Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа № 2-3 по курсу дискретного анализа: Словарь

Студент: А. О. Тояков
Преподаватель: Н. А. Зацепин
Группа: М8О-307Б-18
Дата:
Оценка:
Подпись:

Москва
2020

Условие

1. 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.

2. Красно-чёрное дерево. Тип ключа: последовательность букв английского алфавита длиной не более 256 символов. Тип значения: числа от 0 до $2^{64} - 1$.

Метод решения

В данной работе нужно было реализовать 5 операций: поиск в дереве, удаление элемента из дерева, добавление элемента в дерево, загрузка структуры из бинарного файла, сохранение в бинарный файл.

1. Поиск в дереве: поиск осуществляется так же, как и в BST, то есть мы сравниваем значения по ключу, начиная с корня и идём в левого сына, если искомый ключ меньше данного или в правого сына иначе, пока ключи не совпадут или мы не дойдём до листа (Nil вершина).

2. Добавление в дерево: для начала ищется позиция для вставки алгоритмом, который представлен выше. После этого мы присваиваем Nil вершины листам, не забываем про указатель на родителя и присваиваем красный цвет новой вершине. После этого запускается функция коррекции вставки. В ней для начала проверяется является ли наша вершина корнем. Если да то она перекрашивается в чёрный цвет (корень всегда чёрный). Затем мы смотрим на родителя вершины. Если он чёрный, то ни одно правило не нарушается и мы выходим из алгоритма. После этого стоит условие проверки цвета дяди, то есть брата отца нашей вершины. Если он красный то мы перекрашиваем его и нашего отца в чёрный цвет, а дедушку в красный и запускаем эту функцию рекурсивно от дедушки. Если дядя не оказался красным, то всё исправляется с помощью максимум двух поворотов и конечного перекрашивания.

3. Удаление из дерева: Если удаляется красная вершина, то удаление просиходит по обычным правилам удаления вершины из BST. Если удаляется чёрная вершина, то мы таким же образом удаляем её из дерева и запускаем fix-функцию, где может быть 4 случая: 1) мы берём брата fix-элемента и если он красный, то делаем его чёрным, родителя красным и делаем левый поворот относительно родителя, а затем переприсаиваем брата. 2) Если у брата оба сына чёрные, то перекрашиваем его в чёрный и переходим к родителю. 3) Если у брата правый сын чёрный, то делаем его второго сына чёрным а самого красным и делаем правый поворот относительно брата и переприсваиваем брата. 4) В последнем случае перекрашиваем брата в

цвет родителя, родителя в чёрный, а правого сына брата в чёрный и делаем левый поворот относительно родителя. В конце алгоритма переприсваиваем fix-элемент на корень и делаем его чёрным. Так проходим по вершине циклично пока она не станет корнем или не станет чёрной. Итого максимум может быть 3 поворота. Выше описан случай, когда fix-вершина является левым сыном родителя. Также есть зеркальный случай, когда она правый сын.

4. Загрузка и сохранение дерева: сохраняю из загружаю соответственно в бинарный файл. Первый аргумент наш файл. Второй - корень при сохранении и Nil вершина при загрузке. При сохранении все данные для каждой вершины поочереди записываются в файл. Перемещение между вершинами осущетсвлено так: создаются две булевые переменные, которые принимают true и false в зависимости от того есть ли у нашей вершины левый и правый сын. При значении true рекурсивно запускается функция сохранения иначе выход. При загрузке сначала очищается всё текущее дерево и затем таким же образом, как при сохранении вершины записываются в дерево. Функция возвращает вершину n, которая по итогу окажется корнем нашего красно-чёрного дерева.

Описание программы

Программа реализована одним файлом. В первую очередь хотелось бы описать параметры класса TRBTree. Этот класс шаблонный, где параметры шаблона это ключ K и значение V. Параметры класса: корневой узел Root, Nil узел (по сути общий лист, указатели всех листьев будут присвоены ему) и количетсво узлов. Что касается узла, то он представляет из себя структуру, где есть ключ типа K, значение типа V, enum переменная цвета R (red) или B (black) и три указателя: на родителя, на левого сына и на правого сына. Для узла определён конструктор. Что касается самого дерева, то в нём реализованы конструктор, деструктор, 5 основных функций, а также вспомогательные. Вспомогательными функциями являются: нахождение дяди, fix вставки, fix удаления, замещение элемента другим, левый и правый повороты, а также несколько функций словаря таких, как сохранение в файл, загрузка из файла, поиск по словарю, добавление и удаление элемента в ловарь. Что касается основных функций, то примерные алгоритмы описаны выше.

