МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

С. С. Бульба, В. О. Бречко, Д. О. Лисиця, О. М. Бельорін-Еррера

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

**Навчально-методичний посібник
к для студентів спеціальності 123
«Комп'ютерна інженерія».**

Рекомендовано редакційно-
видавничою радою університету,
протокол № 1 від 15.02.2024 р.

Харків
НТУ «ХПІ»
2024

УДК 004
А 45

Рецензенти:
Олег ГРИБ, д-р техн. наук проф. НТУ "ХПІ"
Марія СКУЛИШ, д-р техн. наук, проф. Національного технічного університету України "Київський політехнічний інститут імені Ігоря Сікорського"

Алгоритми та структури даних : навч.-метод. посібник /
А 45 С. С. Бульба, В. О. Бречко, Д. О. Лисиця, О. М. Бельорін-Еррера.
– Харків : НТУ «ХПІ», 2024. – 121 с.

Викладено основні структури даних, розглянуто питання аналізу алгоритмів, способи досягнення максимальної асимптотичної продуктивності алгоритмів. Матеріал проілюстровано практичними прикладами, до всіх розділів наведено необхідні задачі та вправи. Для студентів спеціальності 123 «Комп'ютерна інженерія».

Табл. 19. Іл. 14 Бібліогр. 8

УДК 004

ВВЕДЕННЯ

"Хороший програміст повинен знати все, що написано до нього, тільки тоді він буде писати хороші програми.»
Дж Бентлі

"Погані програмісти турбуються про код. Хороші програмісти турбуються про структури даних та їх відносини».
Лінус Торвальдс, творець Linux

Слова Дж Бентлі і Л. Торвальдса можуть бути епіграфом до дисципліни «алгоритми та структури даних». Вони кажуть про необхідність вивчення великої кількості вже відомих алгоритмів та структур даних. У сучасному світі не для всіх завдань, що виникають під час роботи, необхідно писати програму. Бувають випадки, коли досить використовувати вже існуючу програму або об'єднати кілька відомих програм. Але це скоріше виключення з правил.

У загальному випадку розробка програмного продукту є складним процесом, в якому власне програмування – тільки один з його етапів. Тому мало знати будь-якої або будь-які мови програмування. Потрібно мати досвід алгоритмічного мислення і оптимального вибору структур даних. Від цього, в кінцевому підсумку, залежить складність і якість розроблюваного програмного виробу. Не кожен програміст може з легкістю розробляти якісні програмні продукти, які б задовольняли вимогам щодо швидкодії, ресурсів пам'яті, надійності, конфіденційності даних, тестування, розширюваності, взаємодії з іншими програмами та інше.

В рамках дисципліни «алгоритми і структури даних» планується приділяти увагу основним структурам даних, розглядати питання аналізу алгоритмів, способів зниження вимог до пам'яті, досягнення хорошої асимптотичної продуктивності алгоритмів.

Мета дисципліни – вивчити множину важливих і корисних алгоритмів; особливу увагу буде приділено фундаментальним алгоритмам, які треба знати і вміти застосовувати їх на практиці. Для досягнення мети на практичних заняттях основну увагу буде приділено реалізації алгоритмів і структур даних, визначенню їх порівняльних характеристик шляхом проведення великої кількості експериментів.

ЗМІСТ

ПРАКТИЧНА РОБОТА 1. ОЦІНКА ТРУДОМІСТКОСТІ АЛГОРИТМУ

Мета: освоєння аналітичних методів аналізу трудомісткості обчислювальних алгоритмів.

Загальні відомості. Алгоритм – це точна послідовність дій, яка за скінченне число кроків веде від довільного вхідного даного (або від деякої сукупності вхідних даних) до досягнення результату, який повністю визначається цими вхідними даними.

Аналізувати алгоритми прийнято за такими параметрами:

- трудомісткість;
- складність;
- продуктивність.

Обчислювальна складність – це функція залежності обсягу роботи, виконуваної деяким алгоритмом, від розміру вхідних даних. Оцінка складності алгоритму може бути дана в бітах, байтах, кількості символів певної мови. Попередня оцінка складності алгоритму та кількості даних виконується експертним шляхом, а точні значення можуть бути відомі тільки після завершення програмування. Складність алгоритму та кількість даних характеризують потребу задачі в оперативній та зовнішній пам'яті.

Трудомісткість алгоритму розуміється як кількість обчислювальної роботи, необхідної для його реалізації. Трудомісткість характеризує витрати часу для реалізації алгоритму на деякій сукупності технічних засобів. Зазвичай трудомісткість оцінюється кількістю процесорних операцій та операцій уведення-виведення. Слід відразу зазначити, що трудомісткість алгоритму є взагалі випадковою величиною і залежить від вхідних даних. Тому трудомісткість алгоритму може бути визначена тільки приблизно в термінах теорії ймовірності: математичними сподіванням, дисперсією і т.д.

Трудомісткість алгоритму приблизно може бути визначена як середня кількість процесорних операцій (Θ), необхідних для виконання алгоритму один раз. Для спрощення розрахунків операції уведення-виведення братися до уваги не будуть. Але за необхідності набір параметрів, що характеризують трудомісткість алгоритму, може бути розширений.

Початкова інформація для розрахунків трудомісткості алгоритму буде взята зі схеми алгоритму, який складається з операторів $V_1, V_2,..., V_{k-1}$ і за

яким розробляється програма.

Середня кількість звертань до операторів $V_1, V_2, \ldots, V_{k-1}$ позначається як $n_1, n_2, \ldots, n_{k-1}$, відповідно. Кожний з операторів V_i $(i=1, 2, \ldots, k-1)$ характеризується середньою кількістю процесорних операцій k_i, виконуваних в операторі V_i. Тоді трудомісткість алгоритму може бути оцінена за такою формулою формулі:

$$\Theta = \sum_{i=1}^{k-1} n_i q_i \qquad (1.1)$$

де Θ – середня кількість процесорних операцій, виконуваних при одному прогоні алгоритму; q_i – середня кількість процесорних операцій оператора V_i; n_i – середня кількість звертань до оператора V_i; k – кількість операторів в алгоритмі.

Для визначення середньої кількості звертань n_i до оператора V_i $(i=1,2,\ldots,k-1)$ як правило, роблять такі припущення:

– ймовірність виконання після оператора V_i оператора V_j дорівнює P_{ij} і є сталою величиною;

– ймовірність P_{ij} залежить тільки від оператора V_i, але ніяк не пов'язана зі способом потраплення в оператор V_j, тобто не залежить від передісторії обчислювального процесу. Тому для усіх операторів виконується умова:

$$\sum_{i=1}^{k-1} P_{ij} = 1$$

При виконанні вказаних вище припущень обчислювальний процес є марковським зі станами $S1, S2, \ldots, Sk$. При цьому оператори $V1, V2, \ldots, Vk-1$ відповідають станам $S1, S2, \ldots, Sk-1$. Стан Sk відповідає скінченній вершині графа алгоритму і є поглинаючим, тобто при досягненні стану Sk процес припиняється. Стани $S1, S2, \ldots, Sk-1$ називаються безповоротними, тому що процес неодмінно їх залишає.

Відомо кілька способів оцінок трудомісткості алгоритму; у роботі буде розглянуто два з них:

– метод теорії марковських ланцюгів;

– мережевий метод.

1. Оцінка трудомісткості алгоритмів методами теорії марковських ланцюгів

Для визначення середньої кількості процесорних операцій, виконуваних за один прогін програми, потрібно граф алгоритму записати у вигляді стохастичної матриці *P*.

Елементами матриці *P* є ймовірності переходу зі стану ***i*** у стан ***j***, які наведені на графі алгоритму. У загальному випадку можна записати (підсумовування за стовпцем стохастичної матриці):

$$n_i = \sum_{j=0}^{k-1} p_{ji} n_j, \quad (n_0 = 1,\ i = 1, 2, ..., k\text{-}1) \tag{1.2}$$

Останній запис являє собою систему лінійних алгебраїчних рівнянь, розв'язання яких дасть середню кількість звертань до операторів, що досліджуються. Для перевірки правильності розв'язання системи лінійних рівнянь можна використовувати очевидну тотожність:

$$n_k = \sum_{j=0}^{k-1} p_{jk} n_j = 1, \quad \text{при} \quad n_0 = 1 \tag{1.3}$$

2). Мережевий підхід до оцінки трудомісткості алгоритмів

Мережевий підхід зручний для аналізу вручну і дозволяє обчислювати середню, мінімальну та максимальну трудомісткість алгоритмів на графах, що не містять циклів. За наявності в графі алгоритму циклів останні слід замінити операторами з еквівалентною трудомісткістю. Якщо в графі алгоритму є вкладені цикли, то спочатку слід замінити еквівалентними операторами внутрішні цикли, а потім переходити до заміни зовнішніх циклів. Розрахунки трудомісткості алгоритму виконуються за наведеною вище методикою.

Позначимо через $\boldsymbol{p_c}$ ймовірність замикання циклу, тобто ймовірність переходу по дузі з кінця циклу в його початок. Тоді відповідно до виразу (1.3) можна записати:

$$n_c = 1 + p_c n_c, \tag{1.4}$$

де n_c – середня кількість повторень циклу.

З виразу (1.4) маємо:

$$n_c = \frac{1}{1-p_c} \tag{1.5}$$

Тоді середня кількість процесорних операцій циклу буде дорівнювати:

$$q_c = n_c q_{mc},$$

де q_{mc} – середня трудомісткість тіла циклу, визначається за формулі (1.1), але враховуються тільки оператори, що становлять тіло циклу; q_c – середня трудомісткість циклау.

Типове завдання

Розрахувати трудомісткість алгоритму, схему якого наведено на рис. 1.1. Використати мережевий метод та метод теорії марковських ланцюгів.

Для розрахунків трудомісткості алгоритму необхідно знати ймовірності переходів із логічних вершин при одиничному *(true)* значенні логічної умови. Якщо відповідну ймовірність позначити через *p*, то ймовірність виходу з логічної вершини при нульовому логічному значенні умови *(false)*, що перевіряється, буде дорівнювати *1 – p*.

Для подальших розрахунків схему алгоритму доцільно зображувати у вигляді графа алгоритму. Для цього потрібно перенумерувати всі оператори схеми алгоритму. У логічних операторах замість логічних умов «1» і «0» необхідно записувати відповідну до цього виходу ймовірність. Ймовірність виходу з операторної вершини дорівнює 1. Отриманий у такий спосіб граф алгоритму зображено на рис. 1.2.

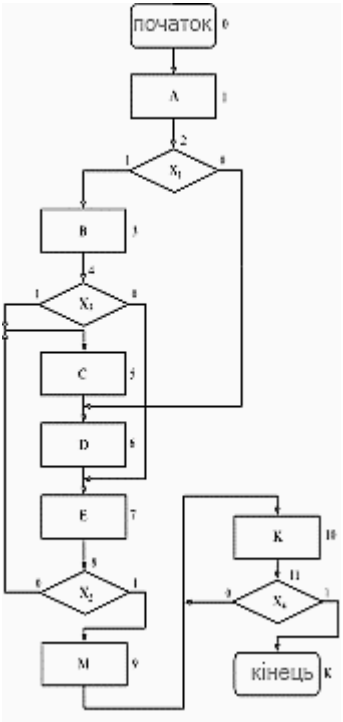


Рисунок 1.1 – Схема алгоритму

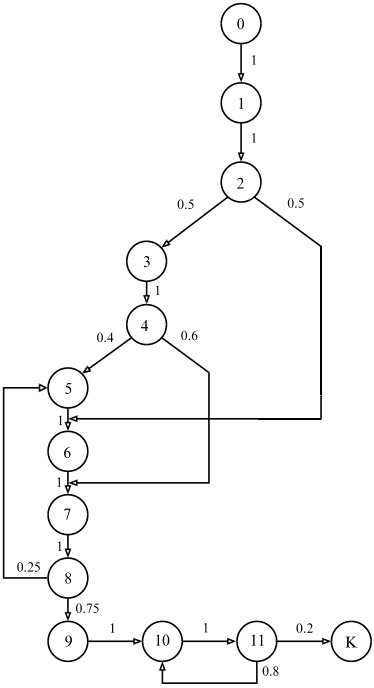


Рисунок 1.2 – Граф алгоритму

Граф алгоритму можна суттєво спростити, якщо трудомісткість виконання логічних вершин не значна порівняно з трудомісткістю виконання операторних вершин. Тоді стани, відповідні логічним вершинам, можна злити з попередніми їм станами, що відповідають операторним вершинам. У поданому прикладі можна злити стани (1, 2), (3, 4), (7, 8), (10, 11). Після перенумерації вершин одержано мінімізований граф алгоритму, зображений на рис. 1.3. Зрозуміло, що при достатньому досвіді до нього можна було б дійти відразу від схеми алгоритму, наведеної на рис. 1.1.

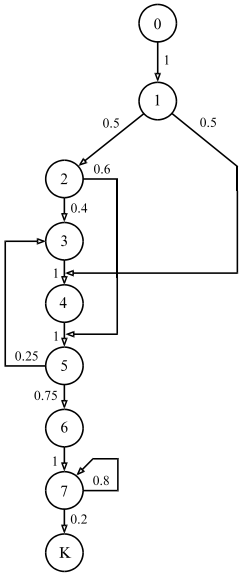


Рисунок 1.3 – Мінімізований граф алгоритму

1. Оцінка трудомісткості алгоритмів методами теорії марковських ланцюгів

Для визначення середньої кількості процесорних операцій, виконуваних за один прогін програми, граф алгоритму потрібно записати у вигляді стохастичної матриці *P*. Для графа (рис. 1.3) стохастичну матрицю подано в табл. 1.1. У таблиці в першому стовпчику вказані назви початкових станів (з яких виконується перехід), у першому рядку – назви скінченних станів (за які виконується перехід).

Таблиця 1.1 – Стохастична матриця

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_k
S_0	1	0	0	0	0	0	0	0
S_1	0	0.5	0	0.5	0	0	0	0
S_2	0	0	0.4	0	0.6	0	0	0
S_3	0	0	0	1	0	0	0	0
S_4	0	0	0	0	1	0	0	0
S_5	0	0	0.25	0	0	0.75	0	0
S_6	0	0	0	0	0	0	1	0
S_7	0	0	0	0	0	0	0.8	0.2

Зі стохастичної матриці випливає, наприклад, що у стані S_4 процес може виявитися при переході з S_1 з імовірністю 0.5 або при переході зі стану S_3 з імовірністю 1. За умови, що відомі середні числа звертань до вершин V_1 і V_3, то число звертань до вершини V_4 буде відповідно дорівнювати:

$$n_4 = p_{14}\, n_1 + p_{34}\, n_3 = 0.5 n_1 + n_3 .$$

Зі стохастичної матриці аналогично визначимо число звертань до усіх вершин графа:

$n_1 = 1 \cdot n_0 = 1 \cdot 1 = 1$; $\quad n_2 = 0.5 \cdot n_1 = 0.5 \cdot 1 = 0.5$;

$n_3 = 0.4 \cdot n_2 + 0.25 \cdot n_5 = 0.2 + 0.25 \cdot n_5$;

$n_4 = 0.5 \cdot n_1 + n_3 = 0.5 + n_3$; $\quad n_5 = 0.6 \cdot n_2 + n_4 = 0.3 + n_4$;

$n_6 = 0.75 \cdot n_5$; $\quad n_7 = n_6 + 0.8 \cdot n_7$.

У результаті розв’язання системи рівнянь одержано:

$n_1 = 1$; $n_2 = 0.5$; $n_3 = 0.533$; $n_4 = 1.033$; $n_5 = 1.333$; $n_6 = 1$; $n_7 = 5$.

Перевірка правильності розв’язання системи лінійних рівнянь

Для заданого алгоритму перевірка виконується так:

$$n_k = 0.2 \cdot n_7 = 0.2 \cdot 5 = 1.$$

Оскільки потраплення у скінченний стан n_k дорівнює 1, розрахунок вірний. За формулою (1.1) за умови, що q_i для всіх операторів V_i становить 1000 операцій, обчислюємо Θ:

Θ = (1 + 0,5 + 0.533 + 1.033 + 1.333 + 1 + 5) ·1000 = **10399** операцій.

Цей метод є універсальним (у випадку марковського процесу), але потребує більших витрат часу при значних розмірах стохастичної матриці.

2. Мережний підхід до оцінки трудомісткості алгоритмів.

Мережевий підхід зручний для аналізу вручну і дозволяє обчислювати середню, мінімальну та максимальну трудомісткість алгоритму на графах, що не містять циклів. Якщо граф містить цикли, спочатку потрібно визначити кількість звертань до кожного з циклів та

відповідну кількість процесорних операцій. Потім перебудувати граф, у якому кожний із циклів замінити на відповідний стан з обчисленою кількістю процесорних операцій.

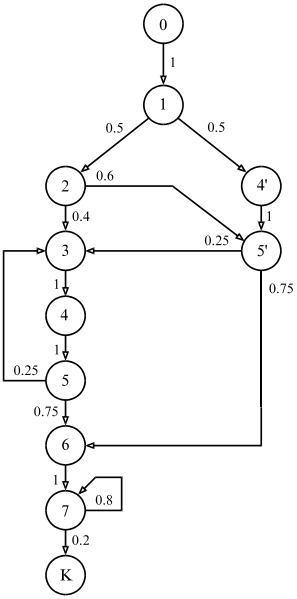
В алгоритмі на рис. 1.3, який розглядається, є два цикли. Перший містить оператори V_3, V_4, V_5, а другий складається тільки з оператора V_7. Перший цикл ускладнений входами усередину цикла з операторів V_1 і V_2 . Для того щоб позбавитися входів у цикл не через початок циклу V_3, зробимо елементарні перетворення графа алгоритму. Еквівалентний граф після очевидних перетворень зображений на рис. 1.4. З рис. 1.4 видно, що оператори V_4 і V_5, які використовувалися для входу в тіло циклу не через його початок, винесені окремо за номерами 4’ і 5’.

Визначимо кількість повторень циклів із виразу (1.5) :

$$n_{C1} = \frac{1}{1-0.25} = 1.33; n_{C2} = \frac{1}{1-0.8} = 5;$$

де n_{c1}, n_{c2} – число повторень першого та другого циклів.

Граф алгоритму без циклів наведено на рис. 1.5.



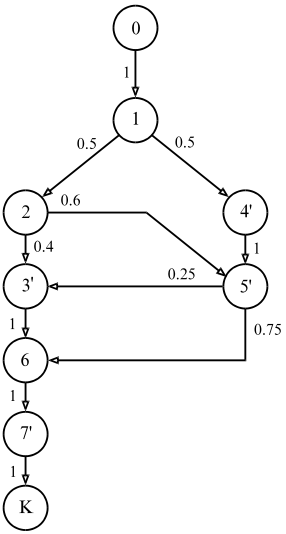


Рисунок 1.4 – Еквівалентний граф алгоритму

Рисунок 1.5 – Граф алгоритму без циклів

Розрахуємо середню кількість процесорних операцій для циклів С1 та С2:

$q_{C1} = 1000 \cdot 3 \cdot 1.333 = 3999$ оп.;

$q_{C2} = 1000 \cdot 5 = 5000$ оп.

Скориставшись формулою (1.2) для рис. 1.5, можна записати таке:

$n_0 = 1;$ $n_1 = 1 \cdot n_0 = 1;$

$n_2 = 0.5 \cdot n_1 = 0.5;$ $n_{4'} = 0.5 \cdot n_1 = 0.5;$

$n_{5'} = n_{4'} + 0.6n_2 = 0.5 + 0.6 \cdot 0.5 = 0.8;$

$n_{3'} = 0.4n_2 + 0.25n_{5'} = 0.4 \cdot 0.5 + 0.25 \cdot 0.8 = 0.4;$

$n_6 = n_{3'} + 0.75n_{5'} = 0.4 + 0.75 \cdot 0.8 = 1;$ $n_{7'} = 1 \cdot n_6 = 1.$

Через відсутність циклів система виявляється такою, що легко розв'язується. Витрати часу на аналіз можуть бути знижені додатково, якщо ввести ефективну нумерацію станів графа алгоритму. Останнє полягає в тому, що номер вершини має бути таким, щоб вхідні в цю вершину дуги починалися у вершинах з меншими номерами. Якщо після такої нумерації

рівняння записувати за зростанням їх номерів, то вони будуть відразу розв'язуватися, тому що невідомі з меншими номерами будуть уже обчислені. У наведеному прикладі нумерація вершин не відповідає зазначеній вимозі, але рівняння записувалися в порядку їх розв'язкання.

Наведений вище підхід дозволяє обчислити середню трудомісткість алгоритму.

За формулою (1.1) за умови, що q_i для всіх операторів V_i становить 1000 операцій обчислюємо Θ.

$$\Theta = 1000 + 500 + 0.4 \cdot 3999 + 500 + 800 + 1000 + 5000 = \mathbf{10399} \text{ оп.}$$

Висновки: результати розрахунку середньої кількості процесорних операцій, що визначена із використанням стохастичної матриці та алгоритму мережевого підходу, збіглися.

Для оцінки максимальної і мінімальної трудомісткості алгоритму необхідно перебрати всі можливі шляхи, що ведуть із початкової вершини графа алгоритму в скінченну, і вибрати з них такі шляхи, що дають максимальну і мінімальну трудомісткість.

За наявності циклів у графі алгоритму для тіла цикла визначається мінімальна та максимальна трудомісткість. Визначаються мінімальне і максимальне число повторень циклу та формуються відповідні еквівалентні за трудомісткістю оператори.

Оцінимо мінімальну та максимальну кількість процесорних операцій для графа алгоритму, що зображений на рис. 1.5. Для спрощення візьмемо, що всі оператори мають трудомісткість, яка дорівнює 1000 процесорних операцій. Через A_i і B_i позначимо відповідно мінімальну та максимальну кількість процесорних операцій, які будуть у момент виходу процесу з i-тої вершини графа.

Результати розрахунків.

1. Мінімальна кількість процесорних операцій

$A_0 = 0;$
$A_1 = \min(A_0) + 1000 = 1000;$
$A_2 = \min(A_1 \cdot 0.5) = 500 + 1000 = 1500;$
$A_{4'} = \min(A_1 \cdot 0.5) = 500 + 1000 = 1500;$
$A_{5'} = \min(A_2 \cdot 0.6, A_{4'}) = \min(1500 \cdot 0.6 = 900, 1500) = 900 + 1000 = 1900;$

$A_{3'} = \min(A_2 \cdot 0.4,\ A_5{*}0.25_{'}) = \min(1500 \cdot 0.4{=}600,\ 1900 \cdot 0.25{=}475) = 475 + 3999 = 4474;$

$A_6 = \min(A_{3'},\ A_{5'} \cdot 0.75_{'}) = \min(4474,\ 1900 \cdot 0.75{=}1425) = 1425 + 1000 = 2425;$

$A_7 = \min(A_6) + 5000 = 2425 + 5000 = \mathbf{7425};$

2. Максимальна кількість процесорних операцій

$B_0 = 0;$

$B_1 = \max(B_0) + 1000 = 1000;$

$B_2 = \max(B_1 \cdot 0.5) + 1000 = 500 + 1000 = 1500;$

$B_{4'} = \max(B_1 \cdot 0.5) + 1000 = 500 + 1000 = 1500;$

$B_{5'} = \max(B_2 \cdot 0.6,\ B_{4'}) = \max(1500 \cdot 0.6{=}900,\ 1500) = 1500 + 1000 = 2500;$

$B_{3'} = \max(B_2 \cdot 0.4\ ,\ B_{5'} \cdot 0.25) = \max(1500 \cdot 0.4{=}600,\ 2500 \cdot 0.25{=}\ 625) = 625 + 3999 = 4624;$

$B_6 = \max(B_{3'},\ B_5 \cdot 0.75_{'}) = \max(4624,\ 2500 \cdot 0.75{=}1875) = 4624 + 1000 = 5624;$

$B_7 = \max(B_6) + 5000 = 5624 + 5000 = \mathbf{10624}.$

Висновки: мінімальна кількість процесорних операцій менша за середню кількість, а максимальна – більша за середню кількість процесорних операцій. Отже, розрахунок вірний.

Індивідуальні завдання

1. З табл. 1.2 обрати логічну схему алгоритму (ЛСА) відповідно до варіанта.

У ЛСА символам «Поч.» і «Кін.» відповідають початковий і кінцевий оператори алгоритму. Символами *A, B, C, D, E, K, M* позначені функціональні оператори алгоритму. Символами x_1, x_2, x_3, x_4 позначені логічні умови. Якщо логічна умова дорівнює одиниці, то виконується наступний один оператор у ЛСА. Якщо логічна умова дорівнює нулю, то здійснюється перехід за стрілкою з відповідним індексом, у цьому випадку у логічної умови стрілка спрямована уверх (наприклад, $\uparrow^i$). У місці переходу з логічної умови стрілка спрямована вниз – $\downarrow^i$.

Нариклад, для схеми алгоритму, зображеної на рис. 1.1, ЛСА має вигляд :

Поч. $Ax_1\uparrow^1 Bx_3\uparrow^3\downarrow^2 C\downarrow^1 D\downarrow^3 Ex2\uparrow^2 M\downarrow^4 Kx4\uparrow^4$ Кін.

З табл. 1.3 вибрати ймовірності переходів при одиничних логічних

умовах.

2. За ЛСА побудувати графічну схему алгоритму, граф алгоритму та мінімальний граф алгоритму.

3. Визначити трудомісткість алгоритму методами теорії марковських ланцюгів.

4. Визначити середню трудомісткість алгоритму за допомогою мережевого підходу. Спочатку, якщо в алгоритмі є цикли, визначити середню трудомісткість циклів.

5. Обчислити мінімальну і максимальну трудомісткість алгоритму.

6. Проаналізувати отримані результати.

