Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

Вятский государственный университет

Колледж ВятГУ

ОТЧЕТ

ПО ДОМАШНЕЙ КОНТРОЛЬНОЙ РАБОТЕ №2

«Реализация элементарных структур данных на основе статической и динамической памяти»

ПО

«МДК 05.02 Разработка кода информационных систем»

Выполнил: студент учебной группы

ИСПк-202-52-00

Шипицын Евгений Алексеевич

Преподаватель:

Сергеева Елизавета Григорьевна

Цель работы: изучение принципов работы с базовыми структурами данных, получение навыков организации case-меню;

Задание:

1. Написать программу для работы со структурой данных “Двусвязный список”;
2. Структура данных должна быть реализована на основе статической и динамической памяти;
3. Работа со структурой должно осуществляться с помощью case-меню. Предусмотреть наглядную визуализацию содержимого структуры.

Двусвязный список : Двусвязный список - это структура данных, состоящая из узлов, каждый из которых содержит данные и две ссылки: одну на предыдущий узел и одну на следующий узел. Это позволяет обходить список как в прямом, так и в обратном направлении.

Свойства двусвязного списка:

1. Двунаправленность: Каждый узел имеет ссылки на предыдущий и следующий узлы, что позволяет эффективно перемещаться по списку в обоих направлениях.
2. Гибкость вставки и удаления: Поскольку каждый узел хранит ссылки на соседние узлы, вставка и удаление элементов в середине списка выполняется быстро и эффективно.

Программа:

1. Вначале программа просить пользователя выбрать тип памяти (статическая или динамическая);
2. Основываясь на предыдущем ответе запускается отдельный файл-модуль для работы с определенным типом памяти и структурой двусвязный список;
3. Если пользователь выбрал статическую память, ему предоставляются следующие возможности:
   1. Добавление элемента в список;
   2. Просмотр данных списка;
   3. Удаление элемента по значению;
   4. Удаление всего списка;
4. Если пользователь выбрал динамическую память, ему предоставляются следующие возможности:
   1. Добавление элемента в конец списка;
   2. Добавление элемента в начало списка;
   3. Добавление элемента в список перед указанным элементом;
   4. Удаление последнего элемента;
   5. Удаление первого элемента;
   6. Удаление определенного элемента;
   7. Отображение списка;

