Лекція 10. Дерева

Сортування списком (стосується попередньої лекції).
Структура даних дерево. Означення.
Поняття рекурсії. Рекурсивні розв'язки, рекурсивні функції.
Алгоритми обходу двійкового дерева.
Дерева пошуку. Сортування деревом.
**Збалансовані дерева, вставка, вилучення.*

Сортування списком

У попередній лекції ми обговорювали можливості використання зв'язної структури список для гнучкого зберігання послідовності значень. Зокрема, ми оголосили процедуру вставляння значення до впорядкованого списку і використали її для оголошення процедури впорядкування масиву чисел.

```
void sortByList(int * A, size_t n)
{
	// щоб упорядкувати масив A, помістимо всі його елементи у впорядкований список
	List_t List = nullptr;
	for (size_t i = 0; i < n; ++i) insert(A[i], List);

	// а тоді повернемо зі списку назад в масив
	for (size_t i = 0; i < n; ++i, curr = curr->link) A[i] = delFirst(L);
}
```

Алгоритм цілком добрий, але має один суттєвий недолік – забагато порівнянь. Справді, в найгіршому випадку йому доведеться виконати одне порівняння для першого елемента масиву, два – для другого, …, n – для останнього. Загалом $\left(n^2+n\right)/2$ порівнянь. Щоб уникнути цього недоліку, познайомимося ще з однією зв'язною структурою даних.

Структура даних «дерево»

У попередній лекції (і в проектах) описано способи реалізації та опрацювання лінійних однозв'язних списків. Зупинимось тепер на розгляді ієрархічних списків – ще однієї структури даних, характерної для багатьох класів задач – деревовидної.

Означення дерева

Деревовидною структурою (деревом) називають множину взаємопов'язаних об'єктів (які називають також вершинами або вузлами), розташованих по рівнях за таким правилом:

- на першому рівні – один вузол так званий *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 потомків, то він називається листком. Всі вузли, крім кореневого, які мають потомків, називаються внутрішніми вузлами, або вершинами.

Дерева застосовують для представлення об'єктів, які мають ієрархічну внутрішню будову. Наприклад, адміністративну структуру університету можна зобразити у вигляді дерева, коренем якого є ректор, а листками – зокрема, студенти; математичну формулу – деревом, листками якого є операнди, а вершинами – знаки операцій і так далі.

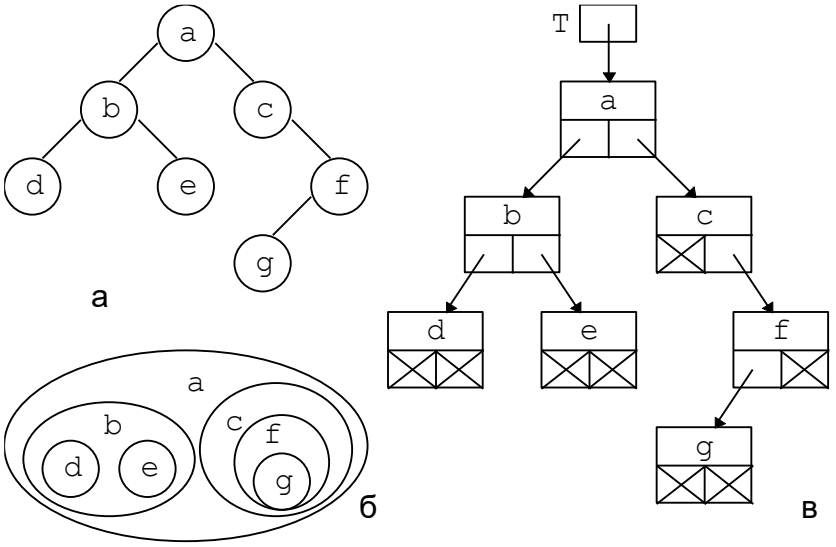


Рис. 1. Способи зображення дерева

Є різні способи зображення дерев: за допомогою графа (рис. 1, а), вкладених множин (рис. 1, б) тощо. Ми інтерпретуватимемо дерева як списки з ланками, що можуть мати по декілька наступників (рис. 1, в). Розглянуті раніше однонапрямлені списки є, по суті, "виродженими" деревами, вершини якого мають тільки одного наступника.

Максимальна кількість безпосередніх нащадків того самого предка називається *степенем дерева*. Наприклад, на рис. 1 зображено дерево степеня два. Такі дерева називають *двійковими (бінарними) деревами*.

Усі вузли дерева, пов'язані з вершиною *X* безпосередньо або через інші вузли, називаються її потомками. Довільна вершина *X* дерева разом з усіма своїми потомками називається піддеревом цього дерева.

Як видно з рис. 1, в, кожна вершина дерева складається з полів зв'язку з потомками та інформативної частини, яка, залежно від потреби, містить інформацію довільного типу. Інформативну частину вершини дерева називатимемо *елементом* дерева.

Над деревом виконують такі основні операції:

- обхід дерева;
- вставка нової вершини (піддерева);
- вилучення вершини (піддерева).

