**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра** **ВТ**

**ОТЧЕТ**

**по лабораторной работе №4**

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

**Тема: «Двоичные деревья»**

**Вариант 5**

|  |  |  |
| --- | --- | --- |
| Студент гр. 0322 |  | Кренев И.Ф. |
| Преподаватель |  | Пестерев Д.О. |

Санкт-Петербург

2022

**Постановка задачи**

Реализовать структуру данных двоичное дерево поиска и следующие методы: нахождение минимума, нахождение максимума, прямой (preorder), центрированный (inorder) и обратный обход (postorder) по дереву, поиск элемента, нахождение следующего и предыдущего элемента, удаление элемента, обход в ширину. Реализовать визуализацию дерева. Указать теоретическую временную сложность для всех операций.

С помощью реализованной структуры данных написать программу, позволяющую преобразовать запись из префиксной/инфиксной/постфиксной нотации в префиксную/инфиксную/постфиксную нотацию.

**Реализуемые функции**

* insertNode – обертка для функции вставки элемента;
* insertNodeDeep – вставка элемента;
* findNext – поиск следующего элемента;
* findPrev – поиск предыдущего элемента;
* findMin – поиск минимума;
* findMax – поиск максимума;
* inOrder – центрированный обход;
* preOrder – прямой обход;
* postOrder – обратный обход;
* searchNumData – поиск элемента;
* remove – удаление элемента;
* width – обход в ширину;
* init – инициация дерева выражений;
* toPrefix – конвертация в префиксную форму;
* toPostfix – конвертация в постфиксную форму;
* toInfix – конвертация в инфиксную форму;

**Теоретическая временная сложность**

* insertNode – O(1)
* insertNodeDeep – простой рекурсивный вызов функции O(h), где h – высота дерева;
* findNext – цикл по поддереву O(h);
* findPrev – цикл по поддереву O(h);
* findMin – простой рекурсивный вызов функции O(h);
* findMax – простой рекурсивный вызов функции O(h);
* inOrder – для каждого узла функция запускается дважды O(N);
* preOrder – для каждого узла функция запускается дважды O(N);
* postOrder – для каждого узла функция запускается дважды O(N);
* searchNumData – цикл по поддереву O(h);
* remove – цикл по поддереву O(h);
* width – функция запускается для каждого узла O(N);
* init – запускается для каждого символа строки O(N);
* toPrefix – совпадает с preOrder;
* toPostfix – совпадает с postOrder;
* toInfix – совпадает с inOrder;

**Пример работы программы**

Интерфейс веб-страницы имеет интуитивно понятный интерфейс с подсказками для полей ввода. При перезагрузке страницы производится рандомная генерация дерева. Начальное состояние программы представлено на рисунке 1.