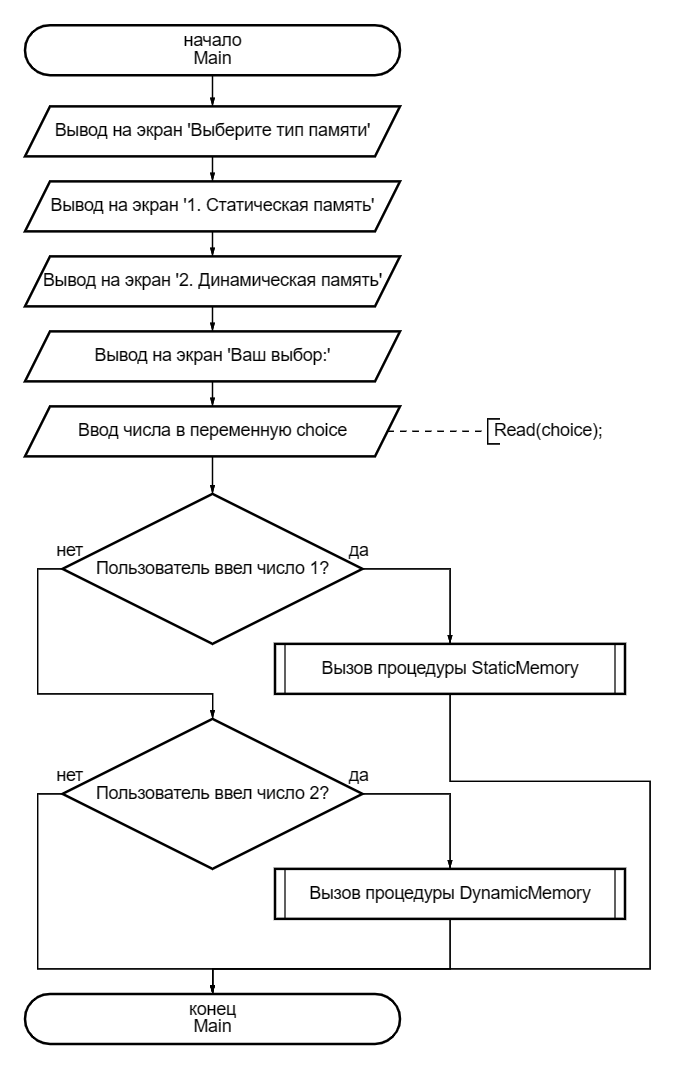


Рисунок 1 – Структура программы main.pas

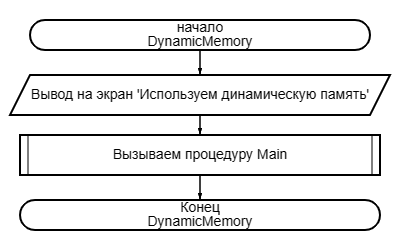


Рисунок 1 – Структура процедуры DynamicMemory

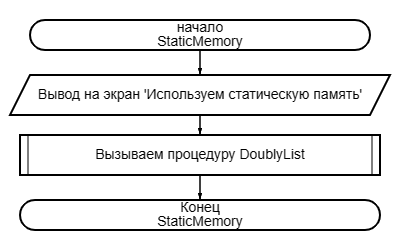


