МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА - Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению практического задания №3

Тема: Нелинейные структуры данных. Бинарное дерево.

Дисциплина: «Структуры и алгоритмы обработки данных»

Выполнил студент Хан А.А.

группа ИКБО-04-20

Москва 2021

Оглавление

Тема

Нелинейные структуры данных. Бинарное дерево.

Цель

Получение умений и навыков разработки и реализаций операций над структурой данных бинарное дерево.

Персональный вариант

Вариант №9

Вариант	Значение информационной части	Операции варианта
9	Символьное значение	Проверить, является ли дерево деревом выражений. Вывести дерево, отобразить его формулу. Определить, содержит ли дерево операцию *

Постановка задачи

Для вариантов с 8 по 15

Разработать программу, которая создает дерево выражений и выполняет операции.

1. Реализовать операции общие для всех вариантов

1) Создать дерево выражений в соответствии с выражением. Структура узла дерева включает: информационная часть узла – символьного типа: либо знак операции +, -, * либо цифра, указатель на левое и указатель на правое поддерево. В дереве выражения операнды в листьях дерева. Исходное выражение имеет формат: ::=цифра|

2) Отобразить дерево на экране, используя алгоритм ввода дерева повернутым справа налево.

2. Реализовать операции варианта.

3. Разработать программу на основе меню, позволяющего проверить выполнение всех операций на ваших тестах и тестах преподавателя.

4. Оформить отчет.

3) Для каждой представленной в программе функции предоставить отчет по ее разработке в соответствии с требованиями разработки программы (подпрограммы).

4) Представить алгоритм основной программы и таблицу имен, используемых в алгоритме.

Пояснение к коду

Теоретическая база.

Будем решать задачу о разборе арифметического выражения методом рекурсивного спуска.

Воспользуемся формой Бэкуса-Наура – формальной системой описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие категории. В нашем случае:

$$< \text{Выражение} > ::= < \text{Слагаемое 1} > \pm < \text{Слагаемое 2} > \pm \cdots \pm < \text{Слагаемое } n >$$

$$< \text{Слагаемое} > ::= < \text{Множитель 1} > * < \text{Множитель 2} > * \ldots * < \text{Множитель } k >$$

$$< \text{Множитель} > ::= < \text{Число} > \mid < \text{Выражение} >$$

Алгоритм

В первую очередь необходимо входную строку (арифметическое выражение) разбить на лексемы (+, -, *, (,), числа). Заведем структуру $lexem$:

```
struct lexem {
    char c;
    int num;

    lexem(char _c = 'n', int _num = 0) {
        c = _c;
        num = _num;
    }
};
```

$c = '+' \mid '-' \mid '*' \mid '(' \mid ')' \mid 'n'$

Тогда конструктор для лексемы-числа: $lexem('n', num)$, где num – само число, а для лексем-операций: $lexem('+')$, $lexem('-')$, $lexem('*'), lexem('('), lexem(')')$.

Введем глобальные переменные:

- $LEXEMS$ – динамический массив лексем.
- IND – индекс лексемы, рассматриваемой в текущий момент.
- $WRONG$ – бинарный флаг, единичное значение которого означает, что введенная строка задает некорректное арифметическое выражение.

И вспомогательные функции:

- $void\ skip\_space(int\ \&i, string\ \&s)$ – пропуск пробельных символов в строке с выражением.
- $bool\ is\_operation(char\ c)$ – является ли символ арифметической операцией.
- $bool\ is\_digit(char\ c)$ – является ли символ цифрой.

```
vector <lexem> LEXEMS;
int IND = 0;
bool WRONG = 0;

void skip_space(int &i, string &s) {
    while (i < s.size() && isspace(s[i])) {
        ++i;
    }
}

bool is_operation(char c) {
    return (c == '+') || (c == '-') || (c == '*') || (c == '(') || (c == ')');
}

bool is_digit(char c) {
    return '0' <= c && c <= '9';
}
```

Далее введем функцию разбиения строки s на лексемы, заполняя динамический массив $LEXEMS$ и, в случае некорректного выражения, заполняя $WRONG = 1$.

```
void get_lexem(string &s) {
    int i = 0;
    while (1) {
        skip_space(i, s);
        if (i >= s.size()) {
            break;
        }
        if (is_digit(s[i])) {
            int x = 0;
            while (i < s.size() && is_digit(s[i])) {
                x *= 10;
                x += (s[i] - '0');
                ++i;
            }
            LEXEMS.push_back(lexem('n', x));
        } else if (is_operation(s[i])) {
            LEXEMS.push_back(lexem(s[i]));
            ++i;
        } else {
            WRONG = 1;
        }
    }
}
```