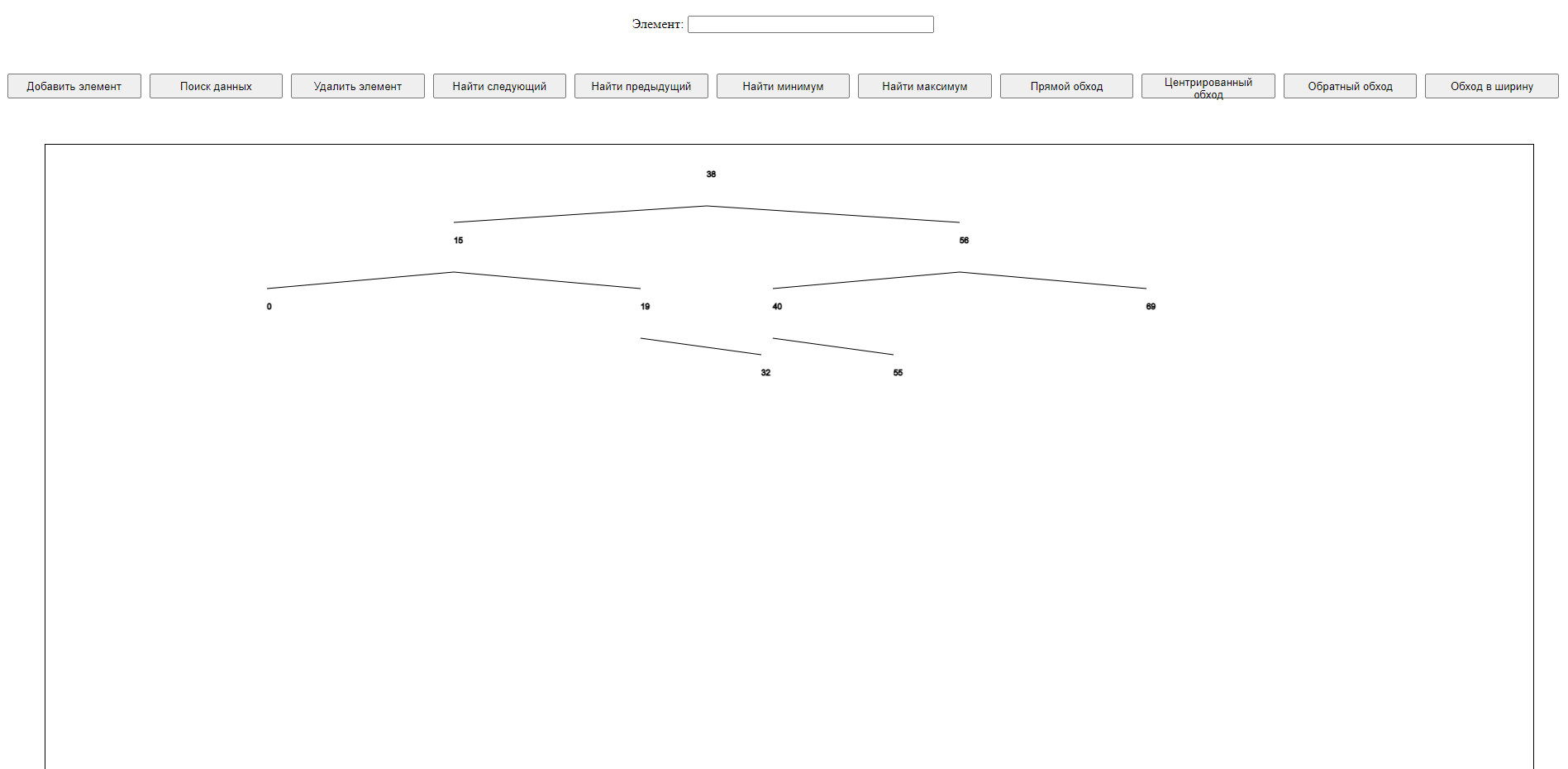


Рисунок 1. Исходное состояние программы

При поиске элемента дерева он будет подчеркнут (рисунок 2).

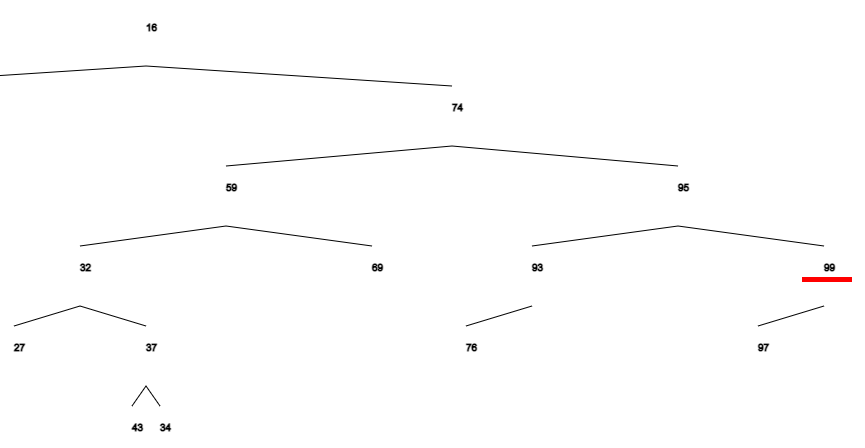


Рисунок 2. Результат поиска элемента

Для нахождения инфиксной/префиксной/постфиксной форм выражения использовано отдельное текстовое поле, оно расположено под окном с деревом (рисунок 3).

