**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра ИС**

отчет

**по лабораторной работе №4**

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

Тема: **Реализация красно-черного дерева поиска**

|  |  |  |
| --- | --- | --- |
| Студент гр. 6363 |  | Листратенко В. В. |
| Преподаватель |  | Михнович А. Г. |

Санкт-Петербург

2017

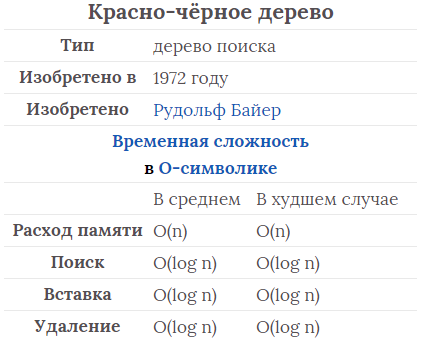
**Цель работы.**

Реализовать структуру данных красно-черное дерево поиска.

**Задачи**.

1. Написать функцию добавления элементов в дерево.
2. Написать функцию удаления элементов по ключу.
3. Написать функцию поиска элементов по ключу
4. Написать функцию поиска максимального и минимального элементов
5. Написать функцию обхода дерева в инфиксной, префиксной и постфиксной формах.

**Теоретическая часть.**

 Красно-чёрное дерево — это одно из самобалансирующихся двоичных деревьев поиска, гарантирующих логарифмический рост высоты дерева от числа узлов и быстро выполняющее основные операции дерева поиска: добавление, удаление и поиск узла. Сбалансированность достигается за счёт введения дополнительного атрибута узла дерева — «цвета». Этот атрибут может принимать одно из двух возможных значений — «чёрный» или «красный».

**Практическая часть.**

Имеется пользовательская структура данных **class tree**, имеющая четыре поля типа **int** и четыре типа **tree**\*.

Пользователю предоставляется меню, в котором он может выбрать следующие действия с деревом: добавление элементов, удаление элемента по ключу, поиск элемента по ключу, удаление дерева, поиск наибольшего и наименьшего значений, обход дерева и вывод его элементов.

В процессе выполнения программы осуществляются следующие действия:

* Добавление элемента в дерево. Функция **void addNode(int, tree\*&, tree\*);** Принимаемые аргументы: число – значение элемента, тип – **int;** ссылка на указатель на корень, тип – **tree\*\*;** указательна родителякорня, тип – **tree\***;Возвращаемого значения нет.
* Удаление элемента из дерева. Функция **void delNode(int, tree);** Принимаемые аргументы: число – значение элемента, тип – **int;** ссылка на указатель на корень, тип – **tree.** Возвращаемого значения нет.
* Вывод дерева в консоль в инфиксной форме. Функция **void showTree(tree\*, int).** Принимаемые аргументы: указатель на корень, тип – **tree**\*; число, хранящее найденный элемент, тип - **int**.Возвращаемого значения нет.
* Вывод дерева в корень в префиксной форме. Функция **void showTree\_prefix(tree\*).** Принимаемые аргументы: указатель на корень, тип – **tree\*.** Возвращаемого значения нет.
* Вывод дерева в корень в постфиксной форме. Функция **void showTree\_postfix(tree\*).** Принимаемые аргументы: указатель на корень, тип – **tree\*.** Возвращаемого значения нет.
* Поиск элемента по значению. Функция **tree\* findNode(int, tree\*);** Принимаемые аргументы: число – ключ, тип – **int**; указатель на структуру, тип – **tree\*.** Возвращает указатель на найденный элемент.
* Поиск максимального элемента. Функция **tree\* findMax(tree\*);** Принимаемые аргументы: указатель на структуру, тип – **tree\*.** Возвращает указатель на найденный элемент.
* Поиск минимального элемента. Функция **tree\* findMin(tree\*);** Принимаемые аргументы: указатель на структуру, тип – **tree\*.** Возвращает указатель на найденный элемент.
* Удаление дерева. Функция **void delTree(tree\*&);** Принимаемые аргументы: ссылка на указатель на корень, тип – **tree\*\*. \*.** Возвращаемого значения нет.

**Вывод.**

В ходе выполнения работы было реализовано бинарное дерево поиска. Основным преимуществом красно-черного дерева поиска перед аналогичными структурами данных является высокая скорость выполнения основных команд: вставка, удаление, поиск.

