Міністерство освіти і науки України
Рівненський державний гуманітарний університет

Володимир Сяський
Степанія Бабич

АЛГОРИТМИ І СТРУКТУРИ ДАНИХ

Навчальний посібник

Рівне – 2023

УДК 004.421:004.6 (075.8)
С 99

Рекомендовано до друку Вченою радою
Рівненського державного гуманітарного університету,
протокол № 5 від 27 квітня 2023 р.

Рецензенти:

Турбал Ю.В., доктор технічних наук, професор, Національний університет водного господарства та природокористування, м. Рівне;

Юскович-Жуковська В.І., кандидат технічних наук, доцент, Міжнародний економіко-гуманітарний університет імені академіка Степана Дем'янчука, м. Рівне.

Сяський В. А., Бабич С. М.

С 99 **Алгоритми і структури даних** : навч. посіб. Рівне : О. Зень, 2023. 124 с.

ISBN 978-617-601-439-3

Навчальний посібник присвячений вивченню теоретичних основ програмування, які передбачають конструювання структур даних і розробку, аналіз ефективності та програмну реалізацію алгоритмів їх обробки. Структури даних визначають спосіб організації даних у пам'яті комп'ютера, а алгоритми забезпечують виконання різних операцій над цими даними.

У посібнику розглянуто ряд фундаментальних задач програмування, що передбачають обробку структур даних, – пошук та сортування. Для кожного із алгоритмів проведено аналіз складності обчислень. Окрім статичних структур даних з прямим і послідовним доступом, також розглянуто алгоритми конструювання та обробки динамічних структур даних – списків. Усі алгоритми проілюстровані прикладами програмної реалізації на мові С++.

Посібник розрахований на студентів різних спеціальностей, які вивчають програмування і в майбутньому займатимуться конструюванням програмного забезпечення.

УДК 004.421:004.6 (075.8)

ISBN 978-617-601-439-3

ЗМІСТ

ПЕРЕДМОВА

Для багатьох програмістів, що розпочинали свою кар'єру ще з 90-их років XX століття, свого роду ідолами або іконами були Ніклаус Вірт, Дональд Кнут, Денніс Рітчі, Чарльз Гоар, Дональд Шелл. Вони були основоположниками сучасного програмування, які заклали теоретичні та практичні основи конструювання алгоритмів, аналізу їх складності та ефективної реалізації. У ті часи студенти буквально «вгризалися в граніт» текстового опису на перший погляд фантастичних, багатьом незрозумілих, алгоритмів обробки даних, щоб осягнути зміст тих дій, що якимось неймовірним чином забезпечували вирішення складних алгоритмічних задач. Хто з тодішніх випускників факультетів кібернетики чи інформатики та обчислювальної техніки, чи ще якихось інших, дотичних до інформаційних технологій, досі не пам'ятає бестселер Н. Вірта «Алгоритми та структури даних» або фундаментальний тритомник Д. Кнута «Мистецтво програмування для персональних ЕОМ»! На цих книгах виросло вже не одне покоління програмістів.

До речі, не варто засмучуватися, що серед згаданих основоположників інформатики та програмування немає прізвищ вітчизняних науковців. Були й свої, але вони відомі досягненнями в дещо інших галузях, зокрема: Андрій Єршов – радянський програміст і математик, один із творців «шкільної інформатики», корифей у галузі теорії й автоматизації програмування; Віктор Глушков – український радянський вчений, основоположник кібернетики, ініціатор і організатор реалізації науково-дослідних програм створення проблемно-орієнтованих програмно-технічних комплексів, нині його іменем названо всесвітньо відомий Інститут кібернетики імені В.М. Глушкова Національної Академії наук України.

Основою програмування є алгоритм – скінченний набір інструкцій або команд, виконання яких у визначеному порядку приводить від вхідних даних до відповідних вихідних даних (результату) за обмежений час. Алгоритми дозволяють формалізувати і автоматизувати процес обчислень, що особливо важливо при обмеженнях на такі параметри обчислювача (комп'ютера), як пам'ять, швидкодія тощо. Детальне розроблення алгоритму є важливою частиною процесу розв'язування різноманітних прикладних задач. У процес конструювання алгоритму і подальшої його програмної реалізації для реальної задачі входить усвідомлення ступеня її складності, з'ясування обмежень на вхідні дані, розбиття задачі на простіші підзадачі.

Структури даних визначають спосіб організації даних в оперативній пам'яті чи на диску комп'ютера. Алгоритми забезпечують виконання різних операцій над цими структурами даних. Звичайно, вибір алгоритмів обробки даних визначається моделлю структури даних. Адже часто аналогічні дії (операції, перетворення) над різними структурами даних передбачають використання алгоритмів неоднакової ефективності.

Саме тому вивчення теоретичних основ програмування є базою підготовки фахівців у галузі інформаційних технологій. Не дивно, що більшість Стандартів вищої освіти для спеціальностей галузі знань 12 Інформаційні технології буквально ставлять вимогу формування у здобувачів вищої освіти компетентностей пов'язаних із «… здатністю до логічного мислення, побудови логічних висновків, використання формальних мов і моделей алгоритмічних обчислень, проектування, розроблення й аналізу алгоритмів, оцінювання їх ефективності та складності, розв'язності та нерозв'язності алгоритмічних проблем для адекватного моделювання предметних областей і створення програмних та інформаційних систем».

В основу навчального посібника покладено декілька курсів лекцій, які під різними назвами багато років читалися авторами на факультеті математики та інформатики РДГУ для здобувачів вищої освіти спеціальностей 113 Прикладна математика, 121 Інженерія програмного забезпечення, 122 Комп'ютерні науки, 014 Середня освіта (Математика), 014 Середня освіта (Інформатика) та 015 Професійна освіта (Цифрові технології).

Перша частина посібника присвячена вирішенню фундаментальних задач програмування, що передбачають обробку структур даних, – пошуку та сортуванню. У якості моделей структур даних з прямим доступом використовуються масиви, а для структур даних з послідовним доступом – файли послідовного доступу. Для кожного з алгоритмів обробки структури даних і для задачі пошуку, і для задачі сортування проведено детальний аналіз складності обчислень по «важких» операціях.

У другій частині посібника розглядаються алгоритми конструювання і обробки динамічних структур даних – списків. Зокрема, детально висвітлені особливості програмної реалізації операцій по обробці лінійних однонапрямлених та двонапрямлених списків; циклічних списків; розгалужених багатозв'язних списків на прикладі бінарних дерев.

Для усіх алгоритмів наведено фрагменти програмного коду на мові програмування C++, надаються рекомендації щодо їх використання.

ВСТУП
ПОНЯТТЯ ПРО СТРУКТУРИ ДАНИХ ТА АЛГОРИТМИ ЇХ ОБРОБКИ

У більшості мов програмування використовуються складені типи даних, що передбачають структурування з елементів *базового типу*. У таких мовах є багато спільних або подібних між собою складених типів даних, хоча вони можуть мати різні назви.

Усі структури даних можна поділити на окремі категорії за певними ознаками. Одним із різновидів є так звані регулярні складені типи – структури даних, елементи яких мають однаковий базовий тип (*масиви, динамічні списки, типізовані файли*). До регулярних складених типів традиційно також відносять *рядки символів* та *множини*. Деякі мови програмування, зокрема Python, оперують іншими складеними типами, які можна вважати умовно регулярними, оскільки їх елементи можуть мати різні базові типи, – це *списки, кортежі, словники* тощо.

Крім такого поділу за базовим типом, складені структури даних розрізняють за способом доступу до окремих елементів:

- *структури даних з прямим доступом (СДПрД)* – складені структури даних, елементи яких індексовані, і доступ до них здійснюється практично миттєво за їх індексним номером (наприклад, масиви);
- *структури даних з послідовним доступом (СДПоД)* – складені структури даних, елементи яких не мають чітко визначених індексних номерів, і доступ до них здійснюється послідовним перебором усіх елементів від початку структури даних (наприклад, списки або файли).

СДПрД традиційно розміщуються у так званому *програмному стеку*, що є частиною достатньо швидкої, але обмеженої за розмірами *внутрішньої оперативної пам'яті*. Тому алгоритми обробки СДПрД прийнято називати *внутрішніми*.

СДПоД традиційно розміщуються у дискових файлах, які є частиною практично необмеженої, але відносно повільної *зовнішньої пам'яті*. Тому алгоритми обробки СДПоД називають *зовнішніми*.

Крім такого поділу, що визначається місцем розміщення складених структур даних, алгоритми обробки або методи розрізняють за їх швидкодією в залежності від довжини структури даних:

- *прямі алгоритми (методи)* – алгоритми, які прості в програмній реалізації; вони за один етап повністю «обробляють» один поточний елемент, а решту елементів практично «не зачіпають». Такі методи переважно застосовують до так званих *коротких* структур даних, у яких число елементів відносно невелике, наприклад, у межах ***100***;
- *швидкі алгоритми (методи)* – алгоритми, які значно складніші в програмній реалізації; вони за один етап не лише «обробляють» один поточний елемент, але й суттєво «опрацьовують» решту елементів, тобто «підготовлюють» їх до остаточної обробки на наступних етапах. Такі методи

переважно застосовують для так званих *довгих* структур даних, у яких кількість елементів є значною – більше ***1000***.

Звичайно, такий поділ на короткі та довгі структури даних, як і на прямі та швидкі алгоритми їх обробки, є достатньо умовним. Можна також говорити про таку собі середню довжину структури даних, при якій і прямі, і швидкі методи даватимуть результат майже за однаковий час. Більш того, окремі алгоритми дійсно займають таку проміжну позицію по швидкодії.

При класифікації алгоритмів обробки структур даних за швидкодією насправді не оцінюється реальний час виконання програми у часових одиницях (мілісекунди, секунди, хвилини, години і т.д.). Адже час роботи будь-якої програми залежить від апаратних можливостей самого комп'ютера, на якому вона виконується. Тому в теорії програмування і в обчислювальній практиці *ефективність* алгоритмів, а точніше – їх програмної реалізації, оцінюють за наступними критеріями:

- *часова складність* – кількість виконуваних ключових операцій;
- *ємнісна складність* – кількість використовуваної пам'яті.

Для встановлення часової складності алгоритму чи програми оцінюється кількість *ключових операцій для відповідної задачі*. Наприклад, для арифметичних задач – це операції додавання, віднімання, множення, ділення, а для задач обробки структур даних – це операції порівняння значень та переміщення або присвоєння значень. Більш того, розрізняють так звані *«важкі»* та *«легкі»* ключові операції задачі. *«Важкі»* – це такі ключові операції розглядуваної задачі, що є значно складнішими для виконання процесором, у порівнянні з іншими операціями цієї ж задачі. Простіші для виконання процесором операції прийнято називати *«легкими»*. Для арифметичних задач – «важкими» є операції множення і ділення, а «легкими» – додавання і віднімання. Для задач обробки структур даних «важкими» вважаються операції порівняння і присвоєння над елементами базового типу. Тоді «легкими» є операції порівняння і присвоєння над індексними номерами елементів.

З метою обробки структур даних найчастіше застосовують такі види задач: *пошук, впорядкування*.

Задача пошуку передбачає встановлення позиції входження деякого елемента (ключа) чи підпослідовності елементів або їх відсутності у структурі даних. Традиційно розрізняють такі різновиди задачі пошуку:

- пошук першого по порядку входження заданого ключа;
- пошук усіх входжень заданого ключа;
- пошук підпослідовності ключів.

Неважко зрозуміти, що пошук усіх входжень заданого ключа – це повторення пошуку першого по порядку його входження після позиції попереднього входження в структуру даних. Тому обмежимося лише пошуком першого входження елемента та пошуком першого входження підпослідовності елементів.

Задача впорядкування або *сортування* передбачає перестановку елементів у структурі даних так, щоб вони задовольняли деякій умові – *критерію впорядкованості*. Не зменшуючи загальності, для випадку структури даних із N елементів деякого базового типу традиційно застосовують такі критерії впорядкованості:

1) $a_i < a_{i+1}$, $i = \overline{1, N-1}$ – по зростанню;
2) $a_i > a_{i+1}$, $i = \overline{1, N-1}$ – по спаданню.

Якщо в структурі даних є декілька елементів з однаковими значеннями, то слід використовувати нестрогі умови:

3) $a_i \le a_{i+1}$, $i = \overline{1, N-1}$ – по неспаданню;
4) $a_i \ge a_{i+1}$, $i = \overline{1, N-1}$ – по незростанню.

Зрозуміло, що сортування часто використовують у якості додаткової або попередньої обробки структури даних з метою суттєвого спрощення й пришвидшення наступного пошуку у впорядкованій структурі даних.

Упорядкування структур даних можна проводити за різними визначальними принципами: *включення (вставка), вибір, обмін, злиття*. Для сортування структур даних з прямим доступом (масиви) з однаковим успіхом можна застосовувати всі принципи, а структури даних з послідовним доступом (файли, списки) сортуються переважно алгоритмами злиття.

При цьому відмінні способи сортування можуть мати різну ефективність для різних випадків початкового порядку елементів у структурі даних. Традиційно виділяють такі варіанти початкового порядку елементів:

- *«хороші»* структури даних – переважна більшість елементів задовольняють умові впорядкованості (приклад: ***2, 3, 4, 5, 6, 7, 8, 9, 1***);
- *«погані»* структури даних – переважна більшість елементів не задовольняють умові впорядкованості (приклад: ***1, 9, 8, 7, 6, 5, 4, 3, 2***);
- з довільним випадковим порядком – лише частина елементів задовольняють умові впорядкованості (приклад: ***2, 1, 4, 3, 9, 6, 5, 8, 7***).

В обчислювальній практиці з однаковим успіхом широко використовуються різні варіанти реалізації кожного із способів сортування: і більш простіші, але не такі високоефективні – прямі алгоритми, та їх потужніші аналоги – швидкі алгоритми.

ЧАСТИНА 1
ПОШУК ТА СОРТУВАННЯ

РОЗДІЛ 1
ЗАДАЧА ПОШУКУ

1.1. ПОШУК ЕЛЕМЕНТА У СТРУКТУРІ ДАНИХ

Не зменшуючи загальності, розглядатимемо структуру даних у вигляді масиву з ***N*** елементів деякого базового типу, наприклад цілих чисел.

Пошук елемента передбачає встановлення позиції входження в масив ***a*** деякого елемента зі значенням ***x*** цього самого базового типу, або фіксація факту його відсутності в масиві.

Розглядатимемо задачу пошуку першого по порядку входження. Пошук усіх входжень – це повторення операції пошуку першого входження після останнього знайденого.

1.1.1. АЛГОРИТМ ПРЯМОГО ПОШУКУ ЕЛЕМЕНТА У МАСИВІ

Ідея методу. Найбільш простим і очевидним способом пошуку є послідовне співставлення заданого значення ***x*** з елементами масиву ***a[i],*** починаючи з першого з приростом індексного номера на одиницю. Такий пошук припиняється в одному з двох випадків:

1) знайдено перше по порядку входження заданого значення ***x***, тобто $(\exists\, i, 0 \le i < N), (\forall\, j, 0 \le j < i) \Rightarrow [(a[i] = x) \& (a[j] \ne x)]$, – це позитивний результат;
2) після чергового збільшення індексу ***i*** має місце вихід за межі масиву, тобто ***i=N***, – це негативний результат.

Оскільки заперечення диз'юнкції є кон'юнкція заперечень, то умовою продовження пошуку буде $(i < N) \& (a[i] \ne x)$.

Програмна реалізація. Реалізація алгоритму на мові С++ може мати вигляд:

```
int i=0;
while (i<N && a[i]!=x) i++;
if (i<N) cout << "є в позиції " << i << endl;
    else cout << "немає" << endl;
```

Аналіз складності. «Важкими» операціями (*hard*) для задачі пошуку є операції порівняння між елементами базового типу, а «легкими» (*easy*) є операції порівняння між індексами елементів. «Важка» операція ***a[i]!=x*** має

місце у виразі умови продовження циклу ***while***. «Легкі» операції ***i<N*** використовуються як в умові продовження циклу, так і при додатковій перевірці умови зупинки.

У найкращому випадку вхідних даних, коли шуканий елемент є відразу першим у масиві, виконується така кількість «важких» і «легких» операцій порівняння:

$$C_{min}^{hard} = 1; \quad C_{min}^{easy} = 1 + 1 = 2.$$

У найгіршому випадку вхідних даних, коли шуканий елемент взагалі відсутній у масиві, виконується така кількість «важких» і «легких» операцій порівняння:

$$C_{max}^{hard} = N; \; C_{max}^{easy} = (N+1) + 1 = N + 2.$$

Як видно, кількість «легких» операцій у будь-якому випадку більша ніж кількість «важких». Якщо базовий тип елементів структури даних значно перевищує за розміром тип індексів, то «легкі» операції не вносять суттєвого впливу на загальний час роботи алгоритму. Але якщо базовий тип структури даних є співрозмірним або меншим від типу індексів, то «легкі» операції насправді є складнішими від «важких», і вони суттєво впливають на швидкодію та ефективність алгоритму. Тому нехтувати ними не можна.

1.1.2. Модифікація алгоритму прямого пошуку елемента у масиві

Ідея методу. Суттєве покращення ефективності алгоритму прямого пошуку можливе за рахунок відмови від «легких» операцій. При цьому потрібно гарантувати коректну зупинку циклу у випадку негативного результату.

З цією метою масив розширюється додатковим ***N+1***-им елементом:

basetype a[N+1].

Цей додатковий елемент виконуватиме функцію «бар'єра», оскільки матиме шукане значення ***x***. Така витрата пам'яті є незначною в порівнянні з покращенням швидкодії. Тепер циклічні порівняння шуканого значення з елементами масиву гарантовано припиняться в будь-якому випадку вхідних даних. Для встановлення остаточного результату пошуку потрібно виконати лише одну «легку» операцію порівняння поточного індексу.

Програмна реалізація. Реалізація алгоритму на мові C++ може мати вигляд:

```
a[N]=x;
int i=0;
while (a[i]!=x) i++;
if (i<N) cout << "є в позиції " << i << endl;
    else cout << "немає" << endl;
```

Аналіз складності. У найкращому випадку вхідних даних, коли шуканий елемент є відразу першим у масиві, виконується така кількість «важких» і «легких» операцій порівняння:

$$C_{min}^{hard} = 1; \quad C_{min}^{easy} = 1.$$

У найгіршому випадку вхідних даних, коли шуканий елемент відсутній у масиві, виконується така кількість «важких» і «легких» операцій порівняння:

$$C_{max}^{hard} = N + 1; \qquad C_{max}^{easy} = 1.$$

Як видно, кількість «легких» операцій у цьому випадку зведена до мінімуму.

1.1.3. Алгоритм бінарного пошуку елемента у впорядкованому масиві

Ідея методу. Як видно з описів попередніх алгоритмів, у випадку відсутності додаткової інформації про характер розміщення елементів у СД ніяким чином не можна пришвидшити пошук – потрібно виконувати послідовне порівняння елементів із шуканим значенням.

Якщо ж відомо, що СД є впорядкована за деякою ознакою, наприклад по неспаданню, то процес пошуку можна значно пришвидшити, користуючись методом поділу відрізка пополам. З цією метою слід виконувати такі дії:

1) довільним чином обирається внутрішній елемент ***a[m]*** розглядуваної частини масиву; початково аналізується весь масив, у цьому випадку $0 \le m < N$;
2) обраний елемент співставляється із шуканим значенням ***x***:
   a) якщо $a[m] = x$, то це точне попадання в шукане значення;
   b) якщо $a[m] < x$ та враховуючи, що всі елементи лівіше від ***a[m]*** не перевищують його, то серед них шуканого значення немає, тому всю цю частину масиву разом з ***a[m]*** можна відкинути;
   c) якщо $a[m] > x$ та враховуючи, що всі елементи правіше від ***a[m]*** не менші від нього, то серед них шуканого значення немає, тому всю цю частину масиву разом з ***a[m]*** можна відкинути;
3) послідовність кроків 1), 2a), 2b), 2c) знову застосовується до тієї частини масиву, що залишилася; такий повторюваний процес завершується в одному з двох випадків:
   - чергове порівняння $a[m] = x$ є істинним – це позитивний результат;
   - після чергового відкидання частини масиву не залишається жодного елемента для порівняння – це негативний результат.

Виникає питання про довільність вибору елемента ***a[m]***. Очевидно, щоб процес пошуку пришвидшити, потрібно щоразу відкидати якомога більшу частину масиву, де заданого значення немає. Тому доцільно обирати

центральний елемент у розглядуваній частині масиву. Адже при цьому щоразу з однаковою ймовірністю відкидатиметься не менше половини від останньої розглядуваної частини. Звідси і назва методу – бінарний пошук, тобто пошук з діленням пополам.

Програмна реалізація. Реалізація алгоритму на мові С++ може мати вигляд:

```
int L=0, R=N-1, m=(L+R)/2;
while (L<=R && a[m]!=x)
 {
  if (a[m]<x) L=m+1; else R=m-1;
  m=(L+R)/2;
 }
if (L<=R) cout << "є в позиції " << m << endl;
      else cout << "немає" << endl;
```

Аналіз складності. Найкращим випадком вхідних даних є варіант масиву, у якому шуканий елемент займає центральну позицію. Адже тоді виконується лише один бінарний поділ, який відразу дає позитивний результат. При цьому виконується така кількість «важких» і «легких» операцій порівняння:

$$C_{min}^{hard} = 1; \quad C_{min}^{easy} = 1 + 1 = 2.$$

Очевидно, що найгіршим випадком вхідних даних знову є варіант масиву, у якому шуканий елемент відсутній. При цьому виконується максимальна кількість бінарних поділів, поки розглядувана частина не стане порожньою. Кількість етапів поділів пополам визначається логарифмічною залежністю:

$$K_{етапів} = \lfloor log_2 N \rfloor + 1,$$

де у квадратних дужках розуміється ціла частина від логарифма.

На кожному етапі поділу виконується по дві «важкі» операції (одна – в умові циклу ***while***, і одна – у тілі цього циклу) і по одній «легкій» операції (в умові циклу ***while***). Окрім цього, ще одна «легка» операція має місце при перевірці остаточного результату пошуку. Таким чином, у найгіршому випадку вхідних даних виконується така кількість «важких» і «легких» операцій порівняння:

$$C_{max}^{hard} = 2(\lfloor log_2 N \rfloor + 1); \quad C_{max}^{easy} = (\lfloor log_2 N \rfloor + 1) + 1 = \lfloor log_2 N \rfloor + 2.$$

Як видно, часова складність бінарного пошуку значно краща в порівнянні з розглянутими раніше прямими методами.

Характерною особливістю даного методу є майже найгірша часова складність і у випадках із позитивним результатом, коли шуканий елемент у

масиві займає такі позиції: перший, останній, передує центральному, наступний після центрального.

Зауваження. Хоча бінарний пошук є досить швидким алгоритмом, проте він не гарантує знаходження саме першого за порядком входження заданого значення, а дає якийсь із можливо декількох підряд однакових елементів. Наприклад:

int a[10]={1, 2, 3, 3, 3, 3, 3, 3, 4, 5}, x=3;

У результаті буде знайдено третю за порядком **трійку** – елемент із номером **4**, хоча перша за порядком **трійка** у масиві має номер **2**.

1.1.4. Модифікація алгоритму бінарного пошуку елемента у впорядкованому масиві

Удосконалення бінарного пошуку має на меті забезпечити встановлення саме першого за порядком, а не довільного входження заданого значення. Окрім цього, – це дозволить зменшити кількість «важких» операцій удвічі.

Ідея методу. Модифікація попереднього алгоритму передбачає відмову від дострокового завершення процесу порівнянь при точному попаданні в елемент із шуканим значенням. Це дозволить відмовитися від «важкої» операції порівняння ***a[m]!=x*** в умові циклу ***while***. Звичайно, тоді постане питання про коректну завершуваність оператора циклу у випадку позитивного остаточного результату.

Проаналізуємо результати порівняння довільно обраного елемента ***a[m]*** із заданим значенням ***x***, але розглянемо не тривіткове, а двовіткове розгалуження:

- якщо ***a[m]<x***, то, як і в попередньому випадку, частину масиву від його початку до елемента ***a[m]*** включно можна відкинути, тобто виконати переприсвоєння ***L=m+1***;
- якщо ***a[m]≥x***, то елемент ***a[m]*** вже не можна відкидати, бо він може співпадати із заданим значенням ***x***. У випадку декількох підряд однакових елементів поки що невідомо, який саме він за порядком. Якщо після нього слідують такі самі за значенням елементи, то ними вже можна нехтувати. Тому в даному випадку варто виконати переприсвоєння ***R=m***.

Виходячи з властивості операції цілочисельного ділення на **2** для ***m=(L+R)/2***, справедливою буде подвійна нерівність ***L≤m<R***. Тому оператор присвоєння ***L=m+1*** гарантовано зміщуватиме ліву межу хоча б на один елемент вправо, і оператор ***R=m*** теж гарантовано зміщуватиме праву межу вліво. Таким чином, розглядувана частина масиву щоразу звужуватиметься чи зліва, чи справа, і це забезпечить коректну завершуваність циклу ***while***. Окрім цього, у випадку декількох підряд однакових елементів із шуканим значенням права межа ***R*** розглядуваної частини буде послідовно зміщуватися вліво і зупиниться саме на першому за порядком входженні.

Варто зауважити, що така модифікація також передбачатиме заміну умови завершення роботи алгоритму. Тепер процес пошуку в будь-якому випадку припиняється, якщо розглядувана частина масиву «стягується» до одного єдиного елемента, тобто при умові ***L=R***. Цей єдиний елемент, без різниці, чи ***a[L]***, чи ***a[R]***, буде або дійсно першим за порядком серед можливо декількох однакових із шуканим значенням, або якимось іншим. Тому потрібно виконати ще одну перевірку з «важкою» операцією порівняння зафіксованого елемента із заданим значенням ***x***.

Програмна реалізація. Реалізація алгоритму на мові C++ може мати вигляд:

```
int L=0, R=N-1, m;
while (L<R)
 {
   m=(L+R)/2;
   if (a[m]<x) L=m+1; else R=m;
 }
if (a[L]==x) cout << "є в позиції " << L << endl;
        else cout << "немає" << endl;
```

Аналіз складності. Для даного алгоритму не можна виділити кращі чи гірші випадки вхідних даних, оскільки цикл не завершується достроково при точному попаданні в шуканий елемент. Таким чином, загальна кількість «важких» і «легких» операцій у будь-якому випадку буде:

$$C_{min}^{hard} = C_{max}^{hard} = \lfloor log_2 N \rfloor + 1; \quad C_{min}^{easy} = C_{max}^{easy} = \lfloor log_2 N \rfloor + 1.$$

Окрім такої достатньо високої часової ефективності, модифікований бінарний пошук характеризується тим, що забезпечує встановлення першого за порядком входження елемента серед можливо декількох підряд однакових.

1.2. ПОШУК ПІДПОСЛІДОВНОСТІ ЕЛЕМЕНТІВ У СТРУКТУРІ ДАНИХ

Не зменшуючи загальності, розглядатимемо пошук у послідовності, що моделюється масивом із N елементів довільного базового типу. При цьому шукана підпослідовність – це теж масив із M елементів цього самого базового типу:

basetype a[N], b[M].

Для забезпечення коректності постановки задачі необхідне виконання умови $N \geq M$, оскільки не можна шукати щось довше в коротшому.

Послідовність a, у якій здійснюється пошук, називають *базовою послідовністю* або просто *базою*. Підпослідовність b, яка шукається, називається *образом*.

Таким чином, задача пошуку першого входження образа в базу передбачає встановлення найменшого з можливих значень індекса k по базі, починаючи з якого елементи бази послідовно співпадають з усіма елементами образа. Якщо такий індекс k існує, тобто справедлива умова

$$(\exists\, k, 0 \leq k \leq N-M) \Rightarrow \left[a[k+j] = b[j],\ j = \overline{0, M-1}\right] \&$$
$$(\exists\, k, 0 \leq k \leq N-M), (\forall\, i, 0 \leq i < k), (\exists\, j, 0 \leq j < M) \Rightarrow \left[a[i+j] \neq b[j]\right], \quad (1)$$

то це – позитивний результат пошуку.

У протилежному випадку, тобто за умови

$$(\forall\, k, 0 \leq k \leq N-M), (\exists\, j, 0 \leq j < M) \Rightarrow \left[a[k+j] \neq b[j]\right], \quad (2)$$

має місце негативний результат пошуку.

1.2.1. Прямий пошук підпослідовності у послідовності

Ідея методу. Найбільш простим та очевидними способом пошуку є послідовне співставлення елементів образа і бази починаючи від початку. При рівності чергової пари елементів $a[k+j]=b[j]$, $0 \leq k \leq N-M$, $0 \leq j < M$ відбувається перехід до наступної пари елементів, тобто виконується приріст індекса j. Якщо для деякої пари елементів має місце неспівпадання $a[k+j] \neq b[j]$, $0 \leq k \leq N-M$, $0 \leq j < M$, то образ умовно «зсувається» відносно бази на одну позицію, тобто виконується приріст індекса k, і співставлення елементів знову починається від початку образа.

Такий повторюваний процес співставлень і можливих «зсувів» образа відносно бази припиняється в одному з двох випадків:

1) усі елементи образа співпали з відповідними елементами бази, тобто $j=M$, – це позитивний результат;
2) після чергового неспівпадання елементів і «зсуву» образ виходить за межі бази справа, тобто $k>N-M$, – це негативний результат.

Виходячи з цього, умовою продовження пошуку буде кон'юнкція виду $(k \le N - M) \& (j < M)$.

Приклад покрокового виконання. Для ілюстрації покрокового виконання пари елементів, що співпадали, – підкреслені, а неспівпадаючі пари елементів – ~~закреслені~~:

```
int a[12]={1,2,3,1,2,3,4,1,2,3,4,5};
int  b[5]={1,2,3,4,5};
```

```
1 2 3 ~~1~~ 2 3 4 1 2 3 4 5
1 2 3 ~~4~~ 5
1 ~~2~~ 3 1 2 3 4 1 2 3 4 5
  ~~1~~ 2 3 4 5
1 2 ~~3~~ 1 2 3 4 1 2 3 4 5
    ~~1~~ 2 3 4 5
1 2 3 1 2 3 4 ~~1~~ 2 3 4 5
      1 2 3 4 ~~5~~
1 2 3 1 ~~2~~ 3 4 1 2 3 4 5
        ~~1~~ 2 3 4 5
1 2 3 1 2 ~~3~~ 4 1 2 3 4 5
          ~~1~~ 2 3 4 5
1 2 3 1 2 3 ~~4~~ 1 2 3 4 5
            ~~1~~ 2 3 4 5
1 2 3 1 2 3 4 1 2 3 4 5
              1 2 3 4 5     stop, yes!
```

Програмна реалізація. Реалізація алгоритму на мові С++ може мати вигляд:

```
int k=0, j=0;
while (j<M && k<=N-M)
 {
  j=0;
  while (j<M && a[k+j]==b[j]) j++;
  k++;
 }
if (j==M) cout << "є в позиції " << k-1 << endl;
      else cout << "немає" << endl;
```

Аналіз складності. Оскільки програмна реалізація є складнішою в порівнянні з алгоритмами пошуку елемента (два вкладені цикли), то важко точно оцінити кількість «легких» операцій. Тому обмежимось оцінкою кількості «важких» операцій.

Найкращих варіантів вхідних даних може бути декілька:

- образ входить у базу відразу на початку, і він значно коротший від бази; у цьому випадку $C_{min}^{hard} = M$, наприклад:

```
a[10]={1,2,3,4,5,6,7,8,9,0};
b[3] ={1,2,3};
```

- образ не входить у базу, він співрозмірний із базою, і при цьому неспівпадання елементів щоразу фіксується на першому елементі образа; у цьому випадку $C_{min}^{hard} = N - M + 1$, наприклад:

```
a[10]={1,2,3,4,5,6,7,8,9,0};
b[8] ={0,2,3,4,5,6,7,8}.
```

У найгіршому випадку на кожному етапі має виконуватися максимальна кількість операцій порівняння, і при цьому виконуються всі можливі зсуви образа відносно бази; це такий варіант вхідних даних, коли на кожному етапі всі елементи образа, окрім останнього, співпадають із відповідними елементами бази, а неспівпадання щоразу фіксується на останньому елементі образа; у цьому випадку $C_{max}^{hard} = M(N - M + 1)$, наприклад:

```
a[10]={1,1,1,1,1,1,1,1,1,1};
b[5] ={1,1,1,1,2};
```

Встановимо тепер найгірший із найгірших випадків. Для цього дослідимо на екстремум функцію $C_{max}^{hard}(M) = M(N - M + 1) \to max$. Продиференціюємо по аргументу M і прирівняємо до нуля:

$$C_{max}^{hard\,\prime}(M) = \left(MN - M^2 + M\right)' = N - 2M + 1 = 0 \Rightarrow M_{extr} = \frac{(N+1)}{2}.$$

Тому $C_{max\,max}^{hard}(M_{extr}) = \left(\frac{N+1}{2}\right)^2$.

Отже, найгіршим із найгірших випадків є варіант вхідних даних, коли, окрім усього вищезазначеного, образ удвічі коротший від бази. Наприклад, при $N=10$, $M=5$ кількість операцій $C_{max\,max}^{hard} = M(N - M + 1) = 5 \cdot 6 = 30$.