Исходный код

```
#include <iostream>
#include <cstring>
#include <fstream>
#include <algorithm>

const size_t BUF_SIZE = 257;
```

```
class TString {
    char* Str;
    size_t Size;
    using iterator = char *;
public:
    TString() : Str(nullptr), Size(0) {}

    TString(const char* s) {
        Size = strlen(s);
        Str = new char[Size + 1];
        std::copy(s, s + Size, Str);
        Str[Size] = '\0';
    }

    TString(const TString& s) {
        Size = s.Size;
        Str = new char[Size + 1];
        std::copy(s.Str, s.Str + Size, Str);
        Str[Size] = '\0';
    }

    char* String() {
        return Str;
    }

    void Move(char* s) {
delete[] Str;
Str = s;
Size = strlen(s);
}

    iterator begin() {
        return Str;
    }

    const iterator begin() const {
        return Str;
    }

    iterator end() {
        if (Str)
            return Str + Size;
```

```
        return nullptr;
    }

    const iterator end() const {
        if (Str)
            return Str + Size;
        return nullptr;
    }

    size_t StrSize() const {
        return Size;
    }

    char operator[](size_t idx) {
        return Str[idx];
    }

    const char operator[](size_t idx) const {
        return Str[idx];
    }

    friend std::ostream& operator<<(std::ostream& out, const TString& s);
    friend std::istream& operator>>(std::istream& in, const TString& s);

    TString& operator= (const TString& s) {
delete[] Str;
Size = s.Size;
Str = new char[Size + 1];
        std::copy(s.Str, s.Str + Size, Str);
        Str[Size] = '\0';
        return *this;
}

TString& operator= (const char* s) {
delete[] Str;
Size = strlen(s);
        Str = new char[Size + 1];
        std::copy(s, s + Size, Str);
        Str[Size] = '\0';
return *this;
}
```

```
    ~TString() {
        delete[] Str;
        Str = nullptr;
        Size = 0;
    }
};

bool operator<(const TString& comp1, const TString& comp2) {
    size_t minSize;
    comp1.StrSize() < comp2.StrSize() ? (minSize = comp1.StrSize()) : (minSize = comp2.S
    for (size_t i = 0; i < minSize; i++) {
        if (comp1[i] != comp2[i])
            return (comp1[i] < comp2[i]);
    }
    return comp1.StrSize() < comp2.StrSize();
}

bool operator==(const TString& comp1, const TString& comp2) {
    return !(comp1 < comp2) && !(comp2 < comp1);
}

bool operator!=(const TString& comp1, const TString& comp2) {
    return !(comp1 == comp2);
}

std::ostream& operator<<(std::ostream& out, const TString& s) {
    for (auto ch : s)
        out << ch;
    return out;
}

std::istream& operator>>(std::istream& in, TString& s) {
    char buf[BUF_SIZE];
    if (in >> buf)
        s = buf;
    return in;
}

using std::cout;
using std::cin;
using std::endl;
```

```
template <typename K, typename V>
class TRBTree {
    enum TColor {R, B};

    struct TNode {
        TColor Color;
        K Key;
        V Value;
        TNode* P;
        TNode* Left;
        TNode* Right;

        TNode(const K& nodeKey, const V& nodeVal) :
        Color(R), Key(nodeKey), Value(nodeVal), P(nullptr), Left(nullptr), Right(nullptr

        TNode() : P(nullptr), Left(nullptr), Right(nullptr) {}
    };

    TNode* Root;
    TNode* Nil;
    size_t Size;

    TNode* Uncle(TNode* n) {
        if (n->P == Nil) {
            return Nil;
        }
        if (n->P->P == Nil) {
            return Nil;
        }
        if (n->P->P->Right == n->P) {
            return n->P->P->Left;
        }
        return n->P->P->Right;
    }

public:
    TRBTree() {
        Nil = new TNode;
        Root = Nil;
        Root->Color = B;
    }
    TRBTree(const K& rootKey, const V& rootVal) {
```