Для решения задачи о построении бинарного дерева арифметического выражения необходимо ввести структуру $node$ – обозначающую вершину дерева. В ней находятся поля:

- $type$ - тип вершины ($'+'|'-'|'*'|'n'$, где $'n'$ - для чисел)
- val – для вершины-числа само значение числа
- $left\_node$ – указатель на левое поддерево данной вершины
- $right\_node$ – указатель на правое поддерево данной вершины

```
struct node {
    char type;
    int val;
    node * left;
    node * right;

    node(char _type = 'n', int _val = 0) {
        type = _type;
        val = _val;
        left = NULL;
        right = NULL;
    }
};
```

Перейдем к основной части программы – функциям:

- $node * expression()$ – создает вершины дерева, отвечающие синтаксической структуре < Выражение >.

```
node * expression() {
    node * v = new node();
    v->left = item();

    while (!WRONG && IND < LEXEMS.size() && (LEXEMS[IND].c == '+' || LEXEMS[IND].c == '-')) {
        v->type = LEXEMS[IND].c;
        ++IND;
        v->right = item();

        node * new_v = new node();
        new_v->left = v;
        v = new_v;
    }

    return v->left;
}
```

- $node * item()$ – создает вершины дерева, отвечающие синтаксической структуре < Слагаемое >.

```
node * item() {
    node * v = new node();
    v->left = mult();

    while (!WRONG && IND < LEXEMS.size() && LEXEMS[IND].c == '*') {
        v->type = LEXEMS[IND].c;
        ++IND;
        v->right = mult();

        node * new_v = new node();
        new_v->left = v;
        v = new_v;
    }

    return v->left;
}
```

- $node * mult()$ – создает вершины дерева, отвечающие синтаксической структуре < Множитель >.

```
node * mult() {
    if (LEXEMS[IND].c == 'n') {
        node * v = new node('n', LEXEMS[IND].num);
        ++IND;
        return v;
    } else if (LEXEMS[IND].c == '(') {
        ++IND;
        node * v = expression();
        if (LEXEMS[IND].c == ')') {
            ++IND;
            return v;
        } else {
            WRONG = 1;
            return NULL;
        }
    } else {
        WRONG = 1;
        return NULL;
    }
}
```

Таким образом, мы можем получить указатель на построенное дерево выражения так:

$$get\_lexem(s);$$
$$node * root = expression();$$

Если после выполнения этих двух строк кода глобальная переменная $WRONG = 1$ – введенное выражение некорректно, и корректное бинарное дерево по нему построить нельзя.

Иначе, можно вызывать дополнительные функции:

- $void\ print\_tree(node\ *\ v, int\ h)$ – вывод дерево в $cout$.
- $int\ ans(node\ *\ v, int\ \&mul\_cnt)$ – подсчет значения выражения по его бинарному дереву и подсчет количеств операций $*$ в выражении.

```
void print_tree(node * v, int h) {
    if (v == NULL) {
        return;
    }
    print_tree(v->left, h + 1);

    for (int i = 0; i < h; ++i) {
        cout << "    ";
    }
    print_node(v);
    cout << "\n";

    print_tree(v->right, h + 1);
}
```

```
int ans(node * v, int &mul_cnt) {
    if (v == NULL) {
        return 0;
    }
    int a1 = ans(v->left, mul_cnt);
    int a2 = ans(v->right, mul_cnt);
    if (v->type == 'n') {
        return v->val;
    } else {
        if (v->type == '*') {
            mul_cnt++;
            return a1 * a2;
        } else if (v->type == '+') {
            return a1 + a2;
        } else {
            return a1 - a2;
        }
    }
}
```

Для интерактивного ввода пользователем выражений и вывода результатов работы всех требуемых функций запишем $main()$ таким образом:

```
int main() {

    while (1) {
        WRONG = 0;
        LEXEMS.clear();
        IND = 0;

        cout << "Enter the expression:\n";
        string s;
        cin >> s;

        get_lexem(s);
        if (WRONG) {
            cout << "Input error\n";
            cout << "_____________________________________________________________\n";
            continue;
        }
        node * root = expression();
        if (WRONG) {
            cout << "Input error\n";
            cout << "_____________________________________________________________\n";
            continue;
        }
        int mul_cnt = 0;
        cout << "\nThe value of expression: " << ans(root, mul_cnt) << "\n";
        cout << "\nThe expression contains " << mul_cnt << " multiplication signs\n";
        cout << "\nBinary tree of the expression:\n";
        print_tree(root, 0);
        cout << "\n";
        cout << "_____________________________________________________________\n";
    }

    return 0;
}
```

