Лабораторная работа №7

Условия задачи

Используя предыдущую программу (задача № 6), сбалансировать полученное дерево. Вывести его на экран в виде дерева. Построить хеш-таблицу из чисел файла. Осуществить поиск введенного целого числа в двоичном дереве поиска, в сбалансированном дереве и в хеш-таблице. Сравнить время поиска, объем памяти и количество сравнений при использовании различных структур данных.

Построить хеш-таблицу по указанным данным. Сравнить эффективность поиска в сбалансированном двоичном дереве, в двоичном дереве поиска и в хеш-таблице (используя открытую и закрытую адресацию). Вывести на экран деревья и хеш-таблицы. Подсчитать среднее количество сравнений для поиска данных в указанных структурах. Произвести реструктуризацию хеш-таблицы, если среднее количество сравнений больше указанного. Оценить эффективность использования этих структур (по времени и памяти) для поставленной задачи. Оценить эффективность поиска в хеш-таблице при различном количестве коллизий и при различных методах их разрешения.

ТЗ

Исходные данные и результаты

Входные данные:

- текстовый файл с целыми числами;
- целое число;

Результат работы:

- выведенное в стандартный поток вывода *(по желанию пользователя)*:
    - AVL-дерево;
    - наивное дерево
    - хеш-таблица с закрытой адресацией;
    - хеш-таблица с открытой адресцией;
- результаты тестов для заданных контейнеров:
    - время поиска заданного элемента;
    - объем занимаемой помяти;
    - количество сравнений элементов.

Описание задачи, реализуемой программой

Целые числа считываются из файла и одновременно добавляются во все контейнеры по-отдельности. После замеряется объем занимаемой памяти, время поиска элемента во всех четырёх контейнерах по-отдельности. По желанию пользователя печатаются контейнеры в приятном для человикавиду. В операции поиска измеряется количество сравнений элементов, эта информация также выводится на экран.

Возможные аварийные ситуации и ошибки пользователя

- Некорректный формат данных во входном файле$;
- Ошибки выделения памяти.

Описание внутренних структур данных

TreeNode

```
struct TreeNode {            // вершина дерева
    value_t data;            // данные
    int n;                   // порядковый номер
    struct TreeNode* left;   // указатель на левого ребёнка
    struct TreeNode* right;  // указатель на правого ребёнка
    struct TreeNode* parent; // указатель на родителя
};
```

Tree

```
struct Tree {                // наивное дерево
    struct TreeNode* root;   // указатель на корень дерева
    int n;                   // количество вершин дерева
};
```

ClosedHash

```
struct ClosedHash {          // хеш-таблица с цепочками
    struct list* buckets;    // ячейки (списки)
    int size;                // количество ячеек
};
```

OpenHashNode

```
struct OpenHashNode {        // элемент таблицы с открытой адресацией
    okey_t key;              // ключ
    int state;               // состояние ячейки
};
```

OpenHash

```
struct OpenHash {            // хеш-таблица с открытой адресацией
    OpenHashNode* buckets;   // ячейки (записи)
    int size;                // количество ячеек
};
```

Тесты

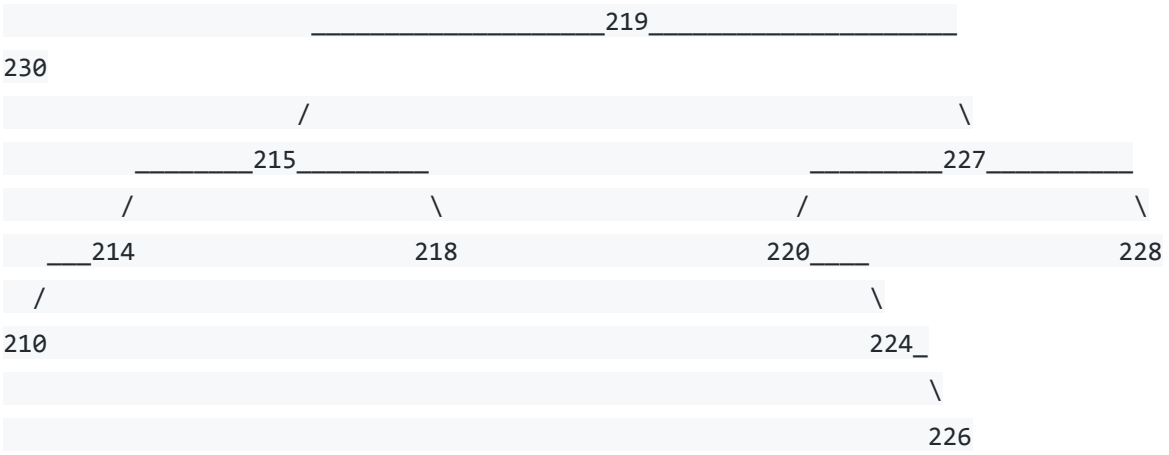
С выводом контейнеорв на экран

```
Do you want to generate new Random data? {1/0}: 1
Enter amount of numbers to generate: 15
Enter diapason of generated numbers {-2^31...2^31}: 210 230
Enter the number to search: 228
Do you want to print all data structures? {1/0}: 1

           _________220_________
          /                     \
     ___215____               ___227____
    /          \             /          \
  214          219         224_         229_
 /            /               \       /    \
210          218              226    228    230



_____________________________________229_____________________________________
                                      /
\
```




```
[134] : {218} -> nullptr
[135] : {227} -> nullptr
[321] : {219} -> nullptr
[416] : {230} -> nullptr
[472] : {215} -> nullptr
[531] : {224} -> nullptr
[658] : {214} -> nullptr
[735] : {210} -> nullptr
[886] : {228} -> nullptr
[946] : {229} -> nullptr
[950] : {220} -> nullptr
[1011] : {226} -> nullptr
```

```
[134] : {218}
[135] : {227}
[321] : {219}
[416] : {230}
[472] : {215}
[531] : {224}
[658] : {214}
[735] : {210}
[886] : {228}
[946] : {229}
[950] : {220}
[1011] : {226}
```

```
Amount of comparisons in AVL Binary Tree: 8
Amount of memory allocated for AVL Binary Tree data structure: 384 (bytes)
```

```
Amount of time taken to find element in AVL Binary Tree: 321 (ns)

Amount of comparisons in Naive Binary Tree: 8
Amount of memory allocated for Naive Binary Tree data structure: 288 (bytes)
Amount of time taken to find element in Naive Binary Tree: 235 (ns)

Amount of comparisons in Closed Addressing Hash Table: 1
Amount of memory allocated for Closed Addressing Hash Table data structure: 180
(bytes)
Amount of time taken to find element in Closed Addressing Hash Table: 3311 (ns)

Amount of comparisons in Open Addressing Hash Table: 1
Amount of memory allocated for Open Addressing Hash Table data structure: 96
(bytes)
Amount of time taken to find element in Open Addressing Hash Table: 2586 (ns)
```

