Министерство науки и высшего образования РФ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский Авиационный Институт»

Национальный Исследовательский Университет

Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №2

по курсу «Дискретный анализ»

Студент:	Хренникова А. С.
Группа:	М8О-208Б-19
Преподаватель:	Капралов Н. С.
Подпись:	
Оценка:	
Дата:	

Москва, 2020

Лабораторная работа №1

Задача: 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 264 - 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».

! Save /path/to/file — сохранить словарь в бинарном компактном представлении на диск в файл, указанный парамером команды. В случае успеха, программа должна вывести «OK», в случае неудачи выполнения операции, программа должна вывести описание ошибки (см. ниже).

! Load /path/to/file — загрузить словарь из файла. Предполагается, что файл был ранее подготовлен при помощи команды Save. В случае успеха, программа должна вывести строку «OK», а загруженный словарь должен заменить текущий (с которым происходит работа); в случае неуспеха, должна

быть выведена диагностика, а рабочий словарь должен остаться без изменений. Кроме системных ошибок, программа должна корректно обрабатывать случаи несовпадения формата указанного файла и представления данных словаря во внешнем файле.

Для всех операций, в случае возникновения системной ошибки (нехватка памяти, отсутствие прав записи и т.п.), программа должна вывести строку, начинающуюся с «ERROR:» и описывающую на английском языке возникшую ошибку.

Используемая структура данных: Красно-чёрное дерево.

1 Описание:

Для решения задачи был создан класс красно-черного дерева, в котором реализованы все основные функции для работы с деревом. Структура узла дерева состоит из данных, где хранятся ключ и значение, цвета, указателей на родителя и на обоих сыновей. Вставка, удаление и поиск производятся как в обычном бинарном дереве. После операций вставки и удаления выполняется балансировка для сохранения всех свойств красно-черного дерева.

Свойства красно-черных деревьев:

1) Каждый узел окрашен либо в красный, либо в черный цвет.
2) Корень окрашен в черный цвет.
3) Листья(так называемые NULL-узлы) окрашены в черный цвет.
4) Каждый красный узел должен иметь два черных дочерних узла. Нужно отметить, что у черного узла могут быть черные дочерние узлы. Красные узлы в качестве дочерних могут иметь только черные.
5) Пути от узла к его листьям должны содержать одинаковое количество черных узлов (черная высота).

Так же для балансировки написаны функции для правого и левого поворота дерева.

В файл, начиная с корня, для каждого узла записывается пара ключ-значение, цвет узла, потом идет запись левого и правого поддеревьев.

Операции с красно-черным деревом:

1. Поиск. Такой же, как и в обычном бинарном дереве поиска.
2. Вставка. Вставка нового элемента производится только в листья так же, как и вставка в бинарное дерево. Затем выполняется балансировка дерева.
3. Удаление. При удалении элемента с нелистовыми потомками, производится поиск либо минимального элемента в правом поддереве, либо максимального в левом, и ставится на место удаляемого. После выполняется балансировка дерева.

4. Запись в файл. Начиная с корня, для каждого узла записывается пара ключ-значение, цвет узла, потом идет запись левого и правого поддеревьев.
5. Чтение из файла. Данные считываются и дерево восстанавливается с той же структурой, с которой и было сохранено.

