МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Факультет Автоматики и Вычислительной Техники

Кафедра Вычислительной Техники

Лабораторная работа

по дисциплине «Кодирование и передача информации»

«**2-3 Дерево**»

**Группа:** АВТ-610

**Студент:** Дунаев Н. Ю.

**Преподаватель:** Катасонов Д. Н.

г. Новосибирск

2019

1. **Введение**

**2-3 дерево**(англ. *2-3 tree*) — структура данных, представляющая собой сбалансированное дерево поиска, такое что из каждого узла может выходить две или три ветви и глубина всех листьев одинакова. Является частным случаем [B+ дерева](https://neerc.ifmo.ru/wiki/index.php?title=B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE#B.2B-.D0.B4.D0.B5.D1.80.D0.B5.D0.B2.D0.BE).

Свойства:

2-3 дерево — сбалансированное дерево поиска, обладающее следующими свойствами:

* Нелистовые вершины имеют либо 2, либо 3 сына,
* Нелистовая вершина, имеющая двух сыновей, хранит максимум левого поддерева. Нелистовая вершина, имеющая трех сыновей, хранит два значения. Первое значение хранит максимум левого поддерева, второе максимум центрального поддерева,
* сыновья упорядочены по значению максимума поддерева сына,
* все листья лежат на одной глубине,
* высота 2-3 дерева O(log n), где n — количество элементов в дереве.

1. **Ход работы**

В качестве языка программирования для реализации шаблонного списка был выбран язык «C++». При выполнении работы был реализован шаблонный класс «Tree», реализующий функционал 2-3 дерева. В состав класса «Tree» входит вложенный класс «Node», реализующий функционал узла. Программная реализация включает в себя файл «Tree.h», где описаны поля и методы классов, а также файл «main.cpp», где продемонстрированы такие операции, как вставка, удаление, поиск и вывод дерева в консоль.

1. **Описание основных методов класса**

bool insert(Key key) – Вставка ключа в дерево

Принимает на вход ключ для вставки.

На выходе *true*, если ключ удалось вставить, иначе *false*.

Вызывает приватный метод *privateInsert*, реализующий алгоритм вставки в 2-3 дерево (см. **п. 4.1**).

bool del(Key key) – Удаление ключа из дерева

Принимает на вход ключ для удаления

На выходе *true*, если ключ удалось удалить, иначе *false*.

Вызывает приватный метод *privateDelete*, реализующий алгоритм удаления из 2-3 дерева (см. **п. 4.2**).

Bool search(Key key) – Опрос наличия ключа в дереве

Принимает на вход ключ для поиска

На выходе *true*, если ключ найден, иначе *false*.

Вызывает приватный метод *privateSearch*, реализующий алгоритм поиска в 2-3 дереве (см. **п. 4.3**).

void print() – Вывод дерева в консоль

Не имеет входных и выходных параметров.

Вызывает приватный метод *privatePrint,* реализующий рекурсивный обход дерева с печатью узлов

1. **Описание алгоритмов вставки, удаления и поиска**
   1. **Описание алгоритма вставки в 2-3 дерево**

При вставке в 2-3 дерево (рекурсивный алгоритм) могут возникнуть следующие ситуации:

* Вставка в пустое дерево

Создаем новый узел, записываем в него ключ.

* Вставка в узел без потомков (далее «лист»)

Добавляем ключ и вызываем метод коррекции дерева, описанный ниже.

* Ключ для вставки меньше минимального ключа в узле

Вызываем метод *privateInsert* для первого поддерева.

* Ключ для вставки больше минимального и меньше максимального ключей в узле, или больше единственного ключа в узле

Вызываем метод *privateInsert* для второго поддерева.

* Ключ для вставки больше любого из ключей в узле

Вызываем метод *privateInsert* для третьего поддерева.

***Метод коррекции дерева***

Рекурсивный метод. Входной и выходной параметры – узлы дерева. Дерево будет перестроено, если узел имеет три ключа (теоретически, узел 2-3 дерева не может иметь три ключа, но в целях упрощения реализации алгоритмов узел может временно иметь три ключа и четырех потомков). Второй по величине ключ записывается в родительский узел, если таковой имеется. Если родительского узла не существует, создается новый узел. Указатели на потомков перезаписываются.