Красно-черное дерево поиска применяется для построения более абстрактных структур, таких, как множества, мультимножества, ассоциативные массивы.

**Приложение**

1. #include <iostream>
2. #include <Windows.h>
3. #include <conio.h>
4. #include <unistd.h>
5. #include "mylib.h"
7. const int STEPX = 35;
8. const int STEPY = 35;
9. const int RADIUS = 24;
10. const int BLACK = 0;
11. const int RED = 1;
13. PAINTSTRUCT ps;
14. HWND hwnd = GetConsoleWindow();
15. HDC dc = GetDC(hwnd);
16. HBRUSH brush;
18. using namespace std;
20. class tree{
21. public:
22. int val;
23. int col;
24. int line;
25. int color;
26. tree\* left;
27. tree\* right;
28. tree\* parent;
29. tree\* next;
31. tree();
32. };
34. tree::tree(){
35. val = 0;
36. next = left = right = parent = nullptr;
37. }
39. void showTree\_infix(tree\*);
41. void showTree\_prefix(tree\*);
43. void showTree\_postfix(tree\*);
45. void showTree\_weightFirst(tree\*, tree\*&, tree\*&, int);
47. void addNode(int, tree\*&, tree\*);
49. int delNode(int, tree\*&);
51. tree\* findNode(int, tree\*);
53. tree\* findMax(tree\*);
55. tree\* findMin(tree\*);
57. void delTree(tree\*&);
59. void menu(int);
61. void alignTree\_left(tree\*);
63. void alignTree\_right(tree\*);
65. int countSteps\_right(int&, tree\*);
67. int countSteps\_left(int&, tree\*);
69. void shiftNode(tree\*);
71. void paintCarcas(tree\*);
73. void rotate\_left(tree\*);
75. void rotate\_right(tree\*);
77. tree\* getGrandparent(tree\*);
79. tree\* getUncle(tree\*);
81. tree\* getBrother(tree\*);
83. void riseTree(tree\*);
85. void downTree(tree\*);
87. void insert\_case1(tree\*);
89. void insert\_case2(tree\*);
91. void insert\_case3(tree\*);
93. void insert\_case4(tree\*);
95. void insert\_case5(tree\*);
97. void delete\_case1(tree\*);
99. void delete\_case2(tree\*);
101. void delete\_case3(tree\*);
103. void delete\_case4(tree\*);
105. void delete\_case5(tree\*);
107. void delete\_case6(tree\*);
109. int main(){
110. SetConsoleCP(1251);
111. SetConsoleOutputCP(1251);
113. system("title Бинарное дерево");
115. tree\* node = nullptr;
116. tree\* find = nullptr;
117. tree\* head = nullptr;
118. tree\* tail = nullptr;
119. char\* temp;
120. char ch;
122. while (1){
123. system("cls");
124. if (node == nullptr){
125. cout << "Дерево пусто**\n\n**";
126. menu(1);
127. ch = getch();
129. while((ch != '1') && (ch != 27)){
130. printf("**\a**");
131. ch = getch();
132. }
133. }
134. else{
135. //Выравниваем дерево
136. //свдигаем всех потомков
137. shiftNode(node->left);
138. shiftNode(node->right);
139. //левое
140. alignTree\_left(node->left);
141. //правое
142. alignTree\_right(node->right);
143. menu(2);
144. // cout << node->right->color;
146. paintCarcas(node); // строим каркас
147. if (find == nullptr) showTree\_weightFirst(node, head, tail, 0);
148. else showTree\_weightFirst(node, head, tail, find->val);
149. // showTree\_weightFirst(node, head, tail, 0);
151. find = nullptr;
152. ch = getch();
154. while(((ch < '1') || (ch>'9')) && (ch != 27)){
155. printf("**\a**");
156. ch = getch();
157. }
158. }
