Федеральное государственное бюджетное образовательное учреждение высшего образования «Сибирский государственный университет телекоммуникаций и информатики»

кафедра ПМиК

КУРСОВАЯ РАБОТА

по дисциплине «Объектно-ориентированное программирование»

Тема: «Построение ДОП (дерево оптимального поиска)»

Выполнил: студент группы ИП-111

Кузьменок Д.В.

Проверил: ассистент кафедры ПМиК

Дементьева К.И.

Новосибирск

2022

Содержание

1. Постановка задачи 3

2. Технологии ООП 4

3. Структура классов 11

4. Программная реализация 13

5. Результат работы 18

6. Заключение 20

7. Приложение. Листинг 21

8. Список литературы и других источников 28

**Постановка задачи**

1. Используя знания из предмета “Структуры и алгоритмы обработки данных” реализовать алгоритм построения дерева оптимального поиска, где новой и, пожалуй, основной чертой является вес вершины.

2. В программе должны использоваться методы объектно-ориентированного программирования (инкапсуляция, наследование, полиморфизм, списки инициализации, абстрактный класс и виртуальные методы, статические переменные, шаблоны).

3. Построить и вывести три матрицы: весов (AW), взвешенных весов поддеревьев (AP), корней поддеревьев (AR) – благодаря которым и строится оптимальное дерево поиска. По завершению программы на экран должны выводиться характеристики дерева – размер, контрольная сумма, высота и средневзвешенная высота – для сравнения с другими деревьями поиска.

**Технологии ООП**

*Инкапсуляция* – это поля, которые меняются или используются только внутри класса, т.е. доступа извне к ним нет (из экземпляра класса). Доступ к закрытым полям можно получить только с помощью геттеров и сеттеров.

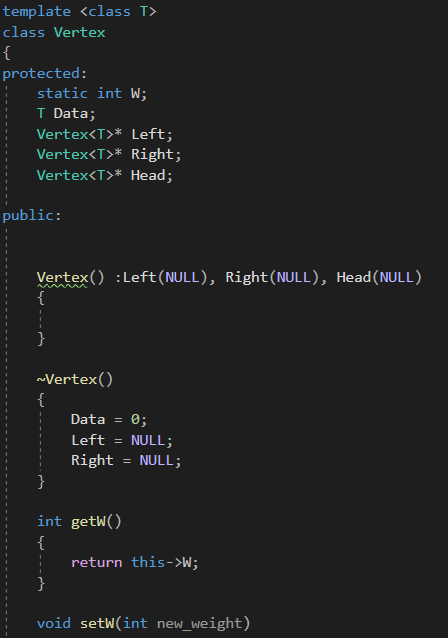


Рисунок 1 – Инкапсуляция.

*Наследование* – механизм базирования объекта или класса на другом объекте (наследование на основе прототипа) или классе (наследование на основе класса) с сохранением аналогичной реализации.

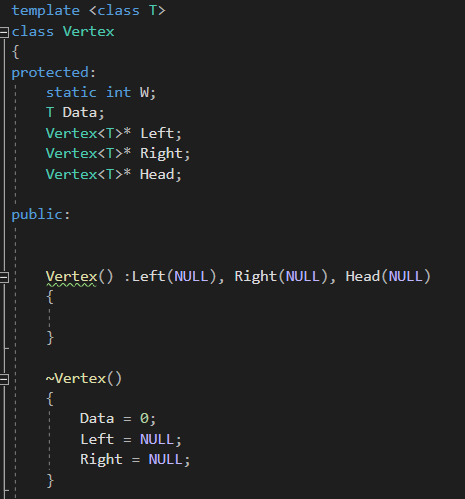


Рисунок 2 – Родительский класс.



Рисунок 3 – Дочерний класс класса Vertex.