Рисунок 3 – Структура процедуры StaticMemory

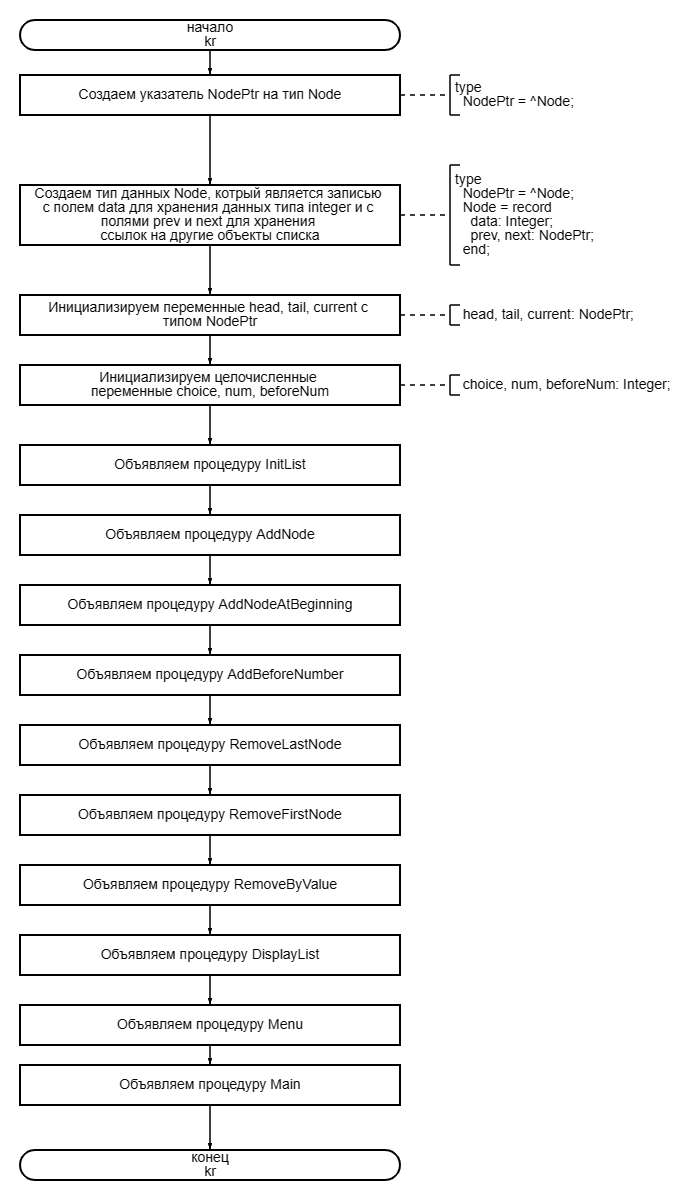


Рисунок 4 – Структура модуля kr

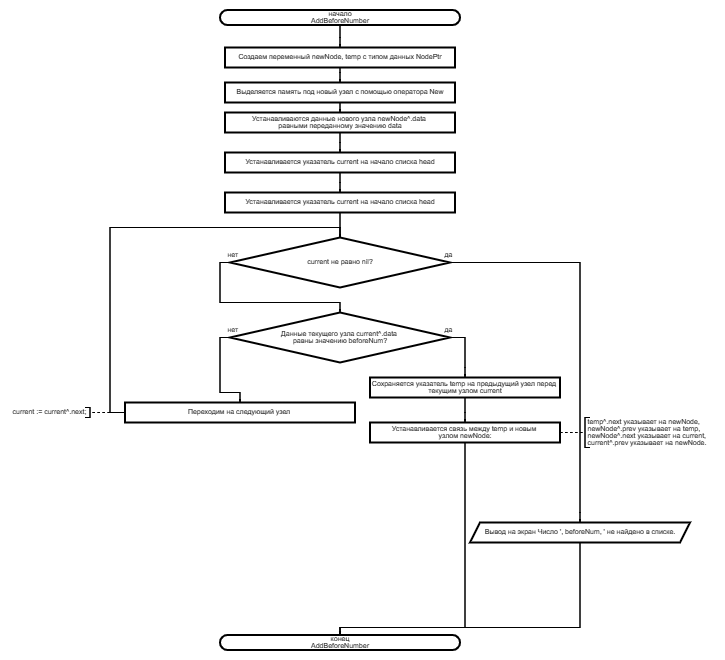


Рисунок 5 – Алгоритм работы процедуры AddBeforeNumber

