**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №5**

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

Тема: Поиск и вставка в БДП

|  |  |  |
| --- | --- | --- |
| Студентка гр. 7382 |  | Головина Е.С. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2018

**Цель работы.**

Изучить структуру рандомизированной пирамиды поиска (treap). Научиться работать с ней, а именно добавлять элементы и выполнять поиск.

**Задание.**

Вариант 13:

БДП: Рандомизированная пирамида поиска (treap), действие 1+2

**Постановка задачи.**

1. По заданному файлу F (типа file of Elem), все элементы которого различны, построить БДП типа рандомизированная пирамида (treap);
2. Для построенного БДП проверить, входит ли в него элемент типа Elem, и если не входит, то добавить элемент в дерево поиска.

**Описание алгоритма вставки элемента.**

Заходим в метод randTreap::insert(elem key, node \*\* temp).

1. Проверяется пустой указатель на узел дерева или нет. Если да п.2., если нет п.3.

2. Тогда выделяется память для нового элемента, со случайным заданием приоритета. Далее п.4.

3. Проверяется, если новый элемент, который необходимо добавить, меньше текущего, происходит переход к левому поддереву, если больше к правому. Т.е. происходит рекурсивный запуск этого же метода, но для другого узла. И возвращаемся в п.1.

4. Полученное дерево проверяем на соответствие требуемому типу (treap) с помощью метода randTreap::iscorrect(node \*), если все хорошо работа метода завершается. Если нет — п.5.

5. Заходим в метод randTreap::check\_heap. Для начала проверяется не пустой ли указатель на узел, если пустой, работа метода завершается, если нет — п.6.

6. Далее проверяем есть ли у узла правый сын, если есть, то проверяем, что приоритет отца выше приоритета сына, иначе поворачиваем пирамиду влево с помощью метода randTreap::left\_turn. И после поворота возвращаемся обратно. п.7.

7. Далее проверяем есть ли у узла левый сын, если есть, то проверяем, что приоритет отца выше приоритета сына, иначе поворачиваем пирамиду вправо с помощью метода randTreap::right\_turn. И после поворота возвращаемся обратно. п.8.

8. Снова проверяем есть ли у узла правый сын, если есть, то метод randTreap::check\_heap(node\*\*) применяется для правого сына. И переходим в п.5.

9. Снова проверяем есть ли у узла левый сын, если есть, то метод randTreap::check\_heap(node\*\*) применяется для левого сына. И переходим в п.5.

10. В итоге получается корректная рандомизированная пирамида поиска.

**Методы класса randTreap и структуры данных:**

1. struct node{

elem key;

double priority;

node \* left;

node \* right;

};

key— ключ, хранящийся в узле рандомизированной пирамиды;

priority — приоритет, хранящийся в узле рандомизированной пирамиды;

left — указатель на левого сына узла;

right — указатель на правого сына узла.

2. struct randTreap{

struct node{

elem key;

double priority;

node \* left;

node \* right;

};

node\* top

};

struct node — структура с содержимым узла пирамиды;

top — указатель на вершину пирамиды.

3. **void randTreap::clean(node \* temp)**

temp — указатель на узел пирамиды.

Метод очищает выделенную под пирамиду память.

4. **void randTreap::insert(elem key,node \*\* temp)**

key— значение ключа, которое нужно добавить в пирамиду;

temp — указатель на указатель на узел пирамиды.

Метод выполняет вставку нового элемента со случайным выбором приоритета.

5. **void randTreap::check\_heap(node \*\* temp)**

temp — указатель на указатель на узел пирамиды.

Метод выполняет изменение пирамиды, чтоб свойство пирамиды по приоритетам было соблюдено.

6. **bool iscorrect(node \* temp)**

temp — указатель на узел пирамиды.

Метод проверяет корректность построенной пирамиды. Если все верно, возвращает true, иначе false.

7. **void print(node \* temp, int height)**

temp — указатель на узел пирамиды;

height — высота пирамиды

Метод печатает вид пирамиды в виде уступчатого списка.

