Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ « Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №6

Группа ИУ7-35Б

Тема работы **Обработка деревьев**

Студент Баранов Николай Алексеевич

Преподаватель Барышникова Марина Юрьевна

2022 г.

Цель работы: получить навыки применения двоичных деревьев, реализовать собственные операции над деревьями: обход деревьев, включение, исключение и поиск узлов.

1. Описание условия задачи

Построить двоичное дерево поиска, в узлах которого содержится имя и атрибуты файла. Вывести его на экран. Реализовать основные операции работы с деревом: обход, включение, исключение и поиск узлов. Реализовать удаление файлов, обращение к которым происходило до определённой даты. Сравнить эффективность алгоритмов сортировки и поиска в зависимости от высоты деревьев и степени их ветвления.

2. Описание ТЗ

2.1. Описание исходных данных и результатов.

При чтении имени файла программа принимает на вход произвольную непустую строку.

При чтении числа программа считывает число в указанном диапазоне.

Выбор действия происходит в меню, где вводится число – номер действия.

При удалении вершин по дате выводится число успешно удаленных вершин.

Дерево при необходимости сохраняется в файле в виде графа на языке описания графов DOT. Далее с помощью утилиты graphviz граф можно визуализировать.

При сравнении эффективности выводится общее время работы и затраченная память на хранение дерева.

2.2. Описание задачи, реализуемой программой.

Программа позволяет ввести информацию о файлах и сохранить их в двоичном дереве поиска. Также программа позволяет выполнять базовые операции над деревом, а также сохранить дерево в файл.

2.3. Способ обращения к программе.

Запуск программы:

./app.exe

В этом случае программа начнёт выполнять свою задачу. Ввод осуществляется вручную с клавиатуры.

2.4. Описание возможных аварийных ситуаций и ошибок пользователя.

2.4.1. Переполнение буфера. Возникает, если строка при чтении оказалась длиннее чем максимальное количество цифр в числе (вместо этого также может случится ошибка выхода числа из диапазона, если число пометилось в буфер).

2.4.2. Неожиданный символ. Возникает, если в строке присутствуют символы, не предусмотренные программой, или из-за неправильного формата входных данных (например, лишний пробел, лишняя точка в числе, пробел посреди числа и т. д.).

2.4.3. Число не входит в диапазон. Возникает, если пользователь ввёл число, которое не входит в диапазон допустимых значений.

2.4.4. Ошибка при выделении памяти. Практически не зависит от действий пользователя. Возникает только при неудачной попытке выделить память на куче, поэтому вероятность возникновения крайне мала.

3. Описание внутренних структур данных.

Для дерева используются 2 структуры:

```
struct tree
{
    node_tree *start;
    size_t size;
};

struct node_tree
{
    char *name;
    date day;
    node_tree *left, *right;
};
```

tree – дерево.
Её поля:
size – целое число. Хранит размер дерева.
start – указатель на структуру node_tree. Хранит вершину дерева.
node_tree – лист дерева.
Её поля:
name – указатель на строку. Содержит имя файла.
day – структура типа date, содержащая 3 числа. Хранит день, месяц и год.
left, right – указатели на node_tree. Содержат левого и правого потомков листа.

4. Описание алгоритма.

При вставке сначала выделяется память под элемент, затем туда записываются данные, далее идёт поиск места для вставки: в зависимости от значения рассматриваемого узла переходим либо в левого потомка (если значение имени текущего узла больше), либо в правого (если меньше), либо перезаписываем значение текущего узла, а память под новый элемент освобождаем и заканчиваем обход (если имена равны, при этом размер дерева не меняется). Если в нужной стороне нулевой указатель, то заменяем его на указатель на добавляемый элемент и завершаем обход, отмечая, что размер дерева увеличился на 1 элемент.

Поиск похож на вставку, только при совпадении возвращаем найденное значение, а при нулевом указатели возвращаем некорректную дату (-1.-1.-1).

