10. Рекурсія

Рекурсія – це спосіб визначення деякого поняття самого через себе. Класичним є приклад означення ідентифікатора, записаного формулами Наура-Бекуса:

```
<ідентифікатор>::=<буква>|<ідентифікатор><буква>|<ідентифікатор><цифра>.
```

Це означення складається з трьох альтернатив. Перша з них дає означення найпростішого ідентифікатора: ідентифікатора, що складається лише з однієї букви (літери латинського алфавіту). Дві інші альтернативи зазначають, що «довгий» ідентифікатор – це «коротший» ідентифікатор, до якого дописано літеру або цифру.

Рекурсію в програмуванні використовують як потужний засіб побудови алгоритмів. Рекурсивний розв'язок задачі завжди містить дві частини. Перша задає «елементарний» розв'язок – розв'язок задачі з найпростішими вхідними даними чи задачі найменшого розміру. Друга частина рекурсивного розв'язку описує, як отримати розв'язок задачі більшого розміру за допомогою розв'язку задачі меншого розміру (чи кількох задач менших розмірів). Дуже часто формулювання задач відразу містять і рекурсивний розв'язок (наприклад, задача обчислення чисел Фібоначчі).

Задача 42. *Числа Фібоначчі задано рекурентними співвідношеннями* $f_0 = f_1 = 0; f_n = f_{n-1} + f_{n-2}, n=2, 3, \ldots$ *Задано натуральне число n. Обчислити* f_n.

Тут розміром задачі є порядковий номер n числа. Найпростішим обчислення f_n є для $n = 0$ або для $n = 1$. Для більших значень n задача зводиться до таких самих задач розміру $n-1$ і $n-2$.

У мові програмування C++ будь-яка функція може бути рекурсивною (може викликати сама себе), тому рекурсивні розв'язки задач легко програмувати. Обчислення потрібного числа Фібоначчі можна виконати за допомогою такої функції:

```
long long HarmFibo(unsigned n)
{
      if (n < 2) return 1LL;                          // елементарний розв'язок
      else return HarmFibo(n - 1) + HarmFibo(n - 2); // зведення
}
```

Рекурсивні розв'язки можна будувати і в інших, не таких очевидних випадках.

Задача 43. *Задано натуральне число n. Обчислити n!.*

Відомо, що $n! = 1 \times 2 \times \ldots \times n$. Проте можна подати і рекурсивний розв'язок цієї задачі: $n! = 1$, якщо $n = 0$, або $n = 1$; $n! = n \times (n-1)!$, якщо $n > 1$. Тепер легко отримати:

```
long long Factorial(unsigned n)
{
      if (n < 2) return 1LL;             // елементарний розв'язок
      else return n * Factorial(n - 1); // зведення
}
```

Приваблива простота! Однак наведені приклади демонструють, як **не варто** застосовувати рекурсію. Обчислення факторіала можна швидше виконати за допомогою такого оператора циклу:

```
      long long f = 1LL;
      for (unsigned i = 2; i <= n; ++i) f *= i;
```

Зазначимо, що одна ітерація циклу виконується набагато швидше і потребує менше пам'яті, ніж один виклик підпрограми, як у функції *Factorial*.

