МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННО БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Учебный Центр Информационных Технологий «Информатика»

Лабораторная работа №5

по дисциплине «Информатика и программирование 2 часть»

Направление подготовки: 230105 - «Программное обеспечение вычислительной техники и автоматизированных систем»

Выполнил слушатель: Бройтман Евгений Давидович
Вариант: 6.4
Дата сдачи:
Преподаватель: Силов Я.В.

Новосибирск, 2016г.

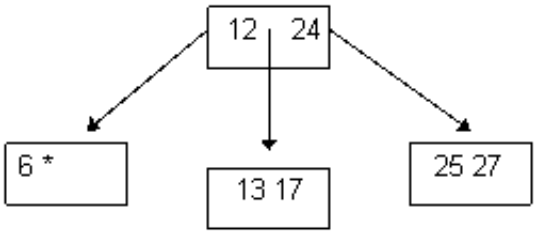
1. Цель

Познакомиться с древовидными структурами данных.

2. Вариант задания

Для древовидных структур данных предусмотреть вывод характеристик сбалансированности дерева (средняя длина ветви) и процедуру выравнивания (балансировки). При выполнении работы произвести измерение зависимости «грязного» времени работы программы от количества вершин дерева. Оценить вид полученной зависимости (линейно-логарифмическая, квадратичная).

Упрощенное 2-3 дерево. Вершина дерева содержит два указателя на объекты и три указателя на поддеревья. Данные в дереве упорядочены.



3. Теория

В самом общем смысле деревом называется набор элементов одного типа с нециклическими однонаправленными непересекающимися связями. В дереве реализована иерархическая структура представления данных: чтобы попасть на какой-то нужный уровень, нужно пройти все предыдущие. Элемент дерева называется вершиной. Связь между двумя вершинами называется ветвью. Вершина, которая ссылается на текущую вершину, называется предком. Вершины, на которые ссылается текущая вершина, называются потомками. Вершина, у которой нет предка, называется корнем. Вершина, у которой нет потомков, называются конечной. Каждый потомок образует свое поддерево, являясь в нем корнем.

Можно подобрать способ представления, в котором физическая структура максимально соответствует логической структуре дерева, т.е. ее внешнему виду: корень, ветви, потомки. Если ветвь считать указателем, то вершина – это структура, содержащая массив указателей на потомков.

Хотя деревья являются и топологически сложными структурами данных, оценить пределы и условия их эффективности довольно легко. Прежде всего, введем такую характеристику дерева, как сбалансированность. Сбалансированность характеризует разброс длин ветвей дерева. Более точно, речь идет о расстояниях от корневой вершины до вершины со свободной ветвью. Дерево называется сбалансированным, если длины максимальной и минимальной ветвей отличаются не более чем на 1. Для такого дерева характерна экспоненциальная (или обратная, логарифмическая) зависимость между длиной ветви и количеством вершин в дереве:

$$N = 1 + m + m^2 + m^3 + \ldots + m^L \sim m^L,\ L \sim \log_m N,$$ где

m – количество потомков одной вершины,

N – количество вершин в дереве,

L – максимальная длина ветви

В самом худшем случае при наличии только одного потомка в каждой вершине дерево вырождается в список, в этом случае длина ветви равна количеству вершин без 1.

Далее, все алгоритмы, основанные на однократном, полном рекурсивном обходе, будут иметь линейную трудоемкость T=N. Их усовершенствование может состоять только в ограничении «глубины погружения», если на это есть достаточные основания. Другое дело, алгоритмы, основанные на ветвлении. В каждой вершине они выбирают только одного из возможных потомков. Такие алгоритмы называют жадными. Сразу же можно увидеть, что их трудоемкость равна длине выбранной ими ветви дерева. Для сбалансированного дерева зависимость будет логарифмической:

$$T = L \sim \log_m N,$$

для вырожденного в список дерева – линейной. Здесь возникают две проблемы. Во-первых, поддержка сбалансированности дерева. Для нее имеется два решения:

– использование алгоритмов, сохраняющих сбалансированность дерева, которые являются значительно более сложными, чем обычно, поскольку используют различные топологические преобразования групп смежных вершин дерева (причем рекурсивно);

– периодическое выравнивание (балансировка) дерева, возможно с использованием дополнительной структуры данных. Данное решение позволяет использовать простые алгоритмы работы с деревом, но требует слежения за его сбалансированностью (заметим, что это могут делать те же самые алгоритмы, например, определяя длину ветви при поиске заданного значения). Процедура балансировки может быть достаточно трудоемкой, но вызываемой сравнительно редко.

На житейском уровне эту альтернативу можно сформулировать как «идеальный порядок или периодическая генеральная уборка». Аналогичная ситуация возникает в любых системах динамического распределения и утилизации ресурсов: в системе динамического распределения памяти, при планировании двоичного файла, в системе управления файлами операционной системы, где она имеет похожие решения.

Вторая проблема упирается в основания, которые имеет алгоритм для однозначного выбора единственного потомка в каждой вершине (художественная аналогия - «рыцарь на распутье»). Здесь опять же возможны два подхода:

– наличие в вершине дополнительной (избыточной) информации, позволяющей сделать такой выбор «здесь и сейчас»;

– наличие определенного порядка размещения данных в дереве.

4. Анализ задачи и алгоритм

1) **Анализ задачи**

Входные данные: несколько ненормализованных упорядоченных деревьев 2-3, отличающихся количеством вершин.

Результат:

Средняя длина ветви деревьев. «Грязное» время балансировки. Оценка зависимости времени балансировки от количества вершин дерева (линейно-логарифмическая, квадратичная).

Метод решения:

Определение рекурсивным способом суммы длин всех ветвей и их количества. Средняя длина ветви определяется делением суммы длин на количество ветвей. Балансировка дерева осуществляется путем:

– рекурсивного разложения дерева в массив, содержащий значения данных вершин дерева, упорядоченные по возрастанию;
– рекурсивного деления массива на равные (или почти равные) части, количество которых равно количеству потомков в вершине (в нашем случае - 3). Указатели на граничные после деления массива значения записываются в текущую вершину дерева (их количество на 1 меньше количества потомков, в нашем случае - 2). Полученные части исходного массива содержат значения данных соответствующих поддеревьев. Каждая из этих частей также подвергается делению на равные части и определению значений данных вершин. И так происходит до тех пор, пока все элементы массива не будут принадлежать соответствующим вершинам сбалансированного дерева.
– Время выполнения балансировки определяем путем засечки времени начала и конца балансировки с последующим вычитанием.
– Оценка зависимости времени балансировки от количества вершин дерева выполняется путем сравнения полученной зависимости времени и эталонных зависимостей (линейно-логарифмическая, квадратичная).

Выделенные подзадачи: Для определения средней длины ветви необходимо знать количество ветвей дерева и сумму длин всех ветвей. Для балансировки дерева потребуется знать количество вершин дерева. Для определения разбалансированности дерева необходимы величины максимальной длины ветви (или высоты) и минимальной длины ветви дерева. Для наглядности нужен вывод дерева на экран. Для вывода дерева на экран необходим вывод одного уровня дерева на экран. Для балансировки дерева потребуется упорядоченное дерево преобразовать в упорядоченный массив, и потом

упорядоченный массив – в нормализованное упорядоченное дерево. Выделим определение этих величин и выполнение действий в качестве отдельных подзадач.

2) Алгоритм решения задачи

Псевдокод:

a. Определение количества ветвей дерева.
   a.1. Начинаем с корневой вершины дерева.
   a.2. Если вершина дерева пустая – дошли до конца ветви, возвращаем 1.
   a.3. Обнуляем количество ветвей дерева для текущего корня.
   a.4. Для каждого потомка по порядку, принимая его за корень, повторяем действия, начиная с п. a.2, при этом увеличивая значение количества ветвей дерева на величину, возвращенную в результате этих действий.
   a.5. Возвращаем количество ветвей.
b. Определение суммы длин всех ветвей дерева.
   b.1. Начинаем с корневой вершины дерева и нулевого уровня.
   b.2. Если вершина дерева пустая – дошли до конца ветви, возвращаем номер уровня, т.е. длину ветви с учетом нулевых вершин.
   b.3. Обнуляем суммарную длину ветвей дерева для текущего корня.
   b.4. Для каждого потомка по порядку, принимая его за корень и увеличивая на 1 номер уровня, повторяем действия, начиная с п. b.2, при этом увеличивая значение суммарной длины ветвей дерева на величину, возвращенную в результате этих действий.
   b.5. Возвращаем суммарную длину ветвей дерева для текущего корня.
c. Определение количества вершин дерева.
   c.1. Начинаем с корневой вершины дерева.
   c.2. Если вершина дерева пустая – дошли до конца ветви, возвращаем 0.
   c.3. Количеству вершин для текущего корня присваиваем значение 1.
   c.4. Для каждого потомка по порядку, принимая его за корень, повторяем действия, начиная с п. c.2, при этом увеличивая значение количества вершин дерева для текущего корня на величину, возвращенную в результате этих действий.
   c.5. Возвращаем количество вершин дерева для текущего корня.
d. Определение максимальной длины ветви (высоты с учетом нулевых вершин) дерева.
   d.1. Начинаем с корневой вершины дерева.
   d.2. Если вершина дерева пустая – дошли до конца ветви, возвращаем 0.
   d.3. Для каждого потомка по порядку, принимая его за корень, повторяем действия, начиная с п. d.2, при этом присваивая соответствующим максимальным длинам ветвей дерева для текущего корня величину, возвращенную в результате этих действий.
   d.4. Возвращаем максимальное значение из вычисленных на предыдущих шагах максимальных длин ветвей каждого потомка дерева для текущего корня, увеличенное на 1.
e. Определение минимальной длины ветви дерева.
   e.1. Начинаем с корневой вершины дерева.
   e.2. Если вершина дерева пустая – дошли до конца ветви, возвращаем 0.
   e.3. Для каждого потомка по порядку, принимая его за корень, повторяем действия, начиная с п. e.2, при этом присваивая соответствующим минимальным длинам ветвей дерева для текущего корня величину, возвращенную в результате этих действий.
   e.4. Возвращаем минимальное значение из вычисленных на предыдущих шагах минимальных длин ветвей каждого потомка дерева для текущего корня, увеличенное на 1.
f. Вывод дерева на экран.
   f.1. Вычисляем высоту дерева.
   f.2. Выводим по порядку с новой строки каждый уровень дерева, начиная с нулевого.
g. Вывод на экран одного уровня дерева.
   g.1. Начинаем с корневой вершины дерева, нулевого уровня.
   g.2. Задаем форматную строку для вывода одного значения вершины дерева. Форматная строка имеет вид «%x<x>s», где x<x> - одно- или двухразрядное десятичное целое число знакомест, занимаемых одним значением вершины дерева.
   g.3. Инициализируем количество пробелов, которые нужно напечатать перед вершиной текущего уровня по формуле: количество знакомест для разделителя между вершинами

