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 файла.

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 сортуваня. 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 j = 1; j < n; ++j)
	{
		// шукаємо місце для j-го елемента
```

```
            int b = a[j];
            int i = j - 1;
            while (i >= 0 && a[i] > b)
            {                                   // посуваєм впорядковані елементи
                  a[i + 1] = a[i];
                  --i;
            }
            // вставляєм j-ий елемент у впорядковану частину
            a[i + 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 j = 0; j < n; ++j)
      {
            simpleInsertSort(a[j], 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 значань двох масивів (рядків матриці)
void exchange(int * a, int * b, unsigned n)
{
      for (unsigned i = 0; i < n; ++i)
      {
            int toSwap = a[i];
            a[i] = b[i];
            b[i] = toSwap;
      }
}
```

```
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 j = n; j > 1; --j)
	{
		// знаходимо найбільший елемент невпорядкованої частини
		unsigned k = 0;                  // початковий номер найбільшого
		for (unsigned i = 1; i < j; ++i) // перевіряємо всі решту
			if (a[i][0] > a[k][0]) k = i;
		// міняємо місцями рядки: j-й з k-им
		exchange(a[j - 1], a[k], 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 *b*. 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 j = n; j > 1; --j)
	{
		// знаходимо найбільший елемент невпорядкованої частини
		unsigned k = 0;                  // початковий номер ключа
		for (unsigned i = 1; i < j; ++i) // перевіряємо всі решту
			if (key[i] > key[k]) k = i;
		if (k != j)
		{
			// міняємо місцями ключі і рядки: j-й з k-им
			int toSwap = key[j - 1];
			key[j - 1] = key[k];
			key[k] = toSwap;
			exchange(a[j - 1], a[k], 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 файла за допомогою списку

На відміну від масиву – структури даних з безпосереднім доступом до даних – файл є структурою з послідовним доступом. Тому сортування записів файла суттєво відрізняється від сортування елементів масиву. Передусім тому, що виконати доступ до конкретного запису файла 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 файла 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 введення з потоку та виведення в потік.

```
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(0), elem() {}
      chainNode(const dataEntry& E, chainNode* L = 0): link(L), elem(E) {}
};
```

Вершина дерева окрім елемента даних містить ще два вказівники: на ліве і праве піддерева.

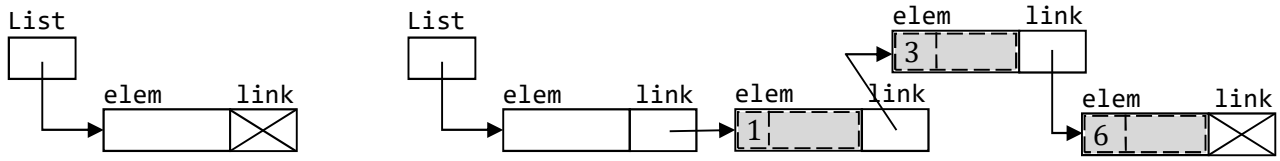
```
struct treeNode     // вершина дерева
{
      treeNode * left;
      treeNode * right;
      dataEntry elem;
      treeNode(): left(0), right(0), elem() {}
      treeNode(const dataEntry& E, treeNode* L = 0, treeNode* R = 0):
            left(L), right(R), elem(E) {}
};
```

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

Припустимо спочатку, що розмір файла 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 файла 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 зворотньому 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*.

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.



а) б)

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

Тепер можемо записати процедуру вставляння значення у впорядкований список.

```
void insertNode(chainNode* head, const dataEntry& E)
{
	chainNode * q = head;
	// Пошук місця
	while (q->link != nullptr && q->link->elem.key < E.key)
		q = q->link;
	// Вставляння
	q->link = new chainNode(E, q->link);
}
```

Алгоритм впорядкування файла за допомогою списку виглядатиме зовсім просто: послідовно прочитати записи файла і вставити їх у список, записати елементи списку назад до файла, 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 inFile(fileName);
	// Порожній список містить лише заголовну ланку
	chainNode * chain = new chainNode();
	while (!inFile.eof())
	{
		dataEntry E; inFile >> E;
		insertNode(chain, E);
	}
	inFile.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;
	}
}
```

6.3. Впорядкування файла бінарним деревом

Алгоритм впорядкування файла 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.

З метою впорядкування файла 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 * inFileName = "data.txt";
const char * outFileName = "treedata.txt";

void sortByTree()
{
	cout << "\n *Сортування файла за допомогою дерева пошуку*\n\n";
	ifstream inFile(inFileName);
	treeNode * tree = nullptr;
	while (!inFile.eof())
	{
		dataEntry E; inFile >> E;
		insertTreeNode(tree, E);
	}
	inFile.close();
```

```
	// Друкуємо результати
	printTree(tree, cout);   // На екран
	ofstream result(outFileName);
	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.key < root->elem.key)
		insertTreeNode(root->left, E);
	else
		insertTreeNode(root->right, E);
}

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

// Вилучення з пам'яті дерева пошуку
void deleteTree(treeNode*& root)
{
	if (root->right != nullptr) deleteTree(root->right);
	if (root->left != nullptr) deleteTree(root->left);
	delete root;
}
```

