Лабораторная работа №7

Сбалансированные деревья, хеш –таблицы

Цель работы: построить и обработать хеш-таблицы, сравнить эффективность поиска в сбалансированных деревьях, в двоичных деревьях поиска и в хеш-таблицах. Сравнить эффективность устранения коллизий при внешнем и внутренним хешировании.

Задание (Вариант 0):

Сбалансировать дерево (задача №6) после удаления повторяющихся букв. Вывести его на экран в виде дерева. Составить хеш-таблицу, содержащую буквы (хеш - по количеству различных букв) и количество их вхождений во введенной строке. Вывести таблицу на экран. Осуществить поиск введенной буквы в двоичном дереве поиска, в сбалансированном дереве и в хеш-таблице. Сравнить время поиска, объем памяти и количество сравнений при использовании различных структур данных.

Требования к входным данным:

Программа предназначена для работы с текстовым файлом. Файл может содержать произвольное количество символов. Также, исходную строку можно изменить из консоли. В структуры добавляются только буквы, при этом программа чувствительна к регистру.

Выходные данные:

По требованию пользователя программа печатает деревья (бинарное и сбалансированное), а также хеш-таблицы (с открытой и закрытой адресацией). Для перечисленных структур реализованы поиск, добавление и удаление буквы. При необходимости, программа производит сравнение времени поиска, кол-ва сравнений и затрат памяти в указанных структурах. Примеры см. Интерфейс программы.

Интерфейс программы:

```
****************************************************************
Выберите одно из следующих действий:
0: Загрузить строку из файла
1: Работа с bst деревом
2: Работа с avl деревом
3: Работа с хэш таблицей с открытой адресацией
4: Работа с хэш таблицей с закрытой адресацией
5: Сравнение времени поиска
6: Изменить исходную строку
7: Закончить работу
```

0 - Добавление в дерево строки, заранее подготовленной в файле. Пример содержания файла:

mnkabcqwerqwebcacmnknmn

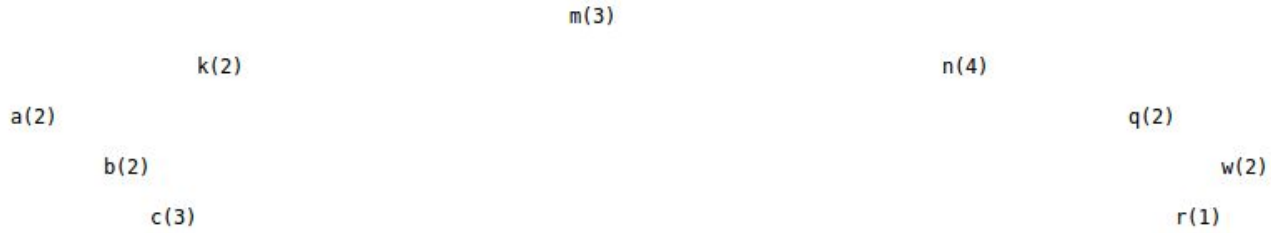
1,2 - работа с деревом (интерфейс для работы со структурами одинаковый).

3,4 - работа с таблицами (интерфейс для работы со структурами одинаковый).
5 - по запросу пользователя для введенного символа печатается кол-во сравнение, необходимых для его поиска, потраченное время, а также общий объем памяти для каждой из структур. Примеры подробнее рассмотрены ниже.
6 - задание новой строки через консоль. При этом обрабатываются (добавляются в структуры) только те символы, которые являются буквами.
7 - закончить работу программы.

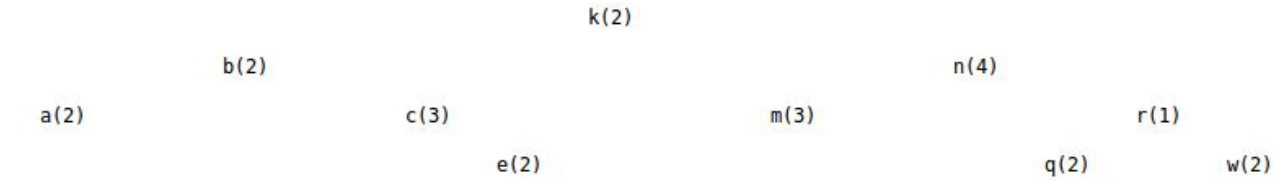
Интерфейс работы с деревом

```
************************************************************
Выберите одно из следующих действий:
1: Отобразить дерево (1 способ)
2: Отобразить дерево (2 способ)
3: Добавить букву в дерево
4: Удалить букву из дерева
5: Удалить все вхождения буквы в дереве
6: Поиск буквы
7: Закончить работу
```

