Статья опубликована в журнале *Компьютерные инструменты в образовании*, 2004, № 3, с 32-37.

Г. А. Корнеев, Н. Н. Шамгунов, А. А. Шалыто
Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Обход деревьев на основе автоматного подхода

На основе автоматного подхода предложены алгоритмы обхода деревьев, отличающиеся от классических наглядностью и универсальностью.

Введение

С 1991 г. в России развивается SWITCH-технология, которая базируется на автоматном программировании [1]. Настоящая работа проведена в рамках исследования применения SWITCH-технологии в вычислительных алгоритмах [2, 3], и посвящена одному из них — обходу деревьев. Первоначально в работе рассматривается обход двоичных деревьев, а затем предлагаемое решение обобщается на случай *k*-ичных деревьев. При этом рассматриваются алгоритмы обхода, как использованием стека, так и без его применения. Использование автоматного подхода позволяет получить наглядные и универсальные (в отличие от работы [4]) алгоритмы решения рассматриваемых задач, которые также весьма экономны по памяти по сравнению с классическими алгоритмами.

Отметим, что в работе [5] рассмотрен обход деревьев с произвольным количеством потомков. Для реализации обхода в этой работе вводится понятие “состояние”, но автомат в явном виде не строится.

1. Постановка задачи обхода двоичного дерева

Пусть задано двоичное дерево, например, представленное на рис. 1.

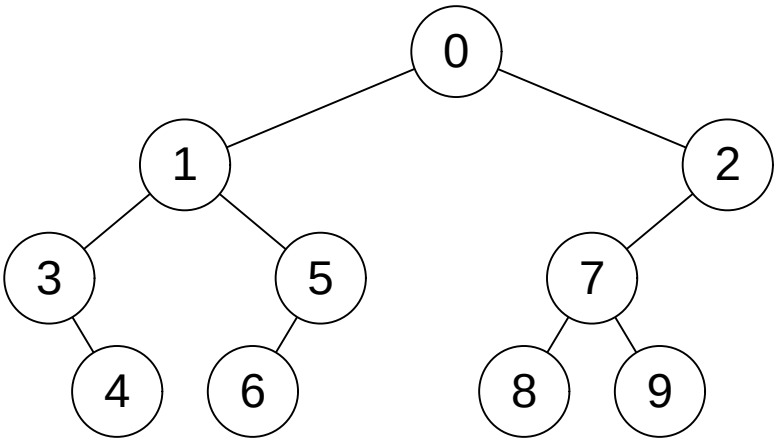

Рис. 1. Двоичное дерево

Необходимо осуществить обход дерева — сформировать последовательность номеров его вершин. Рассмотрим три порядка обхода дерева: прямой (preorder), обратный (postorder) и центрированный (inorder) [4, 6]. Такие порядки используются при обходе дерева в глубину. В качестве примера приведем последовательности, порождаемые обходами дерева, представленного на рис. 1:

- прямой обход: 0, 1, 3, 4, 5, 6, 2, 7, 8, 9;
- центрированный обход: 3, 4, 1, 6, 5, 0, 8, 7, 9, 2;
- обратный обход: 4, 3, 6, 5, 1, 8, 9, 7, 2, 0.

Данная задача может быть решена несколькими способами: рекурсивно [4, 6], итеративно с применением стека и итеративно без использования стека [7].

Ни в одном из этих подходов не используются автоматы, что, по мнению авторов, не позволило обеспечить наглядность и универсальность предлагаемых решений.

1.1. Описание структур данных для представления двоичных деревьев

Будем представлять бинарное дерево в программе следующим образом:

```
struct Node {
    int data;    // Данные в узле
    Node* l;     // Левое поддерево
    Node* r;     // Правое поддерево
    Node* p;     // Указатель на родителя
};
```

В этой структуре содержатся ссылки на правое и левое поддеревья, а также ссылка на родительскую вершину, что в дальнейшем, в частности, позволит реализовать рассматриваемый алгоритм без стека. В приводимых примерах на языке C++ для хранения указателей на вершины дерева будем использовать переменные типа `Node*`. При этом указатель на вершину объявляется как `Node* node`. Доступ к переменным, указывающим на левое и правое поддеревья, а также на родительскую вершину осуществляется при помощи оператора `->`. Поэтому в графах переходов в условиях, помечающих дуги, будем применять следующие обозначение языка C++: `node->l`, `node->r` и `node->p` соответственно.