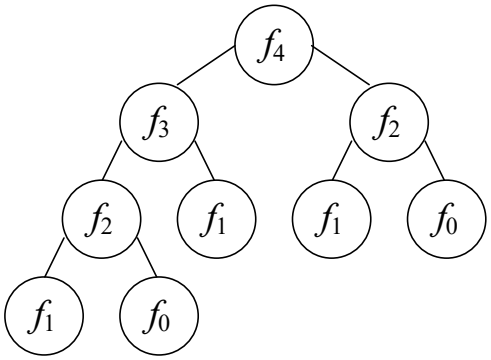

Рис. 8. Схема рекурсивних викликів функції *HarmFibo* під час обчислення f_4

З функцією *HarmFibo* ситуація ще гірша. Проаналізуємо, як відбуватиметься обчислення з її допомогою числа f_4. Чотири не менше, ніж два, тому *HarmFibo*(4) викличе *HarmFibo*(3) і *HarmFibo*(2). Три також не менше, ніж два, тому *HarmFibo*(3) викличе *HarmFibo*(2) і *HarmFibo*(1) і т. д. Увесь процес зображено на рис. 8. Видно, що існує багато повторних викликів функції з однаковими аргументами: *HarmFibo* (0) та *HarmFibo* (2) викликаються двічі, а *HarmFibo* (1) – навіть тричі. Зі збільшенням номера *n* кількість зайвих обчислень зростає експотенційно. Водночас у п. 2.3 було наведено приклад простого циклічного алгоритму, що використовує три змінні для обчислення чисел Фібоначчі. Безумовно, у цій задачі необхідно віддати перевагу саме циклічному алгоритмові.

Заради справедливості реабілітуємо рекурсивний спосіб отримання чисел Фібоначчі. У затратності *HarmFibo* винувата не стільки рекурсія, як її неправильне використання. У ній застосовано *комбінування* викликів, натомість потрібно застосувати *комбінування параметрів*. Його реалізують дві функції: рекурсивна *Fibo* виконує всю роботу, а «парадна» *Fibonacci* викликає її з правильними початковими значеннями параметрів.

```
// Прихована рекурсивна функція містить реалізацію
long long Fibo(long long a, long long b, unsigned n)
{
      if (n < 1) return a;               // елементарний розв'язок
      else return Fibo(b, a + b, n - 1); // зведення
}
// front end функція видима користувачеві
long long Fibonacci(unsigned n)
{
      return Fibo(1, 1, n);
}
```

Приклад функції *HarmFibo* наочно демонструє підступність рекурсії. Не можна застосовувати її бездумно тільки тому, що так буде зручно написати програму. Для кожного рекурсивного алгоритму існує нерекурсивний, який реалізує таку ж задачу, бо рекурсія – це лише спосіб запису алгоритму. Проте існує цілий клас задач, для яких рекурсія є відповідним способом отримання розв'язку. Кілька таких задач зараз розглянемо.

Практична порада: для того, щоб оволодіти рекурсією, варто задумуватися не стільки про те, як що працює, а про те, як описати розв'язок у термінах рекурсії.

10.1. Задача про Ханойські вежі

Пригадайте, як виглядає вежа буддійського храму: безліч трикутних куполів з припіднятими краями, побудованих один над одним, причому верхній купол завжди менший за нижній. Схоже на карпатську смереку. Задача завдячує своєю назвою саме виглядові цих споруд, а йтиметься в ній про трішки інші «вежі».

Задача 44. *На горизонтальній дошці закріплено три вертикальні стержні. На лівий стержень нанизано n дисків різного розміру, на більшому диску лежить менший*

(як зображено на рис. 9). Необхідно перемістити всі диски з лівого стержня на правий, не порушуючи їхнього впорядкування. Диски не можна класти збоку від стержнів, не можна класти диск більшого розміру на менший. Переносити з одного стержня на інший можна тільки по одному диску, середній стержень можна використовувати для тимчасового зберігання дисків.

Схоже, ми зібралися погратися в дитячі пірамідки. Проте розв'язок цієї задачі дитячим не назвеш (хіба що для n=1 чи n=2). Умову задачі можна переформулювати і в термінах інформатики: задано три стеки; перший з них містить послідовність різних цілих чисел, впорядкованих за спаданням; треба перемістити послідовність з першого стека в третій, використовуючи другий як робочий і ніколи не порушуючи впорядкованості чисел у кожному зі стеків.

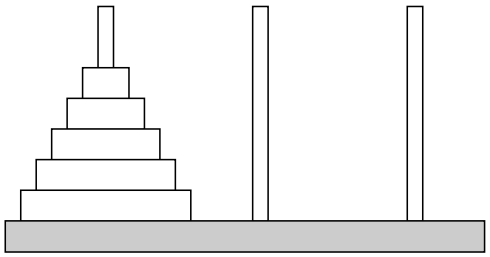

Рис. 9. Початкове розташування дисків у задачі про Ханойські вежі для n=5

Що є розв'язком цієї задачі? Очевидно, послідовність команд про переміщення дисків з одного стержня на інший. Наприклад, для n=2 вона матиме вигляд *«З лівого – на середній. З лівого – на правий. З середнього – на правий».* Як побудувати такий розв'язок? Помилково намагатися записувати розв'язки вручну для різних значень n. Такий підхід не наблизить нас до побудови рекурсивного розв'язку. Тут корисно застосувати дещо специфічний спосіб мислення.

