

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ КОМП'ЮТЕРНИЙ ПРАКТИКУМ

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня магістра
за освітньою програмою «Технічні та програмні засоби автоматизації»
спеціальності 151 Автоматизація та комп'ютерно-інтегровані технології

Укладачі: Л. М. Бугаєва, Д. О. Ковалюк

Електронне мережне навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2022

Рецензент

Гулієнко С. В., к.т.н., доцент, доцент кафедри машин та апаратів хімічних та нафтопереробних виробництв КПІ ім. Ігоря Сікорського

Відповідальний
редактор

Цапар В. С., к.т.н., доцент

*Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 3 від 01.12.2022 р.)
за поданням Вченої ради інженерно-хімічного факультету
(протокол № 10 від 27.10.2022 р.)*

Навчальний посібник є частиною навчально-методичного забезпечення дисципліни та розроблений у відповідності до її силабусу. Посібник призначений для підготовки та виконання робіт комп'ютерного практикуму. Завдання передбачають комп'ютерну реалізацію алгоритмів та структури даних, які є фундаментом сучасної методології розробки програм. Розглядаються основні поняття, які визначають суть алгоритмізації процесів обробки інформації, можливі варіанти реалізації алгоритмів та структур даних. Приклади програмної реалізації алгоритмів наведені на мовах C/C++, Python, JavaScript. Для магістрів, які навчаються за освітньою програмою «Технічні та програмні засоби автоматизації».

Реєстр. № НП 22/23-239. Обсяг 8,5 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Перемоги, 37, м. Київ, 03056

<https://kpi.ua>

Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

Зміст

Вступ.....	4
Практична робота 1	
Використання базових алгоритмів	5
Практична робота 2	
Використання базових алгоритмів. Рекурсивні процедури	10
Практична робота 3	
Реалізація алгоритмів сортування числових даних	14
Практична робота 4	
Дослідження лінійних структур даних: стек та черга. Реалізація стеку та черги на базі вбудованого списку.....	17
Практична робота 5	
Дослідження структур даних: зв'язний список	22
Практична робота 6	
Реалізація алгоритму бінарного пошуку	28
Літературні джерела.....	33

Вступ

Поняття алгоритму – це одне з головних понять математики, обчислювальної математики, техніки, оскільки знаходження прийнятного алгоритму для рішення різних класів задач є одним із завдань цих наук.

Успіхи сучасної обчислювальної математики в значній мірі обумовлені принципом поділу складних обчислювальних процесів на елементарні операції. Це робиться за допомогою послідовних дій, які задаються алгоритмом. Оскільки складність задач обчислювального характеру постійно зростає, актуальною також є задача розробки ефективних алгоритмів обчислювального процесу. В залежності від характеру задач, що вирішуються, використовуються дані різних типів. Це істотно впливає на структуру алгоритмів.

В дисципліні розглядаються алгоритми та структури даних, які є фундаментом сучасної методології розробки програм. В ній вивчаються основні поняття, які визначають суть алгоритмізації процесів обробки інформації, засоби представлення алгоритмів, викладаються можливі варіанти реалізації алгоритмів та структур даних. В посібнику наведені приклади C/C++, Python. Студенти можуть реалізовувати алгоритми на будь-якій мові програмування.

ПРАКТИЧНА РОБОТА №1

ВИКОРИСТАННЯ БАЗОВИХ АЛГОРИТМІВ

Мета роботи: повторення та систематизація основних алгоритмічних структур та їх реалізація на мовах програмування за вибором студента.

Завдання

Показати на реальному прикладі (бажано на задачах автоматизації) роботу:

1. Алгоритмічної структури слідування.
2. Алгоритмічної структури розгалуження (повні і неповні).
3. Алгоритмічної структури повторення (цикл):
 - Цикл із лічильником
 - Цикл із передумовою
 - Цикл із післяумовою

Всі приклади завдань оформити окремими програмними модулями із коментарями. На початку кожного програмного коду має бути коментар, яку задачу даний модуль виконує.

Звіт у форматі документа Word має містити тему, завдання з описом обраного середовища і конкретики виконуваної процедури. Всі рисунки (скріни) з результатами виконання мають супроводжуватися підписами та відповідними коментарями. До роботи прикріплюється додаток з лістингом програми, яка забезпечена коментарями українською мовою.

Теоретичні відомості

Поняття про базові алгоритмічні конструкції

Базові алгоритмічні конструкції або базові алгоритмічні структури – це основні структурні елементи, за допомогою яких створюють алгоритм для розв’язування деякої загальної задачі.

Існують три основні (базові) алгоритмічні структури: слідування (або лінійна структура), розгалуження та повторення (або цикл), komponуючи які можна представити логічну структуру будь-якого алгоритму.

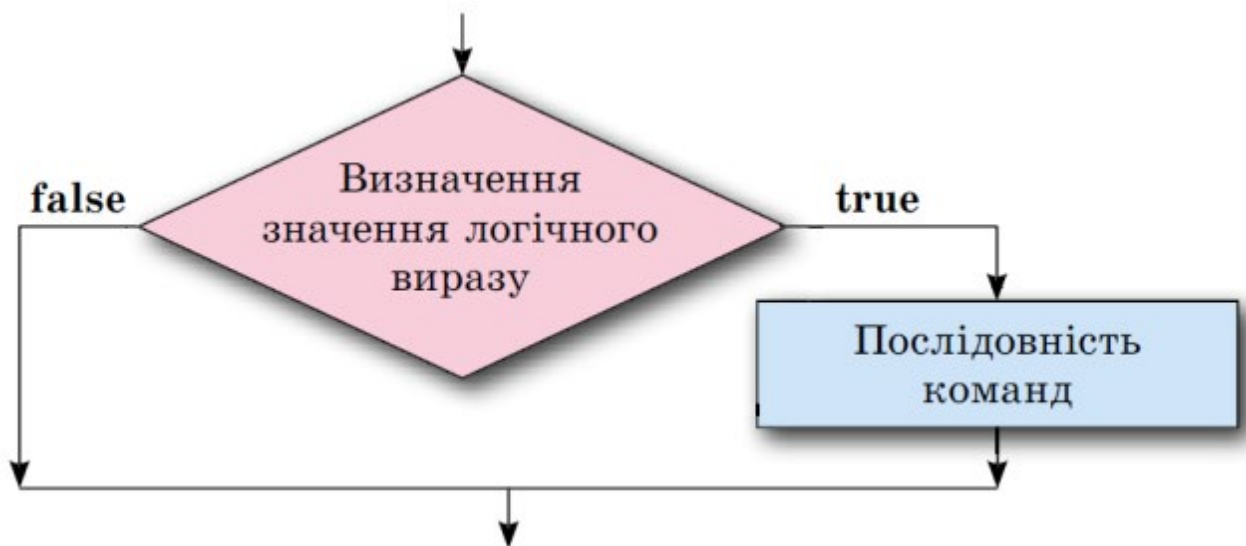
Основною особливістю базових алгоритмічних структур є їх повнота. Це означає, що цих структур достатньо для створення алгоритмів довільної складності та наявність в них одного входу та одного виходу.