Обидва алгоритми впорядкування файла, описані в двох останніх параграфах, легко можна пристосувати для впорядкування лінійних однонапрямлених списків. З цією метою необхідно просто замінити читання даних з файла 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. Впорядкування файла 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 файла, 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.

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 файла (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 файла *a* відрізняється від кількості впорядкованих відрізків файла *b* не більше ніж на одиницю;
3) якщо файли містять різну кількість відрізків, то неповним може бути тільки останній відрізок довшого файла.

Записи двох узгоджено *k*-впорядкованих файлів *a* і *b* можна розташувати у файлах *f* і *g* так, що *f* і *g* будуть узгоджено 2*k*-впорядкованими. З цією метою пару впорядкованих відрізків з файлів *a* і *b* об'єднують у вдвічі довший впорядкований відрізок. Таке об'єднання називають *злиттям*. Відрізки записів, отримані внаслідок злиття, почергово розташовують то у файлі *f*, то у файлі *g*. На рис. 5 показано, що відбувається під час перших двох злиттів.

Впорядкування файла *f* методом *збалансованого двошляхового злиття* описує такий алгоритм (використовуються допоміжні файли *g*, *a* і *b*). Спочатку якимось способом розподіляють записи файла *f* до файлів *a* і *b*, щоб утворити початкові впорядковані відрізки, наприклад, записи з парними номерами – до одного файла, а з непарними – до іншого. Або розглядають пари записів файла *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.

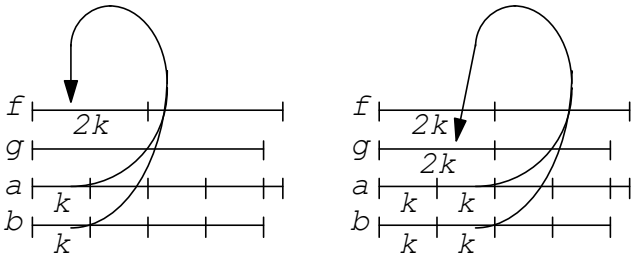


Рис. 5. Початковий етап сортування компонент файла злиттям

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 файла буде завершено.

З метою опису алгоритму впорядкування файла 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 файла методом копіювння однієї половини з них до файла *a*, а другої – до файла *b*, проте це ніяк не наблизить нас до впорядкованості. Тому використаємо інший простий спосіб утворення файлів *a* і *b*: зчитуватимемо з файла *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 файла *a*, то він завжди буде не меншим за файл *b*. Можливі такі варіанти: файли *a* та *b* містять однакову кількість початкових відрізків, у файла *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. З метою об'єднання двох впорядкованих відрізків в один довгий впорядкований використаємо такий алгоритм: прочитаємо по одному запису з цих відрізків, 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. Алгоритм злиття опишемо за допомогою процедури *merge*. Наш алгоритм повинен працювати з відрізками різних довжин, тому задаватимемо їх як параметри цієї процедури.

Впорядкування файла розпочнемо одразу ж після побудови початкового розподілу записів вихідного файла. 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? Для приєднання до файлів у програмі можемо використати динамічні файлові змінні і після кожного об'єднання міняти місцями значення вказівників на файли *a* та *f*, і на *b* та *g*. Отже файли *a* і *b* завжди будуть джерелами, а файли *f* і *g* – приймачами.

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 файла *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 & f, ofstream & a, ofstream & b, long long & ka, long
long & kb)
      // розподіляє записи файла 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 файла
      ka = 0; kb = 0;  // результуючі файли поки що порожні
      // цикл закінчимо процедурою break, коли досягнемо кінця файла
      while (true)
      {
            // *** Спочатку записуємо до файла a ***
            if (f.eof()) break;
            else
            {
                  f >> x; ++ka;
                  if (f.eof()) // прочитане число не має пари
                  {
                        a << ' ' << x; break;
```

