**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Вычислительной техники**

отчет

**по лабораторной работе №4**

**по дисциплине «Алгоритмы и структуры данных»**

Тема: «**ДЕРЕВЬЯ ДВОИЧНОГО ПОИСКА»**

|  |  |  |
| --- | --- | --- |
| Студенты гр. 7307 | Торопов В.А. |  |
| Преподаватель | Колинько П.Г. |  |

Санкт-Петербург

2019

Оглавление

[Цель работы 3](#__RefHeading___Toc188_4150899979)

[Задание 3](#__RefHeading___Toc190_4150899979)

[Обоснование по выбору размера хэш-таблицы и коэффициентов хэш-функции 3](#__RefHeading___Toc302_2376616454)

[Оценка временной сложности 3](#__RefHeading___Toc967_306870553)

[Результаты работы программы 3](#__RefHeading___Toc969_306870553)

[Код программы 6](#__RefHeading___Toc306_2376616454)

[Вывод 9](#__RefHeading___Toc308_2376616454)

Цель работы

Научиться работать с деревьями двоичного поиска.

Задание

Переделать программу, составленную по теме «Хеш-таблицы», под использование деревьев двоичного поиска.

Формула для вычислений: A \ (B ∩ C) ∪ D ⊕ E

средняя мощность множества: 10.

Оценка временной сложности

# 

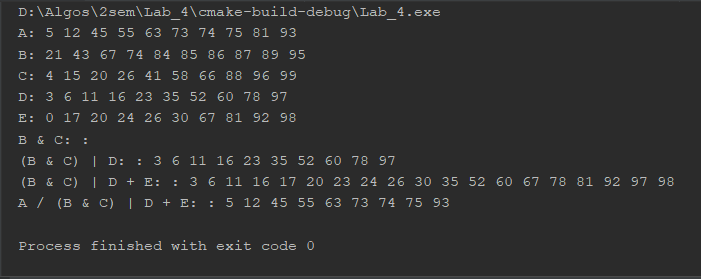
ДДП — это дерево с нагруженными узлами, вес в любом узле которого больше любого веса в левом его поддереве и не больше любого веса в правом поддереве. Количество шагов алгоритма поиска элемента множества в таком дереве не превышает его высоты, т. е. имеет сложность O(log n). Такую же сложность имеют операции вставки нового элемента в дерево и удаления элемента.

Двуместные операции над множествами в ДДП выполняются, используя примитивы проверка–вставка–удаление. Очевидно, что временная сложность двуместной операции при этом будет в среднем *O*(*n* log *n*).

Вывод.

ДДП сами по себе малопригодны для хранения множеств с повторениями: дубликаты ключей искажают дерево, образуя в нём мёртвые зоны: группы указателей, которые никогда не используются. Ситуацию можно улучшить, если вместо дубликатов ключей хранить в каждом узле значение кратности (1 или больше). Если же дубликаты должны быть представлены явно, их можно хранить в узлах как цепочки переполнения, по аналогии с хеш-таблицей.

Результаты работы программы



# 

Код программы

Main.cpp

#include <iostream>  
#include "Tree.h"  
#include <time.h>  
  
  
**int** main(){  
 srand(time(0));  
 Tree A, B, C, D, E, R, B\_and\_C;  
 R.create('R');  
 A.create('A');  
 B.create('B');  
 C.create('C');  
 D.create('D');  
 E.create('E');  
 A.Out();  
 B.Out();  
 C.Out();  
 D.Out();  
 E.Out();  
 R = B & C;  
 R.Out("B & C: ");  
 R = R | D;  
 R.Out("(B & C) | D: ");  
 R = R + E;  
 R.Out("(B & C) | D + E: ");  
 R = A / R;  
 R.Out("A / (B & C) | D + E: ");  
 **return** 0;  
}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Tree.h

#ifndef LAB\_4\_TREE\_H  
#define LAB\_4\_TREE\_H  
#endif //LAB\_4\_TREE\_H  
#include <iostream>  
#include <random>  
#include <string.h>  
  