160. switch (ch){
161. case '1':
162. cout << "Вводите числа. Для завершения ввода нажмите Enter:**\n**";
164. while (1){
165. temp = readInt();
166. if (temp[0] == '**\0**') break;
167. addNode(atoi(temp), node, node);
168. cout << " ";
169. if (node->parent != nullptr)
170. node = node->parent;
171. }
172. // cout << "\n";
173. break;
175. case '2':
176. cout << "Для удаления элемента введите его ключ**\n**";
178. temp = readInt();
179. if (temp[0] == '**\0**') break;
180. ch = delNode(atoi(temp), node);
182. if (!ch){
183. cout << "**\n**Элемент с таким ключем не найден...**\n\n**";
184. system("pause");
185. }
186. break;
188. case '3':
189. cout << "Введите ключ искомого элемента**\n**";
191. temp = readInt();
192. if (temp[0] == '**\0**') break;
193. find = findNode(atoi(temp), node);
195. if (find == nullptr){
196. cout << "**\n**Элемент с таким ключем не найден...**\n\n**";
197. system("pause");
198. }
199. break;
201. case '4':
202. find = findMax(node);
203. break;
205. case '5':
206. find = findMin(node);
207. break;
209. case '6':
210. cout << "Префиксная форма:**\n\n**";
211. showTree\_prefix(node);
212. cout << "**\n**";
213. system("pause");
214. break;
216. case '7':
217. cout << "Инфиксная форма:**\n\n**";
218. showTree\_infix(node);
219. cout << "**\n**";
220. system("pause");
221. break;
223. case '8':
224. cout << "Постфиксная форма:**\n\n**";
225. showTree\_postfix(node);
226. cout << "**\n**";
227. system("pause");
228. break;
230. case '9':
231. delTree(node);
232. node = nullptr;
234. cout << "Дерево удалено...**\n\n**";
235. system("pause");
236. break;
238. case 27:
239. return 0;
240. }
241. if ((node != nullptr) && (node->parent != nullptr)) node = node->parent;
242. }
243. } //main
245. void showTree\_infix(tree\* root){
246. if (root == nullptr) return;
248. showTree\_infix(root->left);
250. cout << root->val << " ";
252. showTree\_infix(root->right);
253. }
255. void showTree\_prefix(tree\* root){
256. if (root == nullptr) return;
258. cout << root->val << " ";
260. showTree\_prefix(root->left);
262. showTree\_prefix(root->right);
263. }
265. void showTree\_postfix(tree\* root){
266. if (root == nullptr) return;
268. showTree\_postfix(root->left);
270. showTree\_postfix(root->right);
272. cout << root->val << " ";
273. }
275. void showTree\_weightFirst(tree\* node, tree\*& head, tree\*& tail, int max){
276. SetBkMode(dc, TRANSPARENT);
277. SetTextColor(dc, RGB(255, 255, 255));
278. char strPrint[3] {0};
280. if (node == nullptr) return;
281. tree \*p = nullptr;
282. char flag = 0;
284. if (head == nullptr) flag = 1;
286. // sleep(1);
287. itoa(node->val, strPrint, 10);//value в строку
289. if (node->val == max){
290. brush = CreateSolidBrush(RGB(0, 255, 0));
291. SelectObject(dc, brush);
292. Ellipse(dc, node->col-5, node->line-5, node->col+RADIUS, node->line+RADIUS);
293. TextOut(dc, node->col+3, node->line, strPrint, 3);
294. }
295. else {
296. if (node->color == RED) brush = CreateSolidBrush(RGB(255, 0, 0));
297. if (node->color == BLACK) brush = CreateSolidBrush(RGB(0, 0, 0));

