Міністерство освіти і науки України
Запорізький національний університет

Н. В. Матвіїшина, Г. А. Циммерман, Г. М. Шило

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Методичні рекомендації до лабораторних занять
для здобувачів ступеня вищої освіти бакалавра
спеціальності «Комп’ютерні науки»
освітньо-професійної програми «Комп’ютерні науки»

Запоріжжя
2025

УДК 004.421:004.6 (076.5)
М338

Матвіїшина Н. В., Циммерман Г. А., Шило Г. М. Алгоритми та структури даних : методичні рекомендації до лабораторних занять для здобувачів ступеня вищої освіти бакалавра спеціальності «Комп'ютерні науки» освітньо-професійної програми «Комп'ютерні науки». Запоріжжя : Запорізький національний університет, 2025. 54 с.

Методичні рекомендації спрямовані на те, щоб допомогти студентам засвоїти фундаментальні аспекти теорії алгоритмів та набути навичок роботи з різноманітними структурами даних. Видання охоплює ключові теми дисципліни: від аналізу обчислювальної складності та методів декомпозиції задач до реалізації складних динамічних структур, таких як однозв'язні та двозв'язні списки, стеки, черги, а також різні способи подання та обробки графів.

Відповідно до силабуса дисципліни «Алгоритми та структури даних», у рекомендаціях поєднано теоретичну інформацію з прикладним інструментарієм програмування.

Для здобувачів ступеня вищої освіти бакалавра спеціальності «Комп'ютерні науки», які навчаються за освітньо-професійною програмою «Комп'ютерні науки». Водночас видання може бути корисним студентам різних спеціальностей.

Рецензент
С. І. Гоменюк, доктор технічних наук, професор, професор кафедри програмної інженерії

Відповідальний за випуск
О. С. Пшенична, кандидат педагогічних наук, доцент, в.о. завідувача кафедри комп'ютерних наук

ВСТУП

У межах навчальної дисципліни «Алгоритми та структури даних» вивчаються основні поняття та методи теорії й практики алгоритмізації, класифікація структур даних і алгоритмів їх обробки; аналіз обчислювальної складності алгоритмів. Надаються систематичні знання та формуються навички щодо реалізації структур даних (масивів, записів, списків, стеків, тощо) у сучасних програмах для ПК.

Метою вивчення навчальної дисципліни «Алгоритми та структури даних» є досягнення здобувачами освіти рівня усвідомлення та впевненого використання основних понять, правил, методів теорії й практики побудови структур даних, алгоритмів і алгоритмізації. До питань, що розглядаються у ході вивчення курсу, відносяться класифікація структур даних і відповідних алгоритмів їх обробки, аналіз обчислювальної складності алгоритмів, програмна реалізація структур даних мовами програмування.

Основними завданнями вивчення дисципліни «Алгоритми та структури даних» є:

– засвоїти фундаментальні поняття теорії алгоритмів та опанувати методи оцінки їхньої обчислювальної складності для аналізу ефективності програмних рішень;

– опанувати методику декомпозиції складних задач на простіші складові та навчитися ефективно застосовувати механізм функцій і підпрограм для створення модульного коду;

– вивчити особливості зберігання та обробки даних у масивах, а також опанувати базові алгоритми сортування та пошуку елементів;

– засвоїти принципи побудови та функціонування динамічних структур даних, зокрема навчитися працювати з однозв’язними та двозв’язними списками, стеками та чергами;

– опанувати різні способи програмного подання графів (матриці суміжності та інцидентності, списки) та навчитися здійснювати взаємоперетворення між цими формами зберігання даних;

– засвоїти та реалізувати на практиці спеціальні алгоритми обробки графів, навчившись обирати найбільш оптимальний спосіб подання даних залежно від специфіки задачі.

Згідно з вимогами освітньої-професійної програми здобувачі освіти мають досягти таких **компетентностей** та **програмних результатів навчання**:

– здатність до абстрактного мислення, аналізу та синтезу

– здатність застосовувати знання у практичних ситуаціях;

– знання та розуміння предметної області та розуміння професійної діяльності

– здатність вчитися і оволодівати сучасними знаннями;

– здатність до пошуку, оброблення та аналізу інформації з різних джерел;

– здатність до логічного мислення, побудови логічних висновків, використання формальних мов і моделей алгоритмічних обчислень, проєктування, розроблення й аналізу алгоритмів, оцінювання їх ефективності та

складності, розв’язності та нерозв’язності алгоритмічних проблем для адекватного моделювання предметних областей і створення програмних та інформаційних систем;

– здатність використовувати сучасні методи математичного моделювання об’єктів, процесів і явищ, розробляти моделі й алгоритми чисельного розв’язування задач математичного моделювання, враховувати похибки наближеного чисельного розв’язування професійних задач;

– здатність проєктувати та розробляти програмне забезпечення із застосуванням різних парадигм програмування: узагальненого, об’єктно-орієнтованого, функціонального, логічного, з відповідними моделями, методами й алгоритмами обчислень, структурами даних і механізмами управління;

– застосовувати знання основних форм і законів абстрактно-логічного мислення, основ методології наукового пізнання, форм і методів вилучення, аналізу, обробки та синтезу інформації в предметній області комп'ютерних наук;

– використовувати сучасний математичний апарат неперервного та дискретного аналізу, лінійної алгебри, аналітичної геометрії, в професійній діяльності для розв’язання задач теоретичного та прикладного характеру в процесі проєктування та реалізації об’єктів інформатизації;

– проєктувати, розробляти та аналізувати алгоритми розв’язання обчислювальних та логічних задач, оцінювати ефективність та складність алгоритмів на основі застосування формальних моделей алгоритмів та обчислюваних функцій;

– розробляти програмні моделі предметних середовищ, вибирати парадигму програмування з позицій зручності та якості застосування для реалізації методів та алгоритмів розв’язання задач в галузі комп’ютерних наук.

Методичні рекомендації складаються зі вступу, 6 лабораторних робіт, які містять теоретичні відомості, що вивчаються студентами спеціальності «Комп’ютерні науки» в рамках дисципліни «Алгоритми та структури даних», завдання та питання для самоконтролю.

Структура та зміст методичних рекомендацій відповідають силабусу дисципліни «Алгоритми та структури даних», яка вивчається студентами спеціальності «Комп’ютерні науки» освітньо-професійної програми «Комп’ютерні науки». Водночас пропоноване видання може бути корисним студентам різних спеціальностей.

Лабораторна робота 1. АЛГОРИТМИ ТА ПРОГРАМИ. ПОКАЗНИКИ ОБЧИСЛЮВАЛЬНОЇ СКЛАДНОСТІ АЛГОРИТМІВ

Мета: Засвоїти фундаментальні аспекти теорії алгоритмів і програм, опанувати практичне застосування базових алгоритмічних структур у різних середовищах розробки та мовах програмування. Визначити підходи до створення швидких алгоритмів та програм

🕮🔄 Теоретичні відомості та методичні рекомендації

Ключові поняття: 🔑🔑	• Алгоритм • Розгалуження • Цикл • Комп'ютерна програма • Реалізація алгоритму • Елементарна операція • Складність алгоритму • Час виконання алгоритму

Під поняттям **алгоритм** традиційно розуміють кінцеву послідовність точних, зрозумілих виконавцю, елементарних команд (інструкцій), яка визначає процес переведення вхідних даних в необхідний підсумковий результат. Термін алгоритм походить від слова *algorithmi* – латинської транслітерації арабського імені хорезмійського математика IX століття аль-Хорезмі (Мухаммед бен Муса аль-Маджус аль-Хорезмі, який в одній зі своїх робіт описав правила виконання дій при розв’язуванні арифметичних задач).

Алгоритм визначає певний обчислювальний процес, який спрямований на обробку певної сукупності вхідних даних для подальшого одержання визначених цими даними результатів. Якщо цей процес закінчується одержанням результатів, то кажуть, що відповідний алгоритм може бути застосований до розглянутої сукупності даних. У протилежному випадку – алгоритм не може бути застосований до цієї сукупності даних.

Лише за наявності певних ознак (властивостей) послідовність інструкцій можна вважати алгоритмом:

Таблиця 1.1 – *Властивості алгоритмів*

Властивість	Пояснення властивості
Скінченність	результат буде отримано після виконання кінцевої кількості елементарних дій
Результативність	обов’язкове одержання необхідного результату після виконання алгоритму
Визначеність (однозначність)	результати, незалежно від виконавця (користувача) алгоритму, мають бути однаковими

Властивість	Пояснення властивості
Масовість	можливість застосування алгоритму до цілого класу однотипних задач, що різняться лише значеннями вхідних даних
Зрозумілість	алгоритм має бути поданий елементарними інструкціями мови виконавця алгоритму (людини, автомата, комп'ютера)
Дискретність	можливість розбиття процесу, на окремі етапи (кроки)

Очевидно, що при невиконанні хоча б однієї властивості з наведеного списку, послідовність інструкції може не відповідати визначенню алгоритму.

Для побудови алгоритму необхідно конкретизувати (описати) його елементи:

– набір об'єктів – сукупність вхідних даних, проміжних і кінцевих результатів;
– правило початку;
– правило переробки (опрацювання) інформації;
– правило виведення результатів;
– правило закінчення.

Алгоритм завжди розрахований (орієнтований) на конкретного виконавця. У нашому випадку таким виконавцем є комп'ютер.

До основних способів опису (подання) алгоритму слід віднести: словесно-формульний, графічний, з використанням мови програмування.

Словесно-формульний спосіб – це запис по пунктах алгоритму у виді тексту з формулами.

Наприклад, необхідно визначити значення виразу Y=3*a*-(*x*+2). Словесно-формульним способом алгоритм розв'язування цієї задачі може бути записаний наступним чином:

1. Початок
2. Увести значення змінних a та x
3. Додати константу 2 до значення змінної x
4. Збільшити значення змінної a у 3 рази
5. Зменшити значення добутку 3a (отриманому командою 4) на величину x+2 (отриманою командою 3) та зробити результат значенням змінної Y
6. Вивести Y як результат обчислення всього виразу
7. Кінець

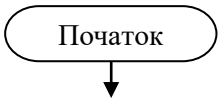
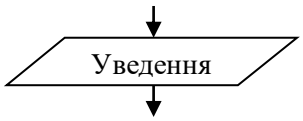
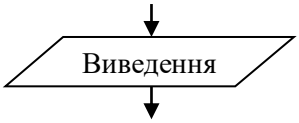
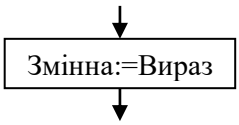
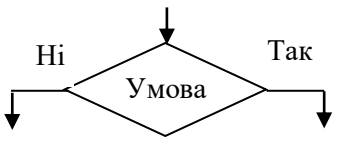
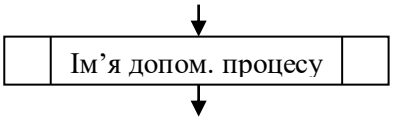
Графічний спосіб використовує послідовно з'єднані спеціальні зображення дій із заздалегідь відомим змістом. Різновидом графічного подання алгоритмів є ***блок-схема*** алгоритму (лінії зі стрільцями, якими зв'язані блоки, вказують послідовність виконання дій, обумовлених цими блоками). Головною відмінністю такого способу подання алгоритмів є наочність (що робить цей спосіб більш універсальним): кожна операція обчислювального процесу має окреме зображення; повне графічне зображення алгоритму показує розгалуження шляхів отримання розв'язку задачі в залежності від різних умов,

повторення окремих етапів обчислювального процесу, інші важливі деталі.

Існує так звана єдина система програмної документації (ЄСПД), що встановлює правила розробки, оформлення програм і супутньої документації, а правила оформлення блок-схем алгоритмів визначені ДСТ 10.002-80 ЄСПД, ДСТ 10.003-80 ЄСПД. Операції обробки даних зображуються на схемі відповідними блоками. В межах однієї схеми рекомендується зображувати блоки однакових розмірів. Усі блоки нумеруються. Лінії зі стрілками, що з'єднують блоки, визначають послідовність дій. З блоку (крім логічного) має виходити тільки одна лінія. Логічний блок має як продовження два блоки, тобто з нього виходять дві лінії. Місця на схемі, де лінії зливаються, виділяються крапкою.

Схему алгоритму варто виконати як єдине ціле, однак у разі потреби допускається обривати лінії, що з'єднують блоки. Якщо при обриві лінії продовження схеми знаходиться на тому ж аркуші, то на першому та другому кінці лінії зображується спеціальний символ – з'єднувач-коло з ідентифікатором усередині. Усередині парних кіл указується однаковий ідентифікатор. Ідентифікатором, як правило, є порядковий номер блоку, до якого спрямована сполучна лінія або велика латинська літера. Блок-схема повинна містити всі розгалуження, цикли і виклики допоміжних програм.

Таблиця 1.2 – *Вигляд типових елементів блок-схем*

Блок *Початок*	
Блок *Кінець*	
Блок *Уведення інформації*	
Блок *Виведення інформації*	
Блок *Обчислення*	
Блок *Перевірка умови* (логічний)	
Блок *Виклик допоміжного процесу*	
Лінія *з’єднання* блоків	

Можна виділити 3 найпростіші (базові) алгоритмічні структури: послідовність двох або більше операцій (конструкція послідовного виконання); вибір напрямку (умовна конструкція або конструкція розгалуження); повторення (циклічна конструкція). Будь-який обчислювальний процес може бути представлений (поданий) як комбінація цих елементарних алгоритмічних структур.

Лінійним прийнято називати обчислювальний процес, у якому операції виконуються послідовно, у порядку їхнього запису, без альтернативних гілок та повторень (рис. 1.1). Кожна операція є самостійною, незалежною від яких-небудь умов. На схемі блоки, що відображають ці операції, розташовуються в лінійній послідовності.

