|  |  |
| --- | --- |
| **Gerb-BMSTU_01** | **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 №7 «Сбалансированные деревья, хеш–таблицы» (ВАР. 1)**

Студент Смирнов Иван Владимирович

Группа ИУ7 – 32Б

Проверил Барышникова Марина Юрьевна

*2023 г.*

Оглавление

[Описание условия задачи 3](#_Toc152598677)

[Описание внутренних структур данных 5](#_Toc152598678)

[Описание алгоритмов 6](#_Toc152598679)

[Сравнение времени работы и ресурсных затрат 7](#_Toc152598680)

[Тесты 10](#_Toc152598681)

[Ответы на контрольные вопросы 13](#_Toc152598682)

[Вывод 15](#_Toc152598683)

# Описание условия задачи

Программа работает с типом данных «дерево», «сбалансированное дерево», «хеш-таблица». Программа умеет строить дерево (двоичное дерево поиска или сбалансированное) и хеш-таблицу по указанным данным из файла. Реализована функция поиска в двоичном дереве и в хеш-таблице (используя открытую и закрытую адресацию). По указанию пользователя можно вывести на экран дерево и хеш-таблицу. Также выводится количество сравнений для поиска данных в указанных структурах.  
Описание технического задания

1. **Исходные данные и результат**

**Входные данные:**

Пользователь выбирает пункты меню с соответствующими действиями.

|  |
| --- |
| Меню:  ---  1. Вывести дерево и хеш-таблицы(graphviz)  2. Сбалансировать дерево  ---  3. Считать дерево (из файла)  4. Считать хеш-таблицу с закрытой адресацией (из файла)  5. Считать хеш-таблицу с открытой адресацией (из файла)  ---  6. Поиск значения (в хеш-таблице с открытой адресацией)  7. Поиск значения (в хеш-таблице с закрытой адресацией)  8. Поиск узла (в дереве)  ---  9. Обход дерева (pre-order)  10. Обход дерева (in-order)  11. Обход дерева (post-order)  ---  0. Выход  Введите ваш выбор: |

**Выходные данные:**

На экран выводятся результаты команд меню.

1. **Задачи, реализуемые программой**

**Программа умеет выводить дерево с помощью утилиты** graphviz, а хеш-таблицу на экран консоли с указанием ключа и значения. Программа умеет считывать дерево, хеш-таблицу из файла с целыми числами. В программе реализованы функции поиска элемента в указанных структурах данных.

**Способ обращения к программе**

**Команда для сборки проекта: make release  
Программа запускается через терминал командой: ./app.exe**

1. **Возможные аварийные ситуации и ошибки пользователя**
   1. **Попытка ввода не целого числа в виде выбора команды**

**На выходе сообщение: «Пожалуйста, введите целое число!»**

* 1. **Попытка ввести несуществующий номер команды**

**На выходе сообщение: «Неверный выбор.»**

* 1. **Попытка вывести пустое дерево**

**На выходе сообщение: «Дерево пусто!»**

* 1. **Попытка вывести пустую хеш-таблицу**

**На выходе сообщение: «Хеш-таблица пуста!»**

* 1. **Попытка считать дерево из несуществующего файла**

**На выходе сообщение: «Такого файла не существует!»**

* 1. **Попытка считать дерево из файла, в котором повторяются элементы**

**На выходе сообщение: «В файле найдены дубликаты вершин!»**

* 1. **Попытка найти не целое число**

**На выходе сообщение: «Пожалуйста, введите целое число!»**

* 1. **Попытка найти несуществующий элемент**

**На выходе сообщение: «Узел не найден!»**

# **Описание внутренних структур данных**

Информация о дереве и о хеш-таблицах хранится в следующих структурах (в виде узлов):

|  |
| --- |
| // Структура для узла дерева  typedef struct Node {  int data;  struct Node\* left;  struct Node\* right;  } node\_t;  // Структура для хеш-таблицы закрытой адресации  typedef struct {  int key;  int value;  } entry\_t;  typedef struct {  int size;  int cnt;  entry\_t \*table;  } hashtable\_t;  // Структура для хеш-таблицы открытой адресации  typedef struct HashEntry {  int key;  int value;  struct HashEntry\* next;  } HashEntry;  typedef struct {  int size;  HashEntry\*\* table;  } OpenedAddressingHashTable; |

node\_t – структура для хранения узла дерева, где data – значение узла; left – указатель на левый узел; right – указатель на правый узел.