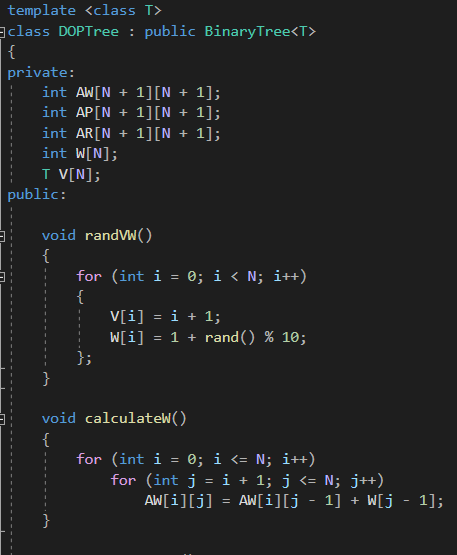


Рисунок 4 – Дочерний класс класса BinaryTree

*Полиморфизм* – это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта. В моём случае он реализован в виде виртуальных методов.



Рисунок 5 – Чистый виртуальный метод, находящееся в классе BinaryTree, где с помощью наследования в классе DOPTree она будет переопределяться.

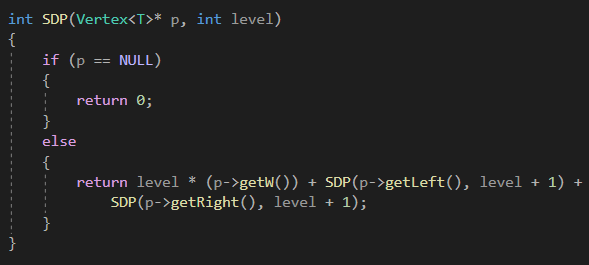


Рисунок 6 – Переопределение метода в классе-наследнике

*Списки инициализации* – метод, при котором инициализируются переменные члены класса вместо присваивания им значений после их создания.

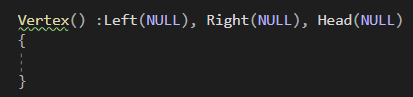


Рисунок 7 – Списки инициализации для указателей вершины в классе Vertex.

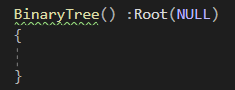


Рисунок 8 – Списки инициализации для корня всего дерева в классе BinaryTree.

*Виртуальный метод* - в объектно-ориентированном программировании метод класса, который может быть переопределён в классах-наследниках так, что конкретная реализация метода для вызова будет определяться во время исполнения.

*Абстрактный класс* - это класс, у которого не реализован один или больше методов.



Рисунок 9 – Чистый виртуальный метод в классе BinaryTree, который будет переопределён в классе-наследнике DOPTree. Также класс BinaryTree является абстрактным, так как содержит чистый виртуальный метод.

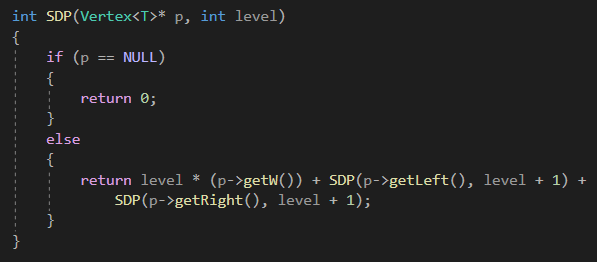


Рисунок 10 – Переопределение виртуального метода в дочернем классе DOPTree.

*Статическая переменная* – это переменная, которая не исчезает после завершения работы функции. Значение статической переменной можно использовать при следующем вызове функции - она по-прежнему будет иметь то же значение, которое получила при последнем вызове функции.

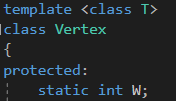
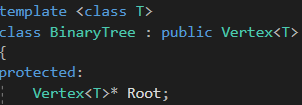
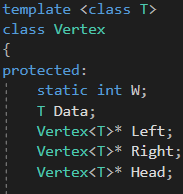
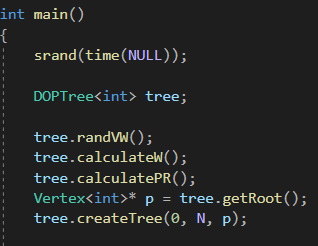


Рисунок 11 – Использование статической переменной при создании вершин дерева.