При обходе сначала записываем элемент в стек, затем переходим к левому потомку, затем к правому, далее возвращаемся к предыдущему в стеке. В

зависимости от режима работы действие над самим листом будет произведено либо перед просмотром левого потомка, либо после просмотра правого потомка, либо между просмотрами двух потомков.

При удалении по имени сначала ищем нужный лист, а когда находим, то изначально смотрим на потомков. Если их нет, то просто освобождаем память, а у родителя зануляем указатель. Если есть только один, то меняем у родителя указатель на потомок удаляемого и освобождаем память. иначе в правом поддереве ищем лист с самым «маленьким» именем, меняем данные удаляемого на данные найденного, затем память из под найденного элемента удаляем.

При удалении по дате просто обходим дерево, а после просмотра потомков, если дата меньше или равна нужной, удаляем.

Ожидаемая сложность вставки, поиска и удаления по имени O(logn), максимальная сложность O(n). Это происходит из-за того, что дерево не сбалансированное, из-за чего при добавлении элементов в порядке возрастания/убывания все полезные свойства дерева теряются. При обходе в каждый узел мы попадаем не более 3 раз (когда приходим, после просмотра левого потомка, после просмотра правого потомка), поэтому максимальная сложность O(n). При удалении по дате сложность получится O(nlogn), так как перемножаются сложности удаления и обхода.

5. Сравнительный анализ разных способов хранения и перемножения.

Для замера времени и памяти программа запускала 1000000 итераций, на каждой из которых измерялось время работы поиска случайного элемента, не имеющего потомков, и обхода. Все деревья имели 15 элементов.

Деревья, которые использовались для измерения эффективности, находятся в Приложении 1.

В таблицах ниже приведены результаты работы программы для разного распределения элементов по дереву.

Вид дерева	Время работы, мкс (обход)	Время работы, мкс (поиск)	Используемая память, байт
Правостороннее	932179	523027	616
Идеально сбалансированное	920531	436969	616
Со случайным порядком добавления элементов	920576	438688	616

Здесь видно, что для правостороннего дерева поиск работал дольше всего. Это связано с тем, что в худшем случае поиск доходит до самого дна, перебирая все возможные элементы, а при измерении для поиска выбирались только элементы без потомков для получения времени работы для наихудших случаев, а в этом дереве есть только один такой элемент. Поэтому время поиска в таком дереве превышает чуть ли не на 20% время поиска в других деревьях. Поиск в идеально сбалансированном дереве оказался немного быстрее, чем поиск в

дереве со случайным порядком добавления элементов. Однако на деревьях большей размерности разница будет больше, в данном случае огромную часть времени отнимает вызов самой функции. Также из-за размера разница в высоте идеально сбалансированного дерева и используемого дерева со случайным порядком добавления элементов составляет всего 3, причём самый низкий элемент без потомков в последнем находится на такой же высоте, на которой находятся все элементы без потомков в идеально сбалансированном дереве. Также огромное влияние могли оказать случайные выбросы. Однако этот пример всё же дает понять, что высота влияет на скорость поиска. Степень ветвления никак не может повлиять на сложность поиска, так как при поиске всегда выбирается только один из потомков, а возврат к родителю никогда не происходит.

Для обхода время работы примерно одинаковое и не зависит ни от степени ветвления дерева, ни от его высоты, так как цикл при обходе всегда делает ровно 3n итераций, где n – число элементов в дереве. Небольшие отличия во времени работы объясняются случайными выбросами.

Размер используемой памяти остаётся неизменным, так как он зависит только от количества элементов в дереве.

6. Набор тестов.

Тесты описаны только для ввода значений. Отсутствие выходных данных означает в данном случае что тест позитивный.

В меню проверяется только то что после каждого действия запускается нужная функция.