Значно кращими з гірших випадків є наступні варіанти вхідних даних:

- образ дуже короткий – тоді він швидко «рухається» вздовж бази, бо усі етапи короткі:

```
a[10]={1,1,1,1,1,1,1,1,1,1};
b[3] ={1,1,2};
```

$C_{max}^{hard} = M(N - M + 1) = 3 \cdot (10 - 3 + 1) = 24$;

- образ співрозмірний із базою – тоді, хоч кожен етап довгий, проте «зсувів» мало:

```
a[10]={1,1,1,1,1,1,1,1,1,1};
b[8] ={1,1,1,1,1,1,1,2};
```

$$C_{max}^{hard} = M\left(N - M + 1\right) = 8 \cdot \left(10 - 8 + 1\right) = 24 .$$

1.2.2. Модифікація прямого пошуку підпослідовності

Недоліком прямого пошуку підпослідовності є те, що при черговій невдачі образ «зсувається» відносно бази лише на одну позицію. Якщо такій невдачі передувала велика кількість співпадань елементів, то доцільно цю додаткову інформацію про характер співпадань якимось чином використовувати на наступних етапах. Це дозволило б відмовитися від багаторазових «повернень» назад і повторних «проходів» по тих самих елементах бази, які вже аналізувалися на попередніх етапах. Напрошується ідея зафіксувати в базі позицію останнього неспівпадання і «зміщувати» образ відносно бази на деяку відстань.

Одним із способів реалізації такої ідеї є «зсув» образа відносно бази не на один елемент вправо, а відразу в саму позицію неспівпадання. Очевидно, що якщо неспівпадання має місце на першому елементі образа, то «зсув» має виконуватися на одну позицію. Розглянемо декілька прикладів:

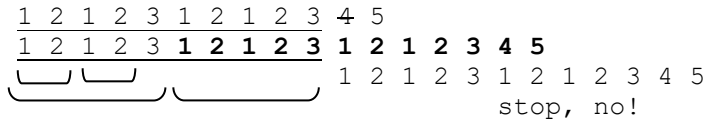
```
1) 1 2 3 ~~1~~ 2 3 4 1 2 3 4 5
   1 2 3 ~~4~~ 5
   1 2 3 1 2 3 4 ~~1~~ 2 3 4 5
         1 2 3 4 ~~5~~
   1 2 3 1 2 3 4 1 2 3 4 5
               1 2 3 4 5        stop, yes!

2) 1 1 1 ~~1~~ 1 1 1 1 1 2 3 4 5
   1 1 1 ~~2~~ 3 4 5
   1 1 1 1 1 1 ~~1~~ 1 1 2 3 4 5
         1 1 1 ~~2~~ 3 4 5
   1 1 1 1 1 1 1 1 1 2 3 4 5
               1 1 1 2 3 4 5      stop, yes!
```

Проте можуть бути і такі випадки:

```
3) 1 1 1 ~~1~~ 1 1 1 1 2 3 4 5
   1 1 1 ~~2~~ 3 4 5
   1 1 1 1 1 1 ~~1~~ 1 2 3 4 5
         1 1 1 ~~2~~ 3 4 5
   1 1 1 1 1 1 1 1 2 3 4 5
               1 1 1 2 3 4 5      stop, no!

4) 1 2 1 2 3 1 2 1 2 3 ~~1~~ 2 1 2 3 4 5
```



Проаналізуємо тепер розглянуті приклади:

- у випадках 1 і 2 швидко і вірно зафіксовано позицію входження образа в базу;
- у випадках 3 і 4 зафіксовано виходи образів за межі баз (фактично – це негативні результати), хоча насправді вони входять у бази (виділено жирним шрифтом).

Таким чином, розглянута модифікація алгоритму пошуку підпослідовності не є результативною. Не складно переконатися, що дефект алгоритму виявляється за наступних умов:

a) на початку образа мають місце повторення однакових елементів або груп із декількох елементів; при цьому важливими є повторення більш довших груп (приклад 4);
b) у базі на початку деякого етапу має місце повторення тих самих елементів або груп елементів, що й в образі;
c) кількість повторень елементів чи груп елементів у базі не кратна кількості їх повторень в образі.

Одним із способів вирішення такої проблеми й одночасно підвищення ефективності алгоритму пошуку підпослідовності є зміщення образа відносно бази з врахуванням повторюваних входжень груп елементів в образі та в базі.

1.2.3. Алгоритм Кнута-Морріса-Пратта (КМП)

Ідея методу. У 1977 р. Дональд Кнут, Джеймс Морріс і Вон Пратт запропонували ефективний алгоритм пошуку підпослідовності в послідовності – алгоритм КМП, який має середню часову складність **O(N+M)**, і при цьому не проявляє жодних дефектів, тобто є результативним.

Як і в розглянутому вище варіанті модифікації прямого пошуку, «зсуви» образа здійснюються не на одиничні, а на значно більші відстані, однак не завжди в точку неспівпадання елементів.

Встановлення величини «зсуву» здійснюється на основі попереднього аналізу образу, що дозволяє виконувати зміщення на великі відстані навіть при наявності повторюваних груп елементів і без втрати результативності. Це є окрема фаза загального алгоритму, яка виконується однократно перед початком співставлення елементів образа і бази.

В основу попереднього аналізу образа покладено підрахунок довжин особливих підпослідовностей елементів образа, які в ньому повторно зустрічаються. Подібно до морфології, що є розділом граматики і вивчає будову слова, у теорії формальних мов, що є частиною теорії програмування,

розглядаються такі структурні елементи як префікс і суфікс. *Префіксом* вважається деяка підпослідовність елементів базової послідовності від її початку, яка не співпадає із самою базою. Аналогічно, *суфіксом* вважається деяка підпослідовність елементів базової послідовності від її кінця, яка теж не співпадає з базою. У префікс не входить останній елемент, а в суфікс не входить перший елемент бази. Таким чином, база з одного елемента не має префікса і суфікса.

Враховуючи прийняті визначення, розглянемо деяку послідовність елементів довільного базового типу

$$\underbrace{b_1, b_2, \ldots, b_{k-1}, b_k}_{\text{префікс}}, b_{k+1}, \ldots, b_{M-k}, \underbrace{b_{M-k+1}, b_{M-k+2}, \ldots, b_{M-1}, b_M}_{\text{суфікс}}. \quad (1)$$

Нас цікавитиме *префікс максимальної довжини*, який поелементно співпадає із суфіксом такої ж довжини:

$b_1 = b_{M-k+1}$, $b_2 = b_{M-k+2}$, …, $b_k = b_M$.

Префікс і суфікс можуть не перетинатися, можуть частково перекриватися, проте вони не можуть повністю накладатися, тобто бути одними і тими самими елементами. Як вже було зазначено, одноелементна послідовність не може мати префікса, або, іншими словами, її префікс має нульову довжину.

Згідно з алгоритмом КМП попередній аналіз образа передбачає обчислення довжин префіксів для всіх підпослідовностей образа від його початку зі зростанням їх довжини на одиницю:

$$\begin{cases} b_1; \\ b_1, b_2; \\ \ldots \\ b_1, b_2, \ldots, b_k; \\ \ldots \\ b_1, b_2, \ldots, b_k, \ldots, b_{M-1}, b_M. \end{cases} \quad ; \quad (2)$$

Для встановлення префіксів більш довгих підпослідовностей використовуються результати аналізу попередніх більш коротких підпослідовностей. Тому довжини префіксів кожної з підпослідовностей у (2) доцільно зберігати в додатковому числовому масиві такого самого розміру, що і образ:

int p[M].

Це є додаткова витрата пам'яті, яку слід враховувати при пошуку особливо довгих образів.

Розглянемо приклад покрокового обчислення довжин префіксів:

```
int b[16]={1,2,1,2,1,2,3,1,2,1,2,1,2,3,4,5};
```

$prefix(1)=0 \Rightarrow p_1=0$;
$prefix(1,2)=0 \Rightarrow p_2=0$;
$prefix(\overline{1},2,\underline{1})=1 \Rightarrow p_3=1$;
$prefix(\overline{1,2},\underline{1,2})=2 \Rightarrow p_4=2$;
$prefix(\overline{1,2},\overline{1},2,\underline{1})=3 \Rightarrow p_5=3$;
$prefix(\overline{1,2},\overline{1,2},\underline{1,2})=4 \Rightarrow p_6=4$;
$prefix(1,2,1,2,1,2,3)=0 \Rightarrow p_7=0$;
$prefix(\overline{1},2,1,2,1,2,3,\underline{1})=1 \Rightarrow p_8=1$;
$prefix(\overline{1,2},1,2,1,2,3,\underline{1,2})=2 \Rightarrow p_9=2$;
$prefix(\overline{1,2,1},2,1,2,3,\underline{1,2,1})=3 \Rightarrow p_{10}=3$;
$prefix(\overline{1,2,1,2},1,2,3,\underline{1,2,1,2})=4 \Rightarrow p_{11}=4$;
$prefix(\overline{1,2,1,2,1},2,3,\underline{1,2,1,2,1})=5 \Rightarrow p_{12}=5$;
$prefix(\overline{1,2,1,2,1,2},3,\underline{1,2,1,2,1,2})=6 \Rightarrow p_{13}=6$;
$prefix(\overline{1,2,1,2,1,2,3},\underline{1,2,1,2,1,2,3})=7 \Rightarrow p_{14}=7$;
$prefix(1,2,1,2,1,2,3,1,2,1,2,1,2,3,4)=0 \Rightarrow p_{15}=0$;
$prefix(1,2,1,2,1,2,3,1,2,1,2,1,2,3,4,5)=0 \Rightarrow p_{16}=0$.

Як видно з прикладу, обчислення довжин префіксів для всіх можливих підпослідовностей образа дозволяє встановити максимальну довжину повторюваних груп елементів на початку і в кінці. Нульові значення довжин префіксів відповідають так званим «обривам» зростаючих префіксів. Точки «обривів» будуть визначати подальші «зсуви» образа відносно бази.

Програмна реалізація алгоритму обчислення довжин префіксів. Алгоритм формування додаткового масиву довжин префіксів на мові С++ доцільно реалізувати у вигляді окремої функції. Масив образа та додатковий масив довжин префіксів для цієї функції є глобальними:

```
void prefix( )
 {
   p[0]=0;
   int k=0;
   for (int j=1; j<M; j++)
    {
      while (k>0 && b[k]!=b[j]) k=p[k-1];
      if (b[k]==b[j]) k++;
```

```
      p[j]=k;
    }
  }
```

Функція ***prefix( )*** буде викликатися в основній програмі. Програма реалізація пошуку першого входження образа в базу може мати вигляд:

```
. . .
prefix( );
int k=0, j=0;
while (j<M && k<N)
 {
   while (j>0 && b[j]!=a[k]) j=p[j-1];
   if (b[j]==a[k]) j++;
   k++;
 }
if (j==M) cout << "є в позиції " << k-M << endl;
      else cout << "немає" << endl;
```

Приклад покрокового виконання. Для ілюстрації покрокового виконання пропонується розглянутий вище варіант образу. Для нього вже сформований додатковий масив довжин префіксів.

```
int const N=23, M=16;
int a[N]={1,2,1,2,1,2,3,1,2,1,2,1,2,3,1,2,1,2,1,2,3,4,5};
int b[M]={1,2,1,2,1,2,3,1,2,1,2,1,2,3,4,5};
int p[M]={0,0,1,2,3,4,0,1,2,3,4,5,6,7,0,0};
--- крок 1 ---
1 2 1 2 1 2 3 1 2 1 2 1 2 3 1 2 1 2 1 2 3 4 5
1 2 1 2 1 2 3 1 2 1 2 1 2 3 4 5
j=0; k=0;   b[j]=1=1=a[k]=>j=1; k=1;
--- крок 2 ---
1 2 1 2 1 2 3 1 2 1 2 1 2 3 1 2 1 2 1 2 3 4 5
1 2 1 2 1 2 3 1 2 1 2 1 2 3 4 5
j=1; k=1;   b[j]=2=2=a[k]=>j=2; k=2;
   .   .   .
--- крок 14 ---
1 2 1 2 1 2 3 1 2 1 2 1 2 3 1 2 1 2 1 2 3 4 5
1 2 1 2 1 2 3 1 2 1 2 1 2 3 4 5
j=13; k=13; b[j]=3=3=a[k]=>j=14; k=14;
--- крок 15 ---
1 2 1 2 1 2 3 1 2 1 2 1 2 3 ~~1~~ 2 1 2 1 2 3 4 5
1 2 1 2 1 2 3 1 2 1 2 1 2 3 ~~4~~ 5
j=14; k=14; b[j]=4≠1=a[k]=>j=p[j-1]=7;
```

Після обриву співпадань образ зміщується так, щоб з елементом бази ***a[14]=1***, який слідує після елемента ***a[13]=3***, сумістився елемент образу, який теж слідує після елемента зі значенням ***3***, тобто елемент ***b[7]=1***:

```
--- крок 16 ---
1 2 1 2 1 2 3 1 2 1 2 1 2 3 1 2 1 2 1 2 3 4 5
              1 2 1 2 1 2 3 1 2 1 2 1 2 3 4 5
j=7; k=14;  b[j]=1=1=a[k]=>j=8; k=15;
--- крок 17 ---
1 2 1 2 1 2 3 1 2 1 2 1 2 3 1 2 1 2 1 2 3 4 5
              1 2 1 2 1 2 3 1 2 1 2 1 2 3 4 5
j=8; k=15;  b[j]=2=2=a[k]=>j=9; k=16;
   .   .   .
--- крок 24 ---
1 2 1 2 1 2 3 1 2 1 2 1 2 3 1 2 1 2 1 2 3 4 5
              1 2 1 2 1 2 3 1 2 1 2 1 2 3 4 5
j=15; k=22; b[j]=5=5=a[k]=>j=16; k=23;
j=M, stop, yes!
```

Аналіз складності. Попередній аналіз образа потребує «важких» операцій порівняння. Цей аналіз виконується до початку основного пошуку, і його складність не залежить від кінцевого результату, а залежить лише від характеру розміщення елементів в образі. Основою цієї фази алгоритму є цикл по всіх елементах образа, які послідовно порівнюються з більш попередніми елементами при зростанні довжин префіксів. Якщо має місце обрив префіксів, такі елементи можуть повторно порівнюватися з більш віддаленими попередніми елементами. Враховуючи це, можна вважати, що складність попереднього аналізу образа пропорційна його довжині $C1^{hard} = O(M)$. Вона буде мінімальною, коли в образі взагалі немає префіксів або префікси без обривів досягають максимально можливої довжини. Тоді $C1^{hard}_{min} = M$.

Гіршими будуть випадки, коли довжини префіксів почергово спочатку зростають, а потім обриваються зі зменшенням довжини. Прикладом може бути наступний варіант образа:

```
1,1,2,1,1,3,1,1,3,1,1,3,...
```

Тут третій елемент ***2*** порівнюватиметься двічі з другим та першим елементами ***1*** і ***1***. Усі елементи зі значенням ***3*** порівнюватимуться тричі послідовно з третім, другим та першим елементами ***2***, ***1***, ***1***. А всі елементи зі значенням ***1***, починаючи з другого, порівнюватимуться по одному разу або з першим, або з другим елементом ***1***. Звичайно, можна навести ще гірші випадки, у яких кількість повторень підряд елементів ***1*** буде більшою. Тоді кратність порівнянь для елемента ***3*** теж буде більшою. Однак у межах фіксованої довжини образа ***N*** кількість таких довших груп елементів буде меншою. А тому функція складності буде обмеженою і досягатиме деякого максимуму. Звичайно, це

буде дуже особливий випадок, який у реальних задачах обробки структур даних зустрічатиметься нечасто.

Фаза основного пошуку передбачає послідовне співставлення елементів бази з елементами образу. У базі «повернень» назад і повторних «проходів» по тих самих елементах, які вже аналізувалися на попередніх етапах, не виконується. Тому теж можна вважати, що складність другої фази пошуку пропорційна довжині бази $C2^{hard} = O(N)$. Як і у випадку попереднього аналізу образа, можливі варіації складності, у залежності від характеру зміни довжин префіксів.

Таким чином, загальну складність алгоритму КМП пошуку підпослідовності в послідовності можна вважати пропорційною сумі довжин образу і бази $C^{hard} = O(M + N)$.

У ряді випадків вхідних даних, коли база приблизно вдвічі довша від образу, і довжини префіксів образу почергово спочатку зростають, а потім обриваються зі зменшенням довжини, може виявитися, що сумарна складність алгоритму КМП буде більшою від складності прямого пошуку.

1.2.4. Алгоритм Бойєра-Мура (БМ)

Ідея методу. Алгоритм КМП давав виграш у тих випадках, коли неспівпадання елементів фіксувалися ближче до кінця образа – тоді зсуви виконувалися на значні відстані. Однак це потребувало багатьох попередніх обчислень. Роберт Бойєр і Джей Мур у 1977 році запропонували виконувати порівняння елементів образа і бази не від початку образа, а навпаки – від його кінця. Це дозволило суттєво пришвидшити процес.

Для розуміння основних положень алгоритму Бойєра-Мура (БМ) розглянемо наступні можливі випадки зупинки чергового етапу при неспівпаданні елементів образу і бази, які базуються на евристиці використання *stop-елемента*:

1) якщо на деякому етапі порівнянь зафіксовано неспівпадання елемента образа ***b[j]*** з деяким елементом бази ***a[k]***, *якого точно немає в образі*, то зміщення образа відносно бази доцільно робити так, щоб він опинився відразу після позиції неспівпадання, наприклад:

```
                k
                ↓
a :   ... 1 2 7 1 2 3 1 2 3 1 2 7 ...
b :       1 2 3 1 2 3
                ↑
                j
зміщення →      1 2 3 1 2 3
```

2) якщо на деякому етапі порівнянь зафіксовано неспівпадання елемента образа ***b[j]*** з деяким елементом бази ***a[k]***, *який входить в образ одноразово і точно останнім*, то зміщення образа відносно бази доцільно робити на всю довжину образа, наприклад:

```
                k
                ↓
a :  ... 1 2 ~~7~~ 1 2 7 1 2 3 1 2 7 ...
b :      1 2 ~~3~~ 1 2 7
                ↑
                j
зміщення →            1 2 3 1 2 7
```

3) якщо на деякому етапі порівнянь зафіксовано неспівпадання елемента образа ***b[j]*** з деяким елементом бази ***a[k]***, *який є в образі, але не в кінці*, то зміщення образа відносно бази доцільно робити так, щоб сумістити елемент бази ***a[k]*** з останнім входженням цього ж елемента в образ, наприклад:

```
                  k
                  ↓
a :  ... 1 2 1 ~~2~~ 3 4 1 2 3 1 2 7 ...
b :      1 2 3 ~~4~~ 3 4
                  ↑
                  j
зміщення →    1 2 3 4 3 4
```

4) якщо на деякому етапі порівнянь зафіксовано неспівпадання елемента образа ***b[j]*** з деяким елементом бази ***a[k]***, *який є в образі неодноразово і в тому числі останнім*, то зміщення образа відносно бази доцільно робити так, щоб сумістити елемент бази ***a[k]*** з передостаннім входженням цього ж елемента в образ, наприклад:

```
                  k
                  ↓
a :  ... 1 2 1 ~~2~~ 3 2 1 2 3 1 2 7 ...
b :      1 2 3 ~~4~~ 3 2
                  ↑
                  j
зміщення →    1 2 3 4 3 4
```

Очевидно, що випадки 3 і 4 є подібними, і їх можна розглядати як різновиди одного спільного.

Ще одним різновидом, точніше, протилежністю, буде випадок, коли спроба суміщення ***a[k]*** з останнім його входженням в образ вимагатиме зміщення образа в протилежну сторону. Таким чином, виділимо ще і цей випадок:

5) якщо на деякому етапі порівнянь зафіксовано неспівпадання елемента образа ***b[j]*** з деяким елементом бази ***a[k]***, *який є в образі, але не в кінці, і останнє його входження має місце після позиції неспівпадання,* то зміщення образа відносно бази виконується на одну позицію, наприклад:

```
                  k
                  ↓
a :  ... 1 2 1 2 2 3 1 2 3 1 2 7 ...
b :      1 2 3 1 2 3
                  ↑
                  j
зміщення → 1 2 3 1 2 3
```

Звичайно, останній варіант зміщення образу є мало ефективним, оскільки суттєво погіршує часову складність пошуку в багатьох випадках вхідних даних. Для вирішення цієї проблеми використовується ще одна евристика попереднього аналізу образа з метою обчислення довжин *суфіксів*. Цей підхід аналогічний визначенню довжин префіксів в алгоритмі КМП.

Для визначення оптимальних зміщень образа відносно бази проводиться попередній аналіз, у процесі якого формується додаткова таблиця. Варто відзначити, що такий аналіз має особливості. Додатковий масив, що формується в процесі аналізу образа, має індексуватися всіма значеннями того ж самого типу, що є базовим для образа і бази. Тому алгоритм БМ можна застосовувати, коли базовий тип є дискретним:

- якщо базовий тип є однобайтовими (***char, unsigned char***), то додатковий масив матиме лише 256 елементів;
- якщо базовий тип є двобайтовим (***short***, ***unsigned***) або чотирибайтовим (***long, unsigned long***), то додатковий масив значно зростає за розміром: 65 536 елементів – для двобайтових та 4 294 967 296 елементів – для чотирибайтових;
- у випадку ще більших розмірів базового типу (наприклад, 8 байт для ***long long***), додатковий масив не вдасться розмістити ні в статичній, ні в динамічній пам’яті програми, оскільки потрібно мати 18 446 744 073 709 551 616 елементів;
- якщо базовий тип не є дискретним (наприклад, дійсні числа), то цей алгоритм взагалі не можна застосовувати.

Враховуючи таку особливість алгоритму, обмежимося випадком однобайтового базового типу ***unsigned char*** із діапазоном значень від ***0*** до ***255***. Це часто має місце при пошуку підпослідовності символів у тексті.

Попередній аналіз образа передбачає встановлення відстані від крайнього правого входження кожного елемента базового типу до правої межі образа. Під правою межею образа розуміється його останній елемент. При цьому слід враховувати різні випадки особливостей образа, що розглянуті вище:

- варіант образа без якихось особливостей відповідає випадку 3;
- якщо якогось зі значень базового типу в образі взагалі немає, то вважається, що це значення знаходиться безпосередньо перед початком образа, і для нього приймається відстань рівною довжині образа ***M*** (випадок 1);
- якщо крайнє праве входження деякого значення базового типу відповідає останньому елементу образа, тобто співпадає з його правою межею, і воно єдине, то приймається відстань рівною ***0*** (випадок 2);
- якщо крайнє праве входження деякого значення базового типу відповідає останньому елементу образа, тобто співпадає з його правою межею, але воно не єдине, то для такого значення визначається відстань від передостаннього входження (випадок 4);
- для оптимізації випадку 5, коли замість зміщення образа на один елемент виконується зміщення на максимально можливу відстань, проводиться обчислення довжин суфіксів.

Розглянемо приклад образа та сформуємо для нього додатковий масив ***d*** із ***256*** цілих чисел:

```
int const M=12;
int b[M]={1,2,3,1,2,3,4,1,2,3,4,5};
int d[256];

d[0]=M=12;              (випадок 1)
d[1]=4;                 (випадок 3)
d[2]=3;                 (випадок 3)
d[3]=2;                 (випадок 3)
d[4]=1;                 (випадок 3)
d[5]=0;                 (випадок 2)
d[6]=M=12;              (випадок 1)
d[7]=M=12;              (випадок 1)
. . .
d[255]=M=12;            (випадок 1)
```

Для даного прикладу випадок 4 відсутній, оскільки він є протилежним випадку 2. З метою ілюстрації випадку 4 пропонується інший приклад:

```
int b[M]={1,2,3,1,2,3,4,1,2,3,5,4};

d[0]=M=12;              (випадок 1)
d[1]=4;                 (випадок 3)
d[2]=3;                 (випадок 3)
d[3]=2;                 (випадок 3)
d[4]=5;                 (випадок 4)
```

```
d[5]=1;                  (випадок 3)
d[6]=M=12;               (випадок 1)
d[7]=M=12;               (випадок 1)
. . .
d[255]=M=12;             (випадок 1)
```

Програмна реалізація.

Як вже зрозуміло з опису особливостей алгоритму, програмна реалізація охоплюватиме декілька основних фаз:

- **попередній аналіз образа** з метою формування додаткового масиву `d` відстаней зміщень. Як і було домовлено, вважатимемо базовим типом елементів бази та образа однобайтний тип цілих чисел без знаку. У цій фазі умовно можна виділити три кроки:

```
for (int i=0; i<256; i++)
  d[i]=M;
```
/ тут початково приймається, що всі елементи базового типу відсутні в образі, а тому для них відстань встановлюється рівною M (випадок 1) */*
```
for (int j=0; j<M-1; j++)
  d[b[j]]=M-j-1;
```
/ тут аналізуються всі елементи образа від початку до передостаннього елемента, і для кожного визначається відстань від правої межі образа; у результаті остаточно буде встановлена відстань для найбільш правого входження кожного елемента базового типу (випадки 3 і 4) */*
```
if (d[b[M-1]]==M) d[b[M-1]]=0;
```
/ тут аналізується останній елемент образа; якщо для нього відстань від правої межі не змінилася, то це означає, що такий ключ останній і єдиний в образі; для нього відстань встановлюється рівною 0 (випадок 2) */*

- **основний пошук,** що передбачає порівняння елементів образа і бази та зсуви образа на основі сформованої таблиці відстаней:

```
. . .
int j=M-1, k=M-1;
while (j>=0 && k<N)
 {
  while (j>=0 && b[j]==a[k]) {j--; k--;}
  if (j>=0)                                  // образ зміщується
   {
    if (d[a[k]]==M) k+=M;                    // випадок 1
      else if (!d[a[k]]) k+=M*2-j-1;         // випадок 2
        else if (d[a[k]]>M-j-1) k+=d[a[k]];  // випадки 3 і 4
          else k+=M-j;                       // випадок 5
    j=M-1;
   }
 }
```

if (j<0) cout << "є в позиції " << k+1 << endl;
else cout << "немає" << endl;

Приклади покрокового виконання.

1)

```
a : 1 2 3 1 2 3 6 1 2 3 4 5 1 2 3 1 2 3 4 1 2 3 4 5
b : 1 2 3 1 2 3 4 1 2 3 4 5
порівнянь 6, випадок 1:
a : 1 2 3 1 2 3 6 1 2 3 4 5 1 2 3 1 2 3 4 1 2 3 4 5
b :               1 2 3 1 2 3 4 1 2 3 4 5
порівнянь 1, випадок 3:
a : 1 2 3 1 2 3 6 1 2 3 4 5 1 2 3 1 2 3 4 1 2 3 4 5
b :                 1 2 3 1 2 3 4 1 2 3 4 5
порівнянь 1, випадок 3:
a : 1 2 3 1 2 3 6 1 2 3 4 5 1 2 3 1 2 3 4 1 2 3 4 5
b :                           1 2 3 1 2 3 4 1 2 3 4 5
порівнянь 12, stop, yes!
всього порівнянь: 20
```

2)

```
a : 1 2 3 1 2 3 5 1 2 3 4 5 1 2 3 1 2 3 4 1 2 3 4 5
b : 1 2 3 1 2 3 4 1 2 3 4 5
порівнянь 6, випадок 2:
a : 1 2 3 1 2 3 5 1 2 3 4 5 1 2 3 1 2 3 4 1 2 3 4 5
b :                           1 2 3 1 2 3 4 1 2 3 4 5
порівнянь 12, stop, yes!
всього порівнянь: 18
```

3)

```
a : 1 2 3 1 2 3 3 1 2 3 4 5 1 2 3 1 2 3 4 1 2 3 4 5
b : 1 2 3 1 2 3 4 1 2 3 4 5
порівнянь 6, випадок 5:
a : 1 2 3 1 2 3 3 1 2 3 4 5 1 2 3 1 2 3 4 1 2 3 4 5
b :   1 2 3 1 2 3 4 1 2 3 4 5
порівнянь 1, випадок 3:
a : 1 2 3 1 2 3 3 1 2 3 4 5 1 2 3 1 2 3 4 1 2 3 4 5
b :             1 2 3 1 2 3 4 1 2 3 4 5
порівнянь 1, випадок 3:
a : 1 2 3 1 2 3 3 1 2 3 4 5 1 2 3 1 2 3 4 1 2 3 4 5
b :                 1 2 3 1 2 3 4 1 2 3 4 5
порівнянь 1, випадок 3:
a : 1 2 3 1 2 3 3 1 2 3 4 5 1 2 3 1 2 3 4 1 2 3 4 5
b :                           1 2 3 1 2 3 4 1 2 3 4 5
порівнянь 12, stop, yes!
всього порівнянь: 21
```

4)

a : 1 1 1 1 1 ~~**1**~~ 1 1 1 1 1 2
b : 1 1 1 1 **1** ~~**2**~~
порівнянь: **1**, випадок **3**:
a : 1 1 1 1 1 1 ~~**1**~~ 1 1 1 1 2
b : 1 1 1 1 **1** ~~**2**~~
порівнянь: **1**, випадок **3**:
a : 1 1 1 1 1 1 1 ~~**1**~~ 1 1 1 2
b : 1 1 1 1 **1** ~~**2**~~
порівнянь: **1**, випадок **3**:
a : 1 1 1 1 1 1 1 1 ~~**1**~~ 1 1 2
b : 1 1 1 1 **1** ~~**2**~~
порівнянь: **1**, випадок **3**:
a : 1 1 1 1 1 1 1 1 1 ~~**1**~~ 1 2
b : 1 1 1 1 **1** ~~**2**~~
порівнянь: **1**, випадок **3**:
a : 1 1 1 1 1 1 1 1 1 1 ~~**1**~~ 2
b : 1 1 1 1 **1** ~~**2**~~
порівнянь: **1**, випадок **3**:
a : 1 1 1 1 1 1 1 1 1 1 1 2
b : 1 1 1 1 1 1
порівнянь **6**, **stop**, **yes!**
всього порівнянь: **12**

5)

a : ~~**1**~~ 1 1 1 1 1 1 1 1 1 1 1
b : ~~**2**~~ 1 1 1 1 1
порівнянь: **6**, випадок **5**:
a : 1 ~~**1**~~ 1 1 1 1 1 1 1 1 1 1
b : ~~**2**~~ 1 1 1 1 1
порівнянь: **6**, випадок **5**:
a : 1 1 ~~**1**~~ 1 1 1 1 1 1 1 1 1
b : ~~**2**~~ 1 1 1 1 1
порівнянь: **6**, випадок **5**:
a : 1 1 1 ~~**1**~~ 1 1 1 1 1 1 1 1
b : ~~**2**~~ 1 1 1 1 1
порівнянь: **6**, випадок **5**:
a : 1 1 1 1 ~~**1**~~ 1 1 1 1 1 1 1
b : ~~**2**~~ 1 1 1 1 1
порівнянь: **6**, випадок **5**:
a : 1 1 1 1 1 ~~**1**~~ 1 1 1 1 1 1
b : ~~**2**~~ 1 1 1 1 1
порівнянь: **6**, випадок **5**:
a : 1 1 1 1 1 1 ~~**1**~~ 1 1 1 1 1
b : ~~**2**~~ 1 1 1 1 1
порівнянь: **6**, випадок **5**:

```
a : 1 1 1 1 1 1 1 1 1 1 1 1
b :               2 1 1 1 1 1
stop, no!
всього порівнянь: 42
```

Зауваження.

1. У прикладі **4)** щоразу виконуються зсуви образу на **1** позицію. Але при цьому кожен етап має мінімальну довжину **1**. Такий варіант вхідних даних є одним із найкращих для БМ. Варто відзначити, що розглянутий у прикладі **4)** варіант даних у свою чергу є найгіршим для прямого пошуку.

2. У прикладі **5)** щоразу виконуються зсуви образа на **1** позицію при максимальній довжині кожного етапу **M**. Такий варіант вхідних даних є найгіршим для БМ. Варто відзначити, що розглянутий у прикладі **4)** варіант даних у свою чергу є одним із найкращих для прямого пошуку.

Варіант вхідних даних, що є найкращим для прямого пошуку, є одним із найгірших для БМ і навпаки.

РОЗДІЛ 2
ЗАДАЧА СОРТУВАННЯ

Знову використаємо в якості моделі структури даних масив із N елементів деякого базового типу, наприклад цілих чисел. Для *структур даних з прямим доступом* допускається довільна зміна індексних номерів поточного елемента, а у випадку *структур даних з послідовним доступом* індексні номери поточного елемента можуть змінюватися виключно з приростом ***+1***.

Сортування передбачає перегрупування елементів структури даних так, щоб вони задовольняли деяку умову. Найчастіше виділяють наступні умови сортування:

5) $a_i < a_{i+1}$, $i = \overline{1, N-1}$ – по зростанню;

6) $a_i > a_{i+1}$, $i = \overline{1, N-1}$ – по спаданню.

Якщо в структурі даних є декілька елементів з однаковими значеннями, то слід використовувати нестрогі умови:

7) $a_i \le a_{i+1}$, $i = \overline{1, N-1}$ – по неспаданню;

8) $a_i \ge a_{i+1}$, $i = \overline{1, N-1}$ – по незростанню.

Не зменшуючи загальності, розглядатимемо сортування за умовою *неспадання* (3). При цьому нас цікавитиме лише *стійке сортування*. Це означає, що всі елементи, які початково задовольняли умову $a_i \le a_j$, $1 \le i < j \le N$, після впорядкування, хоч і можуть змінити місце розташування в структурі даних, проте не змінюють свого взаємного розміщення. Ця вимога є важливою, коли в структурі даних є багато однакових елементів: більш «лівіші» та більш «правіші» елементи з однаковими значеннями залишаються так само «лівішими» і «правішими» відповідно.

Ще однією важливою вимогою, яка ставиться до алгоритмів обробки структур даних, і сортування в тому числі, є базовий принцип «*обробка на тому ж місці*», що означає «*без використання додаткових структур даних такого ж розміру*».

Як і у випадку задачі пошуку, вивчення алгоритмів сортування розпочнемо з *прямих* методів, що базуються на визначальних принципах *включення (вставки), вибору, обміну*. А потім ті самі принципи знайдуть реалізацію у *швидких* алгоритмах. Названі способи стосуються структур даних з прямим доступом (масивів). Для структур даних з послідовним доступом будуть розглянуті алгоритми, що використовують принцип *злиття*.