Найчастіше виконують обхід дерева. Під час обходу алгоритм повинен опрацювати всі вершини дерева, кожну – один раз. Дерево обходять, щоб відшукати певний елемент, роздрукувати всі елементи тощо. Є різні способи обходу, наприклад, лівосторонній, правосторонній, за рівнями. Під час лівостороннього обходу першим опрацьовують крайній лівий листок, а останнім – крайній правий.

Рекурсія

Рекурсія – це метод визначення поняття, функції, розв'язку задачі через те саме поняття, функцію, розв'язок.

Дерево за своєю природою є рекурсивною структурою даних. Адже його означення можна сформулювати так: дерево – це

1) або порожнє дерево,
2) або деяка вершина, корінь, зі скінченним числом пов'язаних з нею окремих дерев, які називаються піддеревами.

Ніклаус Вірт зазначає, що кожній структурі даних відповідає певний алгоритм обробки і відповідна структура керування, яка його реалізує. Масиву відповідає цикл з параметром, списку (а також файлу) – ітераційний цикл. Деревам відповідають рекурсивні алгоритми.

Рекурсивний розв'язок задачі з розміром вхідних даних *n* складається принаймні з двох частин:

1) розв'язок задачі найменшого розміру n_0;
2) визначення шуканого розв'язку через розв'язок задачі з меншим розміром вхідних даних $n-1$, $n-2$ тощо.

або

1) розв'язок тривіального випадку;
2) розв'язок через формулювання такої ж, але простішої задачі (задач).

У рекурсивному алгоритмі перше задає стоп-умову, а друге – спосіб заглиблення в рекурсію.

Проілюструємо сказане кількома прикладами.

1. Факторіал: $n! = 1$, якщо $n = 1$; $n! = n \times (n-1)!$, якщо $n > 1$.
2. Найбільший член заданої послідовності можна знайти за таким правилом:
   $\max(x_1, x_2, \ldots, x_n) = $ більше_з_чисел$(x_1, \max(x_2, \ldots, x_n))$, якщо $n > 2$;
   $\max(x_1, x_2) = $більше_з_чисел$(x_1, x_2)$.
3. Числа Фібоначчі: $f_0 = f_1 = 1$; $f_n = f_{n-1} + f_{n-2}$, $n = 3, 4, \ldots$.

Рекурсію не варто застосовувати, якщо завдання можна виконати за допомогою циклу: ітеративні алгоритми ефективніші за рекурсивні. Проте дуже корисним є механізм рекурсії під час роботи з деревовидними структурами. Адже <*переглянути двійкове дерево*> означає <*переглянути ліве піддерево*>, <*переглянути вершину*>, <*переглянути праве піддерево*>. Використання рекурсії дає змогу програмі "пам'ятати" про всі ще не переглянуті піддерева. Зауважимо, що листок піддерев немає, і для нього перегляд зводиться до <*переглянути_вершину*> (розв'язок найпростішої задачі).

Рекурсивні алгоритми легко реалізувати за допомогою функцій мови C++, оскільки будь-яка функція може бути рекурсивною. Нагадаємо, що рекурсивною називається функція, в тілі якої є виклик самої себе. (Буває також непряма рекурсія, наприклад, у випадку, коли дві функції містять виклики одна одної, але ми її не розглядатимемо.)

Розглянемо таку задачу: нехай елементами дерева є цілі числа, а вершини дерева є структурами типу *Node*. Описати рекурсивну функцію, яка:

- визначає, чи входить елемент E в дерево T;
- обчислює середнє арифметичне елементів непорожнього дерева T.

Розв'язок

```
struct Node
{
	int value;
	Node* left;
	Node* right;
	Node(int x) : value(x), left(nullptr), right(nullptr) {}
};

// функція для перевірки приналежності елемента дереву
bool Contain(int E, Node* tree)
{
	if (tree == nullptr) return false;
	else
		return tree->value == E || Contain(E, tree->left) || Contain(E, tree->right);
}

// допоміжна функція для обходу дерева та накопичення суми й кількості елементів
void SumAndCount(Node* tree, int& Sum, int& Count)
{
	Sum = tree->value;
	Count = 1;
	if (tree->left != nullptr)
	{
		int leftSum, leftCount;
		SumAndCount(tree->left, leftSum, leftCount);
		Sum += leftSum;
```

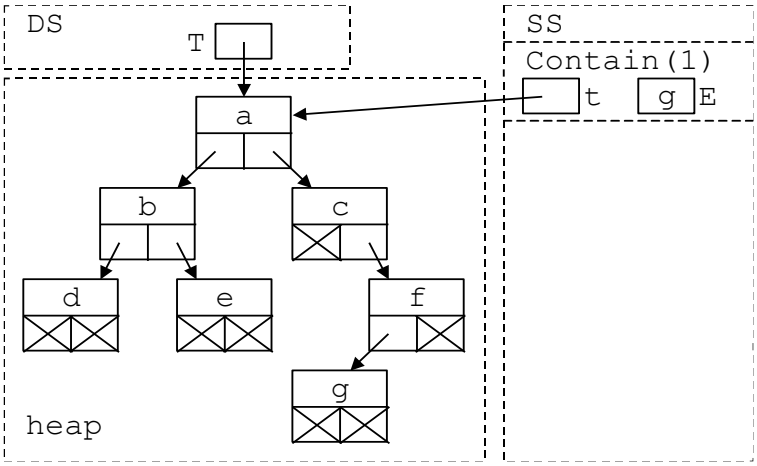
```
            Count += leftCount;
        }
        if (tree->right != nullptr)
        {
            int rightSum, rightCount;
            SumAndCount(tree->right, rightSum, rightCount);
            Sum += rightSum;
            Count += rightCount;
        }
}

// кінцева функція обчислення середнього арифметичного
double Average(Node* tree)
{
        int Sum, Count;
        SumAndCount(tree, Sum, Count);
        return (double)Sum / Count;
}
```