* 1. **Описание алгоритма удаления из 2-3 дерева**

При помощи метода *privateSearch* (см. **п. 4.3**)ищем нужный ключ. Далее могут возникнуть следующие ситуации:

* Ключ не найден

Ничего не делаем.

* Дерево имеет единственный узел с единственным ключом

Удаляем вершину без дальнейшей коррекции дерева.

* Удаляем из листа, имеющего два ключа

Удаляем ключ без дальнейшей коррекции дерева.

* Удаляем ключ из вершины, имеющей единственный ключ
* Хотя бы у одного из братьев есть 2 ключа, у родителя один или два ключа

Циклично сдвигаем ключи, перекидывая указатели.

* У всех братьев 1 ключ, у родителя 1 ключ

Переносим ключ родителя к ключу оставшегося сына.

* У всех братьев 1 ключ, у родителя 2 ключа

Переносим один из ключей родителя к соответствующему сыну.

* 1. **Описание алгоритма поиска в 2-3 дереве**

При поиске ключа в 2-3 дереве (рекурсивный алгоритм) могут возникнуть следующие ситуации:

* Ключ найден

Возвращаем ссылку на узел

* Ключ для поиска меньше минимального ключа в узле

Вызываем метод *privateSearch* для первого поддерева.

* Ключ для поиска больше минимального и меньше максимального ключей в узле, или больше единственного ключа в узле

Вызываем метод *privateSearch* для второго поддерева.

* Ключ для поиска больше любого из ключей в узле

Вызываем метод *privateSearch* для третьего поддерева.

* Ключ не найден

Возвращаем *NULL.*

1. **Выводы**

В ходе лабораторной работы на языке программирования «C++» был реализован шаблонный класс 2-3 дерева, а так же методы для работы с ним, такие, как вставка, удаление, поиск и вывод в консоль.

1. **Листинг программы:**

*Tree.h*

#include <iostream>

using namespace std;

template <class Key>

class Tree{

private:

class Node{

public:

int size;

Key key[3];

Node \*first;

Node \*second;

Node \*third;

Node \*fourth;

Node \*parent;

Node();

Node(Key key);

Node(Key key, Node \*first, Node \*second, Node \*third, Node \*fourth, Node \*parent);

~Node();

bool find(Key key);

void sort();

void addKey(Key key);

void deleteKey(Key key);

void to2Node(Key key, Node \*first, Node \*second);

bool isLeaf();

void print();

};

Node \*root;

int size;

Node \*rebuild(Node \*root);

Node \*privateInsert(Node \*root, Key key);

Node \*privateDelete(Node \*root, Key key);

Node \*getNode(Node \*root, Key key);

Node \*privateSearch(Node \*root, Key key);

Node \*getMin(Node \*root);

Node \*fix(Node \*root);

Node \*merge(Node \*root);

Node \*copy(Node \*root);

Node \*redistribute(Node \*root);

void privatePrint(Node \*root, int level);

public:

Tree();

Tree(const Tree<Key> &tree);

~Tree();

bool insert(Key key);

bool del(Key key);

bool search(Key key);

void print();

int getSize();

};

/////////////////////////////////////////////////////////////////////////////// Методы Node

template<class Key>

void Tree<Key>::Node::deleteKey(Key k)

{

if (size >= 1 && key[0] == k) {

key[0] = key[1];

key[1] = key[2];

size--;

}

else if (size == 2 && key[1] == k) {

key[1] = key[2];

size--;

}

}

template<class Key>

bool Tree<Key>::Node::isLeaf(){

return (first == NULL) && (second == NULL) && (third == NULL);

}

template<class Key>

void Tree<Key>::Node::print(){

for (int i = 0; i < size; ++i)

cout << key[i] << " ";

