Санкт-Петербургский Научный Исследовательский Университет
Информационных Технологий, Механики и Оптики

Задачи седьмой недели

по курсу «Алгоритмы и структуры данных»

на Openedu

Выполнил: студент группы P3218

Артамонов Александр Владимирович

Санкт-Петербург, 2019г

Задача 1. Проверка сбалансированности

АВЛ-дерево является сбалансированным в следующем смысле: для любой вершины высота ее левого поддерева отличается от высоты ее правого поддерева не больше, чем на единицу.

Обратите внимание, что, по историческим причинам, определение баланса в этой и последующих задачах этой недели "зеркально отражено" по сравнению с определением баланса в лекциях! Надеемся, что этот факт не доставит Вам неудобств. В литературе по алгоритмам — как российской, так и мировой — ситуация, как правило, примерно та же.

Дано двоичное дерево поиска. Для каждой его вершины требуется определить ее баланс.

Формат входного файла

Входной файл содержит описание двоичного дерева. В первой строке файла находится число $N(1 \le N \le 2 \cdot 10^5)$ — число вершин в дереве. В последующих N строках файла находятся описания вершин дерева. В (i+1)-ой строке файла $(1 \le i \le N)$ находится описание i-ой вершины, состоящее из трех чисел Ki, Li, Ri, разделенных пробелами — ключа в i-ой вершине ($|Ki| \le 10^9$), номера левого ребенка i-ой вершины ($i < Li \le N$ или Li = 0 если левого ребенка нет) и номера правого ребенка i-ой вершины ($i < Ri \le N$ или Ri = 0, если правого ребенка нет).

Все ключи различны. Гарантируется, что данное дерево является деревом поиска.

Формат выходного файла

Для i-ой вершины в i-ой строке выведите одно число — баланс данной вершины.

Пример

input.txt	output.txt
6 -2 0 2 8 4 3 9 0 0 3 6 5 6 0 0 0 0 0	3 -1 0 0 0 0

Код программы (C++)

```
//Узел хранит позиции левого и правого узлов-потомков
struct node {
	int left, right;
};

//Возвращает максимальное из двух чисел
long max(long a, long b) {
	if (a > b) {
		return a;
	}
	else {
		return b;
	}
}

//Находит высоту i-того элемента дерева tree
long depth(node* tree, long i, long* height) {

	//Если высота этой вершины уже посчитана, то не нужно пересчитывать её снова
	if (height[i] != 200000) {
		return height[i];
	}
	else {
		long d = -1;

		//Если потомок существует, то находим его глубину
		if (tree[i].left != 0) {
			d = max(depth(tree, tree[i].left, height), d);
		}

		if (tree[i].right != 0) {
			d = max(depth(tree, tree[i].right, height), d);
		}

		//Записываем посчитанную высоту в массив высот
		height[i] = ++d;

		return d;
	}
}

int main() {
	long N, K;

	io >> N;

	//Создаём массив узлов
	node* tree = new node[N + 1];

	//Создаём массив высот
	long* height = new long[N + 1];
	for (long i = 0; i < N + 1; i++) {
		height[i] = 200000;
	}

	for (long i = 1; i < N + 1; i++)
	{
		io >> K >> tree[i].left >> tree[i].right;
	}

	long right, left;
```

```
	for (long i = 1; i < N + 1; i++) {
		//Для каждого узла находим высоты левого и правого поддеревьев
		if (tree[i].right) {
			right = depth(tree, tree[i].right, height);
		}
		else {
			right = -1;
		}
		if (tree[i].left) {
			left = depth(tree, tree[i].left, height);
		}
		else {
			left = -1;
		}
		//Выводим баланс равный высота правого - высота левого
		io << right - left << "\n";
	}

	return 0;
}
```

Бенчмарк (задача 1)

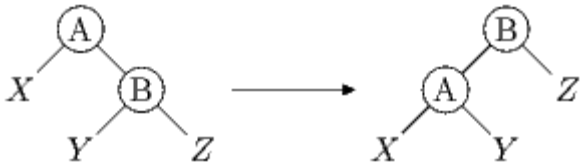
№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.093	17698816	3986010	1688889
1	OK	0.015	2232320	46	19

Задача 2. Делаю я левый поворот…

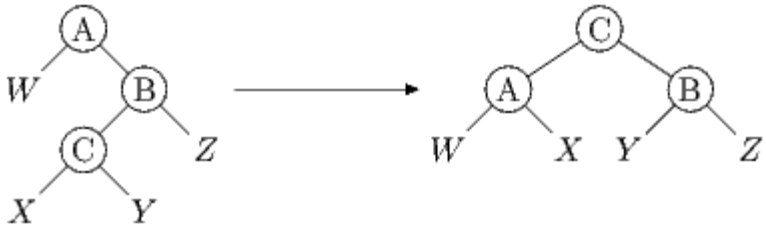
Для балансировки АВЛ-дерева при операциях вставки и удаления производятся *левые* и *правые* повороты. Левый поворот в вершине производится, когда баланс этой вершины больше 1, аналогично, правый поворот производится при балансе, меньшем −1.

Существует два разных левых (как, разумеется, и правых) поворота: *большой* и *малый* левый поворот.