Таблиця 1.2 – Описи схем алгоритмів

Варіант	Логічна схема алгоритму
1	Поч. $Ax_1\uparrow^1Bx_2\uparrow^2\downarrow^3Cx_3\uparrow^3D\downarrow^2E\downarrow^4Mx_4\uparrow^4K\downarrow^1$ Кін.
2	Поч. $Ax_2\uparrow^2Bx_3\uparrow^3Cx_1\uparrow^1DE\downarrow^1\downarrow^3M\downarrow^2x_4\uparrow^4K\downarrow^4$ Кін.
3	Поч. $Ax_2\uparrow^2B\downarrow^3\downarrow^2Cx_3\uparrow^3\downarrow^1Dx_1\uparrow^1Ex_4\uparrow^4MK\downarrow^4$ Кін.
4	Поч. $A\downarrow^2\downarrow^1Bx_1\uparrow^1Cx_2\uparrow^2Dx_4\uparrow^4\downarrow^3Ex_3\uparrow^3M\downarrow^4K$ Кін.
5	Поч. $\downarrow^1Ax_1\uparrow^1Bx_2\uparrow^2Ex_3\uparrow^3C\downarrow^2\downarrow^3Mx_4\uparrow^4D\downarrow^4K$ Кін.
6	Поч. $A\downarrow^4Bx_2\uparrow^2Cx_3\uparrow^3D\downarrow^2\downarrow^3Ex_4\uparrow^4Mx_1\uparrow^1K\downarrow^1$ Кін.
7	Поч. $Ax_2\uparrow^2Cx_1\uparrow^1Dx_3\uparrow^3\downarrow^1E\downarrow^2\downarrow^3\downarrow^4Kx_4\uparrow^4MB$ Кін.
8	Поч. $Dx_1\uparrow^1Ex_2\uparrow^2B\downarrow^2\downarrow^3Ax_4\uparrow^4Mx_3\uparrow^3\downarrow^4CK\downarrow^1$ Кін.
9	Поч. $x_1\uparrow^1A\downarrow^1E\downarrow^2Bx_2\uparrow^2C\downarrow^4Dx_3\uparrow^3K\downarrow^3Mx_4\uparrow^4$ Кін.
10	Поч. $x_1\uparrow^1A\downarrow^4\downarrow^2Bx_2\uparrow^2Cx_3\uparrow^3E\downarrow^3Dx_4\uparrow^4K\downarrow^1M$ Кін.
11	Поч. $\downarrow^3Ax_1\uparrow^1Bx_2\uparrow^2C\downarrow^2D\downarrow^1Ex_3\uparrow^3Kx_4\uparrow^4M\downarrow^4$ Кін.
12	Поч. $x_4\uparrow^4Ax_1\uparrow^1B\downarrow^2Cx_2\uparrow^2D\downarrow^1Ex_3\uparrow^3K\downarrow^4M\downarrow^3$ Кін.
13	Поч. $Ax_1\uparrow^1Bx_2\uparrow^2C\downarrow^2\downarrow^1D\downarrow^3\downarrow^4Ex_4\uparrow^4Mx_3\uparrow^3$ K Кін.
14	Поч. $Ax_1\uparrow^1Bx_2\uparrow^2C\downarrow^2D\downarrow^3Ex_3\uparrow^3\downarrow^1Kx_4\uparrow^4M\downarrow^4$ Кін.
15	Поч. $\downarrow^4Ax_1\uparrow^1B\downarrow^1Cx_2\uparrow^2Dx_3\uparrow^3\downarrow^2K\downarrow^3Mx_4\uparrow^4$ Кін.
16	Поч. $\downarrow^1Ax_1\uparrow^1B\downarrow^4\downarrow^2Cx_2\uparrow^2Dx_4\uparrow^4EKx_3\uparrow^3M\downarrow^3$ Кін.
17	Поч. $\downarrow^1Ax_1\uparrow^1Bx_3\uparrow^3C\downarrow^3Dx_4\uparrow^4E\downarrow^2Kx_2\uparrow^2M\downarrow^4$ Кін.
18	Почч. $\downarrow^4Ax_4\uparrow^4Bx_1\uparrow^1C\downarrow^1Dx_3\uparrow^3Ex_2\uparrow^2K\downarrow^2M\downarrow^3$ Кін.
19	Поч. $x_1\uparrow^1A\downarrow^3Bx_2\uparrow^2Cx_4\uparrow^4D\downarrow^2Ex_3\uparrow^3\downarrow^1KM\downarrow^4$ Кін.
20	Поч. $\downarrow^4Ax_1\uparrow^1B\downarrow^1C\downarrow^2\downarrow^3DEx_2\uparrow^2Kx_3\uparrow^3Mx_4\uparrow^4$ Кін.
21	Поч. $x_1\uparrow^1A\downarrow^3Bx_2\uparrow^2Cx_4\uparrow^4D\downarrow^4E\downarrow^2K\downarrow^1Mx_3\uparrow^3$ Кін.
22	Поч. $x_2\uparrow^2A\downarrow^3Bx_1\uparrow^1C\downarrow^2Dx_4\uparrow^4E\downarrow^1\downarrow^4KMx_3\uparrow^3$ Кін.

23	Поч. $x_1\uparrow^1 Ax_2\uparrow^2 B\downarrow^2\downarrow^1 Cx_4\uparrow^4 D\downarrow^3 Ex_3\uparrow^3 KM\downarrow^4$ Кін.
24	Поч. $x_1\uparrow^1 A\downarrow^2 B\downarrow^1 Cx_2\uparrow^2 D\downarrow^3 Ex_3\uparrow^3\downarrow^4 Mx_4\uparrow^4 K$ Кін.
25	Поч. $x_3\uparrow^3 Ax_2\uparrow^2\downarrow^1 Bx_1\uparrow^1 C\downarrow^2\downarrow^3 D\downarrow^4 Ex_4\uparrow^4 MK$ Кін.

Таблиця 1.3 – Ймовірності переходу (за X=1)

№	P1	P2	P3	P4
1	0.1	0.3	0.6	0.9
2	0.2	0.2	0.7	0.8
3	0.3	0.1	0.8	0.7
4	0.4	0.2	0.9	0.6
5	0.5	0.3	0.8	0.5
6	0.6	0.4	0.7	0.4
7	0.7	0.5	0.6	0.3
8	0.8	0.6	0.5	0.2
9	0.9	0.7	0.4	0.1
10	0.8	0.8	0.3	0.2
11	0.7	0.9	0.2	0.3
12	0.6	0.8	0.1	0.4
13	0.5	0.7	0.2	0.5
14	0.4	0.6	0.3	0.4
15	0.3	0.5	0.4	0.3
16	0.2	0.4	0.5	0.2
17	0.1	0.3	0.6	0.1
18	0.2	0.2	0.7	0.2
19	0.3	0.1	0.8	0.3
20	0.4	0.2	0.9	0.4
21	0.5	0.3	0.8	0.5
22	0.6	0.4	0.7	0.6
23	0.7	0.5	0.6	0.7
24	0.8	0.6	0.5	0.8
25	0.9	0.7	0.4	0.9

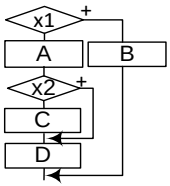
Таблиця 1.4 – Кількість процесорних операцій в операторах (у тисячах)

№	A	B	C	D	E	M	K
1	1	2	3	4	5	6	7
2	8	9	8	7	6	5	4
3	3	2	1	2	3	4	5
4	6	7	8	9	8	7	6
5	5	4	3	2	1	2	3
6	4	5	6	7	8	9	8
7	7	6	5	4	3	2	1
8	2	3	4	5	6	7	8
9	9	8	7	6	5	4	3
10	2	1	2	3	4	5	6
11	7	8	9	8	7	6	5
12	4	3	2	1	2	3	4
13	5	6	7	8	9	8	7
14	6	5	4	3	2	1	2
15	5	3	4	2	1	5	4
16	3	4	5	6	7	8	9
17	1	3	5	7	9	8	6
18	4	2	2	4	6	8	9
19	7	5	3	1	1	3	5
20	7	9	8	6	4	2	2
21	4	6	8	9	7	5	3
22	1	4	7	9	8	7	6
23	5	4	3	9	7	6	8
24	9	4	3	7	8	8	6
25	2	4	9	7	8	6	2

Контрольні запитання

1. Що таке «алгоритм»? Дайте визначення.
2. Які обов’язкові властивості алгоритму?
3. Що таке трудомісткість алгоритму?
4. Наведіть класифікацію алгоритмів за функцією трудомісткості.
5. Що таке складність алгоритму?
6. Як визначається складність алгоритму, і в яких випадках потрібна ця оцінка?
7. Як визначається трудомісткість алгоритму, і з якою метою обчислюється ця величина?
8. Чому трудомісткість алгоритма є, як правило, випадковою величиною?
9. Які параметри можуть бути використані для характеристики трудомісткості алгоритму?
10. Як виконати ефективну нумерацію станів у графі алгоритму для мережевого аналізу?
11. Що таке стохастична матриця?
12. Що таке марковський процес?
13. Як визначити ймовірності виходу з логічної вершини? Навести приклади.
14. Навіщо робиться припущення про обчислювальний процес як марковський при оціненні трудомісткості алгоритму?
15. Чи існує різниця в оцінценні середньої трудомісткості при використанні методів теорії марковських ланцюгів і мережевого підходу? Поясніть її.
16. Визначте трудомісткість наведеного фрагмента алгоритму.

У таблиці подано кількість операцій у блоках обчислення та ймовірність стверджувальних результатів у блоках вибору.



A	B	C	D	x1	x2
40	20	60	50	0,1	0,3

ПРАКТИЧНА РОБОТА 2. РЕКУРСИВНІ ТА ІТЕРАЦІЙНІ АЛГОРИТМИ

Мета: набути навичок та практичного досвіду у розробці рекурсивних програм.

Теми для попередньої роботи:

- ітераційні алгоритми;
- рекурсивні алгоритми.

Загальні відомості

Рекурсивним називається об'єкт, який частково утворений або визначений за допомогою самого себе.

Рекурсивною функцією можна описати нескінченне обчислення, причому така функція не буде містити явних повторень. Тому необхідно забезпечити закінчення її роботи. Найбільш надійний спосіб – увести в неї будь-який параметр (значення), назвемо його ***n***, і при рекурсивному звертанні до ***P***, як параметр задавати ***n – 1***. Якщо в цьому випадку як умова використовується ***n > 0***, то закінчення гарантоване. Це положення може бути виражено двома схемами:

$$P(n) \equiv if\ n>0,\ then\ P[S,P(n-1)]$$

або

$$P(n) \equiv P[S,\ if\ n>0,\ then\ P(n-1)].$$

Рекурсія є ефективним механізмом керування програмою. Коли метод викликає сам себе, у системний стек записується таке: адреса повернення, значення регістрів, параметри і нові локальні змінні. Як наслідок, код методу виконується з цими новими параметрами і змінними з самого початку. При рекурсивному виклику метода не створюється його нова копія, лише використовуються його параметри. А при поверненні з кожного рекурсивного виклику старі локальні змінні і параметри виштовхуються зі стека, читаються значення регістрів та адреса. Як наслідок, виконання програми поновлюється від точки виклику методу.

Рекурсивні варіанти багатьох процедур можуть виконуватися набагато повільніше, ніж їх ітераційні еквіваленти через додаткові витрати

системних ресурсів на багаторазові виклики методу. Якщо ж таких викликів буде дуже багато, то може бути переповнений системний стек. А оскільки параметри і локальні змінні рекурсивного методу зберігаються в системному стеку і при кожному новому виклику цього методу створюється їх нова копія, то в деякий момент стек може виявитися вичерпаним, що призведе до виключної ситуації.

Основна ж перевага рекурсії полягає в тому, що вона дозволяє реалізовувати деякі алгоритми ясніше і простіше, ніж ітераційний метод. Прикладом може бути алгоритм швидкого сортування, який складніше реалізувати ітераційним методом ніж рекурсивним.

Типові завдання

Завдання 1. Обчислити значення функції $f = x^n$.

Текст програми

```
#include <iostream>
#include <conio.h>
using namespace std;

int power( int x, int n); //  Повертає x у степені n (n>=0)

int main(void)
{   setlocale(LC_CTYPE, "rus");      // для роботи з кирилицею
    for ( int n = 0; n< 4; n++)
       cout << " 3 у степені "<< n << " дорівнює "<<power(3, n)<< endl;
    return 0;
}
int power( int x, int n)
{   if ( n <0)
    {    cout <<" Неприпустимий аргумент функції power.\n";
         exit(1);
    }
    if (n >0 )  return power(x, n-1)*x ;
    else return 1 ;// n == 0
}
```

Результат роботи програми

```
3 у степені 0 дорівнює 1
3 у степені 1 дорівнює 3
3 у степені 2 дорівнює 9
3 у степені 3 дорівнює 27
```

Класичним прикладом рекурсії є обчислення факторіала числа.

Завдання 2. Обчислити факторіал ***F = n!.***

Текст програми

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

 int FactR(int n) // рекурсивний метод
 {    int result;
      if (n==1)  return 1;
      result =  FactR(n-1) * n;
      return result;
 }
  int FactI(int n)  // ітераційний метод
 {   int result = 1;
     for (int t=1; t<=n; t++)  result *= t;
     return result;
 }

  void main()
 { setlocale(LC_CTYPE, "rus");       // для роботи з кирилицею
        cout << "Рекурсивний метод" << "5!="<< FactR(5) << endl;
              cout << "Ітераційнний метод" << "5!="<< FactR(5) << endl;
 }
```

Щодо рекурсивного методу ***FactR*** можна зазначить, що його дія організована за таким принципом, якщо метод викликається з аргументом 1, то він повертає значення 1. В протилежному випадку він повертає добуток ***FactR*(*n*-1) * *n*.** Для обчислення цього добутку потрібно ще раз викликати ***FactR***, але з аргументом ***n-1***. Цей процес повторюється доти, доки ***n*** не буде дорівнювати ***1***, після чого з попередніх викликів методу почнуть повертатися отримані значення. Наприклад, Коли обчислюється факторіал числа **2**, то при першому виклику методу ***FactR*** відбувається його повторний виклик з аргументом **1**. З цього виклику повертається значення ***1***, яке потім помножується на ***2***. Як результат буде ***2***.

Результат роботи программи

Рекурсивний метод 5! = 120
Ітераційний метод 5! = 120

Завдання 3. Вивести на екран рядок у зворотному порядку.

Текст програми

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

 void DisplayRev(string str)
```

```
{   if (str.length() > 0)
          DisplayRev(str.substr(1, str.length()-1));
    else  return;
    cout << str[0];
}

 void main()
{   setlocale(LC_CTYPE, "rus");     // для роботи з кирилицею
    string s = "Це тест";
    cout << "Початковий рядок: "<<  s << endl;
    cout << "Перевернутий рядок: ";
         DisplayRev(s);
   cout << endl;
}
```

Результат роботи программи

```
Початковий рядок: Це тест
Перевернутий рядок: тсет еЦ
```

Що разу, коли викликається метод ***DisplayRev,*** у ньому виконується перевірка символьного рядка, що поданий аргументом ***str***. Якщо довжина рядка не є нулем, то метод ***DisplayRev*** викликається рекурсивно з новим рядком, котрий менший від вхідного на один символ. Цей процес повторюється доти, доки методу не буде передано рядок нульової довжини. Після цього почне розкручуватися в зворотному порядку механізм усіх рекурсивних викликів методу ***DisplayRev***. При поверненні з кожного такого виклику на екран виводиться перший символ рядка, поданого аргументом ***str***. Таким чином, весь рядок виводиться в зворотному порядку.

Індивідуальні завдання

Розробити рекурсивний та ітераційний алгоритми розв'язання індивідуального завдання.

Визначити та порівняти час виконання відповідних функцій, зробити висновки.

1. Функція Акермана *A* визначається для всіх додатних (>0) цілих аргументів *m* і *n* так:

$$A(0,n) = n + 1;$$
$$A(m,0) = A(m - 1,1); \quad (m > 0)$$
$$A(m,n) = A(m - 1, A(m, n - 1)); \quad (m > 0, n > 0)$$

Обчислити значення функції *A(m,n)* для *m* і *n*, уведенні з клавіатури.

2. Знайти всі *n!* перестановок для *n* елементів $a_1 \ldots a_n$ та видати їх на

екран. Число *n* та самі числа, що переставляються, увести з клавіатури.

4. Дано текст (ланцюжок символів). Перевірити, чи є паліндромом ланцюжок символів, що починається з індексу *start* і закінчується індексом *end*.

5. Коефіцієнти, що утворюють трикутник Паскаля, визначаються так:

$$C(n,0) = 1;$$
$$C(n,n) = 1; \qquad (n > 0)$$
$$C(n,k) = C(n-1,k-1) + C(n-1,k); \qquad (n > 0, m > 0)$$

Розробити програму, що для заданого *n* будує трикутник Паскаля.

6. Алгоритм перетворення числа *N* з однієї системи числення в іншу (*B*) полягає в багаторазовому діленні на *B*. Якщо $N = d_{n-1}\, d_{n-2}\, d_{n-3} \ldots d_1\, d_0$, то послідовність остач від ділення в порядку $d_0 \ldots d_{n-1}$ дає цифри результату перетворення. Розробити функцію, що перетворює число *N* в систему числення *B*, припустити умову $B<=10$.

7. Для заданого n визначити усі числа Фібоначчі з інтервалу від 0 до n. Числа Фібоначчі визначаються так:

$$fib_{n+1} = fib_n + fib_{n-1},\ для\ n > 0; \qquad fib_0 = 0;\ fib_1 = 1;$$

8. Розробити програму для визначення розміру платежів за позичкою на $100.000 під 10 % річних на 20 років.

9. Обчислити корінь рівняння $f(x) = x^3 - 2x^2 - 3x + 10$ із заданою точністю в інтервалі $a <= x <= b$ методом дихотомії.

10. Метод дихотомії визначається так: якщо *f(a)* і *f(b)* мають різні знаки, то між *a* та *b* корінь є. Обчислюється середня точка $m = (a+b) / 2$. Якщо $f(m) = 0$, то корінь знайдено. В противному випадку перевіряються інтервали $a <= x <= m$ та $m <= x <= b$ і подальші дії виконуються в тому інтервалі, де знак функції змінюється. Процес продовжується, поки інтервал не стане достатньо малим, або не буде знайдено точне значення кореня.

11. Створити односпрямований список цілих чисел (числа читати з файла); видати вміст списку на екран; знайти найбільше число.

12. Створити дві множини *A* та *B* однакового розміру *n* цілих, додатних чисел, що не перетинаються (масиви з різними числами). Знайти всі пари $<a, b>$ з *n* вхідних пар, таких, що *a* належить до *A* і є парним, більшим *10*, *b* належить *B* і є непарним, кратним *5*.

13. Розробити програму, що визначає кількість *n*-розрядних двійкових чисел, які не мають у собі підряд двох одиниць. (Підказка: число починається з нуля або одиниці. Якщо – з нуля, то кількість варіантів визначається *(n – 1)* цифрами, що лишилися. А якщо з одиниці, то якою повинна бути наступна цифра?).

14. Дано текст (ланцюжок символів). Змінити послідовність ланцюжка символів, що починається з індексу *start* і закінчується індексом *end*, на зворотну послідовність.

15. Увести масив з *n* додатних (>0) цілих чисел. Визначити, чи можна з цих чисел вибрати такі, що їх сума дорівнює числу *s*. Якщо варіант існує, результат видати на екран. Всі числа уводити з клавіатури.

Наприклад, вхідні дані: 7, 5, 4, 4, 1; s = 10; 10 = 5 + 4 + 1.

16. Увести два масиви, таких, що їх елементи впорядковані за зростанням. Розробити програму злиття двох масивів у один масив, впорядкований за зростанням.

17. Визначити значення числа *a* в степені *n*. Значення *a* та *n у*вести з клавіатури.

18. Обчислити найбільший спільний дільник чисел *a* і *b*, використовуючи алгоритм Евкліда. Числа *a* і *b у*вести з клавіатури.

19. Розкласти натуральне число *a* типу integer на прості співмножники.

20. Для заданих x та n визначити значення функції

$$P(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + .. + \frac{x^{2n+1}}{(2n+1)!}.$$

Контрольні запитання

1. Який об’єкт називається рекурсивним?
2. На що треба звернути особливу увагу при розробці рекурсивного алгоритму?
3. Яка функція називається прямо рекурсивною? Наведіть приклад.
4. Яка функція називається непрямо рекурсивною? Наведіть приклад.
5. Наведіть приклади рекурсивного визначення функцій.
6. В чому полягає потужність рекурсивного визначення?
7. Що таке «дерево рекурсивних викликів»? Наведіть приклади.
8. Для чого рекурсивні програми потребують додаткової пам’яті порівняно із ітераційними програмами?
9. Що записується в стек при виклику функції?

10. Що відбувається, коли функція, яка викликалася, закінчує свою роботу?

11. Запишіть ітераційний варіант цієї рекурсивної функції

```
int  FR (int n)
{    if (n==0) return 1;
      return n* F(n-1);
}
```

12. Запишіть рекурсивний варіант цієї ітераційної функції

```
int  FI (int key)
{   for (long i=0; i<N; i++)
         if( m[i]== key) return i;
    return –1;
         }
```

ПРАКТИЧНА РОБОТА 3. ВНУТРІШНЄ ПОДАННЯ БАЗОВИХ СТРУКТУР ДАНИХ

Мета: отримати та закріпити знання про внутрішнє (комп'ютерне) подання числових типів даних у мовах програмування.

Теми для попередньої роботи:

- інформація та її подання в ЕОМ;
- адресація пам'яті;
- числові типи даних: цілі та дійсні;
- похибки подання дійсних чисел;
- статичні структури даних – масиви.

Загальні відомості. Розробка програм – творча діяльність, але існує багато стандартних методів, прийомів і алгоритмів, що спрощують процес програмування. Знання таких алгоритмів становить предмет техніки програмування. Передумовою оволодіння цією технікою є поглиблене вивчення структур даних, оскільки дані – це саме той матеріал, на якому базуються алгоритми. Організація роботи з даними впливає на ефективність алгоритмів, а в деяких випадках – і на їх працездатність. Тому так важливо вміти обрати такі структури даних, що відповідають поставленій задачі, оцінити потрібний обсяг пам'яті та час виконання майбутньої програми.

Перше, з чим стикаються при програмуванні, є числові дані.

Цілі числа. Дані цілого типу зберігаються у пам'яті ЕОМ у додатковому коді. Для додатних чисел додатковий код співпадає з прямим кодом. Найстарший біт найстаршого байта пам'яті, що виділяється для зберігання цілих чисел, вважається знаковим.

Приклад 1. Подання цілого числа 258 в пам'яті у форматі int.

У двійковій системі числення це число можна подати так:

$$258_{10} = 256_{10} + 2_{10} = 2^8{}_{10} + 2^1{}_{10} = 1\ 00000010.$$

Дані цілого типу зберігаються в двох або чотирьох байтах. Доповнене число до чотирьох байтів незначущими нулями буде таким:

мол. байт 00000010 00000001 00000000 **0**0000000 старш. байт.

Знаковий біт (позначений напівжирним написом) дорівнює нулю, це є ознакою додатних чисел; число 258 – додатне.

Приклад 2. Подання цілого числа – 258 в пам'яті у форматі int. Запишемо прямий, зворотнний та додатковий коди цього числа у форматі чотирьох байтів.

Прямий код: 00000000 00000000 00000001 00000010
Зворотний код: 11111111 11111111 11111110 11111101
Додатковий код: 11111111 11111111 11111110 11111110
Внутрішнє подання:
(молодш. байт) 11111110 11111110 11111111 **1**1111111 (ст. байт).

Знаковий біт (позначений напівжирним написом) дорівнює одиниці, це є ознакою від'ємних чисел; число – 258 – від'ємне.

Дійсні числа. Дані дійсних типів подаються у пам'яті у вигляді таких складових: знак мантиси, нормалізована мантиса та характеристика. Усі складові записуються у прямому коді.

Характеристика (X) визначається так: $X = 2^{n-1} + p - 1$,
де p – порядок числа, який визначається при його нормалізації; *n* – кількість розрядів, що виділяються на характеристику; визначається типом.

Нормалізована мантиса (M) належить інтервалу: $1 \leq M < 2$. Ціла частина нормалізованої мантиси, як правило, у пам'ять не записується. Це так звана схована одиниця. Найстарший біт найстаршого байта пам'яті, що виділяється для зберігання дійсних чисел, є знаковим.

Приклад 3. Подання в пам'яті числа 10,25 у форматі float.

У двійкову систему числення окремо переводиться ціла та дробова частини числа.

Ціла частина: $10_{10} = 1010_2$. Дробова частина 0,25 ×2
→ 0,50 ×2
→ 1,00 ×2

Стрілками позначені значення дробових розрядів двійкового числа 0,01.
Таким чином, усе число до нормалізації є: 1010,01
Після нормалізації буде таке число: 1,01001

Кома при нормалізації перенесена на три позиції вперед, тобто p = 3.

Обсяг пам'яті для даних типу float становить 4 Б, з них для характеристики – 8 розрядів. Відповідно, $X = 2^{8-1} + 3 - 1 = 2^7 + 2 = 2^7 + 2^1$.

Удвійковій системі числення характеристика буде такою: 10000010.

Внутрішнє подання числа буде таким:

молодш. байт 00000000 00000000 00100100 **0**1000001 старш. байт

Мантиса записується в пам'ять без схованої одиниці.

Для перевірки своїх знань при вивченні цього матеріалу студент може скористатися спеціально розробленою навчальною програмою. Там описані всі розряди числа з помітками про їх призначення.

Приклад 4. Поданняв пам'яті числа – 15,18 у форматі double.

У двійкову систему числення окремо переводиться ціла та дробова частини числа.

Ціла частина $15_{10}=1111_2$. Дробова частина 0,75 ×2

→ 1,50 ×2

→ 1,00 ×2

Стрілками позначені значення дробових розрядів двійкового числа 0,11.

Таким чином, усе число до нормалізації є: 1111,11

Після нормалізації число має вигляд: 1,11111

Кома при нормалізації перенесена на три позиції вперед, тобто p = 3.

Обсяг пам'яті для даних типу double становить 8 Б, з них для характеристики – 11 розрядів.

Відповідно, $X = 2^{n-1} + 3 - 1 = 2^{10} + 2 = 2^{10} + 2^1$ (1000000 0010)

Внутрішнє подання числа буде таким:

(молодш. байт) 00000000 00000000 00000000 00000000
00000000 10000000 00101111 11000000 (ст. байт).

Мантиса записана без схованої одиниці.

Масиви. Часто в програмуванні один і той самий алгоритм призначається для обробки не одного, а набору елементів. Набір даних одного типу називається масивом, що характеризується:

– фіксованим набором елементів однакового типу;

– кожен елемент має унікальний набір значень індексів;

– кількість індексів визначає мірність масиву, наприклад, два індекси – двовимірний масив, три індекси – тривимірний масив, один індекс – одновимірний масив або вектор;

– звернення до елемента масиву виконується за іменем масиву і значенням індексів для цього елемента.

У мові програмування C/C++ індексація елементів у масивах обов'язково починається з нуля. Елементи масиву можуть мати складну структуру. Найважливіша операція для масиву – доступ до заданого елемента. Для цього потрібно обчислити адресу елемента. Компілятор кожне звертання до елемента масиву замінює на вираз, в якому обчислюється адреса саме цього елемента. Так, звертання до елемента вектора буде замінено таким виразом: *@Ім'я[i] = @Ім'я + i· Sizeof(mun).*

Типові завдання

Завдання 1. Видати на монітор двійкове подання даних цілого типу без знака.