Хеш-таблица закрытой адресации:

entry\_t: Структура, представляющая элемент хеш-таблицы. Она содержит ключ (key) и связанное с ним значение (value).

hashtable\_t: Структура, представляющая саму хеш-таблицу с закрытой адресацией. Она содержит размер таблицы (size), количество элементов в таблице (cnt) и указатель на массив элементов (table).

Изначальный размер хеш-таблицы = 10

Хеш-функция для записи:

|  |
| --- |
| size\_t index = (key >= 0) ? (key % hashtable->size) : (hashtable->size + (key % hashtable->size)); |

Берется остаток от деления ключа на размер хеш-таблицы. Если изначальный ключ – отрицательный, то берется отрицательный остаток и складывается с размером для получения положительного. Если по полученному индексу место в хеш-таблице занято (происходит коллизия), то происходит простое рехеширование (увеличение остатка на 1):

|  |
| --- |
| index = (index + 1) % hashtable->size; |

При переполнении изначального размера, хеш-таблица реструктуризируется таким образом, что её размер увеличивается в 2 раза. Соответственно все ключи, находящиеся в хеш-таблице до реструктуризации, преобразуются хеш-функцией, в которой используется уже новый размер.

Хеш-таблица открытой адресации:

HashEntry: Структура, представляющая элемент хеш-таблицы с открытой адресацией. Она также содержит ключ (key) и связанное с ним значение (value), а также указатель на следующий элемент в случае коллизии (next).

OpenedAddressingHashTable: Структура, представляющая хеш-таблицу с открытой адресацией. Она содержит размер массива указателей (size) и сам массив указателей на HashEntry, где каждый указатель указывает на начало связанного списка элементов, разрешающих коллизии, для соответствующего хеша.

Изначальный размер хеш-таблицы = 10

Хеш-функция для записи:

|  |
| --- |
| size\_t index = (key >= 0) ? (key % hashtable->size) : (hashtable->size + (key % hashtable->size)); |

Берется остаток от деления ключа на размер хеш-таблицы. Если изначальный ключ – отрицательный, то берется отрицательный остаток и складывается с размером для получения положительного. Если по полученному индексу место в хеш-таблице занято (происходит коллизия), то в список, хранящийся на месте index, добавляется данный элемент.

Реструктуризация происходит при заполнении одного из списков более чем на 5 элементов. В таком случае, хеш-таблица реструктуризируется таким образом, что её размер увеличивается на 1. Соответственно все ключи и значения, находящиеся в хеш-таблице до реструктуризации, преобразуются под новый размер.

# Описание алгоритмов

1. Пользователь вводит пункт меню как целое число.
2. При выборе пункта меню – вывод – выводится дерево через утилиту graphviz, а хеш-таблицы на экран. Картинка с деревом автоматически открывается.
3. С помощью отдельной функции можно сбалансировать дерево.
4. При чтении дерева из файла, программа заполняет дерево на основе данных из файла. Название файла запоминается программой, чтобы в дальнейшем можно было добавить вершину как в файл, так и в дерево. Аналогично происходит и заполнение хеш-таблиц. При необходимости хеш-таблица реструктуризируется.
5. Элемент добавляется в дерево на основе правила: «меньше – влево, больше – вправо».
6. Можно попытаться найти элемент в дереве, хеш-таблице. Выводится среднее время поиска элемента, сколько байт занимает структура данных, а также количество сравнений.
7. При выборе пункта 0 – программа завершается.