```
    Nil = new TNode;
    Nil->Color = B;
    Root = new TNode(rootKey, rootVal);
    Root->Color = B;
    Root->P = Nil;
    Root->Left = Nil;
    Root->Right = Nil;
    Size++;
}

void Clean(TNode* n) {
    if (n == Nil)
        return ;
    Clean(n->Left);
    Clean(n->Right);
    delete n;
}

~TRBTree() {
    Clean(Root);
    delete Nil;
    Size = 0;
}

void Insert(const K& key, const V& val) {
    TNode* n = new TNode(key, val);
    TNode* tmp = Root;
    TNode* p = tmp;
    if (tmp != Nil) {
        while (tmp != Nil) {
            p = tmp;
            if (n->Key < tmp->Key) {
                tmp = tmp->Left;
            } else {
                tmp = tmp->Right;
            }
        }
        if (n->Key < p->Key) {
            p->Left = n;
        } else {
            p->Right = n;
        }
```

```
    } else {
        Root = n;
    }
    n->Color = R;
    n->P = p;
    n->Left = Nil;
    n->Right = Nil;
    Size++;
    InsertFix(n);
}

void InsertFix(TNode* ins) {
    if (ins == Root) {
        ins->Color = B;
        return ;
    }
    if (ins->P->Color == B) {
        return ;
    }
    TNode* unc = Uncle(ins);
    if (unc->Color == R) { //1
        unc->Color = B;
        ins->P->Color = B;
        ins->P->P->Color = R;
        InsertFix(ins->P->P); //max h/2 iterations
    } else {
        if (ins == ins->P->Right && ins->P == ins->P->P->Left) { //2
            LeftRotate(ins->P);
            ins = ins->Left;
        } else if (ins == ins->P->Left && ins->P == ins->P->P->Right) { //3
            RightRotate(ins->P);
            ins = ins->Right;
        }
        if (ins == ins->P->Left) { //4
            RightRotate(ins->P->P);
            ins->P->Color = B;
            ins->P->Right->Color = R;
        } else if (ins == ins->P->Right) { //5
            LeftRotate(ins->P->P);
            ins->P->Color = B;
            ins->P->Left->Color = R;
        }
```

```
    }
}

void Replace(TNode* n, TNode* rep) {
    if (n == Root)
        Root = rep;
    else
        (n->P->Left == n) ? (n->P->Left = rep) : (n->P->Right = rep);
    rep->P = n->P;
}

void Erase(K& key) {
    TNode* n = FindNode(key);
    if (n != Nil) {
        Erase(n);
    }
}

void Erase(TNode* n) {
    TNode* del = n;
    TNode* fixNode = Nil;
    TColor delColor = n->Color;
    if (n->Left == Nil) {
        fixNode = n->Right;
        Replace(n, n->Right);
    } else if (n->Right == Nil) {
        fixNode = n->Left;
        Replace(n, n->Left);
    } else {
        TNode* tmp = n->Right;
        while (tmp->Left != Nil) {
            tmp = tmp->Left;
        }
        del = tmp;
        delColor = del->Color;
        fixNode = del->Right;
        if (del->P == n) {
            fixNode->P = del;
        } else {
            Replace(del, del->Right);
            del->Right = n->Right;
            del->Right->P = del;
```

```
        }
        Replace(n, del);
        del->Left = n->Left;
        del->Left->P = del;
        del->Color = n->Color; //node replacement (changing Key and Value but Color
    }
    if (delColor == B) {
        EraseFix(fixNode);
    }
    delete n;
}

void EraseFix(TNode* n) {
    while (n != Root && n->Color == B) {
        if (n->P->Left == n) { //n is Left child
            TNode* bro = n->P->Right;
            if (bro->Color == R) { //1
                bro->Color = B;
                bro->P->Color = R;
                LeftRotate(bro->P);
                bro = n->P->Right;
            }
            if (bro->Left->Color == B && bro->Right->Color == B) { //2
                bro->Color = R;
                n = n->P;
            } else {
                if (bro->Right->Color == B) { //3