Як велику задачу звести до меншої? Як отримати розв'язок великої задачі, коли відомо розв'язок меншої? Якби n–1 диск уже був на середньому стержні (див. рис. 10), то нам залишилось би перемістити останній диск з лівого стержня на правий і знову n–1 диск з середнього на правий. А переміщення n–1 диска і є тією задачею меншого розміру, яка нам так необхідна! Остаточно рекурсивний розв'язок задачі про Ханойські вежі можна сформулювати так: *якщо n=1, то перемістити диск зліва направо. Інакше перемістити n*–1 *диск з лівого стержня на середній, використовуючи правий стержень як робочий; перемістити n-й диск зліва направо; перемістити n*–1 *диск з середнього стержня на правий, використовуючи лівий стержень як робочий.* Позначимо лівий, середній, правий стержні літерами L, M, R, відповідно, і для зручності програмування запишемо цей розв'язок за такою схемою:

$$n: L \xrightarrow[M]{} R = \begin{cases} L \to R, & n = 1, \\ (n-1): L \xrightarrow[R]{} M,\ L \to R,\ (n-1): M \xrightarrow[L]{} R, & n \geq 2. \end{cases}$$

Тепер потрібно вирішити, як саме програма друкуватиме команди щодо переміщення дисків. Для позначення стержнів ми могли б використати значення деякого перелічуваного типу. Наприклад,

```
enum Position {Left, Middle, Right};
```

Для друкування назв стержнів доцільно використати відповідні рядки та оголосити окрему функцію для друку команди щодо переміщення одного диска.

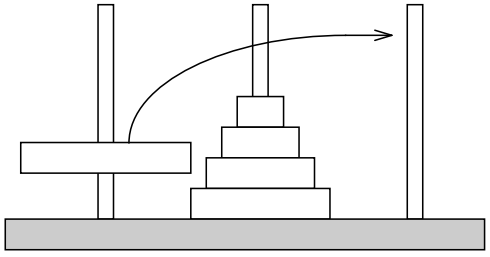

Рис. 10. Щоб перемістити останній диск, треба прибрати всі менші

```
void MoveDisk(Position from, Position to)
{
      static const char* name[3] = {" Left ", "Middle", " Right"};
      cout << name[from] << " --> " << name[to] << '\n';
}
```

Процедура переміщення вежі описує мовою C++ запропоновану вище схему.

```
// Будує послідовність команд про переміщення вежі з
// h дисків зі стержня from на стержень _to
void MoveTower(short h, position from, position to, position through)
{
      if (h == 1) MoveDisk(from, to);   // найменша задача
      else
      {                                  // зведення до двох задач розміру h-1
            MoveTower(h - 1, from, through, to);
            MoveDisk(from, to);
            MoveTower(h - 1, through, to, from);
      }
}
```

Остаточно програма розв'язування задачі про Ханойські вежі матиме вигляд:

```
void SolveTowers()
{
      cout << "* Розв'язування задачі про Ханойські вежі *\n\n";
      short h;
      cout << "Введіть висоту вежі: "; cin >> h;
      MoveTower(h, Left, Right, Middle);
}
```

Для $n = 4$ ця програма друкує такий розв'язок:

```
Введіть висоту вежі: 4
 Left   --> Middle
 Left   -->  Right
Middle  -->  Right
 Left   --> Middle
 Right  -->  Left
 Right  --> Middle
 Left   --> Middle
 Left   -->  Right
Middle  -->  Right
Middle  -->  Left
 Right  -->  Left
Middle  -->  Right
 Left   --> Middle
 Left   -->  Right
Middle  -->  Right
```

Без використання рекурсії отримати цей розв'язок було б значно складніше.