Текст програми

```
#include <iostream>
#include <conio.h>
using namespace std;

void BYTE(unsigned char A) // виведення вмісту байта
{        for(int bit = 128; bit >= 1; bit >>= 1)
                 cout << (A & bit ? '1' : '0');
         cout << ' ';
}
void main (void)
 {   unsigned x;
     cout<<"Enter unsigned int value   " ;
     cin >> x;
     unsigned char *p =  (unsigned char*) &x;
     for(int byte = 0; byte < sizeof(unsigned); byte++, p++ )
         BYTE(*p);
     cout << endl;
}
```

Результат роботи програми

```
Enter unsigned int value   258
```

00000010 00000001 00000000 00000000

Завдання 2. Видати на монітор двійкове подання даних цілого типу *int* та дійсного типу *double*. Для роботи використати шаблонну функцію.

```
#include "stdlib.h"
#include "iostream"
#define PB(value)(byte*)(value)
using namespace std;
typedef unsigned char byte;
typedef unsigned int dword;
template <class T> void show (const T &value)
{        for (dword i=0; i < sizeof(T);i++)
         {        byte Byte =*(PB(&value)+i);
                  for (dword j=0; j<8; j++)
                           cout << dword((Byte >> (7-j))&1);
```

```
                cout << ‘ ‘;
        }
        cout << endl;
}
void main()
{       int val1=0;
double val2=0;
        cout<<"Enter int value :";
        cin>>val1;
        show(val1);
cout<<"Enter double value :";
        cin>>val2;
        show(val2);
}
```

Результат роботи програми:

Enter int value : 12

00001100 00000000 00000000 00000000

Enter double value : 12.34

10101110 01000111 11100001 01111010 00010100 10101110 00101000 01000000

Індивідуальні завдання

Написати програму, яка виводить на екран внутрішнє (комп’ютерне) подання даних чотирьох типів. Типи даних обрати за табл. 3.1 згідно із своїм номером у журналі групи. Тип елементів масиву обрати за своїм розсудом.

За результатами роботи підготувати звіт з лабораторної роботи, де навести отримані результати та дати щодо них пояснення, зробити висновки.

Таблиця 3.1 – Індивідуальні завдання

№ З/П	integer	short int	long int	float	double	long double	char	[] ***n***-вимірний
1	2	3	4	5	6	7	8	9
1	*			*			*	1
2	*			*			*	2
3	*			*			*	3
4		*			*		*	1
5		*			*		*	2
6		*			*		*	1
7			*			*	*	2
8			*			*	*	3
9			*			*	*	1
10	*				*		*	2

11	*				*		*	3
12	*				*		*	1
13		*		*			*	2
14		*		*			*	3
15		*		*			*	1
16			*		*		*	2
17			*		*		*	3
1	2	3	4	5	6	7	8	9
18			*		*		*	1
19	*					*	*	2
20	*					*	*	3
21	*					*	*	1
22		*				*	*	2
23		*				*	*	3
24		*				*	*	1

Примітка: в таблиці подано типи даних мовою С/С++.

Контрольні за питання

1. У вигляді яких складових подаються у пам’яті дані цілих типів?

2. У якому коді подаються у пам’яті цілі дані?

3. Скільки пам’яті виділяється для даних типу коротке ціле, ціле, довге ціле?

4. У вигляді яких складових подаються у пам’яті дані дійсних типів?

5. У якому коді подаються у пам’яті дані дійсних типів?

6. Як будуть подані в пам’яті числа 255 та –255 у форматі короткого цілого?

7. Як будуть подані в пам’яті числа –2 та 2 у форматі короткого дійсного?

8. Чим визначається похибка подання дійсних чисел?

9. Що таке «прихована одиниця»? Навіщо вона потрібна?

10. З якою точністю видаються дійсні числа?

11. Яке ціле число зберігається в пам’яті (у десятковій системі числення), якщо його фізичне подання (вміст пам’яті) у форматі short int таке: 11111110 ?

12. Яке дійсне число зберігається в пам’яті (у десятковій системі числення), якщо його фізичне подання (вміст пам’яті) у форматі float таке:

(молодш. байт) 0..0 0..0 11000000 11000000 (ст. байт) ?

ПРАКТИЧНА РОБОТА 4. ВНУТРІШНЄ ПОДАННЯ ІНТЕГРОВАНИХ СТРУКТУР ДАНИХ

Мета: отримати та закріпити знання про внутрішнє подання інтегрованих структур даних у мовах програмування.

Теми для попередньої роботи:

- фізичне та логічне подання даних;
- інтегровані типи даних;
- вирівнювання даних.

Загальні відомості.

Структура – скінченна множина полів різного типу даних, які об'єднані загальним іменем. Структури є надзвичайно зручним засобом для подання програмних моделей реальних об'єктів предметної області, тому що частіше кожний такий об'єкт має набір властивостей, які характеризуються даними різних типів.

Структуру в пам'яті компілятор C++ подає у вигляді послідовності полів, що займають безперервну ділянку пам'яті. При такій організації за значенням вказівника на початок виділеної ділянки пам'яті і зсуву поля відносно початку визначається адреса цього поля запису. Таке фізичне подання структур дає економію пам'яті, але потребує часу на обчислення адреси полів структури.

Структура може мати варіантну частину. В такому разі структура складається з двох частин. У першій частині описуються поля, загальні для усіх груп об'єктів, які моделюються структурою. Серед цих полів, як правило, буває поле, значення якого дозволяє ідентифікувати групу, до якої цей об'єкт належить і, отже, який з варіантів другої частини структури повинен бути використаний при обробці. Друга частина структури містить описи властивостей, які не перетинаються (є унікальними), – для кожної підмножини таких властивостей приводиться окремий опис. Правила мови C++ вимагають іменування кожного варіанта. Ідентифікація поля, яке перебуває у варіантній частині запису, при звертанні до нього ускладнюється так, що треба вказувати таке:

<ім'я змінної-запису>.<ім'я варіанта>.<ім'я поля>

Під структуру з варіантами виділяється обсяг пам'яті, достатній для розміщення найбільшого варіанта. Якщо ж виділена пам'ять

використовується для меншого варіанта, частина її залишається невикористовуваною. Загальна для всіх варіантів частина запису розміщується так, щоб зсуви усіх полів відносно початку запису були однаковими для всіх варіантів. Очевидно, що найбільш просто це досягається розміщенням загальних полів на початку структури, але це не строго обов'язково. Варіантна частина може бути розміщена між полями загальної частини. Оскільки в кожному разі варіантна частина має фіксований максимальний розмір, зсуви полів загальної частини також залишаться фіксованими.

Вирівнювання даних. Це – дуже важливе поняття, яке повинен брати до уваги кожний програміст, який працює з пам'яттю напряму. Вирівнювання даних впливає на швидкість виконання програм, а інколи – на те, чи будуть вони взагалі працювати. Наприклад, більшість 32-х розрядних комп'ютерів влаштовані так, що машинні слова читаються швидко за умови, коли вони скомбіновані в четвірки (по чотири байти) і адресовані від нуля, а от машинні слова з адресою не кратною 4 – читаються повільно. Тому компілятори більшості мов програмування (так само і С) проводять таку оптимізацію, що між полями структури вставляються незначущі байти для того, щоб усі поля були вирівняними.

Наприклад, для структури:

```
struct
{   char x; // 1 Б
    int y;  // 4 Б
    char z; // 1 Б
};
```

буде виділено 12 Б пам'яті так, як на рис. 4.1.

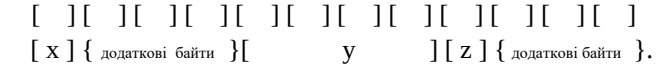


Рисунок 4.1 – Приклад вирівнювання полів структури

Як наслідок, додатково буде виділено 6 Б пам'яті, які ніяк не використовуються.

Вирівнювання полягає в тому, щоб адреса, яка вирівнюється, визначалася як числова адреса за модулем степеня 2. Елемент даних називається вирівняним природно, якщо його адреса вирівняна за розміром цього елемента. А якщо ні, то він буде невирівняним. Наприклад, 8-байтовий елемент даних із рухомою комою вирівняний природно, якщо

адреса, використовувана для його ідентифікації, вирівняна за 8. Компілятори мов програмування намагаються розподілити дані таким чином, щоб не допустити неузгодженості даних.

Те, як проводиться вирівнювання даних, і, як наслідок, – розмір структури в пам'яті залежить від настроювань компілятора та від директив у програмному коді. У більшості випадків турбуватися про вирівнювання не потрібно, тому що вирівнювання за замовчуванням уже є оптимальним.

У наступному прикладі коду показано, як компілятор розміщує доповнену структуру в пам'яті:

```
struct Tstr2
{  char a;     // 1 byte
   char _a[7];  // доповнення до 8 byte
   double b;   // 8 bytes
   short c;    // 2 bytes
   char _з[6];  // доповнення до 8 byte
 };
```

У випадку використання масиву вирівняних структур даних вирівняною в загальному випадку виявляється тільки перша структура масиву. Унаслідок чого компілятор додає додаткові невикористовувані байти у кінець кожної оброблюваної структури. Цей процес відбувається таким чином, щоб розмір структури був кратний 2, 4 або 8.

Інколи можна добитися значного виграшу в продуктивності або забезпечити ощадливу витрату пам'яті шляхом настроювання вирівнювання для структур даних.

Настроювання вирівнювання за допомогою директив компілятора. До появи Visual Studio 2015 для позначення вирівнювання, що перевищує значення за замовчуванням, застосовували ключові слова для систем Microsoft, такі як ***_alignof(<тип>)*** і ***__declspec(align(<число>))*** (з подвійним знаком підкреслення),

_alignof(<тип>) поверне поточні вимоги вирівнювання для цього типу,

__declspec(align(<число>)) використовується для примусового вирівнювання за типом даних (за умови, що він більше або такий же точний, ніж те, що***_alignof*** указав би для цього типу даних).

У наступному прикладі показано застосування ключових слів ***align*** і ***alignof***.

```
__declspec(align(32)) struct Str1 //задається вирівнювання за межею 32 Б
{  int a, b, c, d, e;
};
void main ()
```

```
{   Str1 str;
    std::cout << sizeof (str) << "  " << _alignof (str) << std::endl;
}
```

Директива компілятора #pragma pack (push, <число>) задає вирівнювання в <число> байт.

Директива #pragma pack(pop) повертає попереднє настроювання. Рекомендується повертати попереднє настроювання, що дозволить уникнути помилок виконання. Її пишуть після опису структури.

Типове завдання

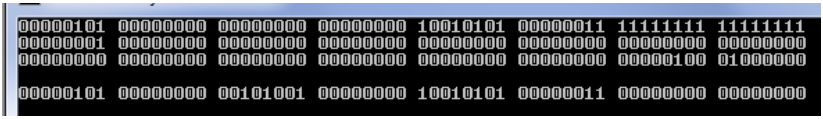
Видати на монітор фізичне подання даних типу «структура» з бітовими полями та варіантними частинами.

Текст програми

```
#include <iostream>
#include <conio.h>
using namespace std;

void BYTE(unsigned char A) // виведення вмісту байта
{        for(int bit = 128; bit >= 1; bit >>= 1)
                 cout << (A & bit ? '1' : '0');
         cout << ' ';
}
struct Tstruct1
{     int s;      /* 4 bytes */
short flip1:1;
short flip2:1;
short flip3:1;
short flip4:1;
short flip5:1;
short flip6:4;
     short flip7:4;
     union
     {  struct
         {   bool  b;
             double dl;
         } prim1;
       struct
         {  short st;
         }        prim2;
    } un;
};
struct Tstruct2
{    short s;     /* 4 bytes */
     int flip1:1;
     int flip2:1;
     int flip3:1;
     int flip4:1;
     int flip5:1;
     int flip6:4;
     int flip7:4;
};
void main()
```

```
{    Tstruct1 obj1 = {5, 1, 0, 1, 0, 1, 12, 1};
     obj1.un.prim1.dl = 2.5;
     obj1.un.prim1.b = true;
     unsigned char *p =  (unsigned char*) &obj1;
     int byte = 0;
     for( ; byte < sizeof(Tstruct1); byte++, p++ )
     {   if (byte && !( byte % 8) ) cout << endl;
                 BYTE(*p);
     }
     cout << endl;
     cout << endl;
     Tstruct2 obj2 = {5, 1, 0, 1, 0, 1, 12, 1};
     p =  (unsigned char*) &obj2;
     for(int byte = 0; byte < sizeof(Tstruct2); byte++, p++ )
         BYTE(*p);
     cout << endl;  cout << endl;
}
```

Результат роботи програми



Висновки за отриманими результатами

Отримані результати свідчать, що змінній *obj1* виділено 24 Б пам'яті так, що:

– для поля *s* – 4 Б;

– для полів *flip1* - *flip7* – 2 Б, ще 2 Б додано для вирівнювання;

– для варіантної частини – 16 Б , після однобайтового поля *b* додано 7 Б .

Змінній *obj2* виділено 8 Б пам'яті так, що:

– для поля *s* – 2 Б , ще 2 Б додано для вирівнювання;

– для полів *flip1* - *flip7* – 4 Б.

Індивідуальні завдання

Написати програму, яка виводить на екран внутрішнє подання структури з варіантною частиною та з бітовими полями, а також масиву структур. Перелік властивостей та відповідні типи полів для об'єктів з табл. 4.1 обрати за своїм розсудом.

Дослідити, як виконується вирівнювання даних та полів структури.

Порівняти час доступу до даних з вирівнюванням та за умови відсутності вирівнювання. За результатами роботи підготувати звіт з лабораторної роботи, де навести отримані результати та дати щодо них пояснення, зробити висновки.

Таблиця 4.1 – Індивідуальні завдання

N	Об’єкт	Тип об’єкта	Стан (так/ні)
1	Автомобільний транспорт	пасажирський, грузовий	– є обігрів кабіни, – вимкнено фари
2	Освіта	вища, середня	– технічна, – є працевлаштування
3	Прикладна програма	офісна, бухгалтерська	– платформозалежна, – платна
4	Залізничний потяг	пассажирський, грузовий	– рух за графіком, – зупинки на усіх станціях
5	Гра	комп’ютерна, настільна	– для дітей, – колективна
6	Морський транспорт	суховантажник, танкер	– завантажений, – іноземний
7	Набір даних – масив	статичний, динамічний	– читання з файлу, – відсортований
8	Тварини	свійські, дикі	– парнокопитні, – травоїдні
9	Джерело струму	батарейка, акумулятор	– заряджено, – багаторазове використання
10	Рівень освіти	магістратура, бакалаврат	– вступ за іспитами, – захист випускної роботи
11	Геометричні фігури	тривимірні, двовимірні	– різнокольорові, – паперові
12	Телефон	стаціонарний, мобільний	– кнопковий, – є вихід в інтернет
13	Дерева	садові, дикі	– косточкові, – ранньої стиглості
14	Підприємство	державне, приватне	– наявність сайта, – наявність корпоративної культури
15	Кава	зернова, розчинна	– наявність кофеїну, – легкість приготування
16	Меблі	офісні, для дому	– корпусні, – вільна модифікація

17	Ліки	протизапальні, жарознижуючі	– за рецептом, – вітчизняне виробництво
18	Література	технічна, художня	– різнокольорові рисунки, – жорстка палітурка

Контрольні запитання

1. Чим відрізняються машини з порядком Little Endian від машин Big Endian?
2. Що таке «вірівнювання» і навіщо воно потрібно?
3. Як можна змінити вирівнювання даних?
4. Яким може бути фізичне подання структури?
5. Як розподіляється пам’ять для структур з варіантними частинами?
6. Як розподіляється пам’ять для структур з бітовими полями?
7. Наведіть приклад доступу до полів структури з варіантної частини.
8. Яке призначення дискриптора структури, хто його створює?
9. Чи залежить обсяг пам’яті, що розподіляється для структури, від послідовності полів у її складі?
10. Які директиви компілятора призначені для впливу на вирівнювання даних?
11. Чи залежить адреса, яку призначає компілятор даним, від типу цих даних?
12. Від чого залежить розмір структури в пам’яті комп’ютера?
13. Як вирівнюються структури у масиві структур?

ПРАКТИЧНА РОБОТА 5. ФІЗИЧНЕ ПОДАННЯ СПЕЦИФІЧНИХ МАСИВІВ

Мета: набуття і закріплення навичок програмування розміщення в пам'яті специфічних масивів.

Теми для попередньої роботи:

- фізичне та логічне подання масивів;
- поняття про дискриптор;
- специфічні масиви: розріджені, асоціативні, симетричні, дерево відрізків.

Загальні відомості

Розріджений масив, або розріджена матриця (sparse array), – це масив, у якому не всі елементи використовуються, є в наявності або потрібні в поточний момент. Розріджена матриця – це матриця з переважно нульовими або фоновими елементами. У випадку, коли більша частина елементів матриці – ненульові, матриця вважається ***щільною***. Серед фахівців немає єдності у визначенні того, яка саме кількість ненульових елементів робить матрицю розрідженою. Різні автори пропонують різні варіанти.

Розріджені масиви корисні за таких умов:

1) розмір масиву, який потрібний додатку, досить великий (можливо, перевищує обсяг доступної пам'яті);

2) не всі елементи масиву є інформативними і використовуваними.

Таким чином, розріджений масив – це, як правило, великий, але рідко заповнений масив, більшість елементів якого не є інформативними (їх значення за замовчуванням – 0 або null, та можуть бути й інші, але обов'язково однакові). Зберігання великої кількості нулів у масиві не ефективне як для зберігання, так і для обробки масиву. У розрідженому масиві можливий доступ і до фонових елементів. У цьому випадку масив поверне нуль (якщо це масив чисел) або null (у випадку масиву об'єктів).

Якщо знадобиться масив більшого розміру, ніж дозволяють можливості комп'ютера, реалізовувати його треба якимось іншим способом. Навіть якщо великий м'асив і розміститься в пам'яті, створення його може суттєво зменшити доступні ресурси системи, оскільки пам'ять, зайнята великим масивом, виявляється недоступною для іншої частини

програми та інших процесів, що працюють у системі. А це може негативно позначитися на загальній продуктивності програми або комп'ютера в цілому. У тих випадках, коли будуть використовуватися не всі елементи масиву, виділення пам'яті під увесь масив є марною витратою системних ресурсів.

Щоб позбутися проблем, викликаних потребою в пам'яті для великих рідко заповнених масивів, були запропоновані незвичайні алгоритми роботи з розрідженими масивами. Усі вони характеризуються однією загальною рисою: пам'ять для елементів масиву виділяється тільки при необхідності. Тому перевага розрідженого масиву полягає в тому, що для його зберігання потрібно саме стільки пам'яті, скільки потрібно для зберігання тільки тих елементів, які дійсно використовуються. При цьому вільна пам'ять може використовуватися для інших цілей. Крім того, такі методи дозволяють створювати дуже великі масиви, розмір яких значно більший, ніж розмір звичайних масивів, що допускається системою.

Типове завдання

Розробити спосіб економного розміщення в пам'яті матриці цілих чисел, в якій нульові елементи розташовані нижче головної діагоналі.

У внутрішньому поданні масиву немає необхідності зберігати нульові (фонові) елементи – це такі, для яких виконується умова:

$$M[x, y] = 0 \text{ при } x < y.$$

Якщо виключити нульові елементи зі збереження і подати матрицю у вигляді одновимірного масиву, то формула лінеаризації (переходу від двокоординатного подання до однокоординатного) запишеться як:

$$j = \sum_{i=0}^{x}(N - i) + (y - x)\,,$$

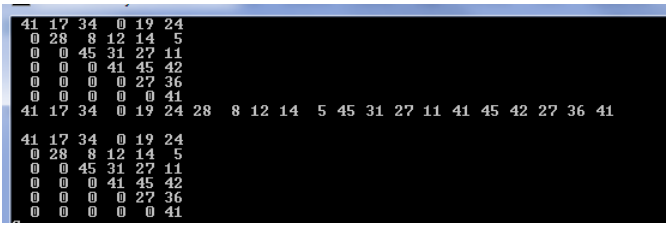
де x, y – номера стовпця і рядка відповідно у розрідженій матриці; N – кількість елементів у рядку; j – номер відповідного елемента у одновимірному масиві.

Текст програми

```
#define N 6
int NewIndex(int x, int y)     //перерахування індексів
{   int j=0;
    for(int i=0; i<x; i++) j+= N-i;
```

```
    return j+y-x;
}
void Put(int vec [], int x, int y, int v)  // Запис у вектор (стиснення)
{  if (y >=x) vec[NewIndex(x, y)] = v;
}
int Get(int vec [], int x, int y) // читання з вектора
{  if (y >=x) return vec[NewIndex(x, y)];
   else return 0;
}
void RandArray(int a[N][N]) //формування початкового масиву
{  for(int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      if (j >=i) a[i][j] = rand()%50;
      else a[i][j] = 0;
}
void PrintArray(int a[N][N])
{    for(int i = 0; i < N; i++)
     {  for (int j = 0; j < N; j++)
           printf("%3i", a[i][j]);
         printf("\n");
     }
}
void main()
{  int vec[N*N/2+N/2];
   int array[N][N];
   RandArray(array);
   PrintArray(array);
   for(int i = 0; i < N; i++)
     for (int j = 0; j < N; j++)
         Put(vec, i, j, array[i][j]);          //стиснення масиву
   for(int i = 0; i < N*N/2+N/2; i++)          //видача результату стиснення
         printf("%3i", vec[i]);
    printf("\n\n");
    for(int i = 0; i < N; i++)
    { for (int j = 0; j < N; j++)    //читання зі стисненого подання
              printf("%3i", Get(vec,i,j));
      printf("\n");
     }
}
```

Результат роботи програми:



Зауваження: в наведеній програмі не виконано тестування доступу за часом до елементів масиву при його повному та економному(стислому) поданні у пам’яті.

Індивідуальні завдання

Розробити спосіб економного розміщення в пам'яті заданої розрідженої таблиці, де записані цілі числа. Розробити функції, що забезпечують доступ до елементів таблиці за номерами рядка і стовпця.

У програмі забезпечити запис і читання всіх елементів таблиці.

Визначити та порівняти час доступу до елементів таблиці при традиційному та економному поданні її в пам'яті. Зробити висновки.

Завдання обрати з табл. 5.1 згідно із своїм номером у журналі групи.

Таблиця 5.1 – Варіанти індивідуальних завдань

Номер з/п	Вміст розрідженої матриці
1	Нульові елементи розташовані в лівій половині матриці
2	Нульові елементи розташовані в правій половині матриці
3	Нульові елементи розташовані у верхній половині матриці
4	Нульові елементи розташовані в нижній половині матриці
5	Усі елементи непарних рядків – нульові
6	Усі елементи парних стовпців – нульові
7	Нульові елементи розташовані в шаховому порядку, починючи з 0-го елемента 0-го рядка
8	Нульові елементи розташовані в шаховому порядку, починаючи з 1-го елемента 0-го рядка
9	Нульові елементи розташовані на місцях з парними індексами рядків і стовпців
10	Нульові елементи розташовані на місцях з непарними індексами рядків і стовпців
11	Нульові елементи розташовані вище головної діагоналі на непарних рядках і нижче головної діагоналі – на парних
12	Нульові елементи розташовані у першій та третій третинах рядків матриці
13	Нульові елементи розташовані у лівій та правій третинах стовпців матриці
14	Нульові елементи розташовані у першій та третій третинах рядків та у лівій та правій третинах стовпців матриці
15	Нульові елементи розташовані на головній діагоналі та у верхній половині матриці вище діагоналі

16	Нульові елементи розташовані на головній діагоналі та у нижній половині матриці нижче діагоналі
17	Нульові елементи розташовані в лівій та правій чвертях матриці
18	Нульові елементи розташовані у верхній та нижній чвертях матриці
19	Нульові елементи розташовані на рядках, індекси яких є кратними трьом
20	Нульові елементи розташовані на стовпцях, індекси яких є кратними трьом
21	Нульові елементи розташовані у верхній третині рядків та середній третині стовпців
22	Нульові елементи розташовані у верхній третині рядків, першій та третій третині стовбців
23	Нульові елементи розташовані у верхньому і нижньому трикутниках, за умов розподілення матриці діагоналями на 4 трикутники
24	Нульові елементи розташовані у лівому та правому трикутниках, за умов розподілення матриці діагоналями на 4 трикутники

Контрольні запитання

1. Що таке «дескриптор» масиву, яке його призначення?
2. Як масиви подаються в пам’яті?
3. Як визначити обсяг пам’яті, що необхідний для запису масиву?
4. Як визначається адреса елемента масиву?
5. Чи залежить час доступу до елемента масиву від мірності цього масиву і чому?
6. Чим динамічні масиви відрізняються від статичних?
7. Чим характеризуються масиви змінної довжини, в чому їх відмінність від динамічних?
8. Що таке «гетерогенні масиви»?
9. На логічному рівні як подаються асоціативні масиви?
10. Які масиви вважаються симетричними, як їх подають на фізичному рівні?
11. За яких умов масиви вважаються розрідженими?

12. Яке призначення дерева відрізків і яке відношення вони мають до масивів?
13. Що таке V-список, у чому його переваги?
14. Коли необхідні паралельні масиви?
15. Яке призначення хеш-таблиць, як вони створюються?

ПРАКТИЧНА РОБОТА 6. ПОДАННЯ РЯДКІВ У ПАМ'ЯТІ

Мета: Отримати практичні навички та закріпити знання про можливі подання даних типу рядок та про операції над рядками.

Теми для попередньої роботи:

- подання рядків;
- операції над рядками;
- засоби обробки рядків у мовах програмування;
- текстові процесори.

Загальні відомості

Рядок – це лінійно впорядкована послідовність символів, які належать скінченній множині символів, що називається алфавітом. Рядки мають такі важливі властивості:

- їхня довжина, як правило, змінюється, а алфавіт фіксований;
- звертання до символів рядка йде з будь-якого кінця послідовності, тобто важлива впорядкованість цієї послідовності, а не її індексація; у зв'язку із цією властивістю рядки називають також ланцюжками;
- найчастіше метою доступу до рядка є не окремий його елемент (хоча це теж не виключається), а деякий ланцюжок символів у рядку.

Говорячи про рядки, частіше мають на увазі текстові рядки – рядки, які складаються із символів, що входять в алфавіт будь-якої обраної мови, цифр, розділових знаків та інших службових символів. Текстовий рядок є найбільш універсальною формою подання інформації. Залежно від конкретних завдань змінюваність рядків може варіюватися від повної її відсутності до практично необмежених можливостей зміни. Орієнтація на той або інший ступінь змінюваності рядків визначає її фізичне подання у пам'яті й особливості виконання операцій над ними. У більшості мов програмування рядки подаються саме як напівстатичні структури.

Операції над рядками. Базовими операціями над рядками є такі:

- визначення довжини рядка; присвоювання рядків;
- конкатенація (зчеплення) рядків;
- виділення підрядка;
- пошук входження підрядка.