2 Исходный код:

```
#include <iostream>
#include <cstdint>
#include <cctype>
#include <cstring>
#include <cstdio>
#include <cerrno>
#include <fstream>

const int STR_SIZE = 257;

namespace NPair {
   const int STR_SIZE = 257;
   struct TPair {
      char First[STR_SIZE + 1];
      std::uint64_t Second;

      TPair() = default;
      TPair(char* first, std::uint64_t second);
      TPair(const TPair& p);
      TPair& operator=(const TPair& p);
      ~TPair() = default;
   };
}

namespace NPair{

   TPair::TPair(char* first, std::uint64_t second) {
      for (int i = 0; i < STR_SIZE; ++i) {
         First[i] = first[i];
      }
      Second = second;
   }

   TPair::TPair(const TPair& p) {
      for (int i = 0; i < STR_SIZE; ++i) {
         First[i] = p.First[i];
      }
      Second = p.Second;
   }

   TPair& TPair::operator=(const TPair& p) {
      for (int i = 0; i < STR_SIZE + 1; ++i) {
         First[i] = p.First[i];
      }
      Second = p.Second;
      return *this;
   }
}

namespace NRBTree {

   const int STR_SIZE = 257;

   struct TRBTreeNode {
```

```
        int Color;
        TRBTreeNode* Parent;
        TRBTreeNode* Left;
        TRBTreeNode* Right;
        NPair::TPair Data;

        TRBTreeNode(): Color(1), Parent(NULL), Left(NULL), Right(NULL), Data() {}
        TRBTreeNode(const NPair::TPair& p): Color(1), Parent(NULL), Left(NULL), Right(NULL),
Data(p) {}
        ~TRBTreeNode() = default;
    };

    class TRBTree {
        private:
            TRBTreeNode* Root;
            bool Search(char* key, NPair::TPair& res, TRBTreeNode* node);
            bool Insert(const NPair::TPair& data, TRBTreeNode* node);
            void Delete(TRBTreeNode* node);
            void FixDelete(TRBTreeNode* node, TRBTreeNode* nodeParent);
            void LeftRotate(TRBTreeNode* node);
            void RightRotate(TRBTreeNode* node);
            void FixInsert(TRBTreeNode* node);
            void DeleteTree(TRBTreeNode* node);
            static void Read(std::ifstream& fs, NRBTree::TRBTreeNode*& node, bool& ok);
            static void Write(std::ofstream& fs, NRBTree::TRBTreeNode* node, bool& ok);

        public:
            TRBTree(): Root(NULL) {};
            bool Search(char key[STR_SIZE], NPair::TPair& res);
            bool Insert(const NPair::TPair& data);
            bool Delete(const char key[STR_SIZE]);
            static void Load(const char path[STR_SIZE], NRBTree::TRBTree& t, bool& ok);
            static void Save(const char path[STR_SIZE], NRBTree::TRBTree& t, bool& ok);
            ~TRBTree();
    };
}

namespace NRBTree {

    bool TRBTree::Search(char key[STR_SIZE], NPair::TPair& res) {
        return Search(key, res, Root);
    }

    bool TRBTree::Search(char key[STR_SIZE], NPair::TPair& res, TRBTreeNode* node) {
        if (node == NULL) {
            return false;
        } else if (strcmp(key, node->Data.First) == 0) {
            res = node->Data;
            return true;
        } else {
            TRBTreeNode* to = (strcmp(key, node->Data.First) < 0) ? node->Left : node->Right;
            return Search(key, res, to);
        }
    }

    bool TRBTree::Insert(const NPair::TPair& data) {
        if (Root == NULL) {
```

```
        try {
            Root = new TRBTreeNode(data);
        } catch (std::bad_alloc& e) {
            std::cerr << "ERROR: " << e.what() << "\n";
            return false;
        }
        Root->Color = 1;
        return true;
    } else {
        return Insert(data, Root);
    }
}

bool TRBTree::Insert(const NPair::TPair& data, TRBTreeNode* node) {
    const char* key = data.First;
    if (strcmp(key, node->Data.First) == 0) {
        return false;
    } else if (strcmp(key, node->Data.First) < 0) {
        if (node->Left == NULL) {
            try {
                node->Left = new TRBTreeNode(data);
            } catch (std::bad_alloc& e) {
                std::cerr << "ERROR: " << e.what() << "\n";
                return false;