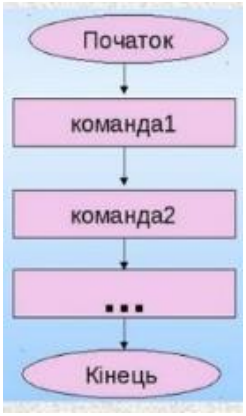


Рисунок 1.1 – Зображення лінійного алгоритму

Обчислювальний процес називається ***розгалуженим,*** якщо для його реалізації передбачено кілька напрямків (варіантів). Розгалуження в програмі – це вибір однієї з декількох послідовностей команд при виконанні програми. Вибір напрямку залежить від раніше визначеної ознаки (умови) (рис. 1.2). Розгалужені процеси, що складаються з двох гілок, називають простими, інші – складними. Складний розгалужений процес можна подати за допомогою простих розгалужених процесів. Один з напрямків розгалуження вибирається перевіркою умови, в результаті якої можливі дві відповіді: «так» – умова виконана, «ні» – умова не виконана. Варто мати на увазі, що, хоча на схемі алгоритму повинні бути показані всі можливі напрямки обчислення в залежності від виконання визначеної умови (або умов), при однократному проходженні програми процес реалізується тільки по одній гілці, а інші виключаються. Будь-яка гілка алгоритму повинна приводити до завершення обчислювального процесу.

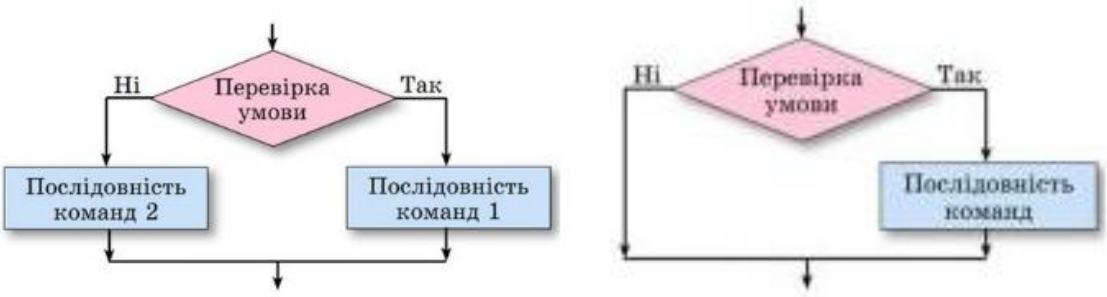


Рисунок 1.2 – Зображення повного та неповного розгалуження

Цикл – це багаторазово повторювана ділянка алгоритму або програми (рис. 1.3). В організації циклу можна виділити наступні етапи:

– підготовка (ініціалізація) циклу;
– виконання обчислень циклу (тіло циклу);
– модифікація параметрів;
– перевірка умови завершення (продовження) циклу.

Порядок виконання цих етапів може змінюватися.

У структурі циклу завжди виокремлюються дві частини: умова циклу, тіло циклу. У залежності від їх взаємного розташування розрізняють цикли з нижнім і верхнім закінченнями або, іншими словами, цикли з передумовою та післяумовою. Для циклу з нижнім закінченням тіло циклу виконується як мінімум один раз, тому що спочатку виконуються обчислення, а потім перевіряється умова виходу з циклу. У випадку з циклом з верхнім закінченням тіло циклу може не виконатися жодного разу, якщо відразу задовольняється умова виходу.

Логічний вираз, який називають умовою циклу і який має значення «істина» при будь-якому виконанні циклу також називається інваріантом циклу.

Цикл називається детермінованим, якщо число повторень його тіла є заздалегідь відомим (вже визначеним, або його можна визначити за потреби).

Цикл називається ітераційним, якщо число повторень тіла циклу заздалегідь є невідомим, залежить від значень параметрів (деяких змінних), що беруть участь в обчисленнях.

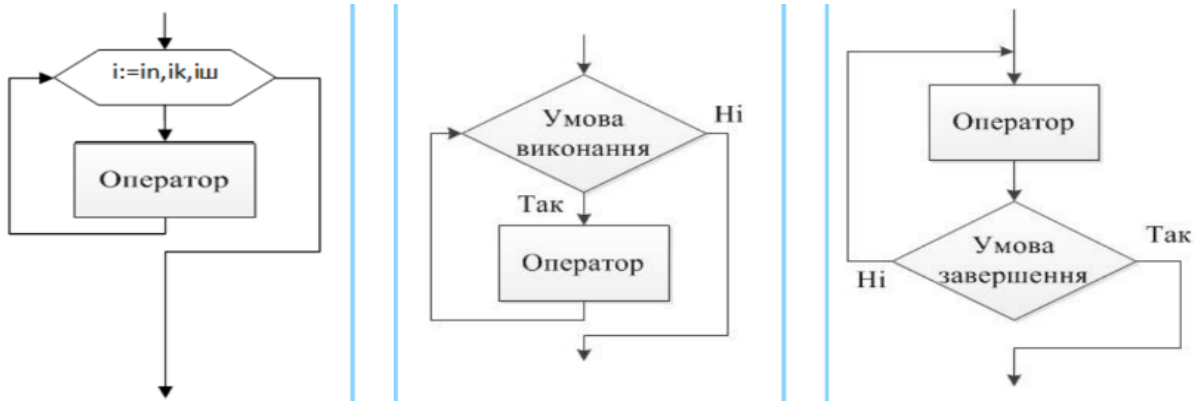


Рисунок 1.3 – Зображення циклів із параметром, із передумовою, з післяумовою

У програмах часто використовуються вкладені цикли – цикл зовнішній вміщує у собі цикл внутрішній (рис. 1.4). Допускається декілька рівнів вкладення.

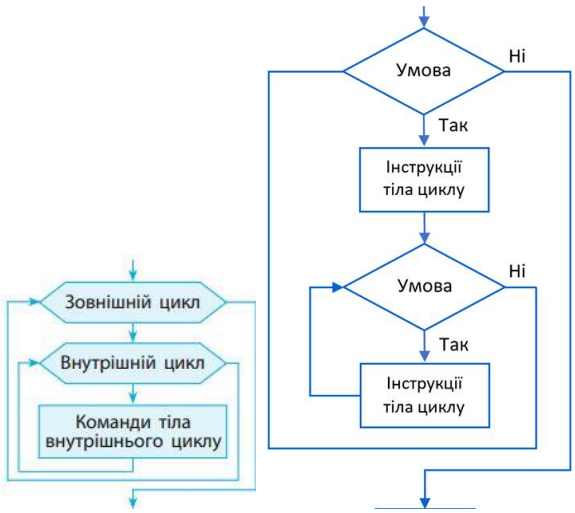


Рисунок 1.4 – Приклади зображень вкладених циклів

Приклад програмної реалізації вкладених циклів:

```
import turtle
t=turtle.Pen()
for i in range(8):
    for i in range (10):
        t.forward(15)
        t.left(45)
        t.forward(15)
        t.backward(15)
        t.right(90)
        t.forward(15)
        t.backward(15)
        t.left(45)
    t.backward(150)
    t.left(45)
```

Перед програмною реалізацією важливим питанням є також виправлення недосконалостей розроблених алгоритмів.

Має сенс виділити наступні класи недосконалості алгоритмів та програм:

Операції, що доповнюють одна одну. Найбільш очевидна недосконаліть, яка полягає в послідовному застосуванні двох доповнюючих один одного операторів до одного і того ж операнду. Компілятори з функцією оптимізації коду для більшості мов програмування видаляють такі конструкції, якщо вони зустрічаються.

Приклад:

```
F+S → T
T*T+T-T → R
```

після вдосконалення одержимо:

```
F+S → T
T*T → R
```

Неоднозначні операнди. Ім'я операнду використовується для позначення різних об'єктів в різних місцях програми кожен раз, коли попереднє входження імені вже не потрібне. Такий підхід дозволяє економити пам'ять в машинному коді, де відсутня можливість вказівки еквівалентності двох імен операндів.

Проте, неоднозначне використання імен операндів веде до погіршення сприйняття програм.

Приклад:

```
F+S → R
R*R → R
```

після вдосконалення одержимо:

```
F+S → T
T*T → R
```

Синонімічні операнди. Протилежністю неоднозначності імен операндів є використання двох різних імен для одного і того ж об'єкту. Якщо дві і більш змінних використовуються так, що їх значення завжди повинні бути однаковими, то очевидна наявність синонімічних операндів.

Приклад:

```
F+S → TI
F+S → T2
Tl * T2 → R
```

після вдосконалення одержимо

```
F+S → TI
TI*TI → R
```

Загальні підвирази. У тих випадках, коли певна комбінація членів виразу повинна використовуватися більше одного разу, їй звичайно призначають нове ім'я і надалі посилаються на нього. При відхиленні від цього правила програма буде містити загальні підвирази.

Приклад:

```
(F+S)*(F+S) → R
```

після вдосконалення одержимо

```
F+S → T
T*T → R
```

Непотрібне присвоєння. Попередній недосконалості протиставляється випадок, коли комбінації членів виразу приписується окреме ім'я, але використовується воно надалі тільки один раз. Нове ім'я не служить якій-небудь корисній меті, оскільки відповідний підвираз не входить в інші частини програми.

Приклад:

```
F+S → T
T↑2 → R
```

після вдосконалення одержимо

```
(P+Q) ↑2 → R
```

Вирази, не представлені у вигляді добутку множників. Як відомо, вираз сприймається легше, якщо воно представлене у вигляді добутку множників.

Приклад:

```
F*F+2*F*S+S*S → R
```

після вдосконалення одержимо

(F+S) ↑2 → R

Програми написані програмістами-початківцями звичайно містять досить велику кількість недосконалостей, зростання досвіду поступово веде до розробки більш досконалих алгоритмів і програм.

Залежно від того, як вдало розроблено алгоритм, він потребує певних витрат часу на отримання результату. Таким чином, має сенс познайомитися з поняттям обчислювальна складність алгоритму, як узагальненим поняттям про відносну кількість використаних для вирішення задачі операцій (елементарних дій).

Завдання до лабораторної роботи 1

1. Запропонуйте вашу версію визначення часу виконання наступних зразків фрагментів алгоритмів (програм), визначте та вкажіть умовний час виконання. Надайте власні коментарі стосовно того, що саме знаходить програма.

Зразок 1

```
1 i = 0
2 while i < n:
3     k += 1
4     i += 1
```

Зразок 2

```
1 i = 0
2 while i < n:
3    if i % 2 == 0:
4         k += 1
5    i += 1
```

Зразок 3

```
1 i = 0
2 while i < n:
3    j = n
4    while j != 0:
5         k += 1
6         j //= 3
7    i += 1
```

Результат не число секунд або хвилин, це математичний вираз, у цьому завданні він залежить від числа n.

Наприклад, є такий алгоритм:

```
i = 0
while i < n:
      i += 1
```

Ми бачимо, що у ньому використано операції додавання, порівняння та доступу (читання або запис) до комірок пам’яті, де зберігаються дані

`i = 0` у цьому рядку 2 операції, отримати доступ до комірки, та розмістити туди 0

`while i < n`: у цьому рядку, оскільки тут записано початок циклу, маємо n раз *(отримати доступ до двох комірок, прочитати їх вміст та виконати порівняння), тобто 5n

`i += 1` у цьому рядку, оскільки він належить тілу циклу маємо `n` раз *(отримати доступ до 1 комірки, прочитати її вміст, виконати додавання одиниці та розмістити результат як нове значення знов до комірки), тобто `4n`.

Таким чином, остаточно маємо `2+5n+4n=9n+2`.

Звичайно, що такі підрахунки досить умовні, але вони дозволяють побачити загальну картину залежності часу роботи алгоритму від того, наскільки економно ми його записали та скільки використали даних.

2. Складіть 3 (три) різні алгоритми та тексти програм для визначення всіх цифр цілого (4-значного) числа та перевірки факту чи є це число паліндромом. Проранжуйте алгоритми (програми) у порядку від повільного до швидкого. Поясніть отримані вами результати. Мова програмування та середовище розробки обирається самостійно.

Наприклад, обрано онлайн середовище onlinegdb.com та мову програмування c++. Тоді один із варіантів програми може виглядати так:

```
#include <iostream>
using namespace std;

int main() {
    int number;

    //std::cout<<"Hello"<< endl;
    // Введення чотиризначного числа
    cout << "Введіть чотиризначне число: ";
    cin >> number;

    // Виокремлення цифр
    int d1 = number / 1000;          // Перша цифра
    int d2 = (number / 100) % 10;    // Друга цифра
    int d3 = (number / 10) % 10;     // Третя цифра
    int d4 = number % 10;            // Четверта цифра

        // Виведення цифр
    cout <<  d1 << endl;
    cout <<  d2 << endl;
    cout <<  d3 << endl;
    cout <<  d4 << endl;

    // Перевірка на паліндром
    if (d1 == d4 && d2 == d3) {
```

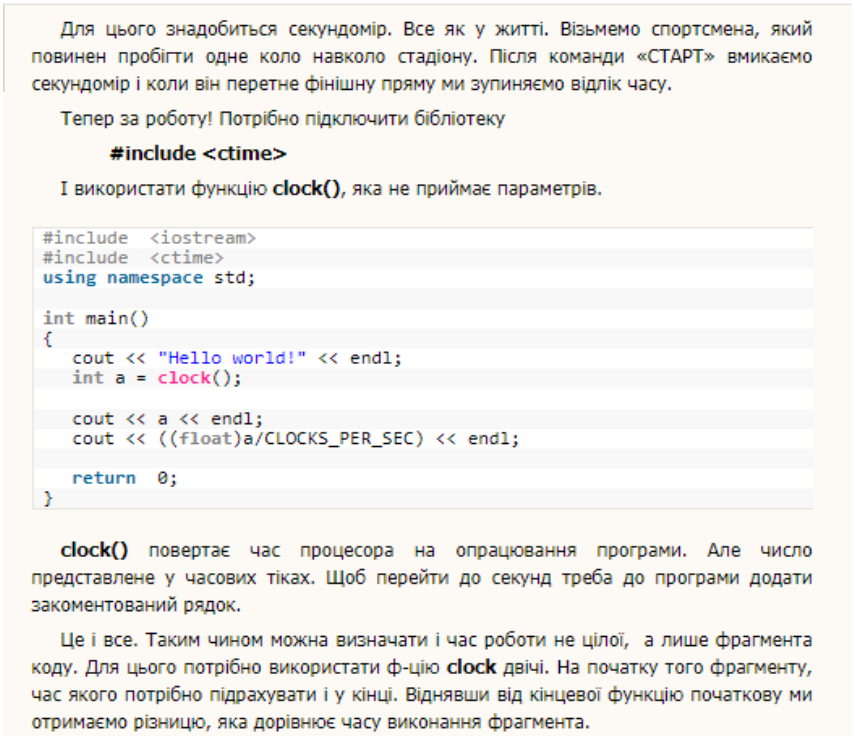
```
        cout << "Число є паліндромом." << endl;
    } else {
        cout << "Число не є паліндромом." << endl;
    }

    return 0;
}
```

Для визначення швидкості можна виокремити основу програми:

```
    cin >> number;
    int d1 = number / 1000;
    int d2 = (number / 100) % 10;
    int d3 = (number / 10) % 10;
    int d4 = number % 10;
    cout <<  d1 << endl;
    cout <<  d2 << endl;
    cout <<  d3 << endl;
    cout <<  d4 << endl;
    if (d1 == d4 && d2 == d3) {
        cout << "Число є паліндромом." << endl;
    } else {
        cout << "Число не є паліндромом." << endl;
    }
```

та застосувати прийом, що описаний раніше.