нижнего уровня / 2. Здесь и далее все действия производятся в целочисленной арифметике.

g.4. Инициализируем количество пробелов, которые нужно напечатать после вершины текущего уровня по формуле: количество знакомест для разделителя между вершинами нижнего уровня минус количество пробелов, которые нужно напечатать перед вершиной текущего уровня.

g.5. Вычисляем количество знакомест, занимаемых одной вершиной дерева по формуле: количество знакомест, занимаемых одним значением данных вершины * количество значений данных в вершине + (количество значений данных в вершине – 1) * количество знакомест для разделителя между значениями данных одной вершины.

g.6. Если текущий уровень меньше запрашиваемого:

g.6.1. Если вершина дерева не пустая – не дошли до конца ветви - для каждого потомка по порядку, принимая его за корень, и увеличивая номер текущего уровня на 1, повторяем действия, начиная с п. g.2, таким образом, переходя к следующему уровню.

g.6.2. Иначе (вершина дерева пустая – дошли до конца ветви) - необходимо напечатать пробелы в запрашиваемом уровне (текстовый курсор находится в запрашиваемом уровне), чтобы восполнить отсутствующие элементы из-за нулевых вершин в текущем уровне и сохранить симметрию дерева:

g.6.2.1. Если количество потомков четное:

g.6.2.1.1. Повторяем количество раз равное разнице между высотой дерева минус 1 и номером текущего уровня:

g.6.2.1.1.1. Временной переменной 1 присваиваем значение количества пробелов, которые нужно напечатать перед вершиной текущего уровня.

g.6.2.1.1.2. Временной переменной 2 присваиваем значение количества пробелов, которые нужно напечатать после вершины текущего уровня.

g.6.2.1.1.3. Вычисляем количество пробелов, которые нужно напечатать перед вершиной по формуле: количество потомков вершины * (количество знакомест, занимаемых одной вершиной дерева + временная переменная 1 + временная переменная 2) / 2 - количество знакомест, занимаемых одной вершиной дерева / 2.

g.6.2.1.1.4. Вычисляем количество пробелов, которые нужно напечатать после вершины по формуле: количество пробелов, которые нужно напечатать перед вершиной + количество знакомест, занимаемых одной вершиной дерева / 2 – (количество знакомест, занимаемых одной вершиной дерева - количество знакомест, занимаемых одной вершиной дерева / 2).

g.6.2.2. Иначе (если количество потомков нечетное):

g.6.2.1.2. Повторяем количество раз равное разнице между высотой дерева минус 1 и номером текущего уровня:

g.6.2.1.2.1. Временной переменной 1 присваиваем значение количества пробелов, которые нужно напечатать перед вершиной текущего уровня.

g.6.2.1.2.2. Временной переменной 2 присваиваем значение количества пробелов, которые нужно напечатать после вершины текущего уровня.

g.6.2.1.2.3. Вычисляем количество пробелов, которые нужно напечатать перед вершиной по формуле: (количество потомков вершины - 1) * (количество знакомест, занимаемых одной вершиной дерева + временная переменная 1 + временная переменная 2) / 2 + временная переменная 1.

g.6.2.1.2.4. Вычисляем количество пробелов, которые нужно напечатать после вершины по формуле: количество пробелов, которые нужно напечатать перед вершиной - временная переменная 1 + временная переменная 2.

g.6.2.3. Для сохранения симметрии дерева - вывод вычисленного количества пробелов перед вершиной, пробелов на месте вершины и вычисленного количества пробелов после вершины.

g.7. Иначе (находимся на запрашиваемом уровне):

g.7.1. Если количество потомков четное:

g.7.1.1. Повторяем количество раз равное разнице между высотой дерева минус 1 и номером текущего уровня:

g.7.1.1.1. Временной переменной 1 присваиваем значение количества пробелов, которые нужно напечатать перед вершиной текущего уровня.

g.7.1.1.2. Временной переменной 2 присваиваем значение количества пробелов, которые нужно напечатать после вершины текущего уровня.

g.7.1.1.3. Вычисляем количество пробелов, которые нужно напечатать перед вершиной по формуле: количество потомков вершины * (количество знакомест, занимаемых одной вершиной дерева + временная переменная 1 + временная переменная 2) / 2 - количество знакомест, занимаемых одной вершиной дерева / 2.

g.7.1.1.4. Вычисляем количество пробелов, которые нужно напечатать после вершины по формуле: количество пробелов, которые нужно напечатать перед вершиной + количество знакомест, занимаемых одной вершиной дерева / 2 – (количество знакомест, занимаемых одной вершиной дерева - количество знакомест, занимаемых одной вершиной дерева / 2).

g.7.2. Иначе (если количество потомков нечетное):

g.7.2.1. Повторяем количество раз равное разнице между высотой дерева минус 1 и номером текущего уровня:

g.7.2.1.1. Временной переменной 1 присваиваем значение количества пробелов, которые нужно напечатать перед вершиной текущего уровня.

g.7.2.1.2. Временной переменной 2 присваиваем значение количества пробелов, которые нужно напечатать после вершины текущего уровня.

g.7.2.1.3. Вычисляем количество пробелов, которые нужно напечатать перед вершиной по формуле: (количество потомков вершины - 1) * (количество знакомест, занимаемых одной вершиной дерева + временная переменная 1 + временная переменная 2) / 2 + временная переменная 1.

g.7.2.1.4. Вычисляем количество пробелов, которые нужно напечатать после вершины по формуле: количество пробелов, которые нужно напечатать перед вершиной - временная переменная 1 + временная переменная 2.

g.7.3. Вывод вычисленного количества пробелов перед вершиной.

g.7.4. Задаем форматную строку для вывода одного значения вершины дерева. Форматная строка имеет вид «%x<x>d», где x<x> - одно- или двухразрядное десятичное целое число знакомест, занимаемых одним значением вершины дерева.

g.7.5. Вывод значений вершины (если значения вершины нулевые выводится символ '*') и вычисленного количества пробелов после вершины.

h. Преобразование упорядоченного дерева в упорядоченный массив.

h.1. Начинаем с корневой вершины дерева и индекса массива равного 0.

h.2. Если вершина дерева не нулевая - для каждого потомка по порядку, принимая его за корень, повторяем действия, начиная с п. h.2, т.е. ищем конец ветки (нулевая вершина).
h.3. После того, как происходит возврат от найденного конца ветки в текущую вершину - записываем в текущий элемент массива ненулевое значение данных текущей вершины, которое больше значений данных вершин поддерева, из которого произошел возврат, но меньше значений данных вершин следующего поддерева.
h.4. Увеличиваем индекс массива на 1.
h.5. После возврата из последнего потомка вершины значения данных в массив не записываем.

i. Преобразование упорядоченного массива – в нормализованное упорядоченное дерево.
i.1. Начинаем с корневой вершины дерева, индекса в массиве вершин дерева равного 0 и количества элементов массива, соответствующего всем вершинам дерева.
i.2. Если количество элементов массива > 0:
i.2.1. Разбиваем массив на равные, по возможности, подмассивы, количество которых равно количеству потомков одной вершины.
i.2.2. Первому полю указателей на данные элемента массива вершин дерева с текущим индексом присваиваем значение указателя на элемент входного упорядоченного массива, который является границей разбиения массива между первым и вторым подмассивами.
i.2.3. Для всех полей указателей на данные элемента массива вершин дерева с текущим индексом:
i.2.3.1. Для каждого значения данных вершины дерева вычисляем индекс значения данных во входном упорядоченном массиве по формуле: индекс предыдущего значения данных во входном упорядоченном массиве + 1 + индекс первого значения данных во входном упорядоченном массиве.
i.2.3.2. Если элементы с вычисленными индексами во входном упорядоченном массиве есть – присваиваем указатели на эти элементы полям указателей на данные элемента массива вершин дерева.
i.2.3.3. Иначе (Если элементов с вычисленными индексами во входном упорядоченном массиве нет, т.е. индекс выходит за рамки массива) – присваиваем полям указателей на данные элемента массива вершин дерева значения NULL.
i.2.4. Для всех по порядку потомков текущего элемента массива вершин дерева повторяем действия начиная с п. i.2, принимая их за корень, при этом входными упорядоченными массивами являются получившиеся после разбиения подмассивы, а индекс в массиве вершин дерева увеличивается для каждого потомка на 1.
i.2.5. Присваиваем указатель на вершину сформированного потомка в соответствующее поле текущей вершины. Если указатель на вершину сформированного потомка == NULL – уменьшаем значение индекса в массиве вершин дерева на 1, т.к. для такого потомка вершина не формируется.
i.2.6. Возвращаем указатель на сформированную вершину родительской вершине.
i.3. Возвращаем NULL.