            }
            node->Left->Parent = node;
            node->Left->Color = 0;
            if (node->Color == 0) {
                FixInsert(node->Left);
            }
            return true;
        } else {
            return Insert(data, node->Left);
        }
    } else {
        if (node->Right == NULL) {
            try {
                node->Right = new TRBTreeNode(data);
            } catch (std::bad_alloc& e) {
                std::cerr << "ERROR: " << e.what() << "\n";
                return false;
            }
            node->Right->Parent = node;
            node->Right->Color = 0;
            if (node->Color == 0) {
                FixInsert(node->Right);
            }
            return true;
        } else {
            return Insert(data, node->Right);
        }
    }
}

void TRBTree::FixInsert(TRBTreeNode* node) {
    TRBTreeNode* grandParent = node->Parent->Parent;
    if (grandParent->Left == node->Parent) {
```

```
        if (grandParent->Right != NULL && grandParent->Right->Color == 0) {
            grandParent->Left->Color = 1;
            grandParent->Right->Color = 1;
            grandParent->Color = 0;
            if (Root == grandParent) {
                grandParent->Color = 1;
                return;
            }
            if (grandParent->Parent != NULL && grandParent->Color == 0 && grandParent->Parent-
>Color == 0) {
                FixInsert(grandParent);
            }
            return;
        } else if (grandParent->Right == NULL ||
        (grandParent->Right != NULL && grandParent->Right->Color == 1)) {
            if (node == node->Parent->Left) {
                grandParent->Color = 0;
                node->Parent->Color = 1;
                RightRotate(grandParent);
                return;
            } else {
                LeftRotate(node->Parent);
                node->Color = 1;
                node->Parent->Color = 0;
                RightRotate(node->Parent);
                return;
            }
        }
    } else {
        if (grandParent->Left != NULL && grandParent->Left->Color == 0) {
            grandParent->Right->Color = 1;
            grandParent->Left->Color = 1;
            grandParent->Color = 0;
            if (Root == grandParent) {
                grandParent->Color = 1;
                return;
            }
            if (grandParent->Parent != NULL && grandParent->Color == 0 && grandParent->Parent-
>Color == 0) {
                FixInsert(grandParent);
            }
            return;
        } else if (grandParent->Left == NULL ||
        (grandParent->Left != NULL && grandParent->Left->Color == 1)) {
            if (node == node->Parent->Right) {
                grandParent->Color = 0;
                node->Parent->Color = 1;
                LeftRotate(grandParent);
                return;
            } else {
                RightRotate(node->Parent);
                node->Color = 1;
                node->Parent->Color = 0;
                LeftRotate(node->Parent);
                return;
            }
        }
```

```
    }
}

bool TRBTree::Delete(const char key[STR_SIZE]) {
    TRBTreeNode* node = Root;
    while (node != NULL && strcmp(key, node->Data.First) != 0) {
        TRBTreeNode* to = (strcmp(key, node->Data.First) < 0) ? node->Left : node->Right;
        node = to;
    }
    if (node == NULL) {
        return false;
    }
    Delete(node);
    return true;
}

void TRBTree::Delete(TRBTreeNode* node) {
    TRBTreeNode* toDelete = node;
    int toDeleteColor = toDelete->Color;
    TRBTreeNode* toReplace;
    TRBTreeNode* toReplaceParent;
    if (node->Left == NULL) {
        toReplace = node->Right;
        if (toReplace != NULL) {
            toReplace->Parent = node->Parent;
            toReplaceParent = node->Parent;
        } else {
            toReplaceParent = node->Parent;
            if (node == Root) {
                toReplaceParent = NULL;