Або можна використати метод, описаний у статті за посиланням https://programming.in.ua/programming/c-plus-plus/323-time-to-program-run-c-plus-plus



Для цього знадобиться секундомір. Все як у житті. Візьмемо спортсмена, який повинен пробігти одне коло навколо стадіону. Після команди «СТАРТ» вмикаємо секундомір і коли він перетне фінішну пряму ми зупиняємо відлік часу.

Тепер за роботу! Потрібно підключити бібліотеку

#include <ctime>

І використати функцію **clock()**, яка не приймає параметрів.

```
#include  <iostream>
#include  <ctime>
using namespace std;

int main()
{
   cout << "Hello world!" << endl;
   int a = clock();

   cout << a << endl;
   cout << ((float)a/CLOCKS_PER_SEC) << endl;

   return  0;
}
```

clock() повертає час процесора на опрацювання програми. Але число представлене у часових тіках. Щоб перейти до секунд треба до програми додати закоментований рядок.

Це і все. Таким чином можна визначати і час роботи не цілої, а лише фрагмента коду. Для цього потрібно використати ф-цію **clock** двічі. На початку того фрагменту, час якого потрібно підрахувати і у кінці. Віднявши від кінцевої функцію початкову ми отримаємо різницю, яка дорівнює часу виконання фрагмента.

Рисунок 1.5 – Опис ідеї вимірювання витрат часу

Ось приклад такої програми:

```
// big array

#include  <iostream>
#include  <iomanip>
#include  <vector>
#include  <time.h>
#include  <cstdlib>
#include  <conio.h>
#include  <cstring>
#include  <cmath>
#include  <algorithm>
#include  <ctime>
using namespace std;


typedef unsigned int ui;

int main ()
{
   // system ("color B");
   const short int size = 200;
   srand(time(NULL));
   // cout << size << endl;

   int ar[size][size];

   // input
   for (int i=0; i<size; ++i)
   {
      for (int j=0; j<size; ++j)
      {
         ar[i][j] = -100 + rand() % 200;
         cout << ar[i][j] << "\t";
      }
      cout << endl;
   }

   ui a = clock();


   int Max = ar[0][0],
      Min = ar[0][0];

   // Search
   for (int i=0; i<size; ++i)
   {
      for (int j=0; j<size; ++j)
      {
         if (Min > ar[i][j])
            Min = ar[i][j];
         if (Max < ar[i][j])
            Max = ar[i][j];
      }
   }

   cout << "Max = " << Max << endl;
   cout << "Min = " << Min << endl;
   cout << "Finish!!!" << endl;
   ui b = clock();

   double t1 = ((double)b/CLOCKS_PER_SEC),
      t2 = ((double)a/CLOCKS_PER_SEC);

   cout << "Results: " << "\n" << "all program - " << t1 << endl;
   cout << "only search - " << t1 - t2 << endl;

   _getch();
   return  0;
}
```

Рисунок 1.6 – Приклад програмної реалізації зазначеної ідеї

3. Складіть 3 (три) різні алгоритми та тексти програм для визначення суми всіх натуральних чисел на проміжку від 1 до 100. Проранжуйте алгоритми (програми) у порядку від повільного до швидкого. Поясніть отримані вами результати. Мова програмування та середовище розробки обирається вами самостійно.

Можна скористатися математичною специфікою вказаної задачі, побачивши задачу про суму членів арифметичної прогресії, а можна використати звичайне повне перебирання усіх чисел зазначеного у завданні діапазону, а можна побачити можливість скоротити це перебирання. Для вимірювання витрат на роботу алгоритмів (програм) слід скористатися підходами, які наведені у попередньому завданні. Отримані результати логічно подати у вигляді таблиці.

❓❓ Питання для самоконтролю

1. Як на вашу думку, чому в сьогоденній практиці алгоритми все менше зображують графічно?
2. Як ви поясните, чому про тестувальників програм є багато інформації, а от про тестувальників алгоритмів ні?
3. Які ви знаєте базові алгоритмічні конструкції та яку з них у принципі можна виключити з цього списку?
4. Чому перевіряти виконання властивостей алгоритму гарна ідея?
5. Чому найефективніший алгоритм може програти неефективному? Наведіть приклади.
6. Що таке недосконалості алгоритму та які різновиди недосконалості алгоритмів в принципі можна ігнорувати?
7. Іноді певні числа, відіграють важливу роль у розумінні можливостей алгоритмів. Уявіть, що два різні алгоритми виконують однакове завдання. Для роботи першого алгоритму потрібно $2N+5$ кроків, а другому потрібно $(N-7)^2$ кроків. При яких значенях N ви надасте перевагу тому, чи іншому алгоритму?

Лабораторна робота 2. ПРАВИЛА ЗАСТОСУВАННЯ ДЕКОМПОЗИЦІЇ ТА ВИКОРИСТАННЯ ФУНКЦІЙ

Мета: Засвоїти принципи декомпозиції складних завдань та опанувати практичні навички розробки модульних програм із використанням допоміжних алгоритмів, підпрограм та функцій. Визначити та обґрунтувати підходи щодо доцільності та правил використання методу декомпозиції.

🕮🔄 Теоретичні відомості та методичні рекомендації

Ключові поняття: 🔑🔑	• Декомпозиція • Допоміжна програма • Особливості використання декомпозиції • Різновиди допоміжних програм (внутрішні та зовнішні, стандартні та нестандартні) • Рекурсія • Бібліотека підпрограм • Правила використання підпрограм з бібліотек

Під поняттям **декомпозиція** задачі традиційно розуміють процес розбиття складної задачі на менші, простіші й таким чином більш зрозумілі частини. Декомпозиція ґрунтується на аналітичному методі дослідження. Цей метод, властивий людському мисленню, у явній формі сформували як самостійний технічний прийом пізнання в XVII ст. представники раціоналізму. Так, наприклад, Рене Декарт пропонував "…розчленувати кожну досліджувану задачу на стільки частин, скільки необхідно для її легкого розв'язання…".

Такий підхід має ряд переваг:

– не лише спрощує вирішення задачі, а й допомагає організувати код (текст програми) у більш читабельний та підтримуваний вигляд;

– кожна така частина (блок) задачі може бути реалізована й одразу продемонстрована замовнику;

– кожна така частина несе в собі певну цінність і такі підзадачі простіше буде пріоритезувати;

– при такому підході у реалізації можуть брати участь різні фахівці;

– розбиття тексту програми на частини (модулі) спрощує пошук та усунення помилок, а також оновлення;

– при тестуванні замість того, щоб перевіряти великий за обсягом текст, можливо перевіряти маленькі, незалежні частинки та економити час та ресурси;

– із досвіду програмістів-розробників відомо, що добре структурований і поділений на частини код є чистішим та ефективнішим;

– новачки в команді розробників можуть швидше зрозуміти сутність проєкту, якщо він поділений на блоки;

– над різними сегментами проєкту можуть працювати різні команди розробників;

– легше додавати нові функції та розширювати проєкт (виконувати масштабування), коли він таким чином структурований;

– більш ефективно використовуються пам'ять та час роботи процесора завдяки чіткому розподілу завдань.

У результаті декомпозиції створюється комплекс програм, якісь з них очевидно виконують функції загального керування цим комплексом програм, а інші – є допоміжними (вузькоспеціалізованими, необхідними лише для певної маленької підзадачі).

Програма, яка використовується для вирішення підзадачі у цьому комплексі програм буде допоміжною програмою або підпрограмою (ПІДпорядкованою головній керуючій програмі).

Залежно від того, яким чином, де саме зберігається ця допоміжна програма кажуть про внутрішні або зовнішні підпрограми.

Залежно від того, хто є розробником допоміжної програми (ті ж самі розробники, що і для головної програми, або якість окремі люди або авторські колективи) допоміжні програми називають стандартними або нестандартними.

Залежно від того, як саме організовано алгоритм роботи допоміжної програми, деякі з них називають рекурсивними (від слова **рекурсія** – процес, що у своїй роботі використовує свою власну копію). Спрощено про це можна прочитати у статті за посиланням https://surl.li/doedwf.

Принципи рекурсії:

– рекурсія повинна мати базовий випадок, коли функція не викликає саму себе. Цей випадок служить базою для рекурсії і дозволяє зупинити процес виконання функції;

– рекурсія повинна також мати рекурсивний випадок, коли функція викликає саму себе з іншими аргументами. Цей випадок служить вхідними даними для рекурсії.

Прикладом рекурсії може бути процес отримання ряду чисел Фібоначчі, де кожне число є сумою двох попередніх чисел. Формула для цієї послідовності: `F(n) = F(n-1) + F(n-2)`, а два перші числа в послідовності дорівнюють 0 і 1. Рекурсивний алгоритм для обчислення n-го числа в послідовності Фібоначчі на мові програмування Python може виглядати так:

```
def fibonacci(n):
if n <= 0:
     return 0
elif n == 1:
     return 1
else:
     return fibonacci(n-1) + fibonacci(n-2)
print(fibonacci(5))
```

У цьому прикладі функція Фібоначчі приймає як параметр ціле число n, яке є n-м числом у послідовності. Функція спочатку перевіряє базові випадки, де `n = 0` або `n = 1`. Якщо n менше або дорівнює 0, вона повертає 0, а якщо n дорівнює 1, вона повертає 1. Якщо n більше 1, функція рекурсивно викликає

себе з параметрами `n-1` і `n-2`, а потім повертає суму двох результатів. Рекурсивні виклики функцій вкладені один в одного доти, доки не буде виконано умову завершення. Можна і по-іншому реалізувати рекурсію для цієї задачі.

Очевидно, що рекурсія – не завжди найкращий спосіб вирішення завдань, у деяких випадках вона знижує продуктивність програми.

Одну й ту саму задачу можна розв’язувати з використанням рекурсії та без її використання. Наприклад, обчислення факторіалу певного натурального числа.

Приклад 1 – без рекурсії

```
#include <iostream>
using namespace std;

// Функція для обчислення факторіалу
unsigned long long factorial(int n) {
    unsigned long long fact = 1;
    for (int i = 1; i <= n; ++i) {
        fact *= i;
    }
    return fact;
}

int main() {
    int number;
    cout << "Введіть натуральне число: ";
    cin >> number;
    if (number < 0) {
        cout << "Факторіал визначений тільки для невід'ємних
чисел." << endl;
    } else {
        cout << "Факторіал числа " << number << " дорівнює " <<
factorial(number) << endl;
    }
    return 0;
}
```

Приклад 2 – з рекурсією

```
#include <iostream>
using namespace std;

// Функція для обчислення факторіалу з використанням рекурсії
int factorial(int n) {
    if (n <= 1) // Базовий випадок: факторіал 0 або 1 дорівнює 1
        return 1;
    else
        return n * factorial(n - 1); // Рекурсивний виклик
}
```

```
int main() {
    int number;
  cout << "Введіть число: ";
    cin >> number;

    // Виклик функції та виведення результату
    cout << "Факторіал числа " << number << " дорівнює " <<
factorial(number) << endl;
    return 0;
    }
```

Як правило, розробники, які вирішують певну задачу протягом певного часу, створюють цілий набір допоміжних програм. Логічно для їх зберігання використати спеціальне сховище, яке отримало назву бібліотека підпрограм (або бібліотечний модуль). Часто бібліотеки для їх ефективного використання зберігаються у вже відкомпільованому вигляді. За правилами, щоб використати бібліотеку, точніше якийсь її елемент – підпрограму, функцію, необхідно підключити до основної програми спеціальною командою – директивою (для Pascal це uses, для c++ це `include`, для Python це `import`). Наприклад, `#include <iostream>` або `import tkinter`

Завдання до лабораторної роботи 2

1. Дайте визначення для поняття декомпозиція задачі. Наведіть ваші міркування стосовно доцільності та правил використання декомпозиції під час розробки програмного забезпечення.

2. На прикладі задачі про загальну суму цифр продемонструйте два способи її вирішення – без використання та з використанням декомпозиції.

Задача про загальну суму цифр. Для кожного натурального числа на проміжку від числа a до числа b (a<b) знайти суму його цифр, а у підсумку знайти загальну суму усіх цифр усіх чисел вказаного проміжку.

Спочатку виокремлюємо кожну цифру (а це число від 0 до 9), а потім додаємо її до спеціально створеної змінної Sum.

3. Наведіть власні приклади задачі та вирішення її без використання та з використанням декомпозиції. Вкажіть переваги та недоліки кожного з підходів залежно від ситуації.