8. **int height(node \* temp)**

temp — указатель на узел пирамиды.

Высчитывает высоту пирамиды.

9. **void left\_turn(node \*\* temp)**

temp — указатель на указатель на узел пирамиды.

Метод выполняет поворот пирамиды влево.

10. **void right\_turn(node \*\* temp)**

temp — указатель на указатель на узел пирамиды.

Метод выполняет поворот пирамиды вправо.

11. **bool search(node \* temp,elem key)**

temp — указатель на узел пирамиды;

key — искомый ключ.

Метод выполняет поиск элемента в пирамиде и если находит его — возвращает true, иначе false.

Тестирование.

Было сделано 9 тестов для демонстрации и проверки работы программы.

Файл с данными для 1 и 2, 3 и 4, 5 и 6, 7 и 8 один, чтобы продемонстрировать, что пирамида действительно рандомизированная.

|  |  |  |
| --- | --- | --- |
| Тест № | Данные | Результат |
| 1 | 40 50 20 80 10  77 | Element isn't here. So that we need to add it.  Treap without new element:  ----50, 0.6  ---10, 0.6  --40, 0.4  -20, 0.4  ---80, 0.6  Treap with new element:  ----50, 0.6  ---10, 0.6  --40, 0.4  -20, 0.4  ---80, 0.6  --77, 0 |
| 2 | 40 50 20 80 10  40 | Element 40 is found!  Builded treap:  -----10, 0.7  ----20, 0.6  ---40, 0.5  --80, 0.5  -50, 0.3 |
| 3 | 10 7 33 32 88 63 55 332 90  90 | Element 90 is found!  Builded treap:  ----55, 0.9  ---7, 0.6  --32, 0.6  -10, 0.4  -33, 0.4  ---88, 0.6  --63, 0  --90, 0.5  -332, 0.3 |
| 4 | 10 7 33 32 88 63 55 332 90  123 | Element isn't here. So that we need to add it.  Treap without new element:  -----332, 0.9  ----88, 0.7  ---33, 0.6  --10, 0.5  -7, 0.3  -32, 0.5  --55, 0.6  -63, 0.3  ---90, 0  Treap with new element:  -----332, 0.9  ----88, 0.7  ---33, 0.6  --10, 0.5  -7, 0.3  -32, 0.5  --55, 0.6  -63, 0.3  ---123, 0.1  --90, 0 |
| 5 | 10 7  15 | Element isn't here. So that we need to add it.  Treap without new element:  --7, 0.6  -10, 0.4  Treap with new element:  ---7, 0.6  --10, 0.4  -15, 0.4 |
| 6 | 10 7  10 | Element 10 is found!  Builded treap:  --10, 0.5  -7, 0.3 |
| 7 | 10 20 30 40 50 60 70 80 90 100  50 | Element 50 is found!  Builded treap:  -----70, 0.9  ----20, 0.6  ---10, 0.4  ---40, 0.6  --30, 0.4  --50, 0.6  -60, 0  ----90, 0.5  ---80, 0.3  ---100, 0.5 |
| 8 | 10 20 30 40 50 60 70 80 90 100  55 | Element isn't here. So that we need to add it.  Treap without new element:  -----80, 0.9  ----50, 0.7  ---30, 0.6  --10, 0.5  -20, 0.3  --40, 0.5  ---70, 0.6  --60, 0.3  ----100, 0.1  ---90, 0  Treap with new element:  -----80, 0.9  ----50, 0.7  ---30, 0.6  --10, 0.5  -20, 0.3  --40, 0.5  ---70, 0.6  --60, 0.3  -55, 0  ----100, 0.1  ---90, 0 |
| 9 | 10 | Tree is empty! Program terminated! |

Для более наглядной демонстрации работы программы был создан bash-скрипт, последовательно выводящий содержимое очередного теста и результат работы программы для этого теста.

Выберем 5 тест для детального обзора. Программа считывает название файла, хранящего последовательность элементов для добавление в пирамиду. Далее считывает элемент, который надо найти в пирамиде. Поэлементно считываются из файла ключи, которые надо добавить в пирамиду.