}

template<class Key>

void Tree<Key>::Node::addKey(Key k){

key[size] = k;

int l = size;

if (l > 0){

int tmp;

while (key[l - 1] > key[l]){

tmp = key[l - 1];

key[l - 1] = key[l];

key[l] = tmp;

l -= 1;

}

}

size++;

}

/////////////////////////////////////////////////////////////////////////////// Методы Tree

template<class Key>

int Tree<Key>::getSize(){

return size;

}

template<class Key>

void Tree<Key>::print(){

privatePrint(this->root, 0);

}

template<class Key>

void Tree<Key>::privatePrint(Node \*root, int level){

if (root == NULL)

return;

privatePrint(root->first, level + 1);

for (int i = 0; i < 10 \* level; i++)

cout << " ";

root->print();

cout << endl;

privatePrint(root->second, level + 1);

cout << endl;

privatePrint(root->third, level + 1);

}

template<class Key>

typename Tree<Key>::Node\* Tree<Key>::copy(Node \*root){

if (!root)

return NULL;

Node \*newRoot = new Node(root->key[0], root->first, root->second, root->third, root->fourth, root->parent);

newRoot->key[1] = root->key[1];

newRoot->first = copy(newRoot->first);

newRoot->second = copy(newRoot->second);

newRoot->third = copy(newRoot->third);

return newRoot;

}

template<class Key>

typename Tree<Key>::Node\* Tree<Key>::rebuild(Node \*root){

if (root->size < 3)

return root;

Node \*x = new Node(root->key[0], root->first, root->second, NULL, NULL, root->parent);

Node \*y = new Node(root->key[2], root->third, root->fourth, NULL, NULL, root->parent);

if (x->first) x->first->parent = x;

if (x->second) x->second->parent = x;

if (y->first) y->first->parent = y;

if (y->second) y->second->parent = y;

if (root->parent) {

root->parent->addKey(root->key[1]);

if (root->parent->first == root) root->parent->first = NULL;

else if (root->parent->second == root) root->parent->second = NULL;

else if (root->parent->third == root) root->parent->third = NULL;

if (root->parent->first == NULL) {

root->parent->fourth = root->parent->third;

root->parent->third = root->parent->second;

root->parent->first = x;

root->parent->second = y;

}

else if (root->parent->second == NULL) {

root->parent->fourth = root->parent->third;

root->parent->third = y;

root->parent->second = x;

}

else {

root->parent->third = x;

root->parent->fourth = y;

}

Node \*tmp = root->parent;

delete root;

return tmp;

}

else {

Node \*newRoot = new Node();

newRoot->addKey(root->key[1]);

x->parent = y->parent = newRoot;

newRoot->first = x;

newRoot->second = y;

delete root;

return newRoot;

}

}

template<class Key>

typename Tree<Key>::Node\* Tree<Key>::privateInsert(Node \*root, Key key){

if (!root){

Node \*tmp = new Node();

tmp->addKey(key);

return tmp;

}

else if (root->isLeaf()){

root->addKey(key);

size++;

}

else if (root->key[0] >= key){

privateInsert(root->first, key);

}

else if ((root->size == 1) || ((root->size == 2) && (root->key[1] >= key))){

privateInsert(root->second, key);

}

else{

privateInsert(root->third, key);

}

return rebuild(root);

}

template<class Key>

typename Tree<Key>::Node\* Tree<Key>::fix(Node \*root)

{

if (root->size == 0 && root->parent == NULL) { // Удаляем единственный ключ

delete root;

return NULL;

}

if (root->size != 0) { // Удаление из вершины с двумя ключами

if (root->parent)

return fix(root->parent);

else

return root;

}

Node \*parent = root->parent;

if (parent->first->size == 2 || parent->second->size == 2 || parent->size == 2)

root = redistribute(root); // Сдвиг ключей

else if (parent->size == 2 && parent->third->size == 2)

root = redistribute(root);

else

root = merge(root); // Перенос ключа из родителя к оставшемуся сыну

return fix(root);

}

template<class Key>

typename Tree<Key>::Node\* Tree<Key>::redistribute(Node \*root){

Node \*parent = root->parent;

Node \*first = parent->first;