# Сравнение времени работы и ресурсных затрат

В программе предусмотрена функция, которая замеряет время (в мкс.) определенного процесса 100 раз и в качестве результата выдает среднее из 100 замеров. При выборе пунктов «6» – Поиск значения (в хеш-таблице с открытой адресацией); «7» – Поиск значения (в хеш-таблице с закрытой адресацией); «8» – Поиск узла; программа дополнительно с выбранным действием выводит время его выполнения. При выборе пунктов «6», «7» выводится размер занимаемой памяти структурой, а также количество сравнений. Результаты для следующего дерева из файла «files/1.txt» приведены ниже:

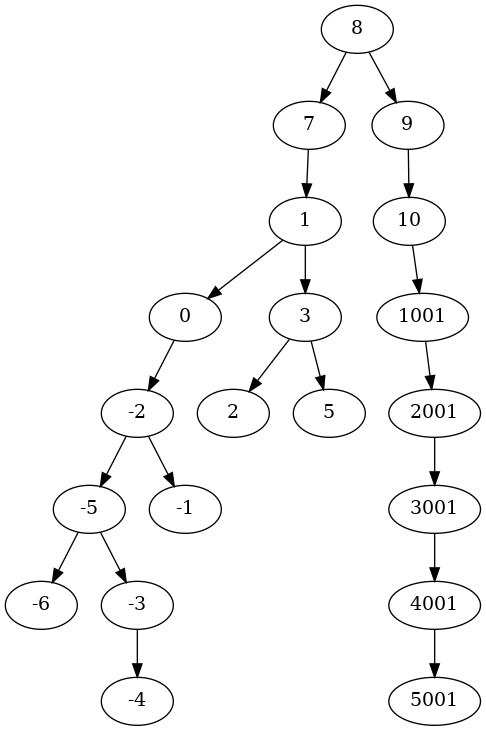


Рисунок 1. Изначальное дерево

|  |
| --- |
| … Введите ваш выбор: 4  Введите название файла:  files/1.txt  Старая хеш-таблица уничтожена  Хеш-таблица c закрытой адресацией (и внешним хешированием) успешно создана!  …  Введите ваш выбор: 5  Введите название файла:  files/1.txt  Старая хеш-таблица уничтожена  Хеш-таблица с открытой адресацией успешно создана!  …  Введите ваш выбор: 1  Хеш-таблица с закрытой адресацией  Key: 0, Value: 0, Index: 0  Key: 1, Value: 1, Index: 1  Key: 2, Value: 2, Index: 2  Key: 3, Value: 3, Index: 3  Key: 1001, Value: 1001, Index: 4  Key: 5, Value: 5, Index: 5  Key: 2001, Value: 2001, Index: 6  Key: 7, Value: 7, Index: 7  Key: 8, Value: 8, Index: 8  Key: 9, Value: 9, Index: 9  Key: 10, Value: 10, Index: 10  Key: 3001, Value: 3001, Index: 11  Key: 4001, Value: 4001, Index: 12  Key: 5001, Value: 5001, Index: 13  Key: -6, Value: -6, Index: 14  Key: -5, Value: -5, Index: 15  Key: -4, Value: -4, Index: 16  Key: -3, Value: -3, Index: 17  Key: -2, Value: -2, Index: 18  Хеш-таблица с открытой адресацией  Key: 1001, Value: 1001, Index: 0, Adress: 0x55d822a5db50  Key: 0, Value: 0, Index: 0, Adress: 0x55d822a5da30  Key: 1, Value: 1, Index: 1, Adress: 0x55d822a5d9d0  Key: 2, Value: 2, Index: 2, Adress: 0x55d822a5dad0  Key: 3, Value: 3, Index: 3, Adress: 0x55d822a5da10  Key: 5, Value: 5, Index: 5, Adress: 0x55d822a5da50  Key: -6, Value: -6, Index: 5, Adress: 0x55d822a5db30  Key: -5, Value: -5, Index: 6, Adress: 0x55d822a5daf0  Key: 7, Value: 7, Index: 7, Adress: 0x55d822a5d9b0  Key: 5001, Value: 5001, Index: 7, Adress: 0x55d822a5dbd0  Key: -4, Value: -4, Index: 7, Adress: 0x55d822a5c900  Key: 8, Value: 8, Index: 8, Adress: 