4. Підготуйте 2 програми на обчислення кількості сполучень – без та з використанням допоміжних програм.

Сполучення – одне з базових понять комбінаторики. Коротку інформацію можна отримати за посиланням https://surl.li/amasmx.

5. Розробіть програму на знаходження суми, добутку або степеня

натурального числа без та з використанням рекурсії.

❓❓ Питання для самоконтролю

1. Як на вашу думку, у чому полягає суттєва причина масового використання мови Python, у контексті теми роботи?
2. Як ви поясните, чому ідея поділу задачі на частини така популярна саме у галузі програмування?
3. Які ви знаєте способи зберігання допоміжних програм, у чому полягає їх сенс?
4. Чому ідея використання допоміжних програм не завжди найефективніша з точки зору використання часу?
5. Що таке рекурсія і чому ідея використання рекурсивних функцій не є найефективнішою з точки зору витраченого часу? Наведіть приклади.
6. Опишіть скорочену технологію підготовки та використання допоміжної програми під час розв’язування задачі з програмування (узагальнено або на конкретному прикладі).
7. На власному простому прикладі порівняйте алгоритмічну складність двох рішень задачі – з використанням допоміжної програми та без неї.
8. На власному простому прикладі порівняйте алгоритмічну складність двох рішень задачі – з використанням рекурсії та без неї.

Лабораторна робота 3. МАСИВИ: ЗБЕРІГАННЯ, ОБРОБКА, СОРТУВАННЯ, ПОШУК

Мета: Засвоїти принципи організації даних у вигляді масивів та опанувати практичні методи розробки алгоритмів для їх обробки як базових структур даних. Визначити основні етапи опрацювання масивів, основні методи вирішення задач сортування та пошуку. Розглянути можливі модифікації методів сортування та порівняти їх ефективність

🕮🔄 Теоретичні відомості та методичні рекомендації

Ключові поняття: 🔑🔑

- Структура даних
- Масив
- Задача сортування
- Метод сортування
- Задача пошуку
- Метод пошуку
- Модифікація

При розв’язанні задач з обробки інформації ми вимушені думати над питаннями – які дані слід обробити та як саме зберігати ці дані. Дані можна зберігати в окремих комірках пам’яті – тоді це будуть одиночні дані. Наприклад, зріст, вага, вік, стать, прізвище (навіть, коли це дані декількох людей), а можна ті ж самі дані зберігати як групу взаємопов’язаних даних, що має відношення до конкретної людини з певним унікальним прізвищем. Тоді це будуть групові дані, або структура.

Таким чином, структура даних – спосіб організації, зберігання та керування даними в комп'ютері; вони визначають не лише те, як дані зберігаються, а й те, як до них можна ефективно отримати доступ і маніпулювати. Правильний вибір структури даних допоможе вирішити задачу обробки певної інформації більш ефективно. Структури даних важливі в розробці програмного забезпечення (ПЗ), від їх вибору залежить робота алгоритму.

У практиці програмування використовують статичні та динамічні, лінійні та нелінійні структури даних.

Найбільш простим переліком популярних структур даних можна вважати такий: масив, список, стек, черга, дерево, хеш-таблиця.

Для виконання цієї роботи розглянемо саме масиви.

Масив – фіксована структура, яка зберігає елементи одного типу в безперервних комірках пам’яті (рис. 3.1). Масиви бувають одномірними та багатовимірними (масиви масивів). Їхні розміри, як правило, фіксовані, тому у вже створений масив не завжди просто вставити новий елемент (наприклад, слід скопіювати старий масив та з нього створити новий, збільшивши розмір).

Рисунок 3.1 – Схема, що пояснює особливості одновимірного масиву

Масив вважається **одновимірним**, якщо кожен його елемент має один індекс, двовимірним (таблицею, матрицею) – якщо кожен елемент має два індекси (номер рядка, а у ньому номер стовпчика), і т.д., таким чином розуміється сутність поняття розмірність. Поняття розмір (або довжина) масиву пояснюється як кількість його елементів. Базовий індекс (або початок відліку) у залежності від обраної для реалізації мови програмування може бути різним (частіше за все 0 або 1).

При залученні масивів у процес вирішення задачі слід виділити наступні етапи роботи з масивами:

– конструкцію (або створення) нового масиву у пам’яті;

– ініціалізацію (початкове заповнення комірок пам’яті певними значеннями);

– запис значень у комірки (через уведення або присвоєння), читання значень з комірок (під час обробки або виведення);

– деструкцію (або повне його видалення) (у випадку використання динамічної пам’яті).

За досить великий період використання масивів у програмуванні існують дві класичні задачі: сортування масиву (або задача сортування), пошук у масиві (або задача пошуку). Для їх вирішення винайдено досить велика кількість ідей, які отримали назви методи сортування та методи пошуку.

Задача пошуку в масиві – це процес знаходження елементів, які відповідають заданим критеріям (наприклад, пошук конкретного значення, максимального або мінімального елемента, унікальних або таких, що повторюються) шляхом перегляду елементів масиву. Для цього використовуються різні алгоритми, такі як лінійний пошук (для несортованих масивів) та бінарний пошук (для упорядкованих масивів), а також методи фільтрації та порівняння.

Типи задач пошуку в масиві:

– знаходження елемента з певним значенням у масиві;

– знаходження найбільшого або найменшого значення серед усіх елементів масиву;

– знаходження елементів, які зустрічаються в масиві лише один раз або, навпаки, кілька разів;

– відбирання елементів, які задовольняють певній умові, і створення нового масиву з цих елементів.

Популярними (бо вони прості для розуміння та реалізації) є наступні алгоритми пошуку:

Лінійний пошук: перевірка кожного елемента масиву послідовно до знаходження потрібного значення (ідея проста, але може бути неефективною для великих масивів).

Бінарний пошук (на вже впорядкованому масиві): пошук здійснюється шляхом послідовного порівняння шуканого значення з елементом посередині масиву, зі звуженням області пошуку вдвічі на кожному кроці (це значно швидше, ніж лінійний пошук, але вимагає, щоб масив попередньо був упорядкований).

Задача сортування масиву передбачає таке переставляння його елементів, після якого буде досягнута певна умова, наприклад, значення елементів масиву будуть розміщуватися за зростанням (або спаданням).

Для загального розуміння, розглянемо два найпростіших методи сортування.

Сортування вибором максимального елемента

Нехай потрібно впорядкувати елементи масиву `X`, що зберігає 10 дійсних чисел за неспаданням:

$$X[1] \le X[2] \le \ldots \le X[10].$$

Алгоритм сортування буде наступним (рис. 3.2):

– відшукати максимальний елемент з послідовності `X[1]..X[10]`;
– максимальний елемент із цієї послідовності поміняти місцями з `X[10]`;
– відшукати максимальний елемент із послідовності `X[1]..X[9]`;
– максимальний елемент із цієї послідовності поміняти місцями з `X[9]`4
– …
– максимальний елемент із послідовності `X[1]..X[2]` поміняти місцями з `X[2]`.

Робота вказаного алгоритму проілюстрована наступним зображенням.

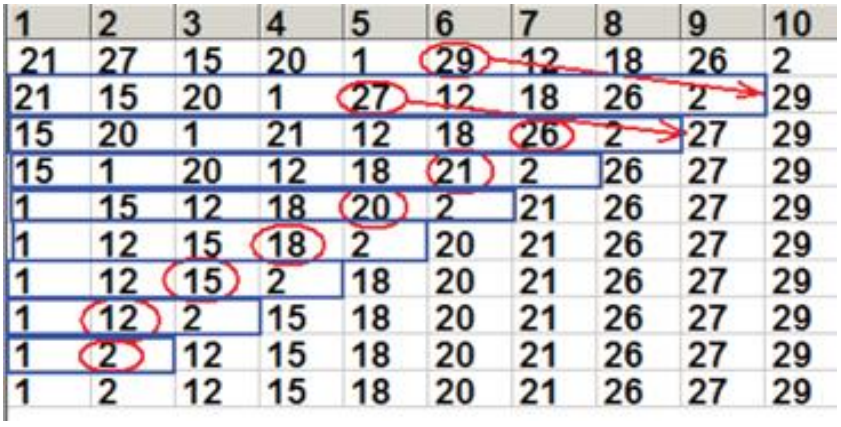


Рисунок 3.2 – Ілюстрація роботи алгоритму сортування масиву вибором

Програмний код, що реалізує описаний алгоритм:

```
   for (int K = 10; K >= 2; --K) {
            int M = 1;
            int Max = X[0];
            for (int i = 1; i < K; ++i) { // i від 1 до K-1
(оскільки індексація з 0)
                if (X[i] > Max) {
                    Max = X[i];
                    M = i;
                }
            }
            // перестановка X[K-1] і X[M]
            int C = X[M];
            X[M] = X[K-1];
            X[K-1] = C;
        }
```

Сортування обміном (метод бульбашки)

Метод бульбашки ґрунтується на порівнянні та перестановці сусідніх чисел (рис. 3.3). Алгоритм сортування цим методом буде наступним:

– послідовно порівнювати пари сусідніх елементів `X[i]` та `X[i+1]` `(i:1..N-1)`, а тоді, якщо `X[i] > X[i+1]`, то поміняти їх місцями, логічній змінній `Flag` надати значення `True`. У результаті першого перегляду послідовності на `N`-му місці буде найбільший з усіх елементів, тобто він, як бульбашка, «спливе» нагору;

– переглянути елементи від `1` до `N-2`; на `(N-1)`-му місці з'явиться найбільший серед `(N-1)` перших елементів і т. д.

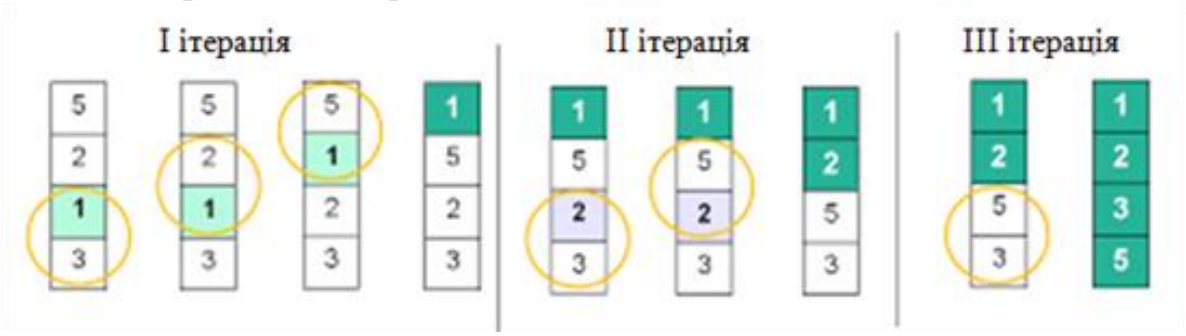


Рисунок 3.3 – Ілюстрація роботи алгоритму сортування масиву обміном

Програмний код, що реалізує описаний алгоритм:

```
do {
        Flag = false;
        for (int i = 0; i < 9; i++) { // індекси 0..8, бо
порівнюємо з i+1
            if (X[i] > X[i + 1]) {
                C = X[i];
                X[i] = X[i + 1];
                X[i + 1] = C;
```

```
                Flag = true;
            }
        }
    } while (Flag);
```

Змінна `Flag` виконує роль сигнального прапорця. Вона отримує значення `True` в тому випадку, якщо відбулась хоча б одна перестановка сусідніх елементів. Якщо значення `Flag` не змінилось, це означає, що елементи масиву вже впорядковані і подальший перегляд послідовності значень не потрібний.

На поточний момент існує великий список алгоритмів сортування – оригінальних та модифікованих, простих та складних, повільних та швидких.

Коли справа стосується вибору алгоритму сортування для конкретної ситуації, потрібно враховувати низку факторів:

1. Розмір вхідних даних: якщо їх багато, особливо якщо їх об’єм вимірюється в десятках тисяч елементів, то на допомогу приходять ефективні алгоритми, такі як швидке сортування або сортування злиттям.
2. Вимога до стійкості: деякі ситуації вимагають збереження порядку елементів з однаковими значеннями. У цьому випадку необхідно використати стійкий алгоритм, наприклад, сортування злиттям.
3. Особливості вхідних даних: різні алгоритми сортування проявляють себе по-різному залежно від характеристик вхідних даних. Наприклад, сортування вставками може бути ефективнішим для частково відсортованих даних.

Досвід вирішення задачі сортування даних показує наступне:

Для невеликих обсягів даних можна використовувати прості алгоритми (сортування бульбашкою або вставками), а для великих обсягів (десятки тисяч елементів і більше) найкраще звернутися до більш ефективних алгоритмів (швидке, пірамідальне, сортування злиттям і т.д.).

Слід зважати на наявність або відсутність вимоги до стійкості (Стійкість алгоритму показує чи стабільно веде себе алгоритм у випадку багаторазового використання для однакових наборів даних. У випадку сортування це стосується питання чи відрізняє алгоритм однакові елементи масиву, які знаходяться на різних позиціях).

Як бачимо, вибір відповідного алгоритму сортування вимагає врахування різних факторів. Дотримуючись наведених порад, ви зможете оптимальним чином вибрати алгоритм для вирішення конкретного завдання.

Завдання до лабораторної роботи 3

1. Дайте визначення для поняття структура даних (як альтернатива одиночним даним). Наведіть приклади структур даних, вкажіть їх тип. Дайте визначення масиву та коротко опишіть його основні характеристики. Наведіть ваші міркування стосовно доцільності та правил використання способів організації доступу до елементів масиву. Способи статичного та динамічного розміщення масиву наведено у прикладах 1 та 2 (див. нижче).