Код приложения

```
#include <iostream>
#include <string>
#include <vector>
#include <cctype>

using namespace std;

struct node {
	char type;
	int val;
	node * left;
	node * right;

	node(char _type = 'n', int _val = 0) {
		type = _type;
		val = _val;
```

```
		left = NULL;
		right = NULL;
	}
};

struct lexem {
	char c;
	int num;

	lexem(char _c = 'n', int _num = 0) {
		c = _c;
		num = _num;
	}
};

vector <lexem> LEXEMS;
int IND = 0;
bool WRONG = 0;

void skip_space(int &i, string &s) {
	while (i < s.size() && isspace(s[i])) {
		++i;
	}
}

bool is_operation(char c) {
	return (c == '+') || (c == '-') || (c == '*') || (c == '(') || (c == ')');
}

bool is_digit(char c) {
	return '0' <= c && c <= '9';
}

void get_lexem(string &s) {
	int i = 0;
	while (1) {
		skip_space(i, s);
		if (i >= s.size()) {
			break;
		}
		if (is_digit(s[i])) {
			int x = 0;
			while (i < s.size() && is_digit(s[i])) {
				x *= 10;
				x += (s[i] - '0');
				++i;
			}
			LEXEMS.push_back(lexem('n', x));
		} else if (is_operation(s[i])) {
			LEXEMS.push_back(lexem(s[i]));
			++i;
		} else {
			WRONG = 1;
		}
	}
}

node * item();
node * mult();

node * expression() {
	node * v = new node();
	v->left = item();
```

```
	while (!WRONG && IND < LEXEMS.size() && (LEXEMS[IND].c == '+' || LEXEMS[IND].c ==
'-')) {
		v->type = LEXEMS[IND].c;
		++IND;
		v->right = item();

		node * new_v = new node();
		new_v->left = v;
		v = new_v;
	}

	return v->left;
}

node * item() {
	node * v = new node();
	v->left = mult();

	while (!WRONG && IND < LEXEMS.size() && LEXEMS[IND].c == '*') {
		v->type = LEXEMS[IND].c;
		++IND;
		v->right = mult();

		node * new_v = new node();
		new_v->left = v;
		v = new_v;
	}

	return v->left;
}

node * mult() {
	if (LEXEMS[IND].c == 'n') {
		node * v = new node('n', LEXEMS[IND].num);
		++IND;
		return v;
	} else if (LEXEMS[IND].c == '(') {
		++IND;
		node * v = expression();
		if (LEXEMS[IND].c == ')') {
			++IND;
			return v;
		} else {
			WRONG = 1;
			return NULL;
		}
	} else {
		WRONG = 1;
		return NULL;
	}
}

void print_node(node * v) {
	if (v == NULL) {
		return;
	}
	if (v->type == 'n') {
		cout << " " << v->val << " ";
	} else {
		cout << "[" << v->type << "] ";
	}
}


void print_tree(node * v, int h) {
```

```
	if (v == NULL) {
		return;
	}
	print_tree(v->left, h + 1);

	for (int i = 0; i < h; ++i) {
		cout << "    ";
	}
	print_node(v);
	cout << "\n";

	print_tree(v->right, h + 1);
}

int ans(node * v, int &mul_cnt) {
	if (v == NULL) {
		return 0;
	}
	int a1 = ans(v->left, mul_cnt);
	int a2 = ans(v->right, mul_cnt);
	if (v->type == 'n') {
		return v->val;
	} else {
		if (v->type == '*') {
			mul_cnt++;
			return a1 * a2;
		} else if (v->type == '+') {
			return a1 + a2;
		} else {
			return a1 - a2;
		}
	}
}

int main() {

	while (1) {
		WRONG = 0;
		LEXEMS.clear();
		IND = 0;

		cout << "Enter the expression:\n";
		string s;
	    cin >> s;

	    get_lexem(s);
	    if (WRONG) {
		cout << "Input error\n";
		cout << "_____________________________________________________________\n";
		continue;
	    }
	    node * root = expression();
	    if (WRONG) {
		cout << "Input error\n";
		cout << "_____________________________________________________________\n";
		continue;
	    }
	    int mul_cnt = 0;
	    cout << "\nThe value of expression: " << ans(root, mul_cnt) << "\n";
	    cout << "\nThe expression contains " << mul_cnt << " multiplication signs\n";
	    cout << "\nBinary tree of the expression:\n";
	    print_tree(root, 0);
	    cout << "\n";
	    cout << "_____________________________________________________________\n";
	}
```

```
    return 0;
}
```