2.1. ПРЯМІ АЛГОРИТМИ СОРТУВАННЯ

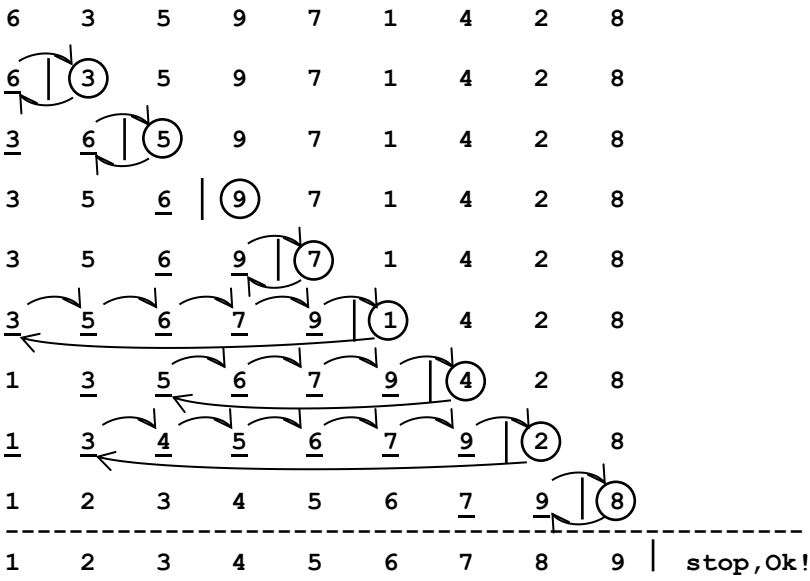
2.1.1. АЛГОРИТМ ПРЯМОГО ВКЛЮЧЕННЯ (ВСТАВКИ)

Ідея методу. Масив умовно розділяється на дві частини: «готова» – містить елементи масиву $a_1, a_2, ..., a_i, i = \overline{1, N-1}$, з яких вже сформована впорядкована підпослідовність (початково «готову» частину утворює один перший елемент a_1); невпорядкована – решта елементів масиву.

З невпорядкованої частини послідовно беруться *нові* елементи, і для них *шукається місце вставки* в «готовій» частині. Новий елемент ніби «проштовхується» на своє місце у впорядкованій частині, рухаючись справа наліво. Після кожного етапу «готова» частина збільшується на один елемент справа, а невпорядкована частина зменшується на один елемент зліва.

Приклад покрокового виконання. Для зручності ілюстрації процесу сортування використані наступні позначення:

- «готова» і невпорядкована частини розділені вертикальною лінією |;
- новий елемент із невпорядкованої частини охоплений колом ○;
- елементи впорядкованої частини, з якими порівнюється новий елемент, – підкреслені;
- на переміщення елементів вказують дугові стрілки ↷, ↶.



Програмна реалізація. Процес пошуку місця вставки нового елемента в «готовій» частині припиняється в одному з двох випадків:

1) поточний елемент впорядкованої частини, з яким виконується порівняння, не перевищує нового елемента – новий елемент має опинитися відразу після поточного;
2) новий елемент менший від усіх елементів впорядкованої частини – він має зайняти першу позицію.

Реалізація алгоритму на мові C++ може мати вигляд:

```
int i, j, x;
for (i=1; i<N; i++)
 {
  x=a[i]; j=i-1;
  while (j>=0 && x<a[j])
   {
    a[j+1]=a[j];
    j--;
   }
  a[j+1]=x;
 }
```

Аналіз складності. У випадку задачі пошуку часова складність алгоритмів визначалася лише операціями порівняння. Це було оправданим, оскільки переприсвоєння між елементами базового типу не виконувалися. Однак ігнорувати будь-які операції присвоєння не можна. Адже, коли базовий тип елементів структури даних за розміром не перевищує тип їх індексних номерів, то операції переприсвоєння індексів суттєво впливатимуть на загальний час роботи алгоритму. У випадку задачі сортування основною дією є перегрупування елементів за певним принципом. Тому, окрім операцій порівняння, не меншу, а деколи і більшу, частину загального часу роботи алгоритму займають операції переприсвоєння між елементами базового типу. Знову найкращими випадками вхідних даних вважатимемо ті, які потребуватимуть мінімальної кількості «важких» операцій порівняння і присвоєння, а найгіршими – коли таких операцій буде найбільше.

Найкращому випадку вхідних даних очевидно відповідають так звані «хороші» структури даних, тобто вже впорядковані або майже впорядковані, у яких лише поодинокі елементи знаходяться не на своїх місцях, але близько до них розміщені. На кожному етапі буде виконуватися лише одне порівняння нового елемента з крайнім правим елементом «готової» частини. При цьому щоразу виконується два присвоєння: ***x=a[i]*** (перед внутрішнім циклом ***while***) та ***a[j+1]=x*** (після циклу). Загалом по всіх етапах буде така кількість операцій порівняння та присвоєння:

$$C_{\min} = \sum_{i=1}^{N-1} 1 = N-1; \quad M_{\min} = \sum_{i=1}^{N-1} 2 = 2(N-1).$$

Найгіршому випадку відповідають так звані «погані» структури даних, тобто обернено впорядковані або близькі до них, у яких переважна більшість елементів знаходяться далеко від своїх місць. У таких випадках на кожному етапі новий елемент порівнюватиметься з усіма елементами «готової» частини і щоразу послідовно «виштовхуватиме» їх із поточної позиції в наступну, аж поки не переміститься на початок:

$$C^i_{\max} = i, \quad i = \overline{1, N-1}; \quad M^i_{\max} = i+2, \quad i = \overline{1, N-1};$$

$$C_{\max} = \sum_{i=1}^{N-1} C^i_{\max} = \sum_{i=1}^{N-1} i = 1+2+...+N-1 = \frac{N^2-N}{2};$$

$$M_{\max} = \sum_{i=1}^{N-1} M^i_{\max} = \sum_{i=1}^{N-1} (i+2) = 3+4+...+N+1 = \frac{N^2+3N-4}{2}.$$

Як видно, ефективність даного алгоритму суттєво залежить від початкового порядку елементів.

2.1.2. Модифікація прямого включення – бінарне включення

Ідея методу. Низька ефективність прямого включення для «поганих» структур даних пов'язана з тим, що практично на кожному етапі новий елемент у пошуку свого місця вставки здійснює далекі переміщення на початок «готової» частини і при цьому послідовно порівнюється з усіма її елементами. Звичайно, від переміщень багатьох елементів на одну позицію вправо для звільнення позиції вставки в «готовій» частині відмовитися не вдасться. Але можна пришвидшити процес пошуку самого місця вставки. Оскільки «готова» частина вже впорядкована, то для цього варто застосовувати алгоритм бінарного пошуку.

Оскільки нас цікавить лише стійке сортування, то новий елемент повинен займати крайню праву позицію серед можливо декількох підряд однакових із ним елементів у «готовій» частині. Тому доцільно застосовувати алгоритм бінарного пошуку з додатковою модифікацією.

Розглянутий раніше модифікований бінарний пошук давав перше по порядку входження заданого значення серед можливо декількох підряд однакових. У випадку ж сортування для вставки нового елемента зі значенням ***x*** на відповідне йому місце в «готовій» частині потрібен пошук останнього серед можливо декількох підряд однакових із ним елементів. Якщо ж новий елемент ***x*** є унікальним у «готовій» частині, тобто однакових із ним немає, то він має опинитися після останнього з менших за нього. Таким чином, для

нового елемента ***x*** позиція вставки ***k*** в «готовій» частині визначається умовами ***a[k-1] ≤ x < a[k+1].***

Програмна реалізація. У випадку модифікованого бінарного пошуку аналізується «готова» частина, що обмежена індексами ***L*** та ***R.*** Пошук завершується при їх рівності ***L=R***, що відповідає позиції вставки. Реалізація алгоритму на мові С++ може мати вигляд:

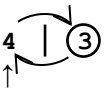
```
int i, j, L, R, x, m;
for (i=1; i<N; i++)
 {
   x=a[i]; L=0; R=i;
   while (L<R)
    {
      m=(L+R)/2;
      if (a[m]<=x) L=m+1; else R=m;
    }
   for (j=i; j>R; j--) a[j]=a[j-1];
   a[R]=x;
 }
```

Приклад покрокового виконання. Пропонується варіант вхідних даних із багатократним повторенням однакових елементів, зокрема, ***3***. Цей приклад ілюструє стійкість бінарного включення, коли кожен новий елемент займатиме крайню праву позицію серед можливо декількох підряд однакових із ним елементів у «готовій» частині. Тут використана індексація масивів, характерна для мов С/С++, коли індексні номери починаються з нуля.

```
4    3    2    3    6    3    1    3    5
-------------------------------------------------
```

1-ий етап:

```
↓m
4  |  ③
↑L    ↑R
i=1,x=3,L=0,R=1,m=0,a[m]=4>3=x => R=m=0,L=0=R,стоп,
         позиція вставки нового елемента L=R=0:
```

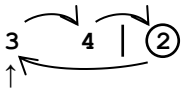


```
↑
L=R=0
-------------------------------------------------
```

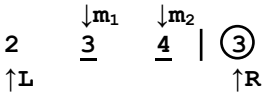
2-ий етап:

```
↓m₂   ↓m₁
3     4  |  ②
↑L          ↑R
```

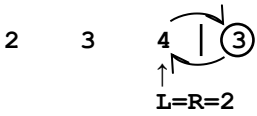
```
i=2,x=2,L=0,R=2,m1=1,a[m1]=4>2=x => R=m1=1,L=0<1=R,
        L=0,R=1,m2=0,a[m2]=3>2=x => R=m2=0,L=0=R,стоп,
        позиція вставки нового елемента L=R=0:
```



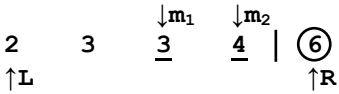
3-ій етап:



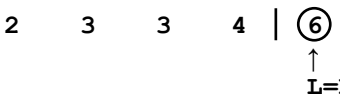

```
i=3,x=3,L=0,R=3,m1=1,a[m1]=3=3=x => L=m1+1=2,L=2<3=R,
        L=2,R=3,m2=2,a[m2]=4>3=x => R=m2=2,L=2=R,стоп,
        позиція вставки нового елемента L=R=2:
```



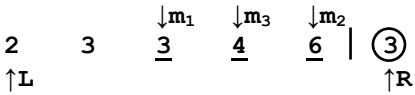
4-ий етап:



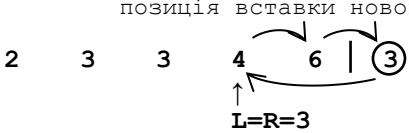

```
i=4,x=6,L=0,R=4,m1=2,a[m1]=3<6=x => L=m1+1=3,L=3<4=R,
        L=3,R=4,m2=3,a[m2]=4<6=x => L=m2+1=4,L=4=R,стоп,
        позиція нового елемента не змінилася L=R=4:
```



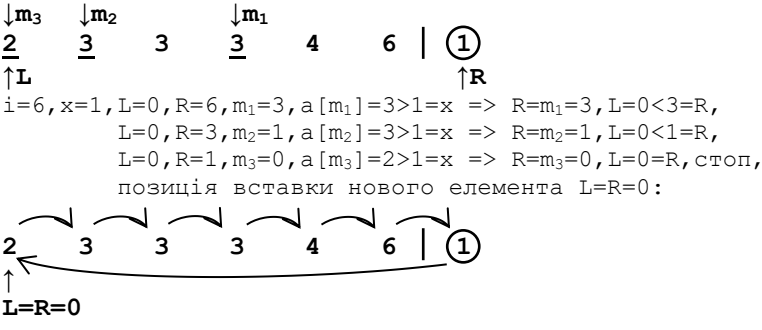
5-ий етап:



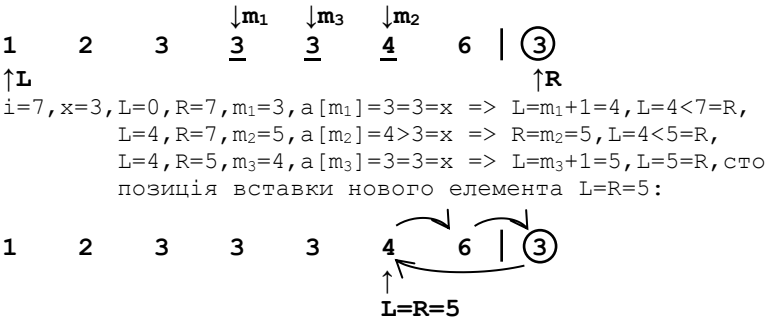

```
i=5,x=3,L=0,R=5,m1=2,a[m1]=3=3=x => L=m1+1=3,L=3<5=R,
        L=3,R=5,m2=4,a[m2]=6>3=x => R=m2=4,L=3<4=R,
        L=3,R=4,m3=3,a[m3]=4>3=x => R=m3=3,L=3=R,стоп,
        позиція вставки нового елемента L=R=3:
```



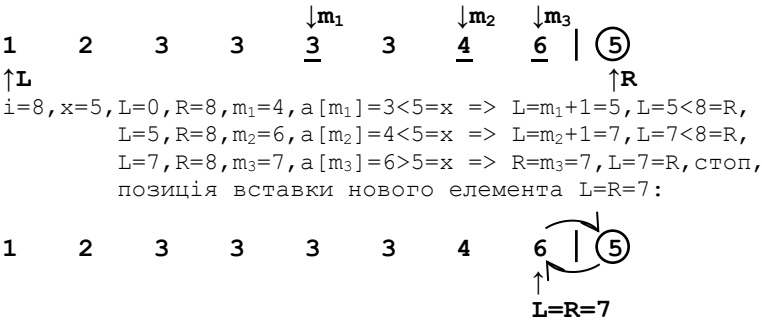
6-ий етап:



7-ий етап:



8-ий етап:



---

1 2 3 3 3 3 4 5 6 | stop,Ok!

Аналіз складності. Зрозуміло, що для бінарного включення відсутнє розділення на кращі чи гірші варіанти вхідних даних по кількості «важких» операцій порівняння. У будь-якому випадку процес пошуку місця вставки нового елемента в «готовій» частині припиняється, коли після бінарних поділів

залишається один єдиний елемент при ***L=R***. Таким чином, ділення пополам на ***i***-ому етапі здійснюється ***log(i)*** раз. Враховуючи, що кількість поділів перелічуються цілими числами, то замість значення ***log(i)*** може використовуватися його цілочисельне наближення зверху. Тобто істинна кількість порівнянь на кожному етапі може бути більшою від ***log(i)*** на ***1***. Отже, кількість операцій порівняння можна вважати такою:

$C = \sum_{i=1}^{N-1} C_i$, де $\lceil \log(i+1) \rceil \le C_i \le \lceil \log(i+1) \rceil + 1$, $i = \overline{1, N-1}$.

Апроксимуючи цю суму інтегралом, отримаємо наступну оцінку:

$\int_1^N \log(x)\,dx \le C \le \int_1^N (\log(x)+1)\,dx$, або

$N(\log(N)-b)+b \le C \le N(\log(N)-b+1)+b-1$, де $b = \frac{1}{\ln(2)} = 1,4426$.

Остаточно можна вважати

$$C = N(\log(N) - 1 \mp 0,5) < N \cdot \log(N) << \frac{N^2 - N}{2}.$$

Кількість операцій порівняння не залежить від початкового порядку елементів у структурі даних, а залежить лише від їх числа. Якщо не брати до уваги дуже «хороші» структури даних, а навпаки розглядати більш «погані» випадки, то бінарне включення вигравatиме саме по операціях порівняння.

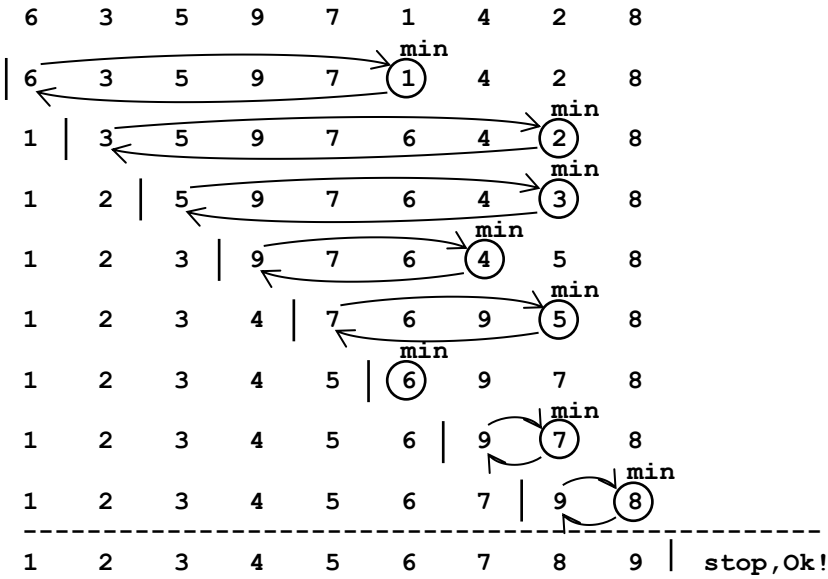
На жаль, розглянута модифікація алгоритму сортування за принципом включення (вставки) не дає жодного виграшу по операціях переприсвоєння. Ефективність бінарного включення за цим критерієм точно співпадає з ефективністю прямого включення. Фактично кількість перестановок елементів залишається величиною порядку $O(N^2)$. Це особливо актуально для структур даних зі складеним базовим типом великого розміру, коли впорядкування проводиться за окремим ключем невеликого розміру. У таких випадках переміщення складеного елемента даних, тобто і ключа, і зв'язаної з ним інформації, займає значно більше часу, ніж порівняння самих ключів. Очевидно, що у цьому випадку кращий результат можуть дати алгоритми, де перестановка, можливо, на велику відстань, буде пов'язана лише з одним елементом, а не з переміщенням на одну позицію цілої групи елементів.

2.1.3. АЛГОРИТМ ПРЯМОГО ВИБОРУ

Ідея методу. Структура даних знову умовно розділяється на «готову» і невпорядковану частини. Початково «готова» частина є порожньою. На кожному етапі серед елементів невпорядкованої частини *вибирається* перший за порядком серед елементів із найменшим значенням. Якщо обраний

найменший елемент не є першим у невпорядкованій частині, то виконується обмін місцями між обраним найменшим і першим у невпорядкованій частині. У результаті «готова» частина розширюється на один елемент вправо, а невпорядкована звужується на один елемент зліва.

Приклад покрокового виконання.



На шостому етапі найменший елемент у невпорядкованій частині ***6*** є першим по порядку, тому обмін його із самим собою не проводиться.

Програмна реалізація.

```
int i, j, min, x;
for (i=0; i<N-1; i++)
 {
   min=i;
   for (j=i+1; j<N; j++)
     if (a[j]<a[min]) min=j;
   if (min>i) { x=a[i]; a[i]=a[min]; a[min]=x; }
 }
```

Аналіз складності. Для даного алгоритму кількість операцій порівняння не залежить від початкового порядку елементів у структурі даних. І у випадку початково впорядкованої, і обернено впорядкованої, і довільної

невпорядкованої послідовності виконується максимальна кількість операцій порівняння:

$$C = (N-1) + (N-2) + \ldots + 2 + 1 = \frac{N^2 - N}{2}.$$

Кількість операцій присвоєння залежить від початкового порядку елементів, але не є значною. У випадку прямо впорядкованої послідовності переміщень елементів взагалі немає, тобто $M_{\min} = 0$.

Цікаво, що варіант обернено впорядкованої послідовності не є найгіршим. Адже після перших $\left[\frac{N}{2}\right]$ етапів отримується впорядкована послідовність, і подальші етапи вже нічого не змінюють, хоча непродуктивні порівняння однаково виконуються:

$$M_{оберн} = 3 \cdot \left[\frac{N}{2}\right].$$

Найгіршим варіантом вхідних даних по операціях присвоєння є особливим чином довільно організована послідовність елементів, для якої на кожному етапі у невпорядкованій частині виконується обмін місцями елементів із найменшим ключем та першого від початку:

$$M_{\max} = 3 \cdot (N-1).$$

2.1.4. АЛГОРИТМ ПРЯМОГО ОБМІНУ

Ідея методу. Структура даних знову умовно розділяється на «готову» і невпорядковану частини. Початково «готова» частина є порожньою. У невпорядкованій частині порівнюється виключно пари сусідніх елементів $a_i, a_{i+1}, i = \overline{1, N-1}$. Якщо чергова пара не є впорядкованою, то виконується *обмін місцями* цих елементів. Такі порівняння і можливі обміни елементів виконуються в русі або від початку до кінця невпорядкованої частини, або від кінця до початку:

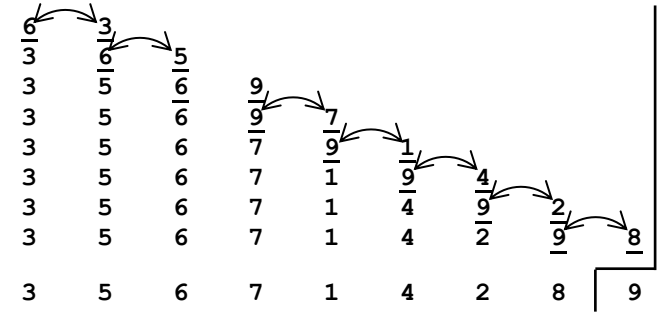
- якщо порівняння здійснюється в русі від початку до кінця, то за один етап останній серед можливо декількох однакових із максимальним ключовим значенням «виштовхнеться» в кінець невпорядкованої частини. Тоді готова частина буде формуватися від кінця структури даних. Такий спосіб сортування обміном називають *методом каміння*;
- якщо порівняння виконується в русі від кінця до початку, то за один етап перший серед можливо декількох однакових із мінімальним ключовим значенням «виштовхнеться» на початок невпорядкованої частини. Тоді готова частина буде формуватися від початку структури даних. Цей спосіб сортування обміном називають *методом бульбашки*.

Приклад покрокового виконання.

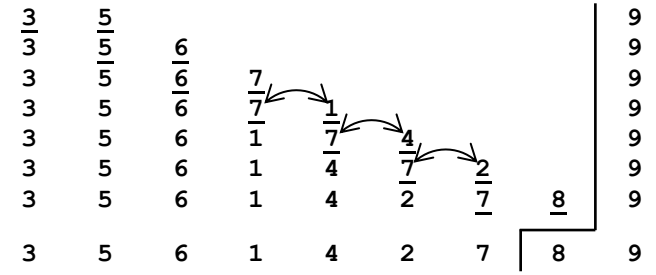
Метод каміння:

```
6   3   5   9   7   1   4   2   8
```

1-ий етап



2-ий етап

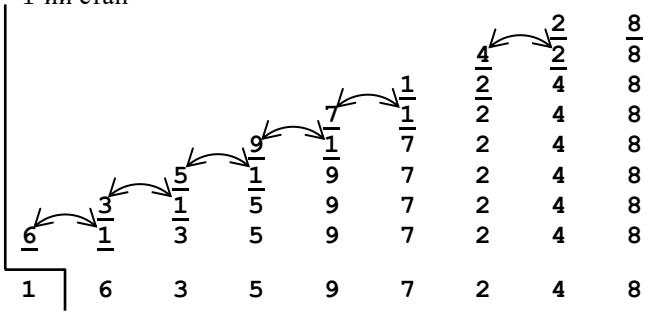


… і т.д.

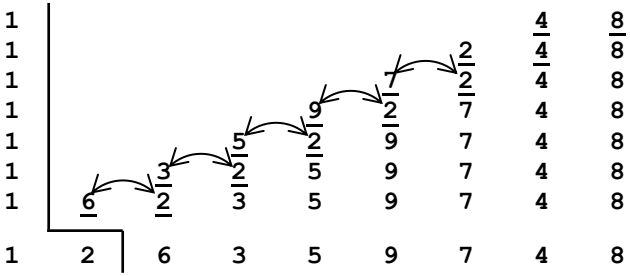
Метод бульбашки:

```
6   3   5   9   7   1   4   2   8
```

1-ий етап



2-ий етап



… і т.д.

Програмна реалізація. У випадку методу камінця, коли рух здійснюється від початку до кінця структури даних і «готова» частина формується справа, ліва межа ***L*** невпорядкованої частини зафіксована при ***L=0***, а права межа ***R*** на кожному етапі зміщується вліво на одну позицію, послідовно набуваючи значень ***N-1, N-2, …, 2, 1***:

```
int L=0, x;
for (int R=N-1; R>L; R--)
 {
   for (int i=L; i<R; i++)
     if (a[i]>a[i+1]) { x=a[i]; a[i]=a[i+1]; a[i+1]=x; }
 }
```

У випадку методу бульбашки, коли рух здійснюється від кінця до початку структури даних і «готова» частина формується зліва, права межа ***R*** невпорядкованої частини зафіксована при ***R=N-1***, а ліва межа ***L*** на кожному етапі зміщується вправо на одну позицію, послідовно набуваючи значень ***0, 1, …, N-3, N-2***:

```
int R=N-1, x;
for (int L=0; L<R; L++)
 {
   for (int i=R-1; i>=L; i--)
     if (a[i]>a[i+1]) { x=a[i]; a[i]=a[i+1]; a[i+1]=x; }
 }
```

Аналіз складності. Для обох різновидів алгоритму прямого обміну кількість операцій порівняння не залежить від початкового порядку елементів у структурі даних. І у випадку початково впорядкованої, і обернено

впорядкованої, і довільної невпорядкованої послідовності виконуються всі етапи з усіма можливими порівняннями:

$$C = (N-1) + (N-2) + \ldots + 2 + 1 = \frac{N^2 - N}{2}.$$

Кількість операцій присвоєння суттєво залежить від початкового порядку елементів. У найкращому випадку початково впорядкованої послідовності не виконується жодного переміщення елементів: $M_{\min} = 0$.

У найгіршому випадку обернено впорядкованої послідовності при кожному порівнянні має місце трикратне присвоєння значень елементів:

$$M_{\max} = 3 \cdot \frac{N^2 - N}{2}.$$

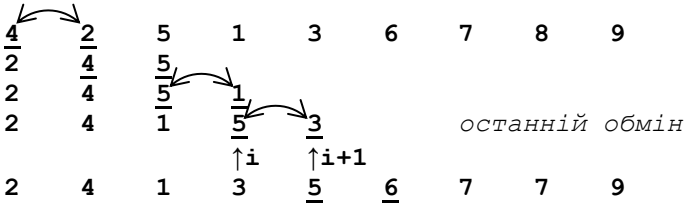
Як видно, складність алгоритму прямого обміну по «важких» операціях є однією з найбільших серед усіх прямих методів. До позитиву цього методу можна віднести простоту програмної реалізації та можливість модифікацій у напрямі суттєвого покращення ефективності для «хороших» структур даних.

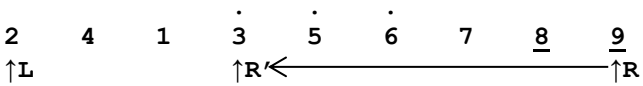
2.1.5. Модифікація прямого обміну – шейкерне сортування

Ідея методу. Низька ефективність прямого обміну (бульбашка, камінець) у випадках «хороших» структур даних пов'язана з тим, що алгоритм не передбачає дострокової зупинки, якщо на деякому етапі послідовність вже впорядкована.

Покращення ефективності можна реалізувати за результатами аналізу обмінів попереднього етапу. По-перше, якщо обмінів не було, то структура даних вже впорядкована, і процес сортування можна завершити. По-друге, якщо обміни мали місце, і останній із них зафіксований задовго до завершення поточного етапу (далеко від правої межі невпорядкованої частини у випадку методу камінця або від лівої – для бульбашки відповідно), то на ділянці невпорядкованої частини після останнього обміну елементи вже впорядковані. У цьому випадку до «готової» частини можна віднести відразу всі такі елементи, тобто значно зменшити кількість етапів сортування.

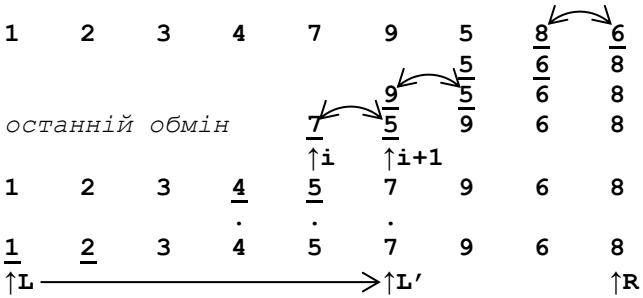
Нехай у випадку методу камінця на деякому етапі ***k***, що значно менше від максимально можливої кількості ***N-1***, зафіксовано останній обмін між елементами $a_i, a_{i+1}, i < N - k$, наприклад:





Тоді праву межу ***R′*** невпорядкованої частини доцільно зміщувати вліво не на один елемент, а в позицію останнього обміну, а точніше – у позицію ***i***.

Нехай у випадку методу бульбашки на деякому етапі ***k***, що значно менше від максимально можливої кількості ***N-1***, зафіксовано останній обмін між елементами $a_i, a_{i+1}, i > k+1$, наприклад:



Тоді ліву межу ***L′*** невпорядкованої частини доцільно зміщувати вправо не на один елемент, а в позицію останнього обміну, а точніше – у позицію ***i+1***.

І по-третє, щоб максимально вкорочувати невпорядковану частину одночасно зліва і справа, доцільно модифіковані методи бульбашки і камінця застосувати почергово. Тоді структура даних ніби «струшується»: за методом камінця найбільший елемент зміщується в кінець, а бульбашка переміщує найменший елемент на початок і т.д. Такий спосіб сортування називають *шейкерним* (англ. *shake – трусити*).

Програмна реалізація. Щоб добитися максимального ефекту від застосування усіх модифікацій, потрібно об'єднати однією спільною реалізацією дві протилежні ідеї: фіксація позиції останнього обміну та відсутність самих обмінів на поточному етапі. З цією метою варто припустити, що для методу камінця останній обмін перед черговим етапом мав місце на лівій межі – у позиції ***L***, а для методу бульбашки останній обмін перед черговим етапом мав місце на правій межі – у позиції ***R***. Якщо на поточному етапі обмінів взагалі не було, то зміщення відповідної для кожного методу межі в раніше зафіксовану позицію відразу приведе до порожньої невпорядкованої частини для наступного етапу, тобто до завершення процесу сортування.

Очевидно, що вибір порядку застосування методів камінця і бульбашки є довільним. Нехай для визначеності першим буде камінець. Реалізація на мові C++ алгоритму шейкерного сортування може мати вигляд:

```
int L=0, R=N-1, Pos=L, i, x;
while (L<R)
 {
   for (i=L; i<R; i++)
     if (a[i]>a[i+1]) { x=a[i]; a[i]=a[i+1]; a[i+1]=x; Pos=i; }
   R=Pos;
   for (i=R-1; i>=L; i--)
     if (a[i]>a[i+1]) { x=a[i]; a[i]=a[i+1]; a[i+1]=x; Pos=i+1; }
   L=Pos;
}
```

Аналіз складності. Даний метод очікувано має високу ефективність у випадку «хороших» структур даних. У найкращому випадку початково впорядкованої послідовності виконається лише один етап того методу, який реалізований першим по порядку. При цьому

$C_{\min} = N - 1; \quad M_{\min} = 0$.

На жаль, у випадках «поганих» структур даних ефективність алгоритму є низькою. У найгіршому випадку обернено впорядкованої послідовності загальна складність шейкерного сортування така сама, як і прямого обміну:

$$C_{\max} = \frac{N^2 - N}{2}; \quad M_{\max} = 3 \cdot \frac{N^2 - N}{2}.$$

2.2. ШВИДКІ АЛГОРИТМИ СОРТУВАННЯ МАСИВІВ

2.2.1. Швидке сортування включенням зі зменшуваними відстанями. Алгоритм Шелла

Ідея методу. Низька ефективність усіх прямих методів пов'язана з тим, що за один етап суттєво обробляється лише один елемент, а решта елементів невпорядкованої частини між собою практично не взаємодіють.

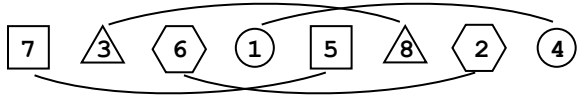
Дональд Шелл у 1959 році запропонував багаторазово сортувати різні частини основної структури даних – підпослідовності, елементи яких знаходяться на значних відстанях. Причому відстані між елементами в підпослідовностях мають поетапно змінюватися, а саме – зменшуватися. Як наслідок, такі підпослідовності «переплітаються» між собою, взаємодіючи різними елементами і постійно змінюючи ділянки взаємодій.

Для полегшення розуміння основної ідеї розглянемо спочатку випадок, коли кількість елементів у структурі даних є степенем числа ***2***, тобто $N=2^k$.

Нехай є деяка послідовність із восьми цілих чисел (***N=8***):

7 3 6 1 5 8 2 4

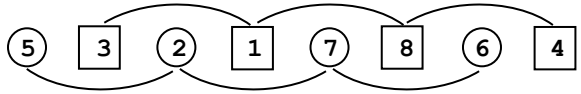
На першому етапі послідовність умовно розділяється на ***N/2=4*** підпослідовностей по два елементи в кожній із них та відстанню між елементами ***N/2=4***:



На схемі прикладу елементи однієї спільної підпослідовності поміщені в однакові геометричні фігурки та об'єднані дугами. Кожна підпослідовність окремо впорядковується *на своїх місцях якимось чином*. Це називається *сортуванням із відстанню* ***N/2***. У результаті отримали нову послідовність:

5 3 2 1 7 8 6 4

На другому етапі послідовність умовно розділяється на ***N/4=2*** підпослідовностей по чотири елементи в кожній із них та відстанню між елементами ***N/4=2***:



Кожна підпослідовність знову окремо впорядковується *на своїх місцях якимось чином*. Це називається *сортуванням із відстанню* ***N/4***. У результаті отримали нову послідовність:

2 1 5 3 6 4 7 8

І так далі.