10.2. Алгоритм швидкого сортування

У 5 розділі ми розглянули кілька поширених алгоритмів упорядкування послідовності чисел. Опишемо ще один такий алгоритм. Його побудовано з використанням рекурсії.

Розглянемо таке перетворення масиву: нехай задано масив a_1, a_2, ..., a_n. Необхідно переставити його елементи так, щоб спочатку в масиві розташувати групу елементів, менших за початкове значення a_1, потім – саме це значення, а потім – групу більших елементів. Для цього треба виконати не більше ніж $n-1$ порівнянь і $n-1$ переміщень.

Використовуючи описане перетворення, можна побудувати рекурсивний алгоритм упорядкування масиву – *швидке впорядкування*. Якщо масив містить один елемент, то він

впорядкований. У протилежному випадку застосовуємо до нього описане перетворення і визначаємо результат упорядкування так: спочатку в масиві розташовано першу групу елементів, упорядкованих за допомогою алгоритму швидкого сортування; потім без змін той елемент, який відокремлював першу і другу групи елементів; далі – другу групу елементів, упорядкованих за допомогою алгоритму швидкого сортування. Цей алгоритм не використовує додаткового масиву і потребує в середньому приблизно $n \log_2 n$ порівнянь і стільки ж переміщень елементів. Проте це лише середнє значення: у найгіршому випадку кількість порівнянь досягатиме n (n–1)/2.

Давайте ще раз сформулюємо алгоритм швидкого сортування у термінах рекурсії. Масив з одного елемента впорядкований – це найпростіший випадок. Більший масив впорядковують так: перетворюють масив, щоб розмежувати менші за вибраний елемент і більші за нього, менші впорядковують алгоритмом швидкого сортування, більші впорядковують алгоритмом швидкого сортування. Розмір задачі кожного разу зменшується в середньому вдвічі. Використовують різні варіанти алгоритму швидкого сортування, що відрізняються передусім способом вибору елемента-дискримінанта, з яким порівнюють усі інші елементи масиву. Вибирають перший елемент, останній, центральний, або, навіть, випадковий.

З метою складення процедури швидкого сортування нам доведеться спочатку відповісти на кілька запитань. Перше з них: *як виконати однократне перетворення масиву?* Щоб відокремити дві групи елементів масиву – менших і більших за значення першого елемента – необхідно один раз перебрати елементи масиву, порівняти їх з цим значенням і виконати необхідні переміщення. Зауважимо, що у цьому випадку індекс елемента, в якому зберігатиметься перше значення, завжди буде меншим за індекс елемента, зі значенням якого його порівнюють. Для зберігання цих індексів використаємо змінні, відповідно, *i* та *j*. Отже, якщо черговий елемент більший за перше значення, то переходимо до наступного. У протилежному випадку групу елементів від *i*-го до *j*–1-го треба перемістити на одну позицію вперед, а *j*-й елемент – на місце, що звільнилося. Схожі зміни відбуваються з масивом під час сортування методом бульбашки. Тільки тут «підіймається» не один (перший) елемент, а ціла група елементів, значення яких не менші за значення першого. Такий варіант алгоритму називають *стабільним*: він зберігає початкове взаємне розташування елементів з рівними значеннями. Така особливість не надто корисна для сортування чисел, але може виявитися дуже важливою для сортування об’єктів, які вважаються рівними в сенсі оператора порівняння, а насправді є різними. Зауважимо також, що після завершення перегляду масиву з порівняннями та переміщеннями його перший елемент займе своє остаточне місце: ліворуч розташовано всі менші від нього, праворуч – більші, групи менших і більших залишаються невпорядкованими.

Сортування частин масиву можна виконати за допомогою рекурсивних викликів процедури сортування, однак у цьому випадку виникає друге запитання: *як передавати цій процедурі частини масиву?* У мові C++ масив передають парою параметрів: вказівник на початок масиву і кількість елементів у ньому. Вказівник може містити адресу довільного елемента масиву. Для функції масив розпочинатиметься саме з нього. Достатньо правильно налаштувати такий вказівник і задати правильну довжину частини масиву.