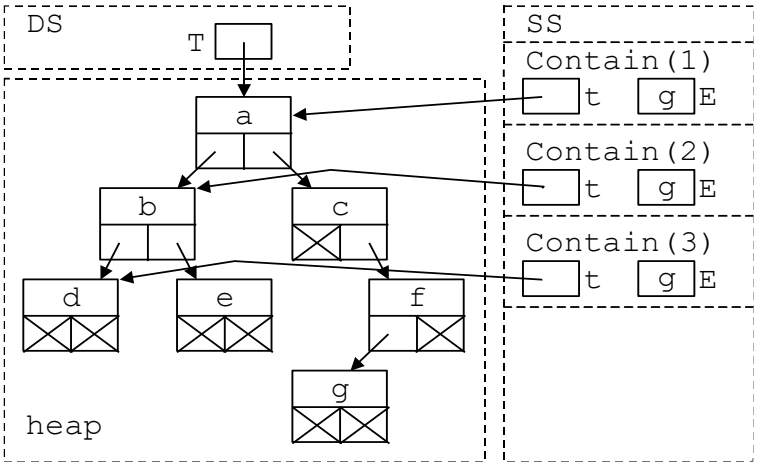
Послідовність дій за алгоритмом рекурсивної функції *Contain* проілюструємо прикладом відшукання елемента *g* у дереві *T*, яке було зображено на рис. 1, в. Вміст вершин цього дерева умовно позначено літерами, бо тип елемента для ілюстрації немає принципового значення. У конкретній ситуації він може бути, наприклад, літерний, числовий, рядковий, чи будь-який інший.

Видно, що час пошуку елемента функцією *Contain* є величиною порядку $O(n)$, де n – загальна кількість вершин дерева. У найгіршому випадку вона мусить обійти всі вершини даного дерева, як зображено на рис. 2.

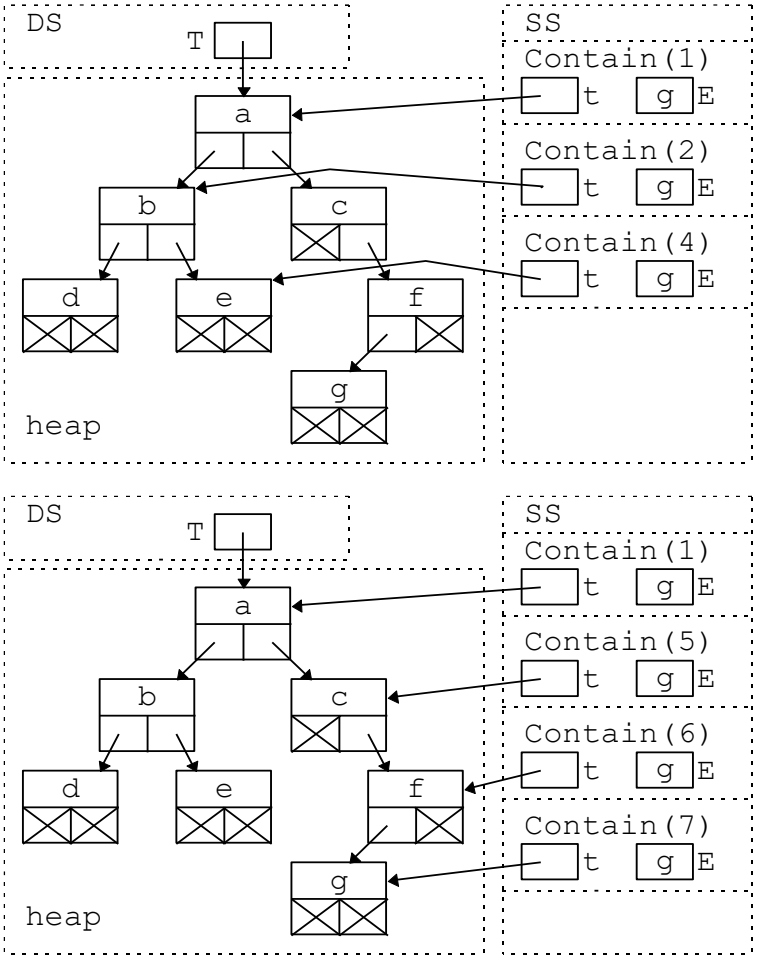
Порівняйте цю оцінку швидкодії з оцінкою часу відшукання елемента у збалансованому дереві пошуку, яка буде наведена у одному з наступних параграфів.



У момент виклику функції `Contain` її параметр `tree` отримав вказівник на дерево `T`, яке було створене раніше.



