6. Сортування структур даних

Тепер, коли ми вміємо кількома способами впорядковувати послідовність чисел, розглянемо складніші випадки сортування даних. Наприклад, сортування рядків заданої матриці та різні способи впорядкування записів файлу.

6.1. Впорядкування рядків матриці

Раніше ми вже описували алгоритми опрацювання матриць і використовували в програмах динамічне виділення пам'яті для матриць потрібних розмірів. У прикладах цього параграфу заради різноманітності ми використаємо матриці статично заданого розміру. Який спосіб створення матриці використати, залежить від умови задачі: якщо розміри відомі наперед, то статичне оголошення зекономить час і спростить передавання матриці функціям. У всьому іншому статичні та динамічні матриці не відрізняються.

Задача 26. *Задано матрицю A:10×12. Впорядкувати за зростанням кожен рядок матриці.*

Розв'язок цієї задачі є простим і очевидним узагальненням алгоритму впорядкування одновимірного масиву: з метою впорядкування кожного рядка, необхідно просто десять разів виконати процедуру сортування. Який з описаних раніше методів впорядкування вибрати? Враховуючи оцінки ефективності, наведені у попередньому параграфі, зупинимо свій вибір на сортуванні простими вставками і використаємо оголошену раніше процедуру *SimpleInsertSort*:

```
// Розміри матриць задано для всієї програми
const unsigned n = 10;
const unsigned m = 12;

// Уведення матриці (поелементне, за рядками)
void ReadMatrix(int a[n][m])
{
	cout << "Введіть елементи матриці " << n << 'x' << m << '\n';
	for (unsigned i = 0; i < n; ++i)
		for (unsigned j = 0; j < m; ++j) cin >> a[i][j];
}

// Виведення матриці по рядках
void PrintMatrix(int a[n][m])
{
	for (unsigned i = 0; i < n; ++i)
	{
		for (unsigned j = 0; j < m; ++j) cout << '\t' << a[i][j];
		cout << '\n';
	}
}

// ВПОРЯДКУВАННЯ МАСИВУ a ПРОСТИМИ ВСТАВКАМИ
void SimpleInsertSort(int* a, unsigned n)
{
	// спочатку впорядкованим є лише перший елемент послідовності
	// переберемо всі інші і кожен з них вставимо на відповідне місце
	for (unsigned index_to_insert = 1; index_to_insert < n; ++index_to_insert)
	{
```

```
            // шукаємо місце для чергового елемента
            int b = a[index_to_insert];
            int comparison_place = index_to_insert - 1;
            while (comparison_place >= 0 && a[comparison_place] > b)
            {                               // посуваєм впорядковані елементи
                  a[comparison_place + 1] = a[comparison_place];
                  --comparison_place;
            }
            // вставляєм черговий елемент у впорядковану частину
            a[comparison_place + 1] = b;
      }
      return;
}

void SortEachRow()
{
      cout << "\n *Впорядкування за зростанням окремо кожного рядка матриці*\n\n";
      // Для спрощення розміри матриці задано в коді програми
      int a[n][m];
      // Вводимо задану матрицю
      ReadMatrix(a);
      // Аби виконати впорядкування, переберемо рядки і для кожного
      // викличемо процедуру сортування простими вставками
      for (unsigned row = 0; row < n; ++row)
      {
            SimpleInsertSort(a[row], m);
      }
      // Друкуємо результати
      PrintMatrix(a);
      return;
}
```

Задача 27. *Задано матрицю A:10×12. Впорядкувати рядки матриці за зростанням їхніх перших елементів.*

Щоб розв'язати цю задачу, необхідно переставити місцями рядки матриці так, щоб їхні перші елементи були впорядковані за зростанням. Тобто кожен рядок розглядають як єдиний запис, елемент структури даних, а його перший елемент – як його ключ. Переміщення рядків матриці є доволі затратною операцією, тому використаємо найекономніший щодо переміщень алгоритм – сортування вибором. Удосконалимо його так, щоб разом з переміщенням ключів відбувалось переставляння рядків.

Введення та виведення матриці у цій задачі нічим не відрізняється від описаного в попередній програмі, тому наведемо тільки той фрагмент програми, який відповідає власне за впорядкування:

```
// Обмін значень двох масивів (рядків матриці)
void SwapArray(int* a, int* b, unsigned n)
{
      for (unsigned i = 0; i < n; ++i)
      {
            int to_swap = a[i];
            a[i] = b[i];
            b[i] = to_swap;
      }
}
```

```
void SortMatrix()
{
	cout << "\n *Сортування рядків матриці за зростанням перших елементів*\n\n";
	// Для спрощення розміри матриці задамо в коді програми
	int a[n][m];
	// Вводимо задану матрицю
	ReadMatrix(a);
	// ВПОРЯДКУВАННЯ ПЕРШОГО СТОВПЦЯ МАТРИЦІ ЗА ДОПОМОГОЮ ВИБОРУ НАЙБІЛЬШОГО
	for (unsigned last_unsorted = n - 1; last_unsorted > 0; --last_unsorted)
	{
		// знаходимо найбільший елемент невпорядкованої частини
		unsigned index_of_max = 0;		// початковий номер найбільшого
		for (unsigned i = 1; i < last_unsorted; ++i) // перевіряємо всі решту
			if (a[i][0] > a[index_of_max][0]) index_of_max = i;
		if (index_of_max != last_unsorted)
			// міняємо місцями рядки: останній з найбільшим
			SwapArray(a[last_unsorted], a[index_of_max], m);
	}
	// Друкуємо результати
	PrintMatrix(a);
	return;
}
```

У цій програмі рядки матриці міняємо місцями за допомогою додаткової функції *exchange*, елементи першого стовпця змінюють місце разом зі своїм рядком.