5. Описание программной реализации

1) Используемые переменные

Keys – массив, используемый для хранения значений данных, на которые ссылаются поля вершин деревьев, массив хранит целочисленные значения типа int;

Node020... Node200 – структуры вершин несбалансированных упорядоченных деревьев, переменные типа SimpleTree;

PRoots, UnbalansedTrees – массив (указатель на) указателей на корневые вершины несбалансированных упорядоченных деревьев, массив хранит значения типа SimpleTree *;

PRoot - указатель на корневую вершину упорядоченного дерева, указатель ссылается на тип SimpleTree;

Level – уровень вершины дерева, целочисленная переменная типа int;

Dim – массив целых чисел типа int;

N – количество элементов в массиве, целочисленная переменная типа int;

ReqLevel – запрашиваемый уровень дерева, целочисленная переменная типа int;

Height – высота дерева (максимальная длина ветви дерева с учетом ветвей к нулевым вершинам), целочисленная переменная типа int;

TempLevel – текущий уровень дерева в текущем рекурсивном вызове, целочисленная переменная типа int;

Array - указатель на буфер, куда помещается упорядоченный массив, сформированный из значений данных, на которые указывают поля вершин дерева, массив хранит целочисленные значения типа int;

TempIndex - ссылка на текущее значение индекса в массиве Array, по которому записываются данные из текущей вершины дерева, ссылка указывает на данные типа int;

Index – ссылка на индекс текущего заполняемого элемента в массиве PRoot, по которому записываются указатели на данные из текущего элемента массива Array, ссылка указывает на данные типа int;

TreesAmount - количество элементов в массиве UnbalancedTrees, целочисленная переменная типа int;

DataPointersAmount - количество указателей на данные в вершине дерева, глобальная константа целочисленного типа int;

ChildPointersAmount - количество указателей на потомков в вершине дерева, глобальная константа целочисленного типа int;

ValChPlacesAmount - количество знакомест для отображения одного значения данных на экране, глобальная константа целочисленного типа int;

InSpaceChPlacesAmount - количество знакомест для разделителя между значениями данных одной вершины, глобальная константа целочисленного типа int;

OutSpaceChPlacesAmount - количество знакомест для разделителя между вершинами нижнего уровня дерева, глобальная константа целочисленного типа int;

2) Используемые функции

Функция int TreeBranchesAmount(SimpleTree *PRoot)

Аргументы функции:

PRoot - указатель на корневую вершину упорядоченного дерева.

Возвращаемый результат: количество ветвей в дереве с учетом ветвей к нулевым потомкам.

Принцип работы: Рекурсивный перебор потомков вершины дерева до обнаружения нулевых потомков с накоплением количества обнаруженных конечных вершин.

Функция int BrunchLengthsSum(SimpleTree *PRoot, int Level)

Аргументы функции:

PRoot - указатель на корневую вершину упорядоченного дерева.

Возвращаемый результат: сумму длин всех ветвей дерева с учетом ветвей к нулевым потомкам.

Принцип работы: Рекурсивный перебор потомков вершины дерева до обнаружения нулевых потомков с накоплением длин ветвей.

Функция double AverageBrunchLength(SimpleTree *PRoot)

Аргументы функции:

PRoot - указатель на корневую вершину упорядоченного дерева.

Возвращаемый результат: среднюю длину ветви дерева с учетом ветвей к нулевым потомкам.

Принцип работы: Деление суммы длин всех ветвей дерева на количество ветвей дерева.

Функция int TreeNodesAmount(SimpleTree *PRoot)

Аргументы функции:

PRoot - указатель на корневую вершину упорядоченного дерева.

Возвращаемый результат: количество вершин дерева.

Принцип работы: Рекурсивный перебор потомков вершины дерева до обнаружения нулевых потомков с накоплением количества вершин дерева.

Функция int TreeHeight(SimpleTree *PRoot)

Аргументы функции:
PRoot - указатель на корневую вершину упорядоченного дерева.
Возвращаемый результат: максимальную длину ветви дерева с учетом веток к нулевым вершинам.
Принцип работы: Рекурсивный перебор потомков вершины дерева до обнаружения нулевых потомков с добавлением 1 к максимальной длине ветви при каждом шаге к вершине родителя от потомка.

Функция int MaxInt(int *Dim, int N)
Аргументы функции:
Dim - массив целых чисел.
N - размер массива.
Возвращаемый результат: индекс максимального целочисленного значения в массиве.
Принцип работы: Перебор всех значений массива с выбором максимального.

Функция int MinBranchLength(SimpleTree *PRoot)
Аргументы функции:
PRoot - указатель на корневую вершину упорядоченного дерева.
Возвращаемый результат: минимальную длину ветви дерева с учетом веток к нулевым вершинам.
Принцип работы: Рекурсивный перебор потомков вершины дерева до обнаружения нулевых потомков с добавлением 1 к минимальной длине ветви при каждом шаге к вершине родителя от потомка.

Функция int MinInt(int *Dim, int N)
Аргументы функции:
Dim - массив целых чисел.
N - размер массива.
Возвращаемый результат: индекс минимального целочисленного значения в массиве.
Принцип работы: Перебор всех значений массива с выбором минимального.

Функция void ShowTree(SimpleTree *PRoot)
Аргументы функции:
PRoot - указатель на корневую вершину упорядоченного дерева.
Возвращаемый результат: не возвращает результата.
Принцип работы: Поуровнево выводит дерево на экран.

Функция void ShowLevel(SimpleTree *PRoot, int ReqLevel, int Height, int TempLevel)
Аргументы функции:
PRoot - указатель на корневую вершину упорядоченного дерева.
ReqLevel - номер уровня, который требуется вывести.
Height – высота дерева.
TempLevel – текущий уровень дерева при рекурсивном переборе.
Возвращаемый результат: не возвращает результата.
Принцип работы: Рекурсивно продвигается к запрашиваемому уровню. В соответствии с заданным количеством знакомест выводит значения данных вершин запрашиваемого уровня с сохранением симметричности расположения потомков относительно родительской вершины. Места в запрашиваемом уровне, где отсутствуют вершины, для сохранения симметрии заполняются пробелами. Нулевые (конечные) вершины, размещенные выше последнего нулевого уровня, а также нулевые данные в ненулевых вершинах обозначаются символом '*'.

Функция void Tree2Array(int *Array, SimpleTree *PRoot, int &TempIndex)
Аргументы функции:
Array - указатель на буфер, куда помещается сформированный массив.
PRoot - указатель на корневую вершину упорядоченного дерева.
TempIndex - ссылка на текущее значение индекса в массиве Array.

Возвращаемый результат: не возвращает результата.

Принцип работы: Рекурсивный перебор потомков ненулевых вершин дерева до обнаружения нулевых потомков с заполнением массива Array значениями данных вершин дерева в возрастающем порядке и инкрементом текущего значения индекса при каждом шаге к вершине родителя от потомка.

Функция void ShowArray(int *Array, int N)

Аргументы функции:

Array - указатель на нулевой элемент сформированного функцией Tree2Array массива.

N - количество элементов в массиве Array.

Возвращаемый результат: не возвращает результата.

Принцип работы: Выводит по порядку в строку через пробел значения элементов массива.

Функция SimpleTree * Array2NormTree(int *Array, int N, SimpleTree *PRoot, int &Index)

Аргументы функции:

Array - указатель на нулевой элемент сформированного функцией Tree2Array массива.

N - количество элементов в массиве Array.

PRoot - указатель на корень дерева. Дерево располагается в массиве типа SimpleTree, нулевой элемент которого – корень.

Index – ссылка на индекс текущего заполняемого элемента в массиве PRoot.

Возвращаемый результат: указатель на сформированную вершину дерева в массиве PRoot.

Принцип работы: рекурсивное деление массива Array на равные (или почти равные) части, количество которых равно количеству потомков в вершине (в нашем случае - 3). Указатели на граничные после деления массива значения записываются в текущую вершину дерева в массиве PRoot (их количество на 1 меньше количества потомков, в нашем случае - 2). Полученные части исходного массива Array содержат значения данных соответствующих поддеревьев. Каждая из этих частей также подвергается делению на равные части и определению значений данных вершин. И так происходит до тех пор, пока все элементы массива Array не будут принадлежать соответствующим вершинам сбалансированного дерева в массиве PRoot.

Функция void ShowLab5Report(SimpleTree **UnbalancedTrees, int TreesAmount)

Аргументы функции:

UnbalancedTrees - массив указателей на корни исходных упорядоченных несбалансированных деревьев.

TreesAmount - количество исходных упорядоченных несбалансированных деревьев.

Возвращаемый результат: не возвращает результата.

Принцип работы: Для каждого несбалансированного упорядоченного дерева выполняется расчет и вывод на экран: количества ветвей дерева, суммы длин всех ветвей дерева, средней длины ветви, количества вершин дерева, максимальной длины ветви дерева, минимальной длины ветви дерева, времени балансировки, внешнего вида несбалансированного и сбалансированного деревьев. Время выполнения балансировки определяется путем засечки времени начала и конца балансировки с последующим вычитанием. Ввиду малого времени выполнения балансировки рассматриваемых экземпляров деревьев, в программе измеряется время выполнения 1000000 балансировок для каждого дерева, а результат выводится в нс.