Відбулося два рекурсивних виклики `Contain(t->left,E)`. Тепер системний стек містить змінні трьох екземплярів функції.



У другому екземплярі функції відбувся виклик
`Contain(t->right,E).`

Так схематично можна зобразити зв'язки між змінними після останнього рекурсивного виклику функції Contain. У наступний момент всі її екземпляри, починаючи з останнього, завершать роботу і будуть вилучені зі стека.

Рис. 2. Лівосторонній обхід дерева рекурсивною функцією

Дерева пошуку

Вузли дерева можуть містити спеціальне поле для їхнього імені – так званого *ключа*. Ключами можуть бути довільні коди (числові, літерні), що допускають порівняння. Всі ключі в дереві повинні бути різними. Вони служать для ідентифікації вузлів дерева. Завдяки використанню ключів алгоритми обробки деревовидних структур стають простішими, адже опрацьовують не власне елементи, а їхні ключі.

Якщо для кожної вершини правильним є твердження про те, що ліве її піддерево містить лише вершини з меншими ключами, а праве – з більшими, то таке дерево називають *деревом пошуку*. Дерева пошуку зручно використовувати для організації швидкого доступу до потрібних частин великих масивів даних. Для цього даним приписують певні ключі і розташовують їх у вузлах відповідного дерева. Потрібного ключа можна знайти, скориставшись алгоритмом бінарного пошуку. Час його виконання є величиною порядку $O(\log_2 n)$, де n – кількість ключів у дереві.

Найефективнішим для пошуку є використання *ідеально збалансованих* дерев. Дерево пошуку називається ідеально збалансованим, якщо кількість вершин у всіх його відповідних лівих і правих піддеревах відрізняється не більше, ніж на одиницю.

Пошук з включенням у дереві

Розглянемо спочатку випадок, коли дерево пошуку тільки росте. Для цього розв'яжемо таку задачу: визначити частоту входження кожного слова у задану послідовність слів, яка міститься у заданому текстовому файлі по одному слову в записі файла.

Щоб розв'язати цю задачу, побудуємо двійкове дерево пошуку. Кожна його вершина міститиме саме слово (ключ), кількість його входжень у послідовність (корисна інформація) і вказівники на праве та ліве піддерева. Спочатку дерево порожнє. Кожне прочитане з файла слово потрібно шукати в дереві. Якщо слово знайдено, його лічильник збільшуємо на одиницю. У іншому випадку слово включаємо в дерево з одиничним значенням лічильника.

Таку задачу називають *пошуком по дереву з включенням*. Її реалізує наведена нижче програма.

```
// вершина дерева
	struct Node
	{
		string key;
		int count;
		Node* left;
		Node* right;
		Node(string word) : key(word), count(1),
			left(nullptr), right(nullptr) {}
	};

// функція додавання елемента даних до дерева
void insert (Node*& tree, string word)
{
	if (tree == nullptr)
	// ключ не знайшли, включаємо до дерева, дерево підросло
	{
		tree = new Node(word);
	}
	else if (tree->key == word)
		// знайшли ключ, просто збільшуємо лічильник
		++(tree->count);
	else if (word < tree->key)
		// шуканий ключ може бути в лівому піддереві
	{
		insert(tree->left, word);
	}
	else
		// ключ у правому піддереві
	{
		insert(tree->right, word);
	}
}

// структуроване виведення дерева в потік
void printWithShift(Node* tree, ostream& os, int shift)
{
	if (tree->left != nullptr) printWithShift(tree->left, os, shift + 5);
	for (int i = 0; i < shift; ++i) os << ' ';
	os << tree->key << " : " << tree->count << '\n';
	if (tree->right != nullptr) printWithShift(tree->right, os, shift + 5);
}

// головна програма
int main()
{
	Node * Text = nullptr;
	string word;
	ifstream fin("LongText.txt");
	while (!fin.eof())
	{
		fin >> word;
		insert(Text, word);
	}
	fin.close();
	printWithShift(Text, cout, 0);
	system("pause");
	return 0;
}
```

Збалансування дерева *Text*, побудованого цією програмою, залежить від порядку слів у заданій послідовності. Не можна передбачити, як воно буде рости і якої набуде форми. Напевне, це дерево не буде ідеально збалансованим. Перебудова дерева після кожного

включення для досягнення ідеальної збалансованості є занадто дорогою (довготривалою) процедурою. Її виконують для дерев, які довго залишаються без змін і використовують переважно для пошуку.

Вправа. Простежити, як відбуватиметься ріст дерева, якщо файл містить слова 'h', 'i', 'c', 'f', 'j', 'd', 'b', 'g', 'e'.

Вилучення з дерева пошуку

Вилучення з дерева є задачею оберненою до попередньої. На жаль, вона є складнішою, особливо, для дерев пошуку. Простою ця задача є лише тоді, коли потрібно вилучити листок або внутрішню вершину з одним нащадком. Якщо ж вершина, що вилучається, має двох нащадків, її треба замінити або на крайній правий елемент лівого піддерева, або на крайній лівий правого. Ці елементи можуть мати тільки одного нащадка.