**enum**{*N* = 10, *\_maxint* = 100};  
**using namespace** std;  
  
**struct** list{  
 list \*left;  
 list \*right;  
 **int** count;  
  
 list(): left(**nullptr**), right(**nullptr**) {}  
 ~list(){  
 **delete**(left);  
 **delete**(right);  
 }  
};  
  
**class** Tree{  
 list \*Root = **new** list();  
 **char** Name;  
 **int** Screen[4\**N*];  
 **int** k = 0;  
**public**:  
 **void** create(**char** \_name){  
 Name = \_name;  
 **int** a = rand() % *\_maxint*;  
 Root->count = a;  
 **for** (**int** i = 0; i < *N* - 1; i++){  
 a = rand() % *\_maxint*;  
 **while**(!input(a, Root)){  
 a = rand() % *\_maxint*;  
 }  
 }  
 }  
 **bool** input(**int** a, list \*\_root);  
 **void** Screen\_insert(list \*Node);  
 **void** Out(string \_name);  
 **void** Out();  
 Tree **operator** & (Tree);  
 Tree **operator** &= (Tree);  
 Tree **operator** | (Tree);  
 Tree **operator** |= (Tree);  
 Tree **operator** + (Tree);  
 Tree **operator** +=(Tree);  
 Tree **operator** / (Tree);  
 Tree **operator** /= (Tree);  
 **bool** search(**int** a, list \*\_root);  
 **void** insert(**int** a, list \*\_root);  
 **void** erase(list \*\_root);  
 **void** travers(list \*rootA, list \*rootB);  
 **void** travers\_or(list \*rootA, list \*rootB);  
 **void** travers\_del(list \*rootA, list \*rootB);  
 **void** copy(list \*B);  
};  
  
**bool** Tree::input(**int** a, list \*\_root) {  
 **if**(a > \_root->count){  
 **if**(\_root->right != **nullptr**){  
 **return**(input(a, \_root->right));  
 } **else**{  
 \_root->right = **new** list;  
 \_root->right->count = a;  
 **return true**;  
 }  
 } **else**{  
 **if** (a == \_root->count){  
 **return false**;  
 } **else**{  
 **if**(\_root->left != **nullptr**){  
 **return**(input(a, \_root->left));  
 } **else**{  
 \_root->left = **new** list;  
 \_root->left->count = a;  
 **return true**;  
 }  
 }  
 }  
}  
  
**void** Tree::Screen\_insert(list \*Node) {  
 **if**(Node != **nullptr**) {  
 **if** (Node->left == **nullptr**) {  
 Screen[k] = Node->count;  
 k++;  
 **if** (Node->right != **nullptr**) {  
 Screen\_insert(Node->right);  
 }  
 } **else** {  
 Screen\_insert(Node->left);  
 Screen[k] = Node->count;  
 k++;  
 **if** (Node->right != **nullptr**) {  
 Screen\_insert(Node->right);  
 }  
 }  
 }  
}  
  
**void** Tree::Out(string \_name) {  
 **if**(Root != **nullptr**) {  
 (\***this**).k = 0;  
 Screen\_insert(Root);  
 cout << \_name << ": ";  
 **for** (**int** i = 0; i < k; i++) {  
 cout << Screen[i] << ' ';  
 }  
 cout << endl;  
 } **else**{  
 cout << \_name << ": " << endl;  
 (\***this**).k = 0;  
 }  
}  
  
**void** Tree::Out() {  
 **if**(Root != **nullptr**) {  
 (\***this**).k = 0;  
 Screen\_insert(Root);  
 cout << Name << ": ";  
 **for** (**int** i = 0; i < k; i++) {  
 cout << Screen[i] << ' ';  
 }  
 cout << endl;  
 } **else**{  
 cout << Name << ": " << endl;  
 (\***this**).k = 0;  
 }  
}  
  