300. SelectObject(dc, brush);
301. Ellipse(dc, node->col-5, node->line-5, node->col+RADIUS, node->line+RADIUS);
302. TextOut(dc, node->col+3, node->line, strPrint, 3);
303. }
305. if (node->left != nullptr){
306. p = new tree;
307. p->val = node->left->val;
308. p->col = node->left->col;
309. p->line = node->left->line;
310. p->left = node->left->left;
311. p->right = node->left->right;
312. p->parent = node->left->parent;
313. p->color = node->left->color;
314. if (tail != nullptr) tail->next = p;
315. tail = p;
316. if (head == nullptr) head = tail;
317. }
319. if (node->right != nullptr){
320. p = new tree;
321. p->val = node->right->val;
322. p->col = node->right->col;
323. p->line = node->right->line;
324. p->left = node->right->left;
325. p->right = node->right->right;
326. p->parent = node->right->parent;
327. p->color = node->right->color;
328. if (tail != nullptr) tail->next = p;
329. tail = p;
330. if (head == nullptr) head = tail;
331. }
333. if (!flag){
334. p = head;
335. head = head->next;
336. if (head == nullptr) tail = nullptr;
337. delete(p);
338. }
340. DeleteObject(brush);
341. showTree\_weightFirst(head, head, tail, max);
342. }
344. void paintCarcas(tree\* node){
345. if (node == nullptr) return;
347. paintCarcas(node->left);
349. if (node->parent == nullptr) MoveToEx(dc, node->col+10, node->line+10, NULL);
351. if (node->left != nullptr) {
352. MoveToEx(dc, node->col+10, node->line+10, NULL);
353. LineTo(dc, node->left->col+10, node->left->line+10);
354. }
356. if (node->right != nullptr) {
357. MoveToEx(dc, node->col+10, node->line+10, NULL);
358. LineTo(dc, node->right->col+10, node->right->line+10);
359. }
361. paintCarcas(node->right);
362. }
364. void addNode(int value, tree\*& node, tree\* parent){
365. if (node == nullptr){
366. node = new tree;
367. node->val = value;
368. node->parent = parent;
369. node->color = RED;
371. if (parent == nullptr){
372. node->col = 400;
373. node->line = 340;
374. }
375. else{
376. if (parent->left == node) node->col = parent->col - STEPX;
377. if (parent->right == node) node->col = parent->col + STEPX;
379. node->line = parent->line + STEPY;
380. }
381. //оптимизация. Случай 1
382. insert\_case1(node);
384. return;
385. }
387. if (value < node->val){
388. addNode(value, node->left, node);
389. }
390. else addNode(value, node->right, node);
391. }
393. void insert\_case1(tree\* node){
394. if (node->parent == nullptr)
395. node->color = BLACK;
396. else insert\_case2(node);
397. }
399. void insert\_case2(tree\* node){
400. if (node->parent->color == BLACK) return;
401. else insert\_case3(node);
402. }
404. void insert\_case3(tree\* node){
405. tree\* uncle = getUncle(node);
406. tree\* grand;
408. if ((uncle != nullptr) && (uncle->color == RED)){
409. node->parent->color = BLACK;
410. uncle->color = BLACK;
411. grand = getGrandparent(node);
412. grand->color = RED;
413. insert\_case1(grand);
414. }
415. else insert\_case4(node);
416. }
418. void insert\_case4(tree\* node){
419. tree\* grand = getGrandparent(node);
421. if ((node == node->parent->left) && (node->parent == grand->right)){
422. rotate\_right(node->parent);
423. node = node->right;
424. }
425. else if ((node == node->parent->right) && (node->parent == grand->left)){
426. rotate\_left(node->parent);
427. node = node->left;
428. }
430. insert\_case5(node);
431. }
433. void insert\_case5(tree\* node){
434. tree\* grand = getGrandparent(node);
435. node->parent->color = BLACK;
436. grand->color = RED;
437. if ((node == node->parent->left) && (node->parent == grand->left))
438. rotate\_right(grand);
439. else rotate\_left(grand);
440. }
442. int delNode(int value, tree\*& node){
443. int res;
444. tree\* p;
446. if (node == nullptr){
447. return 0;
448. }
450. if (node->val == value){
451. //Если это лист
452. if ((node->left == nullptr) && (node->right == nullptr)){
453. //Если корень
454. if (node->parent == nullptr){
455. p = node;
456. node = nullptr;
457. delete p;
458. return 1;
459. }
460. else {
461. if (node->color == BLACK) delete\_case1(node);
463. if (node->parent->left == node) node->parent->left = nullptr;
464. else node->parent->right = nullptr;
466. delete node;
467. return 1;
468. }
469. }
470. //Если есть оба потомока
471. if ((node->left != nullptr) && (node->right != nullptr)){
473. p = findMax(node->left);
474. node->val = p->val;