Структура слідування або лінійна алгоритмічна структура – це алгоритмічна структура, яка забезпечує отримання результату шляхом одноразового виконання дії, незалежно від вхідних даних та проміжних результатів. Дії у такій структурі виконуються послідовно, одна за одною, або лінійно.

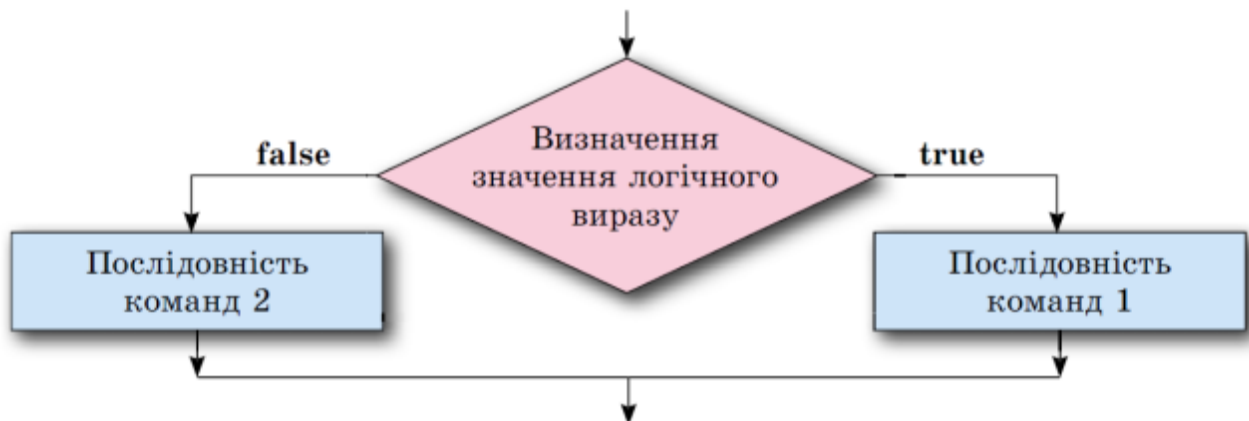
Структура розгалуження (структура вибору або умова) – вибір дії або групи дій (блоку) у разі виконання або невиконання заданої умови. Ця алгоритмічна структура залежно від результату перевірки умови (так чи ні, істина або хибно) забезпечує виконання одного з альтернативних шляхів роботи алгоритму. Кожен із них веде до загального виходу, тому робота алгоритму триватиме незалежно від того, який із варіантів було обрано.

Розгалуження поділяються на два типи: повні і неповні.

Неповне розгалуження – це розгалуження, в якому хід алгоритму визначено лише в разі виконання умови. Якщо умова не виконується, то робота алгоритму визначається алгоритмічною конструкцією, яка розміщена після розгалуження.



Повне розгалуження – це розгалуження, в якому певні дії визначено у двох випадках: виконання та невиконання умови.



Поширеним випадком є розміщення однієї структури вибору в іншій. Такі структури називають вкладеними розгалуженнями.

На алгоритмічній мові структура повного розгалуження виглядає наступним чином:

вибір

за умови 1: дії 1

за умови 2: дії 2

.....

за умови N: дії N

У тому випадку, якщо вхідні дані не задовольняють жодну із умов, яка запропонована у структурі вибору, керування ходом алгоритму передається до команди, що розміщена після вибору.

Структура повторення (структура циклу або цикл) - це алгоритмічна структура, в якій передбачено повторення деякої вказівки або блоку вказівок. За допомогою цієї структури описують однотипні дії, що повторюються багатократно. Такі алгоритми забезпечують виконання значної послідовності дій, записаних порівняно короткою послідовністю команд.

Повторення забезпечує багаторазове виконання деякої сукупності команд, яку називають тілом циклу. Існують три види циклів.

Цикл із лічильником (цикл із параметром) – це алгоритмічна конструкція, в якій кількість повторень задано заздалегідь, керується лічильником (ітератором) і логічними виразами, розташованими на його початку. Це єдиний вид циклу, для якого кількість повторень є наперед заданою.

На алгоритмічній мові структура циклу із параметром виглядає наступним чином:

початок циклу для i від i1 до i2

тіло циклу

Цикл із передумовою – це алгоритмічна конструкція, що забезпечує виконання команди або групи команд доти, поки виконується (є істинною) логічна умова, сформульована на його початку. Для циклу із передумовою кількість повторень не є наперед заданою. Якщо вхідні дані забезпечують хибність логічної умови, яка задана на початку циклу, то його тіло може не виконатись жодного разу. У такому випадку керування ходом алгоритму переходить до команди (вказівки), яка розміщена після циклу.

Цикл із післяумовою – це алгоритмічна конструкція, що забезпечує виконання команди або групи команд доти, поки виконується (є істинною) логічна умова, сформульована у його кінці. Для циклу із післяумовою кількість повторень не є наперед заданою. Якщо вхідні дані забезпечують хибність

логічної умови, яка задана в кінці циклу, то його тіло виконається один раз. Після цього керування ходом алгоритму переходить до команди (вказівки), яка розміщена після циклу. Тобто якщо тіло циклу з передумовою може не виконатись жодного разу, то тіло циклу з післяумовою обов'язково виконається хоча б один раз.