*Шаблоны* - это фрагменты обобщённого кода, в котором некоторые типы или константы вынесены в параметры. Шаблонными могут быть функции, структуры (классы) и даже переменные. Компилятор превращает использование шаблона в конкретный код, подставляя в него нужные параметры на этапе компиляции. Шаблоны позволяют писать общий код, пригодный для использования с разными типами данных.





Рисунки 12 – Использование шаблонов для того, чтобы в вершинах дерева могли записываться различные типы данных.

**Структура классов**

Классы в С++ — это абстракция, которая описывает методы, свойства, еще не существующих объектов.

Объекты – конкретное представление абстракции, имеющие свои свойства и методы.

В ООП существует три основных принципа построения классов:

1. Инкапсуляция;
2. Наследование;
3. Полиморфизм.

Спецификаторы доступа:

1. Public – это члены структуры или класса, к которым можно получить доступ извне структуры или класса.
2. Private – это члены класса, к которым могут получить доступ только другие члены класса.

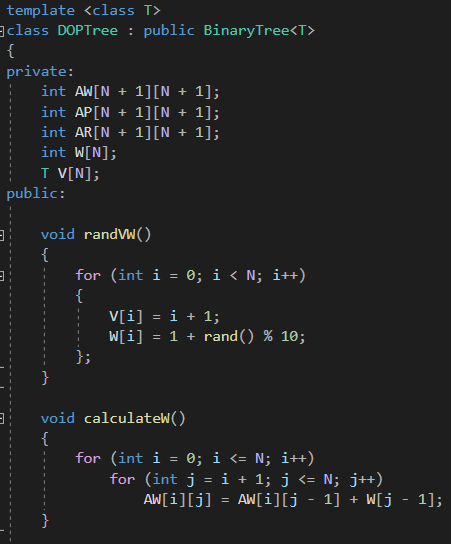


Рисунок 13 – Класс DOPTree, у которого матрицы из private участвуют в методах в public.

1. Protected – позволяет получить доступ к члену класса, к которому принадлежит член, друзья и производные класса. Однако Protected члены недоступны вне класса.

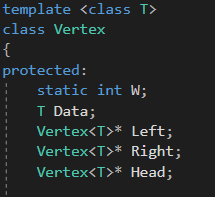


Рисунок 14 – Спецификатор protected, используемый в дочерних классах для инициализации вершины дерева и для их привязки между друг другом.

**Программная реализация**

Все основные действия программы регулируются в функции int main(). В ней я создаю объект дерево, строю матрицы для дерева оптимального поиска, беру указатель на корень дерева из самого основного класса Vertex и в конечном итоге строю само дерево. Также здесь присутствует вывод матриц, на случай ручной проверки корректности работы программы, и вывод основных характеристик дерева.

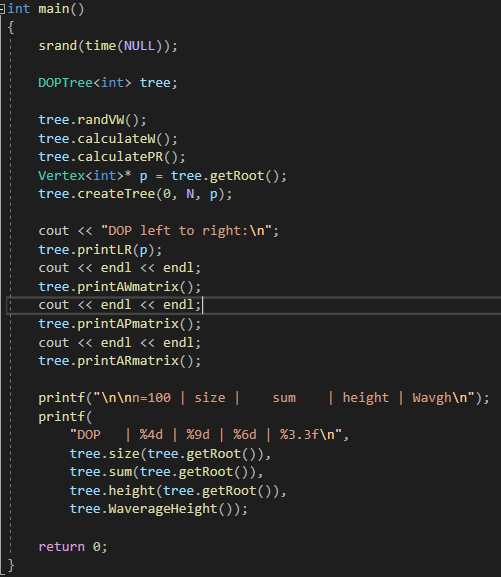


Рисунок 15 – Функция main().

class Vertex – основной родительский класс, в нем происходит инициализация вершин и привязка между ними с помощью указателя Head, который содержится в списках инициализации.

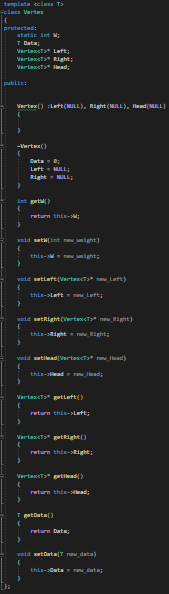


Рисунок 16 – Класс Vertex (основной родительский).

class BinaryTree – дочерний класс класса Vertex. Наследуется указатель на корень дерева. В самом же классе происходит подсчёт всех необходимых параметров дерева оптимального поиска.