Так же для реализации построенного алгоритма на языке Си были использованы следующие **стандартные функции**:

void *malloc(size_t size) – выделение динамической памяти.

BOOL WINAPI SetConsoleCP(_In_ UINT wCodePageID) – установка входной кодовой страницы, используемой консолью, связанной с вызывающим процессом.

BOOL WINAPI SetConsoleOutputCP(_In_ UINT wCodePageID) - установка выходной кодовой страницы, используемой консолью, связанной с вызывающим процессом.

int printf (const char * format, ...) – для вывода на экран.

clock_t clock(void) - определение времени с начала выполнения программы в мс.

int system(const char *command) – организация паузы вывода в консоль.

void free(void *memblock) – освобождение динамически выделенной памяти.

6. Пример работы программы

Тест №1

Балансировка дерева с 4-мя вершинами. Результат работы показан на рисунке 1.

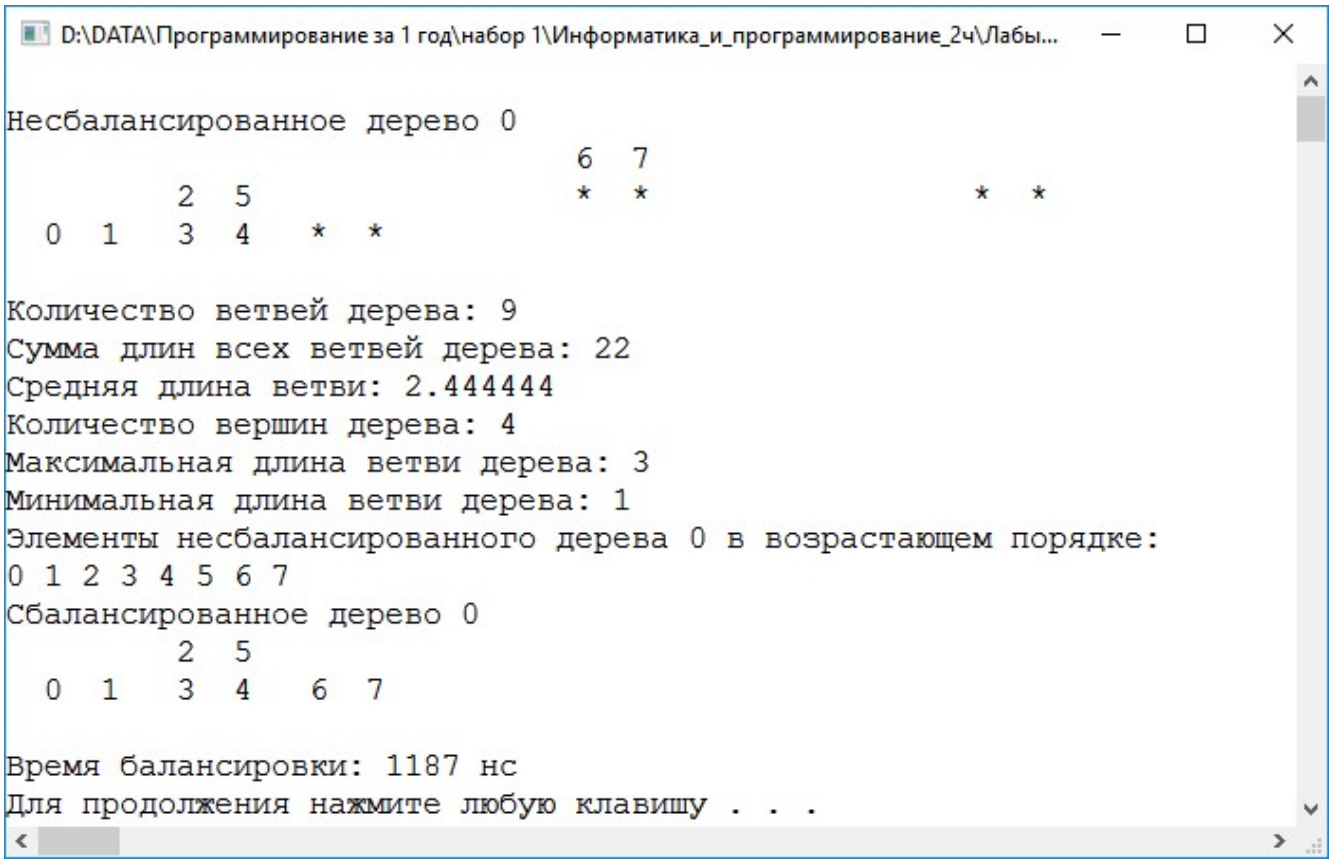
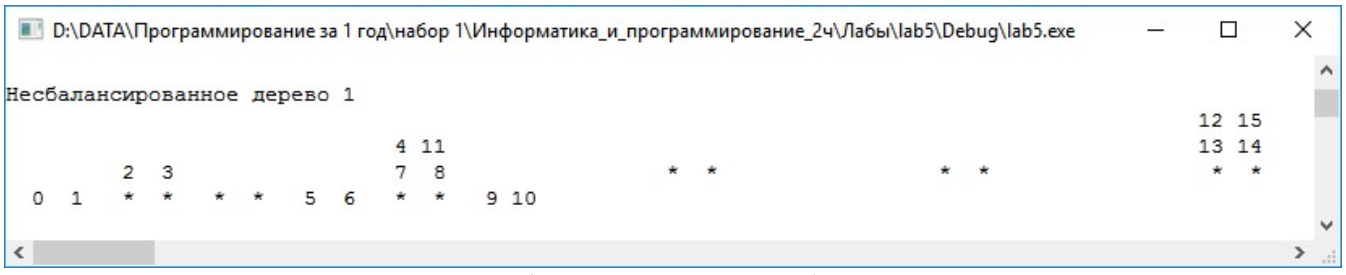


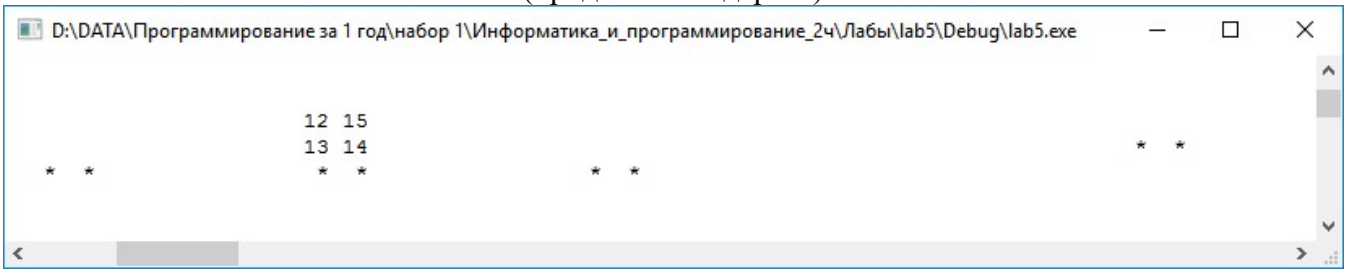
Рисунок 1. Результат работы Теста №1.

Тест №2

Балансировка дерева с 8-ю вершинами. Результат работы показан на рисунке 2.



(продолжение дерева)



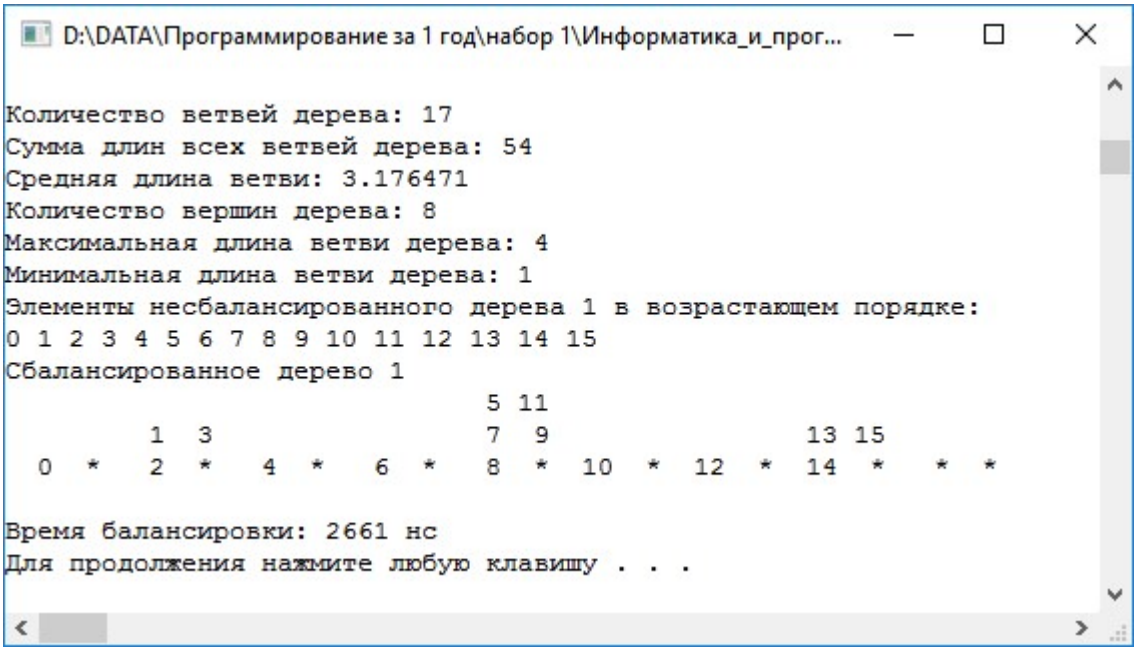
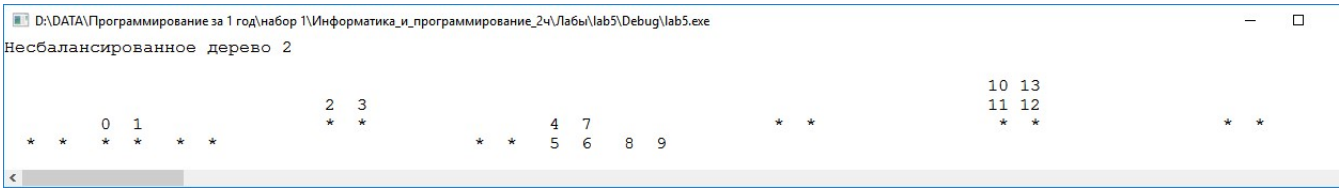