Нижче наведено текст функції *StableQuickSort* і оголошення типу *ExtInt*, який ми використаємо для демонстрації того, що вона справді реалізує стабільне сортування.

```
struct ExtInt
{
	// "розширене ціле" містить додатковий ідентифікатор-літеру
	int val;
	char ind;
};
```

```
ostream& operator<<(ostream& os, const extInt& x)
{
      os << x.val << '-' << x.ind;
      return os;
}
bool operator>(const extInt& a, const extInt& b)
{     // "розширені цілі" порівнюють за числом, нехтують літерою
      return a.val > b.val;
}

void StableQuickSort(extInt* a, int n)
{
      // перетворення масиву
      int pivot_index = 0;       // кінець відсортованої частини,
                                 // номер елемента-дискримінанта
      int begin_of_unsorted = 1; // початок невідсортованої частини
      while (begin_of_unsorted < n)
      {
            if (a[pivot_index] > a[begin_of_unsorted])
            {                // треба звільнити місце для меншого від дискримінанта
                  ExtInt to_transfer = a[begin_of_unsorted];
                  for (int k = begin_of_unsorted - 1; k >= pivot_index; --k)
                        a[k + 1] = a[k];
                  a[pivot_index] = to_transfer;
                  ++pivot_index;
            }
            // більший від дискримінанта залишається праворуч
            ++begin_of_unsorted;
      }
      // усі менші залишилися на початку, впорядковуємо їх
      if (pivot_index > 1) StableQuickSort(a, pivot_index);
      // усі більші – після дискримінанта, впорядковуємо їх також
      if (n - pivot_index > 1)
            StableQuickSort(a + pivot_index + 1, n - pivot_index - 1);
}
```

Наведемо ще один варіант алгоритму швидкого сортування. Як дискримінант він використовує значення елемента, розташованого посередині масиву, і виконує набагато менше обмінів елементів, жертвуючи у цьому випадку початковим розташуванням елементів. За один прохід масиву алгоритм знаходить межі двох груп елементів: менших і більших від дискримінанта. Спочатку вважаємо, що межами цих груп є весь масив, потрібно посунути праву межу менших ліворуч, а ліву межу більших – праворуч. Для цього виконують послідовні перевірки елементів з обох країв масиву: якщо елемент на початку масиву менший від дискримінанта, то залишаємо його на місці, у протилежному випадку міняємо його місцями з першим знайденим меншим від дискримінанта наприкінці масиву. Кожна така перевірка стискає межі груп елементів. Процес завершується, коли права межа групи менших стає меншою за ліву межу групи більших. Далі рекурсивно впорядковують знайдені групи.

```
void QuickSort(ExtInt* a, int low, int high)
{
      // менші розташовано на проміжку [low; end_of_small]
      int end_of_small = high;       // права межа групи менших
      // більші розташовано на проміжку [begin_of_large; high]
      int begin_of_large = low;      // ліва межа групи більших
      ExtInt pivot = a[(low + high) / 2]; // значення дискримінанта
```

```
	while (begin_of_large <= end_of_small)
	{
		// знайдемо "неправильні": великі серед менших
		while (pivot > a[begin_of_large]) ++begin_of_large;
		//                          та малі серед більших
		while (a[end_of_small] > pivot) --end_of_small;
		if (begin_of_large < end_of_small)
		// якщо "неправильні" не в своїх групах, то їх треба обміняти місцями
		{
			ExtInt to_swap = a[begin_of_large];
			a[begin_of_large] = a[end_of_small];
			a[end_of_small] = to_swap;
			++ begin_of_large;
			--end_of_small;
		}
		else if (begin_of_large == end_of_small)
		// пошук завершився на дискримінанті
		{
			++begin_of_large;
			--end_of_small;
		}
	}
	if (end_of_small > low)       // впорядковуємо менші
		QuickSort(a, low, end_of_small);
	if (begin_of_large < high)    // та більші
		QuickSort(a, begin_of_large, high);
}
```