Задача 28. *Задано матрицю A:10×12. Впорядкувати рядки матриці за зростанням сум модулів їхніх елементів.*

Ця задача, як і попередня, передбачає впорядкування рядків матриці за зростанням певних ключів, однак, на відміну від попередньої, тут ключі рядків наперед невідомі. Їх необхідно обчислити і зберігати під час сортування. Використаємо для зберігання ключів одновимірний масив *key*. Його довжина дорівнює кількості рядків матриці:

```
void SortMatrixSum()
{
	cout << "\n *Сортування рядків матриці за зростанням сум модулів*\n\n";
	// Для спрощення розміри матриці задамо в коді програми
	int a[n][m];
	// Вводимо задану матрицю
	ReadMatrix(a);
	// ФОРМУВАННЯ МАСИВУ КЛЮЧІВ
	int key[n] = {0};
	for (unsigned i = 0; i < n; ++i)
		for (unsigned j = 0; j < m; ++j) key[i] += abs(a[i][j]);
	// ВПОРЯДКУВАННЯ МАСИВУ КЛЮЧІВ & РЯДКІВ МАТРИЦІ
	for (unsigned last_unsorted = n - 1; last_unsorted > 0; --last_unsorted)
	{
		// знаходимо найбільший елемент невпорядкованої частини
		unsigned index_of_max = 0;			// початковий номер ключа
		for (unsigned i = 1; i < last_unsorted; ++i) // перевіряємо всі решту
			if (key[i] > key[index_of_max]) index_of_max = i;
		if (index_of_max != last_unsorted)
		{
			// міняємо місцями ключі і рядки
			int toSwap = key[last_unsorted];
```

```
                key[last_unsorted] = key[index_of_max];
                key[index_of_max] = toSwap;
                SwapArray(a[last_unsorted], a[index_of_max], m);
            }
        }
        // Друкуємо результати
        cout << "\nКлючі:\n";
        for (unsigned i = 0; i < n; ++i) cout << '\t' << key[i];
        cout << "\n\nМатриця:\n";
        printMatrix(a);
        return;
}
```

Впорядкування стовпців матриці можна виконувати схожим чином. Відповідні алгоритми будуть відрізнятися тільки індексами біля елементів матриці та способом виконання перестановок: для стовпців їх потрібно виконувати поелементно, як у п. 4.3, у програмі *MoveMax*.

6.2. Впорядкування файлу за допомогою списку

На відміну від масиву – структури даних з безпосереднім доступом до даних – файл є структурою з послідовним доступом. Тому сортування записів файлу суттєво відрізняється від сортування елементів масиву. Передусім тому, що виконати доступ до конкретного запису файлу є складніше і суттєво довше, ніж до елемента масиву. З описаних раніше алгоритмів хіба що впорядкування вибором можна було б адаптувати для сортування файлів. На практиці ж для цього використовують інші алгоритми. Про них ми поговоримо далі.

Вибір способу сортування записів файлу залежить також і від його розміру: якщо файл можна повністю завантажити в оперативну пам'ять, то його можна впорядкувати за допомогою лінійного списку, чи дерева. У протилежному випадку здебільшого використовують впорядкування злиттям.

Нагадаємо, які структури можна використати для побудови однозв'язного списку і двійкового дерева (контейнери бібліотеки STL використовувати не будемо). Елемент даних повинен містити ключ, за яким можна ідентифікувати дані. Зазвичай ключами є цілі числа. Інші складові елемента даних – поле, або декілька полів довільного типу. Не зменшуючи загальності можемо вважати, що воно рядкового типу, адже будь-яке значення можна зобразити рядком літер.

```
struct DataEntry     // елемент даних
{
        int key;
        string value;  // для значення можна вказати довільний тип
};
```

Для зручної роботи з такими даними доцільно оголосити оператори порівняння, введення з потоку та виведення в потік.

```
bool operator<(const DataEntry& left, const DataEntry& right)
{
        return left.key < right.key;
}
bool operator>(const DataEntry& left, const DataEntry& right)
{
        return left.key > right.key;
}
```

```
std::istream& operator>>(std::istream& is, DataEntry& e)
{
	is >> e.key >> e.value;
	return is;
}

std::ostream& operator<<(std::ostream& os, const DataEntry& e)
{
	os << e.key << '\t' << e.value;
	return os;
}
```

Ланка однозв'язного списку містить елемент даних і вказівник на наступну ланку.

```
struct ChainNode    // ланка списку
{
	ChainNode* link;
	DataEntry elem;
	ChainNode(): link(nullptr), elem() {}
	ChainNode(const DataEntry& e, ChainNode* p = nullptr): link(p), elem(e) {}
};
```

Задача 29. *Задано файл, кожен запис якого містить унікальний цілочисловий ключ. Впорядкувати записи файлу за зростанням ключів.*

Припустимо спочатку, що розмір файлу не перевищує розмір доступної динамічної пам'яті і всі його записи можна завантажити в деяку динамічну структуру даних, наприклад, в лінійний однонапрямлений список. Якщо в процесі завантаження ми збережемо початковий взаємний порядок розташування записів, то далі доведеться розв'язувати задачу сортування списку. Зручніше використати інший підхід: створити за даним файлом впорядкований список і тоді елементи списку записати назад у файл. Як створити впорядкований список? Для цього можна перший запис файлу одразу завантажити у першу ланку списку, а кожну нову ланку з наступним записом вставляти у список так, щоб не порушити впорядкування ключів – як в описаному раніше алгоритмі впорядкування простими вставками.