Наведена нижче процедура *withdraw* розрізняє такі три випадки:

1) компоненти з заданим ключем немає;
2) компонента з заданим ключем має не більше, ніж одного нащадка;
3) компонента з заданим ключем має двох нащадків.

```
	Node* toDel;
// допоміжна функція
void delLast(Node*& r)
{
	if (r->right != nullptr) delLast(r->right);
	else
	{
		toDel->key = r->key;
		toDel->count = r->count;
		toDel = r;
		r = r->left;
	}
}
// головна функція для вилучення
bool withdrawRec(Node*& tree, string word)
{
	if (tree == nullptr) return false; // такого ключа нема
	else if (word < tree->key)
	{
		return withdrawRec(tree->left, word);
	}
	else if (word > tree->key)
	{
		return withdrawRec(tree->right, word);
	}
	else
	{
		toDel = tree;
		if (toDel->right == nullptr)
		{
			tree = toDel->left;
		}
		else if (toDel->left == nullptr)
		{
			tree = toDel->right;
		}
		else
		{
			delLast(tree->left);
		}
		delete toDel;
		return true;
	}
}
```

Вправа. Простежити як буде змінюватися збудоване раніше дерево під час вилучення слів 'i', 'b', 'd'.

Сортування деревом

Викладені ідеї використаємо для побудови алгоритму сортування масиву чисел за допомогою дерева пошуку. Подібно, як ми робили це за допомогою впорядкованого списку, вставимо всі елементи масиву до дерева пошуку, після чого обійдемо дерево зліва направо і скопіюємо всі його елементи назад в масив.

У програмі нижче використано дещо простішу структуру вузла дерева, ніж у попередніх прикладах: ми не будемо використовувати поле лічильника значень.

```
struct treeNode;
typedef treeNode* Tree_t;
struct treeNode
{
	int val;
	Tree_t left, right;
	treeNode() : val(0), left(nullptr), right(nullptr) {}
	treeNode(int x, Tree_t l, Tree_t r) : val(x), left(l), right(r) {}
};
void searchIncl(Tree_t& T, int x)
{
	// вставляє значення x в дерево пошуку T
	if (T != nullptr)
	{
		if (x < T->val) searchIncl(T->left, x);
		else searchIncl(T->right, x);
	}
	else
		T = new treeNode(x, nullptr, nullptr);
}
// функція для вилучення дерева з пам'яті
void removeTree(Tree_t& Tree)
{
	if (Tree != nullptr)
	{
		removeTree(Tree->left);
		removeTree(Tree->right);
		delete Tree;
		Tree = nullptr;
	}
}
// допоміжна функція обходить дерево t і збирає значення в масив a
// pos відслідковує місце в масиві, куди заносять значення
void takeValues(Tree_t t, int * a, size_t& pos)
{
	if (t->left != nullptr) takeValues(t->left, a, pos);
	a[pos++] = t->val;
	if (t->right != nullptr) takeValues(t->right, a, pos);
}
void sortByTree(int * A, size_t n)
{
	// щоб упорядкувати масив A, помістимо всі його елементи у впорядковане дерево
	Tree_t Tree = nullptr;
	for (size_t i = 0; i < n; ++i) searchIncl(Tree, A[i]);
	// а тоді повернемо з дерева назад в масив
	size_t k = 0;
	takeValues(Tree, A, k);
	removeTree(Tree);
}
```

Наступні параграфи – лише для сміливців

Цей матеріал не буде винесено на іспит.
Проте його варто хоча б прочитати, оскільки розуміння того, як влаштовані і як функціонують АВЛ-дерева є невід'ємною складовою освіти кваліфікованого програміста.

Збалансовані дерева

Ідеальне збалансування дерева пошуку виконати складно і дорого, тому частіше використовують просте збалансування.

Дерево називається *збалансованим* тоді і лише тоді, коли висоти двох піддерев кожної з його вершин відрізняються не більше, ніж на одиницю.

Ця вимога є суттєво слабшою порівняно з умовами ідеальної збалансованості. Дерево, яке має однакову *кількість* вершин у піддеревах, має піддерева однакової *висоти*, але не навпаки.

Збалансовані дерева називають ще АВЛ-деревами за іменами їхніх винахідників Г. М. Адельсон-Вельського та Є. М. Ландіса (1962). Для АВЛ-дерев час пошуку теж є величина порядку $O(\log_2 n)$, а довжина АВЛ-дерева в найгіршому випадку не перевищує 1,45 довжини ідеально збалансованого дерева.

Включення та вилучення елемента збалансованого дерева виконується дещо складніше.

Вставка у збалансоване дерево

Розглянемо як зміниться збалансоване дерево після включення у нього нової вершини. Нехай дерево містить корінь T, ліве піддерево L з висотою h_L і праве піддерево R з висотою h_R. Припустимо, що нову вершину включили в ліве піддерево. Це збільшить його висоту на одиницю. Тоді можливі три випадки.

1. Було $h_L = h_R$. Після вставки L і R матимуть різну висоту, але критерій збалансованості не порушиться.
2. Було $h_L < h_R$. Після вставки L і R матимуть однакову висоту, збалансованість стала кращою.
3. Було $h_L > h_R$. Після вставки критерій збалансованості буде порушено, і дерево потрібно перебудувати.