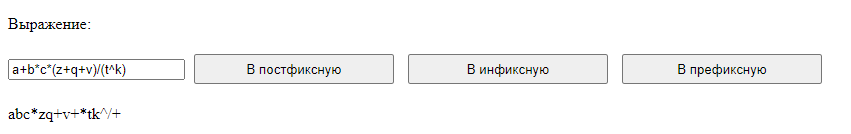


Рисунок 3. Нахождение постфиксной формы

**Выводы**

В ходе выполнения лабораторной работы были приобретены навыки реализации двоичных деревьев поиска, а также использования такого дерева для реализации дерева вычислений. Были изучены методы преобразования записей префиксной / инфиксной / постфиксной нотации в префиксную / инфиксную / постфиксную нотацию.

**Приложение 1. Листинг программного кода**

Ссылка на GitHub: <https://github.com/IlyaKrenev/aisd4>

Ссылка на GitHub Pages: <https://ilyakrenev.github.io/aisd4/>

**trees.ts:**

type Coords = {  
 x: number;  
 y: number;  
}  
  
class TreeNode<T> {  
 public childA: TreeNode<T> | null = null;  
 public childB: TreeNode<T> | null = null;  
 public parent: TreeNode<T> | null = null;  
 public data: T;  
  
 // used for UI  
 public position: Coords;  
  
 constructor (data: T) {  
 this.data = data;  
 this.position = {x: 0, y: 0}  
 }  
  
 findMin() {  
 if (!this.childA) {  
 return this;  
 }  
  
 return this.childA.findMin();  
 }  
}  
  
class BinSearchTree<T extends any = number> {  
 public rootNode: TreeNode<T> | null = null;  
 public startPosition: Coords;  
 public axisX: number;  
 public axisY: number;  
  
 constructor () {  
 this.startPosition = {x: 800, y: 44}  
 this.axisX = 350  
 this.axisY = 80  
 }  
  
 getPosition ({x , y}, ischildA = false){  
 return { x: ischildA ? x - this.axisX + y : x + this.axisX - y, y: y + this.axisY }  
 }  
  
 insertNode (node: TreeNode<T>) {  
 if (this.rootNode === null) {  
 node.position = this.startPosition  
 this.rootNode = node;  
 } else {  
 this.insertNodeDeep(this.rootNode, node);  
 }  
 }  
  
 insertNodeDeep (parentNode: TreeNode<T>, childNode: TreeNode<T>) {  
 if (parentNode.data > childNode.data) {  
 if (parentNode.childA === null) {  
 childNode.position = this.getPosition(parentNode.position, true);  
 childNode.parent = parentNode;  
 parentNode.childA = childNode;  
 } else {  
 this.insertNodeDeep(parentNode.childA, childNode);  
 }  
 } else {  
 if (parentNode.childB === null) {  
 childNode.position = this.getPosition(parentNode.position);  
 childNode.parent = parentNode;  
 parentNode.childB = childNode;  
 } else {  
 this.insertNodeDeep(parentNode.childB, childNode);  
 }  
 }  
 }  
  
 findNext (x: TreeNode<T>) {  
 if (x.childB !== null) {  
 return this.findMin(x.childB);  
 }  
  
 let y = x.parent;  
  
 while (y !== null && x === y.childB) {  
 x = y;  
 y = y.parent;  
 }  
  
 return y;  
 }  
  
 findPrev (x: TreeNode<T>) {  
 if (x.childA !== null) {  
 return this.findMax(x.childA);  
 }  
  
 let y = x.parent;  
  
 while (y !== null && x === y.childA) {  
 x = y;  
 y = y.parent;  
 }  
  
 return y;  
 }  
  
 findMin (node: TreeNode<T>) {  
 if (node.childA === null) {  
 return node;  
 }  
  