                    bro->Left->Color = B;
                    bro->Color = R;
                    RightRotate(bro);
                    bro = n->P->Right;
                }
                bro->Color = bro->P->Color; //4
                bro->P->Color = B;
                bro->Right->Color = B;
                LeftRotate(bro->P);
                n = Root;
            }
        } else { //n is Right child
            TNode* bro = n->P->Left;
            if (bro->Color == R) {
                bro->Color = B;
```

```
                bro->P->Color = R;
                RightRotate(bro->P);
                bro = n->P->Left;
            }
            if (bro->Left->Color == B && bro->Right->Color == B) {
                bro->Color = R;
                n = n->P;
            } else {
                if (bro->Left->Color == B) {
                    bro->Right->Color = B;
                    bro->Color = R;
                    LeftRotate(bro);
                    bro = n->P->Left;
                }
                bro->Color = bro->P->Color;
                bro->P->Color = B;
                bro->Left->Color = B;
                RightRotate(bro->P);
                n = Root;
            }
        }
    }
    n->Color = B;
}

void RightRotate(TNode* pivot) {
    TNode* tmp = pivot->Left;
    pivot->Left = tmp->Right;
    if (tmp->Right != Nil)
        tmp->Right->P = pivot;
    tmp->P = pivot->P;
    if (pivot->P == Nil) {
        Root = tmp;
    } else {
        if (pivot->P->Right == pivot) {
            pivot->P->Right = tmp;
        } else {
            pivot->P->Left = tmp;
        }
    }
    tmp->Right = pivot;
    pivot->P = tmp;
```

```
}

void LeftRotate(TNode* pivot) {
    TNode* tmp = pivot->Right;
    pivot->Right = tmp->Left;
    if (tmp->Left != Nil)
        tmp->Left->P = pivot;
    tmp->P = pivot->P;
    if (pivot->P == Nil) {
        Root = tmp;
    } else {
        if (pivot->P->Right == pivot) {
            pivot->P->Right = tmp;
        } else {
            pivot->P->Left = tmp;
        }
    }
    tmp->Left = pivot;
    pivot->P = tmp;
}

void Print() {
    PrintNode(Root, 0);
    cout << endl << endl;
}

void PrintNode(TNode* n, size_t shift) {
    if (n == Nil) {
        return ;
    }
    PrintNode(n->Right, shift + 7);
    for (size_t s = 0; s < shift; s++) {
        cout << " ";
    }
    cout << n->Key << ": " << n->Value << " ";
    n->Color == R ? (cout << "Red" << endl) : (cout << "Black" << endl);
    PrintNode(n->Left, shift + 7);
}

TNode* FindNode(K& key) {
    TNode* n = Root;
    while (n->Key != key && n != Nil) {
```

```
            if (key < n->Key) {
                n = n->Left;
            } else {
                n = n->Right;
            }
        }
        return n;
    }

//vocabulary
    void VocAdd() {
        K key;
        V val;
        cin >> key >> val;
        std::transform(key.begin(), key.end(), key.begin(), ::tolower);
        TNode* n = FindNode(key);
        if (n == Nil) {
            Insert(key, val);
            cout << "OK\n";
        } else {
            cout << "Exist\n";
        }
    }

    void VocErase() {
        K key;
        cin >> key;
        std::transform(key.begin(), key.end(), key.begin(), ::tolower);
        TNode* n = FindNode(key);
        if (n != Nil) {
            Erase(n);
            cout << "OK\n";
        } else {
            cout << "NoSuchWord\n";
        }
    }

    void VocFind(K& key) {
        std::transform(key.begin(), key.end(), key.begin(), ::tolower);
        TNode* n = FindNode(key);
        if (n != Nil) {
            cout << "OK: " << n->Value << endl;
```