Операція *визначення довжини* рядка має вигляд функції, яка повертає ціле число – поточне число символів у рядку.

Операція *присвоювання* має те саме значення, що і для інших типів даних.

Операція *порівняння рядків* проводиться за такими правилами: порівнюються перші символи двох рядків. Якщо символи не рівні, то рядок, що містить символ, місце якого в алфавіті ближче до початку, вважається меншим. Якщо символи рівні, порівнюються другі, треті і т.д. символи. При досягненні кінця одного з рядків, рядок меншої довжини вважається меншим. При рівності довжин рядків і попарній рівності всіх символів у них рядки вважаються рівними.

Результатом операції *зчеплення двох рядків* є рядок, довжина якого дорівнює сумарній довжині рядків-операндів, а значення відповідає значенню першого операнда, за яким безпосередньо слідує значення другого операнда. Операція зчеплення дає результат, довжина якого в загальному випадку більша за довжину операндів.

Як і у всіх операціях над рядками, що можуть збільшувати довжину рядка (присвоювання, зчеплення, складні операції), можливий випадок, коли довжина результату виявиться більшою, ніж відведений для нього обсяг пам'яті. Природно, ця проблема виникає тільки в тих мовах, де довжина рядка обмежується. Можливі три варіанти розв'язку цієї проблеми, обумовлені правилами мови або режимами компіляції:

– ніяк не контролювати таке перевищення; виникнення такої ситуації неминуче призводить до помилки, що важко локалізується при виконанні програми;

– завершувати програму аварійно з локалізацією і діагностикою помилки;

– обмежувати довжину результату відповідно до обсягу відведеної пам'яті.

Операція *виділення підрядка* виділяє з початкового рядка послідовність символів, починаючи із заданої позиції ***n***, заданої довжини ***l***. При реалізації операції виділення підрядка в функції обов'язково має бути визначено правило одержання результату для випадку, коли початкова позиція ***n*** задана такою, що та частина початкового рядка, яка лишилася, має довжину меншу ніж задана довжина ***l***, або навіть ***n*** перевищує довжину початкового рядка. Можливі варіанти такого правила:

– аварійне завершення програми з діагностикою помилки;

– формування результату меншої довжини, ніж задане, можливо навіть – порожнього рядка.

Операція *пошуку входження* знаходить місце першого входження підрядка-еталона у заданому рядку. Результатом операції може бути номер позиції у початковому рядку, з якого починається входження еталона або вказівник на початок входження. У випадку відсутності входження результатом операції має бути деяке спеціальне значення, наприклад, нульовий номер позиції або порожній вказівник.

На основі базових операцій можуть бути реалізовані і будь-які інші, навіть складні операції над рядками. Наприклад, операція видалення з рядка символів з номерами від ***n1*** до ***n2***, включно, може бути реалізована як послідовність таких кроків:

– виділення з заданого рядка підрядка, починаючи з позиції **0**, завдожки ***(n1-1***) символів;

– виділення з початкового рядка підрядка, починаючи з позиції ***(n2+1),*** завдожки, що дорівнює довжині заданого рядка ***-n2***;

– зчеплення підрядків, отриманих на попередніх кроках.

Втім, з метою підвищення ефективності деякі вторинні операції також можуть бути реалізовані як базові – за власними алгоритмами, з безпосереднім доступом до фізичної структури рядка.

Подання рядків у пам’яті залежить від того, наскільки мінливими є рядки в кожному конкретному завданні, і засоби такого подання змінюються від абсолютно статичного до динамічного. Розрізняють векторне подання рядків (вектор сталої довжини, вектор із лічильником, вектор з ознакою кінця, вектор із керованою довжиною), спискове подання (символьно-зв’язне або блочно-зв’язне з блоками фіксованої, змінної та керованої довжини). Списки можуть бути односпрямованими або двоспрямованими. Універсальні мови програмування здебільшого, забезпечують роботу з рядками змінної довжини, але максимальна довжина рядка має бути зазначена при його створенні.

Типове завдання

Реалізуати та перевірити виконання таких операцій над рядками:

– визначити позицію першого входження символу ***c*** в рядок ***s***. Якщо ***c*** буде знайдено, то повернути номер символу ***c*** у ***s***. В іншому випадку повернути -1;

– визначати і повернути довжину початкової частини рядка ***s2***, що складається з символів, які містяться в рядку ***s1***.

Текст програми

```
int CharInString(char s[], char c)   //Визначається входження символу в рядок
{         int i=0;
          for ( char *p=s; *p; p++,i++)
                  if(*p == c) return i;
          return -1;
 };
//пошук входження підрядка в рядок
 int SubstringInString(char s[], char sb[], int &l)
 {  l=0;
         int i=0, j=0;
         while(s[i] && sb [j] && s[i]!=sb[j])  i++;    //пошук першого збігу
першого символу підрядка
         if (s[i]==sb[j])
         {   while(s[i] && sb [j] && s[i]==sb[j])  j++, i++;     // визначення
довжини ланцюжка символів, що збіглися
             l = j;
         return i-l;
         }
         return -1;
 };

int  main (void)
 { setlocale(LC_ALL, "Rus");
   char *str = (char*)malloc(50*sizeof(char));
   char *substr = (char*)malloc(50*sizeof(char));
   int poz;
   char simbol;
   cout << " which string? :  ";
   gets(str);
   cout << " which substring? :  ";
   gets(substr );
   cout << " which character? :  ";
   cin >> simbol;
   poz=CharInString(str, simbol);
   if (poz>0) cout << poz << endl;
   else cout << "character error \n";
   int len;
   poz=SubstringInString(str, substr, len );
   if (poz>0) cout << poz <<" len ="<< len << endl;
   else cout << "substring in string error \n";
   delete str;
   delete substr;
   return 0;
 };
```

Результат роботи програми:

```
which string? :  this is a test example
which substring? :  est
which character? :  a
8
11 len =3
```

Індивідуальні завдання

Написати програму, в якій передбачити виконання вказаної операції над рядками за умови подання рядків у пам'яті двома способами. Порівняти подання рядків вказаними способами за такими показниками:

- обсягом використовуваної пам'яті;
- часом виконання функції.

Операцію та способи подання рядків вибрати з таблиці 6.1.

Таблиця 6.1 – Варіанти індивідуальних завдань

Варіант	1	2	3	4	5	6	7	8	9	10	11	12	13
Функція	1	2	3	4	5	6	7	8	9	10	11	12	13
Подання рядка	4 8	3 7	2 6	1 5	4 9	3 5	2 6	1 7	4 8	3 9	2 5	14 6	4 7

Варіант	14	15	16	17	18	19	20	21	22	23	24	25
Функція	1	2	3	4	5	6	7	8	9	10	11	12
Подання рядка	3 8	2 9	1 5	4 6	3 7	2 8	1 9	4 5	3 6	2 7	1 8	4 9

Спосіб подання рядка

1) Вектор сталої довжини.
2) Вектор змінної довжини з ознакою кінця рядка.
3) Вектор змінної довжини з лічильником.
4) Вектор з керованою довжиною рядка (з дескриптором).
5) Символьно-зв'язане подання односпрямованим списком.
6) Символьно-зв'язане подання двоспрямованим списком.
7) Блочно-зв'язне подання з фіксованою довжиною.
8) Блочно-зв'язне подання із змінною довжиною.

9) Блочно-зв’язне подання із керованою довжиною.

У завданнях 7 – 9 призначення блока обрати за своїм розсудом.

Функції

1) Визначити кількість символів у рядку *s*.
2) Виконати конкатенацію рядків *s1* та *s2*.
3) Замінити в рядку *s*, починаючи з позиції *n*, всі малі літери на великі.
4) Видалити з рядка *s* підрядок, починаючи з позиції *n* завдожки *k* символів.
5) Скопіювати з рядка *s*, починаючи з *m*-го символу, *n* символів.
6) Вставити в рядок *s* підрядок *s1*, починаючи з позиції *n*.
7) Переписати рядок *s* так, щоб символи в ньому були записані у зворотному порядку.
8) Знайти підрядок *s1* в рядку *s*.
9) Визначити кількість слів завдовжки *k* символів у рядку *s*.
10) Порівняти рядки *s1* та *s2*.
11) Видалити в рядку *s* початкові, хвостові та множинні пропуски.
12) Замінити усі символи *c1* у рядку *s* на символи *c2*.
13) Видалити копії слів у рядку *s* за умови, що однакові слова записані поряд одне з одним.

Контрольні запитання

1. Що таке «рядок», які його властивості?
2. Які базові операції визначено над рядками?
3. За яким алгоритмом виконується порівняння рядків?
4. Що таке «конкатенація» рядків?
5. Які відомі способи подання рядків у пам’яті?
6. Який алгоритм видалення частини рядка?
7. Навіщо створюється дескриптор рядка?
8. Які переваги та недоліки подання рядків масивом?
9. Які переваги та недоліки подання рядків односпрямованим списком?
10. Які переваги та недоліки подання рядків двоспрямованим списком?
11. Яке призначення блоків при блочно-зв’язному поданні рядків?

ПРАКТИЧНА РОБОТА 7. АЛГОРИТМИ ОБСЛУГОВУВАННЯ ЧЕРГ

Мета: набути практичного досвіду та закріпити знання про подання стека, дека, пріоритетної черги та дисципліни їх обслуговування.

Теми для попередньої роботи:

- масиви та списки;
- безпріоритетні та пріоритетні черги;
- дисципліни обслуговування черг.

Загальні відомості

Черга – це така структура даних, що зберігає елементи і забезпечує доступ до них тільки в певному порядку, визначеному пріоритетом. Для реалізації черг можуть бути використані масиви або лінійні списки.

Дек – це послідовний набір даних, у якому включення і виключення елементів може здійснюватися із кожного з двох кінців набору. Пріоритетом є час надходження запитів. Окремі випадки дека: черга LIFO або стек та черга FIFO.

У пріоритетних чергах пріоритет визначається одним або більшою кількістю параметрів. При роботі з пріоритетними чергами мова йде не про елементи, а про запити, що записують у чергу. Можлива організація черг із пріоритетним включенням елемента в чергу і з пріоритетним виключенням запиту з черги.

Операції над чергами: включення елемента в чергу, виключення елемента з черги для обслуговування згідно із пріоритетом, визначення розміру черги, очищення черги.

Типові завдання

Завдання 1. Реалізувати алгоритм Дейкстри побудови зворотного польського запису для заданого арифметичного виразу із застосуванням стека.

Пояснення до тексту програми. Вхідний рядок – *In*, вихідний – *Out*. Стек реалізований на односпрямованому списку. В стек записуються знаки операцій та дужки, що відкриваються. Читання зі стека виконується в трьох випадках:

1) пріоритет поточної операції у рядку *In* менший, ніж пріоритет операції, що розташована на вершині стека;

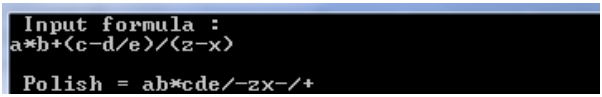
2) поточний символ у рядку *In* – дужка, що закривається;
3) вхідний рядок *In* закінчився.

Текст програми

```
struct Stack                          / Стек реалізовано на списку
{  char c;                            // Символ операції в стеку
   Stack *Next;
} ;
int Prior (char);
Stack* Push( Stack*,char);
Stack* Pop( Stack*,char*);


void main ()
{   Stack *pointSt = NULL;  // Стек операцій пустий, вказівник = 0
    char a, In[N], Out[N];  // Вхідна In та вихідний Out рядки
    int i = 0, j = 0;       // Поточні індекси для рядків
    puts(" Input formula : "); gets(In);
    while ( In[i] )         // Аналіз символів вхідного рядка In
// Якщо символ «)», виштовхуються зі стека у вихідний рядок -> всі операції
    {  if ( In[i] == ‘)’ )
       {   pointSt = Pop(pointSt, &a); // Зняття елемента зі стека
           while ( a!= ‘(‘ )              //  до дужки, яка відкривається
           {     Out[j++] = a;            // Зняття елемента зі стека
         pointSt = Pop(pointSt, &a);  // запис його в рядок Out.
           }
        }
        else // Якщо символ рядка In – літера, записують її в рядок Out
           if ( In[i] >= ‘a’ && In[i] <= ‘z’ ) Out[j++] = In[i];
           else  // Якщо символ – дужка, що відкривається, записують її в стек
              if ( In[i] == ‘(‘ ) pointSt = Push(pointSt,In[i]);
/* Якщо символ – знак операції, переписують зі стека в рядок Out всі операції з
більшим або рівним пріоритетом */
            else
                 if (In[i]==‘+’ || In[i]==‘-’ || In[i]==‘*’ || In[i]==‘/’)
     {while ( (pointSt != NULL) && (Prior (pointSt->c) >= Prior (In[i])))
                 { pointSt = Pop(pointSt,&a); // Зняття наступного символу
                    Out[j++] = a;                 // запис його в рядок Out
                 }
                 pointSt = Push(pointSt,In[i]); // Поточний символ – у стек
            }
            i++;
        } // Кінець циклу аналізу вхідного рядка
    // Якщо стек не пустий, переписують усі операції у вихідний рядок
        while ( pointSt )   // !=NULL)
        {        pointSt = Pop(pointSt,&a);
                 Out[j++] = a;
        }
        Out[j] = ‘\0’;

        printf("\n Polish = %s\n", Out); // Виведення результату на екран
}
//======= Функція реалізації пріоритету операцій =========
int Prior ( char a )
{   if( (a==‘*’)||(a==‘/’))return 3;
```

```
        else
                if( (a=='+')||(a=='-'))return 2;
                else
                        if (a=='(' ) return 1;
                        else return 0;
}
// ============== Додавання елемента в стек =============
Stack* Push( Stack *t, char s)
{   Stack *t1 = (Stack*) malloc(sizeof(Stack));
        t1 -> c = s;
        t1-> Next = t;
        return t1;
}
// ============== Вилучення елемента зі стека ===============
Stack* Pop( Stack *t, char *s )
{   Stack *t1 = t;
        *s = t -> c;
        t = t->Next;
        free(t1);
        return t;
}
```

Результат роботи програми:



Завдання 2. Розробити програму для роботи з пріоритетною чергою. Забезпечити обслуговування запитів із максимальним пріоритетом.

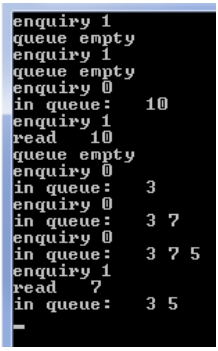
Пояснення до тексту програми. Чергу реалізовано на масиві. Постановка запитів у чергу виконується підряд, у кінець черги, зняття – за пріоритетом. При настроюванні програми команди та запити генерувати випадково. Після кожної команди – очікування натискання будь-якої клавіші.

Текст програми

```
int och[512];                       //черга на 512 запитів
int point=0;                        //вказівник кінець черги

int AddEl (int a)                   //додавання запиту у чергу
{        if (point<512) och[point++]=a;
         else
        {cout << "enough queue\n";   //черга переповнена
                return 0;
        }
    return 1;
}
void DelEl (void)           //читання запиту з черги
```

```
{   if (point>0)
    {    int max=och[0], nmax=0;     // пошук запиту з max параметром
         for(int i=1; i<point; i++)
                  if (max<och[i])
                           max=och[i], nmax=i;
         cout<<"read   "<<max<<endl;
         point--;                    //зсув черги
         for(int i=nmax; i<point; i++)
                  och[i]=och[i+1];
   }
}
void PrintOch (void)                 //видача вмісту черги
{        if (point==0)
                  cout<<"queue empty \n";
         else
         {        cout<<"in queue:   ";
                  for (int i=0;i<point;i++)
                           cout<<och[i]<<" ";
                  cout<<"\n";
         }
}
void main (void)
{        int op,f;
         do
         {        op=rand() % 2;              //0-запис запиту, 1- читання
                  cout<<"enquiry "<< op<< "\n";
                  if (op) DelEl();
                  else
                  {        f=rand() % 15;              //запит
                           if (! AddEl(f))  exit(1);
                  }
                  PrintOch ();
                  _getch();
      } while (1);
}
```

Фрагмент результатів роботи програми



Індивідуальні завдання

Розробити функції, що забезпечують запис та читання запитів із пріоритетної черги, стека або дека.

В кожному завданні для організації вказаної черги використати дві структури. Перевірити працездатність розроблених функцій. Послідовність виконання операцій запису та читання обирати випадково.

Порівняти результати роботи, зробити висновки.

1. Пріоритетна черга. Постановка запитів у чергу виконується за пріоритетом, зняття – підряд з молодших адрес (початку черги). Черга організована на масиві зі зсувом після кожного читання та на масиві зі зсувом після досягнення границі пам'яті, що виділена для черги. Пріоритет: ***min*** значення числового параметра; при збігу параметрів – *LIFO*.

2. Дек. Дек організований на масиві із циклічним заповненням та на двоспрямованому списку. Операції виконуються з обох кінців дека.

3. Черга *FIFO*. Черга організована на масиві зі зсувом після кожного читання та на масиві зі зсувом після досягнення границі пам'яті, що виділена для черги. Черга «зростає» від меншої адреси до більшої.

4. Пріоритетна черга. Постановка запитів у чергу виконується підряд у кінець черги, зняття – за пріоритетом. Черга організована на масиві та списку. Пріоритет: ***min*** значення числового параметра; при збігу параметрів – *LIFO*.

5. Стек. Стек організований на двоспрямованому списку та на масиві і «зростає» від меншої адреси пам'яті до більшої.

6. Дек. Дек організований на масиві із циклічним заповненням та на двоспрямованому списку. Операції виконуються з різних кінців дека.

7. Пріоритетна черга. Постановка запитів у чергу виконується за пріоритетом, зняття – підряд із молодших адрес (з початку черги). Черга організована на масиві із циклічним заповненням та зі зсувом. Пріоритет: ***max*** значення числового параметра; при збігу параметрів – *FIFO*.

8. Черга *FIFO*. Черга організована на масиві зі зсувом після кожного читання та на списку. Черга «зростає» від більшої адреси до меншої.

9. Пріоритетна черга. Постановка запитів у чергу виконується за пріоритетом, зняття – підряд зі старших адрес (кінця черги). Черга організована на масиві та на списку. Пріоритет: ***max*** значення числового параметра; при збігу параметрів – *FIFO*.

10. Дек. Дек організований на масиві із циклічним заповненням та зі зсувом. Операції виконуються з обох кінців дека.

11. Пріоритетна черга. Постановка запитів у чергу виконується за пріоритетом, зняття – підряд із молодших адрес (початку черги). Черга

організована на масиві із циклічним заповненням та списком. Пріоритет: ***max*** значення числового параметра; при збігу параметрів – FIFO.

12. Черга *FIFO*. Черга організована на списку та на масиві із циклічним заповненням. Черга «зростає» від більшої адреси пам'яті до меншої.

13. Дек. Дек організований на масиві із циклічним заповненням та зі зсувом. Операції виконуються з різних кінців дека.

14. Пріоритетна черга. Постановка запитів у чергу виконується за пріоритетом, зняття – підряд із молодших адрес (з початку черги). Черга організована на масиві зі зсувом після кожного читання та на масиві зі зсувом після досягнення границі пам'яті, що виділена для черги. Пріоритет: ***max*** значення числового параметра; при збігу параметрів – *FIFO*.

15. Стек. Стек організований на односпрямовоному списку та на масиві і «зростає» від більшої адреси пам'яті до меншої .

16. Пріоритетна черга. Постановка запитів у чергу виконується за пріоритетом, зняття – підряд із молодших адрес (з початку черги). Черга організована на масиві із циклічним заповненням та зі зсувом. Пріоритет: ***min*** значення числового параметра; при збігу параметрів – *LIFO*.

17. Пріоритетна черга. Постановка запитів у чергу виконується за пріоритетом, зняття – підряд зі старших адрес (кінця черги). Черга організована на масиві та на списку. Пріоритет: ***min*** значення числового параметра; при збігу параметрів – *LIFO*.

18. Черга *FIFO*. Черга організована на списку та на масиві із циклічним заповненням . Черга «зростає» від меншої адреси пам'яті до більшої .

19. Пріоритетна черга. Постановка запитів у чергу виконується за пріоритетом, зняття – підряд із молодших адрес (з початку черги). Черга організована на масиві із циклічним заповненням та списком. Пріоритет: ***min*** значення числового параметра; при збігу параметрів – *LIFO*.

20. Пріоритетна черга. Постановка запитів у чергу виконується підряд у кінець черги, зняття – за пріоритетом. Черга організована на масиві та списку. Пріоритет: ***max*** значення числового параметра; при збігу параметрів – *FIFO*.

21. Стек. Стек організований на односпрямовоному списку та на масиві і «зростає» від меншої адреси пам'яті до більшої.

Контрольні запитання

1. Що таке «черга», дайте визначення.
2. Що являє собою стек?
3. Які відомі види черг?
4. На основі яких структур даних може бути організований стек?
5. Які операції властиві черзі?
6. Які операції властиві стеку?
7. Який алгоритм читання запитів зі стека?
8. Які властивості притаманні черзі?
9. Який недолік простої черги? Який спосіб боротьби з ним?
10. Чим відрізняється циклічна черга від простої?
11. Які сфери застосування черг і стеків ?
12. Чим відрізняється стек на основі масиву від стека на основі зв'язного списку ?
13. Чим відрізняється пріоритетна черга з пріоритетним записом від черги з пріоритетним читанням?
14. Чим відрізняється стек на основі зв'язного списку від власне зв'язного списку?
15. Що являє собою дек?
16. Який алгоритм зняття запиту з пріоритетної черги, в яку запити ставляться підряд, як надходять?
17. Який алгоритм постановки запиту в пріоритетну чергу, в якій запити знімаються підряд?
18. Яку помилку допущено в наведеній функції запису елемента в стек?

```
int StackPush(int elem)
{  St[--top]=elem;  return 1;
}
```

19. Наведіть приклад функцій запису елемента в дек.
20. Наведіть приклад функцій зняття елемента з дека.

ПРАКТИЧНА РОБОТА 8. СПИСКИ

Мета: набути навичок та закріпити знання при виконанні операцій на мультисписках та нелінійних списках.

Теми для попередньої роботи:

- набори даних – списки: лінійні, кільцеві, мультисписки, нелінійні списки;
- операції на списках.

Загальні відомості

Список – це множина елементів, до яких можуть бути застосовані операції включення та виключення. У пам'яті список являє собою сукупність дескриптора й однакових за розміром і форматом записів, розміщених довільно в деякій ділянці пам'яті та зв'язаних один з одним у лінійно впорядкований ланцюжок за допомогою вказівників.

Розрізняють лінійні та нелінійні списки. Лінійні списки, у свою чергу, бувають кільцевими однозв'язними і двозв'язними.

Операції над зв'язними лінійними списками такі:

- перебір елементів списку – це послідовний доступ до всіх елементів списку від його початку до кінця або до знаходження шуканого елемента. У кільцевому списку закінчення перебору має відбуватися не за ознакою останнього елемента, а після досягнення елемента, з якого почався перебір;
- вставка елемента в будь-яке місце списку;
- видалення елемента з будь-якого місця списку;
- перестановка елементів у списку;
- копіювання частини списку. Операція копіювання припускає дублювання даних у пам'яті. При копіюванні вхідний список зберігається в пам'яті, і створюється новий список, але поля зв'язок у новому списку інші, оскільки елементи нового списку розташовані за іншими адресами у пам'яті;
- злиття двох списків. Операція злиття полягає у формуванні із двох списків одного. Ця операція аналогічна операції зчеплення рядків.

У всіх операціях надзвичайно важлива послідовність корекції вказників, яка забезпечує коректну зміну одних елементів списку і не змінює інші елементи. При неправильному порядку корекції вказників

можна втратити частину списку або, навіть, увесь список.

Нелінійним багаторівневим списком є список, елементами якого можуть бути теж списки, як на рис на рис. 8.1.

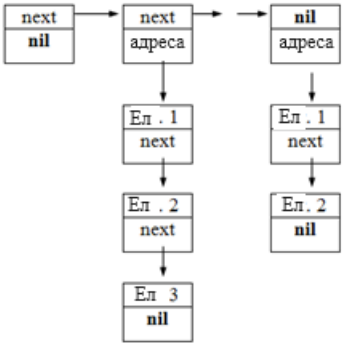


Рисунок 8.1 – Список списків

Елементи нелінійного списку у своєму складі мають більше одного вказівника. Нелінійні розгалужені списки описуються трьома характеристиками: порядком, глибиною та довжиною.

Порядок описує послідовність, у якій елементи з'являються усередині списку.

Глибина – це максимальний рівень, приписуваний елементам усередині списку або усередині будь-якого його підсписку. Глибина визначається вкладеністю підсписків усередині списку.

Довжина – це кількість елементів у першому рівні списку.

Основні операції над нелінійними списками:

- додавання вузла;
- видалення вузла;
- специфічна операція – обхід списку.

Двійкове дерево – це нелінійний список, у якому кожний елемент списку має поле даних і два поля посилань – left і right (рис. 8.2), а також характеризується такими властивостями:

- є один вузол, на який не посилається жоден інший вузол. Цей вузол називається "корінь" або "батьківський вузол";
- є вузли, які не посилаються на жодні інші вузли, – це листи.

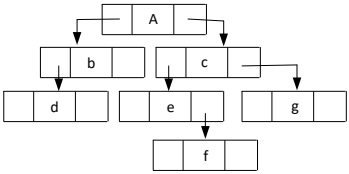


Рисунок 8.2 – Двійкове дерево

Інші вузли – вузли розгалуження. Корінь і вузли розгалуження посилаються на ліве і праве піддерева. Оскільки дерево – по суті рекурсивна структура, то й усі фунції роботи з деревами – рекурсивні.

Ортогональний список (або мультисписок) – це структура, кожний елемент якої входить більш ніж в один список одночасно і має відповідну до числа списків кількість полів вказівників. Реалізація кожного зі списків може бути виконана як одно- або двозв'язний нециклічний або циклічний список. Технологія обробки мультисписків нічим не відрізняється від обробки лінійних списків.

Типове завдання

Розробити програму для подання розрідженої матриці у вигляді мультисписку. Дані читати з файла. Значення елементів розрідженої матриці видати по рядках.