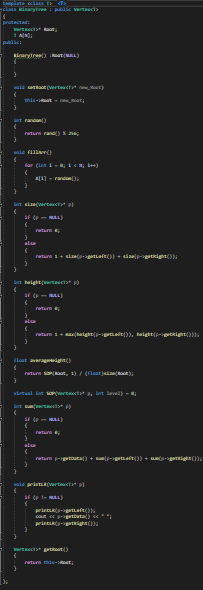


Рисунок 17 – Класс BinaryTree.

class DOPTree – дочерний класс класса BinaryTree. В нём происходит построение и вывод три матрицы: весов (AW), взвешенных весов поддеревьев (AP), корней поддеревьев (AR) – благодаря которым и строится оптимальное дерево поиска.



Рисунок 18 – Класс DOPTree.

**Результаты работы**

Результатом моей курсовой работы стало корректное построение дерева оптимального поиска (было все проверено вручную) и вывод правильных характеристик, которые ещё раз подтверждают, что дерево оптимального поиска стоит на лидирующем месте среди остальных деревьев.

https://sun9-65.userapi.com/impg/BOimLhUTimR5Rhmur47IsGQHD4fkAPz1ftxkZg/3AJIyACDW70.jpg?size=181x43&quality=96&sign=11384b8f4a0e318eefe50e1925adafc0&type=album

Рисунок 19 - Вывод обхода дерева слева направо.

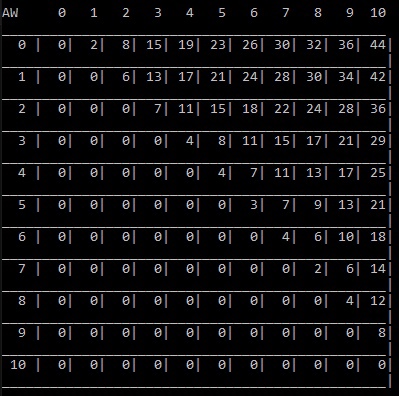


Рисунок 20 – Вывод матрицы весов.

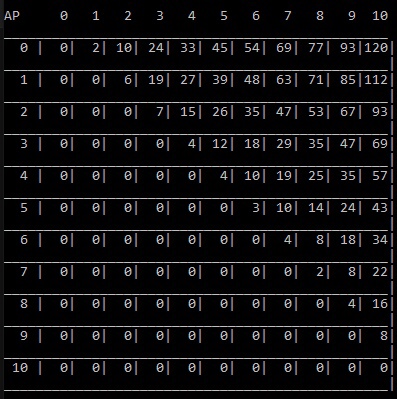


Рисунок 21 – Вывод матрицы взвешенных весов поддеревьев.

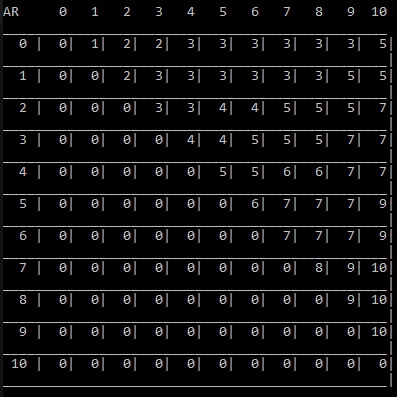


Рисунок 22 – Вывод матрицы корней поддеревьев (в ячейках матрицы содержатся индексы элементов исходного массива).

https://sun9-35.userapi.com/impg/-JBhCQ7w-lhRGz6xXq3tY6KgwjNM2NBK40buYw/9vJARXEGq8Y.jpg?size=337x45&quality=96&sign=9f3905ab5b9d92ba13416c288daebfa5&type=album

Рисунок 23 – Вывод основных характеристик дерева.

**Заключение**

В конечном итоге, у меня получилось реализовать дерево оптимального поиска, используя по максимуму технологии ООП. Были реализованы все необходимые функции подсчёта, строительства и вывода, чтобы в результате получить корректно работающую программу.