0x55d822a5d990  Key: -3, Value: -3, Index: 8, Adress: 0x55d822a5db10  Key: 4001, Value: 4001, Index: 8, Adress: 0x55d822a5dbb0  Key: 9, Value: 9, Index: 9, Adress: 0x55d822a5d9f0  Key: -2, Value: -2, Index: 9, Adress: 0x55d822a5da70  Key: 3001, Value: 3001, Index: 9, Adress: 0x55d822a5db90  Key: -1, Value: -1, Index: 10, Adress: 0x55d822a5da90  Key: 2001, Value: 2001, Index: 10, Adress: 0x55d822a5db70  Key: 10, Value: 10, Index: 10, Adress: 0x55d822a5dab0  ...  Введите ваш выбор: 6  Введите значение для поиска (оно должно быть в хеш-таблице): -6  Введите ваш выбор: 6  Введите значение для поиска (оно должно быть в хеш-таблице): -6  Число найдено в хеш-таблице  Занимаемая память хеш-таблицей: 176Б  Среднее время поиска элемента <-6>: 0.210000мкс.  Теоретическое количество сравниваний: (1 - 20)  Количество сравниваний: 1  Введите ваш выбор: 7  Введите значение для поиска (оно должно быть в хеш-таблице): -6  Число найдено в хеш-таблице  Занимаемая память хеш-таблицей: 336Б  Среднее время поиска элемента <-6>: 0.210000мкс.  Теоретическое количество сравниваний: (1 - 5)  Количество сравниваний: 2  Введите ваш выбор: 8  Введите значение для поиска (оно должно быть в дереве): -6  Узел присутсвует в дереве!  Дерево занимает 480Б  Поиск элемента <-6> в дереве: 0.260000мкс.  Теоретическое количество сравниваний в сбалансированном дереве: (1 - 5)  Количество сравниваний: 7  Введите ваш выбор: 2  Дерево успешно сбалансировано!  …  Введите ваш выбор: 8  Введите значение для поиска (оно должно быть в дереве): -6  Узел присутсвует в дереве!  Дерево занимает 480Б  Поиск элемента <-6> в дереве: 0.190000мкс.  Теоретическое количество сравниваний в сбалансированном дереве: (1 - 5)  Количество сравниваний: 4 |

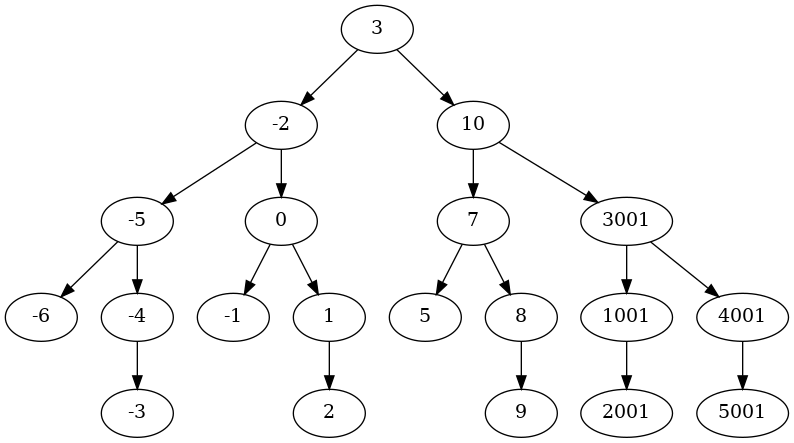


Рисунок 2. Сбалансированное дерево

Из замерного эксперимента видно, что для решения поставленной задачи целесообразно использовать хеш-таблицу, представленную в виде списков, потому что количество сравнений минимально и сравнимо со сравнениями в сбалансированном дереве, при этом в среднем поиск элемента в данной структуре занимает меньше времени, чем в обычном дереве поиска или в хеш-таблице, представленной в виде массива записей. Однако по памяти хеш-таблица на основе списков занимает больше байт, чем другая реализация, но меньше, чем дерево.