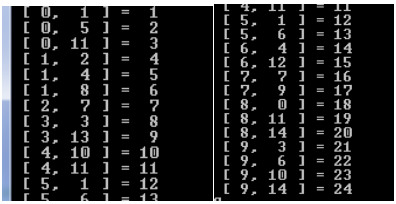
Текст програми

```
//Приклад обробки розрідженої матриці — процедура, що роздруковує значення
елементів матриці по рядках.
#include <stdio.h>
#include<fstream>
#include <malloc.h>
#define NROW  10   // кількість рядків
#define NCOL  15   // кількість стовпців

struct TMatrix              // опис типу вузла мультисписку
{   int  val;               // значення елемента
         int row, col;      // номер рядка, номер стовпця
         TMatrix *frow, *fcol;// атрибути зв'язку в списках зарядком і за стовпцем
};
void Read_File(TMatrix frow[], TMatrix fcol[])
{        TMatrix *temp;
         TMatrix *p;
         FILE *file;
         file = fopen("Text1.txt","r");
         while(!feof(file))
```

```
        {       temp = (TMatrix*)malloc(sizeof(TMatrix));
                fscanf(file, "%i%i%i", &temp->row, &temp->col, &temp->val);
                p = &frow[temp->row];
                while (p->fcol)    // переміщення до кінця списку по рядку
                p = p->fcol;       //перехід до елемета в наст. стовпці
                temp->fcol = p->fcol;
                 p->fcol = temp;
                 p = &fcol[temp->col];
                while (p->frow)    // переміщення до кінця списку по стовпцю
                         p = p->frow;  //перехід до елемента в наст. стовпці
                temp->frow = p->frow;
                 p->frow = temp;
        }
}

void Init(TMatrix frow[], TMatrix fcol[])
{       for (int i = 0; i < NROW; i++)
        {       frow[i].fcol = 0;
                frow[i].frow = 0;
                frow[i].row = 0;
        }
        for (int i = 0; i < NCOL; i++)
        {       fcol[i].fcol = 0;
                fcol[i].frow = 0;
                fcol[i].col = 0;
        }
}
void Print_Matrix(TMatrix frow[] )   //fr – масив указівників на списки рядків
{       TMatrix *p;        // p – допоміжний вказвник для проходження
        for (int i = 0; i < NROW; i++)       // перегляд рядків
        {   p = frow[i].fcol; //установка вказ. на перший елемент списку рядка
                while ( p )        //список рядка не пустий?
                {     printf(" [%2i,%3i ] = %2i\n",p->row, p->col, p->val);
                      p=p->fcol;   //перехід до наступного елемента в рядку
                }
        }
}
void Delete_Matrix(TMatrix frow[] ) //звільнення пам’яті
{    TMatrix *p, *pDel;     // p – допоміжний вказівник для проходження по рядку
        for (int i = 0; i < NROW; i++)       // перегляд рядків
        {   p = frow[i].fcol;
             while ( p )                     //список рядка не пустий?
             {   pDel = p;
                 p=p->fcol;         //перехід до наступного елемента у рядку
                 delete pDel;
             }
        }
}
int main()
{       TMatrix *pRow;     //масив указівників на перші вузли списків стовців
        pRow = (TMatrix*)malloc(NROW*sizeof(TMatrix));
        TMatrix *pCol;     //массив указателей на первые узлы списков столбцов
        pCol = (TMatrix*)malloc(NCOL*sizeof(TMatrix));
        Init(pRow, pCol);
        Read_File(pRow, pCol);
        Print_Matrix(pRow);
        Delete_Matrix(pRow);
        delete pRow ;
        delete pCol;
```

```
}
```

Результат роботи програми



Індивідуальні завдання

Для варіантів завдань 1 – 13 розробити програму, що створює список списків (нелінійний список). Передбачити такі функції:

– додавання елементів у список та підсписок (при додаванні елемента у головний список додається і відповідний підсписок);

– видалення елементів зі списку та підсписків (при видаленні елемента з головного списку видаляється і пов’язаний з ним підсписок);

– видача вмісту списків та підсписків у консоль;

– видалення списків.

Завдання обрати у табл. 8.1 згідно зі своїм номером у журналі групи.

Таблиця 8.1 – Варіанти індивідуальних завдань для створення нелінійних списків

З/П	Список	Підсписок
1	Прізвища авторів творів	Назви творів автора
2	Назви ВНЗ	Назви факультетів ВНЗ
3	Індекси навчальних груп	Прізвища студентів групи
4	Назви країн	Міста країни
5	Маршрути автобусів	Назви пунктів зупинок автобуса
6	Назви міст	Назви підприємств у місті
7	Назви магазинів	Перелік товарів для продажу

8	Номери доріг	Назви населених пунктів уздовж дороги
9	Будівельні фірми	Перелік зданих об’єктів
10	Відділи підприємств	Прізвища співробітників відділу
11	Назви спеціальностей	Назви ВНЗ з підготовки спеціалістів указаних спеціальностей
12	Пристрої	Назви комплектуючих вузлів
13	Олімпіади	За роками прізвища переможців

Для варіантів завдань 14 – 20 розробити програму, що створює мультисписок. Передбачити такі функції:

– додавання елементів у список та підсписки;

– видалення елементів зі головного списку та підсписків (при видаленні елемента з одного зі списків цей елемент не видаляється з пам’яті. І тільки тоді, коли елемент не входить до жодного зі списків, він видаляється з пам’яті);

– видача вмісту списків та підсписків у консоль;

– видалення списків.

Завдання для варіантів 14 – 20 обрати у табл. 8.2 згідно зі своїм номером у журналі групи. Перелік властивостей обрати за своїм розсудом.

Таблиця 8.2 – Варіанти індивідуальних завдань для створення мультисписків

N	Об’єкт	Списки
14	Студенти інституту	1) усі студенти інституту; 2) студенти-іноземці; 3) студенти першого року навчання; 4) студенти-магістри.
15	Книжковий фонд	1) усі книжки бібліотеки; 2) технічна література; 3) художня література; 4) книжки для дітей
16	Мешканці житлового будинка	1) усі мешканці будинку; 2) діти; 3) працюючі мешканці будинку; 4) пенсіонери.

17	Транспортні засоби	1) усі транспортні засоби; 2) автомобільний; 3) залізничний; 4) морський
18	Мови	1) усі мови програмування; 2) мови програмування високого рівня; 3) мови програмування низького рівня; 4) сучасні мови програмування
19	Товари	1) назви усіх товарів; 2) товари вітчизняного виробництва; 3) імпортні товари; 4) товари китайського виробництва
20	Спортсмени	1) прізвища усіх спортсменів; 2) фігуристи; 3) хокеїсти; 4) гімнасти

Контрольні запитання

1. Які відмінності списку від масиву?
2. Які бувають списки?
3. Які операції є характерними для лінійних списків?
4. Які операції є характерними для нелінійних списків?
5. Якими характеристиками описуються нелінійні розгалужені списки?
6. Які характерні риси має двійкове дерево?
7. Що таке мультисписок?
8. Чим відрізняється кільцевий список від простого лінійного списку?
9. Які особливості видалення елементів з мультисписку?
10. Які особливості видалення елементів зі списку списків (нелінійного списку)?
11. Чим визначається глибина нелінійного списку?
12. Чим визначається довжина нелінійного списку?
13. Що описує порядок нелінійного списку?
14. Які мови програмування призначені виключно для обробки списків?
15. Побудуйте нелінійний список для арифметичного виразу

$$(a+b)\cdot(c-d)/e$$

16. Побудуйте нелінійний список для арифметичного виразу

$$d + f / (a+b) \cdot c$$

17. Назвіть відомі методи контроля вільної динамічної пам’яті.
18. Назвіть алгоритми виділення вільної динамічної пам’яті.
19. Які відомі методи звільнення динамічної пам’яті?
20. Наведіть опис алгоритму «найпідходящий» для виділення за запитом динамічної пам’яті.
21. Наведіть опис алгоритму «перший підходящий» для виділення за запитом динамічної пам’яті.
22. Що таке «діри» в пам’яті та якими вони бувають?

ПРАКТИЧНА РОБОТА 9. АЛГОРИТМИ ПРОСТИХ ПОШУКІВ

Мета: набути навичок та закріпити знання при виконанні операцій пошуку.

Теми для попередньої роботи:

- набори даних: масиви, лінійні списки;
- алгоритми пошуку за числовомим ключом: лінійний, лінійний з бар'єром, двійковий, експоненційний, інтерполяційний;
- алгоритми пошуку зразка в тексті: прямий, КМП, БМ алгоритми.

Загальні відомості

Пошук полягає в тому, щоб у фіксованому наборі даних одного типу знайти заданий елемент – ключ пошуку. Серед простих алгоритмів пошуку відомі такі: лінійний, лінійний з бар'єром, двійковий, експоненційний, інтерполяційний.

При порівнянні різних алгоритмів важливо знати, як їх складність залежить від обсягу вхідних даних – це так звана складність алгоритму за часом. При визначенні складності алгоритмів пошуку за часом визначають, насамперед, кількість операцій порівнянь та присвоювань.

Алгоритм лінійного пошуку. Найпростішим методом пошуку елемента, що знаходиться в неупорядкованому наборі даних, за значенням його ключа є послідовний перегляд кожного елемента набору, що продовжується доти, доки не буде знайдений бажаний елемент, для якого $m[i]$ = key. Якщо переглянуто весь набір, але елемент не знайдений – виходить, шуканий ключ є відсутнім у наборі. Для послідовного пошуку в середньому потрібно $(N+1)/2$ порівнянь, отже, порядок алгоритму лінійний – $O(N)$.

Приклад 1. Функція лінійного пошуку

Текст функції

```
int LinearSearch(int arr[], int key)
{   unsigned i=0;
    while(i<N && arr[i]!=key) i++;
    if (i==N) return -1; else return i;
}
```

Оскільки пошук закінчується у випадку хибності умови, то умова виходу з циклу така:

```
(i==n) or (m[I]==key).
```

При цьому, якщо ***i = n*** – ключ не знайдений. Очевидно, що закінчення циклу гарантоване, оскільки на кожному кроці ***i*** збільшується. Отже, за скінченну кількість кроків досягне ***n,*** навіть якщо не було порівняння.

Алгоритм лінійного пошуку з бар'єром. На кожному кроці лінійного пошуку необхідно збільшувати індекс і обчислювати складний логічний вираз. А чи можна прискорити процес пошуку? Єдине рішення – спростити логічний вираз, сформулювавши простий вираз; але при цьому гарантувати, що збіг буде завжди.

Такий варіант реалізований у методі лінійного пошуку з бар'єром. У цьому методі в кінець масиву записується додатковий елемент із значенням ***key***. Він називається бар'єром, тому що обмежує перехід за межі масиву. Але тепер масив буде описаний як ***int m[0..n + 1],*** а функція пошуку показана в такому програмному прикладі:

Приклад 2. Функція лінійного пошуку з бар'єром

Текст функції

```
int LinearSearchWithBarrier(int arr[], int key)
{   unsigned i=0;
    arr[N]=key;
    while(arr[i]!=key) i++;
    if ( i == N) return  -1; else return i;
}
```

Умова виходу з циклу ***m[i] == key***; але якщо вихід з циклу виконується при ***i = n + 1*** – ключ не знайдений.

Алгоритм бінарного пошуку. Іншим, відносно простим методом доступу до елемента, є метод бінарного пошуку, що виконується у заздалегідь упорядкованій послідовності елементів. Такий пошук називають бінарним, двійковим, або пошуком розподілом навпіл. Записи в таблицю заносяться в лексико-графічному (символьні ключі) або числовому (числові ключі) зростаючому порядку. Для досягнення упорядкованості може бути використаний будь-який з методів сортування.

У цьому методі пошук окремого запису з визначеним значенням ключа нагадує пошук прізвища в телефонному довіднику. Спочатку приблизно визначається запис у середині таблиці й аналізується значення ключа цього запису. Якщо воно занадто велике, то аналізується значення ключа у середині першої половини таблиці. Зазначена процедура повторюється в цій половині доти, доки не буде знайдений необхідний запис. Якщо значення ключа занадто мале, випробується ключ, що відповідає запису в середині другої половини таблиці, і процедура повторюється в цій половині. Цей

процес продовжується доти, доки не буде знайдений необхідний ключ або не стане порожнім інтервал, у якому здійснюється пошук.

В алгоритмі бінарного пошуку вводяться дві змінні ***b, e,*** що відзначають ліву (молодшу) і праву (старшу) секції масиву, де ще може бути виявлений необхідний елемент.

Приклад 3. Функція бінарного пошуку

Текст функції

```
bool BinSearch (int *arr, int key, int &m)
{  int b=0, e=N;                     // початкові значення меж
   bool notFound=true;
   while(b <= e && notFound)         //поки інтервал не звузиться до 0
   {   m=(b+e)/2;           //середина інтервалу
       if (arr[m]< key) b=m+1; //пошук у правій частині масиву
       else  if (arr [m] > key) e=m-1; //інакше –  в лівій частині
             else notFound=false;      // пошук відбувся
    return notFound;
}
```

Взагалі ж вибір ***i*** може бути довільним. Але при пошуку середнього значення інтервалу пошуку виключається половина масиву. Для того щоб знайти потрібний запис у таблиці, у гіршому випадку потрібно $\boldsymbol{log_2(N)}$ порівнянь. Отже, порядок алгоритму бінарного пошуку логарифмічний – $\boldsymbol{O(log_2(N))}$**.** Це краще, ніж при лінійному пошуку.

Суттєво поліпшити ефективність алгоритму можна, спростивши умову виконання циклу. При цьому відмовляються від пошуку моменту збігу шуканого ключа у наборі. Це потребує більше порівнянь, але виграш в ефективності на кожному кроці перевищує ці втрати. Швидкий алгоритм оснований на тому, що для шуканого елемента масиву виконується така умова, що ліворуч усі ключі менші, а праворуч – більші або рівні. Пошук закінчується, якщо знаходиться перший елемент із заданим ключем.

Приклад 4. Покращений алгоритм бінарного пошуку

Текст функції

```
int BinSearchQuick (int *arr, int key)
{ int m, b=0, e=N;                   // початкові значення меж
  while (b <= e)           //поки інтервал не звузиться до 0
  {  m = (b + e) / 2;                //середина інтервалу
     if (arr[m] >= key) e=m-1;
     else b=m+1;
  }
  if (arr[b]==key) return b;
  else return -1;
} }
```

Очевидно, що умова виходу досяжна, тому що для середнього

значення справедливо ***b<= i < e*** отже, ***i – 1*** зменьшується, ***i + 1*** зростає. При ***b = e*** цикл закінчується. Але це не гарантія успішного пошуку. Необхідна додаткова перевірка після завершення циклу ***m[i] == key***.

Пошук у масиві називають пошуком у таблиці, якщо ключ сам є інтегрованим об'єктом, таким як масив чисел або символів. Часто зустрічається саме останній випадок, коли масиви символів називають рядками або словами. Пошук у таблиці вимагає «вкладених» пошуків, а саме: пошуку по рядках таблиці, а для кожного рядка послідовних порівнянь – між компонентами.

Типове завдання

Реалізувати алгоритм лінійного пошуку ключа в масиві цілих чисел.

Текст програми

```
#include <iostream>
#include <conio.h>
int linearSearch(int[],int,int);

int main(void)
{         setlocale(LC_CTYPE, "rus"); // для роботи з кирилицею
          const int arraySize=100;
          int a[arraySize],searchKey,element;
          for (int x=0;x<arraySize;x++)
                    a[x]=2*x;
          std::cout<<"Введіть ключ пошуку - ціле число:  ";
          std::cin>>searchKey;
          element=linearSearch(a,searchKey,arraySize);
          if (element!=-1)
                    std::cout<<"Знайдено значення в елементі "<<element<<std::endl;
          else
                    std::cout<<"Значення не знайдено"<<std::endl;
return 0;
}
 int linearSearch(int array[],int key,int sizeOfArray)
  {  for (int n=0; n<sizeOfArray;n++)
          if (array[n]==key)
             return n;
    return -1;
}
```

Результат роботи програми:

Введіть ключ пошуку – ціле число: 98.

Знайдено значення в елементі 49.

Для продовження натисніть будь яку клавішу .

Введіть ключ пошуку – ціле число: 17

Значення не знайдено

Для продовження натисніть будь яку клавішу . . .

Індивідуальні завдання

Розробити та налагодити програму, в якій реалізувати два алгоритми пошуку відповідно до завдання. Порівняти алгоритми за часом роботи.

На етапі тестування для кожного з алгоритмів (варіанти 4 – 15) визначити кількість порівнянь у наборі даних з різною кількістю елементів (20, 100, 1000, 10000) визначити час пошуку, заповнити таблицю за формою табл. 9.1, побудувати графіки, зробити висновки. При роботі з текстом (варіанти 1 – 3) змінювати довжину текста та зразка.

УВАГА! У завданнях 4 – 15 передбачена робота з цілими числами; наступні 12 завдань (номери 16 – 27) вимагають роботу з дійсними числами.

1) Прямий та КМП пошук зразка у тексті.
2) КМП та БМ пошук зразка у тексті.
3) Прямий та БМ пошук зразка у тексті.
4) Двійковий та лінійний пошуки у масиві.
5) Двійковий та лінійний пошуки у лінійному списку.
6) Лінійний з бар’єром та двійковий пошуки у масиві.
7) Лінійний з бар’єром та двійковий пошуки у лінійному списку.
8) Лінійний та лінійний з бар’єром пошуки у масиві.
9) Лінійний та лінійний з бар’єром пошуки у лінійному списку.
10) Експоненційний та інтерполяційний пошуки у масиві.
11) Лінійний пошук у масиві та лінійному списку.
12) Лінійний пошук з бар’єром у масиві та лінійному списку.
13) Двійковий пошук у масиві та лінійному списку.
14) Двійковий та інтерполяційний пошуки у масиві.
15) Двійковий та експоненційний пошуки у масиві.
16) Лінійний з бар’єром та експоненційний пошуки у масиві.
17) Лінійний та експоненційний пошуки у масиві

Таблиця 9.1 – Результати тестування алгоритмів пошуку

Кількість елементів	20	100	1000	10000
Кількість порівнянь				
Час пошуку				

Контрольні запитання

1. Що визначає складність алгоритму?
2. Яка умова має виконуватися при пошуку ключа цілого типу?
3. Яка умова має виконуватися при пошуку ключа дійсного типу?
4. В алгоритмі лінійного пошуку з бар'єром що є бар'єром?
5. Перерахуйте усі відомі прості алгоритми пошуку за числовим ключем за зменшенням їх середнього часу пошуку.
6. Які обмеження накладаються на набір даних при лінійному пошуку з бар'єром та без нього?
7. Поясніть, як виконується двійковий пошук?
8. Реалізацію якого алгоритму наведено в такому фрагменті програмного коду?

```
{   while(m[i]!= key && i<N) i++;
    if( m[i]== key) return i; else return -1;
}
```

9. Які обмеження накладаються на дані при бінарному пошуку?
10. Реалізацію якого алгоритму наведено в такому фрагменті програмного коду?

```
{   while(m[i]!= key) i++;
    if( i!= N) return i;  else return -1;
}
```

11. Накресліть якісний графік залежності часу пошуку від кількості елементів у наборі даних для лінійного, лінійного з бар'єром, двійкового, інтерполяційного та екпоненційного алгоритмів пошуку. Дайте пояснення.
12. Поясніть, як виконується експоненційний пошук, у чому його відмінність від двійкового?
13. Поясніть алгоритм роботи БМ-пошуку зразка в тексті.
14. Поясніть алгоритм роботи КМП-пошуку зразка в тексті.
15. В чому суттєва відмінність алгоритму прямого пошуку зразка в тексті від алгоритмів КМП та БМ пошуків?

ПРАКТИЧНА РОБОТА 10. АЛГОРИТМИ ПОШУКУ З ВИКОРИСТАННЯМ ТАБЛИЦЬ

Мета: закріпити знання про алгоритми пошуку, що потребують додаткової пам'яті; набути навичок виконання операцій пошуку із використанням таблиць прямого доступу, довідників та хешованих таблиць.

Теми для попередньої роботи:

- масиви та списки;
- файли;
- прості алгоритми пошуку;
- алгоритми пошуку із застосуванням таблиць прямого доступу;
- поняття про хеш-таблиці, функції хешування, колізії;
- алгоритми розв'язання колізій.

Загальні відомості

Таблиця прямого доступу – найпростіша таблиця, що забезпечує ідеально швидкий пошук. Ключ пошуку є адресою запису в таблиці і при пошуку за цією адресою вибирається запис. Якщо обраний запис порожній, то запису з таким ключем взагалі немає в таблиці.

Хешована таблиця – це така, для якої ключ пошуку перетворюється на адресу її запису за допомогою функції хешування. Існує ймовірність того, що різні ключі можуть відображатися в одну й ту саму адресу запису. Це називається колізією чи переповненням, а такі ключі називаються синонімами. Колізії (друга назва – конфлікт) – основна проблема для хешованих таблиць. Вдало підібрана функція хешування може мінімізувати кількість колізій, але не може гарантувати їхньої повної відсутності. Відомо багато методів вирішення проблеми колізій у хешованих таблицях, які створюються на базі тільки масивів, масивів і списків або тільки списків.

Алгоритми розв'язання колізій. Повторне хешування, відоме також за назвою відкритої таблиці, передбачає таке: якщо при спробі запису в таблицю виявляється, що необхідне місце в таблиці вже зайняте, то значення записується в ту саму таблицю на будь-яке інше місце. Інше місце визначається за допомогою вторинної функції хешування $H2$, аргументом якої в загальному випадку може бути і вихідне значення ключа, і результат попереднього хешування: $r = H2(k, r')$,

де r' – адреса, отримана при попередньому застосуванні функції хешування.

Якщо отримана у результаті застосування функції *H*2 адреса також виявляється зайнятою, функція ***H2*** застосовується повторно – доти, доки не буде знайдене вільне місце. Найпростішою функцією вторинного хешування є функція: $r = r' + 1$.

Цю функцію іноді називають функцією лінійного випробування. Фактично при застосуванні лінійного випробування, якщо «законне» місце запису (тобто слот, розташований за адресою, одержуваною з первинної функції хешування) уже зайнято, то запис займає перше вільне місце за «законним» (таблиця при цьому розглядається як кільце).

Вибірка елемента за ключем здійснюється аналогічно:

– адреса запису обчислюється за первинною функцією хешування і ключем запису;

– читається запис, що розташований за отриманою адресою, порівнюється із записом – ключем пошуку;

– якщо запис не порожній, а ключі не збігаються, то продовжується пошук із застосуванням вторинної функції хешування.

Пошук закінчується, коли знайдений запис збігається із шуканим ключем (успішне завершення), або перебрана вся таблиця (неуспішне завершення).

Пакетування. Сутність методу пакетування полягає в тому, що записи таблиці поєднуються в пакети фіксованого невеликого розміру. Функція хешування дає на виході не адресу запису, а адресу пакета. Після знаходження пакета, у пакеті виконується лінійний пошук за ключем. Пакетування дозволяє згладити порушення рівномірності розподілу ключів у просторі пакетів і, отже, зменшити кількість колізій, але не може гарантовано ним запобігти. Пакети також можуть переповнюватися. Тому пакетування застосовується як доповнення до більш радикальних методів – до методу повторного хешування чи до методів, описаних нижче.

Хеш-таблиці з загальною ділянкою переповнень. У випадку загальної ділянки переповнень для хеш-таблиці виділяються дві ділянки пам’яті: основна ділянка і ділянка переповнень. Функція хешування на виході дає адресу запису (або пакета) в основній ділянці. При вставленні запису, якщо його «законне» місце в основній ділянці вже зайнято, запис заноситься на перше вільне місце в ділянці переповнення.

При пошуку, якщо «законне» місце в основній ділянці зайнято записом з іншим ключем, виконується лінійний пошук у ділянці переповнення. Загальна ділянка переповнень вимагає більше пам’яті, ніж відкриті таблиці:

якщо розмір відкритої таблиці може не перевищувати розміру фактичної множини записів, то тут ще потрібна додаткова пам'ять для переповнень. Однак ефективність доступу до таблиці з ділянкою переповнення вища, ніж до таблиці з повторним хешуванням. Якщо в таблиці з повторним хешуванням при невдалій першій спробі необхідно продовжувати пошук у всій таблиці, то в таблиці з ділянкою переповнення продовження пошуку обмежується тільки ділянкою переповнення, розмір якої значно менший від розміру основної таблиці.

Хеш-таблиці з розподіленими ланцюжками переповнень. Природним є бажання обмежити пошук за умови наявності колізій лише множиною тих значень ключів, що претендують на це місце в основній таблиці. Ця ідея реалізується в таблицях з ланцюжками переповнення. У структуру кожного запису додається ще одне поле – вказівник на наступний запис. Через ці вказівники записи з ключами-синонімами зв'язуються в лінійний список, початок якого знаходиться в основній таблиці, а продовження – поза нею. При вставленні запису в таблицю за функцією хешування обчислюється адреса запису (або пакета) в основній таблиці. Якщо це місце в основній таблиці вільне, то запис заноситься в основну таблицю. Якщо ж місце в основній таблиці зайняте, то запис розміщується поза нею. Пам'ять для такого запису з ключем-синонімом може виділятися або динамічно для кожного нового запису, або для синоніма призначається елемент із заздалегідь виділеної ділянки переповнення. Після розміщення запису-синоніма поле вказівника із запису основної таблиці переноситься в поле вказівника синоніма, а на його місце в записі основної таблиці записується вказівник на тільки що розміщений синонім.

Хоча в таблицях з ланцюжками переповнень і збільшується розмір кожного запису та трохи ускладнюється обробка за рахунок обробки вказівників, звуження ділянки пошуку дає значний виграш в ефективності.

При будь-якому методі побудови хешованих таблиць виникає проблема видалення елемента із основної ділянки. Запис, що видаляється, має бути знайдений, насамперед, у таблиці. Якщо запис знайдений вторинним хешуванням (відкрита таблиця) або в ділянці переповнення (таблиця із загальною областю переповнення), то запис, що видаляється, достатньо позначити як порожній. Якщо запис знайдений у ланцюжку (таблиця із ланцюжками переповнень), то необхідно також скорегувати вказівник попереднього елемента в ланцюжку.

Якщо запис, що видаляється, знаходиться на своєму «законному» місці,

то, якщо позначити його як порожній, тоді неможливо виконати пошук його синонімів, що, можливо, знаходяться в таблиці. Одним зі способів вирішення цієї проблеми може бути позначка запису спеціальною ознакою «вилучена». Цей спосіб часто застосовується в таблицях із повторним хешуванням і з загальною ділянкою переповнень, але він не забезпечує ані звільнення пам'яті, ані прискорення пошуку при зменшенні кількості елементів у таблиці.

Інший спосіб – знайти будь-який синонім запису, що видаляється, і перенести його на «законне» місце. Цей спосіб легко реалізується в таблицях із ланцюжками, але потребує значних витрат у таблицях з іншою структурою, тому що необхідний пошук у всій відкритій таблиці чи у всій ділянці переповнення із обчисленням функції хешування для кожного елемента, що перевіряється.