1.2. Ввод деревьев

Ввод деревьев в примерах будем производить в следующем виде. В первое строке записано число N (количество вершин в дереве). Последующие N строк содержат описания вершин дерева. Для i-ой вершины в строке с номером $i+2$ (вершины нумеруются с нуля) указываются номера корней левого и правого поддеревьев. В случае отсутствия поддерева вместо номера корня записывается число `-1`. Для рассматриваемого примера исходные данные имеют следующий вид:

```
10
 1  2
 3  5
 7 -1
-1  4
-1 -1
 6 -1
-1 -1
 8  9
-1 -1
-1 -1
```

2. Обход двоичного дерева без использования стека

Рассмотрим обход двоичного дерева без применения стека. Идея предлагаемого алгоритма состоит в том, что при обходе двоичного дерева могут быть выделены следующие направления движения: влево, вправо, вверх. Обход начинается от корня дерева. В зависимости от номера текущей вершины и направления движения, можно принять решение

о том, куда двигаться дальше. При этом стек не требуется — как уже отмечалось выше, используется указатель на родительскую вершину.

Каждому из указанных направлений может быть сопоставлено состояние автомата, реализующего алгоритм. Также вводится начальное состояние, одновременно являющееся конечным.

Таким образом, автомат имеет следующие состояния:

0. *Start* — начальное (конечное) состояние.
1. *Left* — движение влево.
2. *Right* — движение вправо.
3. *Parent* — движение вверх.

Автомат оперирует переменной `node` (имеющей тип `Node*`), являющейся указателем на текущий узел дерева.

Поясним, как строится граф переходов, описывающий поведение автомата. Переходы между состояниями определяются условиями, которыми помечаются соответствующие дуги графа переходов. Условия `node->l`, `node->r` и `node->p` обозначают наличие у текущего узла левого потомка, правого потомка и родителя соответственно. Отрицание будем обозначать восклицательным знаком (`!`). Например, переход из состояния *Parent* в состояние *Start* осуществляется при условии, что у вершины нет родителя.

Некоторые дуги графа помечены не только условиями (расположены в числителе дроби), но и действиями (расположены в знаменателе дроби). Например, переход из состояния *Right* в состояние *Left* производится при условии, что у вершины есть правое поддерево. При этом на указанном переходе выполняется действие — изменяется переменная `node`, в которой хранится указатель на текущую вершину.

Кроме действий на переходах, в автомате также выполняются действия в состояниях, например, в состоянии *Left* производится вывод номера текущей вершины на экран.

Граф переходом для реализации нерекурсивного обхода двоичного дерева без использования стека приведен на (рис. 2).

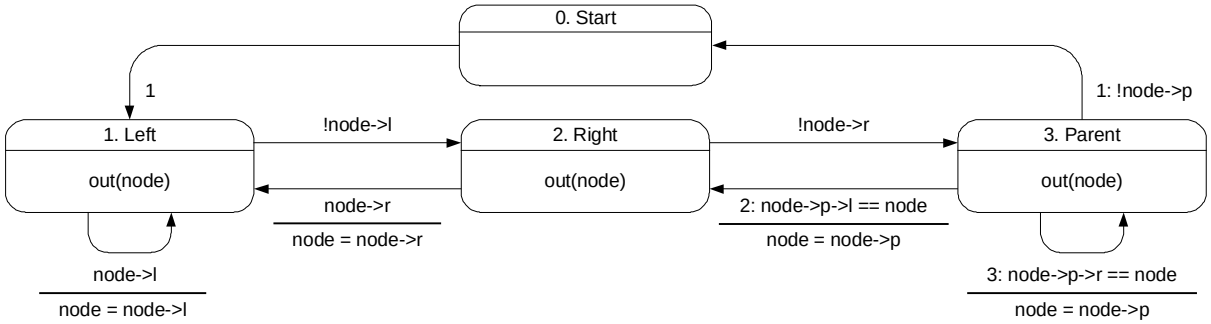

Рис. 2. Граф переходов автомата для реализации нерекурсивного обхода двоичного дерева