**Приложение. Листинг**

#include <algorithm>

#include <iostream>

#include <time.h>

#define N 10

using namespace std;

template <class T>

class Vertex

{

protected:

static int W;

T Data;

Vertex<T>\* Left;

Vertex<T>\* Right;

Vertex<T>\* Head;

public:

Vertex() :Left(NULL), Right(NULL), Head(NULL)

{

}

~Vertex()

{

Data = 0;

Left = NULL;

Right = NULL;

}

int getW()

{

return this->W;

}

void setW(int new\_weight)

{

this->W = new\_weight;

}

void setLeft(Vertex<T>\* new\_Left)

{

this->Left = new\_Left;

}

void setRight(Vertex<T>\* new\_Right)

{

this->Right = new\_Right;

}

void setHead(Vertex<T>\* new\_Head)

{

this->Head = new\_Head;

}

Vertex<T>\* getLeft()

{

return this->Left;

}

Vertex<T>\* getRight()

{

return this->Right;

}

Vertex<T>\* getHead()

{

return this->Head;

}

T getData()

{

return Data;

}

void setData(T new\_data)

{

this->Data = new\_data;

}

};

template <class T>

int Vertex<T>::W = 0;

template <class T>

class BinaryTree : public Vertex<T>

{

protected:

Vertex<T>\* Root;

T A[N];

public:

BinaryTree() :Root(NULL)

{

}

void setRoot(Vertex<T>\* new\_Root)

{

this->Root = new\_Root;

}

int random()

{

return rand() % 256;

}

void fillArr()

{

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

{

A[i] = random();

}

}

int size(Vertex<T>\* p)

{

if (p == NULL)

{

return 0;

}

else

{

return 1 + size(p->getLeft()) + size(p->getRight());

}

}

int height(Vertex<T>\* p)

{

if (p == NULL)

{

return 0;

}

else

{

return 1 + max(height(p->getLeft()), height(p->getRight()));

}

}

float averageHeight()

{

return SDP(Root, 1) / (float)size(Root);

}

virtual int SDP(Vertex<T>\* p, int level) = 0;

int sum(Vertex<T>\* p)

{

if (p == NULL)

{

return 0;

}

else

{

return p->getData() + sum(p->getLeft()) + sum(p->getRight());

}

}

void printLR(Vertex<T>\* p)

{

if (p != NULL)

{

printLR(p->getLeft());

cout << p->getData() << " ";

printLR(p->getRight());

}

}

Vertex<T>\* getRoot()

{

return this->Root;

}

};

template <class T>

class DOPTree : public BinaryTree<T>

{

private:

int AW[N + 1][N + 1];

int AP[N + 1][N + 1];

int AR[N + 1][N + 1];

int W[N];

T V[N];

public:

void randVW()

{

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

{

V[i] = i + 1;

W[i] = 1 + rand() % 10;

};

}

void calculateW()

{

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

for (int j = i + 1; j <= N; j++)

AW[i][j] = AW[i][j - 1] + W[j - 1];

}

void calculatePR()

{

int i, j, m, min, k, h;

for (i = 0; i < N; i++)

{

j = i + 1;

AP[i][j] = AW[i][j];

AR[i][j] = j;

}

for (h = 2; h <= N; h++)

{

for (i = 0; i <= N - h; i++)

{

j = i + h;

m = AR[i][j - 1];

min = AP[i][m - 1] + AP[m][j];

for (k = m + 1; k <= AR[i + 1][j]; k++)

{

int x = AP[i][k - 1] + AP[k][j];

if (x < min)

{

m = k;

min = x;

}

}

AP[i][j] = min + AW[i][j];

AR[i][j] = m;

}

}

}

void add(T D, int W, Vertex<T>\*& p)

{

if (p == NULL)

{

p = new Vertex<T>;

if (this->getRoot() == NULL) {

this->setRoot(p);

this->setHead(NULL);

}

p->setData(D);

p->setW(W);

}

else if (D < p->getData())

{

Vertex<T>\* pp = p->getLeft();

add(D, W, pp);

p->setLeft(pp);

}

else if (D > p->getData())