Приклад 1 програми, що реалізує метод бульбашки для сортування масиву (статичне розміщення масиву):

```
#include <iostream>
using namespace std;

// метод бульбашки реалізуємо функцією
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {            // Перебираємо проходи

        for (int j = 0; j < n - 1; j++) {    // Перебираємо пари сусідів
            // Якщо пара стоїть неправильно, міняємо в ній елементи місцями
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int main() {
    int n;
    cout << "Введіть кількість елементів масиву: ";
    cin >> n;

    int arr[n];
    cout << "Введіть елементи масиву: " << endl;
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    // Викликаємо функцію сортування
    bubbleSort(arr, n);

    cout << "Відсортований масив: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Приклад 2 програми, що реалізує метод бульбашки для сортування масиву (динамічне розміщення масиву):

```
#include <iostream>
    using namespace std;
```

```
int main(){
    setlocale(LC_ALL, "");
    int n = 0;
    cout<<"вкажіть розмір масиву: ";
    cin>>n;
    int *arr = new int[n];
    int *arrBubble = new int[n];

//fill array
    for (int i=0; i<n; i++){
    arr[i] = rand()%100;
    arrBubble[i] = arr[i]; }

//visualing array before sorting
    for (int i = 0; i < n; i++){
    cout<<arr[i]<<" "; }
    cout<<endl;

//array sorting
    for(int i = 0; i < n; i++){
    for(int j = 0; j < n - 1; j++){
    if (arrBubble[j]>arrBubble[j+1]){
    int temp = arrBubble[j];
    arrBubble[j] = arrBubble[j+1];
    arrBubble[j+1] = temp; } } }

// visualing array after sorting
    for (int i = 0; i < n; i++){
    cout<<arrBubble[i]<<" "; }
    cout<<endl;

//cleaning dynamic memory
    delete [] arr;
    delete [] arrBubble;
    return 0; }
```

2. На прикладі задачі про сортування елементів одновимірного масиву (цілочисельний тип, 7 елементів, за спаданням) продемонструйте та порівняйте два методи її вирішення – модифікація простих обмінів (бульбашки) та вибору.

Зверніть основну увагу на традиційні підходи до модифікації відомих методів сортування. Для простоти порівняння методів логічно використати показник часу (або кількості елементарних операцій).

3. Використавши ідею сортування як інструмент, розв’яжіть наступну задачу: для натурального числа з діапазону 1..1000000000 отримати максимальне та мінімальне числа, які можна побудувати з цифр заданого числа.

Має сенс виокремити кожну цифру числа, занести до масиву, відсортувати, зібрати з цих цифр нове число.

4. Реалізуйте методи повного послідовного перебирання та бінарного

пошуку для наступної задачі: у масиві відшукати значення та місцезнаходження елемента з вказаним значенням. Вкажіть переваги та недоліки кожного з підходів залежно від ситуації.

Під час комп'ютерного експерименту має сенс розглянути окремі випадки – елемент є, і він стоїть на початку; елемент є, і він стоїть у кінці; елемент є і він дублюється; елемента у масиві взагалі нема.

❓❓ Питання для самоконтролю

1. У чому полягає перевага використання групових даних у сучасному програмуванні, в контексті теми роботи?
2. Як ви поясните, чому масиви та структури (записи або кортежі) – найбільш популярні структури даних у програмуванні?
3. Які основні відмінності у статичному та динамічному способі використання структур даних?
4. Чому ідея використання масивів при зростанні розмірності масиву призводить до нових витрат часу?
5. Опишіть скорочену технологію використання масиву у програмі (узагальнено або на конкретному прикладі).
6. На прикладі задачі сортування одновимірного масиву порівняйте витрати часу при використанні методу вибору та методу підрахунку (опис основних ідей цього методу пропонується відшукати самостійно).
7. На власному простому прикладі поясніть як конкретизація даних задачі про сортування даних може вплинути на вибір методу вирішення задачі сортування.
8. У чому виграш або навпаки програш часу, коли у програмі для конкретної задачі нема групових даних (тобто всі дані описуються як одиночні) і навпаки?
9. На основі описаного у тексті алгоритму сортування вибором максимального елемента запропонуйте як має виглядати алгоритм сортування вибором мінімального елемента.

Лабораторна робота 4. СТРУКТУРИ ДАНИХ (ДИНАМІЧНІ). ОСОБЛИВОСТІ БУДОВИ ДИНАМІЧНИХ СТРУКТУР (СПИСОК ОДНОЗВ'ЯЗНИЙ ТА ДВОЗВ'ЯЗНИЙ, СТЕК, ЧЕРГА). РОБОТА ЗІ СПИСКАМИ ТА СТЕКАМИ

Мета: Засвоїти фундаментальні відмінності між статичними та динамічними структурами даних, а також опанувати критерії обґрунтованого вибору конкретної структури для ефективного розв'язання прикладних інформаційних задач. Визначити особливості побудови типових динамічних структур – список, стек тощо та етапів їх опрацювання

🕮🔄 Теоретичні відомості та методичні рекомендації

Ключові поняття: 🔑🔑

- Інформація та дані
- Тип даних
- Структура даних
- Статичні та динамічні структури даних
- Базові операції над даними різних структур

Під час роботи певної програми значення деякої статичної змінної звичайно може змінюватися, але кількість оголошених статичних змінних не змінюється. Це не завжди зручно. Наприклад, якщо програма призначена для введення та обробки даних про учнів класу, а для збереження цих даних використовується звичайний масив, то визначаючи розмір масиву, приходиться орієнтуватися на деяке конкретне граничне значення кількості учнів у класі. При цьому очевидно, якщо реально учнів в класі менше цього граничного значення, то пам'ять ПК використовується неефективно. Якщо ж учнів більше, то таку програму використовувати взагалі неможна.

У таких задачах зручно використовувати структури (подібні до масивів), в яких кількість елементів може змінюватися. Такими структурами є зв'язані списки. Зв'язаний список нагадує масив, в якому кількість елементів змінюється під час роботи програми. Зв'язаний список можна побудувати використовуючи динамічні змінні. Динамічні змінні можна створювати під час роботи програми («на ходу») і користуватися ними за допомогою так званих змінних-вказівників. Змінна-вказівник – спеціальна статична змінна, така, в якій зберігається не деяке конкретне значення (наприклад типу int, float), а адреса реальної комірки пам'яті, де такі дані безпосередньо зберігаються.

Має сенс пам'ятати:

1. Статична змінна має назву, а динамічна змінна назви не має.
2. До статичної змінної можна звернутися за її назвою, а до динамічної змінної – за її адресою (тобто за вказівником, де ця адреса міститься).
3. Змінна-вказівник може вказувати лише на динамічну змінну і не може вказувати на статичну.

Динамічні дані, в яких елементи зв’язані друг з другом за допомогою покажчиків називають зв’язаними динамічними даними.

Лінійний список – це скінченна послідовність однотипних елементів, де кожен елемент (крім першого та останнього) має одного попередника і одного наступника.

Це абстрактне поняття. Можна уявити список як чергу людей: кожен знає, хто стоїть перед ним і хто за ним.

Класичне формальне визначення лінійного списку як абстрактної структури даних, часто зустрічається у фундаментальних працях з програмування (наприклад, у Дональда Кнута [7]), оскільки воно чітко відокремлює логічну структуру (як ми бачимо дані) від її фізичної реалізації в пам'яті.

Отже, **лінійний список** – це множина, що складається з `n ≥ 0` елементів `X[1], X[2], ..., X[n]`, структурні властивості якого, по суті, обмежені лише лінійним (одновимірним) відносним положенням елементів, тобто такими умовами, що якщо `n > 0`, тоді `X[1]` є першим елементом; якщо `1 < k < n`, тоді для `k`-го елемента `X[k]` попередній `X[k-1]`, а за ним йде `X[k+1]`; `X[n]` є останнім елементом.

Операції, що ми маємо право виконувати з лінійними списками, включають, наприклад, наступні:

1. Отримати доступ до k-го елемента списку, щоб проаналізувати та/або змінити вміст його полів.
2. Включити (додати) новий елемент безпосередньо перед k-м елементом.
3. Виключити (видалити) k-й елемент.
4. Поєднати два (або більше) лінійних списків в один список.
5. Розбити (розділити) лінійний список на два (або більше) списків.
6. Створити копію лінійного списку.
7. Визначити кількість елементів списку.
8. Виконати сортування елементів списку за зростанням за певними полями елементів.
9. Знайти у списку елемент з певним заданим значенням у певному полі елемента.

Спеціальні випадки `k = 1` і `k = n` в операціях (1), (2) та (3) виокремлюються, оскільки в лінійному списку простіше отримати доступ до першого та останнього елементів, ніж до довільного елемента.

У комп’ютерних програмах рідко використовуються усі 9 операцій.

На **фізичному** рівні лінійний список можна реалізувати двома основними способами:

– статично (через масив); елементи лежать у пам'яті один за одним в одному суцільному блоці;

– динамічно (через зв'язки); елементи розкидані в пам'яті, але «знають» адреси один одного.

Лінійний зв'язаний список – це конкретний спосіб реалізації лінійного списку в пам'яті комп'ютера, де компоненти не обов'язково розташовані поруч.

Кожен компонент такого списку називається вузлом (node) і складається з двох частин:

– інформаційне поле – самі дані, які зберігаються;
– поле зв'язку (покажчик) – адреса в пам'яті наступного елемента.

Кожен компонент списку містить покажчик на наступний (або на наступний і попередній) компонент; У останнього компонента це поле має спеціальне значення (NULL), що сигналізує про завершення списку. Доступ до першого компонента здійснюється за допомогою покажчика на нього, а доступ до кожного наступного компонента – з використанням покажчика, який зберігається у попередньому компоненті. Перший компонент списку називається його вершиною або головою.

Над зв'язними лінійними списками традиційно (за статистикою) виконуються такі дії:

- додавання нового компонента на початок списку;
- додавання нового компонента в кінець списку;
- вставка нового компонента між двома будь-якими компонентами списку;
- видалення будь-якого компонента зі списку.

Зв'язні лінійні списки поділяють на такі різновиди:

– однозв'язні лінійні списки (лінійні та циклічні);
– двозв'язні лінійні списки (лінійні та циклічні);

Однозв'язний лінійний список – це список, в якому попередній компонент посилається лише на наступний (рис. 4.1). Рух можливий тільки в один бік.

Рисунок 4.1 – Структура однозв’язного лінійного списку

Однозв'язний циклічний список – це однозв'язний лінійний список, в якому останній компонент посилається на перший.

Двозв'язний лінійний список – це список, в якому попередній компонент посилається на наступний, а наступний – на попередній (рис. 4.2). Це дозволяє рухатися списком в обох напрямках.

Рисунок 4.2 – Структура двозв’язного лінійного списку

Двозв'язний циклічний список – це двозв'язний лінійний список, в якому останній компонент посилається на перший, а перший компонент – на останній.

Опишемо тип компонентів однозв'язного лінійного списку (рис. 4.1). Кожний його компонент складається з кількох інформаційних полів та покажчика на наступний компонент. Отже, компонент зв'язного лінійного списку є записом (структурою, кортежем). Інформаційні поля компонента списку можуть бути змінними будь-яких типів, а покажчик має бути покажчиком на запис того типу, якому належать компоненти списку. Покажчик в останньому компоненті лінійного списку має значення nil (або NULL) – так позначається кінець списку.

Наведемо приклади оголошення типу компонента однозв'язного лінійного списку на двох мовах програмування – для порівняння. Тут подається найпростіший випадок, коли компонент списку містить лише одне інформаційне поле.

```
//1 приклад
#include <cstdint> // для uint8_t
struct TElem; // попереднє оголошення структури
using TPtr = TElem*; // визначення типу покажчика
struct TElem {
    uint8_t Inf;  // інформаційне поле, аналог Byte
    TPtr Link;    // покажчик на наступний компонент
};