Малый левый поворот осуществляется следующим образом:



Заметим, что если до выполнения малого левого поворота был нарушен баланс только корня дерева, то после его выполнения все вершины становятся сбалансированными, за исключением случая, когда у правого ребенка корня баланс до поворота равен −1. В этом случае вместо малого левого поворота выполняется большой левый поворот, который осуществляется так:



Дано дерево, в котором баланс корня равен 2. Сделайте левый поворот.

Формат входного файла

Входной файл содержит описание двоичного дерева. В первой строке файла находится число N ($3 \le N \le 2 \cdot 10^5$) — число вершин в дереве. В последующих N строках файла находятся описания вершин дерева. В (i+1)-ой строке файла ($1 \le i \le N$) находится описание i-ой вершины, состоящее из трех чисел Ki, Li, Ri, разделенных пробелами — ключа в i-ой вершине ($|Ki| \le 109$), номера левого ребенка i-ой вершины ($i < Li \le N$ или Li=0, если левого ребенка нет) и номера правого ребенка i-ой вершины ($i < Ri \le N$ или Ri=0, если правого ребенка нет).

Все ключи различны. Гарантируется, что данное дерево является деревом поиска. Баланс корня дерева (вершины с номером 1) равен 2, баланс всех остальных вершин находится в пределах от −1 до 1.

Формат выходного файла

Выведите в том же формате дерево после осуществления левого поворота. Нумерация вершин может быть произвольной при условии соблюдения формата. Так, номер вершины должен быть меньше номера ее детей.

Пример

input.txt	output.txt
7 -2 7 2 8 4 3 9 0 0 3 6 5 6 0 0 0 0 0 -7 0 0	7 3 2 3 -2 4 5 8 6 7 -7 0 0 0 0 0 6 0 0 9 0 0

Код программы (C++)

```
//Узел хранит позиции левого и правого узлов-потомков
struct node {
	long value, left, right;
};
//Возвращает максимальное из двух чисел
long max(long a, long b) {
	if (a > b) {
		return a;
	}
	else {
		return b;
	}
}

//Находит высоту i-того элемента дерева tree
long depth(node* tree, long i, long* height) {
	if (height[i] != 200000) {
		return height[i];
	}
	else {
		long d = -1;
		//Если потомок существует, то находим его ывысоту
		if (tree[i].left != 0) {
			d = max(depth(tree, tree[i].left, height), d);
		}
		if (tree[i].right != 0) {
			d = max(depth(tree, tree[i].right, height), d);
		}
		//Возвращаем максимальную высоту потомков + 1
		height[i] = ++d;
		return d;
	}
}
```

```
//Находит баланс pos-того узла
long balance(node* tree, long pos, long* height) {
	long left, right;
	if (tree[pos].right) {
		right = depth(tree, tree[pos].right, height);
	}
	else {
		right = -1;
	}
	if (tree[pos].left) {
		left = depth(tree, tree[pos].left, height);
	}
	else {
		left = -1;
	}
	return right - left;
}

long left_turn(node* tree, long* height) {
	long new_root;

	//Если баланс правого дочернего элемента равен -1 - совершаем большой поворот
	if (balance(tree, tree[1].right, height) == -1) {
		long B = tree[1].right;
		long C = tree[tree[1].right].left;
		tree[1].right = tree[C].left;
		tree[B].left = tree[C].right;
		tree[C].left = 1;
		tree[C].right = B;
		new_root = C;
	}
	else {
		//Если нет, то малый
		long tmp = tree[tree[1].right].left;
		tree[tree[1].right].left = 1;
		new_root = tree[1].right;
		tree[1].right = tmp;
	}
	return new_root;
}

long current_index = 1;

//Присваивает узлам возрастающие индексы начиная с родителя
void calculate_indexes(long root, long*& indexes, node* tree) {
	if (!root) {
		indexes[root] = 0;
		return;
	}
	indexes[root] = current_index++;
	calculate_indexes(tree[root].left, indexes, tree);
	calculate_indexes(tree[root].right, indexes, tree);
}

//Выводит дерево сортируя его по индексам массива indexes
void print_node(long root, long*& indexes, node* tree) {
	if (!root) { return; }

	io << tree[root].value << " " << indexes[tree[root].left] << " " <<
indexes[tree[root].right] << "\n";

	print_node(tree[root].left, indexes, tree);
	print_node(tree[root].right, indexes, tree);
}
```

```
int main() {
	long N;

	io >> N;
	//Создаём массив узлов
	node* tree = new node[N + 1];

	long* height = new long[N + 1];
	for (long i = 0; i < N + 1; i++) {
		height[i] = 200000;
	}

	for (long i = 1; i < N + 1; i++)
	{
		io >> tree[i].value >> tree[i].left >> tree[i].right;
	}

	long new_root = left_turn(tree, height);

	long* indexes = new long[N + 1];
	calculate_indexes(new_root, indexes, tree);
	io << N << "\n";
	print_node(new_root, indexes, tree);

	return 0;
}
```

Бенчмарк (задача 2)

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.187	9809920	3986416	3986416
1	OK	0.000	2220032	54	54