char[] str1 = "Hello, World!";
    //помилка: немає перетворення класу string у клас char[]
    string hello = "Здрастуй, Мир!";
    char[] str1 = hello.ToCharArray();
    PrintCharAr("str1",str1);
    //копіювання підстроки
    char[] World = new char[3];
    Array.Copy(str1,12,World,0,3);
    PrintCharAr("World",World);
    Console.WriteLine(CharArrayToString(World));
} //TestCharArAndString

```

Закоментовані оператори на початку цієї процедури показують, що пряме при-  
своювання рядка масиву символів неприпустимо. Однак метод ToCharArray, яким во-  
лодіють рядки, дозволяє легко перебороти ці труднощі. Ще одну можливість перет-  
ворення рядка в масив символів надає статичний метод Copy класу Array.

У нашому прикладі частина рядка str1 копіюється в масив World. По ходу  
справи в методі викликається процедура PrintCharAr класу Testing, що друкує масив  
символів як рядок. Ось її текст:

```

void PrintCharAr(string name,char[] ar)
{
    Console.WriteLine(name);
    for(int i=0; i < ar.Length; i++)
        Console.Write(ar[i]);
    Console.WriteLine();
} //PrintCharAr

```

Метод ToCharArray дозволяє перетворити рядок у масив символів. На жаль,  
зворотна операція не визначена, оскільки метод ToString, яким, звичайно ж, володі-  
ють всі об'єкти класу Char[], друкує інформацію про клас, а не вміст масиву. Ситуацію  
легко виправити, написавши підходящу процедуру. Ось текст цієї процедури  
CharArrayToString, викликаній в нашому тестованому прикладі:

```

string CharArrayToString(char[] ar)
{
    string result="";
    for(int i = 0; i< ar.Length; i++) result += ar[i];
    return(result);
} //CharArrayToString

```

Клас Char[], як і всякий масив<sup>^</sup>-масив-клас-масив в С#, є спадкоємцем не тільки  
класу Object, але й класу Array, і, отже, має всі методи батьківських класів. А є чи в

нього специфічні методи, які дозволяють виконувати операції над рядками, представленими масивами символів? Таких спеціальних операцій немає. Але деякі перевантажені методи класу `Array` можна розглядати як операції над рядками. Наприклад, метод `Copy` дає можливість виділяти й замінювати підстрічку в тілі рядка. Методи `IndexOf`, `LastIndexOf` дозволяють визначити індекси першого й останнього входження в рядок деякого символу. На жаль, їх не можна використати для більш цікавої операції - знаходження індексу входження підрядка в рядок. При необхідності таку процедуру можна написати самому. Ось як вона виглядає:

```
int IndexOfStr( char[] s1, char[] s2)
{
    //повертає індекс першого входження підстрічки s2 в
    //рядок s1
    int i =0, j=0, n=s1.Length-s2.Length; bool found = false;
    while( (i<=n) && !found)
    {
        j = Array.IndexOf(s1,s2[0],i);
        if (j <= n)
        {
            found=true; int k = 0;
            while ((k < s2.Length)&& found)
            {
                found =char.Equals(s1[k+j],s2[k]); k++;
            }
        }
        i=j+1;
    }
    if(found) return(j); else return(-1);
} //IndexOfStr
```

У реалізації використовується метод `IndexOf` класу `Array`, що дозволяє знайти початок збігу рядків, після чого перевіряється збіг інших символів. Реалізований тут алгоритм є найочевиднішим, але далеко не найефективнішим.

А тепер розглянемо процедуру, у якій визначаються індекси входження символів і підстрічок у рядок:

```
public void TestIndexSym()
{
    char[] str1, str2;
    str1 = "пококо".ToCharArray();
    //визначення входження символу
    int find, lind;
    find= Array.IndexOf(str1,'o');
    lind = Array.LastIndexOf(str1,'o');
    Console.WriteLine("Індекси входження про у пококо:{0},{1}");
```