У п. 3.4 ми розглянули кілька варіантів алгоритму відшукання місця нового елемента у впорядкованій послідовності значень. З метою впорядкування масиву простими вставками ми використали пошук, що починав перегляд масиву з кінця і одночасно виконував порівняння й переміщення елементів (програма *SimpleInsert*). Для побудови впорядкованого списку такий спосіб пошуку місця не підійде: лінійний список можна переглядати тільки від першої ланки до останньої, а не у зворотному порядку. До того ж для нових ланок списку не потрібно звільняти місце, адже зв'язок між окремими ланками здійснюють за допомогою вказівників, яким байдуже, чи розташовано ланки в пам'яті підряд, чи ні. Тому доцільно виконувати пошук місця для нової ланки, як було описано програмою *ForwardInsert*.

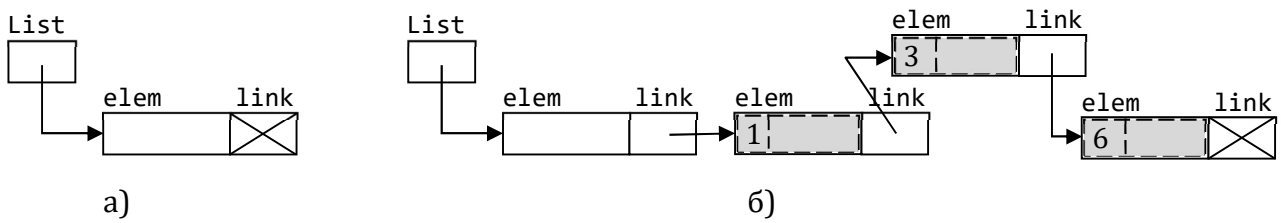

Рис. 4 Схематичний вигляд лінійного однозв'язного списку із заголовною ланкою: а) порожнього, б) з деякими даними (ключі 1, 3, 6)

Відомо, що перша ланка лінійного списку є «особлива»: вказівники на всі інші ланки містяться у полях зв'язку попередніх ланок, а вказівник на першу – в окремій змінній, що вказує на початок всього списку. Тому спосіб опрацювання першої ланки, зокрема, вставки першої ланки, відрізняється від способу опрацювання усіх інших. Щоб уникнути потреби реалізувати ці два способи опрацювання на практиці вставляють у список перед його першою ланкою ще одну, додаткову – *заголовну*. Єдине її завдання – містити вказівник на першу ланку списку. Витрату зайвої пам'яті у цьому випадку виправдано спрощенням алгоритму обробки списку. Саме такий підхід ми і використаємо.

Тепер можемо записати процедуру вставляння значення у впорядкований список.

```
void InsertNode(ChainNode* head, const DataEntry& e)
{
	ChainNode* current = head;
	// Пошук місця
	while (current ->link != nullptr && current->link->elem < e)
		current = current->link;
	// Вставляння
	current->link = new ChainNode(e, current->link);
}
```

Алгоритм впорядкування файлу за допомогою списку виглядатиме зовсім просто: послідовно прочитати записи файлу і вставити їх у список, записати елементи списку назад до файлу, вилучити список з динамічної пам'яті.

```
const char* FileName = "data.txt";
const char* NewFileName = "chaindata.txt";

void SortByChain()
{
	cout << "\n *Сортування файла за допомогою списку*\n\n";
	ifstream input_file(FileName);
	// Порожній список містить лише заголовну ланку
	ChainNode* chain = new ChainNode();
	while (!input_file.eof())
	{
		DataEntry e; input_file >> e;
		InsertNode(chain, e);
	}
	input_file.close();
	// Друкуємо результати
	PrintChain(chain->link, cout);   // На консоль
	ofstream result(NewFileName);
	PrintChain(chain->link, result); // і до файла
	result.close();
	DeleteChain(chain);
	// Очищуємо динамічну пам'ять
	return;
}

// Друк однозв'язного списку
void PrintChain(ChainNode* head, std::ostream& os)
{
	while (head != nullptr)
	{
		os << '\t' << head->elem << '\n';
```

```
            head = head->link;
      }
}

// Вилучення з пам'яті однозв'язного списку
void DeleteChain(ChainNode*& head)
{
      ChainNode* victim;
      while (head != nullptr)
      {
            victim = head;
            head = head->link;
            delete victim;
      }
      head = nullptr;
}
```

6.3. Впорядкування файлу бінарним деревом

Алгоритм впорядкування файлу лінійним списком використовує послідовний пошук місця елемента. Як зазначено у п. 3.4, ефективнішим є пошук місця за допомогою методу поділу відрізка навпіл (програма *BinaryInsert*). Виграш особливо відчутний для послідовностей великих розмірів, а файли, зазвичай, є досить великими.

З метою реалізації бінарного пошуку використовують двійкове дерево (визначення дерева та опис алгоритмів їхньої обробки див. у п. 10.3), кожна вершина якого містить унікальний ключ, причому для кожної вершини правильним є твердження про те, що ліве її піддерево містить лише вершини з меншими ключами, а праве – з більшими. Таке дерево називають *деревом пошуку*. Необхідний ключ можна знайти, скориставшись алгоритмом бінарного пошуку: якщо шуканий ключ менший за ключ у корені, то пошук продовжують у лівому піддереві; якщо більший, – то у правому; якщо рівний, то елемент знайдено; в іншому випадку знайдено місце для нового елемента, його необхідно долучити до дерева. Час виконання цього алгоритму є величиною порядку $O(\log_2 n)$, де n – кількість ключів у дереві.