Отметим, что в случае безусловного перехода из одного состояния в другое дуга помечается символом 1 (переход из нулевой вершины в первую). Еще одной особенностью этого графа является пометка дуг «числами с двоеточием» — приоритетами. Эти пометки указывают последовательность, в которой будут проверяться условия на дугах, исходящих из вершины. В случае если условия являются попарно ортогональными, то приоритеты не расставляются.

Отметим, что если в работах [4, 6] рассмотрено три типа обходов и для каждого из них предложен свой алгоритм, то предлагаемый подход позволяет, используя один и тот же граф переходов автомата, осуществлять все три обхода, за счет вывода номера текущей вершины только в одном из состояний: *Left*, *Right*, *Parent*. При этом в зависимости от выбранного состояния получим три стандартных обхода [4, 6]:

Состояние	Обход
Left	Прямой
Right	Центрированный
Parent	Обратный

Таким образом, предложенный автомат представляет стандартные виды обхода в единой форме.

В Приложении (Листинг 1, функция `traverseWithoutStack`) приведена реализация этого автомата при помощи конструкции `switch`, построенная по графу переходов формально и изоморфно.

3. Обход двоичного дерева с использованием стека

Рассмотрим представление дерева, узлы которого не содержат ссылок на родителей. В этом случае структура `Node` выглядит следующим образом:

```
struct Node {
    int data;    // Данные в узле
    Node * l;    // Левое поддерево
    Node * r;    // Правое поддерево
};
```

Для обходов деревьев, представленных таким образом, необходимо использовать стек. По аналогии с предыдущим разделом для решения данной задачи может быть построен автомат, граф переходов которого представлен на рис. 3.

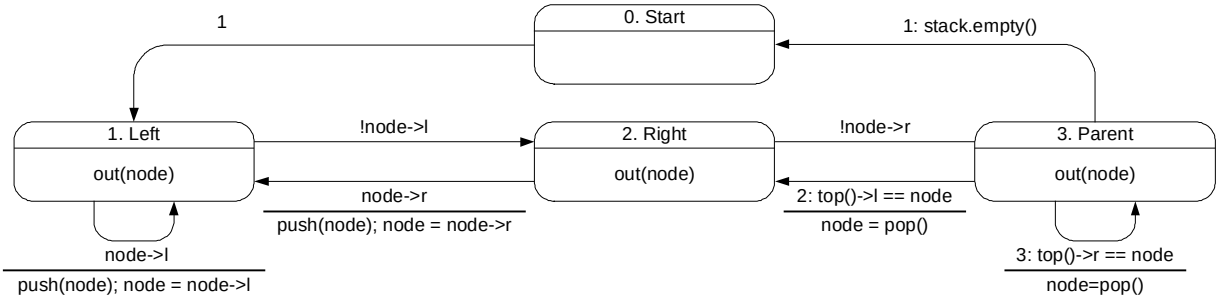

Рис. 3. Граф переходов автомата для реализации обхода двоичного дерева с использованием стека

На этом рисунке используются следующие обозначения: `push(node)` — помещение текущего узла в стек, `top()` — узел в вершине стека, `pop()` — функция удаления узла из стека, возвращающая удаленную вершину, `empty()` — проверка стека на пустоту.

Заметим, что при всех обходах дерева в стек помещается одна и та же информация. Таким образом, один стек может быть использован для реализации всех трех обходов, что позволяет экономить память.

Если вывод в состояниях *Left, Right* и *Parent* осуществлять в разные потоки, то можно получить одновременный вывод любой комбинации рассмотренных обходов.

В Приложении (Листинг 1, функция `traverseWithStack`) приведена реализация этого автомата при помощи конструкции `switch`, построенная по графу переходов формально и изоморфно.