 return this.findMin(node.childA)  
 }  
  
 findMax (node: TreeNode<T>) {  
 if (node.childB === null) {  
 return node;  
 }  
  
 return this.findMax(node.childB)  
 }  
  
 inOrder (node: TreeNode<T>, callback: (node: TreeNode<T>) => void) {  
 if (node != null) {  
 this.inOrder(node.childA, callback);  
  
 callback(node);  
  
 this.inOrder(node.childB, callback);  
 }  
 }  
  
 preOrder (node: TreeNode<T>, callback: (node: TreeNode<T>) => void) {  
 if (node != null) {  
 callback(node);  
  
 this.preOrder(node.childA, callback);  
 this.preOrder(node.childB, callback);  
 }  
 }  
  
 postOrder (node: TreeNode<T>, callback: (node: TreeNode<T>) => void) {  
 if (node != null) {  
 this.postOrder(node.childA, callback);  
 this.postOrder(node.childB, callback);  
  
 callback(node);  
 }  
 }  
  
 searchNumData (node: TreeNode<T>, searchData: number): TreeNode<T> | null {  
 if (node === null) {  
 return null;  
 }  
  
 if (node.data > searchData) {  
 return this.searchNumData(node.childA, searchData);  
 }  
  
 if (node.data < searchData) {  
 return this.searchNumData(node.childB, searchData);  
 }  
  
 if (node.data === searchData) {  
 return node;  
 }  
  
 return null;  
 }  
  
 remove (root : TreeNode<T>, z: number) {  
 if (z < root.data) {  
 root.childA = this.remove(root.childA, z)  
 } else if (z > root.data) {  
 root.childB = this.remove(root.childB, z)  
 } else if (root.childA !== null && root.childB !== null) {  
 const min = this.findMin(root.childB);  
 root.data = min?.data  
 root.childB = this.remove(root.childB, root.data as number)  
 } else {  
 if (root.childA !== null) {  
 root = root.childA  
 } else if (root.childB !== null) {  
 root = root.childB  
 } else {  
 root = null;  
 }  
 }  
  
 return root;  
 }  
  
 width (root: TreeNode<T>, callback: (root: TreeNode<T>) => void) {  
 let nodeQueue = new Queue();  
 nodeQueue.enqueue(root);  
  
 while (!nodeQueue.isEmpty()) {  
 let currentNode = nodeQueue.dequeue();  
  
 callback(currentNode);  
  
 if (currentNode.childA) {  
 nodeQueue.enqueue(currentNode.childA);  
 }  
  
 if (currentNode.childB) {  
 nodeQueue.enqueue(currentNode.childB);  
 }  
 }  
 }  
}  
  
class NotationTree extends BinSearchTree<string | number> {  
 public init (str: string) {  
 function newNode(c)  
 {  
 let n = new TreeNode(c);  
 return n;  
 }  
  
 let s;  
  
 // Stack to hold nodes  
 let stN = [];  
  
 // Stack to hold chars  
 let stC = [];  
 let t, t1, t2;  
  
 // Prioritising the operators  
 let p = new ***Array***(123);  
 p['+'.charCodeAt(0)] = p['-'.charCodeAt(0)] = 1;  
 p['/'.charCodeAt(0)] = p['\*'.charCodeAt(0)] = 2;  
 p['^'.charCodeAt(0)] = 3;  
 p[')'.charCodeAt(0)] = 0;  
  
 const signs = ['-', '+', '\*', '/', '^'];  
  
 if (signs.includes(str[0])) {  
 s = InfixNotation.*fromPrefix*(str)  
 } else if (signs.includes(str[str.length - 1])) {  
 s = InfixNotation.*fromPostfix*(str)  
 } else {  
 s = str;  
 }  
  
 s = "(" + s;  
 s += ")";  
  
 for (let i = 0; i < s.length; i++)  
 {  
 if (s[i] == '(')  
 {  
  
 // Push '(' in char stack  
 stC.push(s[i]);  
 }  
  
 // Push the operands in node stack  
 else if ((/[a**-**zA**-**Z]/).test(s[i]))  
 {  
 t = newNode(s[i]);  
 stN.push(t);  
 }  
 else if (p[s[i].charCodeAt(0)] > 0)  
 {  
  