Типове завдання

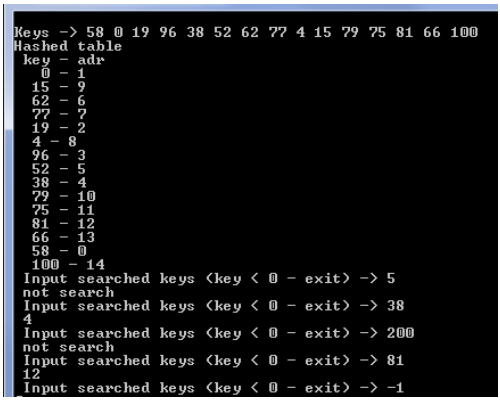
Завдання. Реалізувати алгоритм пошуку ключа в масиві цілих чисел із використанням хеш-таблиці. Функція хешування – ділення за модулем. Алгоритм розв'язання колізій – відкрите хешування з лінійним випробуванням.

Текст програми

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

void Init(void);
int Insert(int key, int adr);
int Search(int key);

#define N 15                //кількість записів у таблиці
#define EMPTY -1

struct ElHashTabl
{         int key;
          int adr;
};
ElHashTabl hashTabl[N];     //хеш-таблиця
int keys[N]={58,0,19,96,38,52,62,77,4,15,79,75,81,66,100};

void main(void)
{    int i, key, res;
     Init();
     cout <<"\nKeys -> ";
     for (i=0;i<N;i++)
               cout << keys[i] <<" ";
     for (i=0;i<N;i++)
               Insert(keys[i], i);
     cout <<"\nHashed table \n key - adr \n ";
```

```
     for (i=0;i<N;i++)
             cout <<"  " <<hashTabl[i].key <<" - " <<hashTabl[i].adr << "\n" ;
          cout <<" Input searched keys (key < 0 - exit) -> ";
          cin >> key;
     while (key >=0)
     {   res=Search(key);
             if (res==EMPTY)           cout << " not search \n";
             else cout << " " <<res <<"\n" ;
        cout <<" Input searched keys (key < 0 - exit) -> ";
             cin >> key;
          }
}

int Hash(int key)           //функція хешування
{     return (key%N);
}
void Init(void)             //ініціалізація хеш-таблиці
{     for(int i=0;i<N;i++)
          hashTabl[i].adr=EMPTY;
}
int Insert(int key, int adr)         //додавання ключа у хеш-таблицю
{    int addr,a1;
     addr=Hash(key);
     if (hashTabl[addr].adr!=EMPTY)
     {   a1=addr;
         do
         {    addr=Hash(addr+1);
         }while(!((addr==a1)||(hashTabl[addr].adr==EMPTY)));
         if (hashTabl[addr].adr != EMPTY)
              return 0;
     }
     hashTabl[addr].key=key;
     hashTabl[addr].adr=adr;
     return 1;
}
int Search(int key)                  //функція пошуку ключа
{    int addr, a1;
     addr=Hash(key);
     if (hashTabl[addr].adr==EMPTY)
         return EMPTY;      //місце вільне – ключа немає в таблиці
     else
         if (hashTabl[addr].key==key)
                  return hashTabl[addr].adr;  //ключ знайдений на своєму місці
              else                            //місце зайняте іншим ключем
              {   a1=Hash(addr+1);
         //Пошук, поки не знайдений ключ чи не зроблене повне обертання
                  while((hashTabl[a1].key != key)&&(a1!=addr))
                       a1=Hash(a1+1);
                       if (hashTabl[a1].key != key)
                           return EMPTY;
                       else
                           return hashTabl[a1].adr;
              }
}
```

Фрагмент результатів роботи програми



Індивідуальні завдання

Початкові дані містяться у текстовому файлі. Прочитати файл, створити таблицю прямого доступу або хеш-таблицю відповідно до завдання з табл. 10.1. Перевірити працездатність створених таблиць на прикладі операцій пошуку.

Порівняти час пошуку із використанням створених таблиць та простих алгоритмів пошуку з лабораторної роботи 4. Для кожного з алгоритмів визначити кількість порівнянь у наборі даних з різною кількістю елементів (20, 1000, 5000, 10000, 50000), визначити час пошуку, заповнити таблицю за формою табл. 10.2, побудувати графіки, зробити висновки.

Таблиця 10.1 – Початкові дані до лабораторної роботи

№	Вміст початкових даних	Таблиця	Хеш-функція	Ключ пошуку
1	2	3	4	5
1	Номер залікової книжки, прізвище студента, середній бал	Хеш-таблиця з повторним хешуванням	Ділення за модулем; повторне хешування: Адр = Адр + 1	Середній бал
2	Номер читацького білета, прізвище	Хеш-таблиця з пакетуванням	Функція згортки та ділення за	Прізвище читача

	читача, кількість книжок		модулем	
3	Модель процесора, тип материнської плати, об’єм вінчестера, розмір ОЗП	Хеш-таблиця із спільним простором переповнень	Ділення за модулем; повторне хешування: Адр = Адр + 1	Об’єм вінчестера/ розмір ОЗП
4	Код, тип двигуна, потужність, кількість обертів	Таблиця прямого доступу		Код
5	Мережева адреса станції, операційна система, мережевий протокол	Хеш-таблиця з розподіленими ланцюжками переповненнь	Функція перетворення системи числення та ділення за модулем	Мережева адреса станції
6	Код, тип літака, швидкість, кількість пасажирів	Хеш-таблиця із спільним простором переповнень	Функція середини квадрата	Код
7	Номер облікової картки, прізвище людини, вік, домашня адреса	Хеш-таблиця з розподіленими ланцюжками переповнень	Функція перетворення системи обчислення та ділення по модулю	Номер облікової картки
8	Код виробу, назва, ціна, кількість	Таблиця прямого доступу		Код виробу
9	Ідентифікаційний код, ім’я користу-вача мережі, E-mail адреса	Хеш-таблиця з розподіленими ланцюжками переповнень	Функція середини квадрата та ділення за модулем	Ідентифіка ційний код
10	Код міста, назва міста, площа, кіль-кість мешканців	Хеш-таблиця з повторним хешуванням	Функція згортки та ділення за модулем	Код міста
11	Код країни, назва країни, столиця	Хеш-таблиця з повторним хешуванням	Функція середини квадрата	Код країни

12	Табельний номер, прізвище працівника, цех або відділ	Хеш-таблиця з розподіленими ланцюжками переповнень	Функція згортки та ділення за модулем	Табельний номер працівника
13	Номер телефону, прізвище власника, адреса, час розмови	Хеш-таблиця з розподіленими ланцюжками переповнень	Функція середини квадрата	Номер телефону
14	Назва файлу, розмір, дата створення, дата останньої модифікації	Хеш-таблиця з пакетуванням та повторним хешуванням	Функція згортки та ділення за модулем. Повторне хешування	Назва файла
15	Шифр книги, прізвище автора, кількість примірників	Хеш-таблиця із спільним простором переповнень	Функція згортки та ділення за модулем	Шифр книги
16	Назва ВНЗ, місто, кількість факультетів, кількість студентів	Хеш-таблиця з розподіленими ланцюжками переповнень	Функція згортки та ділення за модулем	Назва ВНЗ

Таблиця 10.2 – Результати тестування алгоритмів пошуку із використанням таблиць

Кількість елементів	20	1000	5000	10000	50000
Кількість порівнянь					
Час пошуку					

Контрольні запитання

1. Що записується в хеш-таблицю?
2. Чим визначається індекс запису в хеш-таблиці?
3. Які основні проблеми хешування і в чому вони полягають?
4. Скільки операцій порівняння виконується при пошуку за ключем із застосуванням таблиць прямого доступу?
5. Які ключі називають синонімами?

6. Які недоліки алгоритму пошуку за ключем з використанням таблиць прямого доступу?

7. Яке призначення хеш-функції?

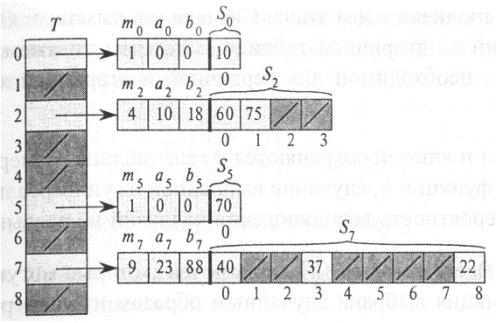
8. Назвіть базові функції хешування.

9. Як створюється хеш-таблиця при реалізації методу відкритої адресації? Що зберігається в елементах такої таблиці?

10. Як створюється таблиця прямого доступу? Що зберігається в її елементах?

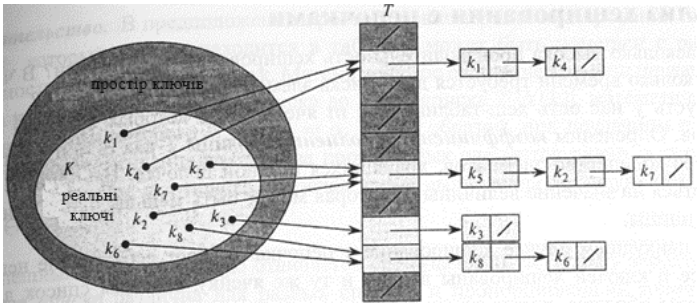
11. Назвіть алгоритми розв'язання колізій при відкритій адресації.

12. Яке хешування зображено на цьому рисунку?



13. В чому суть методу розв'язання колізій при використанні розподілених ланцюжків переповнень?

14. Який метод хешування зображений на рисунку?



15. Чому пошук із використанням таблиць прямого доступу широко не впроваджується?

ПРАКТИЧНА РОБОТА 11. АЛГОРИТМИ СОРТУВАННЯ ВИБОРОМ ТА ВКЛЮЧЕННЯМ

Мета: закріпити теоретичні знання та набути практичного досвіду впорядкування набору статичних та динамічних структур даних.

Теми для попередньої роботи:

- масиви та списки;
- класифікація алгоритмів сортування;
- алгоритми сортування вибором (вибіркою) та включенням.

Загальні відомості

Завданням сортування є перетворення вхідної послідовності на послідовність, що містить ті самі записи, але за збільшенням (або зменшенням) значень ключа. Метод сортування називається стійким, якщо при його застосуванні не змінюється відносне положення записів з однаковими значеннями ключа.

Алгоритми сортування характеризуються порядком (ступеневі – $O(N^a)$, лінійні – $O(N)$, логарифмічні – $O(N\log(N))$, вимогами до ресурсу пам'яті, чутливістю до упорядкованості вхідного набору даних, складністю.

При класифікації алгоритмів сортування за логікою роботи виділяють чотири групи алгоритмів: вибіркою, включенням, розподілом, злиттям.

Сортування вибіркою. Опис загального алгоритму роботи такий: із вхідної множини вибирається наступний за критерієм упорядкованості елемент і включається у вихідну множину на наступне за номером місце.

До цієї групи належать такі алгоритми: сортування простою вибіркою, обмінне сортування простою вибіркою, з пошуком максимального та мінімального елементів, бульбашкове сортування класичне, модифіковане, з ознакою, із запам'ятовуванням місця останньої перестановки, Шейкер-сортування, сортування Шелла. Всі вони характеризуються порядком $O(N^a)$.

Сортування включенням. Опис загального алгоритму роботи: із вхідної множини вибирається наступний за номером елемент і включається у вихідну множину на те (попередньо звільнене) місце, яке він повинен займати відповідно до критерію упорядкованості ключів.

До цієї групи належать такі алгоритми: сортування простими включеннями, обмінне сортування простими включеннями, бульбашкове сортування включеннями, сортування включеннями з бар'єром, методом

двійкового включення. Вказані алгоритми характеризуються порядком $O(N^a)$.

До сортування включенням належать також такі більш складні алгоритми: турнірне, пірамідальне та сортування частково упорядкованим деревом. Ці алгоритми характеризуються порядком O(Nlog(N)). Їх частково можна віднести до групи сортувань розподілом, саме тому вони будуть розглянуті в наступній лабораторній роботі.

Типове завдання

Завдання. Розробити програму сортування масиву цілих чисел згідно із алгоритмами Шейкера та включень із бар'єром.

Текст програми

```
#include<iostream>
#include<conio.h>
#define N 20                //кількість елементів у масиві

void Create(int []);
void CopyArray(const int [], int []);
void OutConsole(int []);
void SortShaker(int []);
void BarierInsertionSort (int []);

void main()
{   setlocale(LC_CTYPE, "rus");      // для роботи з кирилицею
    int array[N], copyArray[N];
    Create(array);
    std::cout << "початковий масив \n";
    OutConsole(array);
    CopyArray (array, copyArray);
    SortShaker(array);
    std::cout << "Відсортованний масив \n";
    OutConsole(array);
         CopyArray (copyArray, array);
         std::cout << "Початковий масив \n";
         OutConsole(array);
         BarierInsertionSort (array);
         std::cout << "Відсортований масив \n";
    OutConsole(array);
}
void Create(int a[])
{   for(int i = 0; i < N; i++)
        a[i]=rand() % 100;
}
void CopyArray(const int a[], int copy_a[])
{    for(int i = 0; i < N; i++)
         copy_a[i] = a[i];
}
void OutConsole(int a[])
{   for(int i = 0; i < N; i++)
         std::cout << "  " << a[i];
```

```
    std::cout << "\n";
}
void SortShaker(int a[])
{   int temp;
    int key=0; //key–ключ, дорівнює 1, якщо не було перестановок(масив
відсортований)
    int i, j;
    for (i = 1; i < N-1; i++)
    {   if (key) break;
        key=1;
        for (j = i; j < N-i; j++)
          if (a[j-1] > a[j])
          {   key=0;
              temp=a[j-1]; a[j-1]=a[j]; a[j]=temp;
          }
       for (; i; i--)
         if (a[j] <a [j-1])
         {  key=0;
            temp=a[j]; a[j]=a[j-1]; a[j-1]=temp;
         }
   }
}
void BarierInsertionSort (int a[])
{  int  i,j,k,t;
   for( i=1; i< N; i++)  // перебір вхідного масиву - [i..N], вих. набір - [1..i]
   {  a[0]=a[i];   //записується бар'єр a[0]
      t=a[i];      // запам'ятовується значення нового ел-та
          j=i-1;
       while (a[j]>t) //пошук місця для ел. у вих. наборі зі зсувом
          { a[j+1]=a[j];  // всі ел-ти, більші від нового зсуваються
              j--;    // цикл від кінця до початку вихідної множини
          }
          a[j+1]=t;       // новий ел-т ставиться на своє місце
         }
}
```

Результат роботи програми

```
початковий масив
  83  86  77  15  93  35  86  92  49  21  62  27  90  59  63  26  40  26  72  36
Відсортованний масив
  15  21  26  26  27  35  36  40  49  59  62  63  72  77  83  86  86  90  92  93
Початковий масив
  83  86  77  15  93  35  86  92  49  21  62  27  90  59  63  26  40  26  72  36
Відсортований масив
  36  15  21  26  26  27  35  36  40  49  59  62  63  72  77  86  86  90  92  93
```

Індивідуальні завдання

Написати програму, що реалізує сортування статичного та/або динамічного набору даних заданим способом згідно з даними табл. 11.1.

Визначити кількість порівнянь та обмінів для наборів даних, що містять різну кількість елементів (20, 1000, 5000, 10000, 50000).

Оцінити час сортування. Дослідити вплив початкової впорядкованості набору даних (відсортований, відсортований у зворотному порядку,

випадковий).

Всі отримані дані записати в таблицю 11.2. Зробити висновки.

Таблиця 11.1 – Варіанти індивідуальних завдань

N	Номер алгоритму	N	Номер алгоритму	N	Номер алгоритму	N	Номер алгоритму
1	13	7	10	13	7	19	4
2	1, 10	8	5, 9	14	8,1	20	11, 13
3	12	9	9	15	6	21	3
4	3, 7	10	6, 2	16	9, 13	22	10, 6
5	11	11	8	17	5	23	2
6	4, 8	12	7, 4	18	2, 3	24	4, 10

де N – номер індивідуального завдання.

Примітка:

1) Для непарного N реалізувати вказаний алгоритм сортування на статичному та динамічному наборах даних.

2) Для парного N реалізувати два алгоритми сортування на одному з наборів даних (статичному або динамічному).

3) Алгоритм сортування за номером обрати з наступного переліку.

Алгоритми сортувань

1. Сортування простою вибіркою.
2. Обмінне сортування вибіркою.
3. Сортування вибіркою максимального та мінімального елементів.
4. Бульбашкове сортування з ознакою.
5. Модифіковане бульбашкове сортування .
6. Бульбашкове сортування з запам’ятовуванням місця останньої перестановки.
7. Шейкерне сортування .
8. Сортування Шелла.
9. Гномьче сортування .
10. Сортування простими включеннями.
11. Бульбашкове сортування включенням.
12. Сортування включенням із бар’єром.
13. Сортування методом двійкового включення.

Таблиця 11.2 – Форма порівняльної таблиці ефективності алгоритму

Відсортований набір даних	20	1000	5000	10000	50000
Кількість порівнянь					
Кількість пересилань					
Час сортування					

Примітка: таблицю заповнити також для *випадкового набору даних* та *відсортованого в зворотному порядку*

Контрольні запитання

1. Назвіть усі алгоритми сортування порядку $O(n^2)$, O(n), O(log(n)).
2. Назвіть усі алгоритми сортувань, що належать до групи бульбашкових. Чим характеризуються ці алгоритми?
3. Вкажіть усі можливі модифікації алгоритму сортування Шейкера. Напишіть одну з вказаних процедур Шейкер-сортування.
4. В алгоритмі сортування Шелла шаг *d* визначається як *N % 2*. А чи можна визначати *d* інакше, чому запропоновано саме такий алгоритм?
5. Чим пояснюється наявність великої кількості алгоритмів сортування?
6. Назвіть відомі вам алгоритми сортувань, час виконання яких не залежить від ступеня впорядкованості вхідного масиву.
7. За логікою роботи на які групи поділяють усі алгоритми сортувань?
8. Назвіть алгоритми сортування, чутливі до ступеня впорядкованості вхіного набору даних.
9. Які алгоритми сортувань потребують дві ділянки пам’яті: для вхідної та вихідної множини даних?
10. Як буде виглядати масив {2,5,4,1,3} при бульбашковому сортуванні за зростанням після першого кроку?
11. Реалізацію якого алгоритму наведено в наступному фрагменті програмного коду?

```
{ for( i=0; i<N; i++)
{ for( k=i, j=i+1; j<N; j++) if (a[j] < a[k])  k=j;
swap(a[k],a[i]);  } }
```

12. За скільки кроків методом бульбашкового сортування з ознакою буде впорядковано за зростанням масив {2, 5, 6, 7, 7, 8, 9}?
13. Які алгоритми сортувань називають стійкими?

ПРАКТИЧНА РОБОТА 12. АЛГОРИТМИ СОРТУВАННЯ РОЗПОДІЛОМ ТА ЗЛИТТЯМ

Мета: закріпити теоретичні знання та набути практичного досвіду впорядкування набору статичних та динамічних структур даних.

Теми для попередньої роботи:

- масиви та списки;
- алгоритми сортування вибором та включенням;
- алгоритми сортування вибором на деревах, розподілом та злиттям.

Загальні відомості

У розглянутих раніше алгоритмах сортувань вибором виконувався пошук мінімального елемента з N елементів, потім з (N – 1), далі з (N – 2) і так до останнього елемента. У результаті кількість порівнянь становить $(N^2 – N)/2$.

Сортування вибором. У сортуваннях вибором, які розглядаються в цій роботі, загальний алгоритм такий: серед N елементів масиву в кожній парі сусідніх елементів визначається менший. При цьому буде виконано N/2 порівнянь і мінімальних елементів буде N/2. Далі із цих елементів визначаються ще мінімальні елементи, але тепер їх – N/4 і т.д., поки не буде знайдено один найменший елемент. Це перший етап – побудова дерева.

Другий етап – спуск по дереву, вибір елемента з найменшим ключем і реорганізація дерева. Саме так працює алгоритм турнірного сортування.

В основі пірамідального сортування лежить побудова піраміди. Флойд запропонував алгоритм сортування, що складається з двох етапів. На першому етапі на основі масиву побудується піраміда; при цьому найбільший елемент переміщається на вершину піраміди. На другому – найбільший елемент записується в кінець вхідної частини набору, а піраміда реорганізується. Порядок алгоритму – O(Nlog(N)).

Сортування розподілом. Опис загального алгоритму роботи такий: вхідна множина розбивається на ряд підмножин (можливо, меншого обсягу), і сортування проводиться усередині кожної такої підмножини. До цієї групи алгоритмів відносять порозрядне цифрове сортування, швидке сортування Хоара, «кишенькове» сортування, сортування виродженим розподілом.

Сортування злиттям. Опис загального алгоритму роботи такий:

вхідна множина з N елементів розглядається як N впорядкованих множин по одному елементу в кожній. Ці множини зливаються у впорядковані множини по два елементи, потім – по чотири і так далі. Вихідна множина утворюється шляхом злиття менших упорядкованих підмножин. До цієї групи алгоритмів відносять сортування прямим злиттям, сортування попарним злиттям. Ці алгоритми характеризуються порядком O(Nlog(N)).

Типове завдання

Завдання. Реалізувати алгоритм пірамідального сортування. Перевірити працездатність.

Текст програми

```
#include<iostream>
#include<conio.h>
#define N 40               //кількість елементів у масиві

void Create(int []);
void OutConsole(int []);
void HeapSort(int []);
#define swap(x,y)  {int t=x; x=y; y=t;}

void main()
{   setlocale(LC_CTYPE, "rus");      // для роботи з кирилицею
    int array[N];
    Create(array);
    std::cout << "Вихідний масив \n";
    OutConsole(array);
    HeapSort(array);
    std::cout << "Відсортований масив \n";
    OutConsole(array);
}

void Create(int a[])
{   for(int i = 0; i < N; i++)
       a[i]=rand() % 100;
}

void OutConsole(int a[])
{   for(int i = 0; i < N; i++)
             std::cout << "  " << a[i];
    std::cout << "\n";
}

void downHeap(int a[], int k, int n) // Процедура просіювання наступного елемента
{ // До процедури: a[k+1]...a[n]  - пірамида
  // Після:  a[k]...a[n]  - пірамида
  int new_elem;
  int child;
  new_elem = a[k];
  while(k <= n/2)                    // поки у a[k] є діти
```

```
  { child = 2*k;
    // обираємо більшого сина
    if( child < n && a[child] < a[child+1] )
      child++;
    if( new_elem >= a[child] ) break;
    a[k] = a[child];       //  інакше переносимо сина догори
    k = child;
  }
  a[k] = new_elem;
}
// Пирамідальне сортування
void HeapSort(int a[])
{ int i;
  // Будуємо піраміду
  for(i=N/2-1; i >= 0; i--) downHeap(a, i, N-1);
  // теперь a[0]...a[size-1] пирамида
  for(i=N-1; i > 0; i--)
  { // міняємо перший з останнім
    swap(a[i], a[0]);
    // відновлюємо пірамідальність a[0]...a[i-1]
    downHeap(a, 0, i-1);
  }
}
```

Результат роботи програми

```
 Вхідний масив
  83  86  77  15  93  35  86  92  49  21  62  27  90
59  63  26  40  26  72  36  11  68  67  29  82  30  62
  23  67  35  29  2  22  58  69  67  93  56  11  42
Відсортований масив
  2  11  11  15  21  22  23  26  26  27  29  29  30  3
5  35  36  40  42  49  56  58  59  62  62  63  67  67
 67  68  69  72  77  82  83  86  86  90  92  93  93
```

Індивідуальні завдання

Написати програму, що реалізує три алгоритми сортування набору даних згідно з табл. 12.1.

Визначити кількість порівнянь та обмінів для початкових наборів даних, що містять різну кількість елементів (50, 1000, 5000, 10000, 50000).

Оцінити час сортування. Дослідити вплив початкової впорядкованості набору даних (відсортований, відсортований у зворотному порядку, випадковий).

Всі отримані дані записати в табл. 12.2. Зробити висновки.

Таблиця 12.1 – Варіанти індивідуальних завдань

N	Номер алгоритму	N	Номер алгоритму	N	Номер алгоритму	N	Номер алгоритму
1	1, 4, 9	7	2, 6, 9	13	4, 1, 8	19	5, 1, 8
2	1, 4, 5	8	2, 7, 8	14	4, 2, 8	20	5, 2, 9
3	1, 6, 8	9	3, 4, 8	15	4, 3, 8	21	5, 3, 9
4	1, 7, 9	10	3, 5, 9	16	4, 1, 9	22	5, 4, 8
5	2, 4, 6	11	3, 7, 9	17	4, 2, 9	23	5, 7, 9
6	2, 5, 7	12	3, 6, 9	18	4, 3, 5	24	5, 6, 9

Примітки:

1) N – номер індивідуального завдання.

2) Алгоритм сортування за номером обрати з наступного переліку.

Алгоритми сортувань

1. Турнірне сортування .
2. Пірамідальне сортування.
3. Сортування частково впорядкованим деревом .
4. Порозрядне цифрове сортування.
5. Сортування Хоара.
6. «Кишенькове» сортування.
7. Сортування виродженим розподілом.
8. Сортування прямим злиттям.
9. Сортування попарним злиттям.

Таблиця 12.2 – Форма порівняльної таблиці ефективності алгоритму

Відсортований набір даних	20	1000	5000	10000	50000
Кількість порівнянь					
Кількість пересилань					
Час сортування					
Відсортований в зворотньому	20	1000	5000	10000	50000

порядку					
Кількість порівнянь					
Кількість пересилань					
Час сортування					
Випадковий набір даних	20	1000	5000	10000	50000
Кількість порівнянь					
Кількість пересилань					
Час сортування					

Контрольні запитання

1. Поясніть, чому алгоритми сортувань на деревах характеризуються порядком $Nlog_2(N)$?
2. В чому основна сутність алгоритму сортування Хоара?
3. Які алгоритми сортувань потребують додаткової ділянки пам'яті, і як вона використовується?
4. Назвіть алгоритми сортувань, в яких немає порівнянь елементів набору даних, що впорядковуються?
5. Які фактори впливають на вибір алгоритму сортування?
6. Напишіть програмний код «кишенькового» сортування з однією ділянкою пам'яті.
7. За скільки кроків буде впорядковано за зростанням такий набір чисел: 231, 24, 3, 35, 15, 932, 176, 83, 8, 54, 112?
8. Яке призначення має додаткова пам'ять, що необхідна алгоритму сортування Хоара?
9. Яке призначення додаткової пам'яті, що необхідна в алгоритмах пошуку підрядка в рядку? Назвіть ці алгоритми.
10. Дайте опис алгоритму турнірного сортування.
11. Дайте опис алгоритму сортування частково впорядкованим деревом.
12. Дайте опис алгоритму сортування прямим злиттям.
13. Дайте опис алгоритму сортування попарним злиттям.
14. Який метод лежить в основі побудови піраміди у вигляді масиву без явної побудови дерева?
15. Дайте опис алгоритму пірамідального сортування.
16. Які алгоритми сортувань називають стійкими?

ПРАКТИЧНА РОБОТА 13. АЛГОРИТМИ ОБРОБКИ ДЕРЕВ

Мета: набути досвіду практичної роботи з бінарними деревами.

Теми для попередньої роботи:
- нелінійні списки;
- графи;
- дерева;
- операції на деревах.

Загальні відомості

Дерево – такий граф, у якому існує єдина вершина, на яку не посилається ніяка інша вершина *(корінь).* На кожну вершину, крім кореня, є тільки єдине посилання і є вершини, що не посилаються на інші вершини дерева *(листи).* В деревах вершини прийнято називати вузлами. Кількість ребер графа, що виходять з вершини, називається напівстепенем виходу цієї вершини. Дерева, в яких напівстепінь виходу кожної вершини $\leq m$ (де m може дорівнювати 0,1,2,3 і т.д.), називають m-арними. Дерева, для яких $m = 2$, називають бінарними.

Ліс – декілька дерев, що становлять єдину систему і розглядаються разом. Ліс та дерева будь-якої арності за певними правилами можна перетворити в бінарне дерево.

Дерева можна подавати за допомогою масивів (послідовне подання) або нелінійних списків (списокове подання). При списковому поданні бінарних дерев для подання вузла створюється структура, що має інформаційне поле (може бути структурою) та два поля вказівників на ліве та праве піддерева (рис. 13.1).

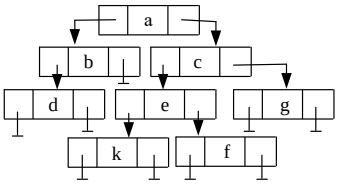


Рисунок 13.1 – Машинне подання дерева нелінійним списком

Бінарні дерева пошуку – такі дерева, в яких забезбечено співвідношення значення ключів у будь-якому вузлі розгалуження та його потомках, а також і між потомками.

Типи дерев: ідеально збалансовані, збалансовані та незбалансовані.

Операції на деревах: обхід дерева (зверху донизу, змішаний, знизу уверх), додавання, видалення, пошук вузла, балансування дерева.

Функції обробки дерев – рекурсивні. Для можливості написання нерекурсивних функцій дерева прошивають.

Прошивка дерева – заміна пустих вказівників у вузлах дерева на певні значення (на наступні вузли) залежно від правила обходу. При цьому у структуру для опису вузла додаються два поля, що характеризують зв'язок. Прошивка ускладнює реалізацію операцій додавання та видалення вузлів.

Типове завдання

Розробити програму, що забезпечить створення бінарного дерева, в якому ключі не повторюються. Видати вміст дерева на екран. Значення даних у вузлах дерева за рішенням користувача вводити з клавіатури або генерувати.

Текст програми