Тесты

Тест 1

Выражение: ((4+16*(7+3))*9)

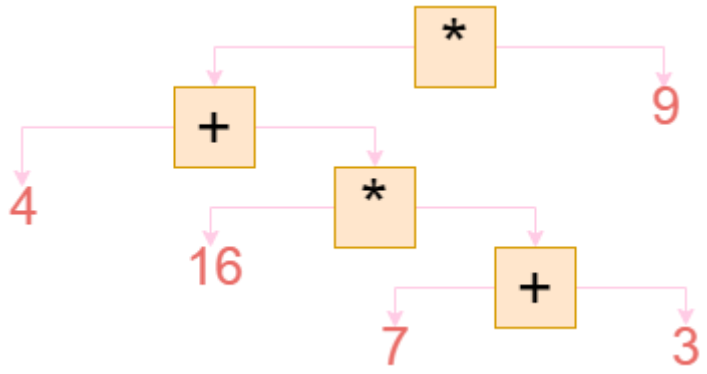


Рис.1. Графическая реализация бинарного дерева

```
Enter the expression:
((4+16*(7+3))*9)

The value of expression: 1476

The expression contains 2 multiplication signs

Binary tree of the expression:
        4
    [+]
            16
        [*]
                7
            [+]
                    3
[*]
    9
```

Рис.2. Построение бинарного дерева в ходе тестирования программы

Тест 2

Выражение: (2+3)*5-2*4

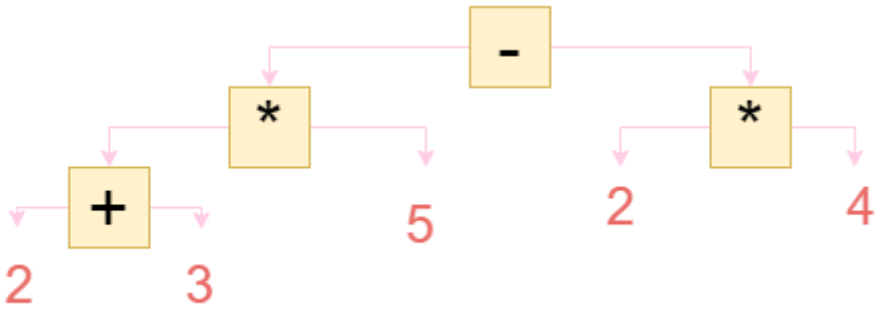


Рис.3. Графическая реализация бинарного дерева

```
Enter the expression:
(2+3)*5-2*4

The value of expression: 17

The expression contains 2 multiplication signs

Binary tree of the expression:
          2
      [+]
          3
   [*]
       5
[-]
       2
   [*]
       4
```

Рис.4. Построение бинарного дерева в ходе тестирования программы

Тест 3

Выражение: (5*9) + (3*3) + (9*9)

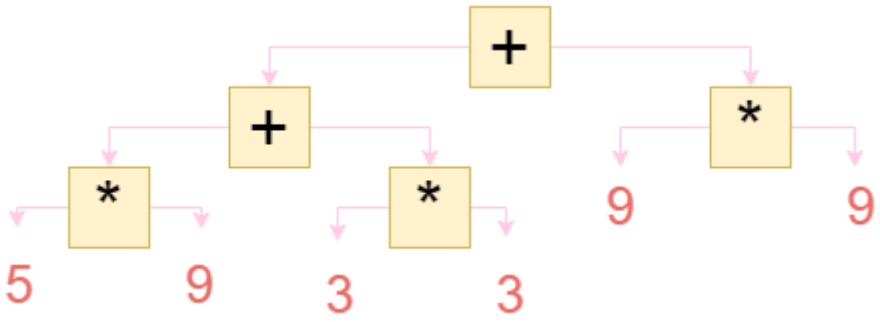


Рис.5. Графическая реализация бинарного дерева

```
Enter the expression:
(5*9)+(3*3)+(9*9)

The value of expression: 135

The expression contains 3 multiplication signs

Binary tree of the expression:
          5
      [*]
          9
   [+]
          3
      [*]
          3
[+]
       9
   [*]
       9
```

Рис.6. Построение бинарного дерева в ходе тестирования программы

Выводы

В результате проделанной работы, я получила навыки по разработке и реализации операций над структурой данных – бинарное дерево. В процессе было разработана программа, которая создает дерево выражений и выполняет операции.