1. Принимаем элемент 10

2. Заходим в метод randTreap::insert. Полученный методом указатель на узел — пустой, т. к. изначально создается новая пирамида, где ничего не содержится, поэтому сразу выделяется память для нового узла с ключом 10. Случайным образом выбирается приоритет. В этот раз он равен 0.4. Он также записывается в узел, указатели на сыновей обнуляются. Полученный узел стал вершиной дерева. Выходим из метода randTreap::insert.

3. Считываем следующий элемент = 7. Заходим в метод randTreap::insert. Так как теперь вершина дерева не пустая сравниваем новый ключ с ключом в вершине, он меньше 10, поэтому переходим на левого сына и снова заходим в метод randTreap::insert. Указатель на левого сына пустой, поэтому выделяется память для нового узла с ключом 7. Случайным образом выбирается приоритет. В этот раз он равен 0.6. Проверяем пирамиду на корректность. Она некорректна, т. к. приоритет сына больше приоритета отца. Поэтому поворачиваем пирамиду вправо. Теперь вершина дерева (7, 0.6), а (10, 0.4) его правый сын.

4. Заходим в метод randTreap::search, чтобы найти искомый элемент 15. Ни в одном из узлов не содержится 15, поэтому нам необходимо добавить этот ключ.

5. Снова заходим в метод randTreap::insert. Пирамида непустая, поэтому происходит сравнение ключа с вершиной. 15 > 7, поэтому переходим на правого сына узла 7. Правый сын также существует и его ключ равен 10. 15>10, поэтому переходим на правого сына узла (10, 0.4). Выделяем память под новый узел и случайным образом выбираем приоритет. В этот раз он равен 0.4. Приоритет отца такой же, значит свойства пирамиды не нарушается и дальнейших преобразований не требуется. Пирамида построена.

**Вывод**

В ходе лабораторной работы были изучены принципы работы с БДП типа рандомизированная пирамида, научились добавлять элементы и выполнять поиск.

**ПРИЛОЖЕНИЕ А**

**ИСХОДНЫЙ КОД, СОДЕРЖАЩИЙ ГОЛОВНУЮ ФУНКЦИЮ**

#include <iostream>

#include <fstream>

#include <cstdlib>

#include <string>

#include <ctime>

#include "structure.h"

int main(){

char name[20];

std::ifstream file;

int key=0, new\_key=0;

int forrand=time(0);

//receiving file .txt

std::cout << "Please, type the name of the file in format <name>.txt" << std::endl;

std::cin.getline(name,20);

file.open(name);

if (!file.is\_open()){

std::cout << "\nFile couldn't be open.\nProgram terminated!" << std::endl;

return 0;

}

//receiving key for the search

std::cout << "Enter key(int) that you want to find: ";

std::cin >> new\_key;

std::cout << std::endl;

//making class object for future treap

randTreap<int> my\_treap;

//receiving elements from file and adding them to the treap

do{

srand(forrand);

file >> key;

if (file.eof()) break;

my\_treap.insert(key,&(my\_treap.top));

forrand++;

}while (1);

if (!my\_treap.height(my\_treap.top)){

std::cout << "Tree is empty! Program terminated!\n";

exit(1);

}

//printing content of the file

file.clear();

file.seekg(0, file.beg);

std::cout << "INPUT = ";

do{

file>>key;

if (file.eof()) break;

std::cout << key << " ";

}while(1);

std::cout << std::endl << std::endl;

file.close();

//searching for new\_key if it is found print the treap,

//if not add new\_key to the treap and print the result.

if (my\_treap.search(my\_treap.top,new\_key)){

std::cout << "Element " << new\_key << " is found!\n\n";

std::cout << "Builded treap:\n";

my\_treap.print(my\_treap.top,my\_treap.height(my\_treap.top));