```
#include<conio.h>
#include<stdlib.h>
#include<iostream>
 using namespace std;

class CtreeNode
{        int key;                              //ключ
         int info;                    //інформаційне поле
         CtreeNode *LLink, *RLink; //вказівникна лівий і правий вузол
public:
         CtreeNode(){key=0;LLink=RLink=NULL;} //конструктор за умовчанням
         CtreeNode(int k, int info_c)              //конструктор с параметром
         {   key=k;
             info=info_c;
                  LLink=RLink=NULL;
         }
         ~CtreeNode()                          //Деструктор
         {
         }
  //функція додавання елемента в дерево. Повертає вказівник на корінь
    void Insert(int number, int info, CtreeNode *ptr);
    void PrintTree(CtreeNode *ptr, int n);
         void DeleteTree(CtreeNode *ptr);
};
void CtreeNode::Insert(int n, int inf, CtreeNode *ptr)
{        cout<<"\n Додається елемент ("<<n<<";"<<inf<<")\n";
         int flag=1;
         CtreeNode *p, *q;
```

```
          p=ptr;
          while(flag)
          { if(n < p->key)
            { q=p->LLink;     //a3
              cout<<"Знайден вузол "<<p->key<<endl<<"Перехід уліво";
              if(q==NULL)  cout<<"NULL"<<endl;
              else cout<<q->key<<endl;
              if(q == NULL)
              {    flag=0;
                   cout<<"Створення новий елемент ("<<n<<","<<inf<<")\n";
                   q=new CtreeNode(n,inf);
                   cout<<"Оновлення зв'яків: "<<p->key<<"->LLink = "<<q->key<<endl;
                   p->LLink=q;
              } else p=q;
            }
            else if(n > p->key)
            {  q=p->RLink;             //a4
               cout<<"Знайден вузол "<<p->key<<endl<<"Перехід управо";
               if(q == NULL)cout<<"NULL"<<endl;
               else cout<<q->key<<endl;
               if(q == NULL)
               {  flag=0;
                   cout<<"Створюється новий елемент ("<<n<<","<<inf<<")\n ";
                   q=new CtreeNode(n,inf);
                   cout<<"Оновлення зв'яків: "<<p->key<<"->RLink = "<<q->key<<endl;
                   p->RLink=q;
              }
                   else p=q;
           }
           else if(n == p->key)
           {       cout<<"Такий елемент вже існує"<<endl;
                   flag=0;//вихід із циклу WHILE
           }
  }
}
  void CtreeNode:: PrintTree(CtreeNode *ptr, int n)
  {      if (ptr)
         {   PrintTree(ptr->RLink, n+3);
             for(int i=1;i<n;i++)
                  cout << " ";
                  cout<< ptr->key<<endl;
              PrintTree(ptr->LLink, n+3);
         }
  }
  void CtreeNode::DeleteTree(CtreeNode *ptr)
  { if (ptr != NULL)
           {   DeleteTree(ptr->LLink);
               DeleteTree(ptr->RLink);
                    delete(ptr);
           }
}

void main()                //головна програма
{
          setlocale(LC_ALL, "Rus"); // для роботи з кирилицею
          int k_elem;
          char rand_elem;
          cout<<"  Скількм елементів додавати?  ";
          cin>>k_elem;
```

```
        cout<<"  Генерувати випадкові елементи? (y/n) ";
        cin>>rand_elem;
        int num,inf;
        if(rand_elem == 'n')
        {   cout<<endl<<"Індекс: "; //уводити з клавіатури
            cin>>num;
            cout<<"Значення: ";
            cin>>inf;
        }
       else
       {    num=rand() % 100 + 1;//випадкові значення
            inf=rand() % 100 + 1;
       }
        CtreeNode *root=new CtreeNode(num,inf);       //вказівник на корень
        for(int tr_i=1;tr_i<k_elem;tr_i++)
        { if(rand_elem == 'n')
          {      cout<<endl<<"Індекс: ";     //уводити з клавіатури
                cin>>num;
                 cout<<"Значення: ";
                 cin>>inf;
           }
           else
           {    num=rand() % 100 + 1;//випадкові значення
                inf=rand() % 100 + 1;
           }
           root->Insert(num,inf,root);
  }
        root->PrintTree(root, 5);
    root->DeleteTree(root);          //звільнення пам’яті
}
```

Фрагмент результатів роботи програми

```
Оновлення зв’яків: 84->LLink = 78

 Додається елемент (94;36)
Знайден вузе 84
Перехід управоNULL
Створюється новий елемент (94,36)
 Оновлення зв’яків: 84->RLink = 94

 Додається елемент (87;93)
Знайден вузе 84
Перехід управо94
Знайден вузел 94
Перехід улів►NULL
Створення новий елемент (87,93)
Оновлення зв’яків: 94->LLink = 87

 Додається елемент (50;22)
Знайден вузел 84
П►ехід улівo78
Знайден вузел 78
Перехід улівoNULL
Ствo►ення новий елемент (50,22)
Оновлення зв’яків: 78->LLink = 50
       94
          87
    84
       78
          50
```

Індивідуальні завдання

Розробити програму, що дозволяє створити бінарне дерево та вирішити індивідуальне завдання. У завданнях 7 – 16 включно (де не вказані обходи) реалізувати два алгоритми обходу, які обрати самостійно.

В інформаційну частину вузлів записати ключ, що є цілим числом. Видати вміст дерева та результати індивідуального завдання на екран.

1. Знайти вузол з ***max*** ключем; використати спадний та змішаний обходи дерева із використанням стека.
2. Знайти вузол з ***min*** ключем; використати висхідний та змішаний обходи дерева із використанням стека.
3. Визначити кількість вузлів, що мають ***max*** ключ; використати рекурсивний алгоритм змішаного та висхідного обходу.
4. Визначити кількість парних вузлів; використати змішаний та спадний обходи із використанням стека.
5. Розробити програму турнірного сортування.
6. Визначити ***K*** – кількість вузлів, ключ яких більший від заданого числа ***N***; використати рекурсивний алгоритм спадного обходу дерева та із використанням стека; додати вузол з ключем ***K***.
7. Розробити програму обчислення висоти дерева.
8. Видалити вузли, ключ яких дорівнює заданому числу ***N***.
9. Створити два дерева: в одне переписати тільки парні ключі, а в друге – непарні.
10. Додати після першого вузла з ключем ***K*** вузол з ключем ***N***. Значення ***K*** та ***N*** ввести з клавіатури.
11. Видалити вузли з непарними ключами.
12. Визначити середньоарифметичне значення парних та непарних ключів дерева.
13. Визначити кількість листів у дереві.
14. Визначити кількість внутрішніх вузлів у дереві.
15. Визначити кількість вузлів, що мають тільки ліве посилання.
16. Визначити кількість вузлів, що мають тільки праве посилання.
17. Порівняти час висхідного та спадного обходів дерева.
18. Порівняти час висхідного та змішаного обходів дерева.
19. Порівняти час змішаного та спадного обходів дерева.
20. Прошити дерево для його спадного обходу. Реалізувати спадний

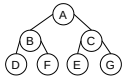
обход дерева. Порівняти час обходу прошитого та не прошитого дерева.

21. Прошити дерево для його змішаного обходу. Реалізувати змішаний обход дерева. Порівняти час обходу прошитого та не прошитого дерева.

22. Прошити дерево для його висхідного обходу. Реалізувати висхідний обход дерева. Порівняти час обходу прошитого та не прошитого дерева.

Контрольні запитання

1. Що таке дерево?
2. Чим визначається арність дерева?
3. Які операції властиві деревам?
4. Що таке «обхід» дерева, які операції обходу відомі?
5. Яке дерево називається деревом пошуку?
6. Напишіть структуру m-арного дерева за своїм розсудом. Перетворіть його в бінарне.
7. Які виділяють типи дерев?
8. Опишіть алгоритм висхідного обходу дерева.
9. Опишіть алгоритм спадного обходу дерева.
10. Опишіть алгоритм змішаного обходу дерева.
11. Вкажіть послідовність вузлів цього дерева



при спадному, висхідному та змішаному його обходах.

12. Яке дерево називається повним m-арним?
13. Яке дерево називається ідеально збалансованим?
14. Яке дерево називається повним бінарним?
15. Дано арифметичний вираз $((a + b) \cdot (c - d) + f) \cdot k$. Побудуйте для нього бінарне дерево.
16. Запишіть арифметичний вираз за своїм розсудом. Створіть відповідне для нього дерево. Виконавши обхід дерева, запишіть префіксну, інфіксну та постфіксну форми запису заданого виразу.
17. Які дерева називаються червоно-чорними, які їх властивості?
18. Яке призначення операції балансування дерев?
19. Якими властивостями характеризуються АВЛ-дерева?
20. Які властивості мають ідеально збалансовані дерева?

ПРАКТИЧНА РОБОТА 14. ВИКОРИСТАННЯ ДЕРЕВ

Мета: набути досвіду практичної роботи розв'язання задач з використанням бінарних дерев.

Теми для попередньої роботи:

- балансування дерев;
- АВЛ-дерева;
- червоно-чорні дерева;
- дерева Хаффмана;
- дерева для арифметичних виразів.

Загальні відомості

Одним із методів, що поліпшують час пошуку в бінарному дереві, є створення збалансованих дерев, що характеризуються мінімальним часом пошуку. Дерево пошуку є ідеально збалансованим, якщо для кожної його вершини кількість вершин у лівому і правому піддеревах відрізняється не більш, ніж на 1. У процесі коректування дерева ідеальне балансування його часто порушується. Більш «м'яке» визначення збалансованості введене для АВЛ-дерев: дерево пошуку збалансовано, якщо висота лівого і правого піддерев будь-якої вершини відрізняється не більш ніж на 1. У таких деревах необхідність в операціях балансування з'являється рідше, а самі ці операції досить прості.

Відновлення балансу в АВЛ-деревах досягається шляхом перетворень, які називають поворотами. Є чотири типи поворотів: L-поворот, R-поворот, LR-поворот і RL-поворот. Перші два з них вважаються одинарними, а останні два – подвійними. Це пов'язане з кількістю операцій, необхідних для відновлення балансу. Вставка нового ключа в АВЛ-дерево або видалення вузла може поліпшити балансування дерева, а може призвести до порушення його збалансованості. У такому випадку виконується балансування поточного вузла.

Вузли АВЛ-дерева описуються такою структурою:

```
struct node
{  int key;
   unsigned char height;
   node* left;
   node* right;
```

```
};
```

Поле *key* зберігає ключ вузла, поле *height* – висоту піддерева з коренем у цьому вузлі, поля *left* і *right* – вказівники на ліве і праве піддерева. Конструктор створює новий вузол (висоти 1) із заданим ключем k.

Червоно-чорне дерево – це одне із самобалансованих двійкових дерев пошуку, що гарантують логарифмічне зростання висоти дерева від кількості вузлів, у яких найбільш швидко виконуються основні операції дерев пошуку: додавання, видалення і пошук вузла. Збалансованість досягається за рахунок уведення додаткового атрибута у структуру вузла дерева – «кольору». Цей атрибут може мати одне із двох можливих значень – «чорний» або «червоний».

```
struct Node
{   Node *left;         //  left спадкоємець
    Node *right;       //  right спадкоємець
    Node *parent;      // батьківський вузол
    nodecolor color;            //  колір вузла (BLACK, RED)
    int key;                     //  ключ
};
```

Дерева мають широке застосування при реалізації трансляторів, при роботі з арифметичними виразами, при створенні і веденні таблиць символів, у базах даних, у системах зв'язку тощо.

Дерева мінімального кодування – спосіб побудови кодової таблиці. Зазвичай коди символів створюються на основі частоти їх появи у всій множині повідомлень. Операція об'єднання двох символів у один використовує структуру бінарного дерева. Кожний вузол містить символ і частоту входження. Код будь-якого символу може бути визначений переглядом дерева знизу уверх, починаючи з листа. Щоразу при проходженні вузла приписують ліворуч до коду 0, якщо піднімаються по лівій гілці і 1 – якщо піднімаються по правій гілці. Як тільки дерево побудоване, код будь-якого символу алфавіту може бути визначений переглядом дерева знизу уверх, починаючи з місця, що становить цей символ.

За допомогою дерев можна подавати довільні арифметичні виразу. Кожному листу в такому дереві відповідає операнд, а кожному батьківському вузлу – операція. Обхід дерева дозволяє побудувати будь-яку форму запису арифметичного виразу: префіксну, інфіксну або постфіксну.

Типове завдання

Розробити програму, що забезпечить створення АВЛ-дерева пошуку, видалення вузлів за значенням ключа та його балансування. Видати вміст дерева на екран. Значення даних у вузлах задаються у програмі.

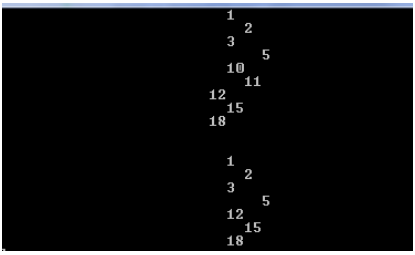
Текст програми

```
#include "stdlib.h"
#include "stdio.h"

struct node // структура для подання вузлів дерева
{
	int key;
	unsigned char height;
	node* left;
	node* right;
};
unsigned char Height(node* p)
{
	return p? p->height : 0;
}
int Bfactor(node* p)
{
	return Height(p->right) - Height(p->left);
}
void FixHeight(node* p)
{
	unsigned char hl = Height(p->left);
	unsigned char hr = Height(p->right);
	p->height = (hl > hr? hl : hr)+1;
}
node* RotateRight(node* p) // правий поворот навколо p
{	node* q = p->left;
	p->left = q->right;
	q->right = p;
	FixHeight(p);
	FixHeight(q);
	return q;
}
node* RotateLeft(node* q) // лівий поворот навколо q
{	node* p = q->right;
	q->right = p->left;
	p->left = q;
	FixHeight(q);
	FixHeight(p);
	return p;
}
node* Balance(node* p) // балансування вузла p
{	FixHeight(p);
	if( Bfactor(p)==2 )
	{	if( Bfactor(p->right) < 0 )
			p->right = RotateRight(p->right);
		return RotateLeft(p);
	}
	if( Bfactor(p)==-2 )
	{	if( Bfactor(p->left) > 0  )
			p->left = RotateLeft(p->left);
```

```
                return RotateRight(p);
        }
        return p; // балансування не потрібне
}
node* Insert(node* p, int k) // вставлення ключа k в дерево з коренем p
{       if( !p )
        {       p = new node;
                p->key = k; p->left = p->right = 0; p->height = 1;
                return p;
        }
        if( k<p->key )
                p->left = Insert(p->left,k);
        else
                p->right = Insert(p->right,k);
        return Balance(p);
}
node* FindMin(node* p) // пошук вузла з інімальним ключем у дереві p
{       return p->left? FindMin(p->left):p;
}
node* RemoveMin(node* p) // видалення вузла з мінімальним ключем із дерева p
{       if( p->left==0 )
                return p->right;
        p->left = RemoveMin(p->left);
        return Balance(p);
}
node* Remove(node* p, int k) // видалення ключа k із дерева p
{       if( !p ) return 0;
        if( k < p->key )
                p->left = Remove(p->left,k);
        else if( k > p->key )
                p->right = Remove(p->right,k);
        else //  k == p->key
        {       node* q = p->left;
                node* r = p->right;
                delete p;
                if( !r ) return q;
                node* min = FindMin(r);
                min->right = RemoveMin(r);
                min->left = q;
                return Balance(min);
        }
        return Balance(p);
}
void Print(node *tree, int level)
{       int i;
        if (tree == NULL)  return;
        Print(tree->left, level - 2);
        for (i = 0; i < level; i++)
                printf(" ");
        printf("%d\n", tree->key);
        Print(tree->right, level - 2);
}
void Delete(node *tree)  //видалення дерева
{       if (tree == NULL) return;
        Delete(tree->left);
        Delete(tree->right);
        free(tree);
}
```

```
int main()
{         node *tree = NULL;
          tree = Insert(tree, 10);    //побудова дерева
          tree = Insert(tree, 5);
          tree = Insert(tree, 3);
          tree = Insert(tree, 1);
          tree = Insert(tree, 2);
          tree = Insert(tree, 11);
          tree = Insert(tree, 12);
          tree = Insert(tree, 15);
          tree = Insert(tree, 18);
          Print(tree, 30);
          printf("\n\n");
          Remove(tree, 10);
          Remove(tree, 11);
          Print(tree, 30);
          Delete(tree);
          return 0;
}
```

Результат роботи програми



Індивідуальні завдання

Розробити програму, що створює бінарне дерево та розв'язує індивідуальне завдання. Видати вміст дерева та результати індивідуального завдання на екран.

1. Обчислити значення арифметичного виразу. У вузлах дерева записати операнди та операції арифметичного виразу у вигляді зворотного польського запису.

2. Побудувати АВЛ-дерево. Визначити значення максимального ключа у дереві; видалити всі вузли, значення ключів яких дорівнює максимальному.

3. Побудувати червоно-чорне дерево. Забезпечити видалення будь-якої кількості вузлів за заданим значенням.

4. Побудувати дерево Хаффмана для кодування одного сповіщення. Видати отримані коди символів та закодоване сповіщення.

5. Побудувати червоно-чорне дерево. Визначити висоту дерева. Додати вузол із ключем, значення якого визначено висотою дерева.

6. Побудувати АВЛ-дерево. Забезпечити видалення будь-якої кількості вузлів за заданими значеннями.

7. Побудувати АВЛ-дерево. Забезпечити додавання будь-якої кількості вузлів у дерево.

8. Побудувати червоно-чорне дерево, в якому однакові ключі можуть повторюватися. Визначити, які ключі скільки разів повторилися в дереві.

9. Розробити програму для побудови дерева Хаффмана для кодування вашого прізвища, ім’ та по батькові. Видати отримані коди символів.

10. Побудувати АВЛ-дерево. Визначити значення мінімального ключа у дереві; видалити всі вузли, значення ключів яких дорівнює мінімальному.

11. Побудувати АВЛ-дерево, в якому однакові ключі можуть повторюватися. Визначити, які ключі скільки разів повторилися в дереві.

12. Побудувати АВЛ-дерево. Визначити висоту дерева. Додати вузол із ключем, значення якого визначено висотою дерева.

Контрольні запитання

1. Які дерева називають червоно-чорними. Які їх властивості?
2. Які дерева називають АВЛ-деревами. Які їх властивості?
3. Які дерева називають ідеально збалансованими. Які їх властивості?
4. Яке призначення дерев Хаффмана?
5. Напишіть арифметичний вираз із кількістю операцій не менше 5. Побудуйте для нього бінарне дерево. Запишіть інфіксний, префіксний та постфіксний записи цього виразу.
6. Напишіть 5 символів та оберіть для них частоту появи у тексті. Побудуйте для них дерево Хаффмана. Побудуйте коди для заданих символів.
7. Напишіть ваше прізвище. Закодуйте його за допомогою дерева мінімального кодування.
8. Що таке «балансування» дерев? При виконанні яких операцій дерево може розбалансуватися?
9. Як виконується балансування дерев?
10. Коли, в який момент, може виконуватися балансування дерева?

ПРАКТИЧНА РОБОТА 15. ФАЙЛОВІ ТИПИ ДАНИХ

Мета: отримати та закріпити знання про типи даних для зовнішніх носіїв інформації.

Теми для попередньої роботи:

- засоби роботи з файлами в мовах програмування;
- хешування, боротьба з колізіями;
- дерева.

Загальні відомості

Логічне та фізичне подання файлів. Комп'ютерним файлом називають ресурс для зберігання даних або програм на постійному запам'ятовувальному пристрої комп'ютера. Файли мають такі ознаки:

- фіксована назва, що однозначно характеризує файл;
- певний метод доступу (відповідні операції читання та запису);
- певна логічна будова (організація даних, що зберігаються).

Ознаками, які відокремлюють один структурний елемент від іншого, можуть служити певні кодові послідовності або просто відомі програмі значення зсувів цих структур відносно початку файла. Підтримка структури даних може бути цілком покладена на додаток, або тією чи іншою мірою цю роботу може виконувати файлова система.

Неструктурований файл. У випадку, коли всі дії, що пов'язані зі структуризацією та інтерпретацією вмісту файла, виконуються додатком, файл є неструктурованою послідовністю даних. Додаток формує запит до файлової системи на уведення-виведення, використовуючи загальні для всіх додатків засоби, наприклад, указуючи зсув від початку файла та кількість байтів, що необхідно зчитати або записати. До додатку надходить потік байтів, який інтерпретується відповідно до закладеної у програмі логіки.

Структурований файл. У цьому випадку підтримка структури файла доручається файловій системі. Файлова система «бачить» такий файл як упорядковану послідовність логічних записів. Додаток звертається до файлової системи із запитами на уведення-виведення на рівні логічних записів. У свою чергу, файлова система повинна мати інформацію про структуру файла. Розвитком цього підходу стали системи керування базами даних, які підтримують не лише складну структуру даних, але і зв'язки між

ними.

У мовах програмування за логічною структурою файли поділяються на двійкові та текстові. Цей поділ відображається у понятті інтерпретації даних у розроблюваних додатках.

Текстовий файл. Файли призначені для збереження текстової інформації. Такі файли є сукупністю символьних рядків змінної довжини. Кінець кожного рядка у такому файлі позначається спеціальним маркером «кінець рядка». Текстові файли характеризуються також тим, що інформація в них має такий самий вигляд, як і на екрані, тобто її можна читати. Відповідно, текстові файли можна читати і змінювати вручну будь-яким текстовим редактором.

Двійкові файли. Такі файли використовуються для збереження даних певного типу. Вони зберігають інформацію у внутрішньому поданні. Такі файли містять у собі лише послідовність байтів, а отже, не можуть створюватись і виправлятись вручну. Однак ці незручності компенсуються швидкістю роботи з даними. У загальному випадку бінарні файли поділяються на типізовані та нетипізовані. Для типізованих файлів властива наявність одиниць читання та запису, що характеризуються певним типом. У двійкових файлах можливий прямий доступ до даних, завдяки можливості безпосередньо вказувати адресу елемента.

Типове завдання

Створити хешований бінарний файл великого розміру, організований розділами. Забезпечити можливість додавання та видалення записів, пошуку за ключовим полем.

Текст програми

```
#include<conio.h>
#include<stdlib.h>
#include<dos.h>
#include<string.h>
#include<iostream>
#include <time.h>

#define N 10
#define M 90
#define first
#define G 1000

struct Tdata
{ int idn;
  int age;
  char name[20];
```

```
  int g[150];
};

struct Trazdel
{  Tdata data;
   int free;
   };

FILE *in,*head;
int hsize=40;
Tdata nulldata={0,0,""};

unsigned int hesh(int idn)
{
   return idn % N;
}

int addkey(Trazdel razdel,int adr);
void heshcreate();

void initfile()
{
  int i;
  Trazdel razdel;
  in=fopen("in.bin","wb");
  head=fopen("head.bin","wb");
  for(i=0;i<M;i++)
  { razdel.data.idn=rand()%100;
    razdel.data.age=10 + rand()%10;
    sprintf(razdel.data.name,"%3iname",i);
    razdel.free=1;
    fwrite(&razdel,sizeof(Trazdel),1,in);
         int adr=ftell(in);
         addkey(razdel,adr);
  }
  heshcreate();
  fclose(in);
  fclose(head);
}

void initshow(int f)
 {  int i;
    Trazdel razdel;
    in=fopen("in.bin","rb");
    while(!feof(in))
    {   fread(&razdel,sizeof(Trazdel),1,in);
      //if(f&razdel.free)
        printf("%4d %3d%s ",razdel.data.idn,razdel.data.age,razdel.data.name);
    }
    fclose(in);
 }

void heshcreate()
 {  head=fopen("head.bin","wb");
    int i,l;
    for(i=0;i<N;i++)
   {   l=i*hsize+N*2;
       fwrite(&l,2,1,head);
   }
```