                Root = NULL;
            }
        }
        if (node->Parent != NULL) {
            if (node->Parent->Left == node) {
                node->Parent->Left = toReplace;
            } else {
                node->Parent->Right = toReplace;
            }
        } else {
            Root = toReplace;
        }
    } else if (node->Right == NULL) {
        toReplace = node->Left;
        toReplace->Parent = node->Parent;
        toReplaceParent = node->Parent;
        if (node->Parent != NULL) {
            if (node->Parent->Left == node) {
                node->Parent->Left = toReplace;
            } else {
                node->Parent->Right = toReplace;
            }
        } else {
            Root = toReplace;
        }
    } else {
        TRBTreeNode* minInRight = node->Right;
```

```
        while(minInRight->Left != NULL) {
            minInRight = minInRight->Left;
        }
        toDelete = minInRight;
        toDeleteColor = toDelete->Color;
        toReplace = toDelete->Right;
        if (toDelete->Parent == node) {
            if (toReplace != NULL) {
                toReplace->Parent = toDelete;
            }
            toReplaceParent = toDelete;
        } else {
            toDelete->Parent->Left = toReplace;
            if (toReplace != NULL) {
                toReplace->Parent = toDelete->Parent;
            }
            toReplaceParent = toDelete->Parent;
            toDelete->Right = node->Right;
            toDelete->Right->Parent = toDelete;
        }
        if (node->Parent != NULL) {
            if (node->Parent->Left == node) {
                node->Parent->Left = toDelete;
            } else {
                node->Parent->Right = toDelete;
            }
        } else {
            Root = toDelete;
        }
        toDelete->Parent = node->Parent;
        toDelete->Left = node->Left;
        toDelete->Left->Parent = toDelete;
        toDelete->Color = node->Color;
    }
    if (toDeleteColor == 1) {
        FixDelete(toReplace, toReplaceParent);
    }
    delete node;
}

void TRBTree::FixDelete(TRBTreeNode* node, TRBTreeNode* nodeParent) {
    while ((node == NULL || node->Color == 1) && node != Root) {
        TRBTreeNode* brother;
        if (node == nodeParent->Left) {
            brother = nodeParent->Right;
            if (brother->Color == 0) {
                brother->Color = 1;
                nodeParent->Color = 0;
                LeftRotate(nodeParent);
                brother = nodeParent->Right;
            }
            if (brother->Color == 1) {
                if ((brother->Left == NULL || brother->Left->Color == 1)
                && (brother->Right == NULL || brother->Right->Color == 1)) {
                    brother->Color = 0;
                    node = nodeParent;
                    if (node != NULL) {
```

```
                nodeParent = node->Parent;
            }
        } else {
            if (brother->Right == NULL || brother->Right->Color == 1) {
                brother->Left->Color = 1;
                brother->Color = 0;
                RightRotate(brother);
                brother = nodeParent->Right;
            }
            brother->Color = nodeParent->Color;
            nodeParent->Color = 1;
            brother->Right->Color = 1;
            LeftRotate(nodeParent);
            break;
        }
    }
} else {
    brother = nodeParent->Left;
    if (brother->Color == 0) {
        brother->Color = 1;
        nodeParent->Color = 0;
        RightRotate(nodeParent);
        brother = nodeParent->Left;
    }
    if (brother->Color == 1) {
        if ((brother->Right == NULL || brother->Right->Color == 1)
        && (brother->Left == NULL || brother->Left->Color == 1)) {
            brother->Color = 0;
            node = nodeParent;
            if (node != NULL) {
                nodeParent = node->Parent;
            }
        } else {
            if (brother->Left == NULL || brother->Left->Color == 1) {
