**Лабораторна робота №3**

**Задачі з графами (продовження)**

**Мета роботи**: закріпити на практиці знання з теорії графів, отримання навичок конструювання ПЗ, розв’язання задач з використанням графів

**Завдання**

1) Розробити програми обходу вершин дерева:

* у прямому порядку;
* внутрішньому порядку;
* зворотному порядку.

2) Розробити програми обходу вершин графу (не обов’язково дерева) з використанням структури даних для подання графів з Л.Р. №1 й алгоритму побудови остовного дерева графу з Л.Р. №2.

*Розв’язання*

1. **Текст програми**

***traversalTree.ts (обхід дерева):***

import Vertex from "../vertex";

import Graph from "../graph";

let tree: Graph<null, Object>;

let type: 'pre' | 'in' | 'post';

/\*\*

\* Возвращает массив вершин дерева образованный при его обходе

\* (дерево может быть любым)

\*

\* @param \_tree Обрабатываемое дерево

\* @param root Корень дерева

\* @param \_type Тип обхода: 'pre' - прямой, 'in' - внутренний, 'post' - обратный

\*/

export default function traversalTree<V=Object>(

\_tree: Graph<null, Object>,

root: Vertex<Object>,

\_type: 'pre' | 'in' | 'post'

): Vertex<V>[] {

tree = \_tree;

type = \_type;

return <Vertex<V>[]>addNext(root);

}

function addNext(root: Vertex<Object>, prev?: Vertex<Object>): Vertex<Object>[] {

const nexts = tree.getVVertices(root); //Инцидентные вершины

if (!nexts.length) return [root]; //Текущая вершина является конечной

const nextsRes = nexts.filter(v => v !== prev)

.map(next => addNext(next, root));

if (!nextsRes.length) return [root]; //Текущая вершина является конечной

if (type === 'pre') {

return [].concat(root, ...nextsRes);

} else if (type === 'in') {

const res = [].concat(nextsRes[0], root);

const last = nextsRes.slice(1);

return (last.length) ? res.concat(...last) : res;

} else if (type === 'post') {

return [].concat(...nextsRes, root);

}

}

В даній роботі використовується дерево отримане в результаті розбору математичного виразу (класс Expression):

const exp = new Expression("2.2+4\*7+6.3\*4/7/4%(5&6|7^8)");

const preorder = (() => traversal(exp, 'pre'));

const inorder = (() => traversal(exp, 'in'));

const postorder = (() => traversal(exp, 'post'));

function traversal(exp: Expression, type: 'pre' | 'in' | 'post'): string {

return traversalTree(exp.tree, exp.root, type).map(v => v.targ).join('');

}

Для обходу вершин графу використовується функція:

function traversalGraph(graph: Graph<any, any>): Vertex<any>[] {

const mst = getMST(graph);

return traversalTree(mst, graph.getVertex(0), 'pre');

}

Розбір математичного виразу:

***expression.ts***

import Graph from "./graph/graph";

import Vertex from "./graph/vertex";

import Edge from "./graph/edge";

/\*\*

\* Вершина в дереве выражения.

\* Является обобщением операндов и операторов

\*

\* @class

\*/

export class NodeExp {

/\*\*

\* @param type тип вершини - оператор или операнд

\* @param val значение

\*/

constructor(

public readonly type: 'operand' | 'operator',

public readonly val: number | Operator,

) {

}

}

class Operator {

/\*\*

\* @param name имя оператора (что используется в выражении)

\* @param precedence приоритет

\* @param call функция вызова оператора

\*/

constructor(

public readonly name: string,

public readonly precedence: number,

public readonly call: (a: number, b: number) => number

) {

}

}

/\*\*

\* Математическое уравнение

\*

```javascript

const exp = new Expression("2+3\*6");

ext.strExt //"2+3\*6"

exp.res //20

exp.tree //дерево выражения

exp.root //вершина дерева выражения

```

\*/

export default class Expression {

private \_tree = new Graph<null, NodeExp>();

private \_root: Vertex<NodeExp> = null;

private \_res: number = null;

/\*\* @param strExp Математическое выражение в виде строки \*/

constructor(private \_strExp: string) {

this.parse();

this.calc();

}

/\*\* @returns результат выражения \*/

public get strExp(): string {

return this.\_strExp;

}

(...)