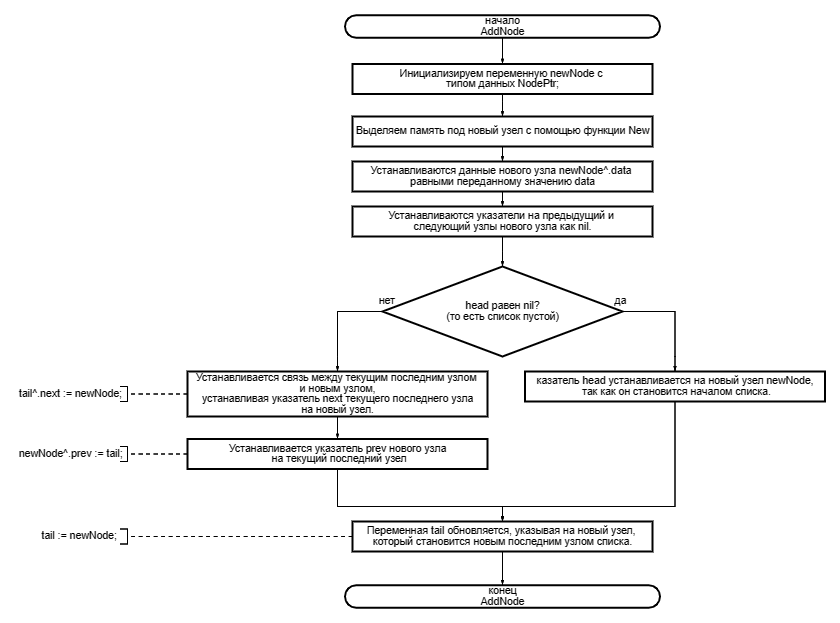


Рисунок 6 – Алгоритм работы процедуры AddNode

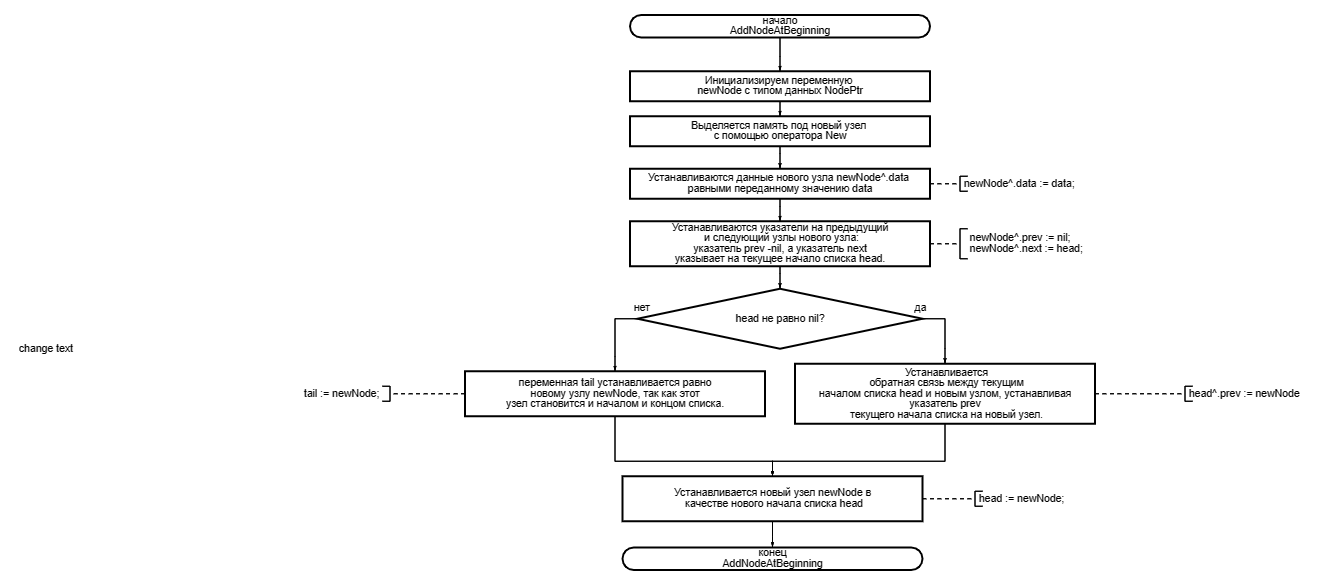


Рисунок 7 – Алгоритм работы процедуры AddNodeAtBegginning

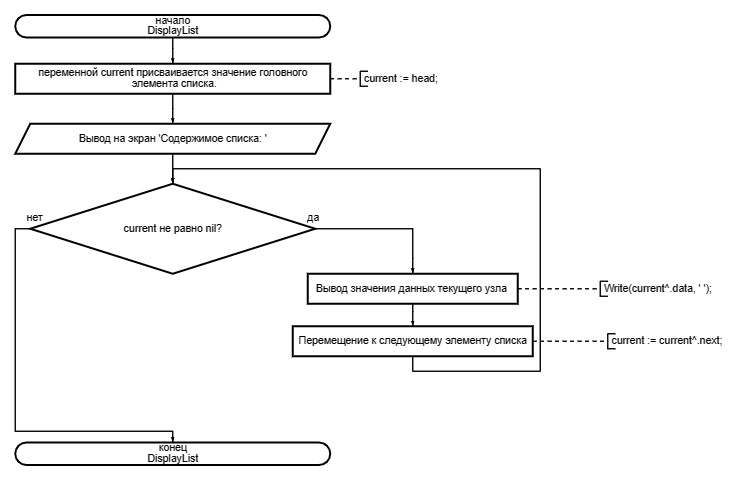