**bool** Tree::search(**int** a, list \*\_root) {  
 **if**(\_root == **nullptr**){  
 **return false**;  
 }  
 **if**(a == \_root->count){  
 **return true**;  
 } **else**{  
 **if**(a < \_root->count){  
 **if**(\_root->left != **nullptr**){  
 **return** search(a, \_root->left);  
 } **else return false**;  
 } **else**{  
 **if**(a > \_root->count){  
 **if**(\_root->right != **nullptr**){  
 **return** search(a, \_root->right);  
 } **else return false**;  
 } **else return false**;  
 }  
 }  
}  
  
**void** Tree::insert(**int** a, list \*\_root){  
 **if**(\_root == **nullptr**){  
 \_root = **new** list();  
 \_root->count = a;  
 } **else** {  
 **if** (a > \_root->count) {  
 **if** (\_root->right == **nullptr**) {  
 list \*buf = **new** list();  
 \_root->right = buf;  
 \_root->right->count = a;  
 } **else** {  
 insert(a, \_root->right);  
 }  
 } **else if** (a < \_root->count) {  
 **if** (\_root->left == **nullptr**) {  
 list \*buf = **new** list();  
 \_root->left = buf;  
 \_root->left->count = a;  
 } **else** {  
 insert(a, \_root->left);  
 }  
 }  
 }  
}  
  
**void** Tree::erase(list \*\_root) {  
 (\***this**).k--;  
 **if**(\_root->right == **nullptr** && \_root->left == **nullptr**){  
 list \*buf = **new** list();  
 buf = (\***this**).Root;  
 **if**(\_root != Root) {  
 **while** (buf->left != \_root && buf->right != \_root) {  
 **if** (buf->count > \_root->count) {  
 buf = buf->left;  
 } **else** buf = buf->right;  
 }  
 **if** (buf->left == \_root) {  
 **delete** (buf->left);  
 buf->left = **nullptr**;  
 } **else** {  
 **delete** (buf->right);  
 buf->right = **nullptr**;  
 }  
 } **else**{  
 **delete** ((\***this**).Root);  
 (\***this**).Root = **nullptr**;  
 }  
 } **else**{  
 **if**(\_root->right == **nullptr**){  
 list \*buf = **new** list();  
 buf = (\***this**).Root;  
 **if**(\_root != Root) {  
 **while** (buf->left != \_root && buf->right != \_root) {  
 **if** (buf->count > \_root->count) {  
 buf = buf->left;  
 } **else** buf = buf->right;  
 }  
 **if** (buf->left == \_root) {  
 buf->left = \_root->left;  
 } **else** {  
 buf->right = \_root->left;  
 }  
 } **else** {  
 Root = \_root->left;  
 }  
 } **else**{  
 **if**(\_root->left == **nullptr**){  
 list \*buf = **new** list();  
 buf = (\***this**).Root;  
 **if**(\_root != Root) {  
 **while** (buf->left != \_root && buf->right != \_root) {  
 **if** (buf->count > \_root->count) {  
 buf = buf->left;  
 } **else** buf = buf->right;  
 }  
 **if** (buf->left == \_root) {  
 buf->left = \_root->right;  
 } **else** {  
 buf->right = \_root->right;  
 }  
 } **else**{  
 Root = \_root->right;  
 }  
 } **else**{  
 list \*buf = **new** list();  
 buf = \_root->right;  
 **while**(buf->left != **nullptr**){  
 buf = buf->left;  
 }  
 buf->left = **new** list();  
 buf->left = \_root->left;  
 list \*buf1 = **new** list();  
 buf1 = (\***this**).Root;  
 **if**(\_root != Root) {  
 **while** (buf1->left != \_root && buf1->right != \_root) {  
 **if** (buf1->count > \_root->count) {  
 buf1 = buf1->left;  
 } **else** buf1 = buf1->right;  
 }  
 **if** (buf1->left == \_root) {  
 buf1->left = \_root->right;  
 } **else** {  
 buf1->right = \_root->right;  
 }  
 } **else**{  
 Root = \_root->right;  
 }  
 }  
 }  
 }  
}  
  