Нагадаємо, що вершина дерева окрім елемента даних містить ще два вказівники: на ліве і праве піддерева.

```
struct TreeNode     // вершина дерева
{
      TreeNode* left_tree;
      TreeNode* right_tree;
      DataEntry elem;
      TreeNode(): left_tree(nullptr), right_tree(nullptr), elem() {}
      TreeNode(const DataEntry& e, TreeNode* pl = nullptr, TreeNode* pr = nullptr):
            left_tree(pl), right_tree(pr), elem(e) {}
};
```

З метою впорядкування файлу за допомогою дерева необхідно спочатку за даними, записаними у файлі, побудувати двійкове дерево пошуку, а тоді виконати *зворотній обхід* (див. п. 10.3) дерева і записати всі його елементи у файл. Пошук місця для нового елемента дерева та збереження дерева зручно реалізувати за допомогою рекурсивних процедур. За умовою задачі вихідний файл містить записи з попарно різними ключами, тому в процедурі пошуку не потрібно перевіряти рівність ключів:

```
const char* InputFileName = "data.txt";
```

```
const char* OutputFileName = "treedata.txt";

void SortByTree()
{
	cout << "\n *Сортування файлу за допомогою дерева пошуку*\n\n";
	ifstream input_file(InputFileName);
	TreeNode* tree = nullptr;
	while (!input_file.eof())
	{
		DataEntry e; input_file >> e;
		InsertTreeNode(tree, e);
	}
	input_file.close();
	// Друкуємо результати
	PrintTree(tree, cout);   // На екран
	ofstream result(OutputFileName);
	PrintTree(tree, result); // і до файла
	result.close();
	// Очищаємо динамічну пам'ять
	DeleteTree(tree);
	return;
}

// Вставляння значення у дерево пошуку
void InsertTreeNode(TreeNode*& root, const DataEntry& e)
{
	if (root == nullptr)
		root = new treeNode(e);
	else if (e < root->elem)
		InsertTreeNode(root->left_tree, e);
	else
		InsertTreeNode(root->right_tree, e);
}

// Виведення елементів дерева в потік (inorder обхід)
void PrintTree(TreeNode* root, std::ostream& os)
{
	if (root->left_tree != nullptr)
		PrintTree(root->left_tree, os);
	os << '\t' << root->elem << '\n';
	if (root->right_tree != nullptr)
		PrintTree(root->right_tree, os);
}

// Вилучення з пам'яті дерева пошуку
void DeleteTree(TreeNode*& root)
{
	if (root->left_tree != nullptr) DeleteTree(root->left_tree);
	if (root->right_tree != nullptr) DeleteTree(root->right_tree);
	delete root;
}
```

Обидва алгоритми впорядкування файлу, описані пп. 6.2, 6.3, можна легко пристосувати для впорядкування лінійних однонапрямлених списків. З цією метою необхідно просто замінити читання даних з файлу на отримання ланки списку, що підлягає сортуванню. У результаті виконання першого алгоритму (після його модифікації) за

заданим списком одразу отримаємо відсортований список. За другим алгоритмом буде побудовано дерево, і замість його збереження у файлі необхідно виконати побудову списку за деревом.

6.4. Злиття двох впорядкованих послідовностей

Як ми вже зазначали раніше, впорядкованість є цінною властивістю послідовності значень. Її варто зберігати під час доповнення послідовності новими членами. У п. 3.4 ми побудували декілька алгоритмів вставляння у впорядковану послідовність. Розв'яжемо тепер складнішу задачу: як об'єднати дві впорядковані послідовності в одну, також впорядковану? Алгоритм такого об'єднання називають алгоритмом *злиття*. Послідовність значень можна зберігати в різних структурах даних програми: масив, зв'язний список, файл. Усі вони схожі на рівні концепцій, оскільки підтримують послідовний доступ до своїх елементів, але відрізняються технічними деталями – як цей доступ отримати. Опишемо алгоритм злиття концептуально та реалізуємо його на прикладах злиття масивів і злиття списків, а в наступному параграфі використаємо для побудови алгоритму впорядкування файлу злиттями.

Задача 30. *Задано впорядковані за зростанням послідовності* $a_1, a_2, ..., a_n$ *та* $b_1, b_2, ..., b_m$. *Отримати нову впорядковану послідовність, складену зі значень заданих.*

Здавалося б, що тут вигадувати, якщо ми вміємо вставляти елемент у впорядковану послідовність? Адже так ми побудували раніше алгоритм впорядкування масиву простими вставками: брали по одному елементи невпорядкованої частини і вставляли у впорядковану, пошук місця вставки розпочинали з початку або з кінця впорядкованої частини. То ж і зараз вставимо по одному елементи послідовності b в a. Отриманий алгоритм матиме квадратичну складність $O(nm)$. Але такий підхід нехтує впорядкованістю послідовності b. Зрозуміло, що значення b_i в новій послідовності розташується *після* того місця, куди вставили b_{i-1}, тому пошук місця нової вставки потрібно продовжувати від місця попередньої, а не щоразу від початку масиву. Так можна отримати алгоритм лінійної складності $O(n+m)$ – ми успішно пристосуємо попередні рішення до нових потреб!

Виявляється, для побудови ефективного алгоритму злиття варто подивитися на задачу з дещо іншого боку. Чому ми вирішили вставляти елементи однієї послідовності в іншу? Чим одна краща від іншої? Нічим. Обидві послідовності рівноправні! Їхні початкові елементи конкурують за потрапляння до об'єднаної послідовності. Перемагає менший з них і переміщується або копіюється зі своєї послідовності в нову, на його місце стає наступний член тієї ж послідовності. Змагання припиниться, коли закінчиться одна з послідовностей – тоді залишиться дописати в результат «хвіст» іншої, що залишився.

Злиття можна виконувати двома способами: *копіювати* в нову послідовність значення заданих, або *переміщувати* їх туди, змінюючи структуру заданих послідовностей. Перший спосіб зручно застосовувати до масивів, що ми зараз і продемонструємо, а згодом застосуємо обидва для злиття однозв'язних списків.

