**Московский авиационный институт (национальный исследовательский университет)**

Институт информационных технологий и прикладной математики

«Кафедра вычислительной математики и программирования»

# Лабораторная работа по предмету "Дискретный анализ" №2

Студент: Кострюков Е.С.

Преподаватель: Макаров Н.К.

Группа: М8О-207Б-22

Дата:

Оценка:

Подпись:

**Москва 2024 г.**

# Оглавление

# Цель работы

Приобретение практических навыков в:

- Написании древовидных структур данных.
- Получение теоретических знаний по теме древовидных структур данных.

# Постановка задачи

Реализовать декартово дерево с возможностью поиска, добавления и удаления элементов. Необходимо создать программную библиотеку, реализующую указанную структуру данных, на основе которой разработать программу-словарь. В словаре каждому ключу, представляющему из себя регистронезависимую последовательность букв английского алфавита длиной не более 256 символов, поставлен в соответствие некоторый номер, от 0 до $2^{64} - 1$. Разным словам может быть поставлен в соответствие один и тот же номер. Программа должна обрабатывать строки входного файла до его окончания. Каждая строка может иметь следующий формат:

**+ word 34** — добавить слово «word» с номером 34 в словарь. Программа должна вывести строку «OK», если операция прошла успешно, «Exist», если слово уже находится в словаре.

**- word** — удалить слово «word» из словаря. Программа должна вывести «OK», если слово существовало и было удалено, «NoSuchWord», если слово в словаре не было найдено.

**word** — найти в словаре слово «word». Программа должна вывести «OK: 34», если слово было найдено; число, которое следует за «OK:» — номер, присвоенный слову при добавлении. В случае, если слово в словаре не было обнаружено, нужно вывести строку «NoSuchWord».

# Общий алгоритм решения

**Инициализация:**

Создается объект класса Treap, который инициализирует корень дерева как nullptr и генерирует случайное начальное значение для seed, которое будет использоваться для генерации приоритетов.

**Обработка команд:**

*Метод process_command анализирует команду и выполняет соответствующую операцию:*

- вставка: <+ word value> - вставляет слово-значение в дерево;

- удаление: <- word> - удаляет слово из дерева;

- поиск: <word> - ищет слово в дереве и выводит его значение.

**Операции с деревом:**

*Для выполнения операций с деревом используются рекурсивные методы:*

- split: разделяет узел на два по заданному слову: левая часть содержит слова, меньшие или равные заданному, а правая - слова, большие заданного;

- merge: объединяет два узла в один, сохраняя свойства дерева (баланс и кучу);

- insert: вставляет новый узел в дерево, используя split и merge для поддержания баланса;

- erase: удаляет узел из дерева, используя split и merge;

- find: ищет узел в дереве.

**Вывод:**

*После каждой операции выводится соответствующее сообщение:*

- “OK”: операция выполнена успешно;

- “Exist”: слово уже существует (при вставке);

- “NoSuchWord”: слово не найдено (при удалении или поиске).

# Реализация

## lab2.cpp

```cpp
#include <iostream>
#include <cstring>
#include <cstdlib>
#include <cstdint>
#include <cctype>

constexpr int MAX_WORD_LENGTH = 257;

class Node {
public:
    char word[MAX_WORD_LENGTH];
    uint64_t priority;
    uint64_t value;
    Node *left, *right;

    Node(const char *word, uint64_t value, unsigned int *seed) {
        strncpy(this->word, word, MAX_WORD_LENGTH);
        this->word[MAX_WORD_LENGTH - 1] = '\0';
        this->value = value;
        this->priority = rand_r(seed);
        this->left = this->right = nullptr;
    }
};

class Treap {
private:
    Node *root;
    unsigned int seed;

    void split(Node *current, const char *word, Node *&left, Node *&right) {
        if (!current) {
            left = right = nullptr;
        } else if (strcasecmp(word, current->word) < 0) {
            split(current->left, word, left, current->left);
            right = current;
        } else {
            split(current->right, word, current->right, right);
            left = current;
        }
    }