На останньому ***k***-ому етапі розглядається уся послідовність із відстанню між елементами рівною ***1***, яка знову сортується тим самим *якимось чином*. У результаті отримали вже повністю впорядковану послідовність:

1 2 3 4 5 6 7 8

Виникають наступні питання.

Питання 1. Навіщо виконувати ***k-1*** попередні етапи сортування, якщо на останньому етапі все одно приходиться сортувати всю послідовність? **Відповідь:** на перших етапах сортується багато, але коротких підпослідовностей і загальна складність усіх цих сортувань є невеликою. На останньому етапі сортується вся послідовність, але вже суттєво «покращена» попередніми етапами. Тому складність цього етапу знову буде малою. Сумарна складність усіх етапів разом виявиться меншою, ніж при сортуванні цілої початкової послідовності.

Питання 2. Яким саме *якимось чином* потрібно сортувати підпослідовності на кожному етапі? **Відповідь:** це має бути простий у реалізації метод, складність якого по операціях порівняння і присвоєння залежить від початкового порядку елементів. Такими методами можуть бути або пряме включення, або шейкерне сортування.

Питання 3. Як обирати відстані між елементами в підпослідовностях на кожному етапі у випадку довільного числа ***N*** елементів у початковій послідовності? **Відповідь:** виявляється, при розділенні послідовності на підпослідовності не варто обирати відстані між елементами, рівними степеням деяких чисел (***2***^{t}, ***3***^{t}, ...). Більш того, відстані на двох послідовних (сусідніх) етапах взагалі не мають бути кратними.

Якщо обирати відстані по степенях числа ***2***, як це мало місце в розглянутому вище прикладі, то на всіх попередніх етапах, окрім останнього, окремо «взаємодіють» між собою елементи лише з *непарними* номерами (у прикладі на другому етапі це елементи ***5, 2, 7, 6***, що поміщені в кола) і окремо елементи лише з *парними* номерами (у прикладі на другому етапі це елементи ***3, 1, 8, 4***, що поміщені у квадрати). Тільки на останньому етапі ці дві групи елементів із непарними і парними індексами «перетнуться». Якщо спробувати обирати відстані по степенях числа ***3***, то в результаті буде окремих три групи елементів, які «не перетинатимуться» між собою аж до останнього етапу (з номерами по модулю ***1, 2, 3***). Тому відстані між елементами в підпослідовностях слід обирати з таких міркувань:

- відстані повинні зменшуватися;
- відстані на двох сусідніх етапах не повинні бути кратними;
- відстань на останньому етапі дорівнює ***1***.

Дональд Кнут запропонував дві схеми вибору відстаней, які формуються, починаючи з останнього етапу:

- схема I: …, 31, 15, 7, 3, 1,

  $V_K = 1;\quad V_t = V_{t+1} \cdot 2 + 1, t = \overline{K-1,1};\quad K = \lceil \log_2 N \rceil - 1;$
- схема II: …, 121, 40, 13, 4, 1,

$V_K = 1;\quad V_t = V_{t+1} \cdot 3 + 1,\, t = \overline{K-1,1};\quad K = \lfloor \log_3 N \rfloor - 1.$

В обох схемах ***K*** – це оптимальна кількість етапів поділу на підпослідовності зі зменшуваними відстанями між елементами. Фактично з цієї величини починається визначення відстаней.

Першу схему доцільно застосовувати для відносно коротких структур даних, коли $N \approx 100 \div 1000$, а друга схема буде більш ефективна для суттєво довгих послідовностей при $N \gg 1000$. У випадку суттєво коротких структур даних ($N < 100$) достатньо якогось із прямих методів сортування.

Тут і далі відношення $\gg$ і $\ll$ означають «значно більше» і «значно менше», а операція $\div$ позначає довільне значення з діапазону.

У якості прикладу розглянемо наступні випадки:

- ***N=500***; за схемою I кількість етапів $K = \lfloor \log_2 500 \rfloor - 1 = 8 - 1 = 7$, а відстані утворять числову послідовність: ***127, 63, 31, 15, 7, 3, 1;***
- ***N=10000***; за схемою II кількість етапів $K = \lfloor \log_3 10000 \rfloor - 1 = 8 - 1 = 7$, а відстані утворять числову послідовність: ***1093, 364, 121, 40, 13, 4, 1.***

Як видно в обох випадках, оптимальна кількість етапів виявилася однаковою. У першому випадку відносно короткої послідовності початково сортується менша кількість підпослідовностей, їх – ***127***, і в кожній по ***3*** або ***4*** елементи. У другому випадку суттєво довшої послідовності початково сортується більша кількість підпослідовностей, їх – ***1093***, і в кожній по ***9*** або ***10*** елементів. Загальна кількість основних операцій на початкових етапах в обох випадках буде незначною. Звичайно, що перший приклад потребуватиме менше операцій у порівнянні з другим прикладом.

Якщо до відносно короткої послідовності застосувати схему II, то отримаємо ***4*** етапи, і на першому з них відстань ***40***. Хоча число початкових підпослідовностей ***40*** є невеликим, проте вони містять по ***12*** або ***13*** елементів, а це вже дасть значну кількість основних операцій на першому етапі. Якщо до суттєво довгої послідовності застосувати схему I, то отримаємо ***12*** етапів, і на першому з них відстань становитиме ***4095***. Тобто буде ***4095*** підпослідовностей по ***2*** або ***3*** елементи. У цьому випадку, хоч початкові підпослідовності дуже короткі, але кількість усіх етапів є надлишковою, і це теж не є ефективним рішенням.

Програмна реалізація. Не зменшуючи загальності, пропонується варіант програми на мові C++ із використанням схеми Кнута-I, відстані якої зберігатимуться в додатковому масиві цілих чисел. Цей масив навіть для дуже великих кількостей елементів ***N*** буде потребуватиме мало пам'яті. Фактично це лише $\lfloor \log_2 N \rfloor$ значень. Тобто достатньо ***32*** елементи для структури даних, що повністю вичерпує всю статичну пам'ять програми.

```
int v[32], K, t, m, i, j, x;
K=(int)(log(N)/log(2))-1;
```

```
v[K]=1;
for (t=K-1; t>0; t--) v[t]=v[t+1]*2+1;
for (t=1; t<=K; t++)
 {
  m=v[t];
  for (i=m; i<N; i++)
   {
    x=a[i]; j=i-m;
    while (j>=0 && x<a[j])
     {
      a[j+m]=a[j];
      j=j-m;
     }
    a[j+m]=x;
   }
 }
```

Аналіз складності. Для алгоритму Шелла практично неможливо провести точний аналіз складності через достатньо довільний вибір зменшуваних відстаней між елементами. Варто зазначити, що навіть обернено впорядковані послідовності вже після першого-другого етапу значно «покращуються»: елементи з малими значеннями ключів переміщуються на початок, а елементи з великими значеннями – у кінець структури даних. А наступні етапи ще більше підготовлюють послідовність до завершального сортування.

Д. Кнут встановив таку оцінку складності по основних операціях у залежності від довжини структури даних:

$$O\left(N^{1,2}\right) \le C(N), M(N) \le O\left(N^{1,5}\right) << O\left(N^{2}\right).$$

2.2.2. Швидке сортування обміном на великих відстанях. Алгоритм Ч. Гоара – QuickSort

Ідея методу. Низька ефективність сортування прямим обміном (камінець і бульбашка) пов'язана з тим, що порівнюються і, можливо, обмінюються місцями виключно сусідні елементи. За один етап лише один елемент (найбільший або найменший) переміщується на відповідне йому місце. Решта елементів або не змінюють своєї позиції, або переміщуються лише на одну позицію, або взагалі віддаляються від свого місця в результуючій послідовності. Якщо вхідна послідовність взагалі обернено впорядкована, то переміщення найбільшого (для камінця) чи найменшого (для бульбашки) елемента через усю структуру даних на відповідне йому місце виконується з

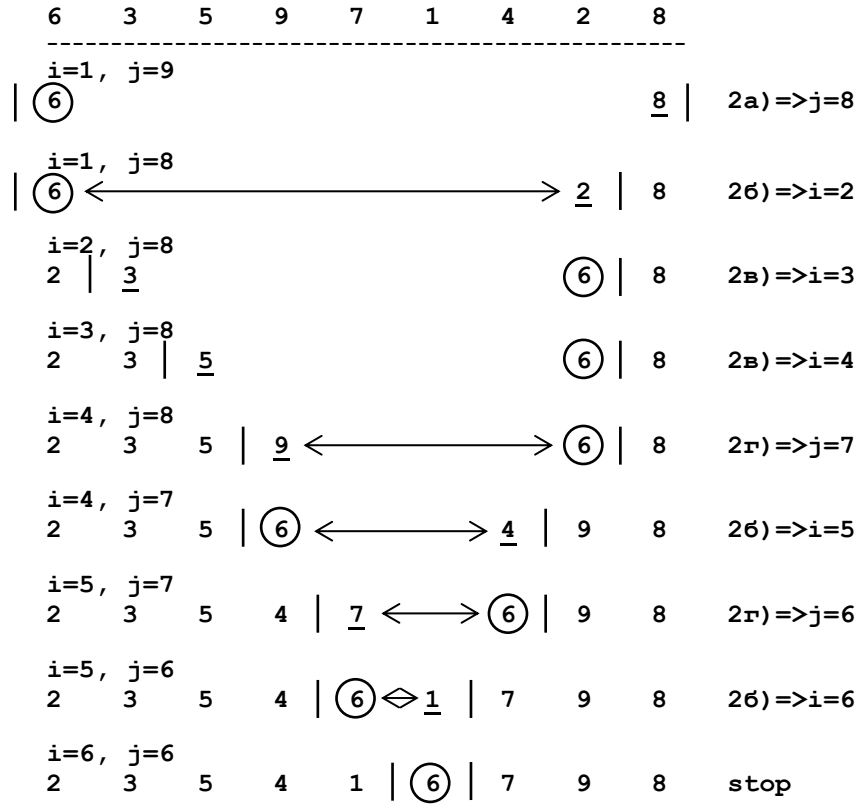
одинарним кроком і потребує дуже багато переприсвоєнь. І така ситуація повторюватиметься на кожному етапі.

Тоні Гоар (сер Чарлз Ентоні Річард Гоар) у 1960 році запропонував виконувати порівняння і можливі обміни у невпорядкованій частині між *найбільш віддаленими елементами, що ще не порівнювалися*. Початково невпорядкована частина – це вся вхідна структура даних, а найбільш віддаленими, що ще не порівнювалися, є перший і останній її елементи. Звичайно, якщо порівнювана пара елементів не є впорядкованою, то виконується обмін місцями. Наступними найбільш віддаленими елементами, що ще не порівнювалися, будуть або перший і передостанній, або другий і останній і т.д. Очевидно, що такі взаємодії у невпорядкованій частині між найбільш віддаленими елементами, що ще не порівнювалися, мають виконуватися або відносно першого елемента з усіма іншими від кінця, або відносно останнього елемента з усіма іншими від початку. Один із крайніх елементів невпорядкованої частини послідовності обирається в якості *ведучого*. Усі порівняння та можливі обміни виконуються саме між ведучим і найбільш віддаленими елементами, з якими він ще не порівнювався.

Нехай для визначеності ведучим буде перший елемент у невпорядкованій частині. Позначимо ліву межу невпорядкованої частини через ***i***, праву – через ***j***. Тоді ведучим елементом буде ***a[i]***. Для особливого виділення ведучого елемента додатково підкреслимо його позначення ***a[i]***. Зміст алгоритму Гоара, що отримав назву *QuickSort*, полягає в наступному:

1) ведучий елемент ***a[i]*** порівнюється з найбільш віддаленим елементом, з яким ще не порівнювався, тобто з ***a[j]***;
2а) поки ***i<j*** та ***a[i]≤a[j]***, виконується перехід до чергового найбільш віддаленого елемента, з яким ще не порівнювався ведучий елемент, тобто ***j=j-1***;
2б) якщо ***a[i]>a[j]***, то виконується обмін місцями між ***a[i]*** та ***a[j]***, тепер ведучим стає елемент ***a[j]***, він буде порівнюватися з найбільш віддаленим, з яким ще не порівнювався, тобто з елементом ***a[i+1]***;
2в) поки ***i<j*** та ***a[i]≤a[j]***, виконується перехід до чергового найбільш віддаленого елемента, з яким ще не порівнювався ведучий елемент, тобто ***i=i+1***;
2г) якщо ***a[i]>a[j]***, то виконується обмін місцями між ***a[i]*** та ***a[j]***, тепер ведучим знову стає елемент ***a[i]***, він буде порівнюватися з найбільш віддаленим, з яким ще не порівнювався, тобто з елементом ***a[j-1]***;
3) процес порівнянь і можливих обмінів (пункти 2а, 2б, 2в, 2г) повторюється, доки ведучий елемент не порівняється з усіма елементами невпорядкованої частини, тобто поки ***i<j***.

Приклад покрокового виконання. Подібно до попередніх методів і для полегшення розуміння ведучий елемент охоплюватимемо колом:



Проаналізуємо тепер отриманий результат. Як видно з прикладу:

- ведучий елемент ***6*** опинився на своєму місці в позиції ***i=j***;
- лівіше від ведучого опинилися усі менші від нього елементи, і їх можна сортувати окремо від решти елементів;
- правіше від ведучого опинилися усі більші від нього елементи, і їх теж можна сортувати окремо від решти елементів.

До кожного фрагмента зліва і справа від ведучого елемента знову можна застосувати такий самий прийом. Тоді їх ведучі елементи теж займуть відповідні їм місця в результуючій послідовності, і для подальшої обробки вже залишиться ***N-3*** елементи в, можливо, чотирьох фрагментах. І т.д. Поступово ведучі елементи різних етапів займатимуть відповідні їм місця. Якщо зліва чи справа від ведучого елемента якогось етапу опиниться лише один елемент, то очевидно він теж гарантовано займе відповідне йому місце. Адже він буде між двома ведучими елементами різних етапів, і при цьому одночасно більший або рівний від одного з них та менший або рівний від другого з них. Отже, такий повторюваний процес переміщення ведучих елементів на відповідні їм місця

можна завершувати, коли не залишиться жодного невпорядкованого фрагмента з двох чи більше елементів.

Якщо ведучий елемент є найменшим чи найбільшим, то він опиниться або на початку, або в кінці результуючої послідовності, і відносно нього буде лише одна невпорядкована частина, що містить ***N-1*** елемент. Якщо такий результат повторюватиметься на всіх етапах, то їх може виявитися аж ***N-1***, і на кожному з них аналізується лише один ведучий елемент, а решта елементів фактично не обробляються. Цей випадок є одним із найгірших варіантів вхідних даних. Тобто розглянутий спосіб сортування без додаткових модифікацій, на жаль, є однаково неефективним і для «хороших», і для «поганих» структур даних.

Очевидно, що найкращим випадком вхідних даних буде особливий довільний порядок елементів, при якому на кожному етапі ведучі елементи щоразу займатимуть центральні або близькі до них позиції у своїх фрагментах. Тоді для подальшої обробки залишатиметься дві частини приблизно однакової довжини, що не перевищує половини довжини початкової ділянки послідовності. Як відомо, сортувати дві частини послідовності половинної довжини більш ніж удвічі швидше, ніж сортувати цілу послідовність. У такому випадку вхідних даних кількість етапів сортування фрагментів є логарифмічною залежністю від початкової довжини.

Програмна реалізація. Ідея сортування обміном на великих відстанях потребує особливих рішень при програмній реалізації. Тому слід попередньо визначитися з рядом важливих питань.

Питання 1. Якщо після чергового етапу зліва чи справа від ведучого елемента отримали достатньо короткі фрагменти, то для подальшого їх сортування можна застосовувати якийсь із простих у реалізації і достатньо ефективний прямий метод, наприклад пряме включення. До якої мінімальної довжини фрагментів доцільно ще застосовувати алгоритм *QuickSort*, а після якої вже можна скористатися прямим методом? **Відповідь:** вибір цілком довільний, можна до кінця всі фрагменти сортувати алгоритмом *QuickSort*.

Питання 2. На всіх етапах, окрім першого, багато фрагментів одночасно потребуватимуть обробки. Точніше в поточний момент оброблятиметься якийсь один фрагмент – активний, а інші очікуватимуть своєї черги. Як організувати збереження лівої та правої меж фрагментів, що очікують на обробку? **Відповідь:** це може бути деяка додаткова стекоподібна структура даних (наприклад, два масиви індексів) або застосовуватися рекурсивний виклик алгоритму до відповідного фрагмента.

Питання 3. Який із двох, отриманих на поточному етапі, фрагментів (довший чи коротший) доцільно обробляти в першу чергу, а який можна «відкласти» на потім? **Відповідь:** спочатку доцільно обробляти довші фрагменти, оскільки вони більш ймовірно потребуватимуть саме алгоритму *QuickSort*, а коротші фрагменти можна «відкласти» на потім, і вони ймовірно оброблятимуться вже прямим методом.

Питання 4. Якщо для збереження фрагментів, що очікують обробку, використовується стекоподібна структура даних, то який має бути її розмір? **Відповідь:** максимальна кількість «відкладених» фрагментів буде, коли всі вони двоелементні, і для їх сортування використовується виключно алгоритм *QuickSort* без застосування прямого методу. Ведучі елементи попередніх етапів розмежовуватимуть такі «відкладені» фрагменти. Очевидно, їх кількість не перевищуватиме ***N/3***.

Реалізація мовою C++ алгоритму *QuickSort* із врахуванням обумовлених вище рішень може мати вигляд:

```
const int M=21;              // нижня межа застосування QuickSort
int left[N/3], right[N/3];   // додаткові стекоподібні структури даних
int k=1;                     // кількість «відкладених» фрагментів
int L, R;                    // межі поточного фрагмента
int i, j;                    // індекси порівнюваних елементів
int x;                       // значення ведучого елемента
left[k]=0; right[k]=N-1;     // спочатку обробляється вся послідовність
while (k)
 {
  L=left[k]; R=right[k]; k--;
  while (R-L+1>=M)                          // алгоритм QuickSort
   {
    i=L; j=R; x=a[i];                       // пункт 1)
    while (i<j)                             // пункт 3)
     {
      while (i<j && a[i]<=a[j]) j--;        // пункт 2а)
      if (i<j) { a[i]=a[j]; a[j]=x; i++; }  // пункт 2б)
      while (i<j && a[i]<=a[j]) i++;        // пункт 2в)
      if (i<j) { a[j]=a[i]; a[i]=x; j--; }  // пункт 2г)
     }
    if (i-L>R-i)              // підготовка до наступного етапу
     {                        // «відкласти» коротший правий фрагмент
      if (R-i>1) { k++; left[k]=i+1; right[k]=R; }
      R=i-1;                  // обробляти довший лівий фрагмент
     }
    else
     {                        // «відкласти» коротший лівий фрагмент
      if (i-L>1) { k++; left[k]=L; right[k]=i-1; }
      L=i+1;                  // обробляти довший правий фрагмент
     }
   }
  for (i=L+1; i<=R; i++)     // пряме включення для коротких фрагментів
   {
    x=a[i]; j=i-1;
```

```
    while (j>=L && x<a[j])
      { a[j+1]=a[j]; j--; }
    a[j+1]=x;
   }
 }
```

Аналіз складності. Якщо проводити аналіз складності лише по операціях порівняння без урахування присвоєння значень елементів, то виявляється, що випадок початково впорядкованої структури даних є одним із найгірших варіантів вхідних даних. Адже в цьому випадку виконується максимальна кількість ***N-1*** етапів з усіма можливими порівняннями. Звичайно, що присвоєнь буде значно менше. Якщо враховувати одне присвоєння значення ведучого елемента ***x=a[i]*** на початку кожного етапу, то

$$C_{прямий} = C_{\max} = \frac{N^2 - N}{2}, \quad M_{прямий} = N - 1.$$

Подібна ситуація і у випадку обернено впорядкованої послідовності. Відмінність буде лише з присвоєннями при обмінах: вони виконуються лише однократно при першому порівнянні і через один етап. Тому

$$C_{обернений} = C_{\max} = \frac{N^2 - N}{2}, \quad M_{обернений} = \frac{3(N-1)}{2} + \frac{N-1}{2} = 2(N-1).$$

Така велика кількість операцій порівняння пов'язана з тим, що ведучі елементи на кожному етапі є або найменшими, або найбільшими, і щоразу займатимуть крайні позиції у своїх фрагментах. У результаті відносно них зліва і справа вже не формується два фрагменти значно меншої довжини, а залишається один фрагмент із кількістю елементів на одиницю менше.

Найкращим варіантом вхідних даних буде така послідовність елементів, для якої ведучі елементи на кожному етапі займатимуть центральну позицію у своєму фрагменті. При цьому число етапів визначатиметься логарифмічною залежністю. Загалом можна вважати, що складність по порівняннях і присвоєннях матиме однаковий порядок:

$$C_{\min}(N), M_{\min}(N) = O\big(N \cdot \log(N)\big).$$

2.2.3. Швидке сортування вибором за допомогою піраміди. Алгоритм HeapSort

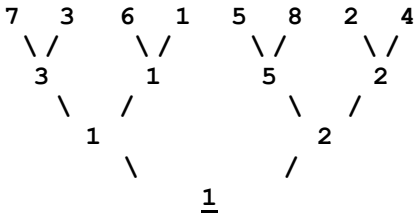
Ідея методу. Низька ефективність алгоритму прямого вибору пов'язана з обов'язковим виконанням на кожному етапі максимально можливої кількості операцій порівняння. Спрощення алгоритму не передбачає збереження і використання на наступних етапах тієї корисної інформації про елементи структури даних – потенційні кандидати на вибір для наступних етапів, – яка накопичується при пошуку найменшого (найбільшого) ключа у

невпорядкованій частині на поточному етапі. Тому для пришвидшення процесу сортування вибором потрібно якомога повніше використовувати всю додаткову інформацію з попередніх етапів.

Знову, не обмежуючи загальності, розглянемо спочатку випадок структури даних із восьми цілих чисел (***N=8***):

7 3 6 1 5 8 2 4

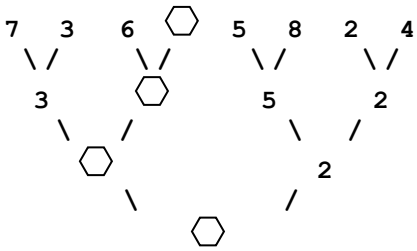
При першому проході вибираємо менший із кожної пари сусідніх елементів. На другому проході вибираємо менший із кожної пари сусідніх елементів, що були вибрані при попередньому проході. Аналогічним чином поступаємо при кожному наступному проході. При цьому на кожній наступній серії виборів аналізується вдвічі менше елементів, ніж при попередній. І так далі. На останньому проході вибирається менший із двох раніше вибраних елементів – найменший елемент у структурі даних:



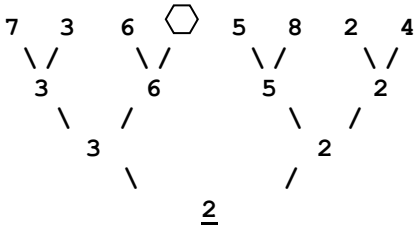
У результаті отримали деревоподібну структуру даних, у кореневій вершині якої знаходиться найменший елемент ***1***. При побудові дерева та ідентифікації найменшого ключа виконано $C_1 = \frac{N}{2} + \frac{N}{4} + \ldots + 2 + 1 = N - 1$ операцій порівняння, що співпадає з класичним алгоритмом пошуку.

Для подальшої реалізації алгоритму вибору застосуємо інший підхід, що передбачає використання додаткової корисної інформації та потребує меншого числа порівнянь. З цією метою виконаємо такі дії:

- запишемо знайдене найменше значення ключа в деяку результуючу послідовність;
- вилучимо мінімальний елемент із побудованого дерева, замінивши всі його входження особливим елементом – «діркою»:



- перерахуємо заново дерево, вважаючи, що всі «дірки» мають як завгодно велике значення. Перерахунок стосується лише елементів-«дірок» – по одному елементу на кожному рівні, окрім першого;
- у результаті в кореневу вершину потрапить наступний по порядку найменший елемент із тих, що залишилися:



У кореневій вершині нового дерева опинився наступний найменший елемент. Для його ідентифікації виконано $C_2 = \log N$ операцій порівняння, що значно менше, ніж за класичним алгоритмом пошуку для $N-1$ елемента ($C_2^{клас.} = N-2$). З отриманим деревом знову виконуємо ті самі дії: знайдений елемент із найменшим ключем переписуємо в результуючу послідовність, вилучаємо цей елемент із дерева, замінюємо всі його входження «дірками», перераховуємо нове дерево і т.д. При цьому щоразу потрібно лише $\log N$ операцій порівняння.

У результаті виконання $\boldsymbol{N}$ етапів вибору в дереві залишаться одні «дірки», і отримаємо впорядковану результуючу послідовність. Загальна кількість операцій порівняння буде:

$$C = C_{\boldsymbol{1}} + C_{\boldsymbol{2}} + ... + C_{N-\boldsymbol{1}} + C_N = N - \boldsymbol{1} + (N - \boldsymbol{1}) \cdot \log N = O(N \cdot \log N) << O(N^{\boldsymbol{2}}).$$

На жаль, значний виграш по кількості порівнянь потребує величезних затрат пам'яті. Адже для побудови дерева вибору потрібно розмістити в пам'яті $K_{дерево} = N + \frac{N}{2} + ... + 2 + 1 = 2N - 1$ елементів даних. Окрім цього, потрібно ще $\boldsymbol{N}$ додаткових елементів для результуючої послідовності. Таким чином, потрібно мати порядку $\boldsymbol{3N}$ елементів даних, а це вже не відповідає базовому принципу алгоритмів обробки структур даних «на тому ж місці».

Отже, слід модифікувати розглянутий алгоритм вибору так, щоб при загальній кількості операцій порівняння порядку $O(N \cdot \log N)$ не використовувати додаткових ресурсів пам'яті, тобто потрібна особлива організація даних за принципом дерева, яка б вимагала $\boldsymbol{N}$ одиниць пам'яті.

Цього вдалося добитися в алгоритмі *HeapSort*, який розробив Джон Вільям Джозеф Вільямс. Він використав спеціальну деревоподібну структуру даних – *піраміду* (англ. – *Heap*), яка дозволяє швидко ідентифікувати

найменший (найбільший) елемент без використання додаткових ресурсів пам'яті.

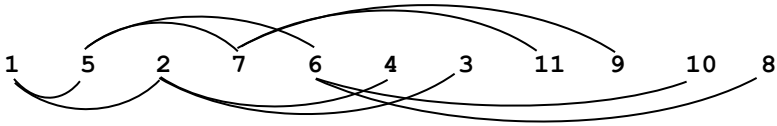
Піраміда – це особливим чином організована послідовність елементів, для якої виконуються один із двох варіантів умов:

$$a_i \le a_{2i} \quad \wedge \quad a_i \le a_{2i+1} \quad , \quad i = \overline{1, N/2}\,; \tag{1}$$

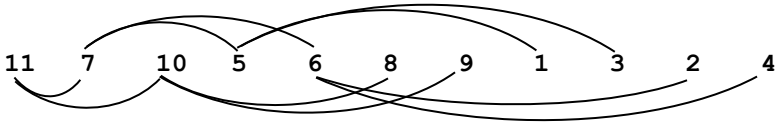
$$a_i \ge a_{2i} \quad \wedge \quad a_i \ge a_{2i+1} \quad , \quad i = \overline{1, N/2}\,. \tag{2}$$

У піраміді, що визначається умовами (1), перший елемент є найменшим, а у піраміді виду (2) перший елемент – найбільший. Піраміда не обов'язково є впорядкованою структурою даних, але перший її елемент завжди або найменший, або найбільший у залежності від визначальних умов (1) чи (2). Взагалі кажучи, з одного і того ж набору елементів (початкової послідовності) можна побудувати багато різних варіантів пірамід.

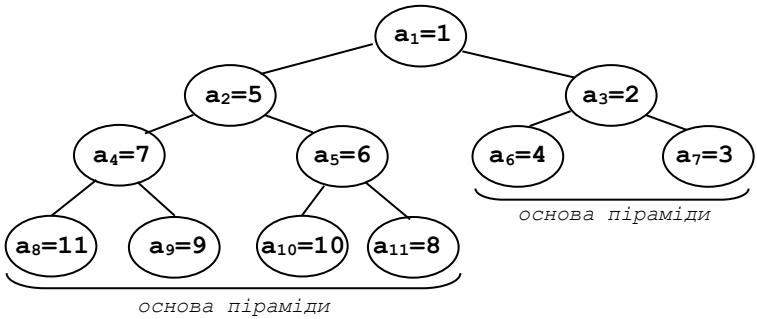
Приклад піраміди за умовами (1):



Приклад піраміди за умовами (2):



Піраміду можна зображати у вигляді бінарного дерева:



Нижні елементи такого дерева, які не мають прилеглих їм згідно з правилом піраміди (1) чи (2), утворюють *основу піраміди*.

Нам потрібно сформулювати ефективний алгоритм побудови піраміди для довільної послідовності елементів. Як вже зазначалося вище, на основі заданого набору елементів можна побудувати декілька різних пірамід. Для нас

важливим є один перший елемент піраміди, а решта можуть розміститися достатньо довільно, але згідно з умовами (1) чи (2). Тому зупинимося на якомусь одному варіанті піраміди.

Не зменшуючи загальності, розглянемо варіант піраміди, що визначається умовами (2), тобто з першим найбільшим елементом. Хоч цей вибір довільний, проте в подальшому він виявиться оптимальнішим при реалізації алгоритму повного сортування.

Піраміда має властивість розширюватися вліво або вгору (піраміда як бінарне дерево) при введенні в таку структуру даних нових елементів. Роберт Флойд запропонував алгоритм побудови піраміди «на тому ж місці»:

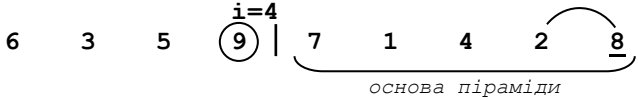
- оскільки визначальні умови піраміди (2) стосуються лише елементів $a_1, a_2, ..., a_{N/2-1}, a_{N/2}$, то всі наступні елементи $a_{N/2+1}, a_{N/2+2}, ..., a_{N-1}, a_N$ можна відразу вважати основною піраміди незалежно від їх значень;
- кожен новий елемент добавляється в піраміду зліва в такому порядку: $a_{N/2}, a_{N/2-1}, ..., a_2, a_1$; при цьому виконується порівняння кожного нового елемента a_i, $i = \overline{N/2,1}$ із прилеглими до нього згідно з умовами піраміди (2) елементами a_{2i}, a_{2i+1}:
  - ✓ якщо новий елемент a_i не менший від більшого з двох прилеглих до нього елементів a_{2i}, a_{2i+1}, тобто $a_i \geq \max\{a_{2i}, a_{2i+1}\}$, то згідно з умовами піраміди (2), новий елемент a_i уже займає відповідну йому позицію в піраміді;
  - ✓ якщо новий елемент a_i менший від більшого з двох прилеглих до нього елементів a_{2i}, a_{2i+1}, тобто $a_i < \max\{a_{2i}, a_{2i+1}\}$, то виконується обмін місцями між новим a_i та більшим із двох прилеглих. У результаті новий елемент переміщується вглиб піраміди вправо або вниз (піраміда як бінарне дерево), його індексний номер змінюється, і в нього будуть вже інші прилеглі елементи;
  - ✓ процес руху кожного нового елемента вглиб піраміди може бути багатокроковий. Після чергового обміну порівняння нового елемента з прилеглими до нього елементами і можливі обміни повторюються, поки він не займає відповідне йому місце в розширеній піраміді: або для нього виконуються умови піраміди (2), або він переміщується в основу піраміди.

Приклад покрокової побудови піраміди. Розглянемо послідовність:
6, 3, 5, 9, 7, 1, 4, 2, 8.

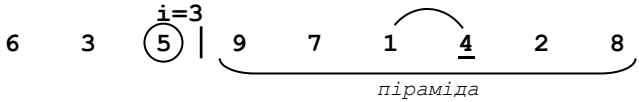
Кількість елементів $N = 9$. Тому основу піраміди складатимуть п'ять останніх елементів: ***7, 1, 4, 2, 8***. Нові елементи будуть вводитися в піраміду в такому порядку: ***9, 5, 3, 6.***

Для зручності ілюстрації процесу побудови піраміди використані наступні позначення:

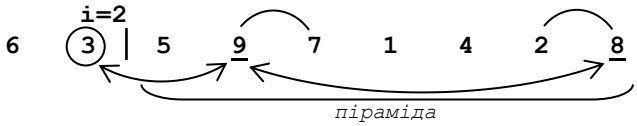
- новий елемент, що вводиться в піраміду, охоплений колом ◯;
- пара прилеглих елементів зверху сполучена дугою ⌒;
- більший із двох прилеглих елементів – підкреслений;
- на переміщення елементів вказують двосторонні дугові стрілки ↩ :



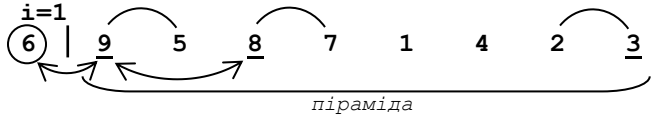
*виконано **2** порівняння: вибір більшого з двох прилеглих та порівняння його з новим;*
*новий елемент **9** без обмінів розширив піраміду зліва;*



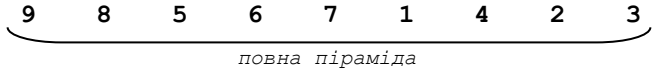
*виконано **2** порівняння: вибір більшого з двох прилеглих та порівняння його з новим;*
*новий елемент **5** без обмінів розширив піраміду зліва;*



*виконано **4** порівняння: двічі вибір більшого з двох прилеглих та порівняння його з новим;*
*новий елемент **3** виконав два переміщення;*



*виконано **6** порівнянь: тричі вибір більшого з двох прилеглих та порівняння його з новим;*
*новий елемент **6** виконав два переміщення;*



У результаті побудови піраміди на початок послідовності перемістився найбільший елемент ***9***. При цьому було виконано $C_1 = 2 + 2 + 4 + 6 = 14$ операцій порівняння. Як видно, є певний програш у порівнянні з класичним алгоритмом пошуку найбільшого, який потребує $N - 1 = 8$ операцій. Проте пошук наступного найбільшого елемента вже буде значно легшим, і всі наступні етапи потребуватимуть суттєво меншої кількості порівнянь.