```
   l=-1;
   for(i=0;i<200;i++)
      fwrite(&l,2,1,head);
   fclose(head);
 }

int addkey(Trazdel razdel,int adr)
 {  int key=hesh(razdel.data.idn);
    int i=0;
    int t=0,j;
    fseek(head,key*2,0);
    fread(&key,2,1,head);
    fseek(head,key,0);
    while((i!=-1)&(t<20))
         {    fread(&i,2,1,head);
              t++;
    }
    if(t==20) {printf("Add fail"); return 0;}
    fseek(head,-2,1);
    j=ftell(head);
    fwrite(&adr,2,1,head);
 }

void heshzap()
 {  Trazdel razdel;
    int i=0;
head=fopen("head.bin","r+b");
in=fopen("in.bin","rb");
while(i<M)
{  fread(&razdel,sizeof(Trazdel),1,in);
   addkey(razdel,i*sizeof(Trazdel));
   i++;
}
fclose(head);
fclose(in);
 }

void show()
{  Trazdel razdel;
         head=fopen("head.bin","rb");
         in=fopen("in.bin","rb");
         int key,i=0,j=0;
         int m=0;
         int x=0,lst=0,y=1;
         for(i=0;i<M; i++)//i<10;i++)
         {   j=0;
                  fseek(head,i*2,0);
                  fread(&key,2,1,head);
                  fseek(head,key,0);
                  while((j<20))
                  {   fread(&key,2,1,head);
                      if(key!=-1)
                      {    fseek(in,key,0);
                           fread(&razdel,sizeof(Trazdel),1,in);
                           printf("|%4d %3d
%s\n",razdel.data.idn,razdel.data.age,razdel.data.name);
                           if(++lst==25)
                                    {    lst=0; x++; y=1;}
                          m++;
```

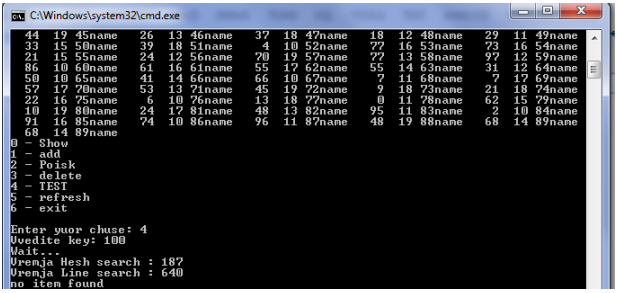
```
                        }
                j++;
      }
   }
}

Tdata hsearch(int key)
 {
  int hsh;
  int j=0;
  hsh=hesh(key);
  Trazdel razdel;
  fseek(head,hsh*2,0);
  fread(&hsh,2,1,head);
  fseek(head,hsh,0);
  while(j<20)
  {  fread(&hsh,2,1,head);
     if(hsh!=-1)
     {   fseek(in,hsh,0);
         fread(&razdel,sizeof(Trazdel),1,in);
         if(razdel.data.idn==key)  return razdel.data;
         j++;
     }
  }
  return nulldata;
}

void add()
 { Trazdel razdel;
   head=fopen("head.bin","r+b");
   in=fopen("in.bin","r+b");
   printf("Vvedite dannie:\nidentifire:");
   std::cin>>razdel.data.idn;
   printf("age:");
   std::cin>>razdel.data.age;
   printf("name:");
   std::cin>>razdel.data.name;
   razdel.free=1;
   if (in != NULL)
   {  fseek(in,0,2);
      int adr=ftell(in);
      addkey(razdel,adr);
      fwrite(&razdel,sizeof(Trazdel),1,in);
   }
   fclose(in);
   fclose(head);
 }

void search()
{  int key;
   printf("Vvedite key:\n");
   std::cin>>key;
   Tdata dat=hsearch(key);
   if(dat.age!=0)
       printf("Najdeno.\n %4d %3d %s",dat.idn,dat.age,dat.name);
   else printf("Ne najdeno");
   system("pause");
}
```

```
void dell()
 {    head=fopen("head.bin","r+b");
         in=fopen("in.bin","r+b");
         int key;
         printf("Vvedite key:\n");
         std::cin>>key;
         int hsh,hsh2;
         int j=0;
         hsh=hesh(key);
         Trazdel razdel;
         fseek(head,hsh*2,0);
         fread(&hsh,2,1,head);
         fseek(head,hsh,0);
         fread(&razdel,sizeof(Trazdel),1,in);
         hsh2=hsh;
         while((j<20)&&(razdel.data.idn!=key))
         {   fread(&hsh,2,1,head);
                  if(hsh!=-1)
                  {     fseek(in,hsh,0);
                        fread(&razdel,sizeof(Trazdel),1,in);
                        j++;
                  }
   }
   if(j==20) printf("No element!");
   else
   {   fseek(in,hsh,0);
       fseek(head,-2,1);
       razdel.free=0;
       fwrite(&razdel,sizeof(Trazdel),1,in);
       hsh2=-1;
       fwrite(&hsh2,2,1,head);
       printf("Del Ok");
    }
  fclose(in);
  fclose(head);
}

void refresh()
 {  initfile();
    heshcreate();
    heshzap();
 }

Tdata lsearch(int key)
  {  Trazdel razdel;
     int j=0;
     fread(&razdel,sizeof(Trazdel),1,in);
     while(!((razdel.free==1)&&(razdel.data.idn==key))&&(j<150))
     {   j++;
         fread(&razdel,sizeof(Trazdel),1,in);
     }
     if(j<150)  return  razdel.data;
     else return nulldata;
  }

void TEST()
 {  clock_t t1,t2;
  head=fopen("head.bin","rb");
  in=fopen("in.bin","rb");
```

```
  long l;
  int key;
  Tdata dat;
  printf("Vvedite key: ");
  std::cin>>key;
  printf("Wait...\n");
  t1=clock();
  for(l=0;l<G;l++)
     dat=hsearch(key);
  t2=clock();
  printf("Vremja Hesh search : %d\n",t2-t1);
  fclose(in);
  fclose(head);
  head=fopen("head.bin","rb");
  in=fopen("in.bin","rb");
  t1=clock();
  for(l=0;l<G;l++)
     dat=lsearch(key);
  t2=clock();
  dat=lsearch(key);
  printf("Vremja Line search : %d\n",t2-t1);
  if(dat.age==0) printf("no item found\n");
  else printf("%3d %3d",dat.idn,dat.age);
  fclose(in);
  fclose(head);
  system("pause");
}

int menu()
{  int k;
   printf("\n0 - Show\n");
   printf("1 - add\n");
   printf("2 - Poisk\n");
   printf("3 - delete\n");
   printf("4 - TEST\n");
   printf("5 - refresh\n");
   printf("6 - exit\n\n");
   printf("Enter yuor chuse: ");
   scanf("%i", &k);
   return k;
}

void main()
{   initfile();
    heshcreate();
    initshow(2);
    int k = menu();
    while(k<7)
   {   switch (k)
       {    case 0: show(); system("pause");; break;
            case 1: add(); show(); system("pause");;break;
            case 2: search(); break;
            case 3: dell(); show(); system("pause");;break;
            case 4: TEST(); break;
            case 5: refresh(); break;
            case 6:  exit(1); break;
      }
         k = menu();
    }
```

```
   system("pause");
}
```

Результат роботи програми:



Індивідуальні завдання

Розробити програму для роботи з даними, що зберігаються на диску.

При цьому забезпечити:

− створення з послідовного невпорядкованого файла початкових (вхідних) даних, тип якого визначається з табл. 15.1 за номером студента у журналі;

− додавання та видалення записів із файла;

− друк вмісту файла;

− пошук запису за ключем.

При виконанні операцій вважати, що файл настільки великий, що не може бути увесь переписаний у пам’ять комп’ютера.

Таблиця 15.1 – Індивідуальні завдання

N	Тип файла, що створюється	Вміст вхідних даних
1	2	3
1	Послідовний із записами змінної довжини	Дані про робітників великого підприємства
2	Послідовний із записами постійної довжини	Дані про країни, міста та селища
3	Хеш-файл (хеш-функцію обрати самостійно)	Дані про ріки Євразії
4	B-дерево	Дані про власників телефонів
5	Індексно-послідовний файл	Дані про учнів школи

6	Хеш-файл (хеш-функцію обрати самостійно)	Дані з лікарняних карток
7	Індексно-послідовний файл	Дані про флору України
8	Послідовний із записами постійної довжини	Дані про студентів інституту
9	B-дерево	Дані про власників телефонів
10	B^+ -дерево	Дані про фауну України
11	Хеш-файл (хеш-функцію обрати самостійно)	Дані про наявність місць у поїздах по всіх напрямках
12	B-дерево	Дані бібліотечного фонду
13	Послідовний із записами змінної довжини	Статистичні дані про збір врожаю сільськогосподарських культур за N років
14	B^+ -дерево	Синоптичні показники за всі роки спостереження

Примітка: N – номер студента в журналі.

Контрольні запитання

1. Що означає термін «Фізична організація файла»?
2. Що таке «фрагментація дискового простору»?
3. Які найбільш використовувані схеми розміщеня даних у файлах на фізичному рівні?
4. Що таке «блок» файла?
5. Як записуються дані у неперервних файлах?
6. Як записуються дані у файлах, які подаються зв’язним списком блоків?
7. Як записуються дані у файлах, які подаються зв’язним списком індексів блоків?
8. Які типи файлів розрізняють на логічному рівні?
9. Що являють собою байторієнтовані файли?
10. Що являють собою структуровані файли?
11. Який вигляд може мати логічна структура файлів прямого доступу?
12. Яка структура хешованих файлів?
13. Як створюється файл за принципом B-дерева?
14. У чому відмінність файлів, які створюються за принципом B+-дерева від фалів, створенних за принципом B-дерева?

ПРАКТИЧНА РОБОТА 16. АЛГОРИТМИ ЗОВНІШНЬОГО СОРТУВАННЯ

Мета: набути та закріпити навички впорядкування вмісту дуже великих файлів.

Теми для попередньої роботи:

- алгоритми пошуку;
- алгоритми внутрішнього сортування;
- фізична та логічна структура файлів;
- двійкові файли;
- послідовні та файли прямого доступу.

Види алгоритмів зовнішнього сортування

Основні поняття. *Зовнішнє сортування* − це сортування даних, які розташовані на зовнішніх пристроях і не уміщуються в оперативну пам'ять. До найбільш відомих алгоритмів зовнішнього сортування належать:

- сортування злиттям (просте злиття і природне злиття);
- поліпшене сортування (багатофазне та каскадне сортування).

Основним поняттям при використанні зовнішнього сортування є поняття серії. *Серія* (упорядкований відрізок) − це послідовність елементів, яка впорядкована за ключем. Кількість елементів у серії називається довжиною серії. Серія, що складається з одного елемента, впорядкована завжди. Остання серія може мати довжину меншу, ніж решта серії файлів. Максимальна кількість серій у файлі N (всі елементи невпорядковані). Мінімальна кількість серій одна (всі елементи впорядковані).

В основі більшості методів зовнішнього сортування лежить процедура злиття і процедура розподілу. *Злиття* − це процес об'єднання двох (або більше) впорядкованих серій в одну впорядковану послідовність за допомогою циклічного вибору елементів, доступних у цей момент. *Розподіл* − це процес розподілу впорядкованих серій на два або кілька допоміжних файлів. Фаза − це дія за одноразовим обробленням усієї послідовності елементів. *Двофазне сортування* − це сортування, в якому окремо реалізується дві фази: розподіл і злиття. *Однофазне сортування* − це сортування, в якому об'єднані фази розподілу і злиття в одну. *Двошляховим злиттям* називається сортування, в якому дані розподіляються на два допоміжних файли. *Багатошляховим злиттям* називається сортування, в

якому дані розподіляються на N (N > 2) допоміжних файлів.

Алгоритм прямого злиття. Нехай є послідовний файл *A*, що складається із записів $a_1, a_2, ..., a_n$ (для простоти припустимо, що *n* є степень числа 2). Для сортування використовуються два допоміжні файли *B* і *C*, розмір кожного з них буде *n/2*.

Сортування складається з послідовності кроків, у кожному з яких виконується розподіл вмісту файла *A* у файли *B* і *C*, а потім злиття файлів *B* і *C* у файл *A*. На першому кроці для розподілу послідовно читається файл *A*, і записи $a_1, a_3, ..., a_{(n-1)}$ пишуться у файл *B*, а записи $a_2, a_4, ..., an$ – у файл *C* (початковий розподіл).

Початкове злиття проводиться над парами $(a_1, a_2), (a_3, a_4), ..., (a_{(n-1)}, an)$, і результат записується у файл *A*. На другому кроці знову послідовно читається файл *A*, і у файл *B* записуються послідовні пари з непарними номерами, а у файл *C* – з парними. При злитті утворюються і пишуться у файл *A* упорядковані четвірки записів. І так далі. Перед виконанням останнього кроку файл *A* буде містити дві впорядковані підпослідовності розміром n/2 кожна. При розподілі перша з них потрапить у файл *B*, а друга – у файл *C*. Після злиття файл *A* буде містити повністю впорядковану послідовність записів.

Природне злиття. При використанні методу прямого злиття не береться до уваги те, що вихідний файл може бути частково відсортованим, тобто містити впорядковані підпослідовності записів – серії. Метод природного злиття ґрунтується на розпізнаванні серій при розподілі та їх використанні при наступному злитті.

Як і у випадку прямого злиття, сортування виконується за кілька кроків, у кожному з яких спочатку виконується розподіл файла *A* по файлах *B* і *C*, а потім злиття *B* і *C* у файл *A*. При розподілі розпізнається перша серія записів і записується у файл *B*, друга – у файл *C* і т.д. При злитті перша серія записів файла *B* зливається з першою серією файла *C*, друга серія *B* – із другою серією *C* і т.д. Якщо перегляд одного файла закінчується раніше, ніж перегляд іншого (через різне число серій), то залишок недопереглянутого файла цілком копіюється в кінець файла *A*. Процес завершується, коли у файлі *A* залишається тільки одна серія.

Багатофазне сортування. Ідея багатофазного сортування полягає в тому, що з наявних *m* допоміжних файлів *(m-1)* файл служить для введення послідовностей, що зливаються, а один – для виведення утворених серій. Як тільки один із файлів уведення стає порожнім, його починають

використовувати для виведення серій, одержуваних при злитті серій нового набору *(m-1)* файлів. Таким чином, на першому кроці, серії початкового файла розподіляються по *m-1* допоміжних файлів, а потім на наступних кроках виконується багатошляхове злиття серій з *(m-1)* файлів, поки в одному з них не утворюється одна серія.

Доведено, що метод трифазного зовнішнього сортування дає бажаний результат і працює максимально ефективно (на кожному етапі зливається максимальне число серій), якщо початковий розподіл серій між допоміжними файлами описується сусідніми числами Фібоначчі.

У загальному вигляді при використанні *m* допоміжних файлів умовою успішного завершення та ефективної роботи методу багатофазного зовнішнього сортування є те, щоб початковий розподіл серій між *m-1* файлами описувався сумами сусідніх *(m-1), (m-2), ..., 1* чисел Фібоначчі порядку *m-2*. Послідовність чисел Фібоначчі порядку *p* починається з *(p-1)* нулів, p-*й* елемент дорівнює 1, а кожний наступний дорівнює сумі попередніх *p* елементів. Для прикладу наведемо послідовність чисел Фібоначчі порядку 5: 0, 0, 0, 0, 1, 1, 2, 4, 8, 16, 31, 61, ...

Зрозуміло, що якщо розподіл заснований на числі Фібоначчі *fi*, то мінімальне число серій у допоміжних файлах буде дорівнювати *fi*, а максимальне – *f(i+1).* Тому після виконання злиття отримано максимальне число серій – *fi*, а мінімальне – *f(i-1).* На кожному етапі буде виконуватися максимально можливе число злиттів, і процес зведеться до наявності лише однієї серії.

Оскільки число серій у вихідному файлі може не забезпечувати можливість такого розподілу серій, застосовується метод додавання порожніх серій, які надалі як можна більш рівномірно розподіляються між проміжними файлами і орієнтуються при наступних злиттях. Зрозуміло, що чим менше таких порожніх серій, тим ближче число початкових серій до вимог Фібоначчі, тим більш ефективно працює алгоритм.

Підвищення ефективності зовнішнього сортування. Зрозуміло, що чим довші серії містить файл перед початком застосування зовнішнього сортування, тим менше буде потрібно злиттів і тим швидше закінчиться сортування. Тому до початку застосування кожного з методів зовнішнього сортування, заснованих на застосуванні серій, початковий файл частинами зчитується в основну пам’ять, до кожної частини застосовується один із найбільш ефективних алгоритмів внутрішнього сортування (наприклад,

Quicksort або Heapsort). Відсортовані частини, утворюючи серії, записуються в новий файл (у старий не можна).

Типове завдання

Реалізувати алгоритм зовнішнього сортування. Перевірити працездатність.

Пояснення до тексту програми. Початковий файл input.txt містить 1000 випадкових чисел. Сворюється 10 тимчасових файлів, у кожен з яких переписується приблизно 10 частина вмісту початкового файла. Їх вміст упорядковується алгоритмом внутрішнього сортування.

Текст програми

```
#include <queue>
#include <fstream>
#include <algorithm>
#include <climits>
#include <ctime>
#include <stdlib.h>
#define chunk_count 10

struct MinHeapNode
{ int element;
  int index;
};
struct Comparator
{  bool operator() (const MinHeapNode lhs, const MinHeapNode rhs) const
  { return lhs.element > rhs.element;
  }
};

void create_chunks(std::string filename, std::string outfilename,int chunk_size,
int achunk_count);
void merge_chunks(std::string filename, int chunk_size, int achunk_count);
void generate_input_file(std::string filename, int length);

int main()
{ int chunk_size = 1000;
  std::string input_filename = "input.txt";
  std::string output_filename = "output.txt";
  generate_input_file(input_filename, chunk_count * chunk_size);
  create_chunks(input_filename, output_filename, chunk_size, chunk_count);
  merge_chunks((char*)output_filename.c_str(), chunk_size, chunk_count);
  return 0;
}

void merge_chunks(std::string filename, int chunk_size, int achunk_count)
{ char str[10];
  std::ifstream input_files[chunk_count];
  for (int i = 0; i < achunk_count; i++)
  {  _itoa_s(i, str, 10);
     input_files[i] = std::ifstream(str);
  }
```

```
  std::ofstream output_file(filename);
  MinHeapNode harr[chunk_count];
  std::priority_queue<MinHeapNode, std::vector<MinHeapNode>, Comparator> pq;
  int i;
  for (i = 0; i < achunk_count; i++)
  { if (!(input_files[i] >> harr[i].element))
    break;
    harr[i].index = i;
    pq.push(harr[i]);
  }
  int count = 0;
  while (count != i)
  { MinHeapNode root = pq.top();
    q.pop();
    output_file << root.element << ' ';
    if (!(input_files[root.index] >> root.element))
    {  root.element = INT_MAX;
       count++;
    }
    pq.push(root);
  }
  for (int i = 0; i < chunk_count; i++)
    input_files[i].close();
  output_file.close();
}
void create_chunks(std::string filename, std::string outfilename, int chunk_size,
int achunk_count)
{ std::ifstream input_file(filename);
  char str[10];
  std::ofstream output_files[chunk_count];
  for (int i = 0; i < achunk_count; ++i)
  { _itoa_s(i, str, 10);
    output_files[i] = std::ofstream(str);
  }
  int* arr = new int[chunk_size];
  int output_file_index = 0;
  bool is_finised = false;
  while (!is_finised)
  { int current_chunk_size = 0;
    while (current_chunk_size < chunk_size)
    { if (!(input_file >> arr[current_chunk_size]))
      { is_finised = true;
        break;
      } else
       ++current_chunk_size;
    }
    std::sort(arr, arr + current_chunk_size);
    for (int i = 0; i < current_chunk_size; i++)
    { output_files[output_file_index] << arr[i] << ' ';
    }
    ++output_file_index;
  }
  for (int i = 0; i < chunk_count; i++)
    output_files[i].close();
  input_file.close();
}
void generate_input_file(std::string filename, int length)
{ std::ofstream input_file(filename);
  std::srand(time(NULL));
```

```
  for (int i = 0; i < length; i++)
    input_file << rand() << ' ';
  input_file.close();
}
```

Результат роботи програми

Склад створених файлів подано на рис. 16.1, вміст початкового файла – на рис. 16.2, вміст тимчасових файлів 0.txt та 9.txt – на рис. 16.3 та 16.4 відповідно, вміст файла з впорядкованими даними – на рис. 16.5.

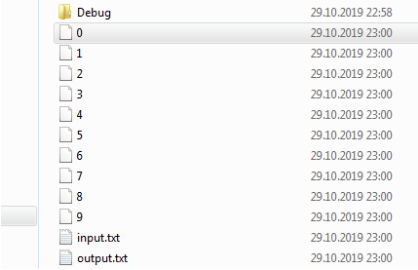


Рисунок 16.1 – Текстові файли проєкту

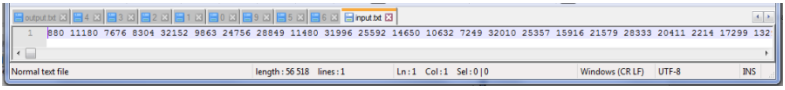


Рисунок 16.2 – Вміст початкового файла input.txt (56 КБ)

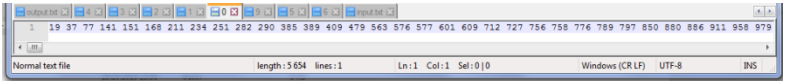


Рисунок 16.3 – Вміст тимчасового файла 0.txt (6 КБ)

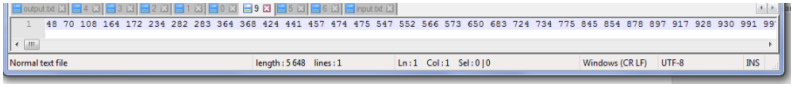


Рисунок 16.4 – Вміст тимчасового файла 9.txt (6 КБ)

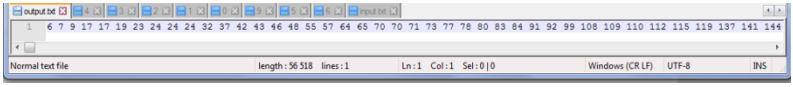


Рисунок 16.5 – Вміст результуючого файла output.txt (56 КБ)

Індивідуальні завдання

Розробити та налагодити програму, в якій реалізувати алгоритм зовнішнього сортування (табл. 16.1).

Таблиця 16.1 – Індивідуальні завдання

N	Назва алгоритму зовнішнього сортування
1	Пряме злиття, 3 файли
2	Природне злиття, 4 файли. Для створення серій – алгоритм Quicksort
3	Багатофазне сортування, 6 файлів. Для серій – алгоритм Quicksort
4	Збалансоване багатошляхове, 6 файлів. Для серій – Quicksort
5	Пряме злиття, 3 файли
6	Природне злиття, 3 файли. Для створення серій – алгоритм Heapsort
7	Багатофазне сортування, 8 файлів. Для серій – алгоритм Heapsort
8	Збалансоване багатошляхове, 8 файлів. Для серій – Heapsort
9	Багатофазне сортування, 10 файлів. Для серій – алгоритм Quicksort
10	Збалансоване багатошляхове, 10 файлів. Для серій – Quicksort
11	Природне злиття, 3 файли. Для створення серій – алгоритм Quicksort
12	Багатофазне сортування, 10 файлів. Для серій – алгоритм Heapsort
13	Збалансоване багатошляхове, 10 файлів. Для серій – Heapsort

Контрольні запитання

1. В чому принципова відмінність алгоритмів зовнішнього сортування від алгоритмів внутрішнього сортування?
2. Дайте опис алгоритму прямого злиття.
3. Які фази виділяють при сортуванні прямим злиттям?
4. Скільки файлів використовується в алгоритмі прямого злиття?
5. Опишіть алгоритм природного злиття.
6. Чи можна врахувати часткову впорядкованість початкового файла?
7. Скільки файлів може використовуватися в алгоритмі природного злиття?
8. Скільки файлів може використовуватися при сортуванні багатофазним злиттям?
9. Дайте опис алгоритму багатофазного злиття.

СПИСОК ЛІТЕРАТУРИ

1. Далека В.Д., Фурман І.О. та ін. Моделі та структури даних у системах автоматизованого керування: Підручник для ВНЗ / М-во освіти і науки України. – К., 2004 – 253 с.
2. Клакович Л. М., Левицька С. М., Костів О. В. Теорія алгоритмів: Навч. Посібник. – Львів: ЛНУ, 2008. – 140 с.
3. Глибовець М.М. Основи комп'ютерних алгоритмів. – К.: Вид. дім „КМ академія", 2003. – 452 с
4. Алгоритми та структури даних: Навчальний посібник / В.М.Ткачук. – ІваноФранківськ: Видавництво Прикарпатського національного університету імені Василя Стефаника, 2016. – 286 с.
5. Ставровський А.Б., Карнаух Т.О. Програмування. Перші кроки. – М.: Вид. дім "Вільямс", 2005. – 400 с.
6. Караванова Т.П. Інформатика: основи алгоритмізації та програмування: 777 задач, з рекомендаціями та прикладами К.: Генеза, 2009.- 285 с
7. Томас Г. Кормен, Чарлз Е. Лейзерсон, Роналд Л. Рівест, Кліфорд Стайн. Вступ до алгоритмів. — К. : К. І. С., 2019. — 1288 с.

ДОДАТКОВА ЛІТЕРАТУРА

8. Algorithm Design. Foundations, Analysis, and Internet Examples / Michael T. Goodrich and Roberto Tamassia. – N.Y.: John Wiley & Sons, Inc., 2014. – 816 p. 4. Clifford A. Shaffer. Data Structures and Algorithm Analysis. Editi
9. Методичні вказівки до виконання лабораторних робіт з дисципліни «Алгоритми та методи обчислень». Частина І «Алгоритми» для студентів ІІ курсу усіх форм навчання напряму «Комп'ютерна інженерія» вищих навчальних закладів. Харків: НТУ «ХПІ», 2014 – 46 с.
1. 10. Clifford A. Shaffer. Data Structures and Algorithm Analysis. Edition 3.2 (C++ Version). Copyright © 2009-2012 by Clifford A. Shaffer. – 596 p.

ДЛЯ ПРИМІТОК

Навчальне-методичне видання

БУЛЬБА Сергій Сергійович
БРЕЧКО Вероніка Олександрівна
ЛИСИЦЯ Дмитро Олександрович
БЕЛЬОРІН-ЕРРЕРА Олександра Михайлівна

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Навчально-методичний посібник
для студентів спец. 123
«Комп'ютерна інженерія».

Відповідальний за випуск проф. Олександр ЗАКОВОРОТНИЙ
Роботу до видання рекомендував проф. Микола ЗАПОЛОВСЬКИЙ

План 2024 р., поз. 25

Видавець:
Видавничий центр НТУ «ХПІ»
Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.
61002, Харків, вул. Фрунзе, 21

Самостійне електронне видання