1 - Печать дерева первым способом. Производится печать первых 5 уровней (это связано с наглядностью расположения эл-тов в дереве). Каждый элемент представлен в виде значения и количества вхождений.
Бинарное дерево



Сбалансированное дерево



2 - Печать дерева вторым способом. Сыновья расположены правее родителей. Верхний сын является левым сыном, нижний - правым сыном для каждого элемента. Для удобства, если у какого-либо элемента есть только один потомок, на месте второго печатается прочерк (-----).
Сбалансированное дерево:

```
k(2)
|----b(2)
|    |----a(2)
|    |----c(3)
|    |    |---------
|    |    |----e(2)
|----n(4)
|    |----m(3)
|    |----r(1)
|    |    |----q(2)
|    |    |----w(2)
```

Бинарное дерево:

```
m(3)
|----k(2)
|    |----a(2)
|    |    |---------
|    |    |----b(2)
|    |    |    |---------
|    |    |    |----c(3)
|    |    |    |    |---------
|    |    |    |    |----e(2)
|    |---------
|----n(4)
|    |---------
|    |----q(2)
|    |    |---------
|    |    |----w(2)
|    |    |    |----r(1)
|    |    |    |---------
```

3 - Добавление буквы в дерево. Программа предложит ввести символ в строке консоли. Если символ не является буквой, программа напечатает соответствующее предупреждение.
4 - Удаление буквы. Если дерево не содержит указанной буквы, программа выдает соответствующее сообщение, иначе - сообщение об успешном удалении. (При этом удаляется только одно вхождение буквы в дереве)
5 - Удаление всех вхождений в дерево указанной буквы.
6 - Поиск буквы. Программа запрашивает у пользователя символ и выдает результат: содержится ли указанное слово в дереве, или нет.
7 - Закончить работу с текущим деревом

Интерфейс работы с таблицей

```
****************************************************************
Выберите одно из следующих действий:
1: Отобразить таблицу
2: Добавить букву в таблицу
3: Удалить букву из таблицы
4: Удалить все вхождения буквы в таблице
5: Поиск буквы
6: Закончить работу
```

1 - Печать таблицы. Таблица печатается полностью. Для каждого индекса печатается элемент (значение и кол-во вхождений) или пустая строка. Для таблицы с открытой адресацией печатается список из нескольких эл-тов, если имеют одинаковый хеш.

Таблица с открытой адресацией

```
0:
1:a(2) q(2)
2:b(2) r(1)
3:k(2) c(3)
4:
5:m(3) e(2)
6:n(4)
7:w(2)
```

Таблица с закрытой адресацией

```
0:c(3)
1:e(2)
2:
3:a(2)
4:
5:b(2)
6:q(2)
7:k(2)
8:
9:
10:r(1)
11:m(3)
12:
13:n(4)
14:
15:w(2)
```

2 - Добавление буквы в дерево. Программа предложит ввести символ в строке консоли. Если символ не является буквой, программа напечатает соответствующее предупреждение.
3 - Удаление буквы. Если таблица не содержит указанной буквы, программа выдает соответствующее сообщение, иначе - сообщение об успешном удалении. (При этом удаляется только одно вхождение буквы в таблице)
4 - Удаление всех вхождений исходной буквы в таблице.
5 - Поиск буквы. Программа запрашивает у пользователя символ и выдает результат: содержится ли указанное слово в таблице, или нет.

6 - Закончить работу с текущей таблицей

Обращение к программе:

Через консоль

Внутренние структуры данных:

Бинарное дерево:

Структура элемента дерева

```
template <typename T>
struct element {
  T key;
  element *left = NULL;
  element *right = NULL;
  element(T x) {
    //cout << "element:" << x << endl;
    key = T(x);
    left = NULL;
    right = NULL;
  }
};

//Binary search tree
template <typename T>
class CBST {
private:
  element<T> *head = NULL;
  int count_of_element = 0;
  void delete_tree(element<T>* tmp);
  void operator_copy(element<T> **head, element<T> *tmp);
  void delete_remove(element<T>* prev, element<T> *tmp);
public:
  CBST(const CBST &obj);
  CBST();
  CBST<T>& operator=(const CBST& obj);
  void Insert(T x);
  bool Remove(T x);
  bool Search(T x);
  ~CBST();
  template <typename X>
  friend void print(X *tmp, int deep, bool flag);
  template <typename X>
  friend void show_as_tree(X* tree);
  template <typename X>
  friend void show(X* tree);
  int Memory();

  };
```

Сбалансированное дерево:

Структура элемента

```
template <typename T>
```

```
struct AVL_Node {
   AVL_Node* left, *right;
   T key;
   int h;
   int cnt;
   AVL_Node() {
      left = right = NULL;
      h = cnt = 0;
   }
   AVL_Node(T& x) {
      left = right = NULL;
      key = x;
      h = cnt = 1;
   }
};
```

Структура дерева:

```
template <typename T>
class CAVL {
private:
   int height(AVL_Node<T> *t);
   int difference_of_h(AVL_Node<T> *t);
   int cnt_value(AVL_Node<T> * t);
   T key_value(AVL_Node<T> * t);
   void right(AVL_Node<T> * &t);
   void left(AVL_Node<T> * &t);
   void balance(AVL_Node<T>* &t); // балансировка узла p
   void add_to_avl(AVL_Node<T>* &t, AVL_Node<T>* tmp);
   AVL_Node<T>* findmin(AVL_Node<T>* t);
   AVL_Node<T>* removemin(AVL_Node<T>* t);
   bool delete_from_avl(AVL_Node<T>* &t, T x);
   //int value_by_index(AVL_Node* t, int ind);
   //int index(AVL_Node* t, int x);
   void free_tree(AVL_Node<T> * &tmp);
   AVL_Node<T>* head = NULL;
   int count_of_element = 0;
public:
   CAVL() {}
   ~CAVL();
   int Memory();
   void Insert(T x);
   bool Remove(T x);
   bool Search(T x);
   template <typename X>
   friend void print(X *tmp, int deep, bool flag);
   template <typename X>
   friend void show_as_tree(X* tree);
   template <typename X>
   friend void show(X* tree);
};
```

Таблица с открытой адресацией

```
template <typename T>
struct ElementTable {
   T value;
   ElementTable<T>* next = NULL;
   ElementTable() {}
   ElementTable(T x) {
      value = x;
      next = NULL;
   }
};

template <class T>
class CHashTableOpen
{
private:
   ElementTable<T>** table;
   bool* status; // 0 - pusto, 1 - is
   int tableSize;
   int count_of_elem;
public:
   CHashTableOpen(int Start_size);
   ~CHashTableOpen();
   void New_Table(int new_size);
   bool Insert(T k);
   bool Search(T x);
   int Memory();
   bool Delete_element(T x);
   void Show();
};
```

Таблица с закрытой адресацией

```
template <class T>
class CHashTableClose
{
private:
   T* table;
   short int* status; // 0 - pusto, 1 - is, 2 - del
   int tableSize;
   int count_of_elem;
public:
   CHashTableClose(int Start_size);
   ~CHashTableClose();
   void New_Table(int new_size);
   bool Insert(T k);
   bool Search(T x);
   int Memory();
   bool Delete_element(T x);
   void Show();
```

```
};
```

Особенности реализации таблиц:
Размер таблицы зависит от кол-ва различных элементов, которые содержатся в таблице. Поэтому при превышении определенного количества, происходит перехеширование таблицы. Это необходимо для того, чтобы избавится от накопления коллизий.
Для таблицы с открытой адресацией перехеширование происходит, если отношение кол-ва различных элементов к размеру таблицы равное 2. Для закрытой адресации это отношение не больше 3/4.
Для эффективного устранения коллизий в таблице с закрытой адресацией, используется линейное пробирование с шагом, зависящем от значения хеш-функции.

Сравнение поиска в различных структурах данных:
Рассмотрим пример поиска в строке указанной выше (для нее отображены все деревья и таблицы)

```
m
Время поиска

Бинарное дерево:
Время работы(мкс):2
Кол-во сравнений: 1
Объем памяти(байт): 240

Avl дерево:
Время работы(мкс):1
Кол-во сравнений: 6
Объем памяти(байт): 320

Таблица с открытой адресацией:
Время работы(мкс):1
Кол-во сравнений: 1
Объем памяти(байт): 160

Таблица с закрытой адресацией:
Время работы(мкс):1
Кол-во сравнений: 2
Объем памяти(байт): 128
```

```
k
Время поиска

Бинарное дерево:
Время работы(мкс):2
Кол-во сравнений: 3
Объем памяти(байт): 240

Avl дерево:
Время работы(мкс):0
Кол-во сравнений: 1
Объем памяти(байт): 320

Таблица с открытой адресацией:
Время работы(мкс):1
Кол-во сравнений: 1
Объем памяти(байт): 160

Таблица с закрытой адресацией:
Время работы(мкс):0
Кол-во сравнений: 2
Объем памяти(байт): 128
```

```
e
Время поиска

Бинарное дерево:
Время работы(мкс):2
Кол-во сравнений: 11
Объем памяти(байт): 240

Avl дерево:
Время работы(мкс):1
Кол-во сравнений: 8
Объем памяти(байт): 320

Таблица с открытой адресацией:
Время работы(мкс):1
Кол-во сравнений: 2
Объем памяти(байт): 160

Таблица с закрытой адресацией:
Время работы(мкс):1
Кол-во сравнений: 4
Объем памяти(байт): 128
```