Випробуємо обидві функції швидкого сортування у невеликій програмі.

```
void PrintArr(ExtInt*a, int n)
{
	for (int i = 0; i < n; ++i)
	{
		cout.width(3); cout << a[i];
	}
	cout << '\n';
}

void UseQuickSort()
{
	ExtInt a[] = { {5,'a'},{3,'a'},{9,'a'},{2,'a'},{1,'a'},{2,'b'},{6,'a'},
	  {5,'b'},{8,'a'},{3,'b'},{4,'a'},{7,'a'},{2,'c'},{8,'b'},{5,'c'},{5,'d'} };
	ExtInt b[] = { {5,'a'},{3,'a'},{9,'a'},{2,'a'},{1,'a'},{2,'b'},{6,'a'},
	  {5,'b'},{8,'a'},{3,'b'},{4,'a'},{7,'a'},{2,'c'},{8,'b'},{5,'c'},{5,'d'} };
	const int n = sizeof a / sizeof a[0];
	cout << "Початковий масив\n";
	PrintArr(a, n);
	cout << "Результати швидкого стабільного сортування\n";
	StableQuickSort(a, n);
	PrintArr(a, n);
	QuickSort(b, 0, n - 1);
	cout << "Результати швидкого сортування\n";
	PrintArr(b, n);
}
```

Отримані результати демонструють, як різні функції по-різному розташували «рівні» елементи.

```
Початковий масив
  5-a  3-a  9-a  2-a  1-a  2-b  6-a  5-b  8-a  3-b  4-a  7-a  2-c  8-b  5-c  5-d
Результати швидкого стабільного сортування
  1-a  2-a  2-b  2-c  3-a  3-b  4-a  5-a  5-b  5-c  5-d  6-a  7-a  8-a  8-b  9-a
Результати швидкого сортування
  1-a  2-b  2-c  2-a  3-a  3-b  4-a  5-c  5-d  5-b  5-a  6-a  7-a  8-a  8-b  9-a
```

Упорядкування частини масиву за алгоритмом швидкого сортування подібне до сортування цілого масиву. Цю особливість алгоритму природно визначати у термінах рекурсії, що ми і продемонстрували на прикладі наведених програм.

10.3. Обхід двійкового дерева

Рекурсивність алгоритму часто буває зумовлена особливостями структури даних, для опрацювання якої його розроблено. Наприклад, алгоритми обходу двійкових дерев.

Деревовидною структурою (*деревом*) називають множину взаємозв'язаних об'єктів, розташованих за рівнями за таким правилом:

- на початковому рівні – один вузол – *корінь* дерева;
- будь-який вузол X наступного, i-го ($i \neq 0$) рівня пов'язаний лише з одним вузлом Y попереднього, (i–1)-го рівня.

У такому випадку Y називають безпосереднім предком вузла X, а X – безпосереднім нащадком Y. Якщо вузол немає нащадків, то його називають *листком*. Усі вузли, крім кореневого, які мають нащадків, називають внутрішніми вузлами, або *вершинами*.

Максимальну кількість безпосередніх нащадків того самого предка називають *степенем дерева*. Дерева степеня два називають *двійковими (бінарними) деревами*. Щоб реалізувати мовою C++ бінарне дерево, використовують такі оголошення:

```
//попереднє оголошення структури потрібне для оголошення вказівника
struct TreeNode;
typedef TreeNode* Tree_t; // вказівнику даємо ім'я задля зручності
typedef int DataType_t;   // тип інформаційної частини вершини може бути довільним

struct TreeNode
{
      DataType_t val;
      Tree_t left;
      Tree_t right;
      TreeNode() : val(0), left(nullptr), right(nullptr) {}
      TreeNode(DataType_t x, Tree_t l, Tree_t r) : val(x), left(l), right(r) {}
};
```

Дерево за своєю природою є рекурсивною структурою даних. Адже його означення можна сформулювати так: дерево з базовим типом *TreeNode* – це або порожнє дерево, або деяка вершина типу *TreeNode* зі скінченною кількістю зв'язаних з нею окремих дерев з базовим типом *TreeNode*, які називають піддеревами. Тому для обходу дерева використовують рекурсивні алгоритми. Під час обходу алгоритм має опрацювати всі вершини дерева, кожну – один раз. Дерево обходять, щоб відшукати певний елемент, роздрукувати всі елементи тощо. Існують різні типи обходу (наприклад, лівосторонній, правосторонній, за рівнями).