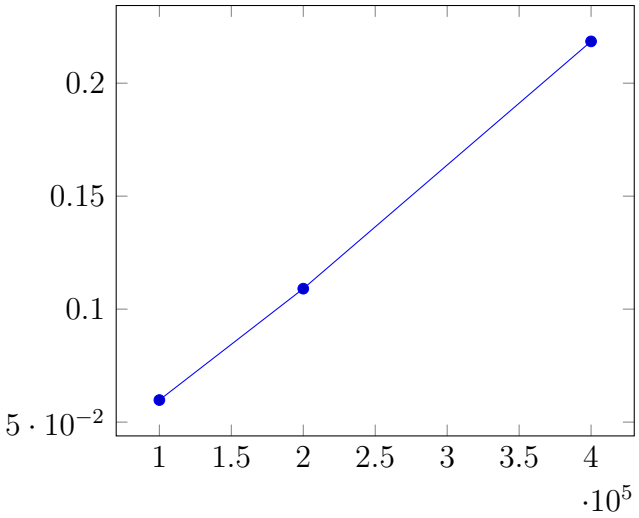
```
    } else {
        cout << "NoSuchWord\n";
    }
}

void VocSave() {
    TString path;
    cin >> path;
    std::ofstream out;
    out.open(path.String(), std::ios_base::binary | std::ios_base::trunc);
    if (out.is_open()) {
        Save(out, Root);
        out.close();
        cout << "OK\n";
    } else {
        cout << "ERROR: couldn't open the output file!\n";
    }
}

void VocLoad() {
    TString path;
    cin >> path;
    std::ifstream in;
    in.open(path.String(), std::ios_base::binary);
    if (in.is_open()) {
        Clean(Root);
        Root = Load(in, Nil);
        in.close();
        cout << "OK\n";
    } else {
        cout << "ERROR: couldn't open the input file!\n";
    }
}

void Save(std::ofstream& out, TNode* root) {
    if (root == Nil)
        return ;
    size_t keySize = root->Key.StrSize();
    out.write((char*)&keySize, sizeof(keySize));
    out.write(root->Key.String(), keySize);
    out.write((char*)&root->Value, sizeof(root->Value));
    out.write((char*)&root->Color, sizeof(root->Color));
```

```
        bool hasLeft = (root->Left != Nil);
        bool hasRight = (root->Right != Nil);
        out.write((char*)&hasLeft, sizeof(hasLeft));
        out.write((char*)&hasRight, sizeof(hasRight));
        if (hasLeft)
            Save(out, root->Left);
        if (hasRight)
            Save(out, root->Right);
    }

    TNode* Load(std::ifstream& in, TNode* par) {
        TNode* n = Nil;
        size_t keySize;
        V val;
        TColor c;
        bool hasLeft = false;
        bool hasRight = false;

        in.read((char*)(&keySize), sizeof(keySize));
        if (in.gcount() == 0)
return n;
        char* key = new char[keySize + 1];
        key[keySize] = '\0';
        in.read(key, keySize);
        in.read((char*)(&val), sizeof(val));
        in.read((char*)(&c), sizeof(c));
        in.read((char*)(&hasLeft), sizeof(hasLeft));
        in.read((char*)(&hasRight), sizeof(hasRight));

        n = new TNode();
        n->Key.Move(key);
        key = nullptr;
        n->Value = val;
        n->Color = c;
        n->P = par;
        if (hasLeft) {
            n->Left = Load(in, n);
        } else {
            n->Left = Nil;
        }
        if (hasRight) {
            n->Right = Load(in, n);
```

```
        } else {
            n->Right = Nil;
        }
        return n;
    }
};

int main() {
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(nullptr);
    TRBTree<TString, unsigned long long> voc;
    TString input;
    TString saveOrLoad;
    while (cin >> input) {
        switch (input[0]) {
            case '+':
                voc.VocAdd();
                break;
            case '-':
                voc.VocErase();
                break;
            case '!':
                cin >> saveOrLoad;
                if (saveOrLoad == "Save")
                    voc.VocSave();
                else if (saveOrLoad == "Load")
                    voc.VocLoad();
                break;
            default:
                voc.VocFind(input);
                break;
        }
    }
    return 0;
}
```

Тест производительности

В данных тестах используются строки только из 50 элементов, сгенерированных рандомно. Поэтому возможен случай, что некоторые ключи будут совпадать и не будут добавлены в дерево.



Пояснения к графику: Ось y - время в секундах. Ось x - количество строк.

Выводы

В ходе данной работы я познакомился со структурой данных RBTree. Она представляет из себя самобалансирущееся бинарное дерево поиска. Эта структура довольно экономична по памяти, так как дополнительно каждая вершина хранит только один бит информации (цвет). В среднем случае красно-чёрное дерево будет работать медленне, чем AVL, но при больших объёмах данных оно может быть эффективнее. Также в данной работе был реализован простой бенчмарк, с помощью которого я смог протестировать функцию вставки.