Контрольні питання

1. Дайте означення базової алгоритмічної конструкції.
2. Що таке структура слідування?
3. Які алгоритми називають лінійними?
4. Чи всі алгоритми, які містять структуру слідування лінійні? Обґрунтуйте.
5. Дайте означення структури розгалуження.
6. Які бувають види розгалужень?
7. Що таке вкладене розгалуження?
8. Дайте означення структури вибору.
9. У чому полягає різниця між структурами “вибір” та “вибір-інакше”?
10. Дайте означення циклу. Назвіть види циклів.
11. Для яких видів циклів наперед відомою є кількість повторень?
12. Для яких видів циклів тіло циклу може не виконатись жодного разу?
13. Для яких видів циклів тіло обов'язково виконається хоча б один раз?

ПРАКТИЧНА РОБОТА №2

ВИКОРИСТАННЯ БАЗОВИХ АЛГОРИТМІВ.

РЕКУРСИВНІ ПРОЦЕДУРИ

Мета роботи: навчитись розробляти рекурсивні процедури та реалізовувати їх на алгоритмічних мовах за вибором студента.

Після виконання роботи студент повинен:

- вміти розробляти алгоритми рекурсії;
- вміти обирати оптимальний тип для представлення вхідних та вихідних даних;
- вміти реалізовувати розроблені алгоритми сортування на мові програмування;
- вміти оформляти алгоритми рекурсії у вигляді функцій;
- вміти виконувати тестування програми;
- вміти пояснювати хід виконання алгоритму та принцип роботи програми.

Теоретичні відомості

У програмуванні рекурсія — виклик підпрограми (функції чи процедури) з неї самої (зазвичай з іншими значеннями вхідних параметрів) безпосередньо чи через інші функції. Кількість вкладених викликів функції чи процедури називається глибиною рекурсії.

Типовими прикладами рекурентних співвідношень є:

- Послідовність Фібоначчі:

$$a_{n+1}=a_n+a_{n-1}, a_1=1, a_2=1$$

- Рекурентне співвідношення арифметичної прогресії:

$$a_{n+1}=a_n+d$$

- Рекурентне співвідношення геометричної прогресії:

$$a_{n+1}=a_n \cdot q$$

- Рекурентне співвідношення послідовності $n!$

$$a_{n+1}=a_n \cdot (n+1)$$

Приклад

Класичним прикладом рекурсії може слугувати визначення факторіалу в такому вигляді:

$$n! = 1 \quad n = 0, \quad n! = n(n - 1)!, \quad \text{де } n - \text{натуральне число.}$$

Цей приклад носить ілюстративний характер через свою зручність для пояснення поняття рекурсії.

```
number = input("Введіть число: ")
n = int(number)
factorial = 1
if n >= 1:
    for i in range (1, n + 1):
        factorial = factorial * i

print("факторіал ", n , " = ", factorial)

факторіал 4 = 24
```

Приклад використання рекурсивної процедури для організації обчислення $n!$:

```
def factorial(n):
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)

number = input("Введіть число: ")
n = int(number)
print("факторіал ", n , " = ", factorial(n))

факторіал 5 = 120
```

Важливим моментом є той факт, що в наведеному прикладі рекурсивний виклик стоїть всередині оператора умови. Це є необхідною умовою того, щоби рекурсивний процес рано чи пізно завершився (рекурсія не продовжувалася до нескінченості). Важливим також є той факт, що функція викликає себе з іншим значенням параметра: не тим, із яким сама вона була викликана.

Завдання для індивідуального виконання

1. Дано натуральне число N . Перевірити, чи є дане число степенем 3. Для розв'язування даної задачі використати рекурсивну функцію.
2. Дано натуральне число N . Підрахувати суму його цифр. Для розв'язування даної задачі використати рекурсивну функцію.
3. Дано натуральне число $N > 1$. Перевірити, чи воно є простим.
4. Розкласти число на множники. Множники вивести на екран у порядку не спадання з врахуванням кратності.
5. Побудувати рекурсивний алгоритм піднесення числа x до степеня y .
6. Побудувати рекурсивний алгоритм множення числа x на число y .
7. Побудувати числову послідовність, в якій число k зустрічається рівно k разів. Наприклад: 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, ...
8. Реалізувати рекурсивну функцію знаходження площі прямокутника.
9. Побудувати рекурсивний алгоритм для обчислення квадрату цілого числа, якщо відома залежність: $n^2 = (n-1)^2 + 2 \cdot (n-1) + 1$.
10. Побудувати рекурсивний алгоритм для обчислити кількість комбінацій з n різних елементів по m .

Контрольні питання

1. Який процес обробки даних називають ітеративним?
2. Який процес обробки даних називають рекурсивним?
3. Дайте означення рекурсії.
4. Яка функція називається прямо рекурсивною?
5. Яка функція називається непрямо рекурсивною?
6. Якщо рекурсивна функція викликається із параметром p , скільки ще викликів цієї функції буде здійснено у програмі?
7. Який із способів обробки даних є більш вигідним з точки зору програмної реалізації: рекурсивний чи ітераційний?
8. Що таке глибина рекурсії? Якщо рекурсивна функція викликається із параметром p , яка у такому випадку глибина рекурсії?
9. Для чого призначена умова виходу з рекурсії?

ПРАКТИЧНА РОБОТА №3

РЕАЛІЗАЦІЯ АЛГОРИТМІВ СОРТУВАННЯ ЧИСЛОВИХ ДАНИХ

Мета роботи: Ознайомитись із основними алгоритмами сортування даних, вміти використовувати алгоритми сортування для впорядкування стовпців та рядків у двовимірному масиві.

Завдання: Реалізувати алгоритм сортування за власним вибором.

Теоретичні відомості

Кількість алгоритмів сортування достатньо велика. В даний час ця кількість разом з екзотичними та мало поширеними алгоритмами наближається до 40. Всі ці алгоритми можна класифікувати за кількома ознаками. Наведемо найпоширеніші.

Самий простий алгоритм групи обмінних сортувань (або сортувань перестановками) отримав назву *«бульбашкового» сортування*. Цей алгоритм вважається найпростішим з універсальних, одночасно будучи одним з найменш ефективних (тобто повільних).

Крім «бульбашкового» сортування до групи обмінних відносяться також швидке сортування і сортування *«гребінцем»*.