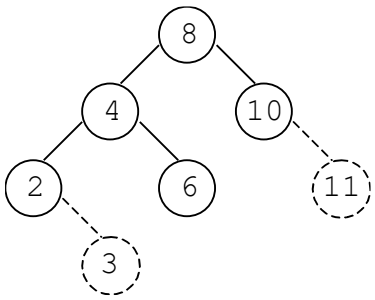


Рис. 3. Збалансоване дерево

Розглянемо приклад, зображений на рис. 3. Включення вершини 11 (або 9) зробить дерево 10 одностороннім без порушення збалансованості (випадок 1), а дерево 8 – краще збалансованим (випадок 2). Включення вершини 3 (або будь-якої з 1, 5, 7) вимагатиме подальшого балансування дерева.

Суттєво різними є лише дві ситуації: включення 1 (або 3) і включення 5 (або 7). Схеми на рис. 4 демонструють як відновити балансування. Вершини дерева переміщуються лише у вертикальному напрямі з рівня на рівень, не порушуючи відношення порядку «ліворуч – праворуч» між вершинами дерева. Впорядкування ключів не змінюється під час переміщень.

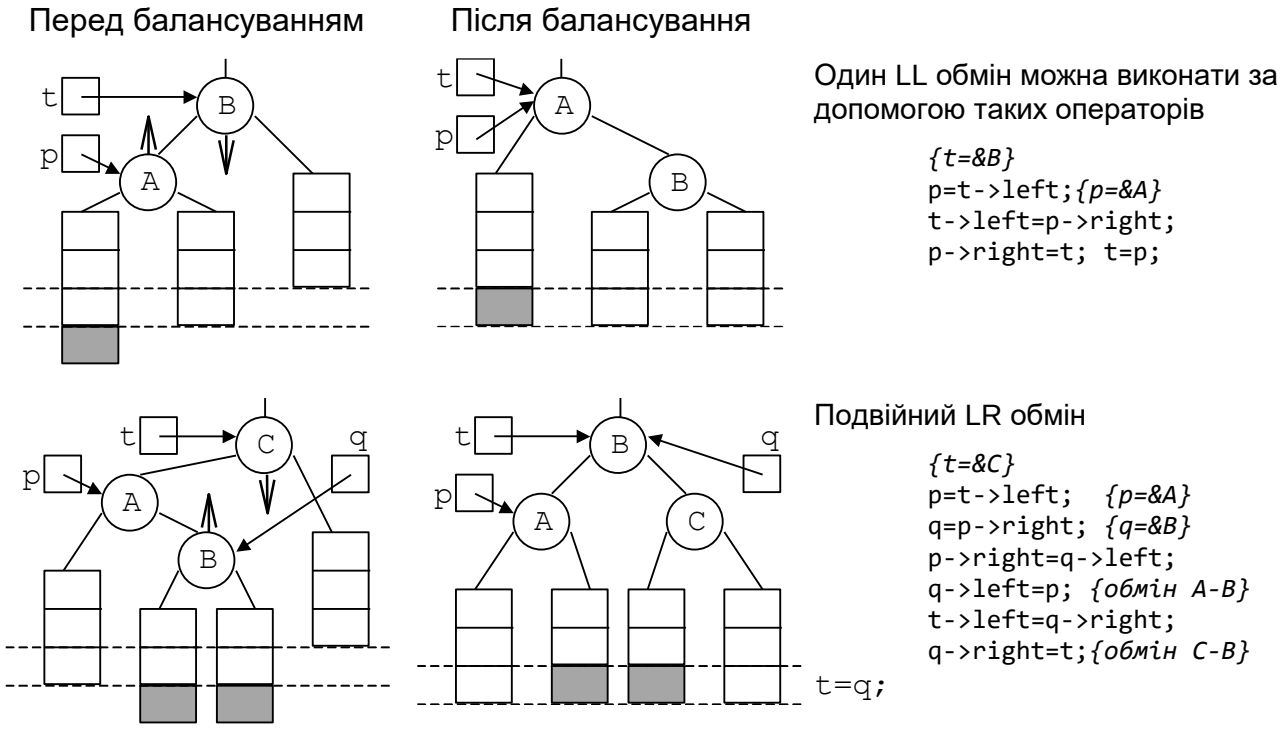


Рис. 4. Характерні ситуації відновлення збалансованості

Алгоритм впорядкування суттєво залежить від того, як зберігається інформація про збалансування. Дерево містить її уже в своїй структурі, тому можна взагалі нічого не дописувати ні в одну вершину. Тоді, щоб перевірити збалансування, довелося б щоразу переглядати піддерева вершини, до якої долучили нову. Це дуже великі затрати. Ми розширимо структуру вершини, додавши в неї поле *bal*, яке міститиме різницю висот правого і лівого піддерев $h_R - h_L$.

Тепер процес включення можна виконати так:

1) хід вперед шляхом пошуку, доки не переконаємось, що ключа в дереві немає;
2) включення вершини і визначення результуючого показника збалансованості;
3) обернений хід шляхом пошуку і перевірка в кожній вершині показника збалансованості, виконання у разі потреби перебудови дерева.

Із зробленими припущеннями можна буде легко вдосконалити вже описану раніше процедуру *Search* для виконання додаткових дій під час повернення шляхом пошуку. З цією метою додамо в заголовок процедури параметр *rise*, який має значення «висота дерева збільшилась».