Граф переходов позволяет легко понять поведение программы, а также является тестом для ее проверки. Это объясняется тем, что состояния декомпозируют входные воздействия на группы, каждая из которых содержит условия, определяющая переходы только из этого состояния.

4. Обход *k*-ичного дерева без использования стека

Изложенный подход может быть обобщен на случай *k*-ичных деревьев [6].

В программе дерево будем представлять в следующем виде:

```
struct KNode {
    int data;
    KNode * c[k];
};
```

где *k* — константа, *c* — массив, хранящий указатели на детей. Метод, предложенный для обхода двоичных деревьев без использования стека, можно обобщить для обхода *k*-ичных деревьев (рис. 4).

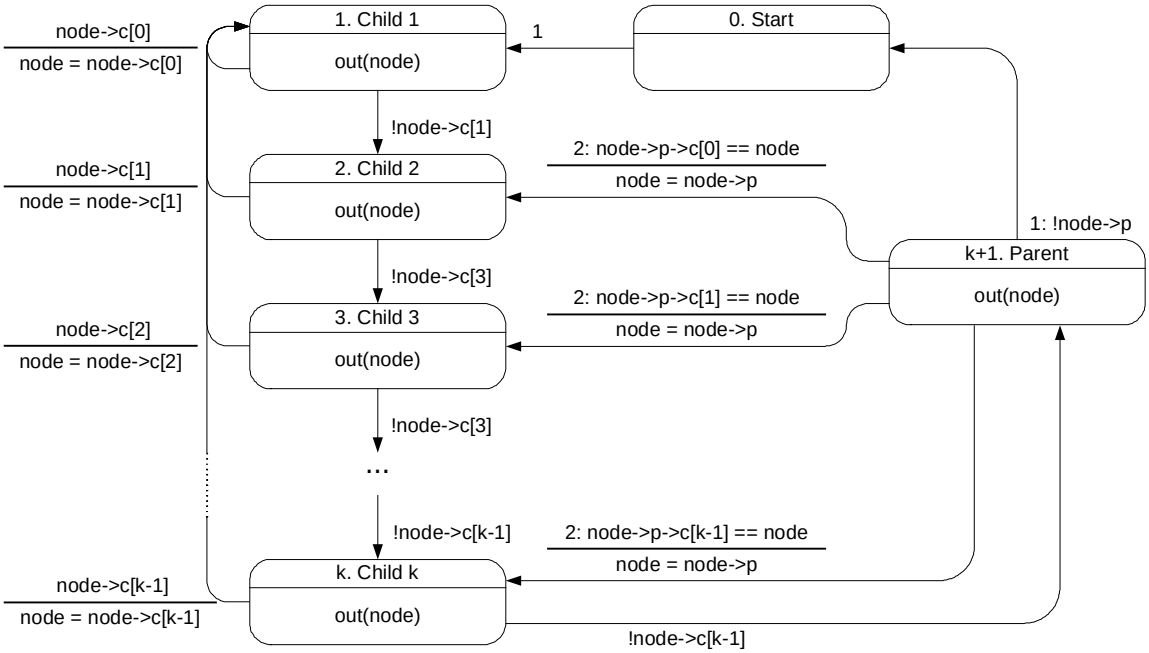

Рис. 4. Граф переходов автомата для реализации обхода *k*-ичного дерева без использования стека

Отметим, что у построенного автомата *k+3* состояний. Таким образом, число состояний автомата зависит от *k*.

В Приложении (Листинг 2, функция `traverseK`) приведена реализация этого автомата при помощи конструкции `switch`, построенная по графу переходов формально и изоморфно.

Как отмечено выше, граф переходов построенного автомата зависит от числа *k*. Таким образом, такая реализация обхода *k*-ичного дерева не является универсальной.

Предложим **универсальное** решение. Из графа переходов следует, что условия перехода и действия в состояниях с первого по *k*-ое очень похожи. Поэтому логично объединить эти состояния в одно. В результате получается редуцированный автомат с четырьмя состояниями (рис. 5), который позволяет произвести обход *k*-ичного дерева при произвольном значении *k*.