При этом открытая адресация хранит указатели на элементы непосредственно в массиве, в то время как закрытая адресация может иметь больше пустых ячеек из-за коллизий.

# Тесты

В следующих тестах символ «>» обозначает ввод пользователем пункта меню, а символ Х – полученное значение в ходе программы.

1. **Позитивные тесты**

|  |  |  |  |
| --- | --- | --- | --- |
| Номер теста | Входные данные | Выходные данные | Что проверяется |
| pos\_01 | >1 | Составленная модель дерева с помощью утилиты “graphviz” (вывод хеш-таблицы с открытой адресацией)  (вывод хеш-таблицы с закрытой адресацией) | Умеет ли программа выводить дерево с помощью “graphviz”, хеш-таблицы |
| pos\_02 | >2 | Дерево было успешно сбалансированно! | Умеет ли программа выполнять балансировку дерева |
| pos\_03 | >3  (название существующего файла с целыми числами) | Старое дерево уничтожено  Считанное дерево:  (выведенное в терминал дерево)  Дерево создалось из данных из файла <название файла> за: **Х** мкс. | Умеет ли программа считывать дерево по данным из файла и замерять время данной операции |
| pos\_04 | >4  (название существующего файла с целыми числами) | Старая хеш-таблица уничтожена  Хеш-таблица c закрытой адресацией (и внешним хешированием) успешно создана! | Умеет ли программа считывать данные из файла в хеш-таблицу с открытой адресацией |
| pos\_05 | >5  (название существующего файла с целыми числами) | Старая хеш-таблица уничтожена  Хеш-таблица с открытой адресацией успешно создана! | Умеет ли программа считывать данные из файла в хеш-таблицу с закрытой адресацией |
| pos\_06 | >6  (существующий элемент) | Число найдено в хеш-таблице  Занимаемая память хеш-таблицей: **Х** Б  Среднее время поиска элемента < **Х**>: **Х** мкс.  Количество сравниваний: **Х** | Умеет ли программа добавлять узел только в само дерево |
| pos\_07 | >7  (существующий элемент) | Число найдено в хеш-таблице  Занимаемая память хеш-таблицей: **Х** Б  Среднее время поиска элемента < **Х**>: **Х** мкс.  Количество сравниваний: **Х** | Умеет ли программа удалять узел из дерева |
| pos\_08 | >8  (существующий элемент) | Узел присутствует в дереве!  Поиск элемента <**Х**> в дереве: **Х** мкс. | Умеет ли программа находить узел в дереве и замерять данный процесс |