Алгоритм не залежить від типу даних у послідовностях, аби лиш їх можна було порівнювати. Для простоти ми використаємо масиви цілих чисел a та b. Функція повертатиме новий збудований масив – об'єднану послідовність. Важливими є змінні-індекси елементів, що взаємодіють, цих трьох масивів. Індекси елементів заданих масивів назвемо *curr_a* та *curr_b* відповідно. Саме ці елементи конкуруватимуть за місце в новому масиві, а змінна *dest* позначатиме номер елемента для переможця.

Головний цикл алгоритму завершиться, коли перебере всі елементи одного з масивів. При цьому щонайменше один елемент іншого не буде скопійовано до результату. Тому потрібно додатково опрацювати залишкові елементи котрогось з масивів. Ми не мусимо з'ясовувати, котрого саме. Достатньо застосувати ітераційний цикл копіювання елементів

до кожного з заданих масивів. Інструкція *while* виконує перевірки не гірше за *if*, якщо опрацьовано всі елементи масиву, тіло циклу не буде виконано жодного разу.

```
// об'єднання двох впорядкованих масивів у новий
int* Merge(int* a, unsigned n, int* b, unsigned m)
{
	unsigned curr_a = 0;  // задані масиви переглядаємо з першого елемента
	unsigned curr_b = 0;
	unsigned dest = 0;    // результат заповнюємо також від початку
	int* c = new int[n + m]; // новий масив міститиме обидва задані
	while (curr_a < n && curr_b < m)
	{                         // конкурують елементи заданих масивів
		if (a[curr_a] < b[curr_b])     // «виграє» менший
			c[dest++] = a[curr_a++];
		else c[dest++] = b[curr_b++];
	}
	// дописування «хвоста», спрацює лише один з циклів
	while (curr_a < n) c[dest++] = a[curr_a++];
	while (curr_b < m) c[dest++] = b[curr_b++];
	return c;
}
```

Запис програми вийшов особливо компактним завдяки використанню постфіксного оператора ++. Вираз *a*[*curr_a*++] означає «звернутися до елемента масиву *a* за індексом *curr_a*, після чого збільшити *curr_a* на 1».

Злиття списків, що створює новий список-результат, дуже схоже до злиття масивів. Відмінність лише в способі доступу до елемента послідовності та переходу до наступного: *a*[*curr_a*] – у масиві, *a->elem* – у списку; *curr_a*++ – у масиві, *a* = *a->link* – у списку.

```
using List_t = ChainNode*;
// об'єднання двох впорядкованих списків у новий копіюванням елементів
List_t Merge(List_t a, List_t b)
{
	// задані списки переглядаємо з першого елемента, на них вказують a і b
	// допоміжна ланка, щоб перша ланка результату не була «особливою»
	ChainNode phantom;
	// результат будуємо поступово, по одній ланці, спочатку він містить
	List_t r = &phantom;     // лише заголовну ланку
	while (a != nullptr && b != nullptr)
	{                        ь    // конкурують елементи заданих масивів
		if (a->elem < b->elem)          // «виграє» менший
		{
			r->link = new ChainNode(a->elem);
			a = a->link;
		}
		else
		{
			r->link = new ChainNode(b->elem);
			b = b->link;
		}
		r = r->link;
	}
	// дописуємо "хвости"
	while (a != nullptr)
	{
```

```
            r->link = new ChainNode(a->elem);
            a = a->link;
            r = r->link;
        }
        while (b != nullptr)
        {
            r->link = new ChainNode(b->elem);
            b = b->link;
            r = r->link;
        }
        return phantom.link;
}
```

Попередні дві функції *копіюють* елементи заданих послідовностей в нову. Списки можна об'єднати інакше: *перемістити* ланки заданих послідовностей в одну нову. При цьому значення елементів не копіюють, а змінюють лише значення полів зв'язку.

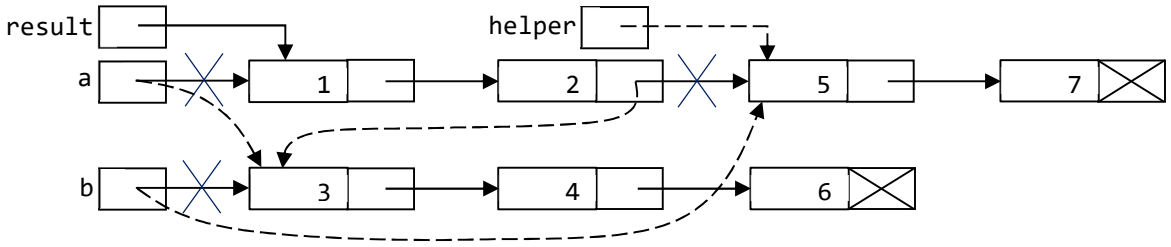

Рис. 5. Перебудова списків алгоритмом *MergeAndCut*

Сформулюємо алгоритм переміщення. На рис. 5 схематично зображено перший його етап. Список-результат починається з того списку, який у своїй першій ланці містить менше значення. Припустимо, що цей список – *a* (у протилежному випадку ми можемо обміняти місцями значення вказівників *a* і *b*). До результату можна зарахувати всі ланки *a*, що мають значення менше за першу ланку *b*. Вказівник *a* можна використовувати для пересування списком. Як тільки знайдено більшу ланку (на рис. 5 на неї вказує *helper*), потрібно виконати обмін вказівників: продовженням результату стає список *b*, а новим списком *b* – частина списку *a*, що починається зі знайденої ланки, вказівник *a* містить адресу першої доданої до результату ланки (колишнє значення *b*). Після такої перебудови ситуація повертається до початкової: *a* вказує на ланку з меншим значенням ніж *b*. Потрібно знову перевіряти, скільки ланок *a* можна долучити до результату. Цикл перевірок і перебудов завершиться, коли список *b* спорожніє. На рис. 5 зображено першу перебудову. Пунктиром зображено нові значення вказівників, перекреслено ті, які втратять чинність.

```
void MergeAndCut(List_t& a, List_t& b)
{
        List_t result; // результат починається з «меншого» списку
        if (a->elem < b->elem) result = a;
        else
        {
            result = b; b = a; a = result;
        }
        // b містить ланки, які ще не стали частиною результату
        while (b != nullptr)
        {
            // скільки ланок без зміни зв'язків потрапляють в результат?
            while (a->link != nullptr && a->link->elem < b->elem)
                a = a->link;
```

```
            // обмін: список b долучається до результату,
            // хвіст списку a стає списом b
            List_t helper = a->link;
            a->link = b;
            b = helper;
            a = a->link;
        }
        a = result;
}
```

Після завершення злиття перший параметр функції міститиме результат, а другий – порожній список.