                brother->Right->Color = 1;
                brother->Color = 0;
                LeftRotate(brother);
                brother = nodeParent->Left;
            }
            brother->Color = nodeParent->Color;
            nodeParent->Color = 1;
            brother->Left->Color = 1;
            RightRotate(nodeParent);
            break;
        }
    }
}
}
if (node != NULL) {
    node->Color = 1;
}
}

void TRBTree::LeftRotate(TRBTreeNode* node) {
TRBTreeNode* rightSon = node->Right;
if (rightSon == NULL) {
    return;
```

```
    }
    node->Right = rightSon->Left;
    if (rightSon->Left != NULL) {
        rightSon->Left->Parent = node;
    }
    rightSon->Parent = node->Parent;
    if (node->Parent == NULL) {
        Root = rightSon;
    } else if (node == node->Parent->Left) {
        node->Parent->Left = rightSon;
    } else {
        node->Parent->Right = rightSon;
    }
    rightSon->Left = node;
    node->Parent = rightSon;
}

void TRBTree::RightRotate(TRBTreeNode* node) {
    TRBTreeNode* leftSon = node->Left;
    if (leftSon == NULL) {
        return;
    }
    node->Left = leftSon->Right;
    if (leftSon->Right != NULL) {
        leftSon->Right->Parent = node;
    }
    leftSon->Parent = node->Parent;
    if (node->Parent == NULL) {
        Root = leftSon;
    } else if (node == node->Parent->Right) {
        node->Parent->Right = leftSon;
    } else {
        node->Parent->Left = leftSon;
    }
    leftSon->Right = node;
    node->Parent = leftSon;
}

void TRBTree::Read(std::ifstream& fs, NRBTree::TRBTreeNode*& node, bool& ok) {
    if (!ok) {
        return;
    }
    NPair::TPair data;
    short len = 0;
    fs.read((char*)&len, sizeof(short));
    if (fs.bad()) {
        std::cerr << "Unable to read from file\n";
        ok = false;
        return;
    }
    if (len == -1) {
        return;
    } else if (len != -1 && (len <= 0 || len > STR_SIZE - 1)) {
        std::cerr << "ERROR: Wrong file format\n";
        ok = false;
        return;
    }
```

```
    char color;
    for (int i = 0; i < len; ++i) {
        fs.read(&(data.First[i]), sizeof(char));
        if (isalpha(data.First[i]) == 0) {
            std::cerr << "ERROR: Wrong file format\n";
            ok = false;
            return;
        }
    }
    if (fs.bad()) {
        std::cerr << "Unable to read from file\n";
        ok = false;
        return;
    }
    data.First[len] = '\0';
    fs.read((char*)&data.Second, sizeof(std::uint64_t));
    if (fs.bad()) {
        std::cerr << "Unable to read from file\n";
        ok = false;
        return;
    }
    fs.read((char*)&color, sizeof(char));
    if (fs.bad()) {
        std::cerr << "Unable to read from file\n";
        ok = false;
        return;
    }
    try {
        node = new TRBTreeNode(data);
    } catch (std::bad_alloc& e) {
        std::cerr << "ERROR: " << e.what() << "\n";
        ok = false;
        return;
    }
    if (color == 'r') {
        node->Color = 0;
    } else if (color == 'b') {
        node->Color = 1;
    } else {
        std::cout << "ERROR: Wrong file format\n";
        ok = false;
        return;
    }
    Read(fs, node->Left, ok);
    Read(fs, node->Right, ok);
    if (node->Left != NULL) {
        node->Left->Parent = node;
    }
    if (node->Right != NULL) {
        node->Right->Parent = node;
    }
}

void TRBTree::Load(const char* path, NRBTree::TRBTree& t, bool& ok) {
    std::ifstream fs;
    fs.open(path, std::ios::binary);
    if (fs.fail()) {
```

```
        std::cerr << "ERROR: Unable to open file " << path << " in read mode\n";
        ok = false;
        return;
    }
    Read(fs, t.Root, ok);
    fs.close();
    if (fs.fail()) {
        std::cerr << "ERROR: Unable to close file " << path << "\n";
        ok = false;
        return;
    }
}

void TRBTree::Write(std::ofstream& fs, NRBTree::TRBTreeNode* t, bool& ok) {
    if (!ok) {
        return;
    }
    short len = 0;
    if (t == NULL) {
