**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра информационных систем**

отчет

**по курсовой работе №2**

**по дисциплине «Программирование»**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 2373 |  | Маркова З.А. |
| Преподаватель |  | Глущенко А. Г. |

Санкт-Петербург

2023

**Цель работы.**

Цели работы: изучение свойств и организации деревьев как структуры данных; получение практических навыков в работе с бинарным деревом поиска; определение преимуществ и недостатков структуры данных вида дерева; проведение сравнительной характеристики скорости вставки, удаления и поиска элемента в различных структурах данных.

**Основные теоретические положения.**

Бинарное дерево поиска – бинарное дерево, которое обладает дополнительными свойствами. Если дерево организовано таким образом, что для каждого узла все ключи левого поддерева меньше ключа этого узла, а все ключи его правого поддерева больше, то его можно называть бинарным деревом поиска.

Иными словами, данные в бинарном дереве поиска хранятся в отсортированном виде. Одинаковые ключи не допускаются. При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется.

**Постановка задачи**

Необходимо реализовать программу, которая выполняет следующие действия.

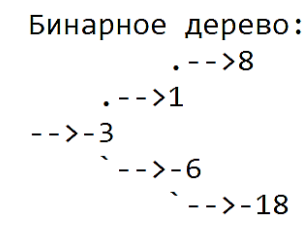
1. Формирование бинарного дерева из N элементов:

a) пользователь вводит количество элементов N бинарного дерева, которое автоматически заполняется случайными числами (–99 до 99);

б) пользователь вводит в консоль элементы массива, N определяется автоматически по количеству введенных элементов;

в) элементы считываются с файла, в котором хранится массив из чисел, N – количество элементов в файле.

Определение скорости формирования бинарного дерева.

2. Вывод в консоль и файл tree бинарного дерева. Бинарное дерево должно иметь подобный вид:

3. Определение скорости вставки, удаления и получения элемента дерева. В отчете сравните скорость работы бинарного дерева с линейной структурой (двусвязным списком или динамическим массивом) и сделайте выводы.

4. Прямой обход, обратный обход и обход в ширину бинарного дерева.

5. Генерация заданий к практической работе по бинарным деревьям. Необходимо сгенерировать задания в файл output\_task в количестве вариантов, которые введет пользователь. В файл output\_key необходимо вывести короткие ответы к заданиям. В файл output\_ans необходимо вывести развернутые ответы к заданиям. Должны быть представлены задания на создание бинарного дерева поиска из массива значений, удаление и вставка элементов в бинарное дерево поиска.

**Выполнение работы.**

Код программы представлен в приложении А.

**Выводы.**

В данной лабораторной работе мы изучили свойства и организацию деревьев как структуры данных; получили практические навыки в работе с бинарным деревом поиска; определили преимуществ и недостатки структуры данных вида дерева; провели сравнительную характеристику скорости вставки, удаления и поиска элемента в различных структурах данных.

Приложение А

рабочий код

#include <iostream>

using namespace std;

#include <ctime>

#include <string>

#include <chrono>

#include <queue>

using namespace chrono;

struct Node

{

int data;

Node\* left; //указатель на левого потомка

Node\* right; //указатель на правого потомка

};

struct Trunk

{

Trunk \*prev;

string str;

Trunk(Trunk \*prev, string str)

{

this->prev = prev;

this->str = str;

}

};

int tabs = 0;

void Add(int value, Node\*& root)

{

if (root==NULL)

{

root = new Node;

root->left = NULL;

root->right = NULL;

root->data = value;

return;

}

else // если узел инициализирован

{

if (value < root->data) // иначе, если искомое ключевое значение меньше ключевого значения узла

Add(value, root->left); // вызывается эта же функция для левого потомка.

else if (value > root->data) // иначе

Add(value, root->right); // вызывается эта же функция для правого потомка.

}

}

void showTrunks(Trunk \*p)//функция вывода связей дерева

{

if (p == nullptr)//если нет поддеревьев

return;

showTrunks(p->prev);//выводим предыдущий узел

cout << p->str;//выводим отступы и связи

}

void printTree(Node\* root, Trunk \*prev, bool isRight)

{

if (root == nullptr)//пустое дерево

return;

string prev\_str = " ";//отступ по уровням (длина как для стрелки)

Trunk \*trunk = new Trunk(prev, prev\_str);//новая связь

printTree(root->right, trunk, true);//правое поддерево

if (!prev)//если нет предыдущего узла (предка) -> корень дерева

trunk->str = "———";//связь корня дерева

else if (isRight)//если правое поддерево

{

trunk->str = ".———";//связь правого поддерева

prev\_str = " |";//в отступ по уровням добавляем черту (дерево идет вширь)

}

else {//в противном случае - левое поддерево

trunk->str = "`———";//левое поддерево

prev->str = prev\_str;//отступ по уровням не меняется

}

showTrunks(trunk);//выводим связи дерева - стебли

cout << " " << root->data << endl;//выводим значение узла

if (prev) //задаем строку отступов для узла, если есть поддеревья

prev->str = prev\_str;

trunk->str = " |";//в отступ по уровням добавляем черту (дерево идет вширь)

printTree(root->left, trunk, false);//левое поддерево

}

void Insert(Node \*(&root), int value)

{

if (root == NULL) {

root = new Node;

root->data = value;

root->left = NULL;

root->right = NULL;

}

else {

if (root->data == value) return;

if (value < root->data) {

Insert(root->left, value);