1. **Негативные тесты**

|  |  |  |  |
| --- | --- | --- | --- |
| Номер теста | Входные данные | Выходные данные | Что проверяется |
| neg\_01 |  | Сообщение: «Пожалуйста, введите целое число!» | Пустой ввод |
| neg\_02 | >-1 | Сообщение: «Неверный выбор.» | Правильность ввода пункта меню |
| neg\_03 | >100 | Сообщение: «Неверный выбор.» | Правильность ввода пункта меню |
| neg\_04 | >А | Сообщение: «Пожалуйста, введите целое число!» | Правильность ввода пункта меню |
| neg\_05 | >1  (длина дерева 0) | Сообщение: «Дерево пусто!» | Существование дерева |
| neg\_06 | >2  (длина дерева 0) | Сообщение: «Дерево пусто!» | Существование дерева |
| neg\_07 | >3  (несуществующий файл) | Сообщение:  «Такого файла не существует!» | Существование файла |
| neg\_08 | >3  (файл с дубликатами) | Сообщение:  «В файле найдены дубликаты вершин!» | Попытка добавить элемент, который уже есть в дереве |
| neg\_09 | >4  (файл с дубликатами) | Сообщение:  «В файле найдены дубликаты вершин!» | Попытка добавить элемент, который уже есть в хеш-таблице |
| neg\_10 | >5  (файл с дубликатами) | Сообщение:  «В файле найдены дубликаты вершин!» | Попытка добавить элемент, который уже есть в хеш-таблице |
| neg\_11 | >7  (не существующий элемент) | Сообщение:  «Число НЕ найдено!» | Поиск несуществующего элемента |
| neg\_12 | >8  (не существующий элемент) | Сообщение:  «Узел не найден!» | Поиск несуществующего элемента |
| neg\_13 | >6  (не существующий элемент) | Сообщение:  «Число НЕ найдено!» | Поиск несуществующего элемента |
| neg\_14 | >6  (не целое число) | Сообщение: «Пожалуйста, введите целое число!» | Корректность ввода элемента |
| neg\_15 | >7  (не целое число) | Сообщение: «Пожалуйста, введите целое число!» | Корректность ввода элемента |
| neg\_16 | >8  (не целое число) | Сообщение: «Пожалуйста, введите целое число!» | Корректность ввода элемента |

# Ответы на контрольные вопросы

1. **Чем отличается идеально сбалансированное дерево от АВЛ дерева?**

Идеально сбалансированное дерево отличается от АВЛ-дерева тем, что в идеально сбалансированном дереве разница между высотами левого и правого поддеревьев для каждого узла может быть не более 1, в то время как в АВЛ-дереве эта разница строго ограничена и не превышает 1.

1. **Чем отличается поиск в АВЛ-дереве от поиска в дереве двоичного поиска?**

Поиск в АВЛ-дереве отличается от поиска в дереве двоичного поиска тем, что в АВЛ-дереве гарантируется балансировка, что обеспечивает время поиска O(log n), в то время как в обычном дереве двоичного поиска время поиска может быть O(n) в худшем случае, если дерево несбалансировано.

1. **Что такое хеш-таблица, каков принцип ее построения?**

Хеш-таблица - это структура данных, которая использует хеш-функцию для отображения ключей или значений в более компактное представление, что позволяет быстро и эффективно искать, добавлять и удалять элементы. Принцип ее построения заключается в вычислении хеш-функции для ключей и размещении соответствующих значений в соответствующих "ячейках" таблицы.

1. **Что такое коллизии? Каковы методы их устранения.**

Коллизии в хеш-таблицах возникают, когда два различных ключа отображаются в одно и то же значение хеш-функции. Методы их устранения включают в себя использование методов открытой адресации (линейное пробирование, квадратичное пробирование, двойное хеширование) и методов закрытой адресации (использование связанных списков или деревьев для разрешения коллизий).

1. **В каком случае поиск в хеш-таблицах становится неэффективен?**

Поиск в хеш-таблицах становится неэффективным в случае частых коллизий, когда большое количество ключей отображается в одну и ту же "ячейку" хеш-таблицы, что может привести к увеличению времени поиска.

1. **Эффективность поиска в АВЛ деревьях, в дереве двоичного поиска, в хеш-таблицах и в файле.**

В АВЛ-деревьях: O(log n)

В дереве двоичного поиска: В сбалансированном дереве - O(log n), в несбалансированном - O(n)

В хеш-таблицах: В идеальном случае O(1), но может ухудшаться из-за коллизий и из-за большого времени на вычисление хеша

В файлах: В зависимости от способа организации данных, обычно O(n)

# Вывод

В рамках задания были реализован такие типы данных как: «дерево двоичного поиска», «сбалансированное дерево», «хеш-таблица с открытой адресацией», «хеш-таблица с закрытой адресацией». Для каждого типа данных (деревья объединены в один тип) реализованы функции считывания данных (целых чисел) из файла, а также поиска элемента.