Сортування відбором (інша назва сортування вибором) працює наступним чином. У структурі даних (масиві або списку) знаходиться найменший (або найбільший) елемент і міняється місцями з першим (або останнім, залежно від напрямку упорядкованості). Тут поєднуються два процеси – пошук та обмін. Потім з решти $N - 1$ елементів знаходиться найменший і змінюється місцем з другим. Процес повторюється до тих пір, поки нічого немає переставляти.

3 Приклад реалізації на мові програмування

Сортування «бульбашками»:

```
def bubble_sort(array):
    n = len(array)
    swapped = False
    for i in range(n-1):
        for j in range(0, n-i-1):
            if array[j] > array[j + 1]:
                swapped = True
                array[j], array[j + 1] = array[j + 1], array[j]

        if not swapped:
            return
```

```
data = [64, 34, 25, 12, 22, 11, 90]
```

```
bubble_sort(data)
```

```
print("відсортований масив:")
```

```
for number in data:
    print(number, end=" ")
```

відсортований масив: 11 12 22 25 34 64 90

Алгоритм сортування вибором:

```
def selection_sort(array):
    size = len(data)
    for step in range(size):
        min_idx = step

        for i in range(step + 1, size):
            if array[i] < array[min_idx]:
                min_idx = i

        (array[step], array[min_idx]) = (array[min_idx], array[step])
```

```
data = [64, 34, 25, 12, 22, 11, 90]
```

```
selection_sort(data)
```

```
print("відсортований масив:")
```

```
for number in data:
    print(number, end=" ")
```

відсортований масив: 11 12 22 25 34 64 90

Завдання для індивідуального виконання

Задано двовимірний масив випадкових цілих чисел $A[m, n]$.

1. Відсортувати елементи непарних рядків масиву.
2. Відсортувати елементи парних рядків масиву.
3. Відсортувати елементи непарних стовпців масиву.
4. Відсортувати елементи парних стовпців масиву.
5. Відсортувати елементи головної діагоналі масиву.
6. Відсортувати елементи бічної діагоналі.
7. Відсортувати елементи кожного стовпця масиву.
8. Відсортувати елементи кожного стовпця масиву.
9. Відсортувати елементи кожного рядка масиву.
10. Відсортувати елементи кожного рядка масиву.

Звіт має містити тему, завдання з описом обраного середовища і конкретики виконуваної процедури. Всі рисунки (скріни) з результатами виконання мають супроводжуватися підписами та відповідними коментарями. До роботи прикріплюється доданок з лістингом програми, яка забезпечена коментарями українською мовою. Вимагаються загальні висновки роботи.

Контрольні питання

1. Що таке "Сортування"?
2. Скільки існує груп алгоритмів сортування?
3. Скільки існує алгоритмів сортування?
4. Що потрібно враховувати при виборі алгоритму сортування?
5. Який алгоритм сортування вважається найбільш ефективним?
6. Що означає поняття «швидкість сортування»?
7. У чому полягає метод бульбашкового сортування?
8. У чому полягає метод сортування відбором?

ПРАКТИЧНА РОБОТА №4

ДОСЛІДЖЕННЯ ЛІНІЙНИХ СТРУКТУР ДАНИХ: СТЕК ТА ЧЕРГА РЕАЛІЗАЦІЯ СТЕКУ ТА ЧЕРГИ НА БАЗІ ВБУДОВАНОГО СПИСКУ

Мета роботи: навчитися реалізовувати структури даних: стек і чергу

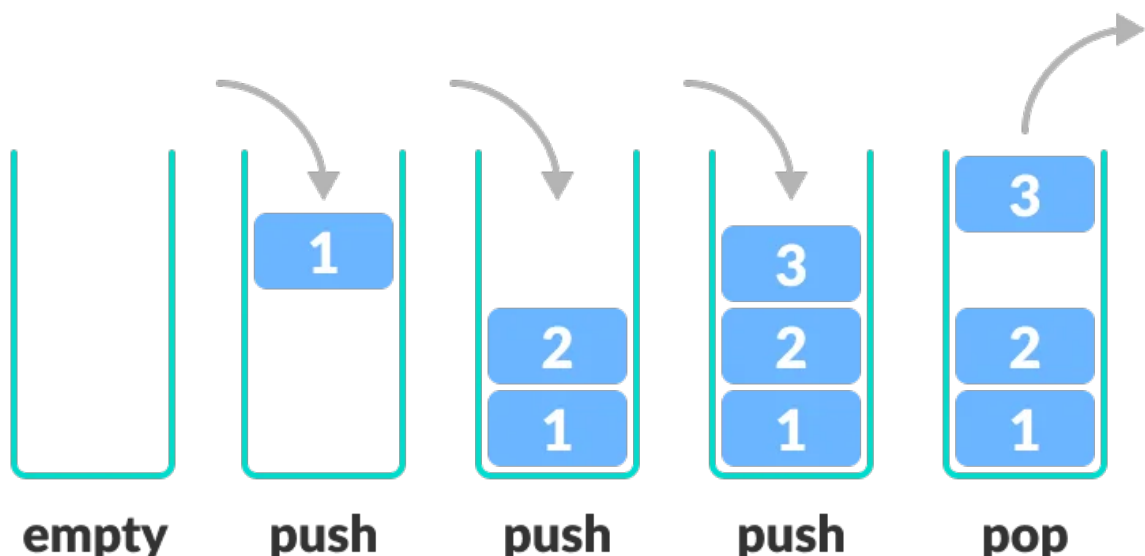
Теоретичні відомості

Лінійні структури даних

Стек – це динамічна структура даних із принципом доступу до елементів "останнім прийшов – першим пішов" (англ. LIFO – last in, first out). Усі операції у стеку проводяться тільки з одним елементом, що називається **верхівкою стека**, та був доданий до стеку останнім.

Для стеку визначають **чотири базових операцій**:

1. Створення порожнього стеку.
2. Операція **empty()** – визначення, чи є стек порожнім. Повертає булеве значення.
3. Операція **push(item)** – вштовхнути (додати) елемент *item* до стека. Після цього стек змінюється. Елемент *item* стає верхівкою стеку.
4. Операція **pop()** – взяти верхівку стека. Ця операція повертає верхівку стеку. Стек при цьому змінюється.