Рисунок 8 – Алгоритм работы процедуры DisplayList

Рисунок 9 – Алгоритм работы процедуры InitList

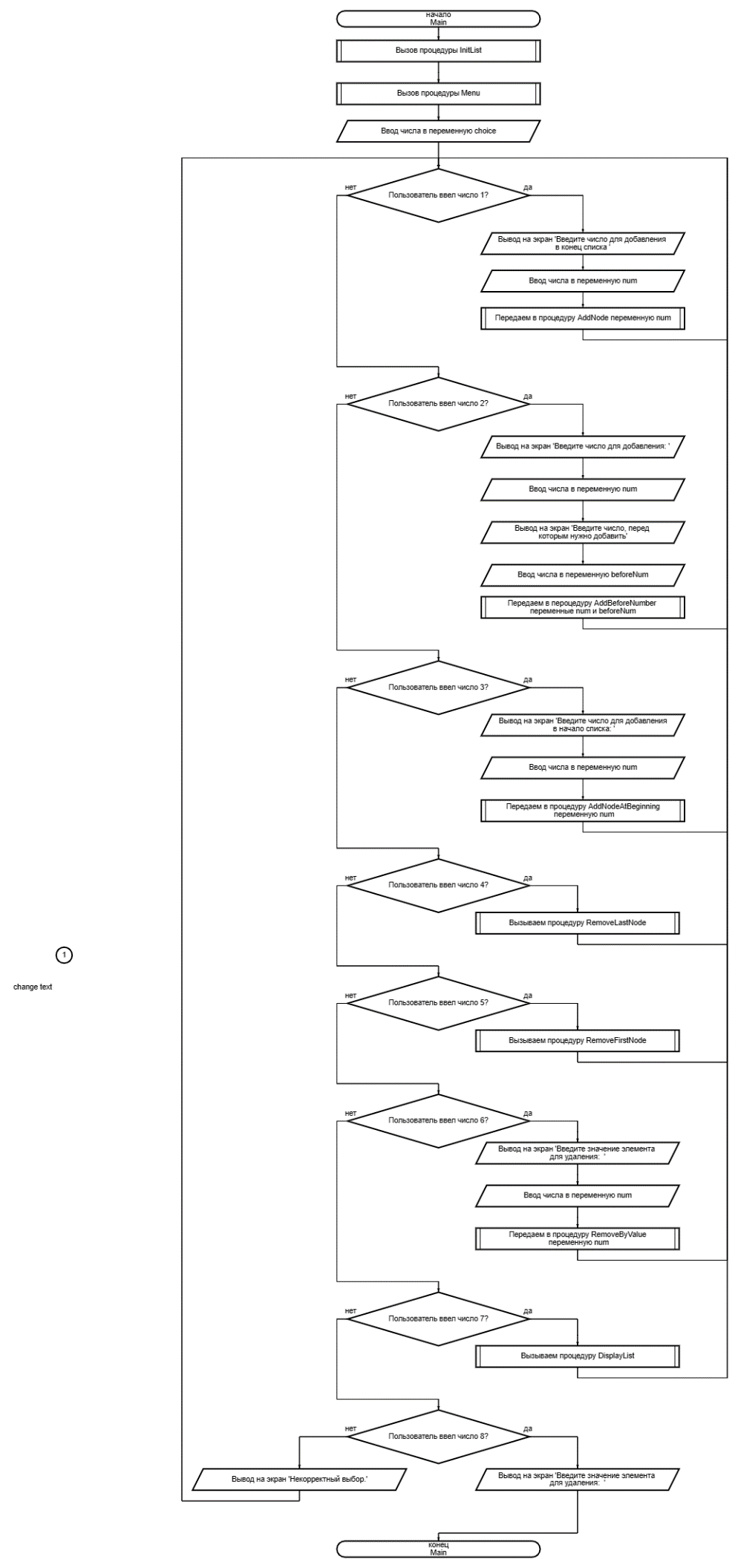


Рисунок 10 – Алгоритм работы процедуры main

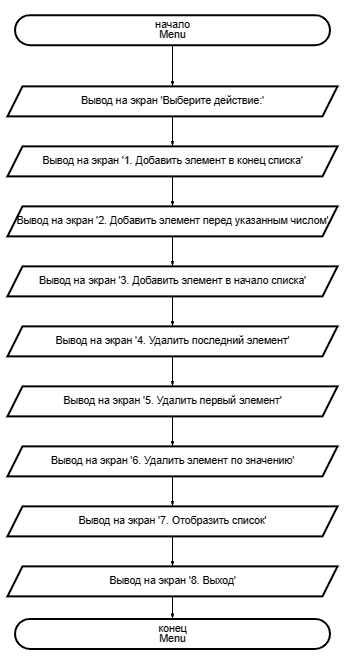