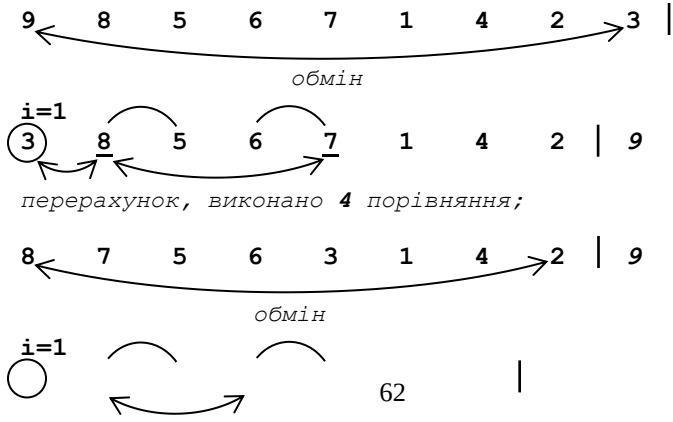
Повне сортування пірамідою. Побудова початкової піраміди – це лише перша фаза сортування. Вона дозволила певним чином організувати структуру даних й одночасно ідентифікувати найбільший елемент, що опинився першим.

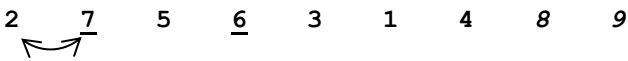
Як шукати наступний найбільший елемент і куди подіти попередньо знайдений? Щоб забезпечити обробку структур даних із дотриманням принципу «на тому ж місці» потрібно виконати такі дії:

- проводиться обмін місцями першого найбільшого та останнього елементів піраміди;
- піраміда «вкорочується» на один елемент від кінця;
- піраміда перераховується лише для одного нового першого елемента.

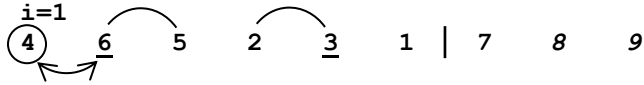
У результаті повторюваного виконання таких дій піраміда вкорочуватиметься на один елемент справа, поки не залишиться один єдиний перший елемент. При цьому від кінця до початку структури даних буде формуватися впорядкована послідовність елементів: останнім буде найбільший елемент, що був ідентифікований при побудові початкової піраміди, а першим буде найменший елемент, що складатиме останню найкоротшу піраміду. Тепер стало зрозуміло, чому на початку ми обрали другий варіант піраміди з першим найбільшим елементом. Якщо за основу алгоритму брати піраміду виду (1) із першим найменшим елементом, то в результаті всіх дій отримаємо обернено впорядковану послідовність. Тоді для прямої впорядкованості прийдеться додатково виконувати *реверс* послідовності – перезапис елементів у протилежному порядку.

Приклад покрокового виконання повного сортування. За основу візьмемо раніше побудовану піраміду:

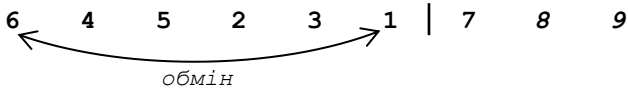


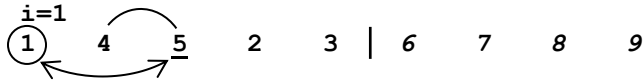


*перерахунок, виконано **4** порівняння;*

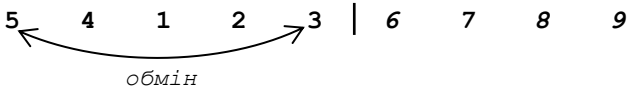


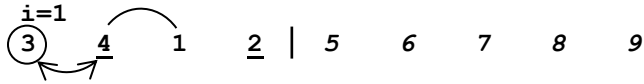
*перерахунок, виконано **4** порівняння;*



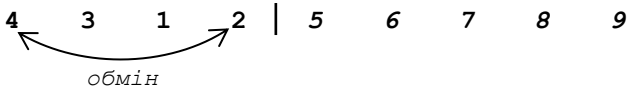


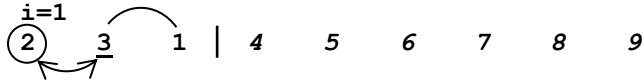
*перерахунок, виконано **2** порівняння;*



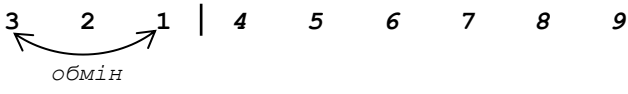


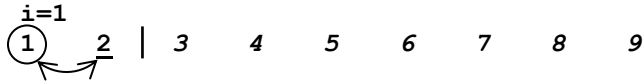
*перерахунок, виконано **3** порівняння: вибір більшого з двох прилеглих **4**, порівняння його з новим елементом **3**, після обміну – порівняння з новим прилеглим **2**;*





*перерахунок, виконано **2** порівняння;*





*перерахунок, виконано **1** порівняння;*

```
2    1 | 3    4    5    6    7    8    9
 ↖ ↗
обмін

-------------------------------------------------
1    2    3    4    5    6    7    8    9    stop,Ok!
після останнього обміну одноелементна піраміда вже не
перераховується, кінець сортування.
```

У результаті отримали впорядковану послідовність і на це витратили $C_{II} = 4+4+4+2+3+2+1 = 20$ порівнянь. Загальна кількість операцій порівняння з врахуванням фази побудови початкової піраміди $C = C_I + C_{II} = 14 + 20 = 34$. Якщо вибір найбільшого елемента на кожному етапі виконувати за класичним алгоритмом, то потрібно $C = 8+7+\ldots+2+1 = 36$ порівнянь. Як видно, алгоритм швидкого сортування вибором за допомогою піраміди є більш ефективним у порівнянні з прямим вибором. Така перевага буде тим більш суттєвою, чим довша структура даних обробляється.

Програмна реалізація.

Основою алгоритму сортування є операція *перерахунку* розширеної піраміди при переміщенні нового елемента на відповідне йому місце – свого роду «просіювання» нового елемента через піраміду. Ця операція використовується і на першій фазі при побудові початкової піраміди, і на фазі повного сортування структури даних при перерахунку вкорочених пірамід. Тому, як і в авторському варіанті, цю операцію доцільно реалізувати окремою підпрограмою ***Sift*** з двома вхідними параметрами індексного типу, що визначають ліву і праву межі розширеної піраміди. Таким чином, новий елемент, що вводиться в піраміду зліва, індексуватиметься значенням першого параметра цієї підпрограми.

Програмна реалізація операції просіювання нового елемента через піраміду на мові С++ потребує незначної корективи умов піраміди (1) і (2). Це пов'язано з тим, що в мові С++ індексація масивів починається з нуля, тобто перший елемент має індексний номер ***0***. Визначальні умови піраміди (1) і (2) з врахуванням вказаних особливостей індексації можна записати так:

$$a_i \le a_{2i+1} \ \wedge \ a_i \le a_{2i+2} \quad , \quad i = \overline{0, N/2-1}; \qquad (3)$$

$$a_i \ge a_{2i+1} \ \wedge \ a_i \ge a_{2i+2} \quad , \quad i = \overline{0, N/2-1}. \qquad (4)$$

Таким чином, програмна реалізація операції перерахунку розширеної піраміди на підставі умов (4) може мати вигляд функції:

```
void Sift(int L, int R)
 {
   int i=L, j=2*L+1, x=a[L];
```

```
  if (j<R && a[j+1]>a[j]) j++;
  while (j<=R && x<a[j])
   {
     a[i]=a[j]; a[j]=x;
     i=j; j=2*i+1;
     if (j<R && a[j+1]>a[j]) j++;
   }
}
```

Процес сортування передбачає повторюваний виклик функції ***Sift*** із різними наборами вхідних параметрів. На першій фазі побудови початкової піраміди індекс правої межі піраміди є фіксованим ***R=N-1***, а ліва межа ***L*** при кожному розширенні зміщується вліво на одну позицію, послідовно набуваючи значень $N/2-1, N/2-2, \ldots, 1, 0$. На другій фазі повного сортування при перерахунку вкорочених пірамід індекс лівої межі піраміди є фіксованим ***L=0***, а права межа ***R*** при кожному вкороченні зміщується вліво на одну позицію, послідовно набуваючи значень $N-2, N-3, \ldots, 2, 1$. Остання найкоротша піраміда з одного найменшого елемента вже не потребує перерахунку. Проте доцільно виконати ще одну ітерацію заради обміну місцями першого й останнього елементів у поточній піраміді, який виконується перед перерахунком вкороченої піраміди.

Програмна реалізація алгоритму *HeapSort* може мати вигляд:

```
int L=N/2, R=N-1, x;
while (L>0)
 {
   L--; sift(L,R);
 }
while (R>0)
 {
   x=a[L]; a[L]=a[R]; a[R]=x;
   R--; Sift(L,R);
 }
```

Аналіз складності. Побудова початкової піраміди передбачала можливе переміщення нових елементів углиб структури даних. Очевидно, що початковий порядок елементів у послідовності визначає кількість кроків заглиблення, а отже, і число операцій порівняння. Якщо послідовність є «хорошою», то при побудові піраміди за умовами (2) кожен новий елемент здійснюватиме максимальні багатокрокові переміщення вглиб піраміди. Число кроків визначатиметься логарифмічною залежністю, і на кожному кроці виконуватиметься по два порівняння (вибір більшого з двох прилеглих елементів та порівняння його з новим елементом). У цьому випадку кількість

операцій порівняння на першій фазі не перевищуватиме величини $N \cdot logN$. Якщо послідовність є «поганою», то вона може відразу утворювати піраміду виду (2) або бути близькою до неї. Тоді більшість або навіть усі нові елементи не будуть переміщуватися вглиб піраміди. У цьому випадку кількість операцій порівняння на першій фазі буде величиною порядку $O(N)$.

При перерахунку вкорочених пірамід на фазі повного сортування кількість операцій порівняння практично не залежить від початкового порядку елементів у послідовності. Адже початкові піраміди, отримані для різних варіантів порядку вхідних даних, будуть досить подібними. Часова складність цієї частини алгоритму залежить від довжини структури даних. Кількість операцій порівняння буде величиною порядку $O(N \cdot logN)$ і перевищуватиме аналогічну складність фази побудови початкової піраміди з точністю до скалярного множника, більшого від одиниці.

Таким чином, загальна складність по операціях порівняння є величиною такого самого порядку $C = O(N \cdot logN)$.

Кількість операцій присвоєння залежить від початкового порядку елементів. Загалом ця характеристика складності алгоритму має той самий порядок, що і порівняння. Однак додатково виконується ще $3 \cdot N$ операцій присвоєння при обмінах першого та останнього елементів кожної піраміди перед їх вкороченням і перерахунком. Отже, загальна складність по операціях присвоєння є величиною порядку $M = O(N \cdot logN) + 3 \cdot N$.

Враховуючи все вище сказане, можна підсумувати:

- алгоритм *HeapSort* сортування швидким вибором за допомогою піраміди є *універсальним*, тобто його ефективність у різних випадках порядку елементів у вхідних послідовностях відрізняється *незначно*;
- у випадку «хороших» структур даних часова складність по порівняннях і присвоєннях незначно гірша, ніж у випадках «поганих» вхідних послідовностей;
- випадки довільних невпорядкованих структур даних займають проміжне місце: вони трохи гірші від «поганих» вхідних послідовностей і трохи кращі від «хороших» вхідних послідовностей.

2.3. СОРТУВАННЯ ПОСЛІДОВНОСТЕЙ

Для структур даних з послідовним доступом (послідовностей) розглянуті раніше алгоритми сортування в більшості випадків є не застосовуваними. Це пов'язано з тим, що майже всі вони передбачають порівняння і переприсвоєння віддалених елементів. Як виключення, для послідовностей можна застосовувати різновид прямого обміну – метод камінця, оскільки цей метод передбачає послідовний доступ до елементів із кроком +1. Спроба використати прямі алгоритми включення або вибору потребуватиме їх модифікацій і використання додаткової результуючої послідовності.

Для сортування послідовностей використовують інший визначальний принцип – *злиття впорядкованих підпослідовностей*.

Не обмежуючи загальності, у якості послідовності розглядатимемо масив із послідовним доступом до елементів із кроком +1.

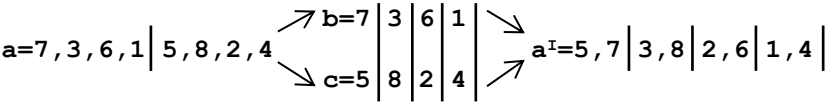
2.3.1. АЛГОРИТМ ПРЯМОГО ЗЛИТТЯ

Ідея методу. З метою спростити і полегшити розуміння принципу сортування розглянемо спочатку випадок, коли число елементів у структурі даних є степенем числа ***2***, тобто $N=2^k$.

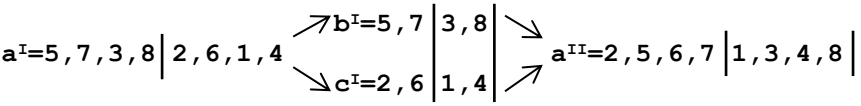
Нехай є послідовність із восьми цілих чисел ($N=8$):

a={7, 3, 6, 1, 5, 8, 2, 4} .

На 1-ому етапі вхідна послідовність шляхом послідовного перезапису розділяється на дві підпослідовності однакової довжини. З кожної підпослідовності послідовно зчитуються одноелементні впорядковані фрагменти і *зливаються* у впорядковані двоелементні фрагменти, які потім послідовно записуються назад у результуючу послідовність:

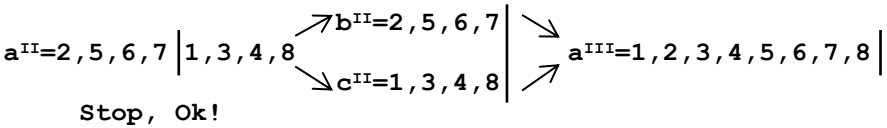


На 2-ому етапі отримана послідовність шляхом послідовного перезапису знову розділяється на дві підпослідовності однакової довжини. З кожної підпослідовності послідовно зчитуються двоелементні впорядковані фрагменти і *зливаються* у впорядковані чотириелементні фрагменти, які потім послідовно записуються назад у результуючу послідовність:



І так далі.

На останньому ***k***-ому етапі отримана послідовність шляхом послідовного перезапису знову розділяється на дві підпослідовності однакової довжини. Кожна з цих підпослідовностей – упорядкований фрагмент із ***N/2*** елементів, отриманий у результаті злиття на попередньому етапі. З кожної підпослідовності послідовно зчитуються ***N/2***-елементні впорядковані фрагменти, вони *зливаються* і записуються у впорядковану результуючу послідовність із ***N*** елементів:



Розглянутий алгоритм передбачає використання трьох ліній елементів ***a, b, c***. При цьому кожен етап складається з двох фаз:

- *фаза розділення* послідовності на дві підпослідовності;
- *фаза злиття* впорядкованих фрагментів із двох підпослідовностей у впорядковані фрагменти сумарної довжини та послідовний їх запис у результуючу послідовність.

Даний спосіб сортування називають *двофазним трилінійним злиттям*. На кожній фазі розділення щоразу виконується ***N*** операцій запису (присвоєння) без порівнянь елементів і врахування їх взаємного порядку, тобто така фаза є непродуктивною.

З метою підвищення ефективності можна відмовитися від непродуктивної фази розділення, але при цьому прийдеться використовувати додаткову четверту лінію елементів. У цьому випадку вхідні дані відразу розділяються і послідовно записуються в перші дві лінії, які вважаються вхідними (умовно позначимо їх через ***a*** і ***b***). На першому етапі з цих ліній зчитуються впорядковані одноелементні фрагменти, вони зливаються у впорядковані двоелементні фрагменти і *почергово* записуються в інші дві лінії, які вважаються результуючими (умовно позначимо їх через ***c*** і ***d***). На наступному етапі напрям читання впорядкованих коротших фрагментів та запису результату їх злиття у довші фрагменти змінюється на протилежний: старі результуючі лінії ***c*** і ***d*** стають вхідними, а старі вхідні ***a*** і ***b*** стають результуючими. Цей процес зі зміною напряму операцій читання-злиття-запису повторюється до тих пір, поки в якійсь із ліній ***a*** чи ***c*** не опиниться вся впорядкована послідовність. Такий спосіб сортування називається *однофазним чотирилінійним злиттям*, оскільки фактично має місце лише одна фаза злиття і використовується чотири лінії елементів.

Звичайно, що всі операції порівняння і запису в цьому випадку будуть продуктивними, однак потрібно резервувати в пам’яті аж чотири структури даних із ***N*** елементів. Точніше кажучи, такими довгими мають бути лінії ***a*** і ***c***, в одній із яких може опинитися результуюча впорядкована послідовність. Лінії

b і ***d*** можуть бути з ***N/2*** елементів за умови, що на кожному етапі почерговий запис результатів злиття починатиметься саме з лінії ***a*** чи ***c***.

Обидва варіанти алгоритму сортування послідовностей за принципом злиття потребують значних затрат пам'яті. Тому говорити про дотримання принципу сортування «на цьому ж місці» не приходиться. Проте слід враховувати, що в якості послідовностей часто розглядаються файли послідовного доступу (файли даних). Вони розміщуються в практично необмеженій, але відносно повільній зовнішній пам'яті – на дисках. Тому вигідніше використати додаткові ресурси пам'яті, ніж виконувати багато операцій читання-запису даних і переміщення файлового вказівника.

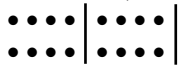
Виникає ще декілька важливих питань.

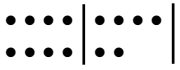
Питання 1. Як здійснювати розділення послідовності на підпослідовності і як вибирати впорядковані фрагменти у випадку структури даних довільної довжини, а не при $N=2^k$?

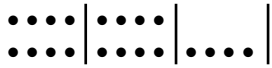
Відповідь. Поділ на дві підпослідовності однакової довжини або з різницею в один елемент (якщо ***N*** непарне) може бути лише на першому етапі, оскільки на цьому етапі зливаються одноелементні впорядковані фрагменти. На всіх наступних етапах поділ має здійснюватися точно між двома впорядкованими фрагментами, отриманими на попередньому етапі. При поділі важливим є не однакова чи майже однакова кількість елементів у підпослідовностях, а однакова чи майже однакова кількість впорядкованих фрагментів. Нехай для визначеності в першу додаткову лінію ***b*** переходить ціла частина від поділу загального числа фрагментів на ***2***.

Коли кількість елементів у послідовності не є степенем за основою ***2***, то на деякому етапі в кінці послідовності опиниться неповний впорядкований фрагмент. Після фази розділення він завершуватиме другу додаткову лінію ***c***. Після фази злиття в кінці результуючої лінії ***a*** знову буде неповний впорядкований фрагмент. Це або результат злиття неповного фрагмента додаткової лінії ***c*** та повного фрагмента додаткової лінії ***b*** (якщо кількість фрагментів у додаткових лініях була однакова), або неповний фрагмент додаткової лінії ***c*** без змін (якщо кількість фрагментів у додаткових лініях відрізнялася на одиницю).

Допустимі варіанти поділу схематично можна зобразити так (повний фрагмент із 4 елементів):

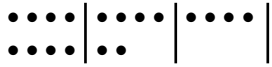
а) 

б) 

в) 

г) 

Недопустимий варіант поділу:



Питання 2. Як реалізувати операцію злиття впорядкованих фрагментів?
Відповідь. Для злиття впорядкованих фрагментів виконуються такі дії:

- результуюча лінія ***a*** звільняється від елементів попереднього етапу;
- з фрагментів додаткових ліній ***b*** і ***c*** послідовно беруться елементи та порівнюються між собою;
- якщо меншим є елемент лінії ***b***, то цей елемент записується в результуючу лінію ***a***, у лінії ***b*** здійснюється перехід на наступний елемент, порівняння елементів ліній ***b*** і ***c*** повторюється;
- якщо меншим є елемент лінії ***c***, то цей елемент записується в результуючу лінію ***a***, у лінії ***c*** здійснюється перехід на наступний елемент, порівняння елементів ліній ***b*** і ***c*** повторюється;
- якщо одна з додаткових ліній ***b*** чи ***c*** вичерпана до кінця, то залишок елементів фрагмента другої лінії без порівнянь переписується в результуючу лінію ***a***.

Програмна реалізація. Не зменшуючи загальності, зупинимося на варіанті двофазного трилінійного злиття. Модифікацію програми для однофазного чотирилінійного злиття залишаємо читачу.

Спочатку розглянемо програмну реалізацію операції злиття впорядкованих фрагментів ліній ***b*** і ***c*** у впорядкований фрагмент лінії ***a*** у вигляді окремої підпрограми-функції ***zlyttya***. У якості параметрів цій функції передаються п’ять цілочисельних індексів: початок і кінець фрагмента лінії ***b***, початок і кінець фрагмента лінії ***c***, індекс поточного елемента результуючої лінії ***a.*** Останній параметр передаватиметься як посилання, оскільки позиція після запису останнього елемента впорядкованого фрагмента фактично є позицією для запису першого елемента наступного впорядкованого фрагмента, вона потрібна для нового виклику функції ***zlyttya***.

```
void zlyttya (int bp, int bk, int cp, int ck, int & ap)
 {
  while (bp<=bk && cp<=ck)
   if (b[bp]<c[cp])
     a[ap++]=b[bp++];
   else
     a[ap++]=c[cp++];
  while (bp<=bk) a[ap++]=b[bp++];
  while (cp<=ck) a[ap++]=c[cp++];
 }
```

Процес сортування всієї послідовності передбачає повторюване застосування функції ***zlyttya*** над двома лініями ***b*** і ***c*** зі зміщенням на величину довжини повного фрагмента для даного етапу. Фактичними параметрами виклику будуть індекси, що обмежують поточні фрагменти в кожній лінії.

На кожному поточному етапі розділення на дві підпослідовності потрібно проводити строго між двома впорядкованими фрагментами, отриманими на попередньому етапі. При цьому може бути не більше ніж один неповний фрагмент і лише останнім. Таким чином, потрібно оперувати наступними величинами:

- ***d_fr=1, 2, 4, 8, …*** – довжина впорядкованого фрагменту на поточному етапі;
- ***k_fr*** – загальна кількість повних і, можливо, одного неповного фрагментів у послідовності;
- ***k_fr_b*** – кількість повних фрагментів, що попадає при розділенні в лінію ***b***;
- ***k_el_b*** – кількість елементів, що попадає при розділенні в лінію ***b***.

Циклічний процес злиттів впорядкованих фрагментів зі збільшенням їх довжин завершується, коли довжина фрагмента досягає або перевищує довжину послідовності ***N***.

Програмна реалізація повного сортування може мати вигляд:

```
int d_fr=1, k_fr, k_fr_b, k_el_b, i, ap, bp, bk, cp, ck;
while (d_Fr<N)
 {
  k_fr=N/d_fr; if (N%d_fr) k_fr++;
  k_fr_b=k_fr/2; k_el_b=k_fr_b*d_fr;
     // фаза розділення
  for (i=0; i<k_el_b; i++) b[i]=a[i];
  for ( ; i<N ; i++) c[i-k_el_b]=a[i];
     // фаза злиття
  ap=0; bp=0; bk=d_fr-1; cp=0; ck=d_fr-1;
  if (ck>N-1-k_el_b) ck=N-1-k_el_b;
  while (bk<k_el_b)
   {
    zlyttya(bp, bk, cp, ck, ap);
    bp+=d_fr; bk+=d_fr; cp+=d_fr; ck+=d_fr;
    if (ck>N-1-k_el_b) ck=N-1-k_el_b;
   }
  d_fr*=2;
 }
```

<u>Аналіз складності</u>. Часова складність алгоритму, тобто загальна кількість операцій порівняння і присвоєння, залежить від кількості етапів злиття, а фактично – від довжини послідовності ***N***. Кількість етапів визначається логарифмічною залежністю і дорівнює $K_{етапів} = \lceil \log(N) \rceil + 1$.

Число операцій присвоєння на всіх етапах є фіксованою і не залежить від початкового порядку елементів: $M_{етап} = N_{розділ.} + N_{злиття} = 2 \cdot N$.

Тоді загальна кількість присвоєнь буде:

$$M = K_{етапів} \cdot M_{етап} = 2 \cdot N \cdot log(N).$$

Кількість операцій порівняння на кожному етапі залежить від початкового порядку елементів. Найкращим випадком буде той, для якого на всіх етапах при кожному злитті впорядкованих фрагментів перший найменший елемент одного фрагмента, послідовно порівнюючись з усіма елементами другого фрагмента, виявиться не меншим від останнього найбільшого елемента другого фрагмента. Тоді в результуючій лінії спочатку опиниться весь другий фрагмент, а перший допишеться після нього. У цьому випадку за етап виконується приблизно $N/2$ порівнянь. А загальне число таких операцій буде:

$$C_{min} = \frac{N}{2} \cdot log(N).$$

Найгіршим випадком буде той, для якого на всіх етапах при кожному злитті впорядкованих фрагментів у результуючу лінію елементи з обох фрагментів писатимуться почергово. Тоді кожен елемент однієї лінії фактично порівнюватиметься двічі: спочатку він виявиться більшим, а потім меншим чи навпаки. У цьому випадку за етап виконується приблизно N порівнянь. А загальне число таких операцій буде:

$$C_{max} = N \cdot log(N).$$

Варто зазначити, що прямо впорядкована, так само, як і обернено впорядкована послідовності, є одними з найгірших випадків вхідних даних. На початкових етапах впорядкована послідовність добряче «псується», а потім на завершальних етапах – «покращується».

2.3.2. Модифікація прямого злиття – злиття впорядкованих серій

Ідея методу. Аналіз складності алгоритму прямого злиття показав проблемні моменти сортування «хороших» послідовностей. Початкові етапи виявляються контрпродуктивними – впорядкована або майже впорядкована послідовність перетворюється до найгіршого порядку елементів, а потім ситуація виправляється з максимальною кількістю операцій порівняння.

Для покращення ефективності доцільно проводити попередній аналіз послідовності, виділяючи всі впорядковані підпослідовності різної довжини. Такі впорядковані фрагменти називатимемо *впорядкованими серіями*.

Чим менша кількість серій, тим переважно вони є довшими, і навпаки. Очевидно, у випадку «хорошої» послідовності буде мало, але достатньо довгих серій, а у випадку «поганої» послідовності буде багато коротких, можливо, одноелементних серій.

Злиття двох серій породжує одну серію сумарної довжини. Таким чином, кожен етап приводить до зменшення числа серій при зростанні їх довжин. Сортування завершиться за умови формування єдиної серії.

Якщо початкова кількість серій є малою, то процес обробки завершиться швидше зі значно меншою кількістю етапів. Прямо впорядкована послідовність – єдина серія максимальної довжини. У цьому випадку сортування взагалі не потрібне.

Виникає питання, як саме потрібно модифікувати фази розділення і злиття. Очевидно, що загальний підхід залишиться незмінним: на всіх етапах поділ має здійснюватися точно між двома впорядкованими серіями, отриманими на попередньому етапі. В обох додаткових лініях буде однакова кількість серій, або їх число відрізнятиметься на одиницю.

Нехай знову в першу додаткову лінію ***b*** переходить ціла частина від поділу загального числа серій на ***2***.

Оскільки на всіх етапах впорядковані серії можуть бути різної довжини, то для реалізації фаз розділення і злиття доцільно мати додаткову структуру даних із початковими позиціями кожної серії. Це дозволить використати ту саму підпрограму-функцію ***zlyttya***.

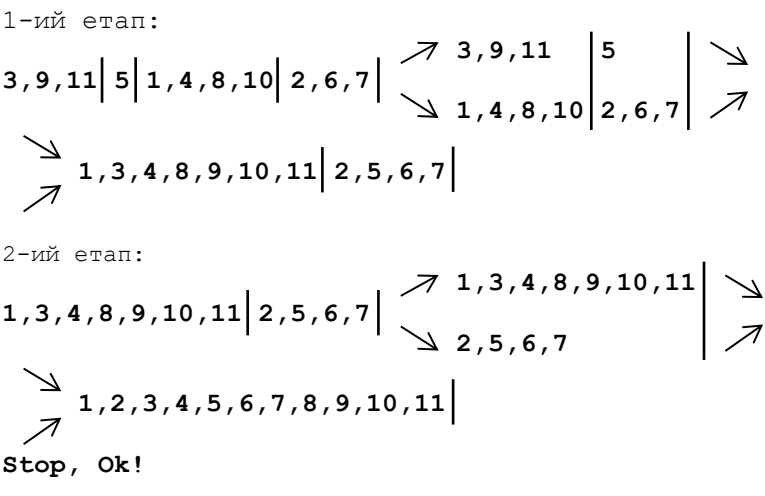
Якщо вхідна послідовність «хороша», тобто початкова кількість серій мала, то додаткова структура даних суттєво не вплине на загальні витрати пам'яті. Якщо ж послідовність «погана», тобто на початку багато коротких серій, то додаткова структура даних буде співрозмірна з основною за числом елементів. А це вже суттєва додаткова витрата пам'яті.

Одним із шляхів подолання такої проблеми є ще одна модифікація алгоритму злиття впорядкованих серій. При цьому не виконується в повному обсязі фаза розділення. Модифікація полягає в наступному:

- в одну з додаткових ліній, наприклад, ***b*** послідовно переписується перша по порядку серія з вхідної лінії ***a***;
- проводиться злиття серії з лінії ***b*** та наступної по порядку серії з вхідної лінії ***a***, результат злиття записується в другу додаткову лінію ***c***;
- проводиться злиття серії з лінії ***c*** та наступної по порядку серії з вхідної лінії ***a***, результат злиття знову записується у звільнену додаткову лінію ***b***;
- процес злиття накопичуваної серії з наступною по порядку серією з вхідної лінії ***a*** та почерговий перезапис результату в додаткові лінії ***b*** і ***c*** повторюється. Як наслідок, з вхідної лінії ***a*** послідовно вибираються впорядковані серії, а в додаткових лініях ***b*** і ***c*** почергово зростає результуюча послідовність. Цей процес завершиться при повному вичерпуванні вхідної лінії ***a***.

Такий спосіб сортування вже не потребує додаткових структур даних.

Приклад покрокового виконання.



Вхідна послідовність містить ***11*** елементів. Початкова кількість впорядкованих серій – ***4***. Виконано лише два етапи, хоча пряме злиття передбачало чотири етапи.

Програмна реалізація. Розглянемо варіант алгоритму із встановленням позицій початку кожної із серій. Цей спосіб дозволяє використати функцію ***zlyttya*** без жодних модифікацій. Нам прийдеться використати додатковий масив розміру ***N***. Програмна реалізація мовою C++ може бути такою:

```
    int a[N], b[N], c[N], d[N+1], k_ser, i, j, flag, k_ser_b, k_el_b, end_c;
    int ap, bp, bk, cp, ck;
// підрахунок кількості серій
    k_ser=0;
    i=0;
    while (i<N)
     {
      j=i+1;
      while (j<N && a[j-1]<=a[j]) j++;
      d[k_ser]=i; k_ser++;
      i=j;
     }
    d[k_ser]=N;                    // потрібно для обмеження останньої серії
// сортування
    while (k_ser>1)
     {
```

```
// розділення
        k_ser_b=k_ser/2;
        k_el_b=d[k_ser_b];
        flag=k_ser%2;              // k_ser_b: 0 – парна, 1 – непарна
        if (flag) end_c=d[k_ser-1]; else end_c=N;
        for (i=0; i<k_el_b; i++) b[i]=a[i];
        for ( ; i<end_c; i++) c[i-k_el_b]=a[i];
// злиття
        ap=0;
        for (i=0; i<k_ser_b; i++)
         {
           bp=d[i]; bk=d[i+1]-1;
           cp=d[k_ser_b+i]-k_el_b;
           ck=d[k_ser_b+i+1]-1-k_el_b;
           d[i]=ap;
           zlyttya(bp, bk, cp, ck, ap);
         }
        k_ser=k_ser_b;
        if (flag) d[k_ser++]=ap;
        d[k_ser]=N;
      }
```

Аналіз складності. Для даного алгоритму чітко розмежовуються за складністю кращі та гірші варіанти вхідних даних. У найкращому випадку прямо впорядкованої послідовності має місце єдина серія з N елементів. Порівняння виконуються лише при визначенні початкової кількості впорядкованих серій. А циклічні повторення фаз розділення і злиття взагалі не працюють. Тому:

$$C_{min} = N - 1 ; \quad M_{min} = 0 .$$

У найгіршому випадку обернено впорядкованої послідовності попередній аналіз дає N одноелементних серій. Тоді модифікований алгоритм працює так само, як і пряме злиття.

Програмну реалізацію варіанту модифікованого алгоритму злиття з послідовним накопиченням результуючої серії без попереднього аналізу початкових серій пропонується розглянути самостійно.

ЧАСТИНА 2
ДИНАМІЧНІ СТРУКТУРИ ДАНИХ – СПИСКИ

Розглянуті задачі та алгоритми обробки структур даних у переважній більшості стосувалися структур даних з прямим доступом до елементів за їх індексним номером – масивів. У більшості сучасних мов програмування характерною особливістю масивів є постійність їх розміру та розміщення в спеціальній частині статичної пам'яті (програмному стеку) протягом усього часу виконання програми. Проте на практиці потреба в постійній присутності всіх елементів не є обов'язковою. Досить часто окремі або навіть більшість елементів резервуються «на запас» для особливих випадків вхідних даних. В окремих мовах програмування, зокрема в C++, використовуються так звані динамічні або вільні масиви. Вони частково вирішують проблему оптимального використання пам'яті відповідно до потреб задачі. Однак вони передбачають динамічну зміну складу структури лише в її кінці і не дозволяють довільно вставляти чи вилучати елементи всередині структури даних.