476. delNode(p->val, p);
477. return 1;
478. }
479. else{//Если один потомок
480. if (node->left != nullptr){
481. p = findMax(node->left);
482. node->val = p->val;
484. delNode(p->val, p);
485. return 1;
486. }
487. else {
488. p = findMin(node->right);
489. node->val = p->val;
491. delNode(p->val, p);
492. return 1;
493. }
494. return 1;
495. }
496. }
498. res = delNode(value, node->left);
499. if (res) return 1;
501. res = delNode(value, node->right);
502. if (res) return 1;
503. else return 0;
505. }
507. void delete\_case1(tree\* node){
508. if (node->parent != nullptr) delete\_case2(node);
509. }
511. void delete\_case2(tree\* node){
512. tree\* brother = getBrother(node);
514. if ((brother != nullptr) && (brother->color == RED)){
515. node->parent->color = RED;
516. brother->color = BLACK;
518. if (node == node->parent->left) rotate\_left(node->parent);
519. else rotate\_right(node->parent);
520. }
522. delete\_case3(node);
523. }
525. void delete\_case3(tree\* node){
526. tree\* brother = getBrother(node);
528. if ((node->parent->color == BLACK) &&
529. (brother->color == BLACK) &&
530. ((brother->left == nullptr) || (brother->left->color == BLACK)) &&
531. ((brother->right == nullptr) || (brother->right->color == BLACK))){
532. brother->color = RED;
533. delete\_case1(node->parent);
534. } else
535. delete\_case4(node);
536. }
538. void delete\_case4(tree\* node){
539. tree\* brother = getBrother(node);
541. if ((node->parent->color == RED) &&
542. (brother->color == BLACK) &&
543. ((brother->left == nullptr) || (brother->left->color == BLACK)) &&
544. ((brother->right == nullptr) || (brother->right->color == BLACK))){
545. brother->color = RED;
546. node->parent->color = BLACK;
547. } else
548. delete\_case5(node);
549. }
551. void delete\_case5(tree\* node){
552. tree\* brother = getBrother(node);
554. if (brother->color == BLACK){
555. if ((node == node->parent->left) &&
556. (brother->left != nullptr) &&
557. ((brother->right == nullptr) || (brother->right->color == BLACK)) &&
558. (brother->left->color == RED)){
559. brother->color = RED;
560. brother->left->color = BLACK;
561. rotate\_right(brother);
562. }
563. else if ((node == node->parent->right) &&
564. (brother->right != nullptr) &&
565. ((brother->left == nullptr) || (brother->left->color == BLACK)) &&
566. (brother->right->color == RED) &&
567. (brother->left->color == BLACK)){
568. brother->color = RED;
569. brother->right->color = BLACK;
570. rotate\_left(brother);
571. }
572. }
573. delete\_case6(node);
574. }
576. void delete\_case6(tree\* node){
577. tree\* brother = getBrother(node);
579. brother->color = node->parent->color;
580. node->parent->color = BLACK;
582. if (node == node->parent->left){
583. brother->right->color = BLACK;
584. rotate\_left(node->parent);
585. } else{
586. brother->left->color = BLACK;
587. rotate\_right(node->parent);
588. }
589. }
591. tree\* findNode(int value, tree\* node){
592. tree\* p;
594. if (node == nullptr) return nullptr;
596. if (node->val == value) return node;
597. p = findNode(value, node->left);
598. if (p != nullptr) return p;
600. p = findNode(value, node->right);
601. if (p != nullptr) return p;
602. else return nullptr;
603. }
605. tree\* findMax(tree\* node){
606. while (node->right != nullptr){
607. node = node->right;
608. }
609. return node;
610. }
612. tree\* findMin(tree\* node){
613. while (node->left != nullptr){
614. node = node->left;
615. }
616. return node;
617. }
619. void alignTree\_left(tree\* node){
620. if (node == nullptr) return;
621. int steps = 0;
622. tree\* p;
623. countSteps\_right(steps, node);
624. countSteps\_left(steps, node->parent->right);
625. node->col -= steps\*STEPX;
627. //свдигаем всех потомков
628. shiftNode(node->left);
629. shiftNode(node->right);
631. //продолжаем сдвигать по шагам
632. alignTree\_left(node->left);
633. if (node->right != nullptr) alignTree\_left(node->right->left);
634. }
636. void alignTree\_right(tree\* node){
637. if (node == nullptr) return;
638. int steps = 0;
639. tree\* p;
640. countSteps\_left(steps, node);
641. countSteps\_right(steps, node->parent->left);
642. node->col += steps\*STEPX;
644. //свдигаем всех потомков
645. shiftNode(node->left);
646. shiftNode(node->right);
648. //продолжаем сдвигать по шагам
649. alignTree\_right(node->right);
650. if (node->left != nullptr) alignTree\_right(node->left->right);