6.5. Впорядкування файлу злиттям

Як впорядкувати файл, занадто великий, щоб повністю помістити його у динамічній пам'яті? Перше, що спадає на гадку, це впорядкувати файл хоча б частково. Наприклад, завантажувати в пам'ять і сортувати стільки записів файлу, скільки там поміститься. У результаті отримаємо файл, який містить впорядковані відрізки. Тепер, щоб завершити сортування, залишиться об'єднати ці відрізки в одну впорядковану послідовність. Об'єднання впорядкованих двох чи більше підпослідовностей в одну називають злиттям. Алгоритм сортування числового масиву за допомогою злиття вперше запропонував Джон фон Нейман, тому алгоритм впорядкування злиттям часто називають його іменем. Ми описали злиття двох масивів у попередньому параграфі.

Якщо послідовність містить більше ніж два впорядковані відрізки, то об'єднувати їх можна по-різному: послідовно, попарно, по три відрізки в один чи ще якось. На практиці використовують різні способи злиття, залежно від конкретних умов та оцінок ефективності цих способів. Ми використаємо один з найпростіших способів, об'єднуючи відрізки попарно. Проміжні результати, отримані на окремих кроках процесу сортування, зберігатимемо у допоміжних файлах.

Нехай a і b – файли, k – натуральне число. Говорять, що файли a і b узгоджено k-впорядковані, якщо:

1) у кожному з файлів a і b перші k записів, наступні за ними k записів і так далі утворюють упорядковані відрізки; останній відрізок файлу (також упорядкований) може містити менше ніж k записів, однак у цьому випадку тільки один з файлів може містити неповний останній відрізок;
2) кількість впорядкованих відрізків файлу a відрізняється від кількості впорядкованих відрізків файлу b не більше ніж на одиницю;
3) якщо файли містять різну кількість відрізків, то неповним може бути тільки останній відрізок довшого файлу.

Записи двох узгоджено k-впорядкованих файлів a і b можна розташувати у файлах f і g так, що f і g будуть узгоджено $2k$-впорядкованими. З цією метою пару впорядкованих відрізків з файлів a і b об'єднують у вдвічі довший впорядкований відрізок за алгоритмом злиття. Відрізки записів, отримані внаслідок злиття, почергово розташовують то у файлі f, то у файлі g. На рис. 6 показано, що відбувається під час перших двох злиттів.

Впорядкування файлу f методом *збалансованого двошляхового злиття* описує такий алгоритм (використовуються допоміжні файли g, a і b). Спочатку якимось способом розподіляють записи файлу f до файлів a і b, щоб утворити початкові впорядковані відрізки, наприклад, записи з парними номерами – до одного файлу, а з непарними – до іншого. Або розглядають пари записів файлу f і почергово записують їх у впорядкованому вигляді до файлів a та b, або утворюють початкові впорядковані відрізки за допомогою сортування бінарним деревом (на скільки це можливо за обсягом доступної динамічної пам'яті), чи іншим способом.

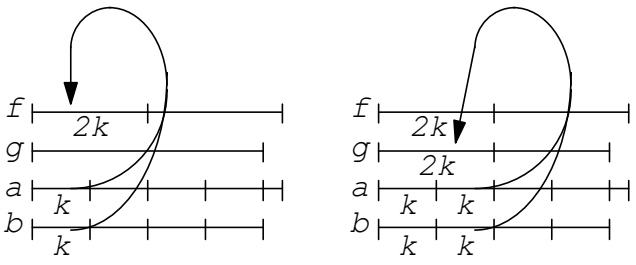

Рис. 6. Початковий етап сортування компонент файлу злиттям

Далі файли *a* і *b* розглядають як *k*-впорядковані (де *k* – розмір початкового відрізка), і утворюють з їхніх компонент 2*k*-впорядковані файли *f* і *g*. Далі з файлів *f* і *g* утворюють 4*k*-впорядковані *a* і *b* і т. д. Оскільки кількість впорядкованих відрізків у файлах зменшується після кожного злиття, то настане момент, коли усі записи буде зібрано в одному файлі у вигляді одного впорядкованого відрізка. На цьому сортування записів файлу буде завершено.

З метою опису алгоритму впорядкування файлу злиттям, необхідно дати відповіді на такі запитання: як утворити початковий розподіл відрізків; як виконувати злиття відрізків; як контролювати повторення та завершення процесу злиття?

Найпростіше отримати *початковий розподіл* записів файлу методом копіювання однієї половини з них до файлу *a*, а другої – до файлу *b*, проте це ніяк не наблизить нас до впорядкованості. Тому використаємо інший простий спосіб утворення файлів *a* і *b*: зчитуватимемо з файлу *f* по два записи і порівнюватимемо їхні ключі. Для пари записів дуже легко знайти більший ключ і записати їх у файл *a* чи *b* у правильному порядку й отримати таким чином впорядковані відрізки, довжина яких дорівнює двом. Якщо файл *f* містить непарну кількість записів, то останній відрізок буде неповним. Легко переконатися: якщо збереження впорядкованих відрізків почати з файлу *a*, то він завжди буде не меншим за файл *b*. Можливі такі варіанти: файли *a* та *b* містять однакову кількість початкових відрізків, у файлу *b* останній відрізок повний або ні; файл *a* містить на один відрізок (повний або неповний) більше. Алгоритм формування початкового розподілу опишемо у вигляді процедури *Distribute*.