Динамічні змінні зручно використовувати, коли структура даних, що обробляються, не є строго визначеною наперед. Використання статичних складених та комбінованих типів даних не завжди себе оправдує. Динамічні змінні дозволяють оптимізувати процес використання пам'яті по мірі потреби в програмі. Повністю оптимізувати процес створення, розміщення та видалення елементів відповідно до поточних потреб можна завдяки використанню одного з різновидів структур даних з послідовним доступом – *динамічних структур даних* або, як їх ще називають, *списків*. Обробка списків базується на використанні особливого типу даних – *вказівників*, який реалізований у багатьох мовах програмування.

Список – це динамічна структура даних з послідовним доступом, кожен елемент якої, окрім ключового інформаційного поля (або просто *ключа*), містить один або більше вказівників на інші елементи цієї ж структури. Таким чином, елементи списків фактично є комбінованими структурами з декількох полів даних різного типу. У мовах C/C++ таку функцію виконують структури (***struct***), а в мовах Pascal, Delphi – записи (***record***). Кількість елементів у списку може змінюватися в процесі виконання програми. Пам'ять для розміщення елементів виділяється і звільняється динамічно по мірі необхідності.

У залежності від кількості вказівних полів у структурі елемента та напрямі зв'язування елементів можна розглядати різні топології списків. Найчастіше використовують такі архітектури списків:

- лінійні однонапрямлені (однозв'язні);
- лінійні двонапрямлені (двов'язні);
- циклічні однонапрямлені та двонапрямлені;
- розгалужені багатозв'язні або дерева.

У *лінійних однонапрямлених списках* кожен елемент має єдиний вказівник на наступний елемент у структурі даних. Таким чином, доступ до довільного елемента є строго послідовним. Очевидно, що обов'язково має бути хоча б одна статична змінна – вказівник на перший елемент списку, позначимо її ***first*** (англ. *first – перший*). Якщо в програмі обмежитися лише вказівником на перший елемент, то список можна формувати виключно за принципом *стеку*. Окрім цієї обов'язкової змінної, часто використовують ще одну додаткову статичну змінну – вказівник на останній елемент списку, позначимо її ***last*** (англ. *last – останній*). Її використання дозволяє формувати списки як за принципом *стеку*, так і за принципом *черги*.

У *лінійних двонапрямлених списках* кожен елемент має два вказівники – на наступний і на попередній елементи у структурі даних. Отже, доступ до довільного елемента теж є строго послідовним, але можливий у двох взаємно протилежних напрямах. Для реалізації доступу в обох напрямах потрібно мати дві статичні змінні – вказівники на перший і на останній елементи списку (***first***, ***last***). Така організація дозволяє довільно формувати список: якщо для введення нових елементів використовується вказівник на перший елемент, то це дає список-стек при доступі через змінну ***first*** або список-чергу при доступі через змінну ***last***; якщо для введення нових елементів використовується вказівник на останній елемент, то це дає список-чергу при доступі через змінну ***first*** або список-стек при доступі через змінну ***last***.

У *розгалужених багатозв'язних списках* кожен елемент має два або більше вказівників на інші елементи структури даних, але без дотримання лінійного порядку доступу до елементів. Часто такі списки мають деревоподібну структуру: кожен елемент попереднього рівня має зв'язок із двома чи більше елементами наступного рівня. Доступ до довільного елемента можливий завдяки переходам від найвищого рівня на більш нижчі рівні. Для доступу до такої деревоподібної структури даних потрібно мати одну статичну змінну – вказівник на вершинний елемент ***top*** (англ. *top – вершина*). Оскільки такі деревоподібні списки мають виражену рекурсивну структуру, то для їх обробки потрібно використовувати рекурсивні підпрограми.

Основними операціями при роботі зі списками є:

- створення списку;
- перегляд списку;
- пошук елемента в списку;
- вставка нового елемента в список;
- видалення елемента зі списку;
- впорядкування списку за ознакою (сортування);
- розділення списку на підсписки;
- з'єднання (конкатенація) списків.

РОЗДІЛ 3
ЛІНІЙНІ СПИСКИ

3.1. ОДНОНАПРЯМЛЕНІ СПИСКИ

3.1.1. Оголошення типу елемента списку

Усі елементи лінійного однонапрямленого списку, окрім основної за умовами задачі інформації, мають вказівник на черговий елемент цього ж списку. Така організація списку означає, що кожен його елемент – це комбінована структура, яка охоплює два поля даних: інформаційне поле ***info*** деякого базового типу (скалярного чи складеного) і вказівне поле ***next***, що фактично є адресою у пам'яті наступного елемента. Поле вказівника ***next*** повинно мати такий самий тип, що й сам елемент списку. Для ідентифікації елемента (або для доступу до нього) потрібно мати деякий вказівник ***P*** цього ж типу.

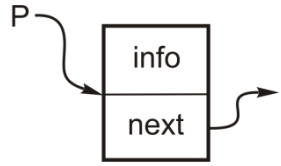


Рис. 1

Таким чином, має місце взаємність визначення як цілого елемента списку, так і його окремого вказівного поля. Така взаємна рекурсивність у визначенні типу допускається в мовах C/C++, Pascal, Delphi та інших лише для вказівних типів.

Зрозуміло, що вказівник на перший елемент списку ***first*** повинен мати такий самий тип, як і решта елементів списку. Природно, що вказівне поле ***next*** останнього елемента списку повинно вказувати ***«в нікуди»*** – на особливу область динамічної пам'яті ***NULL***, що не містить ніяких динамічних змінних.

Враховуючи все вище сказане, лінійний однонапрямлений список схематично можна зобразити так:

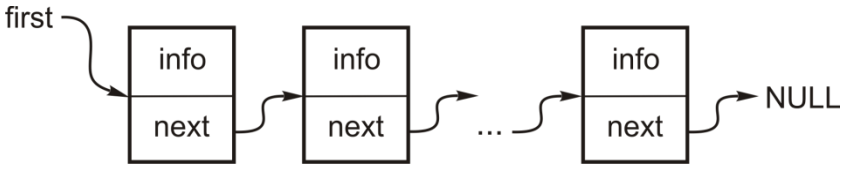


Рис. 2

Не зменшуючи загальності, приймаємо в якості базового типу інформаційного поля даних тип цілих чисел ***int***. Програмна реалізація мовою С++ оголошення спискового типу та деяких необхідних статичних змінних може бути такою:

```
struct List_1
 {
   int info;
   List_1 * next;
 };
List_1 * first, * last, * P, * Q;
```

Тут ***List_1*** – ім'я структури і одночасно базовий тип для вказівного типу елемента списку; ***first, last, P, Q*** – вказівники відповідно на перший, останній і деякі поточні елементи для виконання операцій над списком.

3.1.2. Основні операції по обробці однонапрямлених списків

Стеки і черги

Стеком є така структура даних з послідовним доступом, у якій уведення нових елементів і виштовхування існуючих відбувається виключно з одного кінця, що називається *вершиною* або *головою*. Обробка стеку характеризується правилом «*першим прийшов – останнім вийшов, останнім прийшов – першим вийшов*». Аналогом стеку можна вважати трубку з одним запаяним кінцем (як у пробірці), у яку поміщено кульки відповідного розміру з деякою інформацією. Тоді вершиною стеку буде відкритий кінець трубки. Інформаційний елемент, що першим потрапив у таку структуру даних, опиниться на *дні стеку* (за аналогією з дном пробірки), і доступ до нього буде в останню чергу. Останній уведений елемент виявиться відразу на вершині стеку, і доступ до нього буде першочерговим.

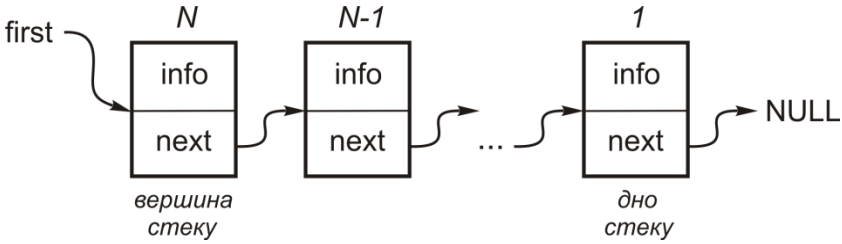


Рис. 3

На перший погляд такий спосіб організації структур даних видається неефективним. Проте для доступу до будь-якого елемента потрібно мати мінімум статичних змінних – лише вказівник ***first*** на вершину стеку, тобто *на останній уведений елемент, що фактично є першим у структурі даних*. При програмній реалізації обробки списків для доступу до внутрішніх елементів насправді не потрібно «виймати» попередні елементи. Замість цього виконується перехід на наступний елемент через вказівне поле ***next***, починаючи від фіксованого вказівника ***first***.

Черги в програмуванні, як і в реальному житті, використовуються тоді, коли потрібно виконувати дії за командами послідовно в порядку їх надходження. *Черга* – це така структура даних з послідовним доступом, у якій уведення нових елементів і виштовхування існуючих відбувається з різних кінців, що називаються відповідно *кінець черги* та *початок черги*. Обробка черги характеризується правилом «*першим прийшов – першим вийшов, останнім прийшов – останнім вийшов*». Для ілюстрації черги можна розглянути трубку з двома відкритими кінцями, у яку поміщені інформаційні кульки. Уведення нових елементів проводиться в кінець черги, а доступ до вже існуючих у структурі даних здійснюється на її початку. Інформаційний елемент, що першим потрапив у чергу, відразу буде на її виході, і доступ до нього буде першочерговим. Останній уведений елемент виявиться в самому кінці черги, і доступ до нього буде в останню чергу.

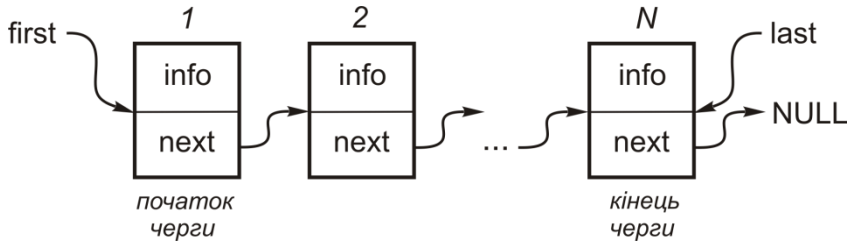


Рис. 4

Як і у випадку стеку, доступ до елементів черги реалізується через фіксований вказівник ***first*** на початок черги, тобто на перший елемент у структурі даних, що також першим був уведений. Однак на відміну від стеку для заповнення черги потрібно мати ще один фіксований вказівник ***last*** на кінець черги, тобто на останній елемент у структурі даних, що також останнім був уведений. Для доступу до довільного елемента черги подібно до стеку виконується перехід на наступний елемент через вказівне поле ***next***, починаючи від фіксованого вказівника ***first***.

Створення списку

У випадку однозв'язних списків процес створення (формування) за принципом *стеку* відрізняється від аналогічної операції за принципом *черги*.

Розглянемо спочатку **ініціалізацію та заповнення стеку**. Весь процес можна розділити на три етапи, з яких два останніх потім об'єднаємо в один:

1) створення порожнього списку:

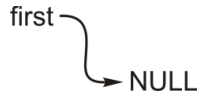


Рис.5

```
first=NULL;          // вказівник на перший вказує «в нікуди»;
```

2) уведення першого нового елемента:

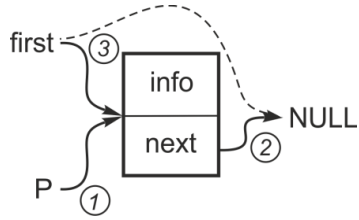


Рис. 6

```
P=new List_1;        // 1) створюється новий елемент;
cin >> P->info;      // ввід інформаційного поля нового елемента;
P->next=first;       // 2) вказівник з нового елемента вказує туди,
                     // куди раніше вказував first;
first=P;             // 3) first переадресовується на новий елемент.
```

Як видно з коментарів, новий елемент списку вказуватиме на місце, на яке раніше вказував ***first*** (початково це буде ***NULL***, а пізніше це буде раніше введений елемент). І лише потім ***first*** переадресовується на новостворений елемент. **Порядок виконання операцій із вказівниками є важливим!** Тут і в подальшому з метою полегшення розуміння й зручності демонстрації на схемах принциповий порядок дій по переадресації та зв'язуванню вказівників пронумеровано числами в колах. При цьому **старі зв'язки** між елементами позначено **штриховими стрілками**, а їх **нові реалізації** після переадресації вказівників – **суцільними**.

3) уведення решти нових елементів. Очевидно, що введення решти елементів нічим не відрізняється від уведення першого елемента. Тому другий етап можна циклічно повторювати задану кількість разів ***N***:

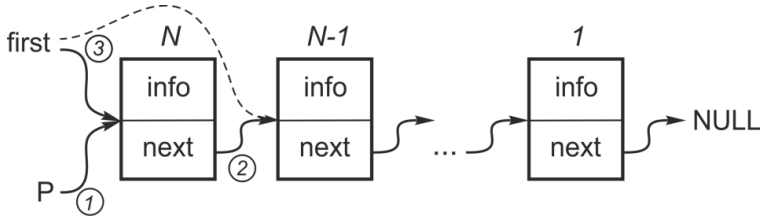


Рис. 7

```
for (int i=0; i<N; i++)
 {
   P=new List_1;
   cin >> P->info;
   P->next=first;
   first=P;
 }
```

Тепер розглянемо **<u>ініціалізацію та заповнення черги</u>**. Як і у випадку стеку, цей процес доцільно розділити на три етапи, але об'єднувати два останніх вже не можна. Це пов'язано з тим, що при формуванні черги введення нового першого елемента потребує використання обох фіксованих вказівників ***first*** і ***last***, а введення наступних нових елементів передбачає використання лише одного вказівника ***last***. Послідовність дій може бути наступною:

1) створення порожнього списку:

Рис. 8

```
first=NULL;          // вказівник на перший вказує «в нікуди»;
last=NULL;           // вказівник на останній вказує «в нікуди»;
```

2) уведення першого нового елемента:

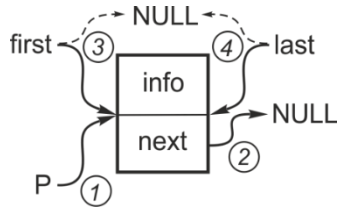


Рис. 9

```
P=new List_1;          // 1) створюється новий елемент;
cin >> P->info;        // увід інформаційного поля нового елемента;
P->next=NULL;          // 2) вказівник із нового елемента вказує
                       // «в нікуди»;
first=P;               // 3) вказівник first вказує на новий елемент;
last=P;                // 4) вказівник last вказує на новий елемент.
```

Новий елемент списку вказуватиме «в нікуди» (***NULL***). Оскільки цей елемент є і першим, і останнім одночасно, то ***first*** і ***last*** адресуються на новоутворений елемент;

3) уведення решти нових елементів. Цей етап не можна об'єднувати з попереднім. Вказівне поле кожного наступного нового елемента має вказувати «в нікуди» (***NULL***), а раніше введений елемент повинен зв'язуватися з новостворюваним. І лише після цього ***last*** переадресовується на новоутворений елемент. Тут теж порядок виконання операцій із вказівниками є важливим. Цей етап передбачає циклічне повторення вказаної послідовності дій число разів, що на одиницю менше від необхідної кількості ***N***:

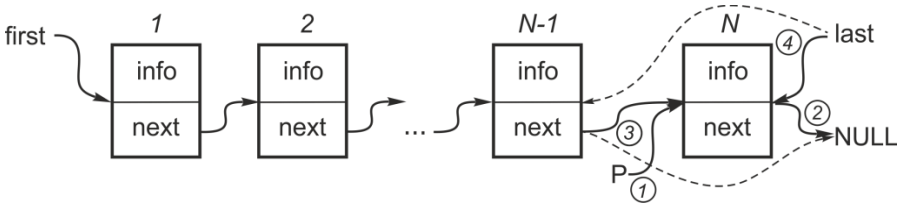


Рис. 10

```
for (int i=1; i<N; i++)
 {
  P=new List_1;
  cin >> P->info;
  P->next=NULL;
  last->next=P;          // раніше введений елемент вказує на новий;
  last=P;                // вказівник last тепер вказує на новий елемент.
 }
```

Перегляд списку

Під *переглядом* розуміється повний перебір елементів списку з можливим виведенням значень їх інформаційних полів. Очевидно, що рух по списку виконується від вказівника ***first*** за рахунок переходу на наступний елемент через вказівне поле ***next***. Оскільки ***first*** є фіксованим вказівником на початок списку, і його в жодному разі не можна змінювати, то для адресації довільного елемента списку слід використовувати додаткову статичну змінну такого ж вказівного типу. Переміщення по списку завершується при досягненні його кінця, тобто при набутті вказівником на поточний елемент значення ***NULL***.

Програмна реалізація операції перегляду списку може бути такою:

```
P=first;                    // початок списку;
while (P)                   // рух, поки не кінець списку;
 {
  cout << P->info << endl;  // вивід значення інформаційного поля;
  P=P->next;                // перехід на наступний елемент.
 }
```

Пошук елемента в списку

Пошук може виконуватися за різними критеріями:

- встановлення значення інформаційного поля елемента за його порядковим номером;
- встановлення позиції першого входження елемента із заданим значенням інформаційного поля;
- встановлення всіх входжень елементів із заданим ключем.

Подібно до попередньої операції перегляду для кожного з варіантів пошуку рух по списку виконується від вказівника ***first*** за рахунок переходу на наступний елемент через вказівне поле ***next***. Для цього так само використовуватиметься додаткова змінна-вказівник. Такий спосіб переміщення буде в основі всіх операцій по обробці списків. Проте умови завершення переходів по списку можуть бути різними.

У багатьох випадках реалізацій обробки списків потрібно використовувати додатковий лічильник поточної позиції у списку. Не зменшуючи загальності, вважатимемо, що нумерація елементів у списку починається з ***1***. Це відрізняється від індексації елементів масивів у мовах C/C++, яка традиційно починається з ***0***. Такий підхід цілком умовний і може бути змінений при потребі.

Пошук елемента за його порядковим номером у списку.

Очевидно, що умови зупинки руху будуть складеними і їх дві:

- ще не досягнутий кінець списку, а лічильник поточної позиції співпав із заданим порядковим номером – це позитивний результат;
- досягнутий кінець списку – це негативний результат.

Другий варіант відповідає випадку, коли задана цільова позиція перевищує реальну кількість елементів у списку. Для такої ситуації потрібно передбачити коректне завершення програми. Це пов'язано з недопустимістю операції розіменування вказівника, що вказує в ***NULL***.

Замість приросту додаткового лічильника поточної позиції можна зменшувати значення цільової позиції, поки воно не досягне одиниці. Програмна реалізація даної операції може мати вигляд:

```
int Pos; cin >> Pos;          // Pos – цільова позиція елемента;
P=first;                      // початок списку;
while (P && Pos>1)            // рух, поки не кінець списку і не цільова позиція.
  { P=P->next; Pos--; }
if (P && Pos==1)
  cout << "значення елемента = " << P->info << endl;
else
  cout << "список порожній чи некоректна позиція елемента" << endl;
```

Варто зазначити, що така реалізація пошуку матиме коректне завершення циклу не лише, коли заданий порядковий номер елемента перевищує їх реальну кількість у списку, а й коли цей номер менше одиниці (від'ємний або нуль).

Пошук першого входження елемента із заданим значенням інформаційного поля.

Знову умови зупинки руху будуть складеними і їх дві:

- ще не досягнутий кінець списку, а значення інформаційного поля поточного елемента співпало із заданим – це позитивний результат;
- досягнутий кінець списку – це негативний результат.

Програмна реалізація даної операції може мати вигляд:

```
int X; cin >> X;              // X – цільове значення елемента;
int i=1;                      // ініціалізація поточної позиції;
P=first;                      // початок списку;
while (P && P->info!=X)       // рух, поки не кінець списку і не цільове значення.
  { P=P->next; i++; }
if (P)
  cout << "елемент " << X << " є в позиції " << i << endl;
else
  cout << "список порожній чи цільового елемента немає" << endl;
```

Пошук усіх входжень елементів із заданим ключем.

Загальна схема реалізації подібна до розглянутої вище операції перегляду списку. Для розділення успішного і невдалого пошуку варто використати додаткову змінну – лічильник кількості входжень:

```
int X; cin >> X;                // X – цільове значення елемента;
int k=0;                        // лічильник кількості входжень;
int i=1;                        // ініціалізація поточної позиції;
P=first;                        // початок списку;
while (P)                       // рух, поки не кінець списку;
 {
  if (P->info==X)               // співпадання із цільовим значенням;
    { cout << "елемент " << X << " є в позиції " << i << endl; k++; }
  P=P->next; i++;               // перехід на наступний елемент.
 }
if (k)
 cout << "усього встановлено " << k << " входжень" << endl;
else
 cout << "список порожній чи цільового елемента немає" << endl;
```

Вставка нового елемента в список

Подібно до операції пошуку доцільно розглядати різні варіанти вставки нового елемента.

Вставка нового елемента в задану позицію списку.

Для визначеності домовимося про наступне: якщо позиція вставки менша одиниці (від'ємна або нуль), то виконуватимемо вставку на початок списку; якщо позиція вставки перевищує реальну кількість елементів у списку, то виконуватимемо вставку в кінець списку. Прийняття такого рішення не є відходом від принципу строгої визначеності в програмуванні. Окрім іншого, це дозволить коректно обробляти і порожні списки. Звичайно, такі домовленості є умовними, і при потребі у подібних випадках завжди можна відмовитися від виконання цієї операції.

Для реалізації вставки нового елемента потрібно передбачити різні можливі ситуації:

- вставка в порожній список (це вимагає одночасного опрацювання вказівників ***first*** і ***last***);
- вставка на початок списку (це вимагає опрацювання вказівника ***first***);
- вставка в кінець списку (це вимагає опрацювання вказівника ***last***);
- вставка всередині списку.

У будь-якому випадку при вставці потрібно мати доступ до елемента, що передує заданій позиції. При цьому необхідно спочатку «прив'язати»

вказівне поле нового елемента до потрібного елемента списку, і лише після цього «розірвати» зв'язок між існуючими елементами списку в позиції вставки та адресувати звільнені вказівники на новий елемент. Схематично це можна зобразити так:

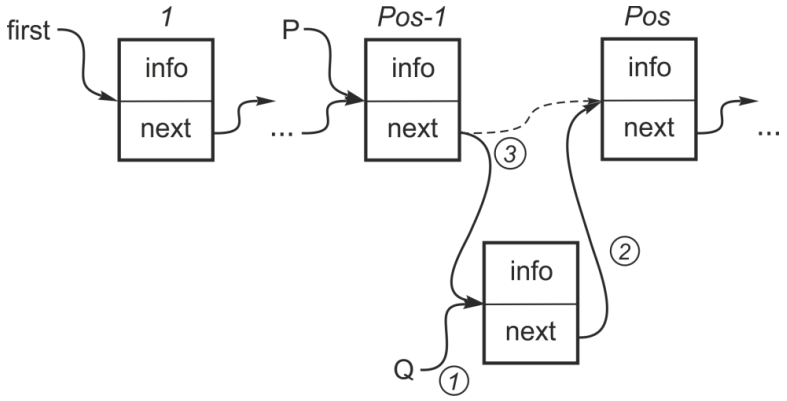


Рис. 11

Програмна реалізація даної операції може мати вигляд:

```
Q=new List_1;   cin >> Q->info;          // створення нового елемента;
int Pos;   cin >> Pos;                   // Pos – цільова позиція вставки;
P=first;                                 // початок списку;
if (!P)                                  // вставка в порожній список;
 { Q->next=NULL; first=last=Q; }
else                                     // вставка в непорожній список;
 if (Pos<=1)                             // вставка на початок списку;
  { Q->next=first; first=Q; }
 else                                    // вставка не на початок списку;
  {
   while (P->next && Pos>2)              // рух до цільової позиції;
    { P=P->next; Pos--; }
   if (P->next)                          // вставка в цільову позицію;
    { Q->next=P->next; P->next=Q; }
   else                                  // вставка в кінець списку.
    { Q->next=NULL; last->next=Q; last=Q; }
  }
```

Читачам пропонується самостійно модифікувати програмну реалізацію з метою відмови від вставки елемента в список, якщо задана позиція не відповідає реальній кількості елементів. При цьому слід врахувати, що вставка в позицію ***N+1*** списку, який містить ***N*** елементів, є абсолютно коректною.

Зокрема, частковим випадком такої задачі є вставка в першу позицію порожнього списку.

Вставка нового елемента перед елементом із заданим значенням інформаційного поля.

З метою коректного опрацювання особливих ситуацій, як і в попередньому випадку, не обійтися без домовленостей: якщо елементів із заданим значенням у списку є декілька, то виконується вставка лише перед першим із них; якщо в списку немає жодного елемента із заданим значенням, у тому числі й для порожнього списку, то вставка не виконується взагалі. Такі домовленості є природними і суттєвими, адже відсутність ключового елемента в списку робить абсолютно невизначеною позицію вставки.

Пропонується реалізація вставки нового елемента для таких випадків:

- відмова від вставки в порожній список;
- вставка перед першим елементом, тобто на початок списку (це вимагає опрацювання вказівника ***first***);
- вставка всередині списку перед довільним елементом, у тому числі й перед останнім (потрібно мати доступ до елемента, що передує цільовому елементу);
- відмова від вставки у випадку відсутності заданого елемента в списку.

Послідовність дій по зв'язуванню елементів вказівниками аналогічна попередньому випадку.

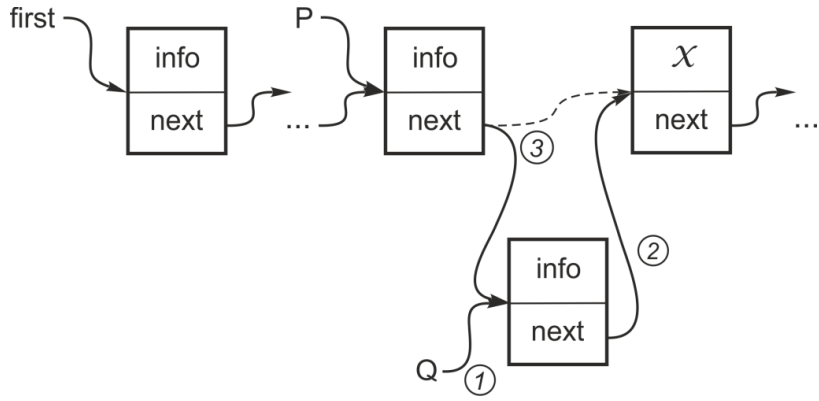


Рис. 12

Програмна реалізація даної операції може мати вигляд:

```
Q=new List_1;   cin >> Q->info;        // створення нового елемента;
int X;   cin >> X;                     // X – цільове значення елемента;
P=first;                               // початок списку;
```

```
if (P)                                    // список непорожній;
  if (P->info==X)                         // вставка на початок списку;
    { Q->next=first; first=Q; }
  else
    {
      while (P->next && P->next->info!=X)  // рух до цільового елемента;
        P=P->next;
      if (P->next)                        // вставка всередині списку.
        { Q->next=P->next; P->next=Q; }
      else cout << "цільового елемента в списку немає" << endl;
    }
else cout << "список порожній – вставка неможлива" << endl;
```

Вставка нового елемента після елемента із заданим значенням інформаційного поля.

Залишимо в силі ті самі домовленості, що й у попередньому випадку. Звичайно, що тут розглядається вставка нового елемента саме після першого серед елементів із цільовим значенням. Можливі такі варіанти реалізації:

- відмова від вставки в порожній список;
- вставка всередині списку після довільного елемента, у тому числі й після першого (потрібно мати доступ до елемента з цільовим значенням);
- вставка після останнього елемента, тобто в кінець списку (це вимагає опрацювання вказівника ***last***);
- відмова від вставки у випадку відсутності заданого елемента в списку.

Послідовність дій по зв'язуванню елементів вказівниками аналогічна попередньому випадку.

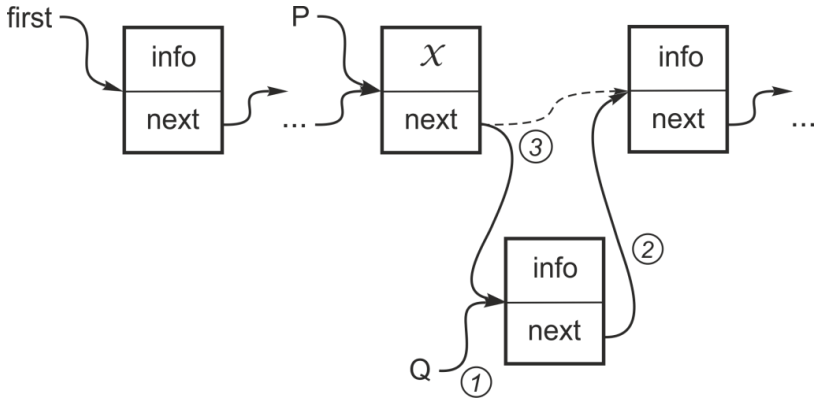


Рис. 13

Програмна реалізація даної операції може мати вигляд:

```
Q=new List_1;   cin >> Q->info;          // створення нового елемента;
int X;   cin >> X;                       // X – цільове значення елемента;
P=first;                                 // початок списку;
if (P)                                   // список непорожній;
 {
  while (P && P->info!=X)                // рух до цільового елемента;
     P=P->next;
  if (P)                                 // вставка всередині списку;
   { Q->next=P->next; P->next=Q; }
  else cout << "цільового елемента в списку немає" << endl;
  if (P==last) last=Q;                   // вставка в кінець списку.
 }
else cout << "список порожній – вставка неможлива" << endl;
```

Видалення елемента зі списку

Видалення є протилежною операцією до вставки. Для неї теж потрібно розглядати різні варіанти вилучення елемента.

Видалення елемента в заданій позиції списку.

Природно, що операція видалення буде успішною, якщо позиція вилучення відповідає реальній кількості елементів у списку. До порожнього списку така операція взагалі не застосовується. Для коректної реалізації поставленого завдання слід розглянути такі можливі ситуації:

- видалення першого і одночасно останнього елемента, тобто єдиного в списку (це вимагає одночасного опрацювання вказівників ***first*** і ***last***);
- видалення першого елемента, що не єдиний у списку (це вимагає опрацювання вказівника ***first***);
- видалення останнього елемента, що не єдиний у списку (це вимагає опрацювання вказівника ***last***);
- видалення елемента всередині списку (потрібно мати доступ до елемента, що передує заданій позиції).

При видаленні потрібно виконати у визначеному порядку наступні дії (див. рис. 14):

1) вказівником з елемента, що передує цільовій позиції, «обминути» елемент, що вилучається, та адресувати цей вказівник на наступний після елемента, що вилучається. Це приведе до «випадання» цільового елемента зі списку (позначено цифрою ***1*** на схемі);
2) цільовий елемент видаляється з динамічної пам’яті за допомогою додаткової змінної-вказівника ***Q*** (позначено цифрою ***2*** на схемі).

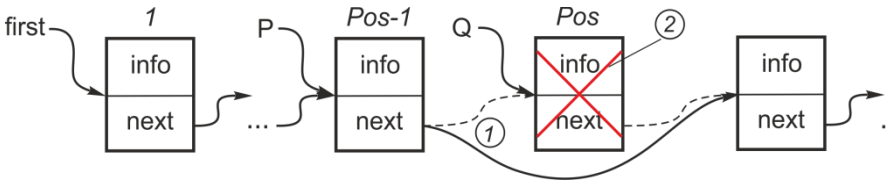


Рис. 14

Програмна реалізація даної операції може мати вигляд:

```
int Pos;   cin >> Pos;                    // Pos – цільова позиція видалення;
P=first;                                  // початок списку;
if (P && Pos>0)                           // список непорожній;
  if (Pos==1)                             // видалення першого елемента;
    if (first==last)                      // перший – єдиний у списку;
      { first=last=NULL; delete P; }
    else                                  // перший – не єдиний у списку;
      { first=first->next; delete P; }
  else                                    // видалення не першого елемента;
    {
      while (P->next && Pos>2)            // рух до цільової позиції;
        { P=P->next; Pos--; }
      if (P->next)                        // цільова позиція досягнута;
        if (P->next==last)                // видалення останнього елемента;
          { P->next=NULL; delete last; last=P; }
        else                              // видалення всередині списку.
          { Q=P->next; P->next=Q->next; delete Q; }
      else cout << "некоректна позиція видалення" << endl;
    }
else cout<<"список порожній чи некоректна позиція видалення"<<endl;
```

Видалення елемента із заданим значенням інформаційного поля.

Загальна схема виконання дій подібна до вище розглянутої операції. Тому основні випадки видалення елемента зі списку за його інформаційним полем будуть такими самими.