private parse() {

this.\_root = this.\_parseNext(this.\_strExp);

this.\_tree.addVertex(this.\_root);

}

private calc() {

this.\_res = this.\_calcNext(this.\_root);

}

private \_parseNext(strExp: string): Vertex<NodeExp> {

let pos = 0;

let operands: string[] = [];

let operators: Operator[] = [];

operands.push(this.getOperand(strExp, pos, (\_pos => pos = \_pos)));

// It's only operand

if (pos === strExp.length) {

let value = this.toNumber(operands[0]);

let newRoot = new Vertex(new NodeExp('operand', value));

this.\_tree.addVertex(newRoot);

return newRoot;

}

while (pos < strExp.length) {

operators.push(this.getOperator(strExp, pos, (\_pos => pos = \_pos)));

operands.push(this.getOperand(strExp, pos, (\_pos => pos = \_pos)));

}

let operandPos = 0;

let operand1 = this.\_parseNext(operands[operandPos++]);

for (let i = 0; i < operators.length; i++) {

let operator = operators[i];

let operand2Str = (function () {

let operand = operands[operandPos++];

let j;

for (j = i + 1; j < operators.length; j++) {

if (operators[j].precedence <= operators[i].precedence) break;

operand += `${operators[j]}(${operands[operandPos++]})`;

}

i = j - 1;

return operand;

}());

let operand2 = this.\_parseNext(operand2Str);

const newRoot = new Vertex(new NodeExp('operator', operator));

this.\_tree.addEdge(new Edge(newRoot, operand1, null, 'uni'));

this.\_tree.addEdge(new Edge(newRoot, operand2, null, 'uni'));

operand1 = newRoot;

}

return operand1;

}

/\*\*

\* Возвращает оператор находящийся на перданной позиции pos

\*

\* @param strExp строка выражения

\* @param pos позиция на которой ожидается оператор

\* @param callback коллбек-функция, которая получает новое значение pos

\* @returns оператор находящийся на перданной позиции pos

\*

\* @private

\*/

private getOperator(strExp: string, pos: number,

callback: (pos: number) => void): Operator {

(...)

}

/\*\*

\* Возвращает операнд находящийся на перданной позиции pos

\*

\* Если оператор предствален просто числом, возвращает его (в виде строки)

\*

\* Если оператор заключен в скобки, то возвращается все что находится

\* между ними

\*

``` javascript

this.getOperand('25+36', 0, (\_pos => {})) // '25'

this.getOperand('(25+36)+5', (\_pos => {})) // '25+36'

```

\* В обоих случаях новый pos указывает на последний "+"

\*

\* @param strExp строка выражения

\* @param pos позиция на которой ожидается операнд

\* @param callback коллбэк-функция, которая получает новое значение pos

\* @returns операнд находящийся на перданной позиции pos

\*

\* @private

\*/

private getOperand(strExp: string, pos: number,

callback: (pos: number) => void): string {

(...)

}

private \_calcNext(vertex: Vertex<NodeExp>): number {

const node = vertex.targ;

if (node.type === 'operator') {

const nexts = this.\_tree.getVVertices(vertex);

return (<Operator>node.val).call(

this.\_calcNext(nexts[0]),

this.\_calcNext(nexts[1])

);

} else if (node.type === 'operand') {

return <number>node.val;

}

}

private static operators: { [name: string]: Operator } = {

'|': new Operator('|', 7, (a, b) => a | b),

'^': new Operator('^', 8, (a, b) => a ^ b),

'&': new Operator('&', 9, (a, b) => a & b),

'+': new Operator('+', 13, (a, b) => a + b),

'-': new Operator('-', 13, (a, b) => a - b),

'\*': new Operator('\*', 14, (a, b) => a \* b),

'/': new Operator('/', 14, (a, b) => a / b),

'%': new Operator('%', 14, (a, b) => a % b),

}

}

Позиціонування вершин бінарного дерева

***positionedTree.ts***

let start: Vector = null;

let step: Vector = null;

let tree: Graph<any, any> = null;

let positioned: Graph<WSEdge, WSVertex>;

export function positionedTree(

\_tree: Graph<any, any>,

root: Vertex<any>,

\_start = new Vector(0, -250),

\_step = new Vector(30, 80)

): Graph<WSEdge, WSVertex> {

start = \_start;

step = \_step;

positioned = new Graph<WSEdge, WSVertex>();

tree = \_tree;

positioned.addVertex(getNextVertex(root, start));

return positioned;