Легко заметить, что таблицы показывают относительно стабильное время поиска и кол-во сравнений. В дереве результат поиска зависит от расположения символа в дереве: m - корень бинарного дерева поиска, k - корень сбалансированного дерева, e - одна из листовых вершин.
Целесообразно также рассмотреть “худший” для бинарного дерева случай: буквы добавляются в дерево в алфавитном порядке. Результаты поиска буквы z в алфавите:

```
Бинарное дерево:
Время работы(мкс):4
Кол-во сравнений: 51
Объем памяти(байт): 624

Avl дерево:
Время работы(мкс):2
Кол-во сравнений: 10
Объем памяти(байт): 832
```

```
Таблица с открытой адресацией:
Время работы(мкс):1
Кол-во сравнений: 2
Объем памяти(байт): 416

Таблица с закрытой адресацией:
Время работы(мкс):2
Кол-во сравнений: 2
Объем памяти(байт): 512
```

Проанализируем затраченное количество памяти:
Деревья - структуры, память на которые выделяется по мере добавление элементов. Объем памяти прямо пропорционален количеству элементов. Так как для реализации сбалансированного дерева поиска необходимо хранить дополнительные данные (высоту поддерева для каждой вершины), то для его хранения необходимо больше памяти.
Так как кол-во элементов в таблице также зависит от количества элементов в данной реализации (при перехешировании размер таблицы удваивается), то количество памяти на хранение таблиц также линейно зависит от кол-ва элементов. Таблица с открытой адресацией требует больше памяти по сравнению с открытой, так как в ней выделяется константный объем памяти под все элементы, тогда как таблица с открытой адресацией может содержать лишь пустые указатели.
“Лучшие” показатели памяти для таблиц по сравнению с деревьями можно объяснить тем, что в деревьях, помимо значение, необходимо хранить указатель на потомков.

Вывод:

Основным преимуществом двоичного дерева поиска перед другими структурами данных является возможная высокая эффективность реализации основанных на нём алгоритмов поиска и сортировки.

Хранение и обработка дерева требует аккуратного обращение с памятью. Поэтому для этой структуры данных нужно особенно тщательно тестировать удаление элементов, а также добавление элементов.

Теоретическая часть:

1. Чем отличается идеально сбалансированное дерево от АВЛ дерева?

Если при добавлении узлов в дерево располагать их равномерно слева и справа, то получится дерево, у которого число вершин в левом и правом поддеревьях отличается не более, чем на единицу. Такое дерево называется идеально сбалансированным. Идеальная балансировка дает наименьшую высоту дерева.

В АВЛ дереве высота левого и правого поддеревьев отличается не более, чем на единицу.

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

Поиск в АВЛ дереве имеет трудоёмкость O(log2n), в то время как в обычном ДДП может иметь O(n). АВЛ дерево никогда не вырождается в линейный список (исключение – дерево из двух элементов), в то время как «внешний вид» ДДП может зависеть от того, в каком порядке в него добавлялись элементы.

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

Хеш-таблицей называется массив, заполненный элементами в порядке, определяемом хеш-функцией. Хеш-функция каждому элементу таблицы ставит в соответствие некоторый индекс. ХФ должна быть простой для вычисления,

распределять ключи в таблице равномерно и давать минимум коллизий.

4. Что такое коллизии? Каковы методы их устранения.

Коллизия – ситуация, когда разным ключам хеш-функция ставит в соответствие один и тот же индекс. Основные методы устранения коллизий: открытое и закрытое хеширование.

При открытым хешировании, конфликтующие ключи просто добавляются в список, находящийся по их общему индексу. Поиск по ключу сводится к определению индекса, а затем к поиску ключа в списке перебором.

При закрытом хешировании, конфликтующий ключ добавляется в первую свободную ячейку после «своего» индекса. Поиск по ключу сводится к определению начального приближения, а затем к поиску ключа методом перебора.

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

Поиск в ХТ становится неэффективен при большом числе коллизий – сложность поиска возрастает по сравнению с O(1). В этом случае требуется реструктуризация таблицы – заполнение её с использованием новой хеш-функции.

6. Эффективность поиска в АВЛ деревьях, в дереве двоичного поиска и в хеш-таблицах.

В хеш-таблице минимальное время поиска O(1). В АВЛ: O(log2n). В дереве двоичного поиска O(h), где h - высота дерева (от log2n до n).