Крім зазначених вище операцій, опціонально визначають інші операції, наприклад, визначають операцію **top()**, що повертає верхівку стеку, без видалення верхівки зі стеку. Також корисною є операція визначення довжини (тобто кількості елементів) у стеку.

У нижченаведеній таблиці наведено приклад роботи операцій зі стеком. У колонці «Вміст стеку після здійснення операції» напівжирним шрифтом виділено елемент, що є верхівкою стека.

Операція над стеком	Вміст стеку після здійснення операції	Значення, що повертається в результаті здійснення операції
<code>stack = Stack()</code>	<code>[]</code>	
<code>stack.empty()</code>	<code>[]</code>	True
<code>stack.push(32)</code>	<code>[32]</code>	
<code>stack.push(17)</code>	<code>[17, 32]</code>	
<code>stack.push(26)</code>	<code>[26, 17, 32]</code>	
<code>len(stack)</code>	<code>[26, 17, 32]</code>	3
<code>stack.empty()</code>	<code>[26, 17, 32]</code>	False
<code>stack.pop()</code>	<code>[17, 32]</code>	26
<code>stack.top()</code>	<code>[17, 32]</code>	17

Приклад реалізації стеку на базі вбудованого списку Python

Найпростішу реалізацію стеку у Python можна здійснити на базі вбудованого списку (**list**). У такій реалізації елементи стеку розміщують у списку, причому вважається, що верхівка стеку знаходиться в кінці списку. Тоді для додавання та віднімання елементів використовують методи списку **append()** та **pop()** відповідно.

```
# створюємо порожній стек
def create():
    stack = []
    return stack

# перевірка чи стек порожній
def is_empty(stack):
    return len(stack) == 0

# додаємо елемент в стек (в кінець списку)
def push(stack, element):
    stack.append(element)
```

```

# видалення елемента зі стеку
def pop(stack):
    if (is_empty(stack)):
        return "стек порожній"
    return stack.pop()

# друкуємо елементи зі стеку
def show(stack):
    for element in stack:
        print(element)

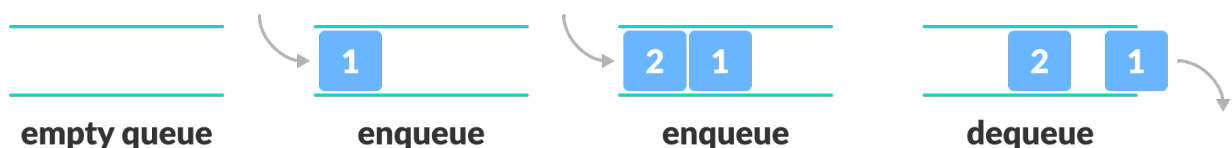
students = create()
push(students, 'Іванов Іван')
push(students, 'Петров Петро')
push(students, 'Сідоров Сергій')
push(students, 'Яценко Оксана')
print("Дістаємо зі стеку та працюємо з: " + pop(students))
show(students)

```

Дістаємо зі стеку та працюємо з: Яценко Оксана
Іванов Іван
Петров Петро
Сідоров Сергій

Черга – динамічна структура даних із принципом доступу до елементів "першим прийшов – першим пішов" (англ. FIFO – first in, first out). Класична реалізація черги вимагає реалізувати структуру даних з такими операціями:

1. Створення порожньої черги.
2. Операція **empty()** – визначення, чи є черга порожньою.
3. Операція **enqueue(item)** – додати елемент *item* до кінця черги.
4. Операція **dequeue()** – взяти елемент з початку черги.



Як і у випадку зі стеком, крім зазначених вище операцій, опціонально визначають інші операції: перегляд елемента в голові та хвості черги без їхнього видалення з черги, розмір черги, тобто кількість елементів у ній, тощо.

У нижче наведеній таблиці наведено приклад роботи операцій із чергою. У колонці «Вміст черги після здійснення операції» напівжирним шрифтом виділено елемент, що є головою черги, а підкресленням – елемент, що знаходиться у її хвості.

Операція над чергою	Вміст черги після здійснення операції	Значення, що повертається в результаті здійснення операції
<code>queue = Queue()</code>	<code>[]</code>	
<code>queue.empty()</code>	<code>[]</code>	True
<code>queue.enqueue(32)</code>	<code>[<u>32</u>]</code>	
<code>queue.enqueue(17)</code>	<code>[32, <u>17</u>]</code>	
<code>queue.enqueue(99)</code>	<code>[32, 17, <u>99</u>]</code>	
<code>queue.enqueue(26)</code>	<code>[32, 17, 99, <u>26</u>]</code>	
<code>queue.enqueue("New")</code>	<code>[32, 17, 99, 26, <u>"New"</u>]</code>	
<code>len(queue)</code>	<code>[32, 17, 99, 26, <u>"New"</u>]</code>	5
<code>queue.empty()</code>	<code>[32, 17, 99, 26, <u>"New"</u>]</code>	False
<code>queue.dequeue()</code>	<code>[17, 99, 26, <u>"New"</u>]</code>	32

Реалізація черги на базі вбудованого списку

Як і у випадку зі стеком, спочатку наведемо простішу реалізацію черги, що базується на вбудованому списку. Опишемо клас `Queue` у якому реалізуємо основні операції, а також функцію визначення розміру черги.

```
class Queue:
```

```
    def __init__(self):
        self.queue = []

    # додаємо елемент
    def enqueue(self, element):
        self.queue.append(element)

    # видалення елементу
    def dequeue(self):
        if len(self.queue) < 1:
            return None
        return self.queue.pop(0)
```

```

def show(self):
    for element in self.queue:
        print(element)

def size(self):
    return len(self.queue)

runway = Queue()
runway.enqueue(['SP23', 'Kyiv', 'Varshava'])
runway.enqueue(['SP23', 'Kyiv', 'Rome'])
runway.enqueue(['QA11', 'Kyiv', 'Qatar'])
runway.enqueue(['RA54', 'Kyiv', 'Barselona'])

print("В черзі на виліт")
runway.show()

print("Літак взлітає")
runway.dequeue()

print("Літак взлітає")
runway.dequeue()

print("В черзі на виліт")
runway.show()

```

```

В черзі на виліт
['SP23', 'Kyiv', 'Varshava']
['SP23', 'Kyiv', 'Rome']
['QA11', 'Kyiv', 'Qatar']
['RA54', 'Kyiv', 'Barselona']

```

```

Літак взлітає
Літак взлітає

```

```

В черзі на виліт
['QA11', 'Kyiv', 'Qatar']
['RA54', 'Kyiv', 'Barselona']

```

Звіт до роботи повинен містити розроблений алгоритм, виконуваний код програми. Здобувач повинен пояснювати хід виконання алгоритму, програми відповідати на питання щодо роботи алгоритму, програми, теми практичної роботи.