Під час лівостороннього обходу першими опрацьовують елементи лівого піддерева, а потім – правого. Існує три способи лівостороннього обходу, що відрізняються порядком опрацювання кореня: *прямий* (preorder), за яким спочатку опрацьовують корінь, далі – рекурсивно ліве піддерево, а наприкінці – праве; *обернений* (inorder), за яким спочатку рекурсивно опрацьовують ліве піддерево, потім – корінь, а наприкінці – праве; *кінцевий* (postorder), за яким спочатку опрацьовують ліве та праве піддерева а наприкінці – корінь. Для багатьох задач вибір способу обходу несуттєвий. Наприклад, для обчислення суми можемо використати будь-який з них. Почнемо з *прямого лівостороннього* обходу. Його алгоритм можна сформулювати так: якщо дерево непорожнє, то треба переглянути корінь, обійти ліве піддерево, обійти праве піддерево.

Задача 45. *Задано двійкове дерево, елементами якого є цілі числа. Написати функцію обчислення суми елементів цього дерева.*

Щоб обчислити суму елементів послідовності, потрібно перебрати її елементи від першого до останнього. Щоб обчислити суму елементів дерева, також потрібно їх перебрати. Але в якому порядку це робити? Припустимо, почнемо з кореня, куди тоді рухатися: ліворуч чи праворуч? Якщо перейдемо за вказівником на ліве піддерево, то як не забути про вказівник на праве? Давайте не будемо турбуватися про деталі обходу дерева, натомість сформулюємо розв'язок у термінах рекурсії: сума елементів порожнього дерева дорівнює нулю, а непорожнього – це сума трьох значень: елемента, записаного в корені дерева, суми елементів лівого піддерева і суми елементів правого піддерева. Такий розв'язок легко записати рекурсивною функцією, а про обхід нехай турбується механізм викликів.

```
DataType_t PreorderSum(Tree_t t)
{
	if (t == nullptr) return 0;
	else
		return t->val + PreorderSum(t->left) + PreorderSum(t->right);
}
```

Обхід дерева ця функція виконує за допомогою рекурсивних викликів, а опрацювання елемента полягає у додаванні його до суми. Кількість рекурсивних викликів у тілі функції можна зменшити, якщо виконувати додаткові перевірки:

```
DataType_t SumEco(Tree_t t)
{
	if (t == nullptr) return 0;
	else
	{
		DataType_t s = t->val;
		if (t->left != nullptr) s += SumEco(t->left);
		if (t->right != nullptr) s += SumEco(t->right);
		return s;
	}
}
```

Тут рекурсивні виклики виконують лише для непорожніх піддерев, однак є певна надлишковість у перевірках, бо кожен екземпляр функції *SumEco* перевіряє, чи непорожнє передане йому дерево. Така перевірка доцільна тільки тоді, коли *SumEco* викликають вперше. Припустимо, що перший виклик функції завжди виконують тільки для непорожніх дерев. Запишемо:

```
DataType_t SumNotEmpty(Tree_t t)
{
	DataType_t s = t->val;
	if (t->left != nullptr) s += SumNotEmpty(t->left);
	if (t->right != nullptr) s += SumNotEmpty(t->right);
	return s;
}
```

Ми щойно сказали, що перевірка, чи дерево порожнє, важлива під час першого виклику функції обчислення суми. То давайте «заховаємо» її у ще одну функцію, яка й буде найзручнішою на практиці.

```
dataType_t Sum(Tree_t t)
{
	return (t == nullptr) ? 0 : SumNotEmpty(t);
}
```

Для обчислення суми елементів дерева у кожній з функцій ми використали прямий обхід. Зрозуміло, що в цьому випадку вибір способу обходу ніяк не впливає на результат, проте для виконання перевірок зручніше починати з кореня.