Злиття. Файл – послідовність, тому алгоритм злиття файлів дуже схожий на злиття масивів і відрізнятиметься лише способом доступу до елементів даних: *a*[*curr_a*++] – для масиву, *file_a* >> *a* – для файлу (зауважимо, що читання елемента даних автоматично пересуває вказівник файлу на наступний елемент). Нагадаємо алгоритм злиття: прочитаємо по одному запису з відрізків файлів, порівняємо між собою їхні ключі і запишемо менший, а замість нього прочитаємо з відповідного відрізка наступний запис. Далі повторюємо порівняння, запис та зчитування аж до завершення одного з відрізків. Після цього копіюємо у результуючий непрочитаний залишок другого відрізка. Доведеться особливо контролювати завершення відрізка: це може статися тому, що прочитано зазначену кількість даних, або тому, що закінчився файл. Алгоритм опишемо за допомогою процедури *MergeFile*. Він повинен працювати з відрізками різних довжин, тому задаватимемо їх як параметри цієї процедури.

Впорядкування файлу розпочнемо одразу ж після побудови початкового розподілу записів вихідного файлу. Впорядкування полягає у багатократному злитті впорядкованих відрізків, що містяться в одній парі файлів, та збереженні результуючих відрізків у парі інших файлів. Початковий розподіл ми отримаємо у файлах *a* і *b*, тому вони є джерелами відрізків для першого злиття, а файли *f* і *g* – приймачами. Проте після першого об'єднання джерелом мусять стати *f* і *g*. Як це зробити, не дуже загромаджуючи алгоритм? Ми працюватимемо з текстовими файлами, а їх можна відкривати або для читання, або для запису. Тому після кожного злиття доведеться закривати та відкривати файли, щоб джерела стали приймачами і навпаки. Домовимося завжди приєднувати джерела до файлових потоків *a* і *b*, а приймачі – до файлових потоків *f* і *g*.

Щоб визначити, скільки відрізків необхідно об'єднати, порахуємо їхню кількість у кожному з файлів під час побудови початкового розподілу та змінюватимемо належним

чином під час виконання об'єднань. Процес впорядкування закінчиться, коли одна з кількостей відрізків дорівнюватиме нулю.

Особливої уваги потребує злиття останніх відрізків файлів *a* і *b*. Якщо файл *a* – довший, то його останній відрізок (незалежно від того, повний він чи ні) не має пари і його необхідно просто переписати у файл *f* чи *g*. Якщо ж файли *a* і *b* містять однакову кількість відрізків, то необхідно врахувати, що останній відрізок файлу *b* може бути неповним.