ПРАКТИЧНА РОБОТА №5

ДОСЛІДЖЕННЯ СТРУКТУР ДАНИХ: ЗВ'ЯЗНИЙ СПИСОК

Мета роботи: навчитися реалізовувати однозв'язний та двозв'язний список.

Теоретичні відомості

Означення. Класичний (зв'язний або однозв'язний) список – це динамічна структура даних, що складається з елементів (як правило одного типу), пов'язаних між собою у строго визначеному порядку: кожен елемент списку вказує на наступний елемент списку

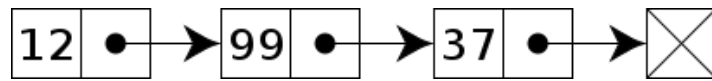
Зв'язаний список в програмуванні — одна з найважливіших структур даних, в якій елементи лінійно впорядковані, але порядок визначається не номерами елементів, а вказівниками, які входять в склад елементів списку та вказують на наступний за даним елемент (в однозв'язаних або одnobічно зв'язаних списках) або на наступний та попередній елементи (в двозв'язаних або двобічно зв'язаних списках).

Список має «голову» - перший елемент та «хвіст» - останній елемент. Зв'язані списки мають серію переваг порівняно з масивами. Зокрема, в них набагато ефективніше незалежно від кількості елементів виконуються процедури додавання та вилучення елементів. Натомість, масиви набагато кращі в операціях, які потребують безпосереднього доступу до кожного елементу, що у випадку зі зв'язаними списками неможливо та потребує послідовного перебору усіх елементів, які передують даному.

Види зв'язаних списків

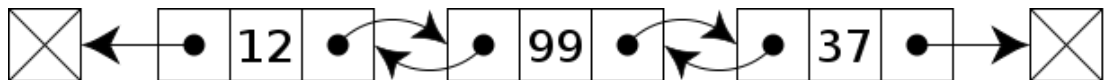
- Одnobічно зв'язані списки. В одnobічно зв'язаному списку, який є найпростішим різновидом зв'язаних списків, кожен елемент складається з двох полів: **data** або даних, та вказівника **next** на

наступний елемент. Якщо вказівник не вказує на інший елемент (інакше: $next = NULL$), то вважається, що даний елемент — останній в списку.



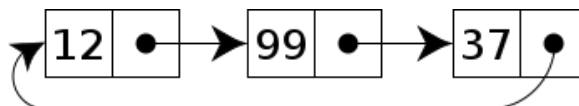
Однобічно зв'язаний список з трьох елементів

- Двобічно зв'язаний список. В двобічно зв'язаному списку елемент складається з трьох полів — вказівника на попередній елемент **prev**, поля даних **data** та вказівника **next** на наступний елемент. Якщо **prev=NULL**, то в елемента немає попередника (тобто він є «головою» списку), якщо **next=NULL**, то в нього немає наступника («хвіст» списку).



Двобічно зв'язаний список

- Кільцевий список. В кільцевому списку перший та останній елемент зв'язані. Тобто, поле **prev** голови списку вказує на хвіст списку, а поле **next** хвоста списку вказує на голову списку.



Кільцевий однобічно зв'язаний список

Застосування списків

Списки застосовуються в програмуванні як самостійні структури. Також на їх основі можуть будуватись складніші структури даних, такі як дерева. На базі списків також можуть бути реалізовані стеки та черги.

Переваги та недоліки

В загальному випадку, якщо необхідно оперувати з динамічними множинами, де присутні інтенсивні операції з додавання або видалення елементів, існують досить переконливі аргументи для використання саме зв'язаних списків.

Завдання: Створення та реалізація зв'язаного списку за допомогою мов програмування.

```
# клас вузол
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

# Linked list
class LinkedList:

    def __init__(self):
        self.head = None

    # вставка на початок
    def insert_to_beginning(self, data):
        node = Node(data)
        node.next = self.head
        self.head = node

    # вставка після вузла
    def insert_after(self, prev_node, data):
        new_node = Node(data)
        new_node.next = prev_node.next
        prev_node.next = new_node

    # вставка в кінець
    def insert_to_end(self, data):
        node = Node(data)

        if self.head is None:
            self.head = node
            return

        last = self.head
        while (last.next):
            last = last.next

        last.next = node
```

```

# пошук елементу
def search(self, key):
    current = self.head
    while current is not None:
        if current.data == key:
            return True
        current = current.next
    return False

# надрукувати елементи
def show(self):
    temp = self.head
    while (temp):
        print(str(temp.data), end=" ")
        temp = temp.next

numbers = LinkedList()
numbers.insert_to_end(1)
numbers.insert_to_beginning(2)
numbers.insert_to_beginning(3)
numbers.insert_to_end(4)
numbers.insert_after(numbers.head, 'нове значення')

print('елементи списку:')
numbers.show()

print()

element_for_search = 12
result = "елемент знайдено" if numbers.search(element_for_search)
else "елемент не знайдено"
print(result, str(element_for_search))

елементи списку: 3 нове значення 2 1 4
елемент не знайдено 12

```

Реалізація двозв'язного списку:

Для реалізації двозв'язного списку треба виконати наступні кроки:

1. Створити список.
2. Визначити, чи порожній список. Повертає булеве значення.
3. Зробити поточними перший елемент списку.
4. Зробити поточними останній елемент списку.
5. Перейти до наступного елемента.
6. Перейти до попереднього елемента.
7. Отримати поточний елемент.
8. Вставити новий елемент перед поточним.
9. Вставити новий елемент після поточного.
10. Видалити поточний елемент.