Node \*second = parent->second;

Node \*third = parent->third;

if ((parent->size == 2) && (first->size < 2) && (second->size < 2) && (third->size < 2)) {

if (first == root) {

parent->first = parent->second;

parent->second = parent->third;

parent->third = NULL;

parent->first->addKey(parent->key[0]);

parent->first->third = parent->first->second;

parent->first->second = parent->first->first;

if (root->first != NULL) parent->first->first = root->first;

else if (root->second != NULL) parent->first->first = root->second;

if (parent->first->first != NULL) parent->first->first->parent = parent->first;

parent->deleteKey(parent->key[0]);

delete first;

}

else if (second == root) {

first->addKey(parent->key[0]);

parent->deleteKey(parent->key[0]);

if (root->first != NULL) first->third = root->first;

else if (root->second != NULL) first->third = root->second;

if (first->third != NULL) first->third->parent = first;

parent->second = parent->third;

parent->third = NULL;

delete second;

}

else if (third == root) {

second->addKey(parent->key[1]);

parent->third = NULL;

parent->deleteKey(parent->key[1]);

if (root->first != NULL) second->third = root->first;

else if (root->second != NULL) second->third = root->second;

if (second->third != NULL) second->third->parent = second;

delete third;

}

}

else if ((parent->size == 2) && ((first->size == 2) || (second->size == 2) || (third->size == 2))) {

if (third == root) {

if (root->first != NULL) {

root->second = root->first;

root->first = NULL;

}

root->addKey(parent->key[1]);

if (second->size == 2) {

parent->key[1] = second->key[1];

second->deleteKey(second->key[1]);

root->first = second->third;

second->third = NULL;

if (root->first != NULL) root->first->parent = root;

}

else if (first->size == 2) {

parent->key[1] = second->key[0];

root->first = second->second;

second->second = second->first;

if (root->first != NULL) root->first->parent = root;

second->key[0] = parent->key[0];

parent->key[0] = first->key[1];

first->deleteKey(first->key[1]);

second->first = first->third;

if (second->first != NULL) second->first->parent = second;

first->third = NULL;

}

}

else if (second == root) {

if (third->size == 2) {

if (root->first == NULL) {

root->first = root->second;

root->second = NULL;

}

second->addKey(parent->key[1]);

parent->key[1] = third->key[0];

third->deleteKey(third->key[0]);

second->second = third->first;

if (second->second != NULL) second->second->parent = second;

third->first = third->second;

third->second = third->third;

third->third = NULL;

}

else if (first->size == 2) {

if (root->second == NULL) {

root->second = root->first;

root->first = NULL;

}

second->addKey(parent->key[0]);

parent->key[0] = first->key[1];

first->deleteKey(first->key[1]);

second->first = first->third;

if (second->first != NULL) second->first->parent = second;

first->third = NULL;

}

}

else if (first == root) {

if (root->first == NULL) {

root->first = root->second;

root->second = NULL;

}

first->addKey(parent->key[0]);

if (second->size == 2) {

parent->key[0] = second->key[0];

second->deleteKey(second->key[0]);

first->second = second->first;

if (first->second != NULL) first->second->parent = first;

second->first = second->second;

second->second = second->third;

second->third = NULL;

}

else if (third->size == 2) {

parent->key[0] = second->key[0];

second->key[0] = parent->key[1];

parent->key[1] = third->key[0];

third->deleteKey(third->key[0]);

first->second = second->first;

if (first->second != NULL) first->second->parent = first;

second->first = second->second;

second->second = third->first;

if (second->second != NULL) second->second->parent = second;

third->first = third->second;

third->second = third->third;

third->third = NULL;

}

}

}

else if (parent->size == 1) {

root->addKey(parent->key[0]);

if (first == root && second->size == 2) {

parent->key[0] = second->key[0];

second->deleteKey(second->key[0]);

if (root->first == NULL) root->first = root->second;

root->second = second->first;

second->first = second->second;

second->second = second->third;

second->third = NULL;

if (root->second != NULL) root->second->parent = root;