651. }
653. int countSteps\_right(int& steps, tree\* node){
654. if (node == nullptr) return steps;
655. if (node->right != nullptr) steps++;
656. else return steps;
657. countSteps\_right(steps, node->right);
658. }
660. int countSteps\_left(int& steps, tree\* node){
661. if (node == nullptr) return steps;
662. if (node->left != nullptr) steps++;
663. else return steps;
664. countSteps\_left(steps, node->left);
665. }
667. void shiftNode(tree\* node){
668. if (node == nullptr) return;
669. if (node == node->parent->left) node->col = node->parent->col - STEPX;
670. else node->col = node->parent->col + STEPX;
671. shiftNode(node->left);
672. shiftNode(node->right);
673. }
675. void delTree(tree\*& root){
676. if (root == nullptr) return;
678. if (root->left != nullptr)
679. delTree(root->left);
680. if (root->right != nullptr)
681. delTree(root->right);
682. // delNode(root->val, root);
683. delete root;
684. return;
685. }
687. void rotate\_left(tree\* node){
688. tree\* pivot;
690. pivot = node->right;
692. if (pivot == nullptr) return;
694. //Меняем родителя node
695. pivot->parent = node->parent;
697. if (node->parent != nullptr){
698. if (node == node->parent->left)
699. node->parent->left = pivot;
700. else node->parent->right = pivot;
701. }
703. //Меняем потомка pivot
704. node->right = pivot->left;
705. if (pivot->left != nullptr)
706. pivot->left->parent = node;
708. //меняем местами node and pivot
709. node->parent = pivot;
710. pivot->left = node;
712. //координаты
713. node->line += STEPY;
714. pivot->line -=STEPY;
715. riseTree(pivot->right);
716. downTree(node->left);
717. }
719. void rotate\_right(tree\* node){
720. tree\* pivot;
722. pivot = node->left;
723. if (pivot == nullptr) return;
725. //Меняем родителя node
726. pivot->parent = node->parent;
728. if (node->parent != nullptr){
729. if (node == node->parent->left)
730. node->parent->left = pivot;
731. else node->parent->right = pivot;
732. }
734. //Меняем потомка pivot
735. node->left = pivot->right;
736. if (pivot->right != nullptr)
737. pivot->right->parent = node;
739. //меняем местами node and pivot
740. node->parent = pivot;
741. pivot->right = node;
743. //координаты
744. node->line += STEPY;
745. pivot->line -= STEPY;
746. riseTree(pivot->left);
747. downTree(node->right);
748. }
750. tree\* getGrandparent(tree\* node){
751. if ((node != nullptr) && (node->parent != nullptr))
752. return node->parent->parent;
753. else return nullptr;
754. }
756. tree\* getUncle(tree\* node){
757. if (node == nullptr) return nullptr;
758. tree\* grand = getGrandparent(node);
760. if (grand == nullptr) return nullptr;
762. if (node->parent == grand->left) return grand->right;
763. else return grand->left;
764. }
766. tree\* getBrother(tree\* node){
767. if (node == nullptr) return nullptr;
769. if (node->parent == nullptr) return nullptr;
771. if (node == node->parent->left) return node->parent->right;
772. else return node->parent->left;
773. }
775. int isLeaf(tree\* node){
776. if (node == nullptr) return -1;
778. if ((node->left == nullptr) && (node->right == nullptr))
779. return 1;
780. else return 0;
781. }
783. void riseTree(tree\* node){
784. if (node == nullptr) return;
786. node->line -= STEPY;
787. riseTree(node->left);
788. riseTree(node->right);
789. return;
790. }
792. void downTree(tree\* node){
793. if (node == nullptr) return;
794. node->line += STEPY;
795. downTree(node->left);
796. downTree(node->right);
797. return;
798. }
800. void menu(int p){
801. int i = 1;
802. if (p == 1){
803. cout << i++ << ". Добавить элементы в дерево.**\n**";
804. cout << "Esc. Завершить программу.**\n**";
805. cout << "**\n**";
806. return;
807. }
809. if (p == 2){
810. cout << i++ << ". Добавить элементы в дерево**\n**";
811. cout << i++ << ". Удалить элемент по ключу**\n**";
812. cout << i++ << ". Найти элемент по ключу**\n**";
813. cout << i++ << ". Найти максимальный элемент**\n**";//4
814. cout << i++ << ". Найти минимальный элемент**\n**";
815. cout << i++ << ". Вывести дерево в префиксной форме**\n**";
816. cout << i++ << ". Вывести дерево в инфиксной форме**\n**";
817. cout << i++ << ". Вывести дерево в постфиксной форме**\n**";
818. cout << i++ << ". Удалить дерево**\n**";//9
819. // cout << "\n";
820. cout << "Esc. Завершить программу**\n\n**";
821. return;
822. }
823. }