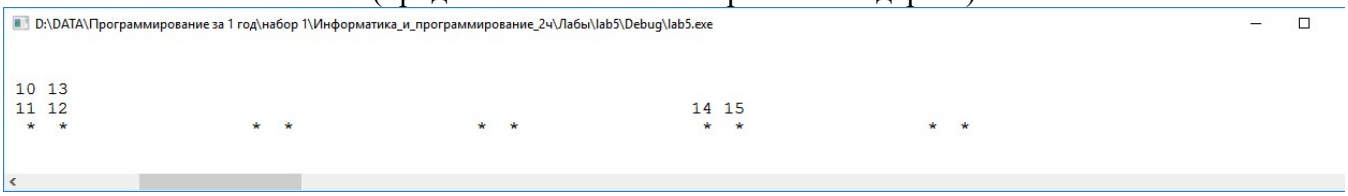
Рисунок 2 Результат работы Теста №2.

Тест №3

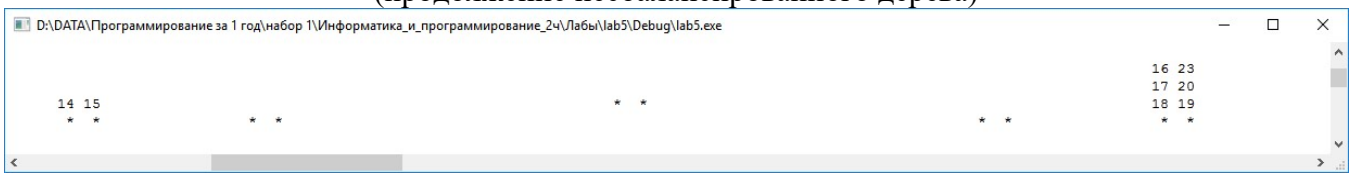
Балансировка дерева с 16-ю вершинами. Итоги (сводная таблица). Результат работы показан на рисунке 3.



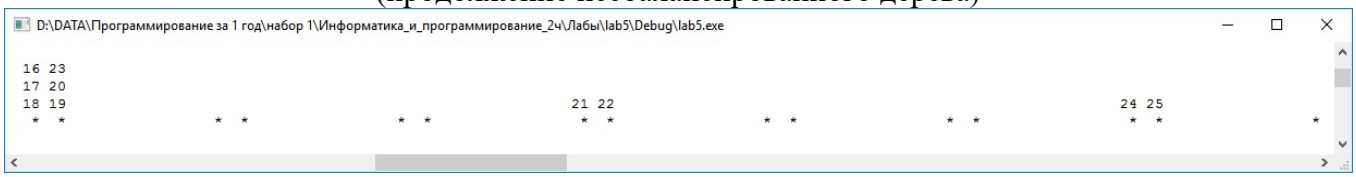
(продолжение несбалансированного дерева)



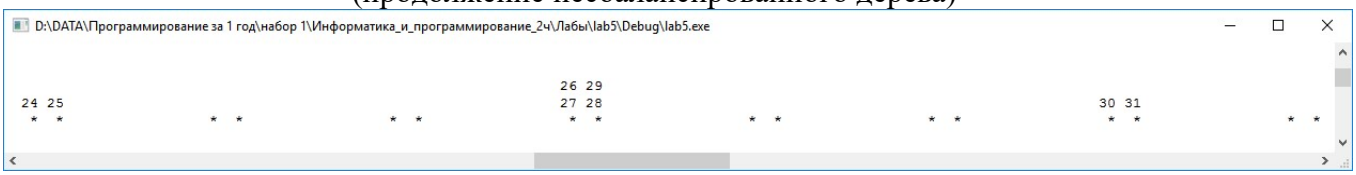
(продолжение несбалансированного дерева)



(продолжение несбалансированного дерева)



(продолжение несбалансированного дерева)



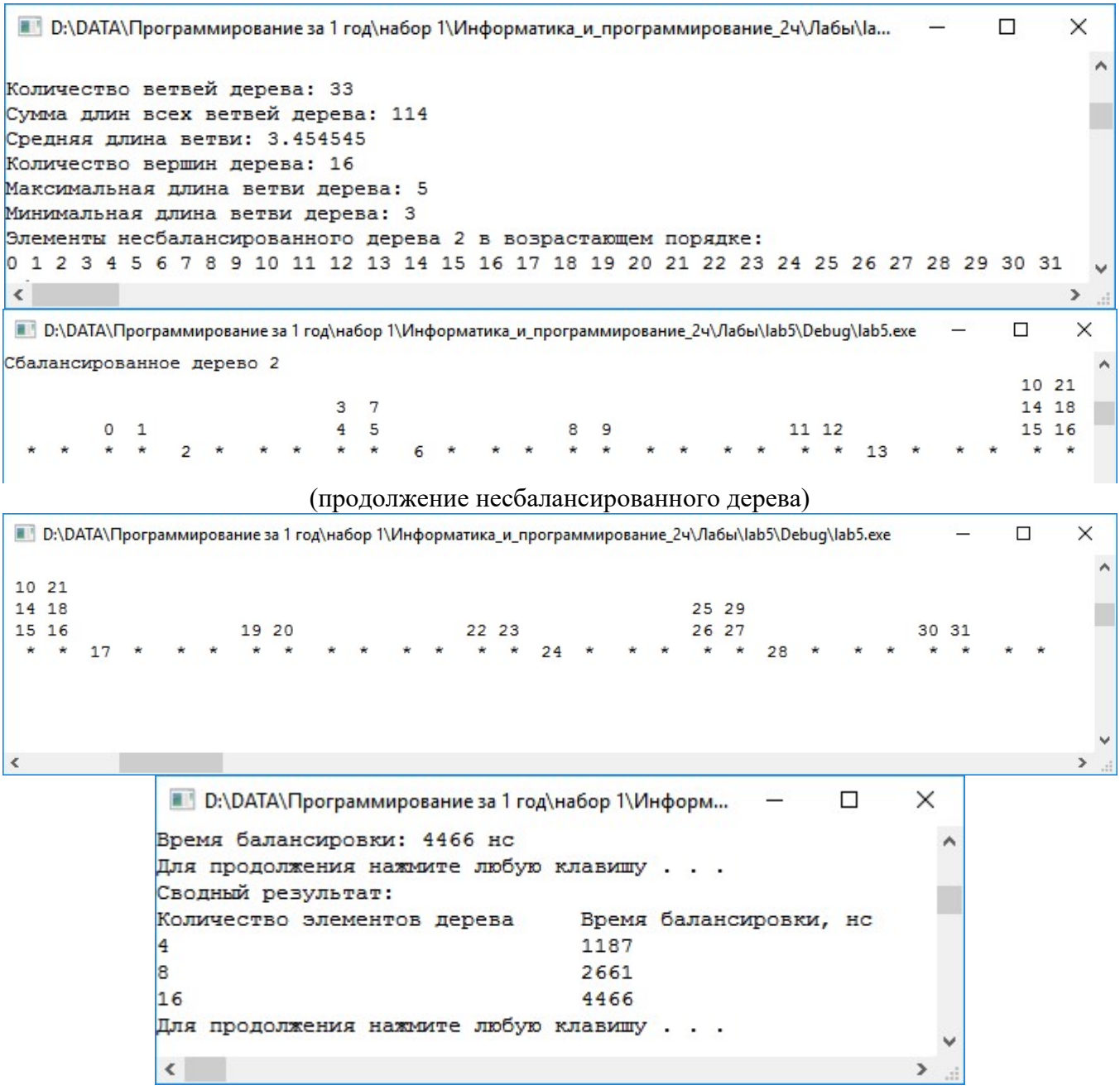


Рисунок 3 Результат работы Теста №3.

7. Выводы

В ходе выполнения лабораторной работы я познакомился с древовидными структурами данных (2-3 деревьями).

Оценка трудоемкости балансировки выполнялась для 3-х несбалансированных произвольных упорядоченных 2-3 деревьев по «грязному» времени выполнения балансировки.

N (вершин дерева)	$T_{эксперим.}$, нс	$T_{линейн.}$, нс (1187/4*N)	Отклонение, %
4	1187	1187	0
8	2661	2374	12
16	4466	4748	-6

Полученные результаты позволяют сделать предположение о линейной зависимости трудоемкости балансировки дерева от количества его вершин, что согласуется с теорией, гласящей, что полный рекурсивный обход дерева имеет линейную трудоемкость.