```

        ", find, lind);
//визначення входження підстрічки
str2 = "доля".ToCharArray();
find = IndexOfStr(str1, str2);
Console.WriteLine("Індекс першого входження доля в
        рококо:{0}", find);
str2 = "око".ToCharArray();
find = IndexOfStr(str1, str2);
Console.WriteLine("Індекс першого входження око в
        рококо:{0}", find);
} //TestIndexSym

```

У цій процедурі спочатку використовуються стандартні методи класу Array для визначення індексів входження символу в рядок, а потім створений метод IndexOfStr для визначення індексу першого входження підстрічки. Коректність роботи методу перевіряється на різних рядках. Ось результати її роботи.

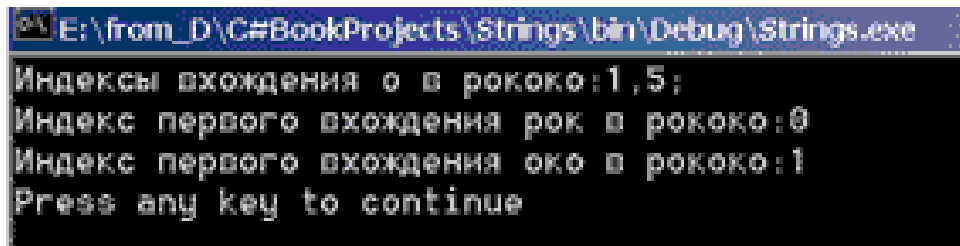


Рис. 3. Індекси входження підрядка в рядок

## 4 Клас String

### 4.1 Конструктори класу string. Операції над рядками. Стрічкові константи

Основним типом при роботі з рядками є тип string, що задає рядка змінної довжини. Клас String у мові C# ставиться до посилальних типів. Над рядками - об'єктами цього класу - визначений широкий набір операцій, що відповідає сучасному поданню про те, як повинен бути влаштований стрічковий тип.

#### *Оголошення рядків. Конструктори класу string*

Об'єкти класу String оголошуються як всі інші об'єкти простих типів - з явною або відкладеною ініціалізацією, з явним або неявним викликом конструктора класу. Найчастіше, при оголошенні стрічкової змінної конструктор явно не викликається, а ініціалізація задається стрірковою константою. Але в класу String досить багато конструкторів. Вони дозволяють сконструювати рядок з:

- символу, повтореного задане число раз;
- масиву символів char[];
- частини масиву символів.

Деяким конструкторам як параметр ініціалізації можна передати рядок, заданий типом `char*`. Але все це небезпечно, і подібні приклади приводитися й обговорюватися не будуть. Приведу приклади оголошення рядків з викликом різних конструкторів:

```
public void TestDeclStrings()
{
    //конструктори
    string world = "Мир";
    //string s1 = new string("s1");
    //string s2 = new string();
    string sssss = new string('s',5);
    char[] yes = "Yes".ToCharArray();
    string stryes = new string(yes);
    string strye = new string(yes,0,2);
    Console.WriteLine("world = {0}; sssss={1}; stryes={2}; "+
        " strye= {3}", world, sssss, stryes, strye);
}
```

Об'єкт `world` створений без явного виклику конструктора, а об'єкти `sssss`, `stryes`, `strye` створені різними конструкторами класу `String`.

Помітьте, не допускається явний виклик конструктора за замовчуванням - конструктора без параметрів. Немає також конструктора, якому як аргумент можна передати звичайну стрічкову константу. Відповідні оператори в тексті закоментовані.

#### *Операції над рядками*

Над рядками визначені наступні операції:

- присвоювання (`=`);
- дві операції перевірки еквівалентності (`==`) і (`!=`);
- конкатенація або зчеплення рядків (`+`);
- узяття індексу (`[]`).

Почну із присвоювання, що має важливу особливість. Оскільки `string` - це посилальний тип, то в результаті присвоювання створюється посилання на константний рядок, збережений в "купі". З однієї й тією ж стрірковою константою в "купі" може бути зв'язане небагато змінних стрічкового типу. Але ці змінні не є псевдонімами - різними іменами того самого об'єкта. Справа в тому, що стрічкові константи в "купі" не змінюються (про незмінюваність строкового типу будемо далі говорити докладно), тому коли одна зі змінних одержує нове значення, вона зв'язується з новим константним об'єктом в "купі". Інші змінні зберігають свої зв'язки. Для програміста це означає, що семантика присвоювання рядків аналогічна семантиці значимого присвоювання.

На відміну від інших посилальних типів операції, що перевіряють еквівалентність, порівнюють значення рядків, а не посилання. Ці операції виконуються як над значимими типами.

Бінарна операція "+" скріплює два рядки, приписуючи другий рядок до хвоста першої.

Можливість узяття індексу при роботі з рядками відбиває той приємний факт, що рядок можна розглядати як масив й одержувати без праці кожен її символ. Кожен символ рядка має тип `char`, доступний тільки для читання, але не для запису.

От приклад, у якому над рядками виконуються дані операції:

```
public void TestOps()
{
    //операції над рядками
    string s1 ="ABC", s2 ="CDE";
    string s3 = s1+s2;
    bool b1 = (s1==s2);
    char ch1 = s1[0], ch2=s2[0];
    Console.WriteLine("s1={0}, s2={1}, b1={2}, " +
        "ch1={3}, ch2={4}", s1,s2,b1,ch1,ch2);
    s2 = s1;
    b1 = (s1!=s2);
    ch2 = s2[0];
    Console.WriteLine("s1={0}, s2={1}, b1={2}, " +
        "ch1={3}, ch2={4}", s1,s2,b1,ch1,ch2);
    //Незмінні значення
    s1= "Zenon";
    //s1[0]='L';
}
```

### *Стрічкові константи*

Без констант не обійтися. В C# існують два види стрічкових констант:

- звичайні константи, які представляють рядок символів, укладені в лапки;
- @-константи, задані звичайною константою з попереднім знаком @.

У звичайних константах деякі символи інтерпретуються особливим чином. Зв'язано це насамперед з тим, що необхідно вміти задавати в рядку не друкують символи, що, такі, як, наприклад, символ табуляції. Виникає необхідність задавати символи їхнім кодом - у вигляді escape-послідовностей. Для всіх цих цілей використається комбінація символів, що починається символом "\" - зворотна коса риса. Так, пари символів: "\n", "\t", "\", "\"" задають відповідно символ переходу на новий рядок, символ табуляції, сам символ зворотної косої риски, символ лапок, що вставляє в рядок, але не сигналізує про її закінчення. Комбінація "\xNNNN" задає символ, обумовлений шістнадцятиричним кодом NNNN. Хоча таке рішення виникаючих проблем зовсім природно, іноді виникають незручності: наприклад, при завданні констант, що

визначають шлях до файлу, доводиться щораз подвоювати символ зворотної косої риски. Це одна із причин, по якій з'явилися @-константи.

В @-константах всі символи трактуються в повній відповідності з їхнім зображенням. Тому шлях до файлу краще задавати @-константою. Єдина проблема в таких випадках: як задати символ лапок, щоб він не сприймався як кінець самої константи. Рішенням є подвоєння символу. От відповідні приклади:

Два види констант