{2 приклад}
type
TPtr=^TElem; // тип покажчика на компонент списку
TElem=record //тип компонента
Inf:Byte; //інформаційне поле
Link:TPtr; //покажчик на наступний компонент
end;
```

Правило послідовності оголошень вимагає, щоб кожен ідентифікатор був описаний, раніш ніж він буде використовуватися для інших оголошень. Однак у наведеному прикладі, як би не розташовувались оголошення типів покажчика `TPtr` та елемента `TElem`, це правило виконано не буде. Тому для оголошення типів компонентів динамічних структур зроблено виключення з правил: Тип покажчика на компонент однозв'язного лінійного списку має бути оголошений

перед оголошенням типу компонента списку.

Дані зберігаються в зв'язаному списку динамічно – кожен вузол створюється по мірі необхідності.

Зв'язані списки переповнюються лише у випадку, якщо в системі не вистачає пам’яті, щоб задовольнити запит на виділення динамічної пам'яті.

Найчастіше для роботи зі списками програмно реалізуються такі функції `isEmpty`, `insert`, `delete`, `printlist`.

Функція `isEmpty` визначає, чи є список пустим (вказівник на перший вузол списку рівень `NULL`). Якщо список пустий, то повертається значення 1 (`true`), або у протилежному випадку 0 (`false`).

Функція `printlist` – роздруковує (виводить по черзі елементи) список. Функція insert – передає адресу списку і символ, який необхідно вбудувати (вставити) в нього. Адреса списку необхідна, щоб встановити де саме знаходиться початок списку.

Функція `delete` отримує адресу вказівника на початок списку і символ, який потрібно видалити.

Структури даних, які визначають правила додавання та видалення елементів:

– **стек** – лінійна структура даних, яка працює за принципом LIFO (Last In, First Out – «останнім прийшов, першим пішов»);

– **черга** – лінійна структура даних, яка працює за принципом FIFO (First In, First Out – «першим прийшов, першим пішов»).

Стек і чергу можна реалізувати за допомогою зв'язного списку, а можна – за допомогою масиву.

Іноді аналогія з переключенням залізничних шляхів, яку запропонував всесвітньо відомий теоретик і практик у сфері комп’ютерних наук Е. Дейкстра (рис. 4.3), допомагає зрозуміти механізм роботи стеку.

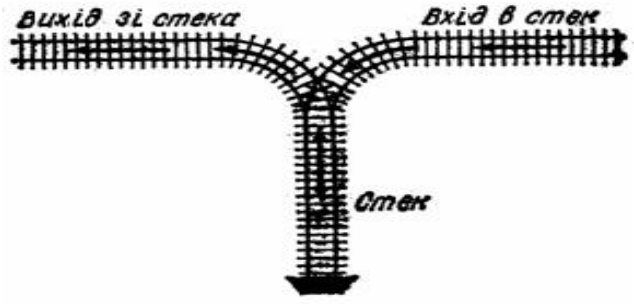


Рисунок 4.3 – Стек, поданий у вигляді залізничного роз’їзду

Зі стеку ми завжди виключаємо «молодший» елемент , тобто той, що був включеним пізніше за всіх інших (рис. 4.4). Стек схожий на стопку з книгами, покладеними одна на іншу – щоб взяти певну книгу треба зняти всі книги, що лежать на ній, а покласти нову книгу можна лише зверху всієї стопки.

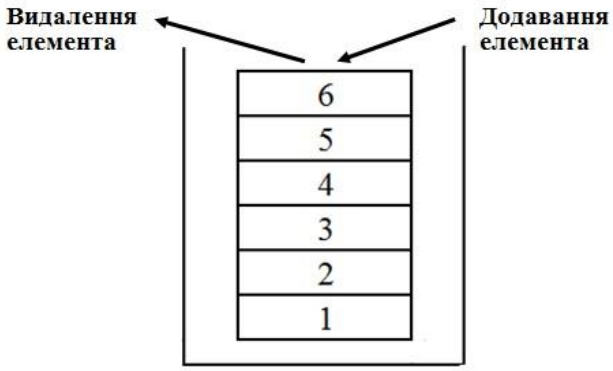


Рисунок 4.4 – Структура стеку

Стеки часто зустрічаються у життєвій практиці. Простим прикладом може бути ситуація, коли ми створюємо, передивляємося, модифікуємо множину завдань на день, виконуємо ці завдання, додаючи (нові) та видаляючи (вже виконані) елементи з цього списку, доки він не стане пустим.

Аналогічно, при виконанні комп'ютерної програми з підпрограмами процес входів у підпрограми та виходів з них також реалізує ідею стеку. Стеки корисні при обробці ситуації з вкладеннями, наприклад, обробка (обчислення або оптимізація запису) арифметичних виразів. Так задачу перетворення арифметичного виразу з інфіксної у постфіксну форму запису (для порівняння зверніть увагу на приклади запису виразу у префіксній форми + a b, інфіксній формі a + b, постфіксній формі a b +) можна вирішити, якщо для роботи з елементами цього арифметичного виразу використовується стек.

Стеки часто використовують у ситуаціях з алгоритмами, що мають явний або неявний рекурсивний характер.

При описі алгоритмів, що використовують стеки, прийнята спеціальна термінологія; Кажуть, що ми розміщуємо елемент на вершину стеку або знімаємо елемент з вершини стеку (рис. 4.4). На дні стеку знаходиться найменш доступний елемент, і він не може бути видаленим, доки не будуть видалені всі інші елементи. Також кажуть, що:

– елемент занурюється (push down) в стек, якщо він (елемент) додається на вершину стеку;

– елемент спливає (pop up), якщо він (елемент) видаляється з його вершини.

Реалізація стеку та операцій зі стеком на послідовному розподілі пам'яті.

Простий та логічний спосіб зберігання лінійного списку в пам'яті ПК зводиться до розміщення елементів цього списку у послідовних комірках пам'яті, один елемент за другим і т.д. Тоді наступна комірка певного елемента списку вираховується за правилом

`X[j] = L0 + c*j`, (зазвичай `c = 1`) (1)

де `L0` є константою – «базовою адресою» (адресою гіпотетичного елемента `X[0]`.

Послідовний розподіл зручний при роботі зі стеком. Для цього достатньо мати змінну T, яку звуть вказівником на стек. Коли стек пустий, T=0. Щоб розмістити новий елемент Q у стек, необхідно встановити

```
T←T+1; X[T]←Q                                    (2)
```

Та, якщо стек не пустий, тоді ми можемо встановити значення змінної Q рівною значенню, що міститься у верхньому елементі та виключити цей елемент діями, протилежними до (2):

```
Q ←X[T]; T←T-1                                   (3)
```

Звичайно, що наведені ідеї (1-3) занадто ідеалізовані, оскільки передбачається, що не виникає колізій.

Коли ми виключили (видалили) елемент зі стеку, вважалося, що хоч би один елемент там є. Коли ми включали (додавали) елемент в стек, ми вважали, що для нього є місце в пам'яті. Але, зрозуміло, при методі (2), (3), в машинній програмі значення T не може перевищувати певну максимальну величину.

Нижче показано, як мають бути переписані розглянуті дії для загального випадку, коли ці обмеження можуть і не виконуватися:

```
T←T+1; якщо T>M тоді ПЕРЕПОВНЕННЯ, інакше X[T]←Q   (2a)
```

(включити в стек)

```
Якщо ж T=0 тоді БРАК РЕСУРСІВ, інакше Q←X[T]; T←T-1   (3a)
```

(виключити зі стеку)

Невирішеною залишається питання що робити, коли виникає ПЕРЕПОВНЕННЯ або БРАК РЕСУРСІВ ДЛЯ ЗБЕРІГАННЯ?

БРАК РЕСУРСІВ виявляють у випадку, коли ми намагаємося виключити неіснуючий елемент.

Наприклад, ми могли б багаторазово виключати елементи до того часу, доки не виникне БРАК РЕСУРСІВ. Однак ПЕРЕПОВНЕННЯ у більшості випадків указує на помилку; тобто ресурс пам'яті вже вичерпано, але існує ще інформація, яку необхідно додати в список. У такій ситуації зазвичай повідомляють, що програма не може продовжувати свою роботу з причини відсутності вільної пам'яті та виконання програми завершується.

Черга – це однозв'язний лінійний список, в якому компоненти додаються в кінець списку, а видаляються з вершини, тобто з початку списку.

У **черзі** вузли видаляються тільки з «голови», а додаються тільки в «хвіст» черги. З цієї причини черги часто називають структурами виду «першим прийшов – першим вийшов» (FIFO). Операції постановки до черги і видалення з черги називають enqueue (поставити в чергу) і dequeue (виключити з черги).

Простим прикладом черги в комп'ютерних системах є перелік задач, які поступово обслуговуються (виконуються) процесором (або окремим його ядром). У кожний конкретний момент часу може обслуговуватися лише одна задача (виконуватися лише одна команда), а інші стають до черги. Кожен, хто стоїть в черзі, поступово переміщується до її початку по мірі обслуговування попередніх елементів черги.

Дерево (від англ. tree) – в комп’ютерних науках та програмуванні одна з популярних структур даних. Важливими операціями на деревах є:

- обхід вершин в різному порядку,
- перенумерація вершин,
- пошук елемента,
- додавання елемента у визначене місце в дереві,
- видалення елемента,
- видалення цілого фрагмента дерева,
- додавання цілого фрагмента дерева,
- перетворення фрагментів дерева,
- знаходження кореня для будь-якої вершини.

Найбільшого розповсюдження ці структури даних набули в тих задачах, де необхідне маніпулювання з ієрархічними даними, ефективний пошук в множині даних, структуроване зберігання або модифікація даних.

Під **динамічними структурами даних** розуміються такі, розмір яких заздалегідь невідомий і (чи) змінюється в процесі виконання програми (чи розмір структури перевищує якусь конкретну межу, наприклад, 64 Кбайт). Зв'язані структури є динамічними. До них відносять: списки, стеки, черги, графи, дерева.

Динамічне розміщення даних передбачає використання динамічної пам'яті. Динамічна пам'ять – це та оперативна пам'ять комп'ютера, що надана програмі при її роботі за винятком сегмента даних, стека, пам'яті використовуваної системними і резидентними програмами, і власне тілом самої програми, що виконується. Розмір динамічної пам'яті можна встановлювати з середовища розробки чи із самої програми спеціальними директивами компілятора.

Зв'язана організація пам'яті визначає багато структур, зв'язки між елементами яких можна реалізувати покажчиками. Кожен елемент такої структури (об'єкт) має, таким чином, властивість «мати зв'язок» з іншим елементом (елементами), на який указує значення цієї властивості. Зв'язана організація пам'яті може використовуватися і для зберігання статичних структур даних, але оскільки реалізація зв'язків через посилання дає можливість використовувати динамічні механізми створення/знищення об'єктів, основним застосуванням зв'язаної організації є динамічне моделювання об'єктно-орієнтованих систем. Таким чином, динамічні структури об'єктів характеризуються наявністю спеціальної властивості: «мати змінний склад елементів структури». Ця властивість дозволяє будь-яку динамічну структуру розглядати як асоціацію зв'язаних об'єктів змінного складу. Вони можуть бути побудовані на основі часу модифікації складу елементів асоціації (наприклад, правило «першим прийшов - першим вийшов» добре відомо тим, хто бував в чергах: кожен, хто тільки но прийшов у чергу стає останнім членом цієї асоціації). Такий порядок визначається відносинами передування: «предок-нащадок», «попередній-наступний» тощо. Ця властивість робить основною реалізаційною структурою асоціації лінійний список.

Як вже зазначалося, якщо до початку роботи з даними неможливо

визначити обсяг необхідної пам'яті, пам'ять буде виділятьсь по мірі необхідності окремими блоками, які зв'язані один з одним вказівниками. Такий спосіб організації даних називається динамічними структурами даних, оскільки їх розмір змінюється під час виконання програми. Із динамічних структур в програмах найчастіше використовують лінійні списки, стеки, черги, бінарні дерева. Вони відрізняються способом зв'язку окремих елементів та переліком дозволених для використання операцій.

Динамічні структури широко використовуються для розв'язування задач сортування, оскільки впорядкування динамічних структур не потребує перестановки елементів, а лише змінює вказівник на ці елементи.

Структури – це продуктивні типи даних, вони створюються з об'єктів інших типів. Наведемо ще один приклад опису структури:

```
struct card {
char*face;
char*suit;
};
```

Ключове слово `struct` – визначає що це структура (група певним чином зв’язаних елементів). Ідентифікатор `card` є ім'ям структури. Ім'я структури використовується з ключовим словом `struct` для оголошення (опису) змінних типу структура. У прикладі тип структури `struct card`. Змінні, оголошені в середні дужок структури, є елементами структури. Елементи однієї структури повинні мати унікальні ідентифікатори (імена), але дві різні структури можуть мати одинакові імена, і це не, викличе конфліктів.

Розглянемо ще один приклад структури, яка на себе посилається.

В якості елемента тут виступає покажчик (вказівник), який посилається на структуру того ж типу.