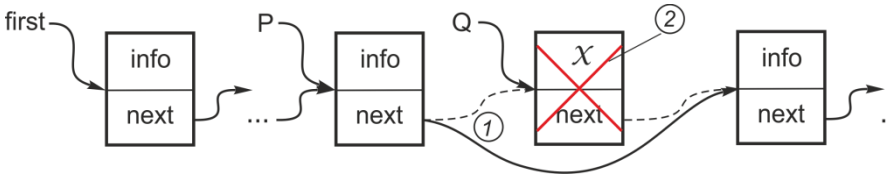


Рис. 15

Програмна реалізація даної операції може мати вигляд:

```
int X;   cin >> X;                          // X – цільове значення елемента;
P=first;                                    // початок списку;
if (P)                                      // список непорожній;
  if (P->info==X)                           // видалення першого елемента;
    if (first==last)                        // перший – єдиний у списку;
      { first=last=NULL; delete P; }
    else                                    // перший – не єдиний у списку;
      { first=first->next; delete P; }
  else                                      // видалення не першого елемента;
    {
      while (P->next && P->next->info!=X)   // рух до цільового елемента;
        P=P->next;
      if (P->next)                          // цільовий елемент досягнутий;
        if (P->next==last)                  // видалення останнього елемента;
          { P->next=NULL; delete last; last=P; }
        else                                // видалення всередині списку.
          { Q=P->next; P->next=Q->next; delete Q; }
      else cout << "цільового елемента в списку немає" << endl;
    }
else cout << "список порожній – видалення неможливе" << endl;
```

3.2. ДВОНАПРЯМЛЕНІ СПИСКИ

3.2.1. Оголошення типу елемента списку

Розглянута вище структура даних у вигляді однонапрямленого списку попри всі свої позитивні риси має суттєвий недолік. Це навіть закладено в саму назву списку – *однонапрямлений*. Тобто в таких структурах даних можливий послідовний доступ до елементів лише в одному напрямі від вершини в глибину. Уявіть собі ситуацію, коли поточним місцем локалізації у достатньо довгому списку є останній елемент і потрібно швидко переміститися на сусідній попередній елемент. Без руху від самого початку майже через увесь список не обійтися. Дуже часто в прикладних застосуваннях потрібно мати можливість швидко переміщуватися по структурі даних у різних напрямах. Така проблема може бути вирішена шляхом незначного ускладнення структури самого елемента списку – введенням ще одного вказівного поля (позначимо його ***prev***), яке адресуватиме попередній елемент. Побудований таким чином список вже буде *двонапрямленим* і по ньому можливий рух у двох напрямах.

Хоча у випадку двонапрямлених списків теж використовуються фіксовані вказівники ***first*** на перший елемент і ***last*** на останній елемент, проте насправді таке визначення понять *початок* і *кінець* є умовними. Адже при русі з переходом через вказівне поле ***next*** умовний початок списку адресуватиметься вказівником ***first***, а його кінець – вказівником ***last***. Якщо ж для переходів використовувати вказівне поле ***prev***, то на умовний початок списку вказуватиме ***last,*** а вказівник ***first*** адресуватиме його кінець. По цій самій причині для лінійних двонапрямлених списків не робиться різниця між стеком і чергою: стек, сформований відносно фіксованого вказівника ***first*** із переходами по зв'язку ***next***, – це черга, сформована відносно фіксованого вказівника з переходами по зв'язку ***prev***, і навпаки. Можна вважати, що такі списки мають дві вершини (голови), які при роботі абсолютно рівноцінні.

Не зменшуючи загальності, домовимося про наступне: елемент, на який вказує ***first***, – *початок* списку, а елемент, на який вказує ***last,*** – *кінець* списку. Очевидно, що і вказівне поле ***prev*** із першого елемента на попередній, і вказівне поле ***next*** з останнього елемента на наступний вказуватимуть «в нікуди», тобто матимуть значення ***NULL***.

Знову приймаємо в якості базового типу інформаційного поля даних тип цілих чисел ***int***. Програмна реалізація мовою C++ оголошення спискового типу та деяких необхідних статичних змінних може бути такою:

```
struct List_2
 {
  int info;
  List_2 * next, * prev;
 };
List_2 * first, * last, * P, * Q;
```

3.2.2. Основні операції по обробці двонапрямлених списків

Створення списку

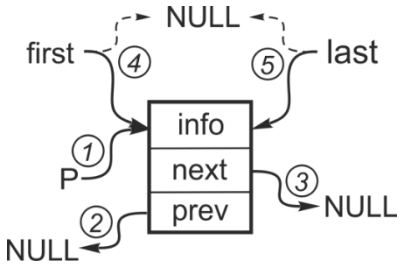
Хоча, як вже було домовлено, формування списку умовно проводиться за принципом стеку відносно фіксованого вказівника ***first***, проте цей процес подібний до формування черги. Адже однаково потрібно використовувати фіксований вказівник ***last*** на останній елемент списку, який насправді буде введений першим. Весь процес складається з трьох етапів.

1) створення порожнього списку:

first=last=NULL;

Рис. 16

2) уведення першого нового елемента:



P=new List_2;
cin >> P->info;
P->next=P->prev=NULL;
first=last=P;

Рис. 17

3) уведення решти нових елементів:

```
for (int i=1; i<N; i++)
 {
  P=new List_2;
  cin >> P->info;
  P->prev=first->prev;
  P->next=first;
  first->prev=P;
  first=P;
 }
```

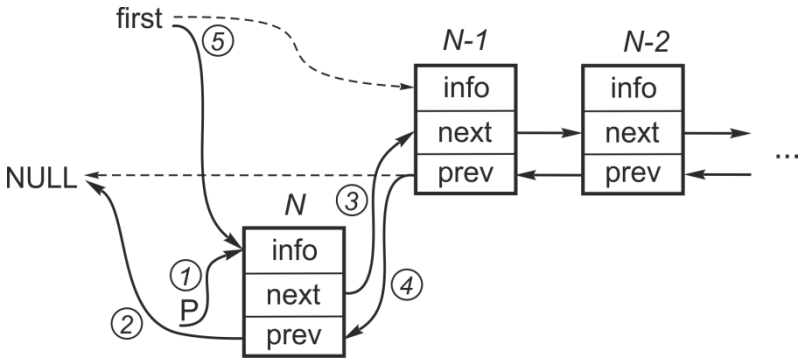


Рис. 18

Пошук елемента в списку

Реалізація пошуку елемента за різними критеріями співпадає з аналогічними операціями для однонапрямленого списку. Звичайно, це стосується випадку, коли рух до потрібного елемента здійснюється переходами по зв'язках ***next*** від початку списку. Якщо ж для доступу до елемента використовуються переходи по зв'язках ***prev*** від кінця списку, то розглянуті раніше фрагменти програмного коду неважко переробити, взаємно замінивши ідентифікатори ***first*** на ***last*** і ***last*** на ***first***, а також ***next*** на ***prev*** і ***prev*** на ***next***.

Вставка нового елемента в список

Розглянемо такі самі варіанти вставки нового елемента, що й для однонапрямлених списків.

Вставка нового елемента в задану позицію списку.

Приймаємо аналогічні домовленості: якщо позиція вставки менше одиниці, то виконується вставка на початок списку; якщо позиція вставки перевищує реальну кількість елементів у списку, то виконується вставка в кінець списку. Слід так само передбачити коректну програмну реалізацію для кожного з можливих випадків:

- вставка в порожній список;
- вставка на початок списку;
- вставка в кінець списку;
- вставка всередині списку.

На відміну від однонапрямлених для двонапрямлених списків не потрібно мати доступ до елемента, що передує заданій позиції, а слід мати

вказівник безпосередньо на позицію вставки. Коректне зв'язування вказівних полів нового елемента з наступним та попереднім елементами реалізується відповідною комбінацією операцій над вказівниками ***next*** і ***prev***. Після цього можна «розірвати» зв'язки між існуючими елементами списку в позиції вставки та адресувати звільнені вказівники на новий елемент. Схематично це можна зобразити так:

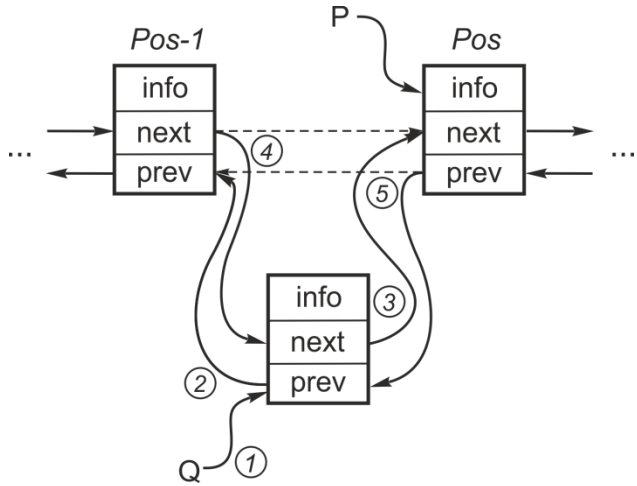


Рис. 19

Програмна реалізація даної операції може мати вигляд:

```
Q=new List_2;   cin >> Q->info;              // створення нового елемента;
int Pos;   cin >> Pos;                       // Pos – цільова позиція вставки;
P=first;                                     // початок списку;
if (!P)                                      // вставка в порожній список;
 { Q->next=Q->prev=NULL; first=last=Q; }
else                                         // вставка в непорожній список;
 if (Pos<=1)                                 // вставка на початок списку;
  { Q->prev=NULL; Q->next=first; first=Q; }
 else                                        // вставка не на початок списку;
  {
   while (P && Pos>1)                        // рух до цільової позиції вставки;
    { P=P->next; Pos--; }
   if (P)                                    // вставка всередині списку;
    { Q->prev=P->prev; Q->next=P; P->prev->next=Q; P->prev=Q; }
   else                                      // вставка в кінець списку.
    { Q->prev=last; Q->next=NULL; last->next=Q; last=Q; }
  }
```

Вставка нового елемента перед елементом із заданим значенням інформаційного поля.

Домовленості, що були прийняті для однонапрямлених списків, залишаємо в силі. Пропонується реалізація вставки нового елемента для таких випадків:

- відмова від вставки в порожній список;
- вставка перед першим елементом, тобто на початок списку;
- вставка всередині списку перед довільним елементом, у тому числі й перед останнім;
- відмова від вставки у випадку відсутності заданого елемента в списку.

Послідовність дій по зв'язуванню елементів вказівниками аналогічна попередньому випадку. Програмна реалізація даної операції може мати вигляд:

```
Q=new List_2;   cin >> Q->info;          // створення нового елемента;
int X;   cin >> X;                       // X – цільове значення елемента;
P=first;                                 // початок списку;
if (P)                                   // список непорожній;
 if (P->info==X)                         // вставка на початок списку;
  { Q->prev=NULL; Q->next=first; first=Q; }
 else                                    // вставка не на початок списку;
  {
   while (P && P->info!=X)               // пошук цільового елемента;
     P=P->next;
   if (P)                                // вставка всередині списку.
     { Q->prev=P->prev; Q->next=P; P->prev->next=Q; P->prev=Q; }
   else cout << "цільового елемента в списку немає" << endl;
  }
else cout << "список порожній – вставка неможлива" << endl;
```

Вставка нового елемента після елемента із заданим значенням інформаційного поля.

Знову залишаємо в силі ті самі домовленості, що були прийняті для однонапрямлених списків. Пропонується реалізація вставки нового елемента для таких випадків:

- відмова від вставки в порожній список;
- вставка всередині списку після довільного елемента, у тому числі й після першого;
- вставка після останнього елемента, тобто в кінець списку;
- відмова від вставки у випадку відсутності заданого елемента в списку.

Послідовність дій по зв'язуванню елементів вказівниками аналогічна попередньому випадку. Програмна реалізація даної операції може мати вигляд:

```
Q=new List_2;   cin >> Q->info;          // створення нового елемента;
```

```
int X;   cin >> X;                    // X – цільове значення елемента;
P=first;                              // початок списку;
if (P)                                // список непорожній;
 {
  while (P && P->info!=X)             // пошук цільового елемента;
   P=P->next;
  if (P)                              // цільовий елемент знайдено;
   if (P==last)                       // вставка в кінець списку;
    { Q->prev=last; Q->next=NULL; last=Q; }
   else                               // вставка всередині списку.
    { Q->prev=P; Q->next=P->next; P->next->prev=Q; P->next=Q; }
  else cout << "цільового елемента в списку немає" << endl;
 }
else cout << "список порожній, вставка неможлива" << endl;
```

Видалення елемента зі списку

Видалення є протилежною операцією до вставки. Для неї теж потрібно розглядати різні варіанти вилучення елемента.

Видалення елемента в заданій позиції списку.

Для коректної реалізації поставленого завдання слід окремо розглянути такі самі можливі випадки, що й для однонапрямлених списків:

- відмова від операції над порожнім списком;
- видалення першого і одночасно останнього елемента, тобто єдиного в списку;
- видалення першого елемента, що не єдиний у списку;
- видалення останнього елемента, що не єдиний у списку;
- видалення елемента всередині списку.

При видаленні необхідно спочатку виконати такі дії: вказівником ***next*** з елемента, що передує цільовій позиції, «обминути» елемент, що вилучається, й адресувати цей вказівник на наступний після елемента, що вилучається; вказівником ***prev*** з елемента, що слідує після цільової позиції, «обминути» елемент, що вилучається, й адресувати цей вказівник на попередній перед елементом, що вилучається. Це приведе до «випадання» цільового елемента зі списку. Після цього цей елемент можна видалити з динамічної пам'яті, користуючись додатковою змінною-вказівником. Схематично це можна зобразити так:

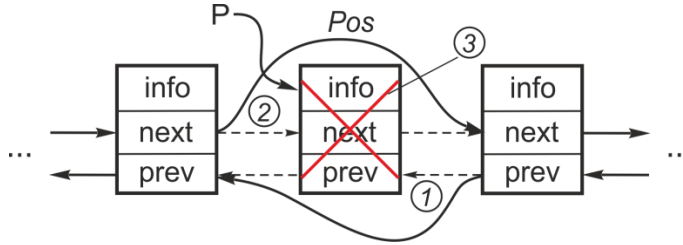


Рис. 20

Програмна реалізація даної операції може мати вигляд:

```
int Pos;   cin >> Pos;                    // Pos – цільова позиція видалення;
P=first;                                  // початок списку;
if (P && Pos>0)                           // список непорожній;
  if (Pos==1)                             // видалення першого елемента;
    if (first==last)                      // перший – єдиний у списку;
      { first=last=NULL; delete P; }
    else                                  // перший – не єдиний у списку;
      { first->next->prev=NULL; first=first->next; delete P; }
  else                                    // видалення не першого елемента;
    {
      while (P && Pos>1)                  // рух до цільової позиції;
        { P=P->next; Pos--; }
      if (P)                              // цільова позиція досягнута;
        if (P==last)                      // видалення останнього елемента;
          { P->prev->next=NULL; last=P->prev; delete P; }
        else                              // видалення всередині списку.
          { P->next->prev=P->prev; P->prev->next=P->next; delete P; }
      else cout << "некоректна позиція видалення" << endl;
    }
else cout<<"список порожній чи некоректна позиція видалення"<<endl;
```

Видалення елемента із заданим значенням інформаційного поля.

Загальна схема виконання дій подібна до вище розглянутої операції. Тому основні випадки видалення елемента зі списку за його інформаційним полем будуть такими самими. Програмна реалізація даної операції може мати вигляд:

```
int X;   cin >> X;                        // X – цільове значення елемента;
P=first;                                  // початок списку;
if (P)                                    // список непорожній;
  if (P->info==X)                         // видалення першого елемента;
    if (first==last)                      // перший – єдиний у списку;
```

```
    { first=last=NULL; delete P; }
   else                                    // перший – не єдиний у списку;
    { first->next->prev=NULL; first=first->next; delete P; }
  else                                     // видалення не першого елемента;
   {
    while (P && P->info!=X)                // рух до цільового елемента;
      P=P->next;
    if (P)                                 // цільовий елемент досягнутий;
      if (P==last)                         // видалення останнього елемента;
        { P->prev->next=NULL; last=P->prev; delete P; }
      else                                 // видалення всередині списку.
        { P->next->prev=P->prev; P->prev->next=P->next; delete P; }
    else cout << "цільового елемента в списку немає" << endl;
   }
else cout << "список порожній – видалення неможливе" << endl;
```

3.3. ЦИКЛІЧНІ СПИСКИ

Циклічні списки – це така особлива трансформація лінійних списків, у яких початок і кінець зв'язані між собою, тобто вони замкнуті в кільце.

Якщо розглядати циклічні однонапрямлені списки, то вказівне поле останнього елемента адресується не в ***NULL***, а на перший елемент, тобто туди, куди вказує ***first***.

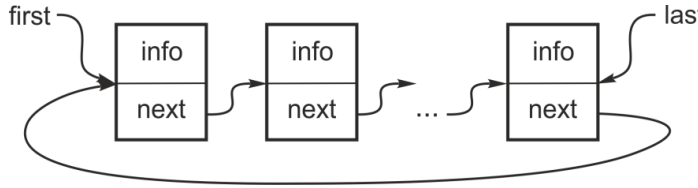


Рис. 21

У випадку циклічних двонапрямлених списків подібний зв'язок між умовними початком і кінцем буде по обох вказівних полях: вказівне поле ***next*** останнього елемента адресується не в ***NULL***, а на перший елемент, тобто туди, куди вказує ***first***; вказівне поле ***prev*** першого елемента адресується не в ***NULL***, а на останній елемент, тобто туди, куди вказує ***last***.

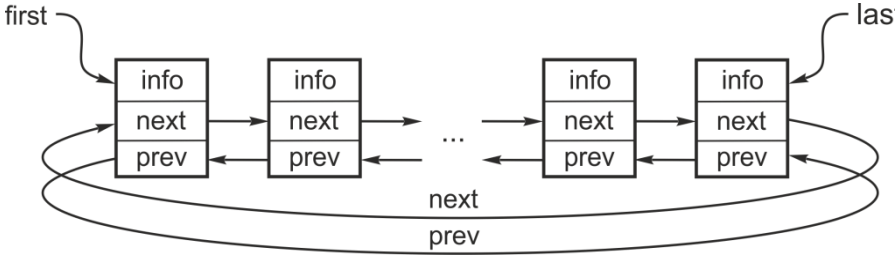


Рис.22

Враховуючи безпосередній зв'язок умовно першого та умовно останнього елементів, від фіксованого вказівника ***last*** можна відмовитися.

У багатьох випадках практичних застосувань динамічних структур даних використання циклічних списків має ряд переваг. Зокрема, це стосується обробки послідовностей функціональних елементів, наприклад, користувацьких меню інформаційних систем. Використання в якості базової схеми саме циклічних структур дозволяє швидко переходити з кінця на початок або навпаки при русі лише в одному напрямі.

Реалізація більшості операцій по обробці циклічних списків не має принципових відмінностей від аналогічних операцій для лінійних списків. Відмінність лише в значенні вказівного поля умовно останнього елемента списку: замість значення ***NULL*** воно має таке саме значення, як і ***first***.

3.3.1. Основні операції по обробці циклічних списків

Не зменшуючи загальності, розглянемо декілька операцій над однонапрямленими циклічними списками. Решта операцій та їх реалізація для двонапрямлених циклічних списків здійснюється за аналогією зі звичайними лінійними списками.

Створення списку

Враховуючи визначальний принцип організації циклічних списків, немає змісту розглядати порожні списки. Проте їх побудова може починатися з етапу порожнього списку, щоб не відходити від загальної схеми конструювання лінійних списків.

Незалежно від того, який принцип формування (стек чи черга) покладено в основу, для реалізації зв'язку останнього елемента з першим потрібно використовувати два фіксовані вказівники: ***first*** – на умовний початок та ***last*** – на умовний кінець списку. Уведення нових елементів передбачає їх розміщення або після елемента, на який раніше вказував ***last***, або перед елементом, на який раніше вказував ***first***, тобто фактично між ними. На перший погляд ніби немає жодної різниці. Проте в першому випадку ***last*** змінює своє значення, а ***first*** залишається незмінним – це буде черга. А в другому випадку ***first*** змінює своє значення, а ***last*** залишається незмінним – це буде стек. Різниця буде в порядку слідування елементів при переходах через вказівне поле ***next***:

1) створення порожнього списку:

Рис.23

```
first=last=NULL;
```

2) уведення першого нового елемента:

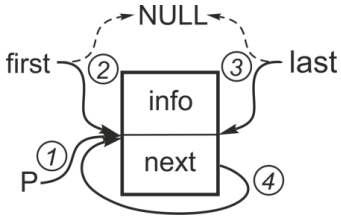


Рис. 24

```
P=new List_1;
cin >> P->info;
first=last=P;
last->next=first;
```

3) уведення решти нових елементів. Цей етап не можна об'єднувати з попереднім. Він матиме відмінності в реалізації при побудові стеку і черги.

У випадку **формування списку за принципом стеку** вказівник ***first*** буде переадресовуватися на кожен новий елемент, а вказівник ***last*** залишатиметься без змін:

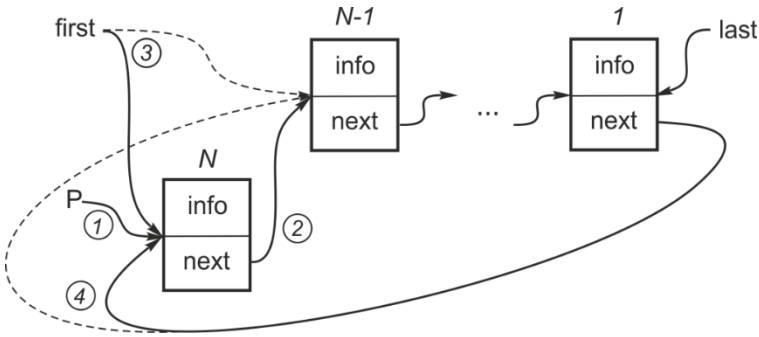


Рис. 25

```
for (int i=1; i<N; i++)
 {
   P=new List_1;
   cin >> P->info;
   P->next=first;
   first=P;
   last->next=first;
 }
```

У випадку **формування списку за принципом черги** вказівник ***last*** буде переадресовуватися на кожен новий елемент, а вказівник ***first*** залишатиметься без змін:

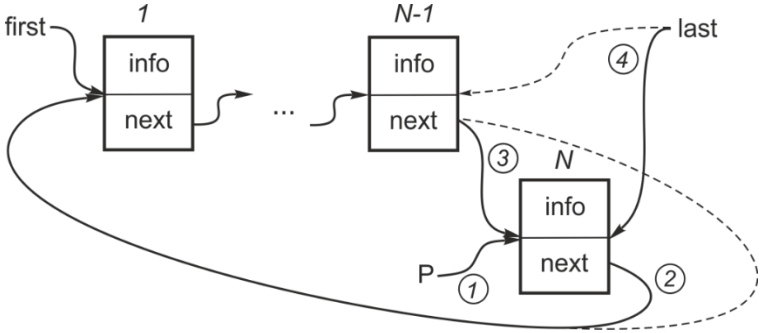


Рис. 26

```
for (int i=1; i<N; i++)
 {
  P=new List_1;
  cin >> P->info;
  P->next=first;
  last->next=P;
  last=P;
 }
```

Пошук елемента в списку

Розглянемо для прикладу **пошук першого входження елемента із заданим значенням інформаційного поля**. Тут повна аналогія з однонапрямленими лінійними списками за виключенням умови зупинки при поверненні на початок списку у випадку негативного результату. Програмна реалізація операції може мати вигляд:

```
int X; cin >> X;
int i=1;
P=first;
if (P)
 {
  while (P->next!=first && P->info!=X)
   { P=P->next; i++; }
  if (P->info==X)
   cout << "елемент є в позиції " << i << endl;
  else
   cout << "цільового елемента немає" << endl;
 }
else
 cout << "список порожній" << endl;
```

Читачу пропонується самостійно реалізувати решту операцій для циклічних одно- та двонапрямлених списків (**вставка, видалення**).

РОЗДІЛ 4
РОЗГАЛУЖЕНІ БАГАТОЗВ'ЯЗНІ СПИСКИ

4.1. ДЕРЕВА

Окрім розглянутих вище лінійних та циклічних однонапрямлених або двонапрямлених списків, в обчислювальній практиці набули широкого використання багатозв'язні списки зі складною розгалуженою будовою. До таких структур даних можна віднести так звані *дерева, ліси*.

Дерево – це багатонапрямлений список з розгалуженою структурою, у якому кожен елемент, окрім інформаційного поля, має деяку фіксовану кількість (не менше двох) вказівних полів на інші елементи цього ж списку.

Число вказівних полів у структурі елементів дерева, що є визначальним при оголошенні типу списку, називається *розмірністю дерева*. Кожен із таких вказівників може або адресувати якийсь елемент списку, або бути вільним, тобто вказувати «в нікуди». Загальна кількість елементів, що утворює такий список, визначає *розмір дерева*.

Такі списки за будовою нагадують звичайні дерева, тільки перевернуті коренем догори. Пропоноване трактування будови списків-дерев є достатньо умовним. Для зручності та з метою полегшення розуміння алгоритмів обробки домовимося, що перший елемент такої структури даних – це так звана *коренева вершина*, або просто *корінь*. Він – єдиний елемент на першому рівні дерева. Корінь може мати деяку кількість прилеглих йому вершин другого рівня. У свою чергу ці вершини можуть мати свої прилеглі елементи третього рівня і т.д. У кожному іншому випадку кількість прилеглих елементів може бути різна, але не може перевищувати розмірності дерева. Елементи всіх рівнів, починаючи з другого, утворюють так звану *крону дерева*.

Як вже було сказано, всі елементи дерева, у тому числі й коренева вершина, можуть мати різну кількість прилеглих їм елементів наступного рівня. Вершини дерева, які не мають жодного прилеглого елемента наступного рівня, тобто всі їх вказівні поля вільні, називатимемо *висячими вершинами* або *листками дерева*. Листки можуть бути на будь-якому рівні дерева. Вершини дерева, які мають хоча б один прилеглий елемент наступного рівня та хоча б одне вільне вказівне поле, тобто частина їх вказівників є зв'язаними, а частина вказує «в нікуди», називатимемо *напіввисячими вершинами* або *гілками дерева*. Вершини дерева, всі вказівні поля яких є зв'язаними з прилеглими елементами наступного рівня, називатимемо *вузловими вершинами* або просто *вузлами*. Очевидно, крону дерева утворюють усі вузли, гілки і листки за виключенням кореня. Сам корінь дерева може бути як вузлом, так і гілкою, і навіть листком. В останньому випадку дерево складається лише з одного елемента.

На відміну від лінійних або циклічних двонапрямлених списків у деревах немає послідовного доступу до елементів. Точніше, переміщення по дереву передбачає послідовні переходи з поточного рівня на наступний, але при цьому на кожному рівні можливий перехід на один із декількох елементів,

що адресуються вказівними полями поточної вершини. Очевидно, що доступ до елементів дерева реалізується через кореневу вершину. Усім операціям по обробці дерев у якості параметра передаватиметься саме корінь. Тому при їх програмній реалізації обов'язково має бути фіксований вказівник на кореневу вершину. Позначатимемо його ***top*** – вершина.

Незалежно від розмірності всі дерева є рекурсивними структурами даних. Адже дерево – це або порожній список, або непорожній список, коренева вершина якого в якості прилеглих структур даних має дерева такої ж розмірності, але меншого розміру – *піддерева*.

Подібний спосіб конструювання списків можна застосувати і у випадках, коли їх елементами є цілі дерева. Такі списки з дерев утворюють більш складну динамічну структуру даних – *ліс*. Програмування з використанням структурованих (складених) комбінованих типів даних з альтернативними полями (***об'єднання*** в мовах C/C++, ***записи з варіантами*** в мовах Паскаль, Delphi) дозволяють будувати ліси навіть із дерев різної розмірності.

4.2. БІНАРНІ ДЕРЕВА

В обчислювальній практиці широкого застосування набув один із простіших різновидів розгалужених списків – *бінарні дерева*. З назви зрозуміло, що такі дерева мають розмірність 2, тобто кожен елемент, окрім інформаційного поля, матиме два вказівники на два прилеглих йому елементи наступного рівня. Такі списки зручно використовувати при програмній реалізації алгоритмів, які передбачають організацію динамічної структури даних за деякою бінарною ознакою: умовно зліва від поточного елемента розміщуватиметься *піддерево з меншими ключами* або *ліве піддерево*, а умовно справа – *піддерево з більшими ключами* або *праве піддерево*. Зрозуміло, що можуть бути елементи з однаковими інформаційними полями. Попередньо слід визначитися з напрямом зв'язування елементів з рівними ключами: вони попадатимуть виключно або в ліве, або в праве піддерево. Адже бінарність ознаки передбачає дві протилежні умови. Якщо використати дерево розмірності **3**, то можна реалізувати динамічну структуру даних за тринарною ознакою (з умовами переходу: менше, дорівнює, більше).

При роботі з бінарними деревами, окрім аналізу вузлів, гілок та листків, також визначають їх габаритні параметри – *висоту* та *ширину дерева.*

Шириною бінарного дерева називається сумарне число переходів, виконаних по вказівнику вліво від кореня до найменшого (крайнього лівого) елемента та по вказівнику вправо від кореня до найбільшого (крайнього правого) елемента, та збільшене на одиницю за рахунок самого кореня. Іншими словами, *ширина бінарного дерева* – це число тих його вершин, що зв'язані переходами вліво і (або) вправо та утворюють ламану, яка огинає зверху дерево

від крайньої лівої вершини через кореневу вершину до крайньої правої вершини.

Висотою бінарного дерева називається максимальне число переходів, виконаних по вказівниках вліво і (або) вправо від кореня до найбільш віддаленого листка, та збільшене на одиницю за рахунок самого кореня. Іншими словами, *висота бінарного дерева* – це число рівнів його вершин від кореня до найбільш віддаленого листка.

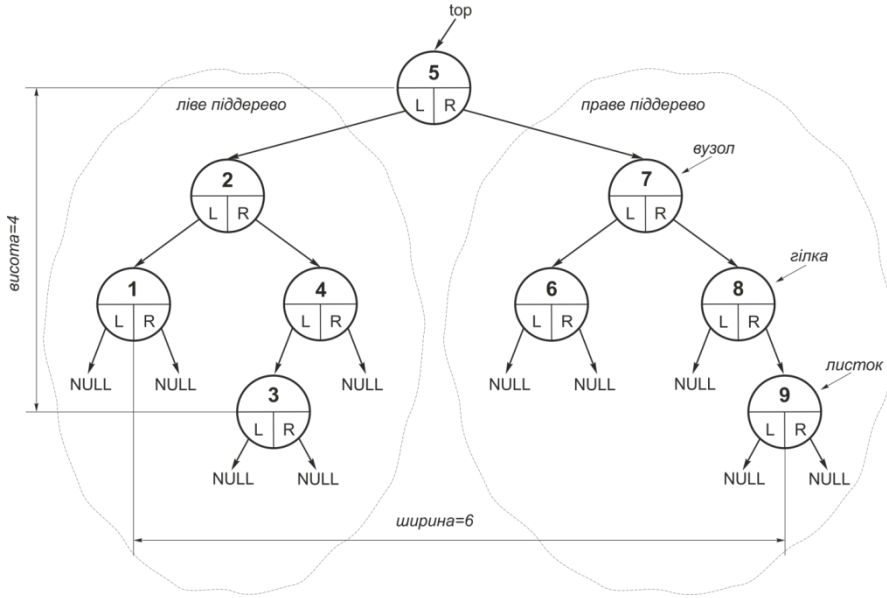


Рис. 27

4.2.1. Оголошення типу елемента списку

Елементи бінарного дерева, окрім основної за умовами задачі інформації, мають два вказівники на прилеглі їм зліва і справа піддерева. Позначимо їх відповідно ***left*** і ***right***. У подальшому для простоти коду в окремих функціях і на рисунках будуть використовуватися позначення ***L*** і ***R***. Але варто пам'ятати, що ***L*** – це те саме, що ***left***, а ***R*** – це те саме, що ***right.***

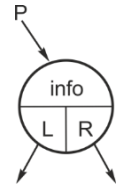


Рис. 28

Вказівники ***left*** і ***right*** повинні мати такий самий тип, що й сам елемент списку. Для ідентифікації елемента (для доступу до нього) потрібно мати деяку статичну змінну ***P***, що є вказівником цього ж типу.

Зрозуміло, що вказівник на кореневу вершину ***top*** матиме такий самий тип, як і решта елементів списку.

Знову приймаємо в якості базового типу інформаційного поля даних тип цілих чисел ***int***. Програмна реалізація мовою C++ оголошення спискового типу та деяких необхідних статичних змінних може бути такою:

```
struct Bin_tree
 {
  int info;
  Bin_tree * left, * right;
 };
Bin_tree * top, * P, * Q;
```

Тут ***Bin_tree*** – ім'я структури й одночасно базовий тип для вказівного типу елемента списку; ***top, P, Q*** – відповідно вказівники на кореневу вершину і деякі поточні елементи для виконання операцій над списком.

4.2.2. Основні операції по обробці бінарних дерев

Створення списку

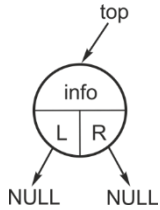
Бінарні дерева утворюють розгалужену структуру даних, конфігурація якої визначається базовим принципом формування, що був сформульований вище: зліва від поточного елемента розміщуватиметься піддерево з меншими ключами, а справа – піддерево з більшими ключами. Для визначеності домовимося, що нові елементи з ключами, рівними ключам поточного елемента, потраплятимуть до лівого піддерева.

У процесі **ініціалізації та формування бінарного дерева** можна виділити три етапи.

1) порожній список:

```
top=NULL;
```

2) уведення першого елемента – кореневої вершини:



top=new Bin_tree;
cin >> top->info;
top->left=top->right=NULL;

Рис. 29

3) уведення решти нових елементів. Очевидно, що введення решти елементів суттєво відрізнятиметься від введення кореневої вершини:

- кожен новий елемент долучається до дерева шляхом порівняння з уже існуючими елементами, починаючи від кореня;
- якщо ключ нового елемента *менший або рівний* від ключа поточного елемента і лівий вказівник із поточного елемента *вільний*, тобто вказує «в нікуди», то лівий вказівник із поточного елемента зв'язується з новим елементом; власні вказівники з нового елемента вказують «в нікуди»;
- якщо ключ нового елемента *менший або рівний* від ключа поточного елемента і лівий вказівник із поточного елемента вже *зв'язаний* з іншим елементом, то новий елемент переходить на наступний рівень дерева відносно поточного елемента по його лівому вказівнику, і алгоритм формування дерева повторюється на новому рівні відносно вже іншого поточного елемента;
- якщо ключ нового елемента *більший* від ключа поточного елемента і правий вказівник із поточного елемента *вільний*, тобто вказує «в нікуди», то правий вказівник із поточного елемента зв'язується з новим елементом; власні вказівники з нового елемента вказують «в нікуди»;
- якщо ключ нового елемента *більший* від ключа поточного елемента і правий вказівник із поточного елемента вже *зв'язаний* з іншим елементом, то новий елемент переходить на наступний рівень дерева відносно поточного елемента по його правому вказівнику, і алгоритм формування дерева повторюється на новому рівні відносно вже іншого поточного елемента.