Тепер, здається, усі попередні пояснення наведені і можна записати саму програму. Окремі її кроки пояснені також у коментарях.

```
void Distribute(ifstream& source, ofstream& file_a, ofstream& file_b,
            long long& ka, long long& kb)
      // розподіляє записи файлу f до файлів a і b у відрізки по 2 записи
      // ka - кількість відрізків у файлі a, kb - у файлі b
{
      int x, y;         // елементи даних, отримані з файлу
      ka = 0; kb = 0;  // результуючі файли поки що порожні
      // цикл закінчимо процедурою break, коли досягнемо кінця файлу
      while (true)
      {
            // *** Спочатку записуємо до файлу a ***
            if (source.eof()) break;
            else
            {
                  source >> x; ++ka;
                  if (source.eof()) // прочитане число не має пари
                  {
                        file_a << ' ' << x;
                        break;
                  }
                  else
                  {
                        source >> y;
                        if (x < y) file_a << ' ' << x << ' ' << y;
                        else file_a << ' ' << y << ' ' << x;
                  }
            }
            // *** Тепер повторимо все для файлу b ***
            if (source.eof()) break;
            else
            {
                  source >> x; ++kb;
                  if (source.eof()) // прочитане число не має пари
                  {
                        file_b << ' ' << x;
                        break;
                  }
                  else
                  {
                        source >> y;
                        if (x < y) file_b << ' ' << x << ' ' << y;
                        else file_b << ' ' << y << ' ' << x;
                  }
            }
      }
}
```

```
void MergeFile(ifstream& file_a, ifstream& file_b, ofstream& result, long long k)
{
	// об'єднує відрізки довжини k з файлу file_a і з файлу file_b
	// і записує їх у файл result
	int a, b;		// елементи даних, отримані з файлів
	file_a >> a; file_b >> b;  // прочитали перші елементи відрізків
	long long count_a = 1;	// лічильники прочитаного
	long long count_b = 1;
	while (true)
	{
		if (a < b)
		{
			result << ' ' << a;
			if (count_a >= k || file_a.eof()) // файл a закінчився
			{
				result << ' ' << b;
				break;
			}
			else		// читаємо наступне значення з файлу a
			{
				file_a >> a; ++count_a;
			}
		}
		else
		{
			result << ' ' << b;
			if (count_b >= k || file_b.eof()) // файл b закінчився
			{
				result << ' ' << a;
				break;
			}
			else		// читаємо наступне значення з файлу b
			{
				file_b >> b; ++count_b;
			}
		}
	}
	while (count_a < k && !file_a.eof()) // дописуємо "хвости"
	{
		file_a >> a; ++count_a;
		result << ' ' << a;
	}
	while (count_b < k && !file_b.eof())
	{
		file_b >> b; ++count_b;
		result << ' ' << b;
	}
}

void SortFile(const char* file_name)
{
	// впорядкування файлу збалансованим двошляховим злиттям
	ifstream source(file_name);
	if (!source.is_open())
	{
		cout << "File " << file_name << " don\'t exists.\n";
```

```
        return;
    }

    // Будуємо початковий розподіл записів у тимчасових файлах
    ofstream file_f("_1.tmp");
    ofstream file_g("_2.tmp");
    long long ka, kb; // кількості відрізків у початковому розподілі
    Distribute(source, file_f, file_g, ka, kb);
    source.close();
    file_f.close();
    file_g.close();

    // приготуємося до виконання злиття
    ifstream file_a;
    ifstream file_b;
    long long segment_length = 2;  // початковий розмір впорядкованих відрізків
    bool odd = true;
    int a, b;          // елементи даних
    // Виконаємо злиття, поки відрізки є в обох файлах
    while (ka > 0 && kb > 0)
    {
        // джерела відкрили для читання приймачі - для запису
        if (odd)
        {
            file_a.open("_1.tmp"); file_b.open("_2.tmp");
            file_f.open(file_name); file_g.open("_3.tmp");
        }
        else
        {
            file_a.open(file_name); file_b.open("_3.tmp");
            file_f.open("_1.tmp"); file_g.open("_2.tmp");
        }
        // зливаємо відрізки почергово в різних приймачах
        for (long long segment_number=0; segment_number<kb; ++segment_number)
        {
            if (segment_number % 2 == 0)
                MergeFile(file_a, file_b, file_f, segment_length);
            else MergeFile(file_a, file_b, file_g, segment_length);
        }
        // дописуємо «хвіст»
        if (ka > kb) // ПЕРШИЙ ФАЙЛ ДОВШИЙ
        {
            if (kb % 2)
            {
                ka /= 2; kb = ka;
                while (!file_a.eof()) // копіюємо його залишок
                {
                    file_a >> a;
                    file_g << ' ' << a;
                }
            }
            else // усі пари вже об’єднано!
            {
                kb /= 2; ka = kb + 1;
                while (!file_a.eof()) // копіюємо залишок першого файлу
                {
```

```
                        file_a >> a;
                        file_f << ' ' << a;
                    }
                }
            }
            else // ФАЙЛИ МІСТЯТЬ ОДНАКОВУ К-СТЬ ВІДРІЗКІВ
            {
                if ((kb - 1) % 2)
                {
                    ka /= 2; kb = ka;
                }
                else
                {
                    kb /= 2; ka = kb + 1;
                }
            }
            segment_length *= 2;
            odd = !odd;
            file_f.close(); file_g.close();
            file_a.close(); file_b.close();
        }
        // Вилучимо тимчасові файли
        if (odd)
        {
            remove(file_name);
            rename("_1.tmp", file_name);
        }
        else remove("_1.tmp");
        remove("_2.tmp");
        remove("_3.tmp");
}
```

Зауважимо, що алгоритм побудовано так, що перший з пари файлів-приймачів завжди є не меншим від другого, тому результат сортування міститиметься у файлі, до якого приєднано потік *file_a*. Усі інші файли більше непотрібні, тому їх можна просто вилучити з диска. Оскільки в програмі багаторазово виконується почергове відкривання файлів, то остаточний результат може міститися у заданому файлі або в тимчасовому файлі *"_1.tmp"*, залежно від кількості виконаних злиттів. Якщо кількість непарна, перед закінченням програми потрібно перейменувати цей файл.

Для того щоб випробувати дію функції *SortFile()*, потрібно заповнити файл невпорядкованою послідовністю цілих чисел, надрукувати її на екрані, впорядкувати файл за допомогою *SortFile*, знову надрукувати його вміст.

```
void FillFile(ofstream& f, unsigned n)
{
        // заповнює файл випадковими цілими значеннями
        int x;
        srand(time(0));
        for (int i = 0; i < n; ++i)
        {
            x = rand() % (n * 2);
            f << ' ' << x;
        }
        f.close();
}
```

```
void ShowFile(const char* file_name)
{
	// виводить вміст файла на екран
	ifstream f(file_name);
	int x;
	while (!f.eof())
	{
		f >> x;
		cout << ' ' << x;
	}
	cout << '\n' << '\n';
	f.close();
}

void SortByMerge()
{
	cout << "\n *Зовнішнє сортування файлу парним злиттям*\n\n";
	const char * file_name = "mergeDat.txt";
	cout << "Розмір файлу? >>> ";
	int n; cin >> n;
	ofstream file(file_name);
	FillFile(file, n);
	ShowFile(file_name);
	SortFile(file_name);
	ShowFile(file_name);
	return;
}
```

Алгоритм впорядкування фон Неймана важко назвати простим, тому радимо читачам уважно розібрати всі його етапи та технічні деталі цих етапів. Функція *SortFile* впорядковує цілі числа, записані до *текстового* файлу. Її можна пристосувати також для опрацювання двійкового файлу. Перш за все зміни торкнуться способів читання та запису елементів даних. Замість інструкції *file_a >> a* потрібно записати *file_a.read((char*)&a, sizeof a)*, а замість *file_f << a – file_f.write((char*)&a, sizeof a)*. Щоб відкрити двійковий потік, вказують додатковий параметр: *ifstream file_f(file_name, iosbase::binary)*. Двійковий файл не переглянеш за допомогою редактора текстів, проте він має низку переваг: дані у ньому зберігаються так, як в пам'яті комп'ютера, без спотворень, читання та запис відбувається в рази швидше. Відкритий двійковий потік можна використовувати і для читання, і для запису файлу, тому відпадає потреба щоразу закривати і відкривати потік – достатньо просто встановити вказівник файлу на його початок, що також суттєво пришвидшує виконання.