Репозиторий программы находится по адресу: https://github.com/broitman-eugeny/Informatics2Lab5

Приложение. Текст программы

```
//Файл lab5.h
#pragma once
#include <stdio.h>//Для printf
#include <stdlib.h>//Для system
#include <Windows.h>//Для SetConsoleCP
#include <ctime>//Для clock
const int DataPointersAmount = 2;//Количество указателей на данные в вершине дерева
const int ChildPointersAmount = DataPointersAmount+1;//Количество указателей на потомков в вершине
дерева
const int ValChPlacesAmount = 2;//Количество знакомест для отображения одного значения данных на
экране
const int InSpaceChPlacesAmount = 1;//Количество знакомест для разделителя между значениями данных
одной вершины
const int OutSpaceChPlacesAmount = 2;//Количество знакомест для разделителя между вершинами нижнего
уровня
//Структура упрощенного  2-3 дерева.
//Вершина дерева содержит DataPointersAmount указателей на объекты и ChildPointersAmount  указателей
на  поддеревья.
struct SimpleTree
{
	int *PData[DataPointersAmount];
	SimpleTree *PChildren[ChildPointersAmount];
};
//Функция вычисления количества ветвей дерева
//PRoot - указатель на корень дерева
int TreeBranchesAmount(SimpleTree *PRoot);
//Функция вычисления суммы длин всех ветвей дерева.
//PRoot - указатель на корень дерева
//Level - уровень корня дерева - 0
int BrunchLengthsSum(SimpleTree *PRoot, int Level);
//Функция вычисляет среднюю длину ветви дерева.
//Деление суммы длин ветвей на количество нулевых узлов дает среднюю
//длину ветви дерева.
//PRoot - указатель на корень дерева
double AverageBrunchLength(SimpleTree *PRoot);
//Функция вычисляет количество вершин дерева
//PRoot - указатель на корень дерева
int TreeNodesAmount(SimpleTree *PRoot);
//Функция вычисляет максимальную длину ветви дерева с учетом веток к нулевым вершинам
//PRoot - указатель на корень дерева
int TreeHeight(SimpleTree *PRoot);
//Функция возвращает индекс максимального целочисленного значения в массиве
//Dim - массив целых чисел
//N - размер массива
int MaxInt(int *Dim, int N);
//Функция вычисляет минимальную длину ветви дерева с учетом веток к нулевым вершинам
//PRoot - указатель на корень дерева
int MinBranchLength(SimpleTree *PRoot);
//Функция возвращает индекс минимального целочисленного значения в массиве
//Dim - массив целых чисел
//N - размер массива
int MinInt(int *Dim, int N);
//Функция отображает дерево
//PRoot - указатель на корень дерева
void ShowTree(SimpleTree *PRoot);
//Функция отображает один уровень дерева
//PRoot - указатель на корень дерева
//ReqLevel - номер печатаемого уровня
//Height - высота дерева, включающая ветки к нулевым вершинам
//TempLevel - текущий уровень (для поиска вершин печатаемого уровня)
void ShowLevel(SimpleTree *PRoot, int ReqLevel, int Height, int TempLevel = 0);
//Функция формирования массива из дерева
//Array - указатель на буфер, куда помещается сформированный массив. Буфер резервирует вызывающая
функция
//PRoot - указатель на корень дерева
//TempIndex - ссылка на текущее значение индекса в массиве Array
void Tree2Array(int *Array, SimpleTree *PRoot, int &TempIndex);
```

```
//Функция отображает массив int
//Array - указатель на нулевой элемент массива
//N - количество элементов в массиве
void ShowArray(int *Array, int N);
//Функция формирует из упорядоченного массива целых нормализованное дерево
//Array - указатель на нулевой элемент массива, в котором хранятся значения данных вершин дерева
//N - количество элементов в массиве Array
//PRoot - указатель на корень дерева. Дерево располагается в массиве типа SimpleTree, нулевой эле-
мент которого - корень
//Index - индекс текущего заполняемого элемента в массиве PRoot
//Возвращает индекс в массиве SimpleTree сформированной вершины дерева
SimpleTree * Array2NormTree(int *Array, int N, SimpleTree *PRoot, int &Index);
//Функция выводит на экран отчет по характеристикам сбалансированности дерева и трудоемкости балан-
сировки
//UnbalancedTrees - массив корней деревьев.
//TreesAmount - количество элементов в массиве UnbalancedTrees
void ShowLab5Report(SimpleTree **UnbalancedTrees, int TreesAmount);

//Файл Main.cpp
#include "lab5.h"
//Для древовидных структур данных предусмотреть вывод характеристик сбалансированности дере-
ва(средняя длина ветви)
//и процедуру выравнивания(балансировки).
//При выполнении работы произвести измерение зависимости «грязного» времени работы программы
//и ее трудоемкости(количества базовых операций).
//Оценить вид полученной зависимости(линейно - логарифмическая, квадратичная).
//Упрощенное  2 - 3 дерево.
//Вершина дерева содержит два указателя на объекты и три  указателя на  поддеревья. Данные  в  дере-
ве  упорядочены.
//Нелистовые вершины содержат два поля, указывающие на диапазон значений в их поддеревьях.
//Значение первого поля строго больше наибольшего значения в левом поддереве и меньше
//или равно наименьшему значению в центральном поддереве;
//аналогично, значение второго поля строго больше наибольшего значения в центральном поддереве
//и меньше или равно, чем наименьшее значение в правом поддереве.
//Эти нелистовые вершины используются для направления функции поиска к нужному поддереву и,
//в конечном итоге, к нужному листу.
void main()
{
	//Массив ключевых значений
	int Keys[] =
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31};
	//Несбалансированные деревья
	//Дерево 0
	//2-й уровень дерева 0
	SimpleTree Node020 = { { &Keys[0], &Keys[1] },{ NULL,NULL,NULL } };//Потомок вершины Node010
	SimpleTree Node021 = { { &Keys[3], &Keys[4] },{ NULL,NULL,NULL } };// -//-
	//1-й уровень дерева 0
	SimpleTree Node010 = { { &Keys[2], &Keys[5] },{ &Node020,&Node021,NULL } };//Потомок вершины
Node000
	//0-й уровень дерева 0
	SimpleTree Node000 = { { &Keys[6], &Keys[7] },{ &Node010,NULL,NULL } };//Корень
	//Дерево 1
	//3-й уровень дерева 1
	SimpleTree Node130 = { { &Keys[0], &Keys[1] },{ NULL,NULL,NULL } };//Потомок вершины Node120
	SimpleTree Node131 = { { &Keys[5], &Keys[6] },{ NULL,NULL,NULL } };//Потомок вершины Node121
	SimpleTree Node132 = { { &Keys[9], &Keys[10] },{ NULL,NULL,NULL } };// -//-
	//2-й уровень дерева 1
	SimpleTree Node120 = { { &Keys[2], &Keys[3] },{ &Node130,NULL,NULL } };//Потомок вершины
Node110
	SimpleTree Node121 = { { &Keys[7], &Keys[8] },{ &Node131,NULL,&Node132 } };// -//-
	//1-й уровень дерева 1
	SimpleTree Node110 = { { &Keys[4], &Keys[11] },{ &Node120,&Node121,NULL } };//Потомок вершины
Node100
	SimpleTree Node111 = { { &Keys[13], &Keys[14] },{ NULL,NULL,NULL } };// -//-
	//0-й уровень дерева 1
	SimpleTree Node100 = { { &Keys[12], &Keys[15] },{ &Node110,&Node111,NULL } };//Корень
	//4-й уровень дерева 2
	SimpleTree Node240 = { { &Keys[5], &Keys[6] },{ NULL,NULL,NULL } };//Потомок вершины Node231
	SimpleTree Node241 = { { &Keys[8], &Keys[9] },{ NULL,NULL,NULL } };// -//-
```

```
	//3-й уровень дерева 2
	SimpleTree Node230 = { { &Keys[0], &Keys[1] },{ NULL,NULL,NULL } };//Потомок вершины Node220
	SimpleTree Node231 = { { &Keys[4], &Keys[7] },{ NULL,&Node240,&Node241 } };// -//-
	//2-й уровень дерева 2
	SimpleTree Node220 = { { &Keys[2], &Keys[3] },{ &Node230,NULL,&Node231 } };//Потомок вершины
Node210
	SimpleTree Node221 = { { &Keys[11], &Keys[12] },{ NULL,NULL,NULL } };// -//-
	SimpleTree Node222 = { { &Keys[14], &Keys[15] },{ NULL,NULL,NULL } };// -//-
	SimpleTree Node223 = { { &Keys[18], &Keys[19] },{ NULL,NULL,NULL } };//Потомок вершины
Node011
	SimpleTree Node224 = { { &Keys[21], &Keys[22] },{ NULL,NULL,NULL } };// -//-
	SimpleTree Node225 = { { &Keys[24], &Keys[25] },{ NULL,NULL,NULL } };//Потомок вершины
Node012
	SimpleTree Node226 = { { &Keys[27], &Keys[28] },{ NULL,NULL,NULL } };// -//-
	SimpleTree Node227 = { { &Keys[30], &Keys[31] },{ NULL,NULL,NULL } };// -//-
	//1-й уровень дерева 2
	SimpleTree Node210 = { { &Keys[10], &Keys[13] },{ &Node220,&Node221,&Node222 } };//Потомок
вершины Node200 (корня)
	SimpleTree Node211 = { { &Keys[17], &Keys[20] },{ NULL,&Node223,&Node224 } };// -//-
	SimpleTree Node212 = { { &Keys[26], &Keys[29] },{ &Node225,&Node226,&Node227 } };// -//-
	//0-й уровень дерева 2 (корень)