Список информационных источников

1. Лекционный материал по структурам и алгоритмам обработки данных Сартакова М. В. (дата обращения 27.09.2021)
2. Дополнительный материал к практическим работам по структурам и алгоритмам обработки данных Сорокина А. В. (дата обращения 27.09.2021)

Ответы на вопросы

1. Что определяет степень дерева?

   Степень дерева определяет максимальную степень его узлов.

2. Какова степень сильноветвящегося дерева?

   Степень сильноветвящегося дерева больше 2.

3. Что определяет путь в дереве?

   Путь в дереве определяет последовательность узлов от корня до нужного узла.

4. Как рассчитать длину пути в дереве?

   Чтобы рассчитать длину пути в дереве нужно посчитать сумму длин его ребер. (Длина пути дерева определяется как сумма длин путей ко всем его вершинам.)

5. Какова степень бинарного дерева?

   Степень бинарного дерева равна 2.

6. Может ли дерево быть пустым?

   Дерево называется пустым, если оно не содержит ни одной вершины.

7. Дайте определение бинарного дерева?

   Бинарное дерево - это дерево, у каждого узла которого не более 2 потомков.

8. Дайте определение алгоритму обхода.

   Обход дерева – это упорядоченная последовательность вершин дерева, в которой каждая вершина встречается только один раз. Над данными узла можно выполнять операции. Обход формирует список пройденных узлов.

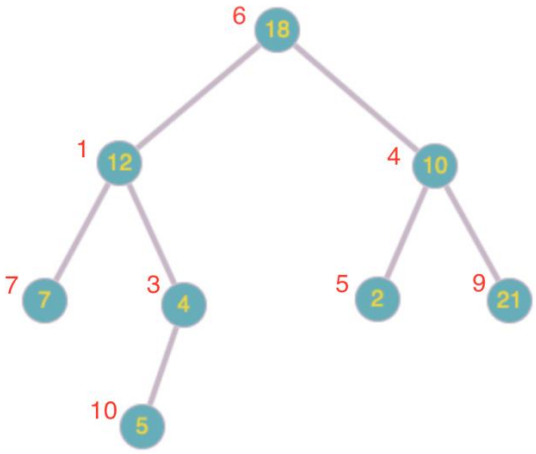
9. Приведите рекуррентную зависимость для вычисления высоты дерева.

$$h(T) = \begin{cases} -1, & \text{если } T = NULL \\ 1 + max(h(T.left), h(T.right)), & \text{иначе} \end{cases}$$

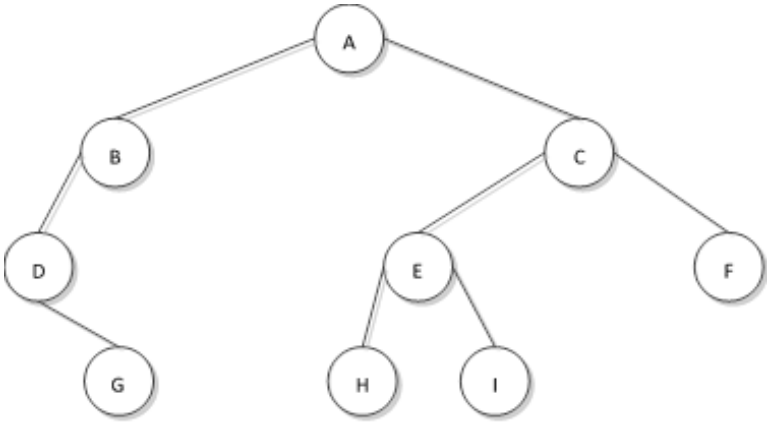
10. Изобразите бинарное дерево, корень которого имеет индекс 6, и которое представлено в памяти таблицей вида:

Индекс	**Key**	**left**	**right**
1	12	7	3
2	15	8	NULL
3	4	10	NULL
4	10	5	9
5	2	NULL	NULL
6	18	1	4
7	7	NULL	NULL

8	14	6	2
9	21	NULL	NULL
10	5	NULL	NULL



11. Укажите путь обхода дерева по алгоритму: прямой; обратный; симметричный



Путь обхода дерева в прямом порядке: ABDGCEHIF.