```
struct node {
int data
struct node * nextPtr;
};
```

Як видно з прикладу, структура типу `struct node` складається з двох елементів – цілого `data` і вказівника `nextPtr`. Елемент `nextPtr` - вказівник на структуру типа `struct node` – структура того ж самого типу, що нами оголошена, звідси і термін «структура, що посилається на себе».

Елемент `nextPtr` – інколи називають зв'язкою, `nextPtr` можна використовувати для того, щоб зв'язати структуру типу `struct node` з другою структурую того ж типу. За необхідністю за таким принципом можна отримати корисні для конкретної задачі структури даних, такі як списки, черги, стеки, дерева.

Завдання до лабораторної роботи 4

1. Дайте визначення для поняття динамічна структура даних (як альтернатива статичним структурам даних). Наведіть приклади динамічних структур даних. Дайте визначення списку та стеку, коротко опишіть основні характеристики цих структур. Наведіть ваші міркування щодо: доцільності використання зазначених динамічних структур даних, особливих правил, що стосуються базових дій з компонентами зазначених структур.

2. Реалізуйте стек та базові операції зі стеком (створення стеку, додавання елемента, видалення елемента, виведення значень елементів стеку), використовуючи масив.

Має сенс виокремити кожну з перелічених задач та оформити окремими функціями.

Роботу програми слід перевірити на тестовому наборі рядкових даних. Тестові дані слід читати з підготованого текстового файлу, в якому попередньо розміщено декілька речень. Речення слід «порізати» на слова (ігноруючи розділові знаки), а слова у порядку їх читання розмістити в стек. Також слід виводити стан стеку (варіанти поточного стану стеку такі – 1) пустий, 2) непустий і тоді виводяться всі його елементи, 3) переповнення стеку) на консоль після виконання кожної операції. Також після виконання кожної операції запитувати завершення роботи.

3. Програмно реалізуйте однозв’язний список (без використання масиву, у списку довільна кількість елементів, кожен елемент містить прізвище, рік народження, стать, середній бал студента за семестр) та базові операції зі списком (створення-видалення списку, додавання елемента, видалення елемента, виведення елементів списку) та додаткові операції: знаходження прізвище наймолодшої людини, виведення даних студентів, що мають середній бал > 90.

При розробці програм середовище розробки студент обирає самостійно.

Роботу програми бажано реалізувати з використанням меню доступних дій користувача над списком та виведення проміжних та остаточних результатів.

❓❓ Питання для самоконтролю

1. Чи завжди динамічні структури даних кращі за статичні?
2. Поясніть необхідність існування та використання двох видів списків (одно- та двохзв’язних).
3. Як програмування на мові Python пов’язується з роботою зі стеками і як це впливає на швидкість остаточного отримання рішення задач?
4. Який має бути набір базових операцій над списками і чому він саме такий?
5. Опишіть скорочено технологію організації роботи стеку з використанням масиву.

6. Чи доречно реалізовувати стек шляхом використання масиву?
7. Опишіть сутність двох особливих станів списку та специфіку їх програмної обробки.
8. У чому виграш або навпаки програш часу, коли у програмі для конкретної задачі використовується вказівник на масив або масив вказівників?
9. Чому у визначенні лінійного списку існує обмеження $n \geq 0$?
10. Наведіть приклади запису складних арифметичних виразів з використанням префіксної, інфіксної та постфіксної форм.
11. Чому префіксну форму також називають польською нотацією?
12. Наведіть приклади задач та доречних для їх розв’язання типів структур даних.

Лабораторна робота 5. ПОДАННЯ ГРАФІВ. СТВОРЕННЯ ТА ВЗАЄМОПЕРЕТВОРЕННЯ МАТРИЦЬ СУМІЖНОСТІ ТА ІНЦИДЕНТНОСТІ. ВИКОРИСТАННЯ СПИСКУ ІНЦИДЕНТНОСТІ. АНАЛІЗ ЕФЕКТИВНОСТІ ВИКОРИСТАННЯ ПЕРЕЛІЧЕНИХ СПОСОБІВ ПОДАННЯ ГРАФІВ

Мета: Засвоїти теоретичну базу та опанувати програмні методики представлення графів у вигляді матриць та списків. Вивчити механізми взаємоперетворення способів зберігання інформації про графи та обґрунтувати ефективність їх застосування

🕮🔄 Теоретичні відомості та методичні рекомендації

Ключові поняття: 🔑🔑

- Дані
- Структура даних
- Граф
- Способи подання графів
- Списки інцидентності
- Матриця суміжності
- Ефективність різних способів подання графів

Граф `G` визначається двома множинами - множиною вершин `V` та множиною ребер або дуг (пар вершин) `E`: `G=(V, E)`. Якщо пара вершин неупорядкована (тобто їх порядок не важливий), то її прийнято називати ребром, а якщо упорядкована - дугою. Граф, що має лише ребра називається неорієнтованим графом, а граф, що має лише дуги - орієнтованим графом (або орграфом).

Візуально граф можна показати як точки, що називають вершинами графа, та лінії, що з'єднують вершини і називають ребрами графа, або спрямовані від вершини до вершини лінії, що називають дугами графа.

Дві вершини `x` та `y`, що з'єднані ребром `(x, y)`, називають суміжними вершинами. Якщо вершини з'єднані не ребром, а дугою `(x, y)`, то вершина `x` суміжна вершині `y`, а от зворотної суміжності немає.

Два ребра називають суміжними ребрами, якщо вони мають спільну вершину.

Будь-які дві вершини графа, що з’єднані ребром графа називаються інцидентними.

Кожному ребру чи вершині графа можна призначити (поставити у відповідність) число – вагу (або вартість). Тоді граф буде зватися зваженим. Вага вершини характеризує вершину, вага ребра характеризує відношення між двома вершинами. Наприклад, для графа автомобільних доріг вага ребра може означати довжину дороги від одного перехрестя до іншого.

Залежно від ситуації (особливості задачі) використовують наступні основні 4 способи подання графів:

1) Граф G =(V,E) зручно зображувати за допомогою рисунка на площині, який називають **діаграма графа** G. Вершинам графа G ставляться у відповідність точки площини; точки, що відповідають вершинам v і w, з'єднуються лінією (відрізком або кривою) тоді і тільки тоді, коли v і w суміжні вершини. Зрозуміло, що діаграма графа змінюватиме свій вигляд у залежності від вибору відповідних точок на площині.

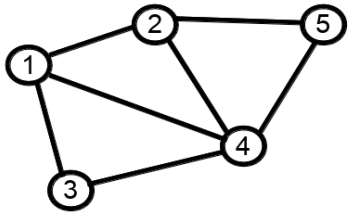
2) **Списки інцидентності**. Списки складаються з рядків. Перший рядок містить перелік вершин, інцидентних першій вершині, другий рядок містить перелік вершин, інцидентних другій вершині і т.д.

Графи можна задавати також за допомогою матриць.

3) Занумеруємо всі вершини графа G натуральними числами від 1 до n. **Матриця суміжності** A графа G - це квадратна n*n-матриця, в якій елемент aij (i-го рядка і j-го стовпчика) дорівнює 1, якщо вершини vi та vj (з номерами відповідно i та j) суміжні, і дорівнює 0 у протилежному випадку. Матриці суміжності неорієнтовних графів симетричні, а для орієнтованих - ні.

4) Занумеруємо всі вершини графа G числами від 1 до n і всі його ребра числами від 1 до m. **Матриця інцидентності** B графа G - це n*m-матриця, в якій елемент bij (i-го рядка і j-го стовпчика) дорівнює 1, якщо вершина vi з номером i інцидентна ребру ej з номером j, і дорівнює 0 у протилежному випадку.

Наприклад, є граф з такою діаграмою:



Тоді списки інцидентності будуть наступними:

2 3 4
1 4 5
1 4
1 2 3 5
2 4

А матриця суміжності матиме вигляд:

	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	1	1
3	1	0	0	1	0
4	1	1	1	0	1
5	0	1	0	1	0

Для цього ж випадку матриця інцидентності (одна з можливих) така:

	1	2	3	4	5	6	7
1	1	1	1	0	0	0	0
2	0	0	1	0	1	1	0
3	1	0	0	1	0	0	0
4	0	1	0	1	1	0	1
5	0	0	0	0	0	1	1

Трохи проаналізувавши вже надану інформацію нескладно сформулювати ідеї, а потім розробити алгоритми перетворення списків у матриці, матриць у списки, одного типу матриці в іншу. Спосіб подання надає можливості щодо автоматизації обробки даних у задачах, де можна застосувати графи.

Завдання до лабораторної роботи 5

1. Опишіть власними словами, на обраному вами прикладі, які можна використати способи подання графу. Поясніть основні ідеї алгоритмів взаємоперетворення обраних вами способів.

2. Підготуйте власний візуальний приклад графу (кількість вузлів від 7 до 10). За цим прикладом створіть списки інцидентності та запишіть їх у файл.

3. Реалізуйте програмно читання даних з файлу та побудову відповідно вашій задачі матриці інцидентності.

4. Реалізуйте програмно читання даних з файлу та побудову відповідної вашій задачі матриці суміжності.

5. Реалізуйте взаємоперетворення способу подання графу зі списку (з матриці суміжності створюємо матрицю інцидентності, або з матриці інцидентності створюємо матрицю суміжності, або з матриці суміжності створюємо списки інцидентності, або з матриці інцидентності створюємо списки інцидентності, або на екрані створюємо зображення графу, взявши за основу один з формалізованих способів його подання).

Для цього випадку логічно розробити відповідні функції конвертації даних.

❓ ❓ Пропонуються питання та завдання для самоконтролю

1. Які переваги та недоліки має графічний спосіб подання графу?
2. Чому списки інцидентності, матриці суміжності та матриці інцидентності стали альтернативою графічному способу подання графу?
3. Як способи подання графу з тих, що перелічені (списки, матриця суміжності, матриця інцидентності) пов’язані зі структурами даних (якими саме)?
4. Як будуть виглядати матриці суміжності та інцидентності у випадку орієнтованого графу?
5. Чому до продемонстрованого прикладу матриці інцидентності є коментар «одна з можливих»?
6. Які переваги та недоліки мають матричні способи подання графу?
7. Які ви знаєте вже готові інструменти (бібліотеки підпрограм, сервіси) для вирішення задач на графах?
8. Поясніть, чому саме матриця суміжності орієнтованого графу симетрична.
9. Поясніть, чому матриця суміжності неорієнтованого графу не є симетричною.

Лабораторна робота 6. РОЗРОБКА ПРОГРАМ З РЕАЛІЗАЦІЄЮ СПЕЦІАЛЬНИХ АЛГОРИТМІВ ОБРОБКИ ГРАФІВ

Мета: Розглянути «класичні» задачі теорії графів, алгоритми їх вирішення, зосередити увагу на питаннях вибору структур даних для зберігання інформації про граф, розробити та протестувати відповідні задачам програми

🕮🔄 Теоретичні відомості та методичні рекомендації

Ключові поняття: 🔑🔑	• Граф • Способи подання та обробки графів • Класичні задачі на графах та алгоритми їх розв’язування • Алгоритм пошуку найкоротшого шляху між вершинами в графі • Хвильовий алгоритм

Лабораторна робота потребує попереднього розгляду декількох відомих задач і відповідних їм алгоритмів.

Алгоритм Дейкстри

Автор алгоритму Едсгер Вібе Дейкстра, Нідерланди. Цей алгоритм призначений для пошуку найкоротшого шляху від заданої вершини графу до всіх інших вершин.

Алгоритм працює тільки для графів без ребер від'ємної довжини.