	SimpleTree Node200 = { { &Keys[16], &Keys[23] },{ &Node210,&Node211,&Node212 } };
	SimpleTree *PRoots[] = { &Node000,&Node100,&Node200 };
	ShowLab5Report(PRoots, 3);
}

//Файл lab5.cpp
#include "lab5.h"
//Функция вычисления количества ветвей дерева
//PRoot - указатель на корень дерева
int TreeBranchesAmount(SimpleTree *PRoot)
{
	if (PRoot == NULL)//Дошли до конца ветви
	{
		return 1;//Добавить одну ветку
	}
	int BranchesAmount = 0;
	for (int i = 0; i < ChildPointersAmount; i++)//По всем потомкам корня
	{
		BranchesAmount += TreeBranchesAmount(PRoot->PChildren[i]);
	}
	return BranchesAmount;
}
//Функция вычисления суммы длин всех ветвей дерева.
//PRoot - указатель на корень (вершину) дерева
//Level - уровень корня (вершины) дерева - 0
int BrunchLengthsSum(SimpleTree *PRoot, int Level)
{
	if (PRoot == NULL)//Дошли до конца ветви
	{
		return Level ;//Учитываем ветки к нулевым вершинам
	}
	int LengthsSum = 0;
	for (int i = 0; i < ChildPointersAmount; i++)//По всем потомкам корня
	{
		LengthsSum += BrunchLengthsSum(PRoot->PChildren[i], Level + 1);
	}
		return LengthsSum;
}
//Функция вычисляет среднюю длину ветви дерева.
//Деление суммы длин ветвей на количество нулевых узлов дает среднюю
//длину ветви дерева.
//PRoot - указатель на корень дерева
double AverageBrunchLength(SimpleTree *PRoot)
{
	int LengthsSum = BrunchLengthsSum(PRoot, 0);//Суммарная длина всех ветвей
	int BranchesAmount = TreeBranchesAmount(PRoot);//Количество ветвей
	return (double)LengthsSum/ (double)BranchesAmount;
}
```

```
//Функция вычисляет количество вершин дерева
//PRoot - указатель на корень дерева
int TreeNodesAmount(SimpleTree *PRoot)
{
	if (PRoot == NULL)//Дошли до конца ветви
	{
		return 0;
	}
	int NodesAmount = 1;
	for (int i = 0; i < ChildPointersAmount; i++)//По всем потомкам корня
	{
		NodesAmount += TreeNodesAmount(PRoot->PChildren[i]);
	}
	return NodesAmount;
}
//Функция вычисляет максимальную длину ветви дерева с учетом веток к нулевым вершинам
//PRoot - указатель на корень дерева
int TreeHeight(SimpleTree *PRoot)
{
	if (PRoot == NULL)//Дошли до конца ветви
	{
		return 0;
	}
	int Heights[ChildPointersAmount];
	for (int i = 0; i < ChildPointersAmount; i++)//По всем потомкам корня
	{
		Heights[i] = TreeHeight(PRoot->PChildren[i]);
	}
	return Heights[MaxInt(Heights, ChildPointersAmount)]+1;
}
//Функция возвращает индекс максимального целочисленного значения в массиве
//Dim - массив целых чисел
//N - размер массива
int MaxInt(int *Dim, int N)
{
	int MaxInd = 0;
	for (int i = 1; i < N; i++)
	{
		if (Dim[i] > Dim[MaxInd])
		{
			MaxInd = i;
		}
	}
	return MaxInd;
}
//Функция вычисляет минимальную длину ветви дерева с учетом веток к нулевым вершинам
//PRoot - указатель на корень дерева
int MinBranchLength(SimpleTree *PRoot)
{
	if (PRoot == NULL)//Дошли до конца ветви
	{
		return 0;
	}
	int Heights[ChildPointersAmount];
	for (int i = 0; i < ChildPointersAmount; i++)//По всем потомкам корня
	{
		Heights[i] = TreeHeight(PRoot->PChildren[i]);
	}
	return Heights[MinInt(Heights, ChildPointersAmount)] + 1;
}
//Функция возвращает индекс минимального целочисленного значения в массиве
//Dim - массив целых чисел
//N - размер массива
int MinInt(int *Dim, int N)
{
	int MinInd = 0;
	for (int i = 1; i < N; i++)
	{
		if (Dim[i] < Dim[MinInd])
```

```
		{
			MinInd = i;
		}
	}
	return MinInd;
}
//Функция отображает дерево
//PRoot - указатель на корень дерева
void ShowTree(SimpleTree *PRoot)
{
	int Height = TreeHeight(PRoot);//Высота дерева с учетом нулевых веток
	printf("\n");
	for (int i = 0; i < Height; i++)
	{
		ShowLevel(PRoot,i, Height);//Отображает один уровень дерева
		printf("\n");
	}
}
//Функция отображает один уровень дерева
//PRoot - указатель на корень дерева
//ReqLevel - номер печатаемого уровня
//Height - высота дерева, включающая ветки к нулевым вершинам
//TempLevel - текущий уровень (для поиска вершин печатаемого уровня)
void ShowLevel(SimpleTree *PRoot, int ReqLevel, int Height, int TempLevel)
{
	char FormatS[5];
	FormatS[0] = '%';
	FormatS[1] = (ValChPlacesAmount>9) ? '0' + ValChPlacesAmount / 10 : '0' + ValChPlacesAmount;
	FormatS[2] = (ValChPlacesAmount>9) ? '0' + ValChPlacesAmount % 10 : 's';
	FormatS[3] = (ValChPlacesAmount>9) ? 's' : '\0';
	FormatS[4] = '\0';
	int Temp1 = OutSpaceChPlacesAmount / 2,//Временная переменная 1
		Temp2 = OutSpaceChPlacesAmount - Temp1,//Временная переменная 2
		SpacesBeforeNode = Temp1,//Количество пробелов, которые нужно напечатать перед верши-
ной текущего уровня
		SpacesAfterNode = Temp2,//Количество пробелов, которые нужно напечатать после вершины
текущего уровня
		W = ValChPlacesAmount*DataPointersAmount + (DataPointersAmount -
1)*InSpaceChPlacesAmount;//Количество знакомест, занимаемых одной вершиной дерева
	if (TempLevel < ReqLevel)//Пока не достигли требуемого уровня
	{
		if (PRoot != NULL)//Не достигли конца ветви
		{
			for (int i = 0; i < ChildPointersAmount; i++)
			{
				ShowLevel(PRoot->PChildren[i], ReqLevel, Height, TempLevel + 1);//Переход
к следующему уровню
			}
		}
		else//достигли конца ветви. Необходимо напечатать пробелы в запрашиваемом уровне (тек-
стовый курсор находится в запрашиваемом уровне), чтобы восполнить отсутствующие элементы из-за нуле-
вых вершин в текущем уровне и сохранить симметрию дерева
		{
			if (ChildPointersAmount % 2 == 0)//Если количество потомков четное
			{
				for (int i = Height - 2; i >= TempLevel; i--)
				{
					Temp1 = SpacesBeforeNode;//Значение со следующего уровня
					Temp2 = SpacesAfterNode;//Значение со следующего уровня
					//Количество пробелов, которые нужно напечатать перед вершиной
					SpacesBeforeNode = ChildPointersAmount*(W + Temp1 + Temp2) / 2 - W
/ 2;

					//Количество пробелов, которые нужно напечатать после вершины
					SpacesAfterNode= SpacesBeforeNode+ W / 2-(W-W/2);//Т.к. W м.б. не-
четным и W/2 < половины W
				}
			}
			else//Если количество потомков нечетное
			{
```

```
                        for (int i = Height - 2; i >= TempLevel; i--)
                        {
                                Temp1 = SpacesBeforeNode;//Значение со следующего уровня
                                Temp2 = SpacesAfterNode;//Значение со следующего уровня
                                //Количество пробелов, которые нужно напечатать перед вершиной
                                SpacesBeforeNode = (ChildPointersAmount-1)*(W + Temp1 + Temp2) / 2
+ Temp1;
                                //Количество пробелов, которые нужно напечатать после вершины
                                SpacesAfterNode = SpacesBeforeNode- Temp1+ Temp2;
                        }
                }
                //Печать пробелов перед вершиной
                for (int i = 0; i < SpacesBeforeNode; i++)
                {
                        printf(" ");
                }
                for (int i = 0; i < DataPointersAmount; i++)//Печать пробелов чтобы сохранить
симметрию
                {
                        printf(FormatS, " ");
                        if (i < DataPointersAmount - 1)//Не последнее значение вершины
                        {
                                for (int j = 0; j < InSpaceChPlacesAmount; j++)
                                {
                                        printf(" ");//Пробелы между значениями одной вершины
                                }
                        }
                }
                //Печать пробелов после вершины
                for (int i = 0; i < SpacesAfterNode; i++)
                {
                        printf(" ");
                }
        }
    }
    else//Достигли требуемого уровня
    {
        if (ChildPointersAmount % 2 == 0)//Если количество потомков четное
        {
                for (int i = Height - 2; i >= TempLevel; i--)
                {
                        Temp1 = SpacesBeforeNode;
                        Temp2 = SpacesAfterNode;
                        //Количество пробелов, которые нужно напечатать перед вершиной
                        SpacesBeforeNode = ChildPointersAmount*(W + Temp1 + Temp2) / 2 - W / 2;
                        //Количество пробелов, которые нужно напечатать после вершины