 // If an operator with lower or  
 // same associativity appears  
 while (stC.length != 0 && stC[stC.length - 1] != '('  
 && ((s[i] != '^' &&  
 p[stC[stC.length - 1].charCodeAt(0)] >=  
 p[s[i].charCodeAt(0)])  
 || (s[i] == '^'&&  
 p[stC[stC.length - 1].charCodeAt(0)] >  
 p[s[i].charCodeAt(0)])))  
 {  
  
 // Get and remove the top element  
 // from the character stack  
 t = newNode(stC[stC.length - 1]);  
 stC.pop();  
  
 // Get and remove the top element  
 // from the node stack  
 t1 = stN[stN.length - 1];  
 stN.pop();  
  
 // Get and remove the currently top  
 // element from the node stack  
 t2 = stN[stN.length - 1];  
 stN.pop();  
  
 // Update the tree  
 t.childA = t2;  
 t.childB = t1;  
  
 // Push the node to the node stack  
 stN.push(t);  
 }  
  
 // Push s[i] to char stack  
 stC.push(s[i]);  
 }  
 else if (s[i] == ')')  
 {  
 while (stC.length != 0 &&  
 stC[stC.length - 1] != '(')  
 {  
 t = newNode(stC[stC.length - 1]);  
 stC.pop();  
 t1 = stN[stN.length - 1];  
 stN.pop();  
 t2 = stN[stN.length - 1];  
 stN.pop();  
 t.childA = t2;  
 t.childB = t1;  
 stN.push(t);  
 }  
 stC.pop();  
 }  
 }  
 t = stN[stN.length - 1];  
  
 this.rootNode = t;  
 }  
  
 public toPrefix () {  
 let str = [];  
  
 const callback = (node: TreeNode<string | number>) => {  
 str.push(node.data);  
 }  
  
 this.postOrder(this.rootNode, callback);  
  
 return str.reverse().join('');  
 }  
  
 public toPostfix () {  
 let str = [];  
  
 const callback = (node: TreeNode<string | number>) => {  
 str.push(node.data);  
 }  
  
 this.postOrder(this.rootNode, callback);  
  
 return str.join('');  
 }  
  
 public toInfix () {  
 let str = [];  
  
 const callback = (node: TreeNode<string | number>) => {  
 str.push(node.data);  
 }  
  
 this.inOrder(this.rootNode, callback);  
  
 return str.join('');  
 }  
}

**main.ts:**

class Main {  
 private binSearchTree: BinSearchTree = new BinSearchTree();  
  
 constructor () {  
 this.initTree();  
 this.initButtons();  
 this.printTree();  
 this.initExpTree();  
 }  
  
 private initTree () {  
 const randAmount = randomInteger(5, 25);  
  
 for (let i = 0; i < randAmount; i++) {  
 const randNum = idGenerator.*generateId*();  
 const treeNode = new TreeNode(randNum);  
  
 this.binSearchTree.insertNode(treeNode)  
 }  
  
 ***console***.log(this.binSearchTree);  
 }  
  
 private initButtons () {  
 const container = ***document***.querySelector('.main');  
  
 const buttons = [  
 {  
 name: 'Добавить элемент',  
 handler: this.handleAddData.bind(this)  
 },  
 {  
 name: 'Поиск данных',  
 handler: this.handleSearchData.bind(this)  
 },  
 {  
 name: 'Удалить элемент',  
 handler: this.handleRemove.bind(this)  
 },  
 {  
 name: 'Найти следующий',  
 handler: this.handleFindNext.bind(this)  
 },  
 {  
 name: 'Найти предыдущий',  
 handler: this.handleFindPrev.bind(this)  
 },  
 {  
 name: 'Найти минимум',  
 handler: this.handleMin.bind(this)  
 },  
 {  
 name: 'Найти максимум',  
 handler: this.handleMax.bind(this)  
 },  
 {  
 name: 'Прямой обход',  
 handler: this.handleOrder.bind(this)  
 },  
 {  
 name: 'Центрированный обход',  
 handler: this.handlePreorder.bind(this)  
 },  
 {  
 name: 'Обратный обход',  
 handler: this.handlePostorder.bind(this)  
 },  
 {  
 name: 'Обход в ширину',  
 handler: this.handleWidth.bind(this)  
 },  
 ];  
  