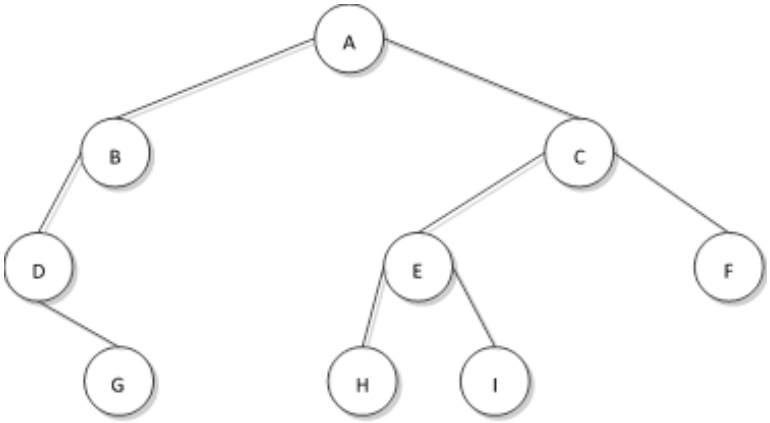
Путь обхода дерева в обратном порядке: GDBHIEFCA.

Путь обхода дерева в симметричном порядке: DGBAHEICF.

12. Какая структура используется в алгоритме обхода дерева методом в «ширину»?

В алгоритме обхода дерева методом в «ширину» используется очередь.

13. Выведите путь при обходе дерева в «ширину». Продемонстрируйте использование структуры при обходе дерева.



Путь обхода дерева в «ширину»: ABCDEFGHI.

Шаг 1. Очередь: A.

Шаг 2. Очередь: BC. Вывод: A.

Шаг 3. Очередь: CD. Вывод: B.

Шаг 4. Очередь: DEF. Вывод: C.

Шаг 5. Очередь: EFG. Вывод: D.

Шаг 6. Очередь: FGHI. Вывод: E.

Шаг 7. Очередь: GHI. Вывод: F.

Шаг 8. Очередь: HI. Вывод: G.

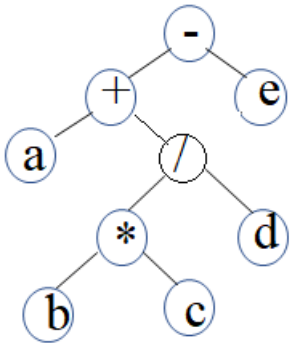
Шаг 9. Очередь: I. Вывод: H.

Шаг 10. Очередь: пусто. Вывод: I.

14. Какая структура используется в не рекурсивном обходе дерева методом в «глубину»?

В не рекурсивном обходе дерева методом в «глубину» используется стек.

15. Выполните прямой, симметричный, обратный методы обхода дерева выражений.



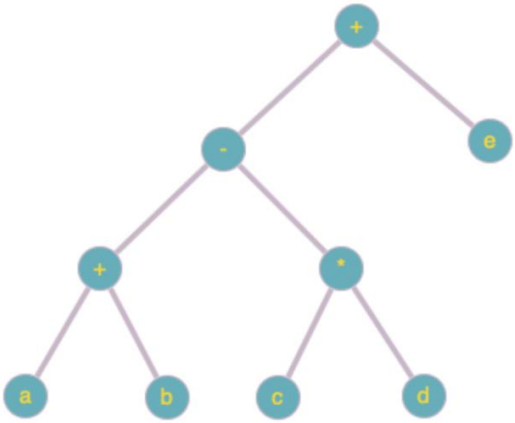
Путь обхода дерева в прямом порядке: -+a/*bcde.

Путь обхода дерева в симметричном порядке: a+b*c/d-e.