        len = -1;
        fs.write((char*)&len, sizeof(short));
        if (fs.bad()) {
            std::cerr << "Unable to write in file\n";
            ok = false;
            return;
        }
        return;
    }
    char color = t->Color == 1 ? 'b' : 'r';
    for (int i = 0; i < STR_SIZE - 1 && t->Data.First[i] != '\0' && isalpha(t->Data.First[i]) != 0; ++i) {
        len++;
    }
    fs.write((char*)&len, sizeof(short));
    if (fs.bad()) {
        std::cerr << "Unable to write in file\n";
        ok = false;
        return;
    }
    fs.write(t->Data.First, sizeof(char) * len);
    if (fs.bad()) {
        std::cerr << "Unable to write in file\n";
        ok = false;
        return;
    }
    fs.write((char*)&(t->Data.Second), sizeof(std::uint64_t));
    if (fs.bad()) {
        std::cerr << "Unable to write in file\n";
        ok = false;
        return;
    }
    fs.write((char*)&color, sizeof(char));
    if (fs.bad()) {
        std::cerr << "Unable to write in file\n";
        ok = false;
        return;
    }
    Write(fs, t->Left, ok);
```

```
        Write(fs, t->Right, ok);
    }

    void TRBTree::Save(const char* path, NRBTree::TRBTree& t, bool& ok) {
        std::ofstream fs;
        fs.open(path, std::ios::binary);
        if (fs.fail()) {
            std::cerr << "ERROR: Unable to open file " << path << " in write mode\n";
            ok = false;
            return;
        }
        Write(fs, t.Root, ok);
        fs.close();
        if (fs.fail()) {
            std::cerr << "ERROR: Unable to close file " << path << "\n";
            ok = false;
            return;
        }
    }

    void TRBTree::DeleteTree(TRBTreeNode* node) {
        if (node == NULL) {
            return;
        } else {
            DeleteTree(node->Left);
            DeleteTree(node->Right);
            delete node;
        }
    }

    TRBTree::~TRBTree() {
        DeleteTree(Root);
        Root = NULL;
    }
}

void Low(char* str) {
    for (int i = 0; i < STR_SIZE && str[i] != '\0'; ++i) {
        str[i] = std::tolower(str[i]);
    }
}

int main() {
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(NULL);
    char req[STR_SIZE];
    NRBTree::TRBTree* tPtr = new NRBTree::TRBTree;
    while (std::cin >> req) {
        if (strcmp(req,"+") == 0) {
            char key[STR_SIZE];
            std::uint64_t val;
            std::cin >> key >> val;
            Low(key);
            if (tPtr->Insert({key, val})) {
                std::cout << "OK\n";
            } else {
                std::cout << "Exist\n";
```

```
            }
        } else if (strcmp(req,"-") == 0) {
            char key[STR_SIZE];
            std::cin >> key;
            Low(key);
            if (tPtr->Delete(key)) {
                std::cout << "OK\n";
            } else {
                std::cout << "NoSuchWord\n";
            }
        } else if (strcmp(req,"!") == 0) {
            char path[STR_SIZE];
            std::cin >> req;
            std::cin.get();
            std::cin.getline(path, STR_SIZE - 1, '\n');
            bool ok = true;
            if (strcmp(req,"Save") == 0) {
                NRBTree::TRBTree::Save(path, *tPtr, ok);
                if (ok) {
                    std::cout << "OK\n";
                }
            }  else {
                NRBTree::TRBTree* tmpTreePtr = new NRBTree::TRBTree;
                NRBTree::TRBTree::Load(path, *tmpTreePtr, ok);
                if (ok) {
                    std::cout << "OK\n";
                    delete tPtr;
                    tPtr = tmpTreePtr;
                } else {
                    delete tmpTreePtr;
                }
            }
        } else {
            Low(req);
            NPair::TPair ans;
            if (tPtr->Search(req, ans)) {
                std::cout << "OK: " << ans.Second << "\n";
            } else {
                std::cout << "NoSuchWord\n";
            }
        }
    }
    delete tPtr;
    return 0;
}
```