На основі двозв'язного списку може бути реалізована така структура даних, як черга з пріоритетом. У багатьох мовах програмування ця структура уже є вбудованою.

```
from queue import PriorityQueue
```

```
students = PriorityQueue()
```

```
# insert into queue
```

```
students.put((2, 'Dima'))
```

```
students.put((3, 'Oksana'))
```

```
students.put((2, 'Alla'))
```

```
students.put((5, 'Vika'))
```

```
students.put((1, 'Anton'))
```

```
print(students.get())
```

```
print(students.get())
```

```
# check size
```

```
print('Items in queue :', students.qsize())
```

```
# check is empty
```

```
print('Is queue empty :', students.empty())
```

```
(1, 'Anton')  
(2, 'Alla')  
Items in queue : 3  
Is queue empty : False
```

Реалізацію та тестування проводити студент може в програмному середовищі за власним вибором. Звіт має містити тему, завдання з описом обраного середовища і конкретики виконуваної процедури. Всі рисунки (скріни) з результатами виконання мають супроводжуватися підписами та відповідними коментарями. До роботи прикріплюється додаток з лістингом програми, яка забезпечена коментарями українською мовою. Вимагаються загальні висновки.

Контрольні питання

1. Що представляють собою зв'язні списки?
2. До яких структур даних відносяться списки?
3. Які існують різновиди зв'язних списків?
4. У чим відмінність зв'язного списку від масиву?
5. Які недоліки однозв'язного списку?
6. У чому відмінність однозв'язного списку від двозв'язного?
7. Які операції застосовуються для зв'язних списків?
8. У чому відмінність зчитування інформації з списку від черги або стеку?

ПРАКТИЧНА РОБОТА №6.

РОЗРОБКА АЛГОРИТМУ БІНАРНОГО ПОШУКУ

Мета і зміст роботи: Знати алгоритм бінарного пошуку, вміти реалізовувати та застосовувати алгоритм бінарного пошуку для знаходження елементів у рядках та стовпцях матриці.

Теоретичні відомості

Метод декомпозиції (інша назва – метод «розділяй і владарюй»), імовірно, найбільш популярний метод розробки алгоритмів. Ряд дуже ефективних алгоритмів являють собою реалізації цієї загальної стратегії.

Алгоритми, засновані на методі декомпозиції, працюють у відповідності з наступним планом.

1. Екземпляр задачі розбивається на кілька менших екземплярів тої ж задачі, в ідеалі – однакового розміру.
2. Вирішуються менші екземпляри задачі (зазвичай рекурсивно, хоча іноді для невеликих екземплярів застосовується який-небудь інший алгоритм).
3. При необхідності розв'язок вихідної задачі отримується шляхом комбінації розв'язків менших екземплярів.

Бінарний пошук. Алгоритм пошуку, який базується на стратегії методу декомпозиції. Особливість алгоритму в тому, що він працює на відсортованій послідовності ключів.

Якщо говорити про пошук елементів у впорядкованому масиві, то бінарний пошук полягає у порівнянні серединного елемента масиву з шуканим значенням, і повторенням алгоритму для тієї або іншої половини, залежно від результату порівняння.

Завдання до практичної роботи

Для виконання поставленої мети, необхідно вирішити наступні задачі:

1. Реалізувати алгоритм В.
2. Провести математичний аналіз алгоритму В.
3. Провести емпіричний аналіз алгоритму В.
4. Для тих самих вхідних даних виконати пошук алгоритмом В и алгоритмом Q (швидкий послідовний пошук). Провести емпіричний порівняльний аналіз двох алгоритмів.
5. Передбачити можливість виводу елемента випадкової послідовності по його позиції для перевірки результатів пошуку.
6. Реалізувати інтерфейс користувача. Функціональні можливості інтерфейсу:
 - Виводити запрошення на введення об'єму випадкової послідовності й аргументу пошуку.
 - Виводити позицію шуканого елемента в списку.

Приклад виконання завдання

Задано матрицю $A[n, n]$. На головній діагоналі матриці визначити присутність цілого числа x та його індекс шляхом застосування алгоритму двійкового (бінарного) пошуку, якщо елементи цієї діагоналі впорядковані у порядку незростання. Вхід: двовимірний масив M , розмірність масиву n , шукане значення Key .

```
L = 0; R = N-1
поки L < R
{
    m = (L+R) / 2
    якщо a[m] < Key
        то L = m+1
    інакше R = m
}
якщо a[R] = Key
    то елемент має номер R
інакше елемент відсутній
```

Програмна реалізація:

```
#include <iostream>
using namespace std;
int main(){
    int const N = 5;
    int B[N], M[N][N];
    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            M[i][j] = rand() % 2;
            cout << M[i][j] << " ";
        }
        cout << endl;
    }

    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            if (i + j == 4){
                B[i] = M[i][j];
                cout << B[i] << " ";
            }
        }
    }

    int l = 0, r = N - 1, x, m, max, maxi;
    cin >> x;
    for (int i = 0; i < N; i++){
        if (B[i] == x){
            max = B[i]; maxi = i;
            cout << maxi << endl;
        }
    }

    while (l <= r){
        m = (l + r) / 2;
        if (B[m] == x){
            cout << "число присутнє на діагоналі" << endl;
            break;
        } else {
            if (B[m] < x){
                r = m - 1;
            } else {
                l = m + 1;
            }
        }
    }
    if (l > r){
        cout << "число відсутнє на діагоналі ";
    }
}
```

Варіанти завдань для індивідуального виконання:

1. Задано матрицю $A[m, n]$. Окремо у кожному рядку матриці визначити присутність цілого числа x та його індекс шляхом застосування алгоритму двійкового (бінарного) пошуку, якщо елементи кожного рядка окремо впорядковані у порядку не спадання.
2. Задано матрицю $A[m, n]$. Окремо у кожному стовпцеві матриці визначити присутність цілого числа x та його індекс шляхом застосування алгоритму двійкового (бінарного) пошуку, якщо елементи кожного стовпця окремо впорядковані у порядку не спадання.
3. Задано матрицю $A[n, n]$. На головній діагоналі матриці визначити присутність цілого числа x та його індекс шляхом застосування алгоритму двійкового (бінарного) пошуку, якщо елементи на головній діагоналі впорядковані у порядку не спадання.
4. Задано матрицю $A[m, n]$. У першому рядку та останньому стовпцеві матриці визначити присутність цілого числа x та його індекс шляхом застосування алгоритму двійкового (бінарного) пошуку, якщо елементи цих рядка та стовпця впорядковані у порядку неспадання.
5. Задано матрицю $A[m, n]$. В останньому рядку та першому стовпцеві матриці визначити присутність цілого числа x та його індекс шляхом застосування алгоритму двійкового (бінарного) пошуку, якщо елементи цих рядка та стовпця впорядковані у порядку неспадання.
6. Задано матрицю $A[m, n]$. В останньому рядку та останньому стовпцеві матриці визначити присутність цілого числа x та його індекс шляхом застосування алгоритму двійкового (бінарного) пошуку, якщо елементи цих рядка та стовпця впорядковані у порядку неспадання.
7. Задано матрицю $A[m, n]$. В першому рядку та останньому стовпцеві матриці визначити присутність цілого числа x та його індекс шляхом застосування алгоритму двійкового (бінарного) пошуку, якщо елементи цих рядка та стовпця впорядковані у порядку неспадання.

8. Задано матрицю $A[m, n]$. В другому рядку та $(n-2)$ -му стовпцеві матриці визначити присутність цілого числа x та його індекс шляхом застосування алгоритму двійкового (бінарного) пошуку, якщо елементи цих рядка та стовпця впорядковані у порядку неспадання.

9. Задано матрицю $A[m, n]$. В $(m-2)$ -му рядку та другому стовпцеві матриці визначити присутність цілого числа x та його індекс шляхом застосування алгоритму двійкового (бінарного) пошуку, якщо елементи цих рядка та стовпця впорядковані у порядку неспадання.

10. Задано матрицю $A[m, n]$. Окремо у кожному рядку матриці визначити присутність цілого числа x та його індекс шляхом застосування алгоритму двійкового (бінарного) пошуку, якщо елементи кожного рядка окремо впорядковані у незростаючому порядку.

Звіт до практичної роботи повинен містити розроблений алгоритм, виконуваний код програми. Здобувач повинен пояснювати хід виконання алгоритму, програми відповідати на питання щодо роботи алгоритму, програми, теми практичної роботи.

Контрольні питання

1. У чому полягає особливість методу декомпозиції?
2. Особливості алгоритмів, розроблених методом декомпозиції.
3. Назвіть задачі, для яких ефективні алгоритми рішення можуть бути розроблені методом декомпозиції.
4. У якому випадку може використовуватися бінарний пошук?
5. У чому його основна ідея бінарного пошуку?
6. Сформулюйте алгоритм бінарного пошуку.

Список рекомендованої літератури

Базова література

1. Крєневич А.П. Алгоритми і структури даних. Підручник. – К.: ВПЦ "Київський Університет", 2021. – 200 с. Доступне у дистанційному курсі.
2. Новотарський М.А. Алгоритми та методи обчислень [Електронний ресурс]: навч. посіб. для студ. спеціальностей 121 “Інженерія програмного забезпечення”, спеціалізації “Програмне забезпечення високопродуктивних комп’ютерних систем та мереж” та 123 “Комп’ютерна інженерія”, спеціалізації “Комп’ютерні системи та мережі”. - Електронні текстові дані. - Київ : КПІ ім. Ігоря Сікорського, 2019. — 407 с. — https://ela.kpi.ua/bitstream/123456789/27864/1/Alhorytmy_ta_metody_obchislenn.pdf Доступне у дистанційному курсі.
3. Алгоритми і структура даних: Навчальний посібник / В.М.Ткачук. - Івано- Франківськ: Видавництво Прикарпатського національного університету імені Василя Стефаника, 2016. - 286 с. Доступне у дистанційному курсі.
4. Алгоритми і структури даних [Текст]: практичний практикум для студентів напряму підготовки "Телекомунікації і радіотехніка" / Уклад.: М.А. Скулиш, С.В. Суліма,. – К.: КПІ ім. Ігоря Сікорського, 2021. –109с. Доступне у дистанційному курсі.
5. Курс лекцій з дисципліни «Алгоритми та структури даних» [Електронний ресурс] / Т.О. Гришанович; ВНУ імені Лесі Українки. Електронні текстові дані. Луцьк : ВНУ імені Лесі Українки, 2021. – 110 с. Доступне у дистанційному курсі.
6. Т. Кормен, Ч. Лейзерсон, Р. Рівест, К. Стайн, Вступ до Алгоритмів, 3-є вид. : Переклад з англ. — К. : К. І. С., 2019. — 1288 с. ISBN 978-617-684-239-2.

Додаткова література

7. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, Third Edition, The MIT Press, 2009, Cambridge, Massachusetts/London, England, 1292 p.
8. Alfred Aho, Jeffrey Ullman, John Hopcroft, Data Structures and Algorithms, Pearson, 1983. – 448 p.
9. Alfred Aho, Jeffrey Ullman, John Hopcroft, Design and Analysis of Computer Algorithms, Addison-Wesley; 1st edition, 1974, 470 p.
10. Steven S. Skiena, The Algorithm Design Manual, Second Edition, Springer-Verlag London Limited, 2008, 730 p.
11. Adam Drozdek, Data Structures and Algorithms in C++, Fourth Edition, Cengage Learning, 2012, 784 p.
12. Robert Sedgewick, Algorithms in C++ Part 5: Graph Algorithms, Third Edition, Addison-Wesley Professional, 2001, 528 p.
13. Donald E. Knuth, Art of Computer Programming, Volume 1: Fundamental Algorithms, Addison-Wesley Professional; 3rd edition, 1997, 650 p.
14. Donald E. Knuth, Art of Computer Programming, Volume 2: Seminumerical Algorithms, Addison-Wesley Professional; 3rd edition, 1997, 762 p.
15. Donald E. Knuth, Art of Computer Programming, Volume 3: Sorting and Searching, Addison-Wesley Professional; 3rd edition, 1998, 780 p.
16. Donald E. Knuth, Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Addison-Wesley Professional; 1st edition, 2011, 3168 p.
17. Niklaus Wirth, Algorithms + Data Structures = Programs, Prentice-Hall Inc., 1976, 381 p.