}

function getNextVertex(vertex: Vertex<any>, coords: Vector): Vertex<WSVertex> {

const root = new Vertex(new WSVertex(vertex.targ.toString(), coords));

const nexts = tree.getVVertices(vertex);

if (!nexts.length) return root;

const v1 = getNextVertex(nexts[0], getCoords(nexts[0], coords, -1));

const v2 = getNextVertex(nexts[1], getCoords(nexts[1], coords, 1));

positioned.addEdge(new Edge(root, v1, new WSEdge()));

positioned.addEdge(new Edge(root, v2, new WSEdge()));

return root;

}

function getCoords(root: Vertex<any>, base: Vector, dir: 1 | -1): Vector {

const height = getHeight(root, tree) - 1;

return new Vector(

base.x + (1 << height) \* step.x \* dir,

base.y + step.y

);

}

function getHeight(root: Vertex<any>, tree: Graph<any, any>): number {

const nexts = tree.getVVertices(root);

if (!nexts.length) return 1;

return 1 + Math.max(

getHeight(nexts[0], tree),

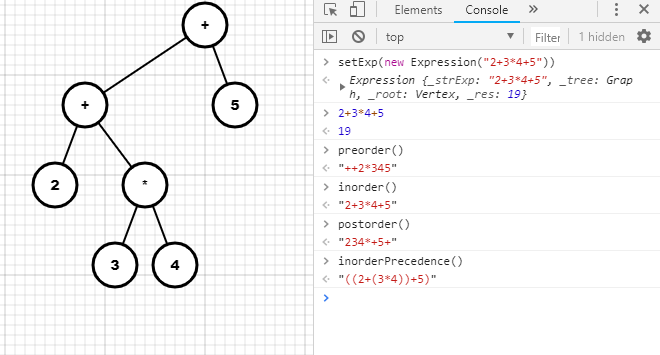
getHeight(nexts[1], tree)

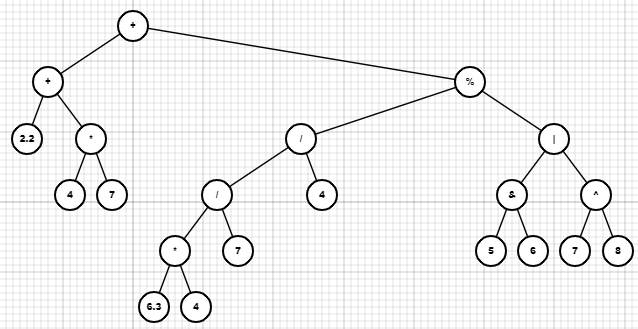
);

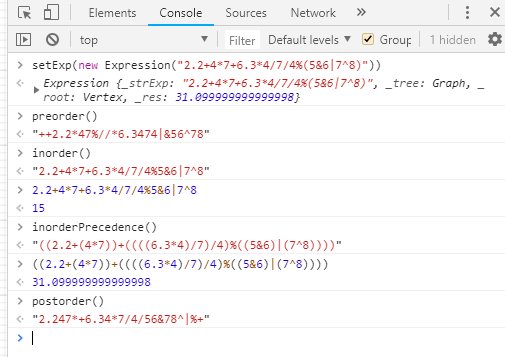
}

1. Результат виконання програми:

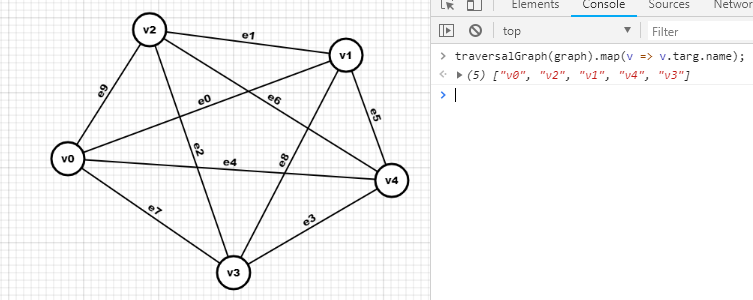
* Обхід дерева:







Обхід вершин графа:



**Висновок:** на цій лабораторній роботі я закріпив на практиці знання з теорії графів, отримав навички конструювання ПЗ, розв’язання задач з використанням графів*.*