 buttons.forEach((button) => {  
 const buttonEl = ***document***.createElement('button');  
  
 buttonEl.innerText = button.name;  
  
 buttonEl.addEventListener('click', () => {  
 this.hideHighlighter();  
 button.handler();  
 });  
  
 container.appendChild(buttonEl);  
 });  
 }  
  
 private handleAddData () {  
 const value = this.getInputValue()  
  
 if (!value) {  
 return;  
 }  
  
 const node = new TreeNode(value);  
  
 this.binSearchTree.insertNode(node);  
  
 this.redrawTree();  
 }  
  
 private getInputValue (): number | undefined {  
 const input = ***document***.querySelector('.searchInput') as HTMLInputElement;  
  
 if (!input) {  
 return;  
 }  
  
 if (input.value === '') {  
 return;  
 }  
  
 return parseInt(input.value)  
 }  
  
 private handleFindNext () {  
 const value = this.getInputValue()  
  
 if (!value) {  
 return;  
 }  
  
 const foundNode = this.binSearchTree.searchNumData(this.binSearchTree.rootNode, value);  
 const foundNext = this.binSearchTree.findNext(foundNode);  
  
 if (foundNext) {  
 const {x, y} = foundNext.position;  
  
 this.drawHighlighter(x, y);  
 }  
 }  
  
 private handleFindPrev () {  
 const value = this.getInputValue()  
  
 if (!value) {  
 return;  
 }  
  
 const foundNode = this.binSearchTree.searchNumData(this.binSearchTree.rootNode, value);  
 const foundNext = this.binSearchTree.findPrev(foundNode);  
  
 if (foundNext) {  
 const {x, y} = foundNext.position;  
  
 this.drawHighlighter(x, y);  
 }  
 }  
  
 private handleRemove () {  
 const value = this.getInputValue()  
  
 if (!value) {  
 return;  
 }  
  
 this.binSearchTree.remove(this.binSearchTree.rootNode, value);  
 this.redrawTree();  
 }  
  
 private handleOrder () {  
 const callback = (node: TreeNode<number>) => {  
 ***console***.log(node.data);  
 }  
  
 this.binSearchTree.inOrder(this.binSearchTree.rootNode, callback)  
 }  
  
 private handlePreorder () {  
 const callback = (node: TreeNode<number>) => {  
 ***console***.log(node.data);  
 }  
  
 this.binSearchTree.preOrder(this.binSearchTree.rootNode, callback)  
 }  
  
 private handlePostorder () {  
 const callback = (node: TreeNode<number>) => {  
 ***console***.log(node.data);  
 }  
  
 this.binSearchTree.postOrder(this.binSearchTree.rootNode, callback)  
 }  
  
  
 private handleWidth () {  
 const callback = (node: TreeNode<number>) => {  
 ***console***.log(node.data);  
 }  
  
 this.binSearchTree.width(this.binSearchTree.rootNode, callback)  
 }  
  
 private handleMin () {  
 const foundNode = this.binSearchTree.findMin(this.binSearchTree.rootNode);  
  
 if (foundNode) {  
 const {x, y} = foundNode.position;  
  
 this.drawHighlighter(x, y);  
 }  
 }  
  
 private handleMax () {  
 const foundNode = this.binSearchTree.findMax(this.binSearchTree.rootNode);  
  
 if (foundNode) {  
 const {x, y} = foundNode.position;  
  