Без вывода контейнеров на экран

```
   Do you want to generate new Random data? {1/0}: 1
Enter amount of numbers to generate: 10000
Enter diapason of generated numbers {-2^31...2^31}: 0 2000
Enter the number to search: 1488
Do you want to print all data structures? {1/0}: 0

Amount of comparisons in AVL Binary Tree: 28
Amount of memory allocated for AVL Binary Tree data structure: 63584 (bytes)
Amount of time taken to find element in AVL Binary Tree: 585 (ns)

Amount of comparisons in Naive Binary Tree: 34
Amount of memory allocated for Naive Binary Tree data structure: 47688 (bytes)
Amount of time taken to find element in Naive Binary Tree: 1204 (ns)

Amount of comparisons in Closed Addressing Hash Table: 1
Amount of memory allocated for Closed Addressing Hash Table data structure: 39510
(bytes)
Amount of time taken to find element in Closed Addressing Hash Table: 4237 (ns)

Amount of comparisons in Open Addressing Hash Table: 4
Amount of memory allocated for Open Addressing Hash Table data structure: 15896
```

```
(bytes)
Amount of time taken to find element in Open Addressing Hash Table: 3119 (ns)
```

Сравнения реализаций

Время работы (нс)

Размер	Наивное дерево	AVL-дерево	Закрытая адресация	Открытая адресация
100	2761	403	4087	3292
1000	24756	436	4147	3703
10000	205991	452	4250	3430
100000	1973013	409	3466	3968

Занимаемая память

Размер	Наивное дерево	AVL-дерево	Закрытая адресация	Открытая адресация
100	2400	3200	1500	800
1000	24000	32000	15000	8000
10000	240000	320000	159735	80000

1000000	2400000	32000000	1509735	800000

Колисество сравнений

Размер	Наивное дерево	AVL-дерево	Закрытая адресация	Открытая адресация
100	85	13	1	3
1000	810	19	1	3
10000	7951	27	2	4
100000	84515	33	1	6

Выводы

Самой оптимальной структурой по времени для поиска данных является сбалансированное дерево (при не очень большом количестве элементов). Реструктуризация хеш-таблицы позволяет уменьшить среднее количество сравнений при поиске, но это не сильно влияет на время выполнения, т.к. большая часть тратится на вычисление значения хеш-функции. Таблица с цепочками несколько выигрывает по времени у таблицы с открытой адресацией, но в некоторых случаях проигрывает по памяти.

Контрольные вопросы

1. Чем отличается идеально сбалансированное дерево от АВЛ дерева? Адельсон-Вельский и Ландис сформулировали менее жесткий критерий сбалансированности таким образом: двоичное дерево называется сбалансированным, если у каждого узла дерева высота двух поддеревьев отличается не более чем на единицу. Такое дерево называется АВЛ-деревом.
2. Чем отличается поиск в АВЛ-дереве от поиска в дереве двоичного поиска? Алгоритм поиска одинаковый, но в идеально сбалансированном дереве в среднем выполняется быстрее.

3. Что такое хеш-таблица, каков принцип ее построения? Массив, заполненный в порядке, определенным хеш-функцией, называется хеш-таблицей.
4. Что такое коллизии? Каковы методы их устранения. Может возникнуть ситуация, когда разным ключам соответствует одно значе-ние хеш-функции, то есть, когда $h(K1) = h(K2)$, в то время как $K1 \neq K2$. Такая ситуация называется коллизией.
5. В каком случае поиск в хеш-таблицах становится неэффективен? При большом количестве коллизий.