{

Vertex<T>\* pq = p->getRight();

add(D, W, pq);

p->setRight(pq);

}

}

void createTree(int L, int R, Vertex<T>\*& root)

{

if (L < R)

{

int k = AR[L][R];

add(V[k - 1], W[k - 1], root);

createTree(L, k - 1, root);

createTree(k, R, root);

}

}

int SDP(Vertex<T>\* p, int level)

{

if (p == NULL)

{

return 0;

}

else

{

return level \* (p->getW()) + SDP(p->getLeft(), level + 1) +

SDP(p->getRight(), level + 1);

}

}

void printAWmatrix()

{

cout << "AW ";

for (int i = 0; i < N + 1; i++) printf("%3d ", i);

cout << endl;

for (int i = 0; i < N + 1; i++) cout << "\_\_\_\_";

cout << "\_\_\_\_" << endl;

for (int i = 0; i < N + 1; i++)

{

printf("%3d |", i);

for (int j = 0; j < N + 1; j++) printf("%3d|", AW[i][j]);

cout << endl;

for (int j = 0; j < N + 1; j++) cout << "\_\_\_\_";

cout << "\_\_\_\_|" << endl;

}

cout << endl;

}

void printAPmatrix()

{

cout << "AP ";

for (int i = 0; i < N + 1; i++) printf("%3d ", i);

cout << endl;

for (int i = 0; i < N + 1; i++) cout << "\_\_\_\_";

cout << "\_\_\_\_" << endl;

for (int i = 0; i < N + 1; i++)

{

printf("%3d |", i);

for (int j = 0; j < N + 1; j++) printf("%3d|", AP[i][j]);

cout << endl;

for (int j = 0; j < N + 1; j++) cout << "\_\_\_\_";

cout << "\_\_\_\_|" << endl;

}

cout << endl;

}

void printARmatrix()

{

cout << "AR ";

for (int i = 0; i < N + 1; i++) printf("%3d ", i);

cout << endl;

for (int i = 0; i < N + 1; i++) cout << "\_\_\_\_";

cout << "\_\_\_\_" << endl;

for (int i = 0; i < N + 1; i++)

{

printf("%3d |", i);

for (int j = 0; j < N + 1; j++) printf("%3d|", AR[i][j]);

cout << endl;

for (int j = 0; j < N + 1; j++) cout << "\_\_\_\_";

cout << "\_\_\_\_|" << endl;

}

cout << endl;

}

int sumW(Vertex<T>\* p)

{

if (p == NULL)

{

return 0;

}

else

{

return p->getW() + sumW(p->getLeft()) + sumW(p->getRight());

}

}

float WaverageHeight()

{

return (float)SDP(this->getRoot(), 1) / sumW(this->getRoot());

}

};

int main()

{

srand(time(NULL));

DOPTree<int> tree;

tree.randVW();

tree.calculateW();

tree.calculatePR();

Vertex<int>\* p = tree.getRoot();

tree.createTree(0, N, p);

cout << "DOP left to right:\n";

tree.printLR(p);

cout << endl << endl;

tree.printAWmatrix();

cout << endl << endl;

tree.printAPmatrix();

cout << endl << endl;

tree.printARmatrix();

printf("\n\nn=100 | size | sum | height | Wavgh\n");

printf(

"DOP | %4d | %9d | %6d | %3.3f\n",

tree.size(tree.getRoot()),

tree.sum(tree.getRoot()),

tree.height(tree.getRoot()),

tree.WaverageHeight());

return 0;

}

**Список литературы и других источников**

1. Лафоре, Р. Объектно-ориентированное программирование в C++. – СПб : Питер, 2003. – С. 124-129.

2. Страуструп, Б. Язык программирования С++. – М. : Бином, 2010. – С. 34-41.

3. Мейер, Б. Почувствуй класс: учимся программировать хорошо с объектами и контрактами. – М. : Интернет-университет информационных технологий, 2011. – С. 111-114.

4. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений. – СПб : Бином, 1998. – С. 36.

5. Вайсфельд, М. Объектно-ориентированное мышление. – СПб : Питер Пресс, 2014. – С. 84-86.