Входные данные	**Выходные данные**	**Что и где проверялось**
1		Ввод минимального целого числа
3		Ввод целого числа не на границе
2022		Ввод целого числа максимального размера
000003		Ввод целого числа с ведущими нулями (без переполнения буфера)
filename		Ввод имени файла
	Ошибка. Пустой ввод.	Ввод пустой строки при вводе числа
	Ошибка. Не удалось считать строку.	Ввод пустой строки при вводе имени файла
what	Ошибка. Встречен неожиданный символ.	Ввод не числа, если нужно целое число

1234567890	Ошибка. Переполнение буфера при вводе.	Переполнение буфера (при чтении целого числа)
2023	Ошибка. Число выходит за пределы допустимого диапазона.	Ввод числа, не входящего в допустимый диапазон чисел.

7. Выводы по проделанной работе.

В ходе выполнения работы я изучил операции с деревом. Ещё я реализовал двоичное несбалансированное дерево поиска, для которого также реализовал удаление нескольких элементов по дате. Также я измерил эффективность работы обхода дерева и поиска элемента.

8. Ответы на вопросы.

8.1. Что такое дерево?

Дерево – структура данных, используемая для представления иерархических связей, имеющих отношение «один ко многим».

8.2. Как выделяется память под представление деревьев?

Для дерева удобнее всего выделять память отдельно под каждый лист дерева. Если число потомков ограничено, то можно выделить в виде массива, но в таком случае в случае отсутствия балансировки дерева этот способ может оказаться затратным по памяти.

8.3. Какие бывают типы деревьев?

Деревья можно классифицировать по балансировке (сбалансированные/несбалансированные), а также по количеству потомков (двоичные и т. д.).

8.4. Какие стандартные операции возможны над деревьями?

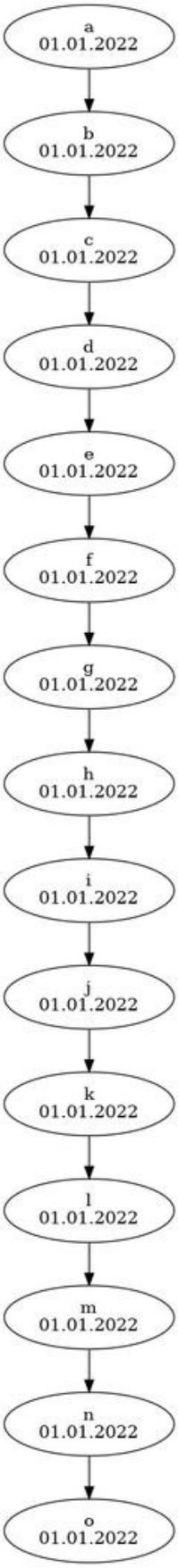
Обход дерева, поиск по дереву, включение в дерево, исключение из дерева.

8.5. Что такое дерево двоичного поиска?

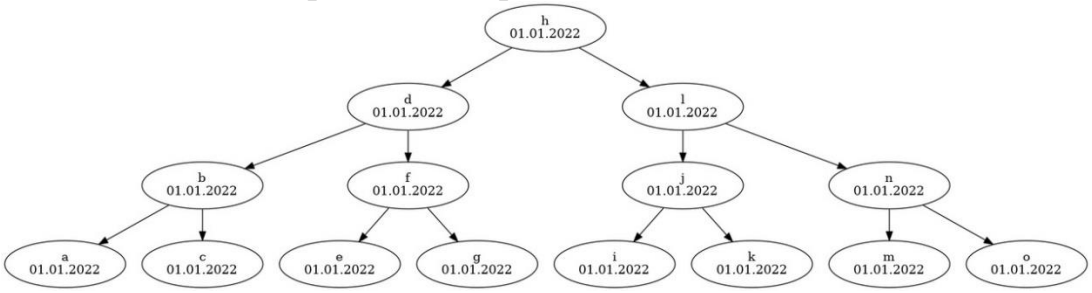
Дерево двоичного поиска – это дерево, каждый лист которого имеет не более 2-х потомков, при этом все левые потомки меньше значения данного листа, а правые – больше (листья с одинаковыми значениями в таком дереве не предусмотрены).

Приложение 1.

Правостороннее дерево.



Идеально сбалансированное дерево



Дерево со случайным порядком добавления элементов.