Наведемо приклад програми, що виконує зворотний лівосторонній обхід дерева.

Задача 46. *Описати процедуру, що друкує елементи заданого двійкового дерева, відображаючи його структуру за допомогою відступів, у такому порядку: спочатку друкується ліве піддерево з відступом на одну позицію, потім – корінь з початку рядка, потім – праве піддерево також з відступом на одну позицію.*

Структура процедури виведення дерева може бути схожою до структури функції *SumEco*: для обходу використаємо оператори виклику процедури (рекурсивного), а опрацювання елемента полягатиме у виведенні його на друк після відповідної кількості пропусків. Для задання кількості пропусків використаємо додатковий параметр. На екрані дерево «ростиме» зліва направо (а не згори донизу, як звично зображають дерева на папері). Вершини одного рівня перебуватимуть на однаковій відстані від лівого краю екрана.

```
// Друкує дерево з відображенням структури
void PrintTree(const Tree_t t, unsigned shift)
{
	// друк елементів дерева по одному в рядку з відступами,
	// що моделюють структуру дерева
	if (t != nullptr)
	{
		PrintTree(t->left, shift + 1);  // друк лівого піддерева
		// Друк кореня двоетапний:
		for (unsigned i = 0; i < shift; ++i)
			cout << '\t';                    // спочатку - відступи
		std::cout << t->val << '\n';         // потім - значення
		PrintTree(t->right, shift + 1); // друк правого піддерева
	}
}
```

Таку структуру процедури можна використати і для інших задач, що потребують виконання обходу. Зміниться лише та частина, яка виконує власне обробку елемента.

Випробувати написані функції допоможе нескладна програма. Побудову дерева задано за допомогою виклику конструкторів. Суму елементів обчислено різними функціями.

```
void TraverseTree()
{
      Tree_t t = new treeNode(1, new treeNode(2, new treeNode(4, nullptr, nullptr),
            new treeNode(5, nullptr, nullptr)),
            new treeNode(3, nullptr, new treeNode(6, nullptr, nullptr)));
      PrintTree(t, 0);
      cout << " s = " << PreorderSum(t) << '\n';
      cout << " S = " << Sum(t) << '\n';
}
```

Отримали:

```
            4
      2
            5
1
      3
            6
 s = 21
 S = 21
```

Звичайно, залишилося запитання «як все це працює?» Щоб простежити кожен крок алгоритму, можна використати налагоджувач середовища програмування. Пропонуємо також читачеві самостійно зобразити стек рекурсивних викликів функції *PreorderSum* або *PrintTree*. Постарайтеся відобразити *кожен* виклик функції і стан її локальних змінних.

Зауважимо також, що в параграфі 6.3 ми вже використовували бінарне дерево та виконували зворотний лівосторонній обхід для впорядкування файлу.

10.4. Запитання та завдання для самоперевірки

1. Що таке рекурсія в математиці? В програмуванні? Які рекурсивні визначення використано в попередніх розділах цього посібника?
2. Чим саме шкідлива рекурсія у виконанні функції *HarmFibo*?
3. Як правильно збудувати рекурсивну функцію обчислення послідовності Фібоначчі?
4. Поясніть наведений вище символьний запис розв'язку задачі про Ханойські вежі.
5. Припустимо, у вас є три стеки для цілих чисел, дані в яких можна зберігати лише у впорядкованому за зростанням вигляді: у вершині стеку – найменше значення. Один зі стеків містить *n* чисел. Побудуйте послідовність команд для переміщення цих чисел до другого стеку. Третій стек можна використовувати для тимчасового зберігання.
6. Чим відрізняється алгоритм *QuickSort* від *StableQuickSort*? Який з них швидший?
7. Які є різновиди обходів двійкового дерева? Як їх реалізують?
8. Як збудувати функцію виведення дерева на екран, що відображає його структуру?
9. Поміркуйте, як написати *нерекурсивну* функцію, що виконує обхід дерева.
10. Завантажте програми за наведеним посиланням, запустіть їх на виконання.
11. Запропонуйте та випробуйте власні зміни та доповнення до програм.