```
s1= "\x50";
s2=@"\x50""";
b1= (s1==s2);
Console.WriteLine("s1={0}, s2={1}, b1={2}", s1,s2,b1);
s1 = "c:\\c#book\\ch5\\chapter5.doc";
s2 = @"c:\c#book\ch5\chapter5.doc";
b1= (s1==s2);
Console.WriteLine("s1={0}, s2={1}, b1={2}", s1,s2,b1);
s1= "\"A\"";
s2=@"\"A\"";
b1= (s1==s2);
Console.WriteLine("s1={0}, s2={1}, b1={2}", s1,s2,b1);
```

Гляньте на результати роботи наведених фрагментів коду, отримані при виклику процедур TestDeclStrings й TestOperators.

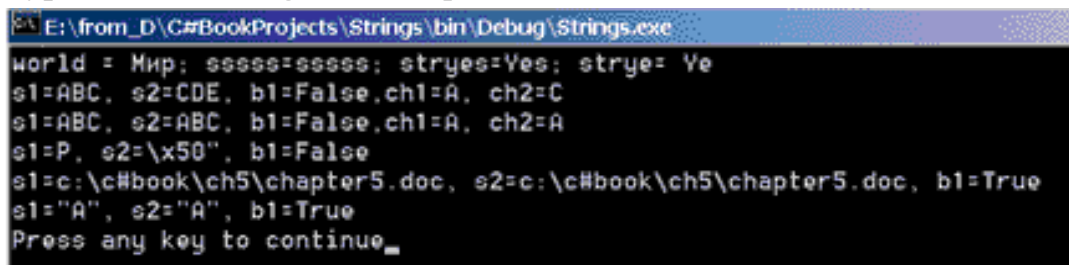


Рис. 4. Оголошення, константи й операції над об'єктами string

## 4.2 Незмінний клас string. Статичні та динамічні властивості й методи класу String

У мові C# існує поняття *незмінний (immutable) клас*. Для такого класу неможливо змінити значення об'єкта при виклику його методів. Динамічні методи можуть створювати новий об'єкт, але не можуть змінити значення існуючого об'єкта.

До таких незмінних класів ставиться й клас String. Жоден з методів цього класу не міняє значення існуючих об'єктів. Звичайно, деякі з методів створюють нові значення й повертають як результат нові рядки. Неможливість змінювати значення рядків стосується не тільки методів. Аналогічно, при роботі з рядком як з масивом дозволене тільки читання окремих символів, але не їхня заміна. Оператор присвоювання з нашого останнього приклада, у якому робиться спроба змінити перший символ рядка, неприпустимий, а тому закоментований.

```
//Незмінні значення
```

```
s1= "Zenon"; ch1 = s1[0];  
//s1[0]='L';
```

### Статичні властивості й методи класу String

Таблиця 2. Статичні методи й властивості класу String

| Метод            | Опис                                                                                                                                                                                                                                                                                                                   |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Empty            | Повертається порожній рядок. Властивість зі статусом read only                                                                                                                                                                                                                                                         |
| Compare          | Порівняння двох рядків. Метод перевантажений. Реалізації методу дозволяють порівнювати як рядки, так і підрядки. При цьому можна враховувати або не враховувати регістр, особливості національного форматування дат, чисел і т.д.                                                                                      |
| CompareOrdinal   | Порівняння двох рядків. Метод перевантажений. Реалізації методу дозволяють порівнювати як рядки, так і підрядки. Рівняються коди символів                                                                                                                                                                              |
| Concat           | Конкатенація рядків. Метод перевантажений, допускає зчеплення довільного числа рядків                                                                                                                                                                                                                                  |
| Copy             | Створюється копія рядка                                                                                                                                                                                                                                                                                                |
| Format           | Виконує форматування відповідно до заданих специфікацій формату. Нижче наведене більше повний опис методу                                                                                                                                                                                                              |
| Intern, IsIntern | Відшукується й повертається посилання на рядок, якщо така вже зберігається у внутрішньому полі даних. Якщо ж рядка немає, то перший з методів додає рядок у внутрішнє поле, другий - повертає null. Методи застосовуються звичайно тоді, коли рядок створюється з використанням будівника рядків - класу StringBuilder |
| Join             | Конкатенація масиву рядків у єдиний рядок. При конкатенації між елементами масиву уставляються роздільники. Операція, задана методом Join, є зворотною до операції, заданої методом Split. Останній є динамічним методом й, використовуючи роздільники, здійснює поділ рядка на елементи                               |

#### Метод Format

Метод Format у наших прикладах зустрічався багаторазово. Щораз, коли виконувався вивід результатів на консоль, неявно викликався й метод Format. Розглянемо оператор печатки:

```
Console.WriteLine("s1={0}, s2={1}", s1,s2);
```

Тут рядок, що задає перший аргумент методу, крім звичайних символів, містить формати, укладені у фігурні дужки. У даному прикладі використовується найпрості-

ший вид формату - він визначає об'єкт, що повинен бути підставлений у ділянку рядка, зайнята даним форматом. Крім неявних викликів, нерідко виникає необхідність явного форматування рядка.

Давайте розглянемо загальний синтаксис як самого методу `Format`, так і використовуваних у ньому форматів. Метод `Format`, як і більшість методів, є перевантаженим і може викликатися з різним числом параметрів. Перший необов'язковий параметр методу задає провайдера, що визначає національні особливості, які використовуються в процесі форматування. Як такий параметр повинен бути заданий об'єкт, що реалізує інтерфейс `System.IFormatProvider`. Якщо цей параметр не заданий, то використовується культура, задана за замовчуванням. От приклади двох реалізацій цього методу:

```
public static string Format(string, object);  
public static string Format(IFormatProvider, string, params  
object[]);
```

Параметр типу `string` задає форматизацію рядка. Заданий рядок містить один або кілька форматів, вони розпізнаються за рахунок навколишніх формат фігурних дужок. Число форматів, вставлених у рядок, визначає й число об'єктів, переданих при виклику методу `Format`. Кожен формат визначає форматування об'єкта, на який він посилається і який, після перетворення його в рядок, буде підставлений у результуючий рядок замість формату. Метод `Format` як результат повертає переданий йому рядок, де всі специфікації формату замінені рядками, отриманими в результаті форматування об'єктів.

Загальний синтаксис, специфікований формат, такий:

```
{N [,M [:<коди_форматування>]] }
```

Обов'язковий параметр `N` задає індекс об'єкта, що заміняє формат. Можна вважати, що методу завжди передається масив об'єктів, навіть якщо фактично переданий один об'єкт. Індксація об'єктів починається з нуля, як це прийнято в масивах. Другий параметр `M`, якщо він заданий, визначає мінімальну ширину поля, що приділяється рядку, що вставляє замість формату. Третій необов'язковий параметр задає коди форматування, що вказують, як варто формувати об'єкт. Наприклад, код `C` (`Currency`) говорить про те, що параметр повинен формуватися як валюта з урахуванням національних особливостей подання. Код `P` (`Percent`) задає форматування у вигляді відсотків з точністю до сотої частки.

Усе стає ясным, коли з'являються відповідні приклади. От вони:

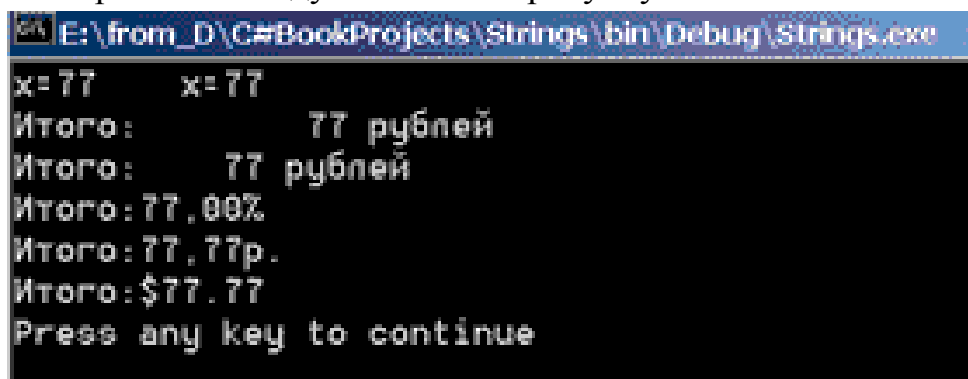
```
public void TestFormat()  
{  
    //метод Format  
    int x=77;  
    string s= string.Format("x={0}",x);  
    Console.WriteLine(s + "\tx={0}",x);  
    s= string.Format("Разом:{0,10} рублів",x);  
    Console.WriteLine(s);  
}
```