}

else{

std::cout << "Element isn't here. So that we need to add it.\n\n";

std::cout << "Treap without new element:\n";

my\_treap.print(my\_treap.top,my\_treap.height(my\_treap.top));

my\_treap.insert(new\_key,&(my\_treap.top));

std::cout << "Treap with new element:\n";

my\_treap.print(my\_treap.top,my\_treap.height(my\_treap.top));

}

return 0;

}

**ПРИЛОЖЕНИЕ Б**

**ИСХОДНЫЙ КОД ЗАГОЛОВОЧНИКА**

//template for randomized treap, type of the key selectable

template <class elem>

struct randTreap{

//struct for the node of the treap, which contains pair(key,priority)

//and pointers to the sons

struct node{

elem key;

double priority;

node \* left;

node \* right;

};

//pointer to the top node of the randomized treap

node \* top;

//constructor

randTreap(){

top = NULL;

}

~randTreap(){

if (top)

randTreap::clean(top);

}

//release of the allocated memory, method for destructor

void clean(node \* temp){

if (temp->left)

randTreap::clean(temp->left);

if (temp->right)

randTreap::clean(temp->right);

delete temp;

}

//adding new node, with correction

void insert(elem key,node \*\* temp){

if (!(\*temp)){

std::cout << "\nAdding " << key << "\n";

\*temp = new node;

(\*temp)->key=key;

(\*temp)->priority=(double)(rand() % 10)/10;

(\*temp)->left=NULL;

(\*temp)->right=NULL;

std::cout << "Result:\n";

randTreap::print(top,randTreap::height(top));

while(!(randTreap::iscorrect(top))){

randTreap::check\_heap(&top);

std::cout << "After correction:\n";

randTreap::print(top,randTreap::height(top));

}

}

else if (key < (\*temp)->key){

randTreap::insert(key,&((\*temp)->left));

}

else if (key > (\*temp)->key){

randTreap::insert(key,&((\*temp)->right));

}

}

//changing treap to correct one

void check\_heap(node \*\* temp, double max=1.0){

if (\*temp){

if ((\*temp)->right){

if ((\*temp)->priority < (\*temp)->right->priority)

randTreap::left\_turn(temp);

}

if((\*temp)->left){

if ((\*temp)->priority < (\*temp)->left->priority)

randTreap::right\_turn(temp);

}

if ((\*temp)->right)

randTreap::check\_heap(&((\*temp)->right));

if((\*temp)->left)

randTreap::check\_heap(&((\*temp)->left));

}

}

//check if treap is correct

bool iscorrect(node \* temp){

if (!temp) return true;

if (temp->left){

if (temp->priority < temp->left->priority) return false;

}

if (temp->right){

if (temp->priority < temp->right->priority) return false;

}

return randTreap::iscorrect(temp->left) && randTreap::iscorrect(temp->right);

}

//printing treap

void print(node \* temp, int height){

std::string dashes(height,'-');

if (temp){

// std::cout.width(height+3);

std::cout << dashes << temp->key << ", " << temp->priority << std::endl;

}

if (temp->left){

randTreap::print(temp->left,height-1);

}

if (temp->right){

randTreap::print(temp->right,height-1);

}

}

//counting the height of the tree

int height(node \* temp){

if (!temp) return 0;

else return 1+std::max(randTreap::height(temp->left),randTreap::height(temp->right));

}

//making left turn of the treap

void left\_turn(node \*\* temp){

node \* forswap=NULL;

if (\*temp){

forswap = (\*temp)->right;

(\*temp)->right = forswap->left;

forswap->left = (\*temp);

(\*temp) = forswap;

}

}

//making right turn of the treap

void right\_turn(node \*\* temp){

node \* forswap=NULL;

if (\*temp){

forswap = (\*temp)->left;

(\*temp)->left = forswap->right;

forswap->right = (\*temp);

(\*temp) = forswap;

}

}

//searching for the key

bool search(node \* temp,elem key){

if (!temp) return false;

if (temp->key == key) return true;

else return randTreap::search(temp->left,key) || randTreap::search(temp->right,key);

}

};