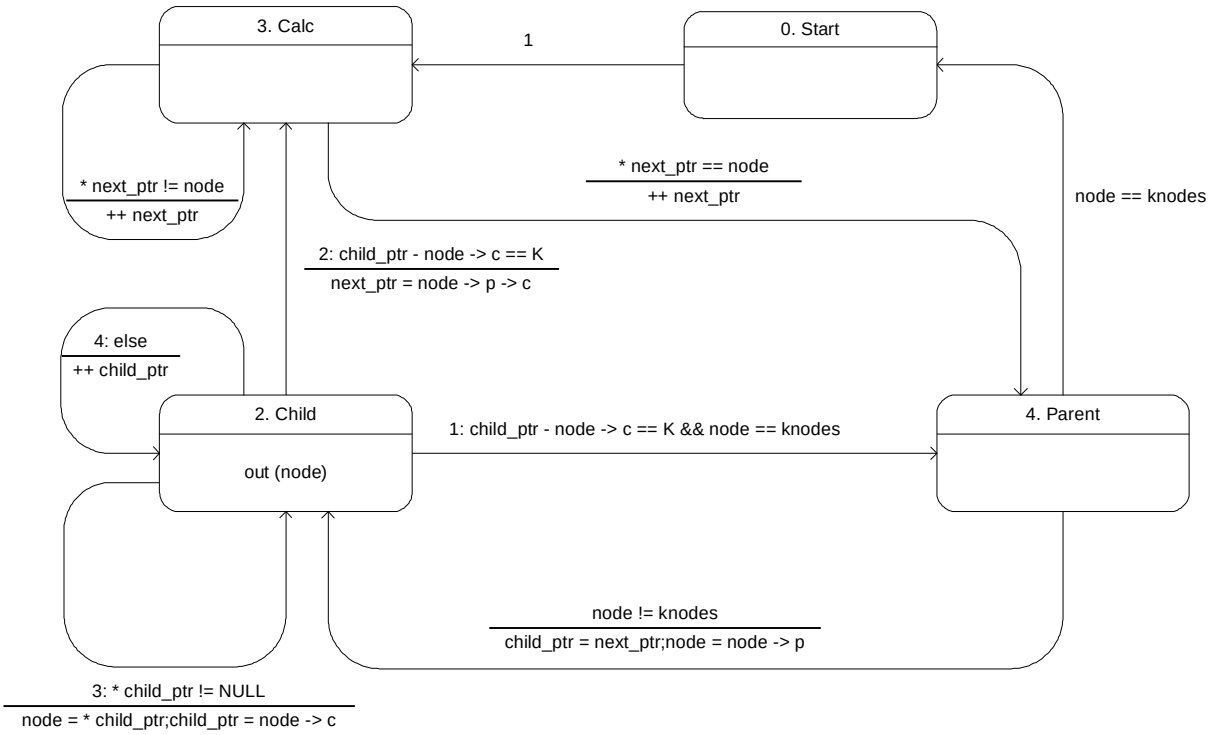

Рис. 5. Редуцированный граф переходов автомата для реализации обхода *k*-ичного дерева без использования стека

В Приложении (Листинг 2, функция `traverseKReduced`) приведена реализация этого автомата при помощи конструкции `switch`, построенная по графу переходов формально и изоморфно.

Заключение

Таким образом, в настоящей работе показано, что использование автоматного подхода позволило получить наглядные и, в отличие от классических, универсальные алгоритмы решения задач обхода деревьев, которые весьма экономны по памяти.

Литература

1. *Шалыто А. А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.
2. *Туккель Н. И., Шамгунов Н. Н., Шалыто А. А.* Ханойские башни и автоматы //Программист. 2002 № 8. http://is.ifmo.ru, раздел «Статьи».
3. *Туккель Н. И., Шамгунов Н. Н, Шалыто А. А.* Задача о ходе коня //Мир ПК 2003. №1. http://is.ifmo.ru, раздел «Статьи».
4. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: Центр непрер.матем. образования, 2000.
5. *Шень А.* Программирование. Теоремы и задачи. М.: Центр непрер.матем. образования, 2004.
6. *Кнут Д.* Искусство программирования. Т. 1. Основные алгоритмы. М:.: Вильямс, 2003.
7. *Касьянов В. Н., Евстигнеев В. А.* Графы в программировании: обработка, визуализация и применение. СПб.: БХВ-Петербург, 2003.