```

```
42      Node* merge(Node *left, Node *right) {
43          if (!left || !right) {
44              return left ? left : right;
45          }
46          if (left->priority > right->priority) {
47              left->right = merge(left->right, right);
48              return left;
49          } else {
50              right->left = merge(left, right->left);
51              return right;
52          }
53      }
54
55      Node* insert(Node *current, Node *new_node) {
56          if (!current) {
57              return new_node;
58          }
59          if (strcasecmp(new_node->word, current->word) == 0) {
60              return current;
61          }
62          if (new_node->priority > current->priority) {
63              split(current, new_node->word, new_node->left, new_node->right);
64              return new_node;
65          } else if (strcasecmp(new_node->word, current->word) < 0) {
66              current->left = insert(current->left, new_node);
67          } else {
68              current->right = insert(current->right, new_node);
69          }
70          return current;
71      }
72
73      Node* erase(Node *current, const char *word) {
74          if (!current) {
75              return nullptr;
76          }
77          if (strcasecmp(word, current->word) == 0) {
78              Node *temp = merge(current->left, current->right);
79              delete current;
80              return temp;
81          }
82          if (strcasecmp(word, current->word) < 0) {
```

```
83              current->left = erase(current->left, word);
84          } else {
85              current->right = erase(current->right, word);
86          }
87          return current;
88      }
89
90      Node* find(Node *current, const char *word) {
91          if (!current) {
92              return nullptr;
93          }
94          if (strcasecmp(word, current->word) == 0) {
95              return current;
96          }
97          if (strcasecmp(word, current->word) < 0) {
98              return find(current->left, word);
99          } else {
100             return find(current->right, word);
101         }
102     }
103
104 public:
105     Treap() : root(nullptr) {
106         seed = time(nullptr);
107     }
108
109     ~Treap() {
110         while (root) {
111             erase(root->word);
112         }
113     }
114
115     void insert(const char *word, uint64_t value) {
116         if (strlen(word) >= MAX_WORD_LENGTH) {
117             std::cerr << "Word too long" << "\n";
118             return;
119         }
120         Node *new_node = new Node(word, value, &seed);
121         root = insert(root, new_node);
122     }
123
```

```
124     void erase(const char *word) {
125         root = erase(root, word);
126     }
127
128     Node* find(const char *word) {
129         return find(root, word);
130     }
131
132     void process_command(const char *command) {
133         char word[MAX_WORD_LENGTH];
134         uint64_t number;
135
136         if (sscanf(command, "+ %257s %lu", word, &number) == 2) {
137             Node *found = find(word);
138             if (found) {
139                 std::cout << "Exist" << std::endl;
140             } else {
141                 insert(word, number);
142                 std::cout << "OK" << std::endl;
143             }
144         } else if (sscanf(command, "- %257s", word) == 1) {
145             Node *found = find(word);
146             if (found) {
147                 erase(word);
148                 std::cout << "OK" << std::endl;
149             } else {
150                 std::cout << "NoSuchWord" << std::endl;
151             }
152         } else if (sscanf(command, "%257s", word) == 1) {
153             Node *found = find(word);
154             if (found) {
155                 std::cout << "OK: " << found->value << std::endl;
156             } else {
157                 std::cout << "NoSuchWord" << std::endl;
158             }
159         }
160     }
161 };
162
163 int main() {
164     Treap dictionary;
```

```
165     char command[512];
166
167     while (fgets(command, sizeof(command), stdin)) {
168         dictionary.process_command(command);
169     }
170
171     return 0;
172 }
```

# Пример работы

| Input | Output |
|---|---|
| + kek 123 | OK |
| + abacaba 228 | OK |
| + q 11223344 | OK |
| q | OK: 11223344 |
| - q | OK |
| q | NoSuchWord |
| abacaba | OK: 228 |
| + kek 123 | Exist |

# Вывод

В ходе выполнения лабораторной работы я успешно освоил реализацию декартового дерева. Эта уникальная структура данных, объединяющая свойства двоичного дерева поиска и кучи, предоставляет эффективные решения для различных задач.

Процесс реализации алгоритмов над декартовым деревом позволил мне усовершенствовать навыки программирования и углубить понимание алгоритмических принципов.