```

s= string.Format("Разом:{0,6:#####} рублів",x);
Console.WriteLine(s);
s= string.Format("Разом:{0:P} ",0.77);
Console.WriteLine(s);
s= string.Format("Разом:{0,4:C} ",77.77);
Console.WriteLine(s);
//Національні особливості
System.Globalization.CultureInfo ci =
    new System.Globalization.CultureInfo("en-US");
s= string.Format(ci,"Разом:{0,4:C} ",77.77);
Console.WriteLine(s);
} //TestFormat

```

Приведем деякі коментарі до цієї процедури. Спочатку демонструється, що і явний, і неявний виклики методу `Format` дають той самий результат. У подальших прикладах показане використання різних специфікацій формату з різним числом параметрів і різних кодів форматування. Зокрема, показаний вивід відсотків і валют. В останньому прикладі з валютами демонструється завдання провайдером національних особливостей. Із цією метою створюється об'єкт класу `CultureInfo`, ініціалізований так, щоб він задавав особливості форматування, прийняті в США. Помітьте, клас `CultureInfo` успадковує інтерфейс `IFormatProvider`. Російські національні особливості форматування встановлені за замовчуванням. При необхідності їх можна встановити в такий же спосіб, як це зроблено для США, задавши відповідно константу "ru-RU". Результати роботи методу показані на рисунку 5.



```

E:\from_D\C#BookProjects\Strings\bin\Debug\Strings.exe
x=77 x=77
Итого: 77 рублей
Итого: 77 рублей
Итого: 77.00%
Итого: 77.77p.
Итого: $77.77
Press any key to continue

```

Рис. 5. Результати роботи методу `Format`

### *Методи Join й Split*

Методи `Join` й `Split` виконують над рядком тексту взаємно зворотні перетворення. Динамічний метод `Split` дозволяє здійснити розбір тексту на елементи. Статичний метод `Join` виконує зворотну операцію, збираючи рядок з елементів.

Заданий рядком текст найчастіше являє собою сукупність структурованих елементів - абзаців, пропозицій, слів, скобкових виражень і т.д. При роботі з таким текстом необхідно розділити його на елементи, користуючись спеціальними роздільниками елементів, - це можуть бути пробіли, дужки, розділові знаки. Практично подібні

завдання виникають постійно при роботі зі структурованими текстами. Методи Split й Join полегшують рішення цих завдань.

Динамічний метод Split, як звичайно, перевантажений. Найбільше часто використовувана реалізація має наступний синтаксис:

```
public string[] Split(params char[])
```

На вхід методу Split передається один або кілька символів, інтерпритуємих як роздільники. Об'єкт string, що викликав метод, розділяється на підрядки, обмежені цими роздільниками. Із цих підрядків створюється масив, що повертає як результат методу. Інша реалізація дозволяє обмежити число елементів масиву, що повертає.

Синтаксис статичного методу Join такий:

```
public static string Join(string delimiters, string[] items )
```

Як результат метод повертає рядок, отриманий конкатенацією елементів масиву items, між якими уставляється рядок роздільників delimiters. Як правило, рядок delimiters складається з одного символу, що і розділяє в результуючому рядку елементи масиву items; але в окремих випадках обмежником може бути рядок з декількох символів.

Розглянемо приклади застосування цих методів. У першому з них рядок представляє складнопідрядну пропозицію, що розбивається на прості пропозиції. У другому пропозицію розділяється на слова. Потім виробляється зворотна зборка розібраного тексту. От код відповідної процедури:

```
public void TestSplitAndJoin()
{
    string txt = "А це пшениця, що у темному прикомірку
        зберігається," + " у будинку, що побудував Джек!";
    Console.WriteLine("txt={0}", txt);
    Console.WriteLine("Поділ тексту на прості пропозиції:");
    string[] SimpleSentences, Words;
    //розмірність масивів SimpleSentences й Words
    //установлюється автоматично відповідно до
    //розмірності масиву, що повертає методом Split
    SimpleSentences = txt.Split(',');
    for(int i=0;i< SimpleSentences.Length; i++)
        Console.WriteLine("SimpleSentences[{0}] = {1}",
            i, SimpleSentences[i]);
    string txtjoin = string.Join(",", SimpleSentences);
    Console.WriteLine("txtjoin={0}", txtjoin);
    Words = txt.Split(' ', ' ', ' ');
    for(int i=0;i< Words.Length; i++)
        Console.WriteLine("Words[{0}] = {1}", i, Words[i]);
    txtjoin = string.Join(" ", Words);
    Console.WriteLine("txtjoin={0}", txtjoin);
} //TestSplitAndJoin
```

Результати виконання цієї процедури показані на рисунку 6.

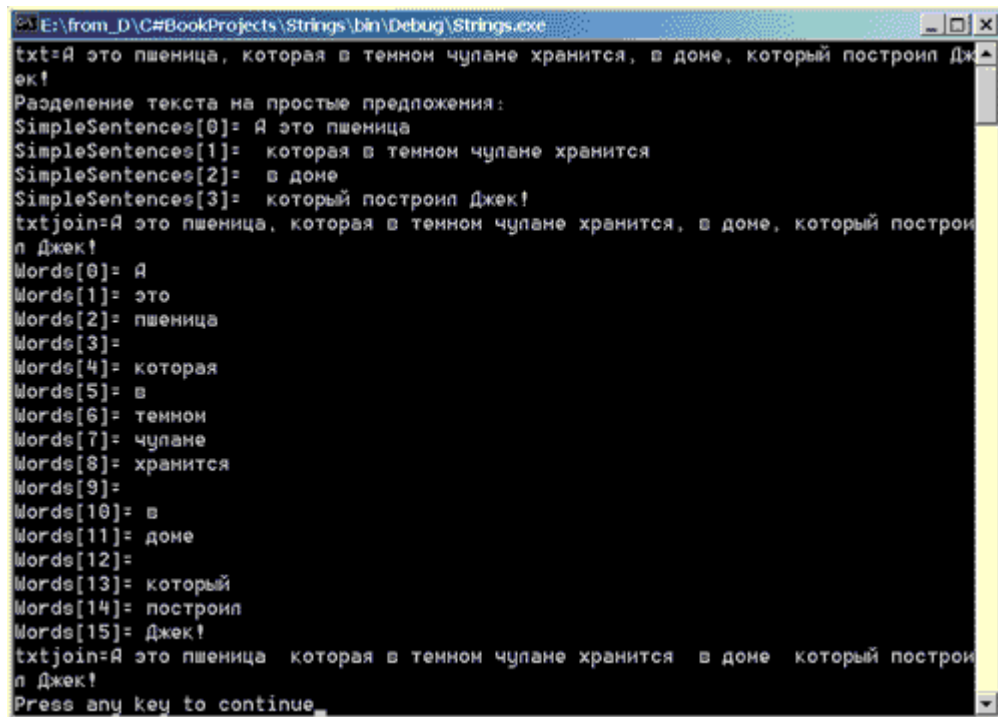


Рис. 6. Розбір і зборка рядка тексту

Зверніть увагу, що методи Split й Join добре працюють, коли при розборі використовується тільки один роздільник. У цьому випадку зборка дійсно є зворотною операцією й дозволяє відновити вихідний рядок. Якщо ж при розборі задається деякі безліч роздільників, то виникають дві проблеми:

- неможливо при зборці відновити рядок у колишньому виді, оскільки не зберігається інформація про те, який з роздільників був використаний при розборі рядка. Тому при зборці між елементами уставляється один роздільник, можливо, що складається з декількох символів;
- при розборі двох підряд, що йдуть роздільників, передбачається, що між ними перебуває порожнє слово. Зверніть увагу в тексті нашого приклада, як і покладено, після коми треба пробіл. При розборі тексту на слова як роздільники зазначені символи пробілу й коми. Із цієї причини в масиві слів, отриманому в результаті розбору, є порожні слова.

Якщо при розборі пропозиції на слова використати як роздільник тільки пробіл, то порожні слова не з'являться, але кома буде частиною деяких слів.

Як завжди, є кілька способів упоратися із проблемою. Один з них полягає в тому, щоб написати власну реалізацію цих функцій, іншої - у коректуванні отриманих результатів, третій - у використанні могутнішого апарата регулярних виражень, і про нього ми поговоримо трохи пізніше.

### *Динамічні методи класу String*

Операції, дозволені над рядками в C#, різноманітні. Методи цього класу дозволяють виконувати вставку, видалення, заміну, пошук входження підрядки в рядок.

Клас String успадковує методи класу Object, частково їх перевизначаючи. Клас String успадковує й, отже, реалізує методи чотирьох інтерфейсів: IComparable, ICloneable, IConvertible, IEnumerable. Три з них уже розглядалися при описі класів-масивів.

Розглянемо найбільш характерні методи при роботі з рядками.

Зведення методів, наведене в таблиці 3, дає досить повну картину широких можливостей, наявних при роботі з рядками в C#. Варто пам'ятати, що клас string є незмінним. Тому Replace, Insert й інші методи являють собою функції, що повертають новий рядок як результат і не змінюють рядок, що викликав метод.

Таблиця 3. Динамічні методи й властивості класу String