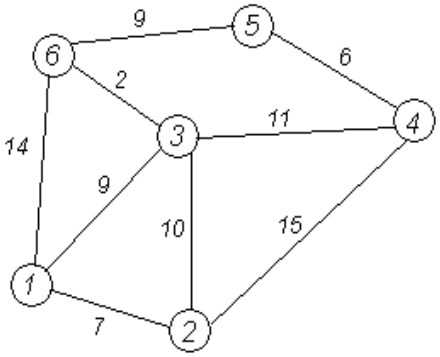
Приклади задач, які можна вирішити скориставшись саме цим інструментом:

– Надано (і на карті, і у текстовому вигляді) мережа доріг, що з'єднують населені пункти певного регіону (наприклад, Запорізька область). Знайти найкоротшу відстань від обраного населеного пункту (наприклад, Запоріжжя) до інших зазначених у списку населених пунктів, якщо рухатись можна тільки по дорогах.

– Є план міста з нанесеними на нього місцями лікарень. Знайти найближчу до нашого дому лікарню.

– Знаючи спрощену мапу України, де нанесено обласні центри та залізничні шляхи між ними, знайти мінімальну відстань, яку треба проїхати, щоб дістатися від Запоріжжя до Ужгорода.

Розглянемо цей алгоритм на наступному прикладі графу (у нас він неорієнтований, зважений, зв’язний). Вершини (вузли) пронумеровані в кружечках числами від 1 до 6, на ребрах позначена їх вага (у нашій задачі – це відстань або довжина шляху). Над кружечком позначена поточна найкоротша відстань до вершини. Будемо шукати відстані від 1-ї вершини до всіх інших.



Будемо зберігати поточну мінімальну відстань до всіх вершин з множини **v** (вершини) від даної вершини **a** і на кожному кроці алгоритму будемо намагатися зменшити цю відстань.

Спочатку встановимо відстані до всіх вершин рівними нескінченості, а до вершини **a** – нулю.

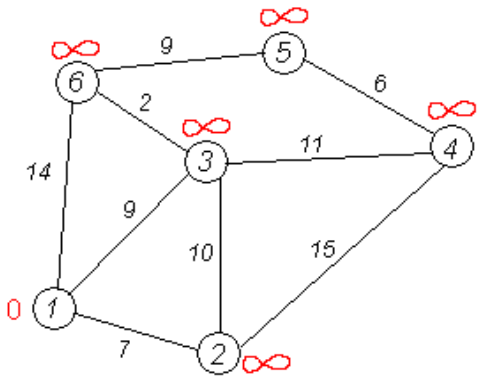
Процес роботи алгоритму розглянемо поетапно.

Етап 1

Ініціалізація. Відстань до всіх вершин ставимо∞.

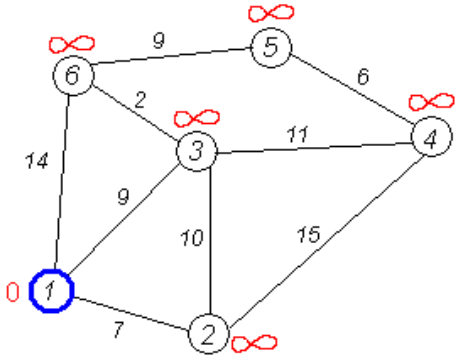
Відстань до **a = 0**.

Жодної вершини графу ще не опрацьовано.



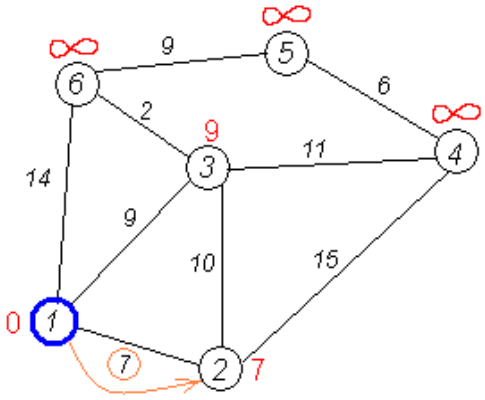
Етап 2

Знаходимо таку вершину (яку ще не опрацьовано), поточна найкоротша відстань до якої мінімальна. В нашому випадку це вершина 1. Обходимо всіх її сусідів і, якщо шлях в сусідню вершину через 1 менший за поточний мінімальний шлях в цю сусідню вершину, то запам'ятовуємо цей новий, коротший шлях як поточний найкоротший шлях до сусіда.



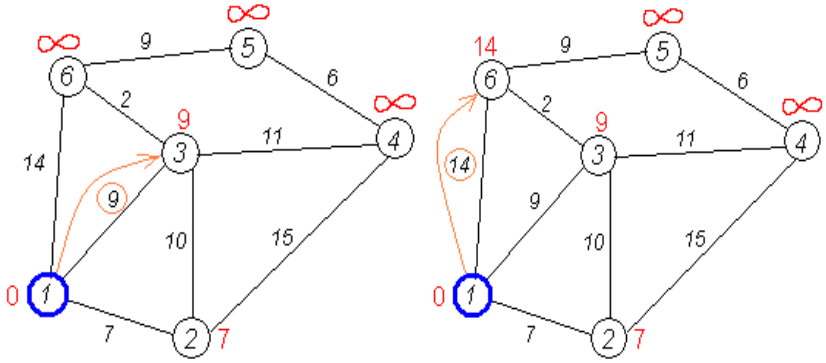
Етап 3

Перший по порядку сусід 1-ї вершини — 2-а вершина. Шлях до неї через 1-у вершину дорівнює найкоротшій відстані до 1-ї вершини + довжина дуги між 1-ю та 2-ю вершиною, тобто 0 + 7 = 7. Це менше поточного найкоротшого шляху до 2-ї вершини, тому найкоротший шлях до 2-ї вершини дорівнює 7.



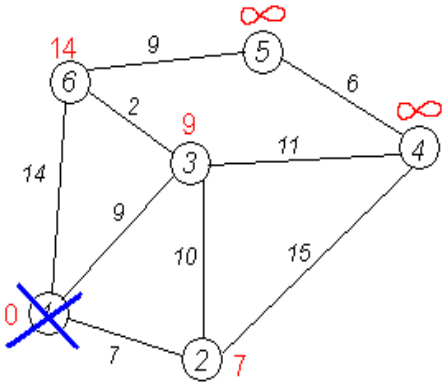
Етап 4 та 5

Аналогічні дії виконуємо з двома іншими сусідами 1-ї вершини — 3-ю та 6-ю.



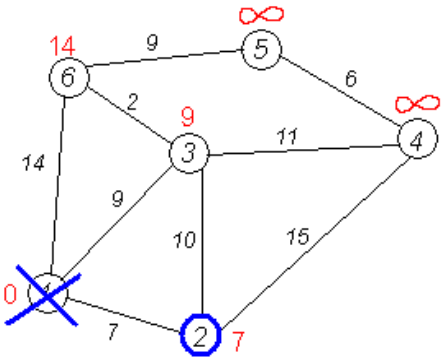
Етап 6

Усі сусіди вершини 1 перевірені. Поточна мінімальна відстань до вершини 1 вважається остаточною, викреслюємо її.



Етап 7

Практично відбувається повернення до кроку 2. Знову знаходимо «найближчу» необроблену (невикреслену) вершину. Це вершина 2 з поточною найкоротшою відстанню до неї = 7.



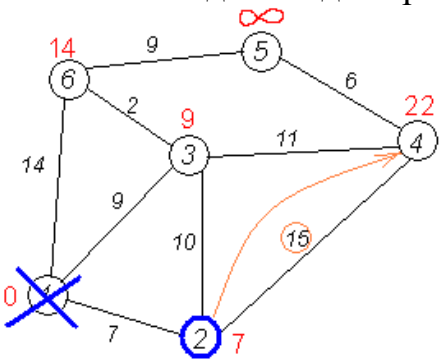
Знову намагаємося зменшити відстань до всіх сусідів 2-ї вершини, намагаючись пройти в них через 2-у. Сусідами 2-ї вершини є 1, 3, 4.

Етап 8

Перший (по порядку) сусід вершини 2 - це вершина 1. Але вона вже оброблена (викреслена на кроці 6). Тому з 1-ю вершиною нічого не робимо.

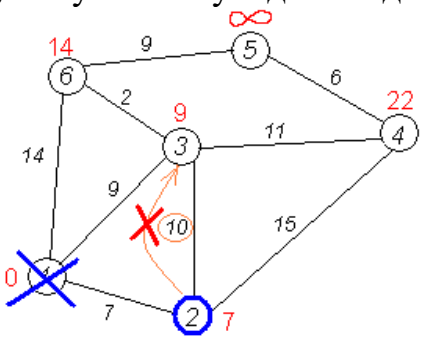
Етап 8 (з іншими вхідними даними)

Наступний сусід вершини 2 - вершина 4. Якщо йти в неї через 2-у, то шлях буде = найкоротша відстань до 2-ї + відстань між 2-ю і 4-ю вершинами = 7 + 15 = 22. Оскільки 22 < ∞, встановлюємо відстань до вершини 4 рівною 22.



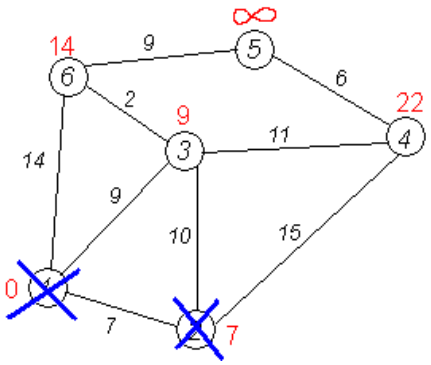
Етап 9

Ще один сусід вершини 2 - вершина 3. Якщо йти в неї через 2-у, то шлях буде = 7 + 10 = 17. Але 17 більше за відстань, що вже запам'ятали раніше до вершини 3, яка дорівнює 9, тому поточну відстань до 3-ї вершини не міняємо.



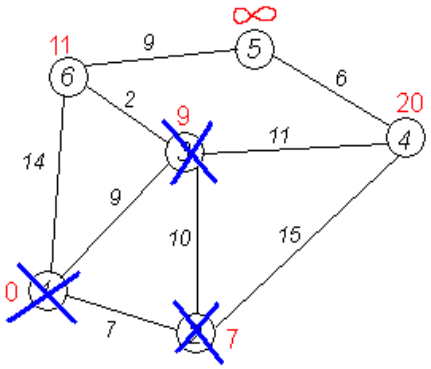
Етап 10

Всі сусіди вершини 2 переглянуті, заморожуємо відстань до неї і викреслюємо її також.



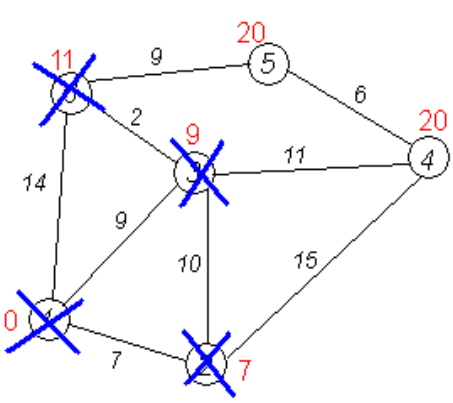
Етапи 11-15

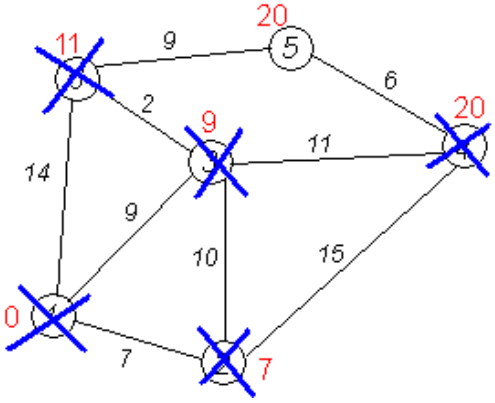
По вже «відпрацьованій» схемі повторюємо кроки 2 — 6. Тепер «найближчою» виявляється вершина 3. Після її «обробки» отримаємо такі результати:

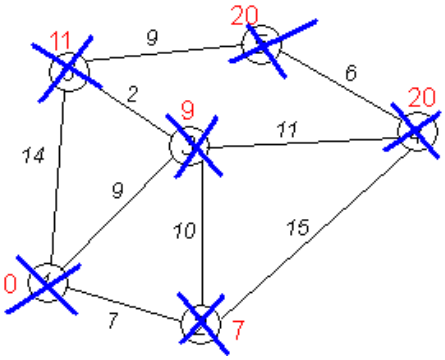


Наступні етапи

Виконуємо ті ж самі дії з вершинами, що залишилися (6, 4 і 5).







Завершальний етап

Алгоритм закінчує роботу, коли викреслені всі вершини. Результат його роботи видно на останньому рисунку: найкоротший шлях від 1-ї вершини до 2-ї становить 7, до 3-ї – 9, до 4-ї – 20, до 5-ї – 20, до 6-ї – 11.

Для програмної реалізації пропонується використати масив відстаней та масив позначок (прапорців). На початку алгоритму відстані заповнюють однаковим великим позитивним числом (більшим максимального можливого шляху в графі), а масив позначок заповнюють нулями. Потім відстань для початкової вершини вважається рівною нулю і запускається основний цикл.

На кожному кроці циклу шукаємо вершину з мінімальною відстанню і прапорцем рівним нулю. Потім ми встановлюємо в ній позначку 1 і перевіряємо всі сусідні з нею вершини. Якщо в ній відстань більша, ніж сума відстані до поточної вершини і довжини ребра, то зменшуємо його. Цикл завершується коли позначки всіх вершин стають рівними 1.

Для автоматизації отримання розв'язків усіх задач такого типу пишеться програма (як правило на мовах програмування C++ або Python – вибір мов пояснюється розвиненою мережею бібліотек підпрограм). Структурами даних для такої програми можна обрати як масиви так і списки. Отриманий програмний продукт обов'язково проходить тестування на спеціально підготованих наборах даних (тестовий набір даних складається з вхідних даних та відповідних їм вже перевірених правильних результатів).

Хвильовий алгоритм

Хвильовий алгоритм (друга назва – алгоритм Лі) – досить відомий алгоритм, що обробляє граф з ребрами одиничної довжини. Він застосовується для знаходження найкоротшого шляху в графі (але у загальному випадку знаходить лише його довжину). Нескладне пояснення роботи алгоритму пропонується прочитати самостійно за посиланням Хвильовий алгоритм – Вікіпедія https://surl.li/wlyvpl.

Завдання до лабораторної роботи 6

1. Підготуйте власний приклад зваженого графу. Для цього прикладу реалізуйте програмно алгоритм Дейкстри. Підготуйте дані для перевірки правильності роботи програми на кожному етапі вирішення задачі.

Для простоти виконання завдання використовуйте граф з кількістю вузлів від 5 до 10.

2. Підготуйте власний приклад лабіринту (у вигляді прямокутного поля з прохідними та непрохідними клітинками). Для цього прикладу реалізуйте програмно хвильовий алгоритм. Підготуйте дані для перевірки правильності роботи програми.

Схематично покажіть логіку розповсюдження хвилі. Покажіть як саме мають виглядати остаточні результати (які знайдено шляхи від точки Початок руху до точки Кінець руху).

❓ ❓ Питання для самоконтролю

1. В яких випадках має сенс використовувати графи як базову структуру даних?
2. Які способи подання графів ви запропонуєте для використання у програмах і чому саме?
3. Як будуть відрізнятися вищезгадані способи у разі використання неорієнтованих та орієнтованих графів?
4. Назвіть декілька прикладів задач на використання графів. Поясніть якими методами їх можна вирішити. Як зазвичай підходять до визначення швидкості роботи з графами.
5. Опишіть скорочену технологію роботи з графом у задачі.
6. На прикладі задачі про лабіринт поясніть правила роботи хвильового алгоритму.
7. Чи доречно використовувати у реалізації хвильового алгоритму рекурсію?

ВИКОРИСТАНА ЛІТЕРАТУРА

1. Басюк Т. М., Думанський Н. О., Пасічник О. В. Основи інформаційних технологій : навч. посіб. Львів : «Новий Світ-2000», 2024. 390 с.
2. Кормен Томас Г. Алгоритми доступно. Київ : К.І.С., 2021. 194 с.
3. Ришковець Ю. В., Висоцька В. А. Алгоритмізація та програмування. у 2-х Ч. : навч. посіб. Львів : «Новий Світ-2000», 2025. Ч. 1. 336 с.
4. Ришковець Ю. В., Висоцька В. А. Алгоритмізація та програмування. у 2-х Ч. : навч. посіб. Львів : «Новий Світ-2000», 2025. Ч. 2. 315 с.
5. Угрин Д. І., Галочкін О. В., Яцько О. М. Структури даних та алгоритми : навч. посіб. Чернівці : Чернівецький національний університет ім. Ю. Федьковича, 2022. 324 с.
6. Шаховська Н. Б., Голощук Р. О. Алгоритми і структури даних : навч. посіб. Львів : «Магнолія 2006», 2024. 215 с.
7. Knuth D. E. Art of Computer Programming. Pearson Education, Limited, 2019. 640 p.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Бармак О. В., Манзюк Е. А., Радюк П. М. Теорія алгоритмів. Теоретичний курс та лабораторний практикум : навч. посіб. Хмельницький : ХНУ, 2023. 168 с.
2. Клевцовський А. В., Креневич А. П. Методичні вказівки до лабораторних занять із дисципліни «Алгоритми і структури даних» для студентів механіко-математичного факультету. Київ : ВПЦ «Київський Університет», 2024. 70 с.
3. Коротєєва Т. О. Алгоритми і структури даних : навч. посіб. Львів : Видавництво Львівської політехніки, 2014. 280 с.
4. Креневич А. П. Алгоритми і структури даних. Київ : ВПЦ «Київський Університет», 2021. 200 с
5. Кублій Л. І. Алгоритми і структури даних. Основи алгоритмізації. Київ : КПІ ім. Ігоря Сікорського, 2022. 528 с.
6. Knuth D. E. Art of Computer Programming. Pearson Education, Limited, 2019. 640 p.

Інформаційні ресурси

1. Coursera. URL: https://www.coursera.org/.
2. Learn C++. URL: https://www.learncpp.com/.
3. NumPy Documentation. *NumPy.* URL: https://numpy.org/doc/.
4. Prometheus. URL: https://courses.prometheus.org.ua/.

ЗМІСТ

Вступ 3
Лабораторна робота 1. Алгоритми та програми. Показники обчислювальної складності алгоритмів 5
Лабораторна робота 2. Правила застосування декомпозиції та використання функцій 17
Лабораторна робота 3. Масиви: зберігання, обробка, сортування, пошук 22
Лабораторна робота 4. Структури даних (динамічні). Особливості будови динамічних структур (список однозв'язний та двозв'язний, стек, черга). Робота зі списками та стеками 30
Лабораторна робота 5. Подання графів. Створення та взаємоперетворення матриць суміжності та інцидентності. Використання списку інцидентності. Аналіз ефективності використання перелічених способів подання графів 41
Лабораторна робота 6. Розробка програм з реалізацією спеціальних алгоритмів обробки графів 45
Використана література 52
Рекомендована література 52

Навчально-методичне видання
(*українською мовою*)

Матвіїшина Надія Вікторівна
Циммерман Геннадій Анатолійович
Шило Галина Миколаївна

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ
Методичні рекомендації до лабораторних занять
для здобувачів ступеня вищої освіти бакалавра
спеціальності «Комп'ютерні науки»
освітньо-професійної програми «Комп'ютерні науки»

Рецензент *С. І. Гоменюк*
Відповідальний за випуск *О. С. Пшенична*
Коректор *Г. А. Циммерман*