}

else if (value > root->data) {

Insert(root->right, value);

}

}

}

Node\* GetMax(Node\* root) {

while (root->right) {

root = root->right;

}

return root;

}

int k;

void Del(Node\* (&root), int data) {

if (root == NULL)

{

cout << "Значение не найдено\n";

return;

}

if (data < root->data)

Del(root->left, data);

else if (data > root->data)

Del(root->right, data);

else if (data==root->data) {

if (root->left == NULL and root->right == NULL) {

delete root;

root = NULL;

}

else if (root->right == NULL) {

Node\* del= root;

root = root->left;

delete del;

}

else if (root->left == NULL) {

Node\* del = root;

root = root->right;

delete del;

}

else {

Node \*max = GetMax(root->left);

root->data = max->data;

Del(root->left, max->data);

}

}

}

void Get(Node\* (&root), int data) {

if (root == NULL)

{

k = 100;

cout << "Значение не найдено\n";

return;

}

if (data < root->data)

Get(root->left, data);

else if (data > root->data)

Get(root->right, data);

if (data == root->data)

k = root->data;

}

void Direct(Node\* root) {

if (root) {

cout << root->data << " ";

Direct(root->left);

Direct(root->right);

}

}

void Reverse(Node\* root) {

if (root) {

Reverse(root->left);

Reverse(root->right);

cout << root->data << " ";

}

}

void Wide(Node\* root)

{

queue <Node\*> q;

q.push(root);

while (!q.empty()) {

Node \*tmp = q.front();

q.pop();

cout << tmp->data << " ";

//Если есть левый наследник, то помещаем его в очередь для дальнейшей обработки

if (tmp->left) {

q.push(tmp->left);

}

//Если есть правый наследник, то помещаем его в очередь для дальнейшей обработки

if (tmp->right) {

q.push(tmp->right);

}

}

}

void FreeTree(Node\* root)

{

if (!root) return;

FreeTree(root->left);

FreeTree(root->right);

delete root;

return;

}

int main()

{

setlocale(0, "");

srand(time(NULL));

time\_point<steady\_clock> start;

time\_point<steady\_clock> end;

time\_point<steady\_clock> start1;

time\_point<steady\_clock> end1;

time\_point<steady\_clock> start2;

time\_point<steady\_clock> end2;

time\_point<steady\_clock> start3;

time\_point<steady\_clock> end3;

time\_point<steady\_clock> start4;

time\_point<steady\_clock> end4;

Node\* Root = NULL;

while (true)

{

int choose;

int level = 0;

cout << "\n1)Создать дерево\n2)Вставить элемент\n3)Удалить элемент\n4)Получить элемент\n5)Прямой обход\n6)Обратный обход\n7)Обход в ширину\n0)Выход\n";

cout << "Ваш выбор: ";

cin >> choose;

if (choose == 0)

break;

switch (choose)

{

case 1:

int choose1;

cout << "\n1)Ввести количество элементов\n2)Ввести с клавиатуры\n";

cout << "Ваш выбор: ";

cin >> choose1;

switch (choose1)

{

case 1:

FreeTree(Root);

Root = NULL;

int n;

cout << "\nВведите кол-во элементов: ";

cin >> n;

int a;

start = chrono::steady\_clock::now();

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

{

Add(rand() % 199 - 99, Root);

}

end = chrono::steady\_clock::now();

printTree(Root, nullptr, false);

cout << "\nЗатраченное время: " << duration\_cast<microseconds>(end - start).count() << " мкс.\n";

break;

case 2:

FreeTree(Root);

Root = NULL;

cout << "\nЧтобы прекратить введите 00";

cout << endl;

string data;

cin >> data;

while (data != "00" )

{

start1 = chrono::steady\_clock::now();

Add(stoi(data), Root);

end1 = chrono::steady\_clock::now();

cin >> data;

}

printTree(Root, nullptr, false);

cout << "\nЗатраченное время: " << duration\_cast<microseconds>(end1 - start1).count() << " мкс.\n";

break;

}

break;

case 2:

{

int value;

cout << "\nВведите значение элемента, который хотите вставить: ";

cin >> value;

start2 = chrono::steady\_clock::now();

Insert(Root, value);

end2 = chrono::steady\_clock::now();

printTree(Root, NULL, false);

cout << "\nЗатраченное время: " << duration\_cast<microseconds>(end2 - start2).count() << " мкс.\n";

break;

}

case 3:

{

int value;

cout << "\nВведите значение элемента, который хотите удалить: ";

cin >> value;

start4 = chrono::steady\_clock::now();

Del(Root, value);

end4 = chrono::steady\_clock::now();

printTree(Root, NULL, false);

cout << "\nЗатраченное время: " << duration\_cast<microseconds>(end4 - start4).count() << " мкс.\n";

break;

}

case 4:

{

int value;

cout << "\nВведите значение элемента: ";

cin >> value;

start3 = chrono::steady\_clock::now();

Get(Root, value);

end3 = chrono::steady\_clock::now();

if (k!=100)

cout << "Значение найдено: " << k<<"\n";

k = 0;

cout << "Затраченное время: " << duration\_cast<microseconds>(end3 - start3).count() << " мкс.\n";

break;

}

case 5:

{

Direct(Root);

break;

}

case 6:

{

Reverse(Root);

break;

}

case 7:

{

Wide(Root);

break;

}

}

}

cout << endl;

}