 this.drawHighlighter(x, y);  
 }  
 }  
  
 private handleSearchData () {  
 const value = this.getInputValue()  
  
 if (!value) {  
 return;  
 }  
  
 const foundNode = this.binSearchTree.searchNumData(this.binSearchTree.rootNode, value);  
  
 if (foundNode) {  
 const {x, y} = foundNode.position;  
  
 this.drawHighlighter(x, y);  
 }  
 }  
  
 private redrawTree () {  
 const myCanvas = ***document***.querySelector('.myCanvas') as HTMLCanvasElement;  
 const ctx = myCanvas.getContext('2d');  
  
 ctx.clearRect(0, 0, 1000, 1000);  
  
 this.printTree();  
 }  
  
 private printTree () {  
 const myCanvas = ***document***.querySelector('.myCanvas') as HTMLCanvasElement;  
 const ctx = myCanvas.getContext('2d');  
  
 this.drawGraphics(ctx)  
 }  
  
 private drawGraphics (ctx: CanvasRenderingContext2D) {  
 const queue: TreeNode<number>[] = [];  
 const black = "#000"  
 queue.push(this.binSearchTree.rootNode)  
  
 while (queue.length !== 0) {  
 const node = queue.shift()  
 const {x, y} = node.position  
 const color = "#" + ( (1<<24) \* ***Math***.random() | 0 ).toString(16)  
  
 ctx.beginPath()  
 ctx.strokeStyle = black  
 ctx.fillStyle = color  
 ctx.fill()  
 ctx.stroke()  
 ctx.strokeStyle = black  
 ctx.strokeText(`${String(node.data)}`, x, y - 5);  
  
 [node.childA, node.childB].forEach((child) => {  
 if (child) {  
 const {x: x1, y: y1} = child.position  
 ctx.beginPath();  
 ctx.moveTo(x, y + 30)  
 ctx.lineTo(x1, y1 - 30)  
 ctx.stroke()  
 queue.push(child)  
 }  
 })  
 }  
 }  
  
 private drawHighlighter (x: number, y: number) {  
 const el = ***document***.querySelector('.highlighter') as HTMLDivElement  
  
 if (el) {  
 el.className = 'highlighter';  
  
 el.style.left = `${x + 37}px`;  
 el.style.top = `${y + 180}px`;  
 el.style.display = 'block';  
 }  
 }  
  
 private hideHighlighter () {  
 const el = ***document***.querySelector('.highlighter') as HTMLDivElement;  
  
 el.style.display = 'none';  
 }  
  
 private initExpTree () {  
 const button1 = ***document***.querySelector('.expButton1') as HTMLButtonElement;  
 const button2 = ***document***.querySelector('.expButton2') as HTMLButtonElement;  
 const button3 = ***document***.querySelector('.expButton3') as HTMLButtonElement;  
 const result = ***document***.querySelector('.expRes') as HTMLParagraphElement;  
  
  
 button1.addEventListener('click', () => {  
 const expInput = ***document***.querySelector('.expInput') as HTMLInputElement;  
 const notationTree = new NotationTree();  
  
 if (expInput.value === '') {  
 return;  
 }  
  
 notationTree.init(expInput.value);  
  
 result.textContent = notationTree.toPostfix();  
 });  
  
 button2.addEventListener('click', () => {  
 const expInput = ***document***.querySelector('.expInput') as HTMLInputElement;  
 const notationTree = new NotationTree();  
  
 if (expInput.value === '') {  
 return;  
 }  
  
 notationTree.init(expInput.value);  
  
 result.textContent = notationTree.toInfix();  
 });  
  
 button3.addEventListener('click', () => {  
 const expInput = ***document***.querySelector('.expInput') as HTMLInputElement;  
 const notationTree = new NotationTree();  
  
 if (expInput.value === '') {  
 return;  
 }  
  
 const reversed = expInput.value.split('').reverse().join('');  
  
 notationTree.init(reversed);  
  
 result.textContent = notationTree.toPrefix();  
 });  
 }  
}  
  
  
  
const ***a*** = new Main();