}

else if (second == root && first->size == 2) {

parent->key[0] = first->key[1];

first->deleteKey(first->key[1]);

if (root->second == NULL) root->second = root->first;

root->first = first->third;

first->third = NULL;

if (root->first != NULL) root->first->parent = root;

}

}

return parent;

}

template<class Key>

typename Tree<Key>::Node\* Tree<Key>::merge(Node \*root){

Node \*parent = root->parent;

if (parent->first == root) {

parent->second->addKey(parent->key[0]);

parent->second->third = parent->second->second;

parent->second->second = parent->second->first;

if (root->first != NULL) parent->second->first = root->first;

else if (root->second != NULL) parent->second->first = root->second;

if (parent->second->first != NULL) parent->second->first->parent = parent->second;

parent->deleteKey(parent->key[0]);

delete parent->first;

parent->first = NULL;

}

else if (parent->second == root) {

parent->first->addKey(parent->key[0]);

if (root->first != NULL) parent->first->third = root->first;

else if (root->second != NULL) parent->first->third = root->second;

if (parent->first->third != NULL) parent->first->third->parent = parent->first;

parent->deleteKey(parent->key[0]);

delete parent->second;

parent->second = NULL;

}

if (parent->parent == NULL) {

Node \*tmp = NULL;

if (parent->first != NULL) tmp = parent->first;

else tmp = parent->second;

tmp->parent = NULL;

delete parent;

return tmp;

}

return parent;

}

template<class Key>

typename Tree<Key>::Node\* Tree<Key>::privateDelete(Node \*root, Key key){

Node \*tmp = privateSearch(root, key);

if (!tmp)

return root;

Node \*min = NULL;

if (root->key[0] == key)

min = getMin(tmp->second);

else

min = getMin(tmp->third);

if (min){

int &tmpKey = (key == tmp->key[0] ? tmp->key[0] : tmp->key[1]);

int swap = min->key[0];

min->key[0] = tmpKey;

tmpKey = swap;

tmp = min;

}

tmp->deleteKey(key);

size--;

return fix(tmp);

}

template<class Key>

typename Tree<Key>::Node\* Tree<Key>::getMin(Node \*root)

{

if (!root)

return root;

if (!(root->first))

return root;

else

return getMin(root->first);

}

template<class Key>

typename Tree<Key>::Node\* Tree<Key>::privateSearch(Node \*root, Key key){

if (!root)

return NULL;

if (root->size == 1){

if (key == root->key[0])

return root;

else if (key < root->key[0])

return privateSearch(root->first, key);

else

return privateSearch(root->second, key);

}

else{

if (key == root->key[0] || key == root->key[1])

return root;

else if (key < root->key[0])

return privateSearch(root->first, key);

else if (key > root->key[0] && key < root->key[1])

return privateSearch(root->second, key);

else

return privateSearch(root->third, key);

}

}

template<class Key>

bool Tree<Key>::insert(Key key){

int tmp = size;

this->root = privateInsert(root, key);

return (tmp != size);

}

template<class Key>

bool Tree<Key>::del(Key key){

int tmp = size;

this->root = privateDelete(root, key);

return (tmp != size);

}

template<class Key>

bool Tree<Key>::search(Key key){

return privateSearch(root, key) ? 1 : 0;

}

/////////////////////////////////////////////////////////////////////////////// Конструкторы

template<class Key>

Tree<Key>::Node::Node(){

key[0] = key[1] = key[2] = NULL;

first = second = third = fourth = parent = NULL;

size = 0;

}

template<class Key>

Tree<Key>::Node::Node(Key k, Node \*first, Node \*second, Node \*third, Node \*fourth, Node \*parent){

key[0] = k;

key[1] = key[2] = NULL;

this->first = first;

this->second = second;

this->third = third;

this->fourth = fourth;

this->parent = parent;

size = 1;

}

template<class Key>

Tree<Key>::Tree(){

root = NULL;

size = 0;

}

template<class Key>

Tree<Key>::Tree(const Tree<Key> &tree){

this->root = NULL;

this->size = 0;

this->root = copy(tree.root);

}