Таким чином, дерево розростатиметься як в ширину, так і в глибину. Для кожного поточного елемента лівіше і нижче розміщуватимуться усі елементи, що не перевищують поточного, а правіше і нижче розміщуватимуться усі елементи, що більші від поточного. При цьому один і той самий набір ключів, що матиме інший порядок, даватиме відмінне за конфігурацією дерево (рис. 30).

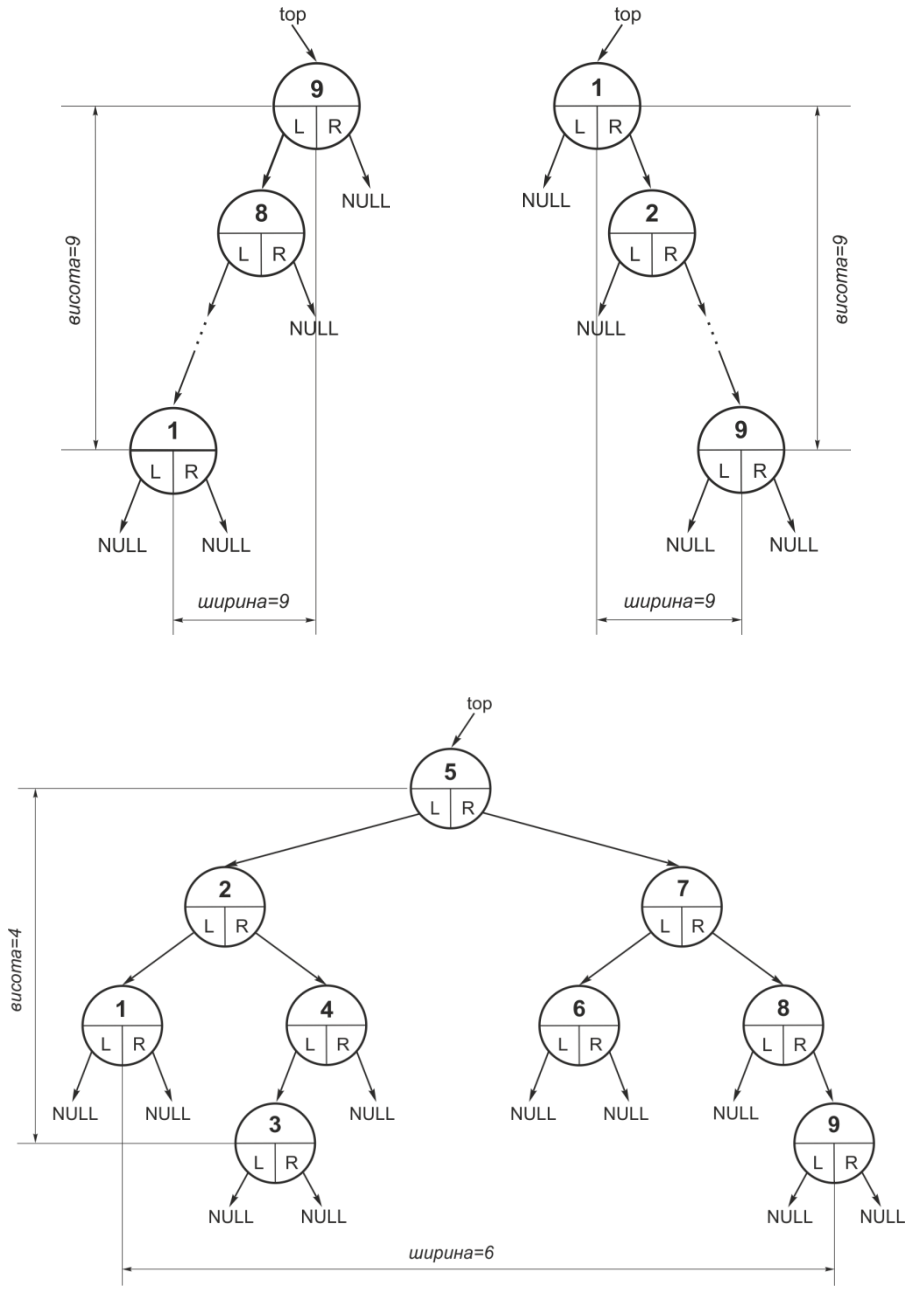

top
9
L R
8
L R
1
L R
NULL
NULL
NULL
NULL
NULL
высота=9
ширина=9
top
1
L R
2
L R
9
L R
NULL
NULL
NULL
NULL
NULL
высота=9
ширина=9
top
5
L R
2
L R
7
L R
1
L R
4
L R
6
L R
8
L R
3
L R
9
L R
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL
высота=4
ширина=6

Рис. 30

Сформульований алгоритм формування бінарного дерева має чітко виражений рекурсивний характер: при переході на наступний рівень виконуються ті самі дії, що й на попередньому. Тому для його програмної реалізації доцільно використовувати рекурсивну підпрограму, яка в якості формальних параметрів матиме два вказівники: ***Q*** – на новий елемент та ***P*** – на поточний елемент у дереві. Програмна реалізація такої рекурсивної підпрограми може мати вигляд функції:

```
void New_element(Bin_tree * Q, Bin_tree * P)
 {
  if (Q->info<=P->info)
    if (!P->left) P->left=Q; else New_element(Q, P->left);
  else
    if (!P->right) P->right=Q; else New_element(Q, P->right);
 }
```

Така функція викликатиметься циклічно для введення чергового нового елемента в дерево, що формується. Щоразу в якості фактичних параметрів передаватимуться вказівник на новий елемент та фіксований вказівник ***top*** на корінь дерева.

```
for (int i=1; i<N; i++)
 {
  Q=new Bin_tree;
  cin >> Q->info;
  Q->left=NULL;
  Q->right=NULL;
  if (top) New_element(Q, top);
  else cout << " відсутня коренева вершина дерева " << endl;
 }
```

Зауваження. До порожнього списку не можна застосовувати рекурсивну підпрограму виду ***New_element***. Адже в цьому випадку в якості формального параметра ***P*** мало б передаватися фактичне значення ***NULL***. А це привело б до недопустимої операції розіменування вказівника зі значенням ***NULL*** у першому і єдиному операторі підпрограми – операторі розгалуження ***if (Q->info<=P->info) …else …*** .

Перегляд списку

У силу рекурсивного характеру такої динамічної структури даних, як бінарне дерево, очевидно, що всі операції по їх обробці реалізуватимуться

рекурсивними підпрограмами, подібними до функції формування дерева ***New_element***. При цьому обхід дерева здійснюється за наступною схемою:

1) рух вліво, поки лівий вказівник поточної вершини *не вільний*, тобто *зв'язаний* із прилеглим йому елементом – це є обробка лівого піддерева відносно поточної вершини;

2) якщо лівий вказівник поточної вершини *вільний*, тобто вказує «в нікуди», то обробляється інформаційне поле цього елемента;

3) якщо правий вказівник поточної вершини *зв'язаний* із прилеглим йому елементом, то виконується перехід управо, і вся послідовність дій повторюється – це є обробка правого піддерева відносно поточної вершини.

Якщо на деякому етапі обходу дерева разом із поточною кореневою вершиною повністю обійдено її ліве і праве піддерева, то це означає повну обробку ширшого фрагмента дерева, що в свою чергу є піддеревом якоїсь іншої кореневої вершини вищого рівня. Тоді виконується рекурсивний відкіт на цей попередній рівень. Після повного обходу всього бінарного дерева зверху вниз і зліва направо останній відкіт знову поверне поточний вказівник на самий верхній корінь дерева ***top***.

Характерною особливістю такого рекурсивного обходу списку є те, що обробка вершини здійснюється в порядку зростання (точніше – неспадання) їх ключів. **Тобто перегляд бінарного дерева автоматично забезпечує сортування вхідної послідовності елементів даних!**

На рис. 31 зображено послідовність кроків при обході бінарного дерева, що перенумеровані в кружечках. Штриховими стрілками позначені кроки прямого ходу рекурсії, тобто рух у глибину дерева; словом «обробка» та відповідним номером позначено виконання основної дії над поточним елементом; штрихпунктирними стрілками позначені кроки зворотного ходу рекурсії, тобто відкоти по дереву на попередній рівень.

Очевидно, що єдиним формальним параметром такої рекурсивної підпрограми буде вказівник на поточний елемент дерева. Фактичним параметром у точці першого виклику підпрограми буде корінь дерева ***top***. Програмна реалізація підпрограми-функції може мати вигляд:

```
void Print_element(Bin_tree * P)
 {
  if (P->left) Print_element(P->left);
  cout << P->info << endl;
  if (P->right) Print_element(P->right);
 }

  . . .

if (top) Print_element(top); else cout << "список порожній" << endl;
```

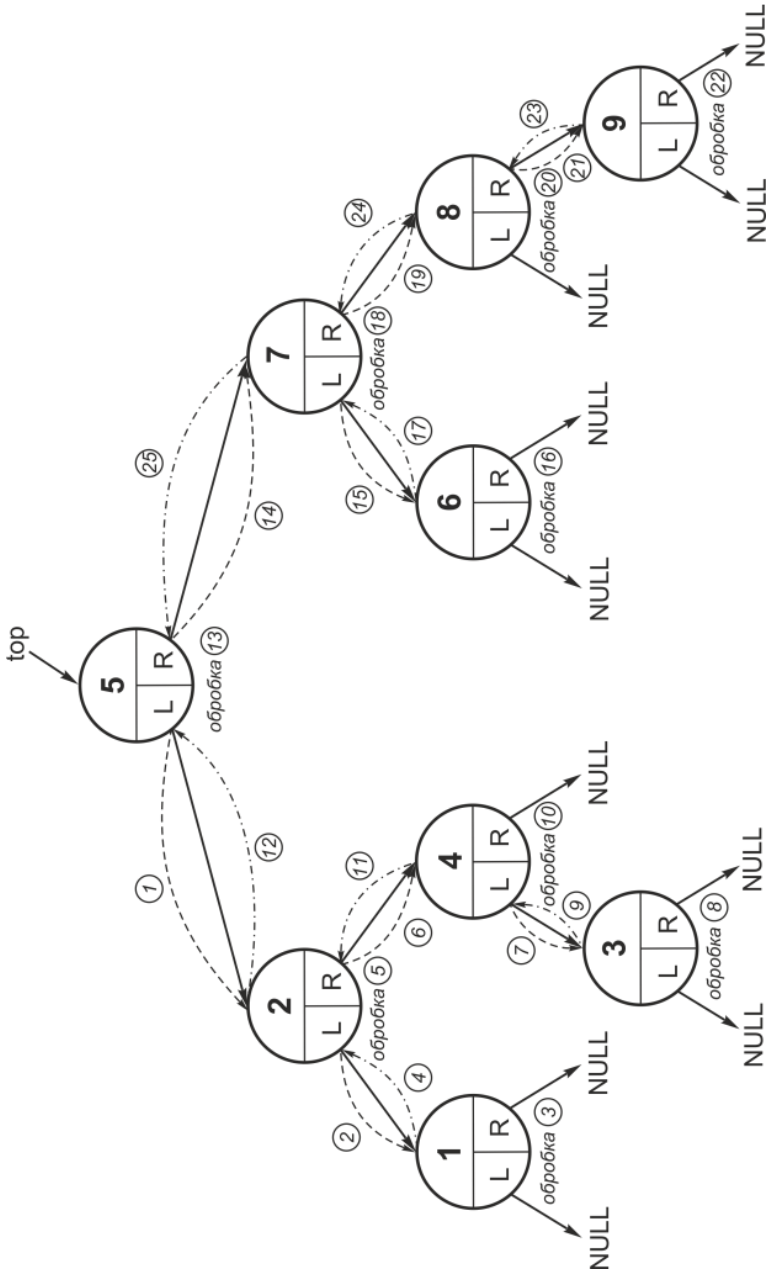

top
5
L
R
обробка
7
обробка
6
обробка
8
обробка
9
обробка
2
обробка
4
обробка
1
обробка
3
обробка
NULL

Рис. 31

Пошук елемента в списку

У випадку бінарних дерев результатом пошуку може бути лише повідомлення про наявність чи відсутність елементів з цільовим ключем. З попереднього матеріалу зрозуміло, що говорити про позицію входження немає змісту. Як вже було зазначено, практично всі операції по обробці бінарних дерев реалізуються за спільною рекурсивною схемою. Рекурсивна функція для пошуку з одночасним підрахунком кількості входжень цільового значення може мати вигляд:

```
int X;
int k=0;
void Find_element(Bin_tree * P)
 {
  if (P->left) Find_element(P->left);
  if (P->info==X) k++;
  if (P->right) Find_element(P->right);
 }

  .  .  .

cin >> X;
if (top)
 {
  Find_element(top);
  if (k) cout << "усього встановлено " << k << " входжень" << endl;
  else cout << "цільового елемента немає" << endl;
 }
else cout << "список порожній" << endl;
```

Вставка нового елемента в список

Дана операція реалізується вже розглянутою функцією ***New_element***. Новий елемент займе відповідне йому місце в списку.

Видалення елемента зі списку

Видалення є протилежною операцією до вставки. Природно, у силу визначального принципу побудови бінарних дерев, можна говорити лише про видалення елемента із заданим значенням. Не зменшуючи загальності, домовимося про видалення одного елемента з можливо декількох однакових елементів із заданим значенням. При цьому немає змісту говорити про порядок такого елемента в бінарному дереві. У бінарних деревах наявність декількох

елементів з однаковим значенням не передбачає обов'язкового їх послідовного розміщення в межах деякої гілки. У залежності від порядку введення однакові елементи можуть бути суттєво віддаленими. Можна лише стверджувати, що видалятиметься найближчий до кореневої вершини елемент із можливо декількох однакових елементів із заданим значенням.

Операція видалення, як і всі попередні, реалізуватиметься рекурсивною функцією. Проте вона потребує більш ґрунтовного і детального обговорення. Очевидно, що пошук елемента із заданим значенням, що видаляється, слід організувати так, щоб отримати вказівник ***P*** на вершину, для якої цей елемент є прилеглим зліва чи справа. У залежності від того, яку саме вершину потрібно вилучити з дерева (листок, гілку, вузол чи корінь), виникає потреба в різних варіантах трансформації бінарного дерева. Функцію видалення слід будувати так, щоб усі можливі варіанти були окремими реалізаціями багатовіткового розгалуженого рекурсивного алгоритму.

Наперед забігаючи, проведемо оголошення інтерфейсу такої функції:

void Delete_el(Bin_tree * P, int X, int k).

Тут перший параметр ***P*** – це вказівник на вершину, для якої елемент зі значенням другого параметра ***X***, що має видалятися, є прилеглим зліва чи справа. Третій параметр ***k*** – числовий код із можливими значеннями **0, -1, 1,** який визначатиме напрям переходу від вершини з вказівником ***P*** на прилеглі їй елементи. Значенню ***k*=0** відповідає окремий аналіз кореневої вершини; значенню ***k*=-1** відповідає перехід на наступний рівень вліво від поточної вершини з вказівником ***P***, а значенню ***k*=1** відповідає перехід на наступний рівень вправо від поточної вершини з вказівником ***P***.

Аналіз дерева починається з кореневої вершини. Цьому відповідає такий виклик функції видалення:

Delete_el(top, X, 0).

Вказівник ***top*** є глобальною змінною, через яку здійснюється доступ до бінарного дерева. Фактичному параметру ***top*** виклику функції відповідатиме формальний параметр ***P*** в тілі цієї функції. Звичайно, що операцію видалення можна застосовувати лише до непорожнього дерева. Спочатку окремо аналізується коренева вершина, оскільки для кореня немає вершини вищого рівня. Якщо коренева вершина співпадає із шуканим значенням, то проводиться видалення. При цьому можливі наступні варіанти реалізації в залежності від виду кореневої вершини:

- коренева вершина є листком:
  if (!P->L && !P->R) { top=NULL; delete P; }
- коренева вершина є гілкою, маючи ліве підддерево:
  if (P->L && !P->R) { top=P->L; delete P; }
- коренева вершина є гілкою, маючи праве підддерево:
  if (!P->L && P->R) { top=P->R; delete P; }
- коренева вершина є вузлом:
  if (P->L && P->R)
  { Q=move_R(P->L); Q->R=P->R; top=P->L; delete P; }

// або ***{ Q=move_L(P->R); Q->L=P->L; top=P->R; delete P; }***

В останньому прикладі коду використано додаткові функції ***move_R*** або ***move_L***. Функція ***move_R*** здійснює переміщення на крайній правий елемент лівого піддерева відносно елемента, що вилучається, і повертає своїм значенням вказівник на цей елемент. Аналогічно функція ***move_L*** здійснює переміщення на крайній лівий елемент правого піддерева відносно елемента, що вилучається, і повертає своїм значенням вказівник на цей елемент. Програмна реалізація цих функцій може мати вигляд:

```
Bin_tree * move_R(Bin_tree * P)
  {
    Bin_tree * Q=P;
    while (Q->R) Q=Q->R;
    return Q;
  }

Bin_tree * move_L(Bin_tree * P)
  {
    Bin_tree * Q=P;
    while (Q->L) Q=Q->L;
    return Q;
  }
```

Це потрібно, щоб при видаленні вершини-вузла зв'язати між собою обидва піддерева, що залишаються: або вказівник вправо ***R*** крайнього правого елемента лівого піддерева адресується на праве піддерево, або вказівник вліво ***L*** крайнього лівого елемента правого піддерева адресується на ліве піддерево (рис. 34, 35).

Якщо аналіз кореневої вершини не дав бажаного результату, то для кожного з піддерев зліва і справа від поточної вершини (початково – кореня) застосовується така сама послідовність дій. З цією метою виконуються рекурсивні виклики функції ***Delete_el*** із відповідними значеннями третього параметра:

- видалення у лівому піддереві
  if (P->L && P->info>X) Delete_el(P, X, -1)

або

- видалення у правому піддереві
  if (P->R && P->info<X) Delete_el(P, X, 1)

Альтернативою є випадок, коли поточна вершина вже не є коренем дерева. Тоді виконуються рекурсивні переходи по дереву зі зміщеннями вліво або вправо від поточної вершини. Якщо встановлено елемент із заданим значенням ***X***, що має видалятися, то послідовність дій така сама, як і для

розглянутої вище кореневої вершини. Звичайно, що адресація елементів буде складнішою, оскільки поточний вказівник має адресувати не той елемент, що видаляється, а попередній перед ним. Також потрібно розрізняти напрями переходів по дереву, що залежать від значення третього параметра функції ***Delete_el*** (**-1** відповідає переходу вліво, а **1** відповідає переходу вправо).

Схеми дій, що виконуються в різних випадках видалення елемента, наведені на рис. 32-35. Не зменшуючи загальності, ілюструються випадки вилучення довільних внутрішніх елементів (не кореневої вершини): на рис. 32 проілюстровано видалення листка, на рис. 33 – видалення правої гілки, на рис. 34 – видалення вузла із заміною на ліве піддерево, на рис. 35 – видалення вузла із заміною на праве піддерево.

На цих рисунках для кожного можливого випадку видалення елемента позначено послідовність основних дій, що перенумеровані в кружечках. Номером **1** всюди позначено цілу складену дію – пошук елемента, що передує елементу, який має видалятися, і встановлення вказівника ***P*** на цю вершину. На рис. 34 і рис. 35 вказівник ***P*** співпадає з фіксованим вказівником ***top*** на корінь дерева, оскільки ці приклади ілюструють видалення елемента з другого рівня дерева. На цих самих рисунках дія з номером **2**, що позначена стрілкою з пунктирним розривом, – це робота додаткових функцій ***move_R*** або ***move_L***, що забезпечують переміщення відповідно на крайній правий елемент лівого піддерева відносно елемента, що вилучається, або на крайній лівий елемент правого піддерева відносно елемента, що вилучається.

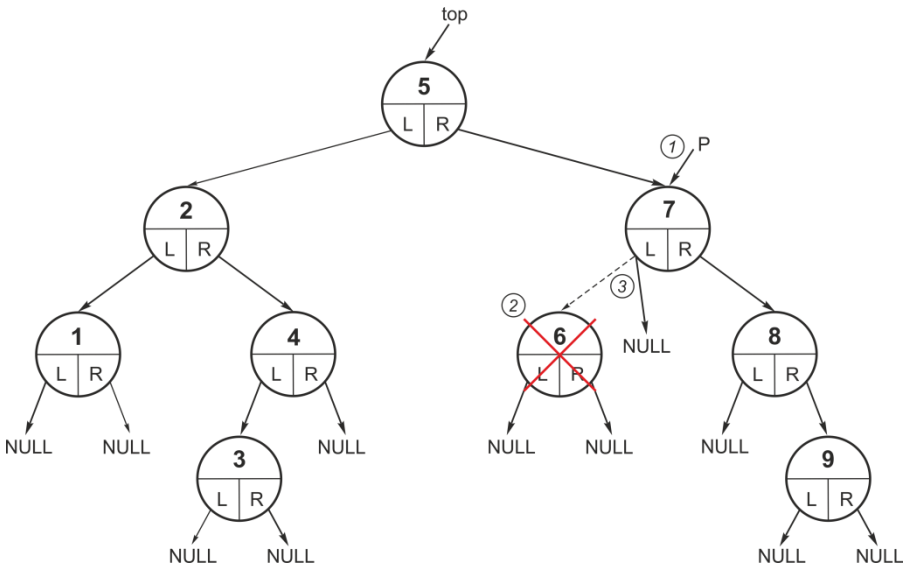


Рис. 32. Видалення листка

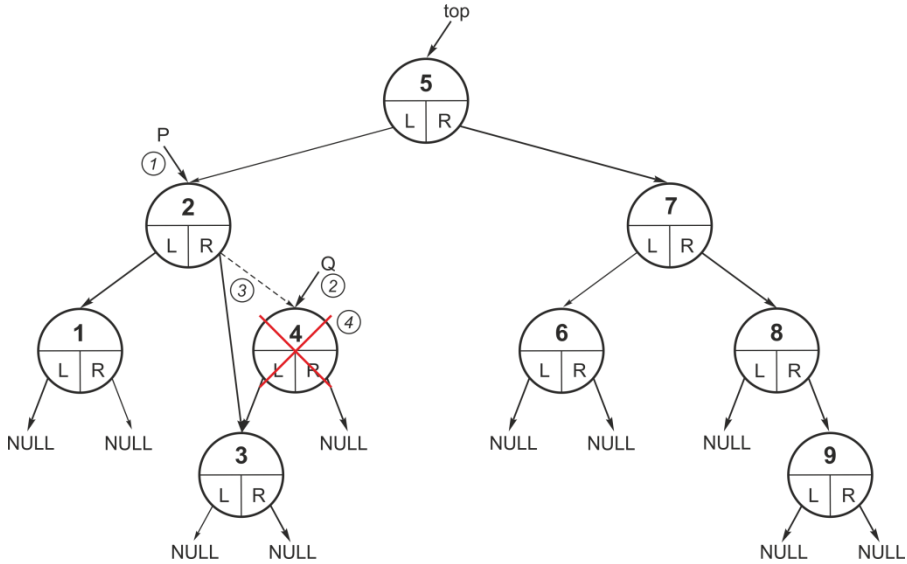


Рис. 33. Видалення правої гілки

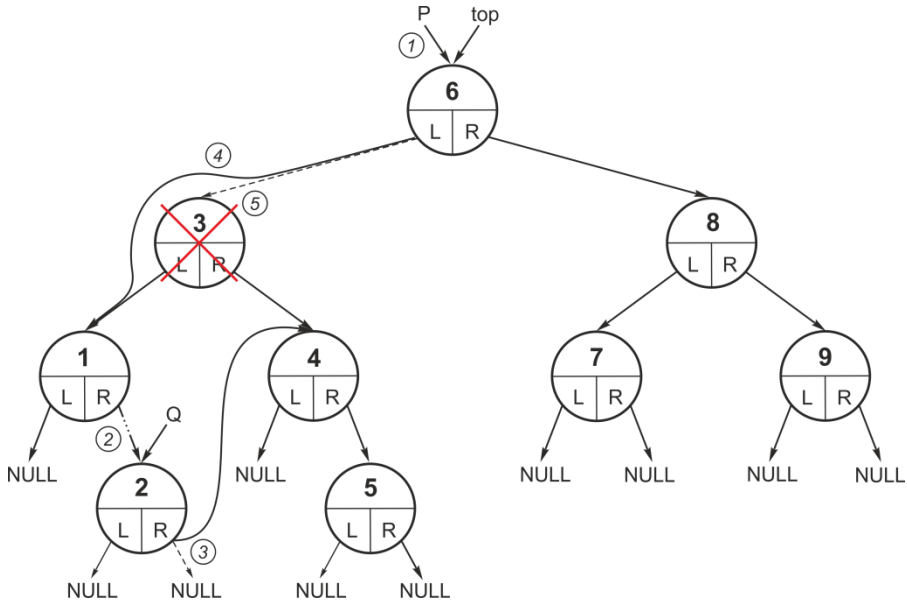


Рис. 34. Видалення вузла із заміною на ліве піддерево

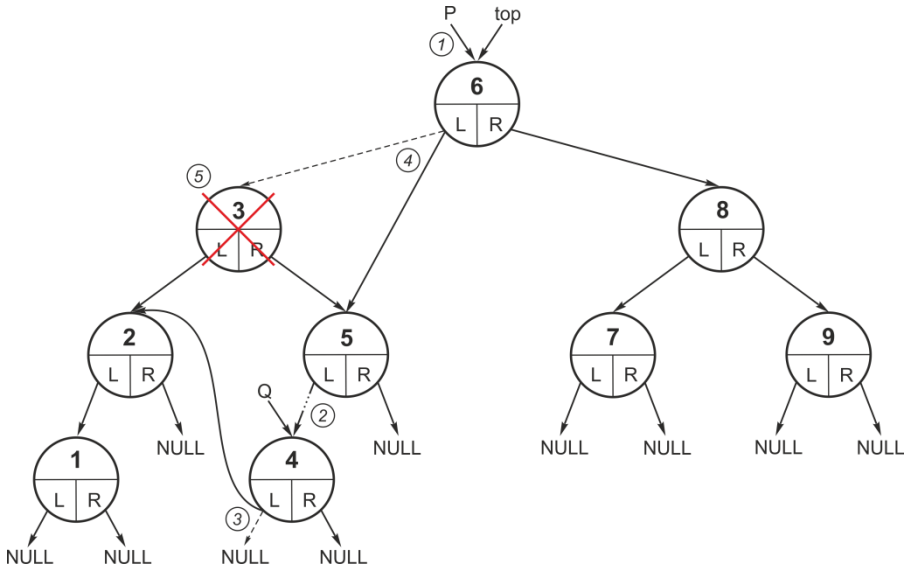


Рис. 35. Видалення вузла із заміною на праве піддерево

Остаточна програмна реалізація функції видалення елемента може мати вигляд:

```
void Delete_el(Bin_tree * P, int X, int k)
  {
    if (P)                                          // не порожнє дерево
      if (k==0)                                     // корінь
      {
        if (P->info==X)                             // той, що треба
          if (!P->L && !P->R)                       // листок
            { top=NULL; delete P; }
          else
            if (P->L && !P->R)                      // гілка зліва
              { top=P->L; delete P; }
            else
              if (!P->L && P->R)                    // гілка справа
                { top=P->R; delete P; }
              else                                  // вузол
                { Q=move_R(P->L); Q->R=P->R; top=P->L; delete P; }
// або            { Q=move_L(P->R); Q->L=P->L; top=P->R; delete P; }
        else                                        // не той, що треба
```

```
      if (P->L && P->info>X) Delete_el(P, X, -1);          // ліве піддерево
      else
        if (P->R && P->info<X) Delete_el(P, X, 1);         // праве піддерево
    }
    else                                                   // не корінь
      if (k==-1)
        {
          if ( P->L && P->L->info>X) Delete_el(P->L, X, -1);  // рух вліво
          if (P->L && P->L->info==X)                       // той, що треба
            {
              if (!P->L->L && !P->L->R)                    // листок
                { delete P->L; P->L=NULL; }
              else
                if (P->L->L && !P->L->R)                   // гілка зліва
                  { Q=P->L; P->L=Q->L; delete Q; }
                else
                  if (!P->L->L && P->L->R)                 // гілка справа
                    { Q=P->L; P->L=Q->R; delete Q; }
                  else                                     // вузол
                    { Q=move_R(P->L->L); Q->R=P->L->R;
                      Q=P->L; P->L=Q->L; delete Q; }
// або                 { Q=move_L(P->L->R); Q->L=P->L->L;
// або                   Q=P->L; P->L=Q->R; delete Q; }
            }
          else
            if (P->L && P->L->info<X) Delete_el(P->L, X, 1);  // крок вправо
        }
      if (k==1)
        {
          if (P->R && P->R->info<X) Delete_el(P->R, X, 1);  // рух вправо
          if (P->R && P->R->info==X)                       // той, що треба
            {
              if (!P->R->L && !P->R->R)                    // листок
                { delete P->R; P->R=NULL; }
              else
                if (P->R->L && !P->R->R)                   // гілка зліва
                  { Q=P->R; P->R=Q->L; delete Q; }
                else
                  if (!P->R->L && P->R->R)                 // гілка справа
                    { Q=P->R; P->R=Q->R; delete Q; }
```

```
                else                                    // вузол
                  { Q=move_R(P->R->L); Q->R=P->R->R;
                    Q=P->R; P->R=Q->L; delete Q; }
// або               { Q=move_L(P->R->R); Q->L=P->R->L;
// або                 Q=P->R; P->R=Q->R; delete Q; }
            }
          else
            if (P->R && P->R->info>X) Delete_el(P->R, X, -1);   // крок вліво
        }
    }
```

Зауваження.

1. Дана функція видаляє один елемент із врахуванням конфігурації дерева. На це слід зважати, коли список містить декілька елементів з однаковим значенням, що співпадає з ключем видалення. У структурі бінарного дерева елементи з однаковими значеннями не обов'язково мають бути поряд у межах деякої гілки. Ті елементи, що вводилися в список відносно раніше, опиняються в ньому на вищих рівнях. Тому видалятиметься найближча до кореня вершина із можливо декількох однакових елементів із заданим значенням.

2. Запропонована функція видалення елемента має значно більшу складність програмної реалізації в порівнянні з іншими операціями. Така складність не є її недоліком, це є її характерною особливістю. Великий обсяг програмного коду пов'язаний із необхідністю об'єднати в одне ціле різні реалізації операції для різних видів елементів (корінь, листок, гілка, вузол).

3. Автори зауважують, що ними не було знайдено у відкритих джерелах публікацій із прикладами цілісних програмних реалізацій операції видалення елемента з бінарного дерева в загальному випадку.

ЛІТЕРАТУРА

1. Воробйова О.Д., Глазунова Л.В. Алгоритми та структури даних: Конспект лекцій. Частина 1. Структури даних. Одеса : ОНАЗ ім. О.С. Попова, 2017. 48 с.
2. Воробйова О.Д., Глазунова Л.В. Алгоритми та структури даних: Конспект лекцій. Частина 2. Алгоритми пошуку, стиснення даних, внутрішнього та зовнішнього сортування, алгоритми на графах. Одеса : ОНАЗ ім. О.С. Попова, 2017. 52 с.
3. Коротєєва Т.О. Алгоритми та структури даних: Навч. посібник. Львів : Видавництво Львівської політехніки, 2014. 280 с.
4. Креневич А.П. Алгоритми і структури даних: Підручник. К. : ВПЦ «Київський Університет», 2021. 200 с.
5. Мелешко Є.В., Якименко М.С., Поліщук Л.І. Алгоритми та структури даних: Навчальний посібник для студентів технічних спеціальностей денної та заочної форми навчання. Кропивницький : Видавець Лисенко В.Ф., 2019. 156 с.
6. Стратієнко Н.К., Годлевський М.Д., Бородіна І.О. Алгоритми і структури даних: Практикум: Навч. посібник. Харків : НТУ «ХПІ», 2017. 224 с.
7. Ткачук В.М. Алгоритми і структури даних: Навч. посібник. Івано-Франківськ : Видавництво Прикарпатського національного університету імені Василя Стефаника, 2016. 286 с.
8. Томас Г. Кормен, Чарлз Е. Лейзерсон, Роналд Л. Рівест, Кліфорд Стайн. Вступ до алгоритмів. Переклад з англійської третього видання. Навчальне видання. К. : «К.І.С. », 2019. 1288 с.

Для нотаток

Навчально-методичне видання

СЯСЬКИЙ Володимир Андрійович

БАБИЧ Степанія Михайлівна

АЛГОРИТМИ І СТРУКТУРИ ДАНИХ

Навчальний посібник

Підп. до др. 02.05.2023.
Формат 60x84 $^{1}/_{16}$.
Папір офсет.
Друк офсет.
Гарнітура Times.
Ум. друк. арк. 7,2.
Обл. вид. арк. 7,2.
Тираж 100 прим.

Видавець: Олег Зень
Свідоцтво суб'єкта видавничої справи
серія РВ № 26 від 6 квітня 2004 р.
вул. Кн. Романа, 9/24, м. Рівне, 33022,
тел. 0680250674, olegzen@ukr.net

Друк: кафедра інформаційних технологій та моделювання РДГУ
вул. Пластова, 31, м. Рівне, 33000
тел. 0673621961