3 Консоль:

```
lina_tucha@LAPTOP-44CRFC1U:~/labs/da/lab2/02$ cat test
+ a 1
+ A 2
+ aa 18446744073709551615
aa
A
- A
A
lina_tucha@LAPTOP-44CRFC1U:~/labs/da/lab2/02$ ./123 <test
OK
Exist
OK
OK: 18446744073709551615
OK: 1
OK
NoSuchWord
lina_tucha@LAPTOP-44CRFC1U:~/labs/da/lab2/02$ cat file1
+
ssjsfooxxehatujpdtdqsfqprygklnchjmaodqnbuweqspgvjjoeqgtkhbvtoabamdprtfvqcahurqsbcihtqcgxdcttfw
urcmmxriovkytcpmftwmpnowotbkqginbmanltxcpidjkuyrpvefmvedpgnioxvplvfyrdbilhtxcsprpwaesehkkut
cvprhoaggdjqrqlrteinvhnaashkffghbwckneqvhcnbtytpdedalt 56243402
+ okatrhesgidtmvpvxdatetliarkllmyaybvqicjokokymdxlixhnttxtnihy 1082272647
+ bwjwpucclmqwoecorkobxiwxekinlgvmfhmucowo 1872453353
+
ntwpiqawrbhpynakboqwbxhpsjgsxmllgidryeqshbiiolsqajpdjwvfjcajqlxatbrufknolyxbkqrmdjpmjlrsqrchda
hycbvjmmbxlybxpsksfaiooajfuloaowcsxaekmflxemxwiipplaeabnhxbvxr 1450475118
bwjwpucclmqwoecorkobxiwxekinlgvmfhmucowo
- bwjwpucclmqwoecorkobxiwxekinlgvmfhmucowo
okatrhesgidtmvpvxdatetliarkllmyaybvqicjokokymdxlixhnttxtnihy
- okatrhesgidtmvpvxdatetliarkllmyaybvqicjokokymdxlixhnttxtnihy
ssjsfooxxehatujpdtdqsfqprygklnchjmaodqnbuweqspgvjjoeqgtkhbvtoabamdprtfvqcahurqsbcihtqcgxdcttfw
urcmmxriovkytcpmftwmpnowotbkqginbmanltxcpidjkuyrpvefmvedpgnioxvplvfyrdbilhtxcsprpwaesehkkut
cvprhoaggdjqrqlrteinvhnaashkffghbwckneqvhcnbtytpdedalt
-
ssjsfooxxehatujpdtdqsfqprygklnchjmaodqnbuweqspgvjjoeqgtkhbvtoabamdprtfvqcahurqsbcihtqcgxdcttfw
urcmmxriovkytcpmftwmpnowotbkqginbmanltxcpidjkuyrpvefmvedpgnioxvplvfyrdbilhtxcsprpwaesehkkut
cvprhoaggdjqrqlrteinvhnaashkffghbwckneqvhcnbtytpdedalt
ntwpiqawrbhpynakboqwbxhpsjgsxmllgidryeqshbiiolsqajpdjwvfjcajqlxatbrufknolyxbkqrmdjpmjlrsqrchda
hycbvjmmbxlybxpsksfaiooajfuloaowcsxaekmflxemxwiipplaeabnhxbvxr
-
ntwpiqawrbhpynakboqwbxhpsjgsxmllgidryeqshbiiolsqajpdjwvfjcajqlxatbrufknolyxbkqrmdjpmjlrsqrchda
hycbvjmmbxlybxpsksfaiooajfuloaowcsxaekmflxemxwiipplaeabnhxbvxr
! Save file
! Load file
lina_tucha@LAPTOP-44CRFC1U:~/labs/da/lab2/02$ ./solution <file1
OK
OK
OK
OK
OK: 1872453353
OK
OK: 1082272647
OK
OK: 56243402
OK
```