```
                        //a.write((char*) &x, sizeof x); break;
                }
                else
                {
                        f >> y;
                        if (x < y) a << ' ' << x << ' ' << y;
                        else a << ' ' << y << ' ' << x;
                }
        }
        // *** Тепер повторимо все для файла b ***
        if (f.eof()) break;
        else
        {
                f >> x; ++kb;
                if (f.eof()) // прочитане число не має пари
                {
                        b << ' ' << x; break;
                }
                else
                {
                        f >> y;
                        if (x < y) b << ' ' << x << ' ' << y;
                        else b << ' ' << y << ' ' << x;
                }
        }
    }
}
void mergeFile(ifstream & a, ifstream & b, ofstream & c, long long k)
{
    // об'єднує відрізки довжини k з файла a і з файла b і записує їх у файл c
    //
    int x, y;          // елементи даних, отримані з файлів
    a >> x; b >> y;  // прочитали елементи перші відрізків
    long long i = 1; // лічильники прочитаного
    long long j = 1;
    while ( true )
    {
        if (x < y)
        {
            c << ' ' << x;
            if (i >= k || a.eof()) // файл a закінчився
            {
                c << ' ' << y; break;
            }
            else
            {
                a >> x; ++i;
            }
        }
        else
        {
            c << ' ' << y;
            if (j >= k || b.eof()) // файл b закінчився
            {
                c << ' ' << x; break;
            }
```

```
                else
                {
                        b >> y; ++j;
                }
        }
    }
    while (i < k && !a.eof()) // дописуємо "хвости"
    {
        a >> x; ++i;
        c << ' ' << x;
    }
    while (j < k && !b.eof())
    {
        b >> y; ++j;
        c << ' ' << y;
    }
}

void sortFile(const char * fileName)
{
    // впорядкування файла збалансованим двошляховим злиттям
    ifstream a(fileName);
    if (!a.is_open())
    {
        cout << "File " << fileName << " don\'t exists.\n";
        return;
    }
    ifstream b;
    // Будуємо початковий розподіл записів у тимчасових файлах
    ofstream u("_1.tmp");
    ofstream v("_2.tmp");
    long long ka, kb;
    distribute(a, u, v, ka, kb);
    a.close(); u.close(); v.close();
    long long k = 2; // початковий розмір впорядкованих відрізків
    bool odd = true;
    int x, y;
    // Виконаємо злиття, поки відрізки є в обох файлах
    while (ka > 0 && kb > 0)
    {
        // джерела відкрили для читання приймачі - для запису
        if (odd)
        {
            a.open("_1.tmp"); b.open("_2.tmp");
            u.open(fileName); v.open("_3.tmp");
        }
        else
        {
            a.open(fileName); b.open("_3.tmp");
            u.open("_1.tmp"); v.open("_2.tmp");
        }
        odd = !odd;
        for (long long kk = 0; kk < kb; ++kk)
        {
            if (kk % 2 == 0) mergeFile(a, b, u, k);
            else mergeFile(a, b, v, k);
```

```
            }
            if (ka > kb) // ФАЙЛ a - ДОВШИЙ
            {
                  if (kb % 2)
                  {
                        ka /= 2; kb = ka;
                        while (!a.eof()) // копіюємо залишок файла a
                        {
                              a >> x; v << ' ' << x;
                        }
                  }
                  else // усі пари вже об'єднано!
                  {
                        kb /= 2; ka = kb + 1;
                        while (!a.eof()) // копіюємо залишок файла a
                        {
                              a >> x; u << ' ' << x;
                        }
                  }
            }
            else // ФАЙЛИ МІСТЯТЬ ОДНАКОВУ К-СТЬ ВІДРІЗКІВ
            {
                  if ((kb - 1) % 2)
                  {
                        ka /= 2; kb = ka;
                  }
                  else
                  {
                        kb /= 2; ka = kb + 1;
                  }
            }
            k *= 2;
            a.close(); b.close(); u.close(); v.close();
      }
      // Вилучимо тимчасові файли
      if (odd)
      {
            remove(fileName);
            rename("_1.tmp", fileName);
      }
      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, на який вказує змінна *a*. Усі інші файли більше непотрібні, тому їх можна просто вилучити з диска. Оскільки в програмі багаторазово виконуються переприсвоєння значень вказівників на файлові змінні, то змінна *a* може вказувати на заданий файл або на тимчасовий файл *'_1.tmp'*. Щоб не виникало плутанини, перед закінченням програми виконано переіменування цього файла.

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 * fileName)
{
	// виводить вміст файла на екран
	ifstream f(fileName);
	int x;
	while (!f.eof())
	{
		f >> x;
		cout << ' ' << x;
	}
	cout << '\n' << '\n';
	f.close();
}

void sortByMerge()
{
	cout << "\n *Зовнішнє сортування файла парним злиттям*\n\n";
	const char * fileName = "mergeDat.txt";
	cout << "Розмір файла? >>> "; int n; cin >> n;
	ofstream file(fileName);
	fillFile(file, n);
	showFile(fileName);
	sortFile(fileName);
	showFile(fileName);
	return;
}
```