**void** Tree::travers(list \*rootA, list \*rootB) {  
 **if**(rootA->left == **nullptr**){  
 **if**(rootA->right != **nullptr**){  
 travers(rootA->right, rootB);  
 }  
 **if**(!search(rootA->count, rootB)){  
 erase(rootA);  
 }  
 } **else**{  
 travers(rootA->left, rootB);  
 **if**(rootA->right != **nullptr**){  
 travers(rootA->right, rootB);  
 }  
 **if**(!search(rootA->count, rootB)){  
 erase(rootA);  
 }  
 }  
}  
  
**void** Tree::travers\_or(list \*rootA, list \*rootB) {  
 **if**(rootB->left == **nullptr**){  
 **if**(rootB->right != **nullptr**){  
 travers\_or(rootA, rootB->right);  
 }  
 **if**(!search(rootB->count, rootA)){  
 insert(rootB->count, rootA);  
 }  
 } **else**{  
 travers\_or(rootA, rootB->left);  
 **if**(rootB->right != **nullptr**){  
 travers\_or(rootA, rootB->right);  
 }  
 **if**(!search(rootB->count, rootA)){  
 insert(rootB->count, rootA);  
 }  
 }  
}  
  
**void** Tree::travers\_del(list \*rootA, list \*rootB) {  
 **if**(rootA->left == **nullptr**){  
 **if**(rootA->right != **nullptr**){  
 travers\_del(rootA->right, rootB);  
 }  
 **if**(search(rootA->count, rootB)){  
 erase(rootA);  
 }  
 } **else**{  
 travers\_del(rootA->left, rootB);  
 **if**(rootA->right != **nullptr**){  
 travers\_del(rootA->right, rootB);  
 }  
 **if**(search(rootA->count, rootB)){  
 erase(rootA);  
 }  
 }  
}  
  
**void** Tree::copy(list \*B) {  
 **if**(B->left == **nullptr**){  
 **if**(B->right != **nullptr**){  
 copy(B->right);  
 }  
 insert(B->count,(\***this**).Root);  
 } **else**{  
 copy(B->left);  
 **if**(B->right != **nullptr**){  
 copy(B->right);  
 }  
 insert(B->count,(\***this**).Root);  
 }  
}  
  
Tree Tree::**operator**&(Tree B) {  
 Tree R(\***this**);  
 **return** R &= B;  
}  
  
Tree Tree::**operator**&=(Tree B) {  
 travers((\***this**).Root, B.Root);  
 **return** (\***this**);  
}  
  
Tree Tree::**operator**|(Tree B) {  
 Tree R(\***this**);  
 **return** R |= B;  
}  
  
Tree Tree::**operator**|=(Tree B) {  
 **if**((\***this**).Root == **nullptr**){  
 **if**(B.Root != **nullptr**){  
 (\***this**).Root = B.Root;  
 **return** (\***this**);  
 }  
 } **else** {  
 **if** (B.Root != **nullptr**) {  
 travers\_or((\***this**).Root, B.Root);  
 }  
 }  
 **return** (\***this**);  
}  
  
Tree Tree::**operator**+ (Tree B) {  
 Tree R(\***this**);  
 **return** R += B;  
}  
  
Tree Tree::**operator**+=(Tree B) {  
 Tree A;  
 A.copy((\***this**).Root);  
 A = A | B;  
 Tree C;  
 C.copy((\***this**).Root);  
 C = C & B;  
 **return** A / C;  
}  
  
Tree Tree::**operator**/(Tree B) {  
 Tree R(\***this**);  
 **return** R /= B;  
}  
  
Tree Tree::**operator**/=(Tree B) {  
 travers\_del((\***this**).Root, B.Root);  
 **return** (\***this**);  
}