Об авторах

Корнеев Георгий Александрович, магистрант кафедры «Компьютерные технологии» Санкт-Петербургского государственного университета информационных технологий, механики и оптики (СПбГУ ИТМО). Призер чемпионатов мира по программированию ACM

Шамгунов Никита Назимович, аспирант кафедры «Компьютерные технологии» СПбГУ ИТМО. Призер чемпионата мира по программированию ACM.

Шалыто Анатолий Абрамович, доктор технических наук, профессор, заведующий кафедрой «Технологии программирования» СПбГУ ИТМО.

Приложение

Листинг 1. Обходы двоичных деревьев

```
#include <iostream>
#include <stack>
using namespace std;

/* Представление узлов дерева */
struct Node {
    int data;    // Данные в узле
    Node* l;     // Левое поддерево
    Node* r;     // Правое поддерево
    Node* p;     // Указатель на родителя
};

const SHOW_LEFT   = 1;
const SHOW_RIGHT  = 2;
const SHOW_PARENT = 4;

/* Состояния автомата */
enum State {START, LEFT, RIGHT, PARENT};



/*
 * Обход двоичного дерева без использования стека
 *
 * Параметр show определяет разновидность обхода:
 *     прямой (show == SHOW_LEFT),
 *     центрированный (show == SHOW_RIGHT),
 *     обратный (show == SHOW_PARENT)
 */

void traverseWithoutStack(Node const* node, int show) {
    State state = START;
    do {
        switch (state) {
            case START:
                state = LEFT;
                break;
            case LEFT:
                if (show & SHOW_LEFT) cout << node->data << " ";
                if (node->l) {
```

```
                    node = node->l;
                    state = LEFT;
                } else {
                    state = RIGHT;
                }
                break;
            case RIGHT:
                if (show & SHOW_RIGHT) cout << node->data << " ";
                if (node->r) {
                    node = node->r;
                    state = LEFT;
                } else {
                    state = PARENT;
                }
                break;
            case PARENT:
                if (show & SHOW_PARENT) cout << node->data << " ";
                if (node->p) {
                    if (node->p->l == node) {
                        node = node->p;
                        state = RIGHT;
                    } else {
                        node = node->p;
                        state = PARENT;
                    }
                } else {
                    state = START;
                }
                break;
        }
    } while(state != START);
}

/*
 * Обход двоичного дерева с использованием стека
 *
 * Параметр show определяет разновидность обхода:
 *     прямой (show == SHOW_LEFT),
 *     центрированный (show == SHOW_RIGHT),
 *     обратный (show == SHOW_PARENT)
 */
void traverseWithStack(Node const* node, int show) {
    std::stack<Node const*> stack;

    State state = START;
    do {
        switch (state) {
            case START:
                state = LEFT;
                break;
            case LEFT:
                if (show & SHOW_LEFT) cout << node->data <<  " ";
                if (node->l) {
```

```
                    stack.push(node);
                    node = node->l;
                    state = LEFT;
                } else {
                    state = RIGHT;
                }
                break;
            case RIGHT:
                if (show & SHOW_RIGHT) cout << node->data << " ";
                if (node->r) {
                    stack.push(node);
                    node = node->r;
                    state = LEFT;
                } else {
                    state = PARENT;
                }
                break;
            case PARENT:
                if (show & SHOW_PARENT) cout << node->data << " ";
                if (stack.empty()) {
                    state = START;
                } else if (stack.top()->l == node) {
                    node = stack.top(); stack.pop();
                    state = RIGHT;
                } else if (stack.top()->r == node) {
                    node = stack.top(); stack.pop();
                    state = PARENT;
                }
                break;
        }
    } while (state != START);
}


int cNodes;     // Количество узлов
Node* nodes;    // Узлы


/*
 * Чтение входных данных
 */

void readInput() {
    cin >> cNodes;
    nodes = new Node[cNodes];

    for (int i = 0; i < cNodes; i++) {
        nodes[i].data = i;
        nodes[i].p = NULL;
        nodes[i].l = NULL;
        nodes[i].r = NULL;
    }
```

```
    for (int i = 0; i < cNodes; i++) {
        int l, r;
        cin >> l >> r;
        if (l != -1) {
            nodes[i].l = nodes + l;
            nodes[l].p = nodes + i;
        }
        if (r != -1) {
            nodes[i].r = nodes + r;
            nodes[r].p = nodes + i;
        }
    }
}

void main() {
    readInput();

    cout << "Preorder Without Stack" << endl;
    traverseWithoutStack(nodes, SHOW_LEFT);
    cout << endl;

    cout << "Preorder With Stack" << endl;
    traverseWithStack(nodes, SHOW_LEFT);
    cout << endl;

    cout << "Inorder Without Stack" << endl;
    traverseWithoutStack(nodes, SHOW_RIGHT);
    cout << endl;

    cout << "Inorder With Stack" << endl;
    traverseWithStack(nodes, SHOW_RIGHT);
    cout << endl;

    cout << "Postorder Without Stack" << endl;
    traverseWithoutStack(nodes, SHOW_PARENT);
    cout << endl;

    cout << "Postorder With Stack" << endl;
    traverseWithStack(nodes, SHOW_PARENT);
    cout << endl;

    delete[] nodes;
}
```