                        SpacesAfterNode = SpacesBeforeNode + W / 2 - (W - W / 2);//Т.к. W м.б.
нечетным и W/2 < половины W
                }
        }
        else//Если количество потомков нечетное
        {
                for (int i = Height - 2; i >= TempLevel; i--)
                {
                        Temp1 = SpacesBeforeNode;
                        Temp2 = SpacesAfterNode;
                        //Количество пробелов, которые нужно напечатать перед вершиной
                        SpacesBeforeNode = (ChildPointersAmount - 1)*(W + Temp1 + Temp2) / 2 +
Temp1;
                        //Количество пробелов, которые нужно напечатать после вершины
                        SpacesAfterNode = SpacesBeforeNode - Temp1 + Temp2;
                }
        }
        //Печать пробелов перед вершиной
        for (int i = 0; i < SpacesBeforeNode; i++)
        {
                printf(" ");
        }
        char Format[5];
```

```
			Format[0] = '%';
			Format[1] = (ValChPlacesAmount>9)?'0'+ ValChPlacesAmount /10: '0' + ValChPlacesAmount;
			Format[2] = (ValChPlacesAmount>9) ? '0' + ValChPlacesAmount % 10 : 'd';
			Format[3] = (ValChPlacesAmount>9) ? 'd' : '\0';
			Format[4] = '\0';
			for (int i = 0; i < DataPointersAmount; i++)//Печать значений вершины
			{
				if (PRoot != NULL)//Вершина имеется (не конец ветви)
				{
					if (PRoot->PData[i] != NULL)//Если значение в вершине имеется
					{
						printf(Format, *(PRoot->PData[i]));
					}
					else
					{
						printf(FormatS, "*");//Нет значения в вершине
					}
				}
				else//Нулевая вершина (конец ветви)
				{
					printf(FormatS, "*");
				}
				if (i < DataPointersAmount - 1)//Не последнее значение вершины
				{
					for (int j = 0; j < InSpaceChPlacesAmount; j++)
					{
						printf(" ");//Пробелы между значениями одной вершины
					}
				}
			}
			//Печать пробелов после вершины
			for (int i = 0; i < SpacesAfterNode; i++)
			{
				printf(" ");
			}
		}
	}
	//Функция формирования массива из дерева
	//Array - указатель на буфер, куда помещается сформированный массив. Буфер резервирует вызывающая функция
	//PRoot - указатель на корень дерева
	//TempIndex - ссылка на текущее значение индекса в массиве Array
	void Tree2Array(int *Array, SimpleTree *PRoot, int &TempIndex)
	{
		if (PRoot != NULL)
		{
			for (int i = 0, N= ChildPointersAmount-1; i < N; i++)
			{
				Tree2Array(Array, PRoot->PChildren[i], TempIndex);
				if (PRoot->PData[i] != NULL)
				{
					Array[TempIndex++] = *(PRoot->PData[i]);
				}
			}
			Tree2Array(Array, PRoot->PChildren[ChildPointersAmount - 1], TempIndex);//Т.к. за последним потомком в вершине нет данных
		}
	}
	//Функция отображает массив int
	//Array - указатель на нулевой элемент массива
	//N - количество элементов в массиве
	void ShowArray(int *Array, int N)
	{
		printf("\n");
		for (int i = 0; i < N; i++)
		{
			printf("%d ", Array[i]);
		}
	}
```

```
//Функция формирует из упорядоченного массива целых нормализованное дерево
//Array - указатель на нулевой элемент массива, в котором хранятся значения данных вершин дерева
//N - количество элементов в массиве Array
//PRoot - указатель на корень дерева. Дерево располагается в массиве типа SimpleTree, нулевой элемент которого - корень
//Index - индекс текущего заполняемого элемента в массиве PRoot
//Возвращает указатель на сформированную вершину дерева в массиве PRoot
SimpleTree * Array2NormTree(int *Array, int N, SimpleTree *PRoot, int &Index)
{
	if (N > 0)//Элементы на участке разбиения есть
	{
		int NodeIndexes[DataPointersAmount];//Индексы в массиве Array значений вершины
		NodeIndexes[0] = (N - 1) / ChildPointersAmount;//Разбиваем участок на ChildPointersAmount подучастков
		int TempIndex = Index;//Для развязки с Index, который меняется во время рекурсивных вызовов
		PRoot[TempIndex].PData[0] = &(Array[NodeIndexes[0]]);
		for (int i = 1; i < DataPointersAmount; i++)//Для каждого значения данных вершины дерева
		{
			//Вычисляем индекс значения данных в массиве Array
			NodeIndexes[i] = NodeIndexes[i - 1] + 1 + NodeIndexes[0];
			if (i <= N - 1)//Если индекс не превосходит максимально возможное значение на подучастке (есть элементы для записи)
			{
				//Записываем указатель на данные в вершину
				PRoot[TempIndex].PData[i] = &(Array[NodeIndexes[i]]);
			}
			else//Элементы на подучастке закончились
			{
				PRoot[TempIndex].PData[i] = NULL;
			}
		}
		int MaxChIndex = ChildPointersAmount - 1;//Максимальный индекс потомка одной вершины дерева
		for (int i = 0; i < MaxChIndex; i++)//Для всех потомков, кроме последнего, т.к. в его поддереве может оказаться другое количество вершин
		{
			//Рекурсивно формируем вершину потомка и получаем указатель на нее
			PRoot[TempIndex].PChildren[i] = Array2NormTree(&(Array[i*(NodeIndexes[0]+1)]), NodeIndexes[0], PRoot, ++Index);
			if (PRoot[TempIndex].PChildren[i] == NULL)
			{
				Index--;//Т.к. нулевая вершина не формируется в массиве
			}
		}
		//Рекурсивно формируем вершину последнего потомка и получаем указатель на нее
		PRoot[TempIndex].PChildren[MaxChIndex] = Array2NormTree(&(Array[MaxChIndex*(NodeIndexes[0] + 1)]), N-1-NodeIndexes[DataPointersAmount-1], PRoot, ++Index);
		if (PRoot[TempIndex].PChildren[MaxChIndex] == NULL)
		{
			Index--;//Т.к. нулевая вершина не формируется в массиве
		}
		//Возвращаем указатель на сформированную вершину родительской вершине
		return &(PRoot[TempIndex]);
	}
	//Элементы на участке разбиения массива Array закончились
	return NULL;
}
//Функция выводит на экран отчет по характеристикам сбалансированности дерева и трудоемкости балансировки
//UnbalancedTrees - массив корней деревьев.
//TreesAmount - количество элементов в массиве UnbalancedTrees
void ShowLab5Report(SimpleTree **UnbalancedTrees, int TreesAmount)
{
	int *NodesAmount;
	int *Time;
	NodesAmount = (int*)malloc(sizeof(int)*TreesAmount);
```

```
	Time = (int*)malloc(sizeof(int)*TreesAmount);
	SetConsoleCP(1251);//Ввод русских букв
	SetConsoleOutputCP(1251);//Вывод русских букв
	for (int i = 0; i < TreesAmount; i++)
	{
		printf("\nНесбалансированное дерево %d",i);
		ShowTree(UnbalancedTrees[i]);
		printf("\nКоличество ветвей дерева: %d", TreeBranchesAmount(UnbalancedTrees[i]));
		printf("\nСумма длин всех ветвей дерева: %d", BrunchLengthsSum(UnbalancedTrees[i],
0));
		printf("\nСредняя длина ветви: %f", AverageBrunchLength(UnbalancedTrees[i]));
		NodesAmount[i] = TreeNodesAmount(UnbalancedTrees[i]);//Количество вершин в дереве
		printf("\nКоличество вершин дерева: %d", NodesAmount[i]);
		printf("\nМаксимальная длина ветви дерева: %d", TreeHeight(UnbalancedTrees[i]));
		printf("\nМинимальная длина ветви дерева: %d", MinBranchLength(UnbalancedTrees[i]));
		int *Array = (int *)malloc(sizeof(int)*NodesAmount[i]*DataPointersAmount);
		int Index = 0;
		Tree2Array(Array, UnbalancedTrees[i], Index);//формирование массива из дерева
		printf("\nЭлементы несбалансированного дерева %d в возрастающем порядке:", i);
		ShowArray(Array, NodesAmount[i]*DataPointersAmount);
		SimpleTree *PRoot = (SimpleTree
*)malloc(sizeof(SimpleTree)*NodesAmount[i]*DataPointersAmount);//Домножаем на DataPointersAmount,
т.к. в вырожденном случае в вершине может оказаться всего одно значение данных
		Time[i] = clock();//мс, время с начала выполнения программы
		for (int j = 0; j < 1000000; j++)
		{
			Index = 0;
			Tree2Array(Array, UnbalancedTrees[i], Index);//формирование массива из дерева
			//printf("\nЭлементы несбалансированного дерева %d в возрастающем порядке:", i);
			//ShowArray(Array, NodesAmount*DataPointersAmount);
			Index = 0;
			Array2NormTree(Array, NodesAmount[i]*DataPointersAmount, PRoot, Index);
		}
		Time[i] = clock() - Time[i];
		printf("\nСбалансированное дерево %d", i);
		ShowTree(PRoot);
		printf("\nВремя балансировки: %d нс", Time[i]);
		printf("\n");
		system("pause");
		free(PRoot);
		free(Array);
	}
	printf("Сводный результат:\n");
	printf("Количество элементов дерева\tВремя балансировки, нс\n");
	for (int i = 0; i < TreesAmount; i++)
	{
		printf("%d\t\t\t\t%d\n",NodesAmount[i], Time[i]);
	}
	free(NodesAmount);
	free(Time);
	system("pause");
}
```