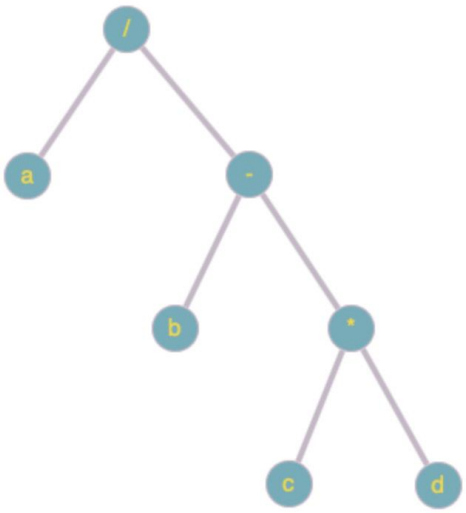
Путь обхода дерева в обратном порядке: abc*d/+e-.

16. Для каждого заданного арифметического выражения постройте бинарное дерево выражений:

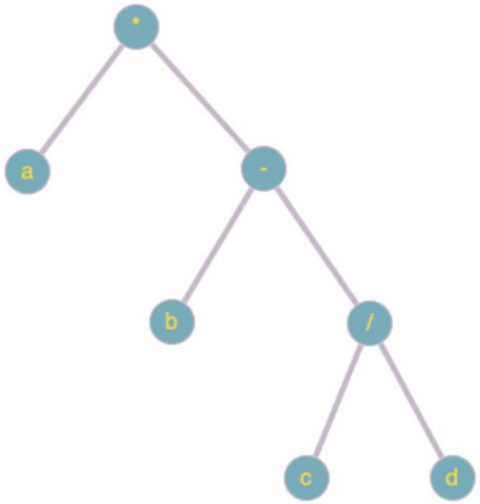
1. a+b-c*d+e



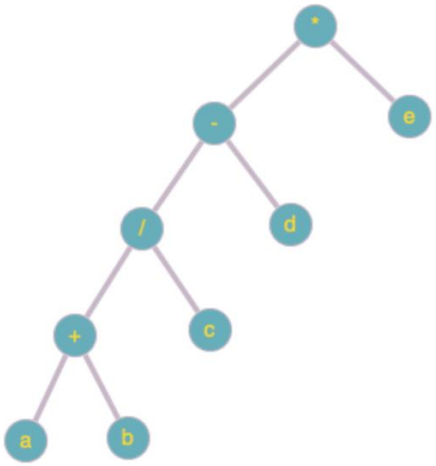
2. /a-b*c d



3. a b c d/ - *



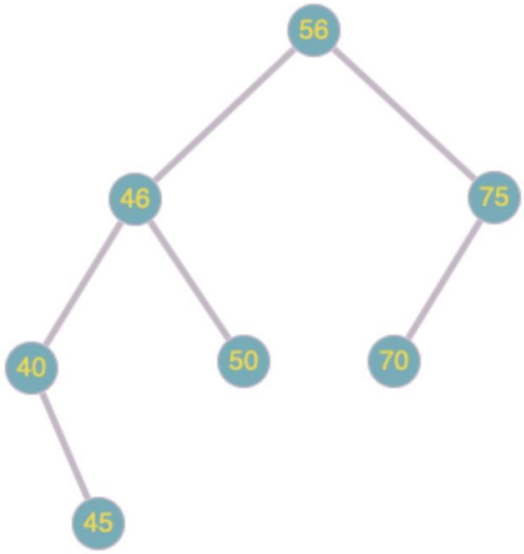
4. *-/+abcde



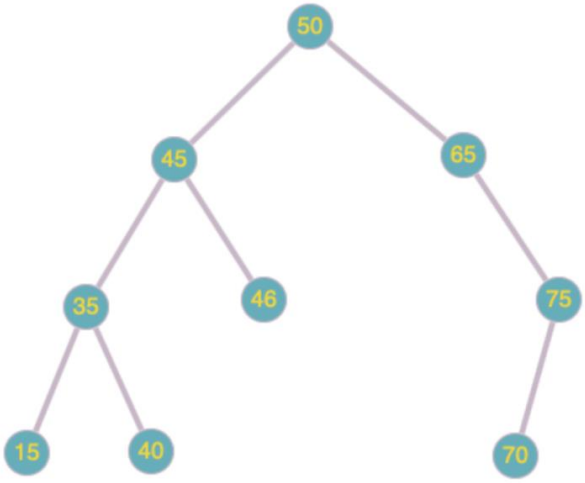
17. В каком порядке будет проходиться бинарное дерево, если алгоритм обхода в ширину будет запоминать узлы не в очереди, а в стеке?

    Если алгоритм обхода в ширину будет запоминать узлы в стеке, то бинарное дерево будет проходиться в прямом порядке.