Розглянемо детальніше, які перевірки і зміни потрібно виконати у вершині дерева після повернення з її лівого піддерева. (Повернення з правого піддерева обробляється аналогічно з відповідною заміною імен полів зв'язку вершини дерева.) Нехай процес повернувся у вершину `p^`. Залежно від висот її піддерев перед включенням потрібно розрізняти такі ситуації.

1. Було $h_L = h_R$, $h_R - h_L = p\text{->}bal = 0$. Стало $++h_L$, $p\text{->}bal = -1$. Ліве піддерево переважує на 1, але критерій збалансованості не порушено.
2. Було $h_L < h_R$, $h_R - h_L = p\text{->}bal = 1$. Стало $++h_L$, $p\text{->}bal = 0$. Досягли рівноваги.
3. Було $h_L > h_R$, $h_R - h_L = p\text{->}bal = -1$. Стало $++h_L$, $p\text{->}bal = -2$. Дерево потребує перебудови. Щоб її виконати, потрібно вияснити, котра зі згаданих раніше ситуацій балансування має місце:
   a) якщо $p\text{->}left\text{->}bal = -1$, то застосуємо перший спосіб балансування;
   b) якщо $p\text{->}left\text{->}bal = 1$, то – другий.

Балансування полягає у циклічному переприсвоюванні вказівників, що призводить до одно- чи двократного обміну двох чи трьох вершин, які беруть участь у процесі. Крім обертання вказівників потрібно, також належно виправити показники збалансованості.

```
	enum Balance {Lefter=-1, Equal, Righter};

	struct Node
	{
		string key;
		int count;
		Balance bal;
		Node* left;
		Node* right;
		Node(string word) : key(word), count(1), bal(Equal),
			left(nullptr), right(nullptr) {}
	};

void insertRec(Node*& tree, string word, bool& rise)
{
	if (tree == nullptr)
	// ключ не знайшли, включаємо до дерева, дерево підросло
	{
		tree = new Node(word); rise = true;
	}
	else if (tree->key == word)
		// знайшли ключ, просто збільшуємо лічильник
		++(tree->count);
	else if (word < tree->key)
		// шуканий ключ може бути в лівому піддереві
	{
		insertRec(tree->left, word, rise);
		if (rise)
		{
			switch (tree->bal)
			{
			case Righter: tree->bal = Equal; rise = false;
				// вузол став збалансованим
				break;
			case Equal: tree->bal = Lefter; // можна терпіти
				break;
			case Lefter: // ліве піддерево завелике, потрібне балансування
				Node* p = tree->left;
				if (p->bal == Lefter)
					// ліве (зовнішнє) піддерево, один обмін
				{
					tree->left = p->right;
					p->right = tree;
					tree->bal = Equal;
					tree = p;
				}
				else
					// праве (внутрішнє) піддерево, подвійний обмін
				{
					Node* q = p->right;
					p->right = q->left;
					q->left = p;
					tree->left = q->right;
					q->right = tree;
					tree->bal = (q->bal == Lefter) ? Righter : Equal;
					p->bal = (q->bal == Righter) ? Lefter : Equal;
					tree = q;
				}
				tree->bal = Equal;
				rise = false;
				break;
			}
		}
	}
	else
```

```
            // ключ у правому піддереві
      {
            insertRec(tree->right, word, rise);
            if (rise)
            {
                  switch (tree->bal)
                  {
                  case Lefter: tree->bal = Equal; rise = false;
                        // вузол став збалансованим
                        break;
                  case Equal: tree->bal = Righter; // можна терпіти
                        break;
                  case Righter: // праве піддерево завелике, потрібне балансування
                        Node* p = tree->right;
                        if (p->bal == Righter)
                              // Праве (зовнішнє) піддерево, один обмін
                        {
                              tree->right = p->left;
                              p->left = tree;
                              tree->bal = Equal;
                              tree = p;
                        }
                        else
                              // Ліве (внутрішнє) піддерево, подвійний обмін
                        {
                              Node* q = p->left;
                              p->left = q->right;
                              q->right = p;
                              tree->right = q->left;
                              q->left = tree;
                              tree->bal = (q->bal == Righter) ? Lefter : Equal;
                              p->bal = (q->bal == Lefter) ? Righter : Equal;
                              tree = q;
                        }
                        tree->bal = Equal;
                        rise = false;
                        break;
                  }
            }
      }
}
```

Складність операції балансування передбачає, що збалансовані дерева треба використовувати лише тоді, коли пошук інформації відбувається набагато частіше, ніж її включення.

Вправа. Нехай ключами дерева пошуку є натуральні числа. Простежити, як відбувається його ріст і балансування, якщо дерево будується з послідовності чисел 4, 5, 7, 2, 1, 3, 6.

Вилучення зі збалансованих дерев

Для звичайних дерев вилучення вершини було більш трудомістким, ніж включення. Така ж ситуація спостерігається і для збалансованих дерев.