OK: 1450475118
OK
OK
OK

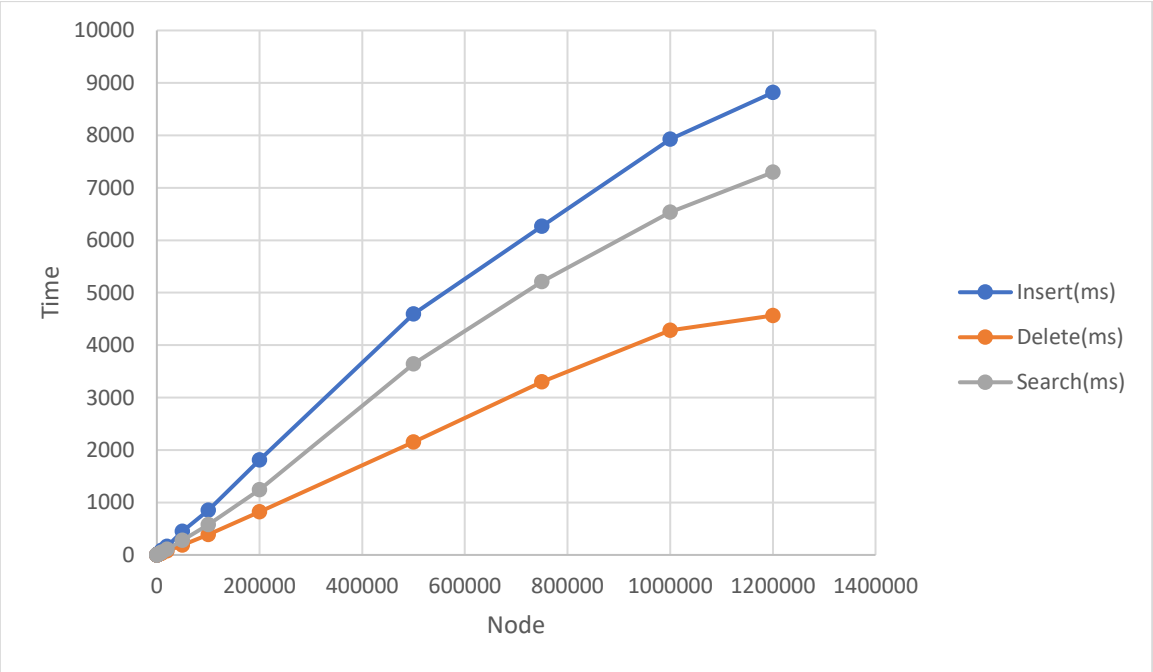
4 Тест производительности:

Красно-черное дерево с n внутренними вершинами (т. е. не считая листьев) имеет высоту не больше 2lg(n+1).

Добавление вершины в красно-черное дерево производится за время O(lg n). При балансировке цикл повторяется максимально O(lg n) раз, и общее время работы есть O(lg n). При этом выполняется не более двух вращений (после которых производится выход из цикла).

Удаление вершины из красно-черного дерева требует времени O(lg n). Процедура балансировки после удаления тоже требует O(lg n) времени (при этом производится не более трех вращений).

Поиск не отличается от поиска в бинарном дереве. Время: O(lg n).



Красно-черное дерево сравнивается с std::map.

Тесты для 100, 1000, 10000, 100000 вставок/удалений/поисков.

lina_tucha@LAPTOP-44CRFC1U:~/labs/da/lab2/02$ ./map <file1

insert: 240ms

delete: 207ms

search: 144ms

lina_tucha@LAPTOP-44CRFC1U:~/labs/da/lab2/02$ ./rbtree <file1

insert: 361ms

delete: 108ms

search: 200ms

lina_tucha@LAPTOP-44CRFC1U:~/labs/da/lab2/02$ ./map <file1

insert: 2370ms

delete: 2101ms

search: 1891ms

lina_tucha@LAPTOP-44CRFC1U:~/labs/da/lab2/02$ ./rbtree <file1

insert: 3717ms

delete: 1065ms

search: 2070ms

lina_tucha@LAPTOP-44CRFC1U:~/labs/da/lab2/02$ ./map <file1

insert: 27494ms

delete: 22878ms

search: 20930ms

lina_tucha@LAPTOP-44CRFC1U:~/labs/da/lab2/02$ ./rbtree <file1

insert: 40402ms

delete: 17832ms

search: 27353ms

lina_tucha@LAPTOP-44CRFC1U:~/labs/da/lab2/02$ ./map <file1

insert: 344525ms

delete: 278006ms

search: 274826ms

lina_tucha@LAPTOP-44CRFC1U:~/labs/da/lab2/02$ ./rbtree <file1

insert: 465271ms

delete: 193293ms

search: 297864ms

5 Выводы:

Выполняя вторую лабораторную работу, я познакомилась с такой структурой данных как красно-черное дерево. Это структура данных, предназначенная для хранения, быстрого доступа и изменения данных. Лучше всего применять красно-черное дерево для данных, чьи ключи можно легко и быстро сравнивать. Для ключей, где используются длинные строки, скорость работы операций над деревом значительно ниже.