Листинг 2. Обходы к-ичных деревьев без использования стека

```
/*
 * Обход к-ичного дерева без использования стека
 */

void traverseK ()
{
     enum State
```

```
{
     Start,
     Child1,
     Child2,
     Child3,
     Parent
};

State state = Start;
KNode * node = knodes;
do
{
     switch ( state )
     {
     case Start :
          state = Child1;
          break;
     case Child1 :
          cout << node -> data << ' ';
          if ( node -> c [ 0 ] != NULL )
               node = node -> c [ 0 ];
          else
               state = Child2;
          break;
     case Child2 :
          if ( node -> c [ 1 ] != NULL )
          {
               node = node -> c [ 1 ];
               state = Child1;
          }
          else
               state = Child3;
          break;

     case Child3 :
          if ( node -> c [ 2 ] != NULL )
          {
               node = node -> c [ 2 ];
               state = Child1;
          }
          else
               state = Parent;
          break;
     case Parent :
          if ( node == knodes )
               state = Start;
          else if ( node == node -> p -> c [ 0 ] )
          {
               state = Child2;
               node = node -> p;
          }
          else if ( node == node -> p -> c [ 1 ] )
          {
```

```
                    state = Child3;
                    node = node -> p;
               }
               else
                    node = node -> p;
               break;
          }
     } while ( state != Start );
}


/*
 * Обход к-ичного дерева без использования стека.
 * Редуцированный граф переходов
 */

void traverseKReduced ()
{
     enum State
     {
          Start,
          Child,
          Parent,
          Calc
     };

     State state = Start;
     KNode * node;
     KNode * * child_ptr;
     KNode * * next_ptr;




     do
     {
          switch ( state )
          {
          case Start :
               state = Child;
               node  = knodes;
               child_ptr = node -> c;
               break;
          case Child:
               if ( child_ptr == node -> c )
                    cout << node -> data << ' ';
               if ( child_ptr - node -> c == K && node == knodes )
               {
                    state = Parent;
               }
               else if ( child_ptr - node -> c == K )
               {
                    next_ptr = node -> p -> c;
```

```
                    state = Calc;
               }
               else if ( * child_ptr != NULL )
               {
                    node = * child_ptr;
                    child_ptr = node -> c;
               }
               else
               {
                    ++ child_ptr;
               }
               break;
          case Calc :
               if ( * next_ptr == node )
               {
                    ++ next_ptr;
                    state = Parent;
               }
               else
                    ++ next_ptr;
               break;
          case Parent :
               if ( node != knodes )
               {
                    child_ptr = next_ptr;
                    node = node -> p;
                    state = Child;
               }
               else
                    state = Start;
               break;
          }
     } while ( state != Start );
}
```