Принципова схема алгоритму залишається такою самою: простими є випадки вилучення листка чи вершини з одним нащадком. Якщо ж вершина має два піддерева, то її замінюємо на крайню праву вершину її лівого піддерева. Для контролю за необхідністю балансування використовується логічна змінна *diminish*. Вона вказує, чи зменшилась висота дерева. Перебудови дерева з метою збалансування (як і під час включення) базуються на одно- або двократних обмінах вершин.

```
      Node* toDel;
// допоміжні функції
```

```
void balanceL(Node*& t, bool& diminish)
{
	// diminish=true ліва гілка стала коротшою
	switch (t->bal)
	{
	case Lefter: t->bal = Equal;
		break;
	case Equal: t->bal = Righter; diminish = false;
		break;
	case Righter: // перебудова
		Node* p = t->right; Balance bp = p->bal;
		if (bp >= Equal)
		{
			// один обмін
			t->right = p->left;
			p->left = t;
			if (bp = Equal)
			{
				t->bal = Righter;
				p->bal = Lefter;
				diminish = false;
			}
			else
			{
				t->bal = Equal;
				p->bal = Equal;
			}
			t = p;
		}
		else
		{
			// подвійний обмін
			Node* q = p->left; Balance bq = q->bal;
			p->left = q->right;
			q->right = p;
			t->right = q->left;
			q->left = t;
			t->bal = (bq = Righter) ? Lefter : Equal;
			p->bal = (bq = Lefter) ? Righter : Equal;
			t = q;
			q->bal = Equal;
		}
		break;
	}
}
void balanceR(Node*& t, bool& diminish)
{
	// diminish=true права гілка стала коротшою
	switch (t->bal)
	{
	case Righter: t->bal = Equal;
		break;
	case Equal: t->bal = Lefter; diminish = false;
		break;
	case Lefter: // перебудова
		Node* p = t->left; Balance bp = p->bal;
		if (bp <= Equal)
		{
			// один обмін
			t->left = p->right;
			p->right = t;
			if (bp = Equal)
			{
				t->bal = Lefter;
				p->bal = Righter;
```

```
                        diminish = false;
                }
                else
                {
                        t->bal = Equal;
                        p->bal = Equal;
                }
                t = p;
        }
        else
        {
                // подвійний обмін
                Node* q = p->right; Balance bq = q->bal;
                p->right = q->left;
                q->left = p;
                t->left = q->right;
                q->right = t;
                t->bal = (bq = Lefter) ? Righter : Equal;
                p->bal = (bq = Righter) ? Lefter : Equal;
                t = q;
                q->bal = Equal;
        }
        break;
    }
}
void delLast(Node*& r, bool& diminish)
{
    //diminish == false
    if (r->right != nullptr)
    {
        delLast(r->right, diminish);
        if (diminish) balanceR(r, diminish);
    }
    else
    {
        toDel->key = r->key;
        toDel->count = r->count;
        toDel = r;
        r = r->left;
        diminish = true;
    }
}
// головна функція для вилучення
bool withdrawRec(Node*& tree, string word, bool& diminish)
{
    // diminish == false
    if (tree == nullptr) return false; // такого ключа нема
    else if (word < tree->key)
    {
        bool result = withdrawRec(tree->left, word, diminish);
        if (diminish) balanceL(tree, diminish);
        return result;
    }
    else if (word > tree->key)
    {
        bool result = withdrawRec(tree->right, word, diminish);
        if (diminish) balanceR(tree, diminish);
        return result;
    }
    else
    {
        toDel = tree;
        if (toDel->right == nullptr)
        {
            tree = toDel->left;
```

```
                diminish = true;
            }
            else if (toDel->right == nullptr)
            {
                tree = toDel->right;
                diminish = true;
            }
            else
            {
                delLast(tree->left, diminish);
                if (diminish) balanceL(tree, diminish);
            }
            delete toDel;
            return true;
        }
}
```

Балансування виконується, якщо змінна *diminish* набуває значення «істина». Це трапляється тоді, коли з дерева вилучено знайдену вершину, або коли висота дерева змінилась у ході балансування.

У поведінці процедур *insertRec* і *withdrawRec* є суттєва різниця. Якщо включення одного елемента може призвести щонайбільше до одного обміну (двох чи трьох вершин), то вилученя може потребувати обмінів у всіх вершинах шляху пошуку. Проте ймовірність балансування дерева після вставки є вищою, ніж після вилучення. Експериментально встановлено, що один обмін трапляється на кожні дві вставки і на кожні п’ять вилучень, тому затрати на вставку і вилучення є приблизно однаковими.

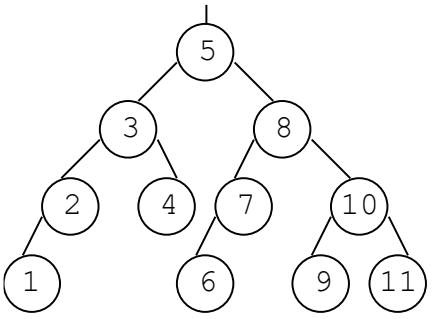


Рис. 5. Збалансоване дерево перед вилученням значень

Вправа. Простежити, як буде змінюватися збалансоване дерево, зображене на рис. 5 під час вилучення ключів 4, 8, 6, 5, 2, 1, 7.

Література

1. Бублик В.В. Об'єктно-орієнтоване програмування мовою С++.