| Метод                                                     | Опис                                                                                                               |
|-----------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------|
| Insert                                                    | Вставляє підрядок в задану позицію                                                                                 |
| Remove                                                    | Видаляє підрядок в заданій позиції                                                                                 |
| Replace                                                   | Заміняє підрядок в заданій позиції на нову підстроку                                                               |
| Substring                                                 | Виділяє підрядок в заданій позиції                                                                                 |
| IndexOf,<br>IndexOfAny,<br>LastIndexOf,<br>LastIndexOfAny | Визначаються індекси першого й останнього входження заданого підрядка або будь-якого символу із заданого набору    |
| StartsWith,<br>EndsWith                                   | Повертається true або false, залежно від того, починається або закінчується рядок заданої підрядкової              |
| PadLeft, PadRight                                         | Виконує набивання потрібним числом пробілів на початку й наприкінці рядка                                          |
| Trim, TrimStart,<br>TrimEnd                               | Зворотні операції до методів Pad. Віддаляються пробіли на початку й наприкінці рядка, або тільки з одного її кінця |
| ToCharArray                                               | Перетворення рядка в масив символів                                                                                |

### Питання та завдання для самоконтролю знань студентів

1. Які правила запису символьних даних (констант, змінних, масивів)?
2. Як виконується опис символьних даних?
3. Які методи введення і виведення символьних даних?
4. Які методи обробки символьних даних?
5. Які операції виконуються над символьними даними?
6. Вказати функцій, що використовують для обробки символьних даних?
7. Які правила створення програм із використанням символьних даних?

8. Як здійснюється обробка рядків у мові C#?
9. Що є ознакою кінця стрічки?
10. Підрахуйте кількість ком в заданому тексті.
11. Підрахуйте, скільки разів в заданому тексті зустрічається заданий символ.
12. Замініть в заданому тексті буквосполучення "min" на "max".
13. У заданому тексті подсчитайте загальна кількість букв "x" і "y".
14. У заданому тексті всюди букву "a" замініть на букву "б", а букву "б" - на букву "а".
15. Подвійте кожну букву в заданому тексті.
16. У заданому слові кожну букву "б" замініть буквосочетанием "ку".
17. Викресліть із заданого слова всі букви "а".
18. Задану рядок А переписіть у зворотному порядку в рядок В.
19. З'ясуйте, чи є в заданому пропозиції буква "и".
20. З'ясуйте, чи вірно, що в заданому пропозиції Р є всі букви, що входять в заданий слово S.
21. Визначте кількість пропозицій в заданому тексті (пропозиція закінчується або точкою, або знаки питання та оклику).
22. У заданій послідовності слів знайдіть всі слова, що мають задане закінчення.
23. Знайдіть найдовше і найкоротше слово в заданому реченні.