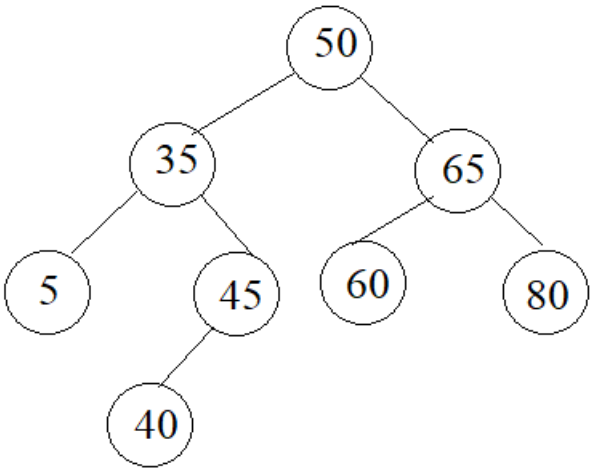
18. Постройте бинарное дерево поиска, которое в результате симметричного обхода дало бы следующую последовательность узлов: 40 45 46 50 65 70 75.



19.Последовательность {50 45 35 15 40 46 65 75 70} получена путем прямого обхода бинарного дерева поиска. Постройте это дерево.

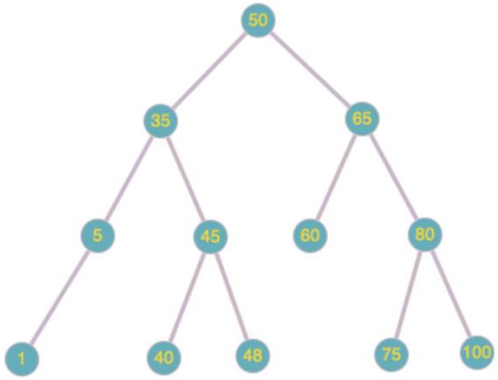


20. Дано бинарное дерево поиска.

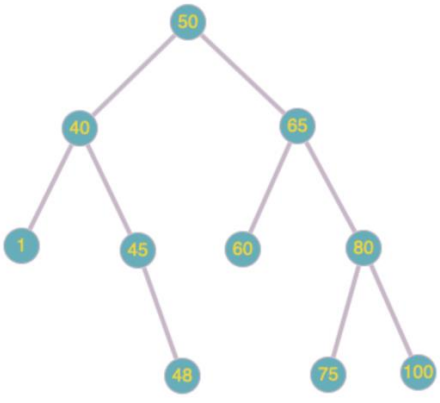


Выполните действия над исходным деревом и покажите дерево:

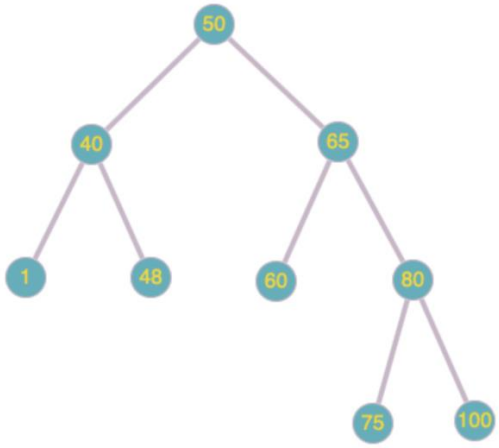
1) после включения узлов 1, 48, 75, 100



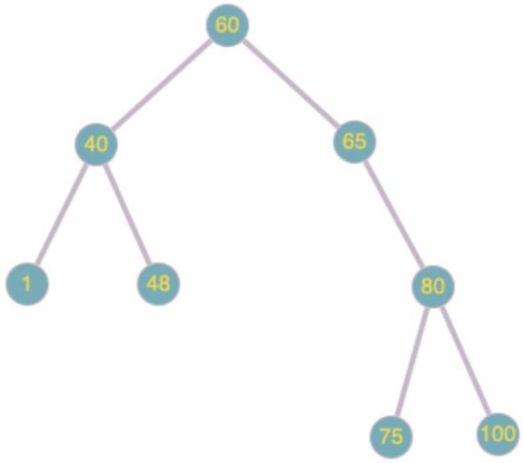
2) после удаления узлов 5, 35



3) после удаления узла 45



4) после удаления узла 50



5) после удаления узла 65 и вставки его снова