Рисунок 11 – Алгоритм работы процедуры menu

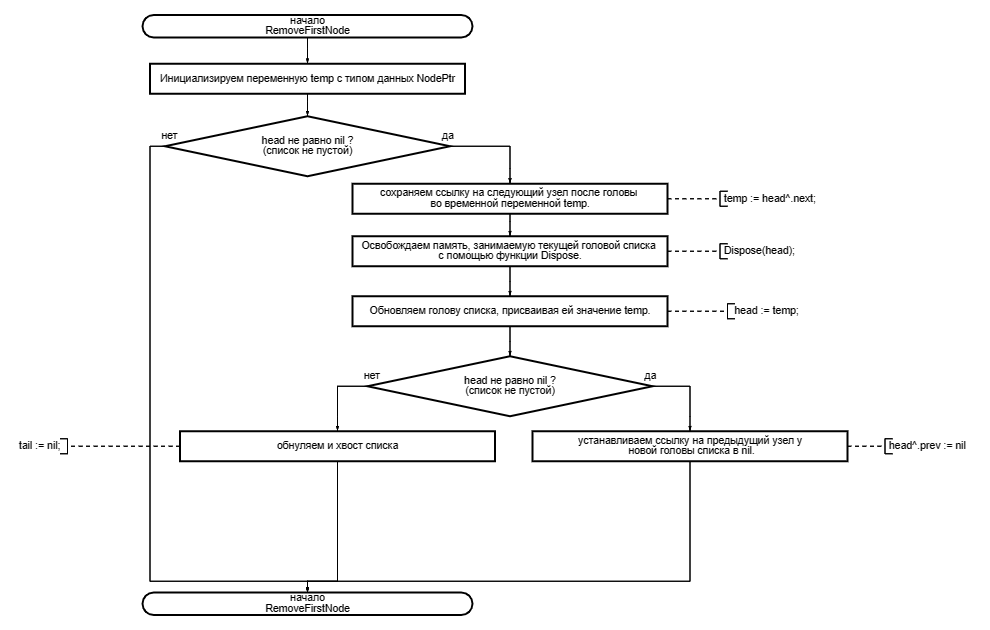


Рисунок 12 – Алгоритм работы процедуры RemoveFirstNode

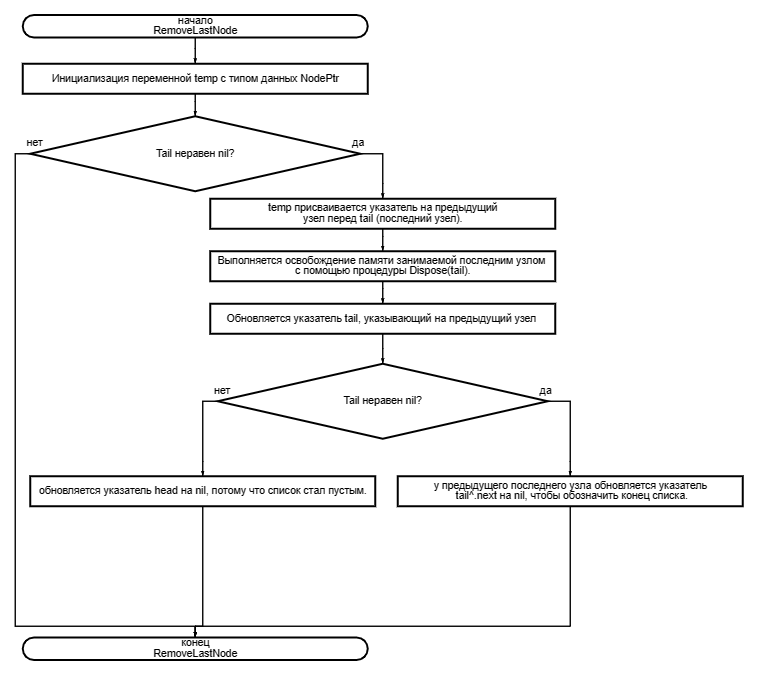


Рисунок 13 – Алгоритм работы процедуры RemoveLastNode

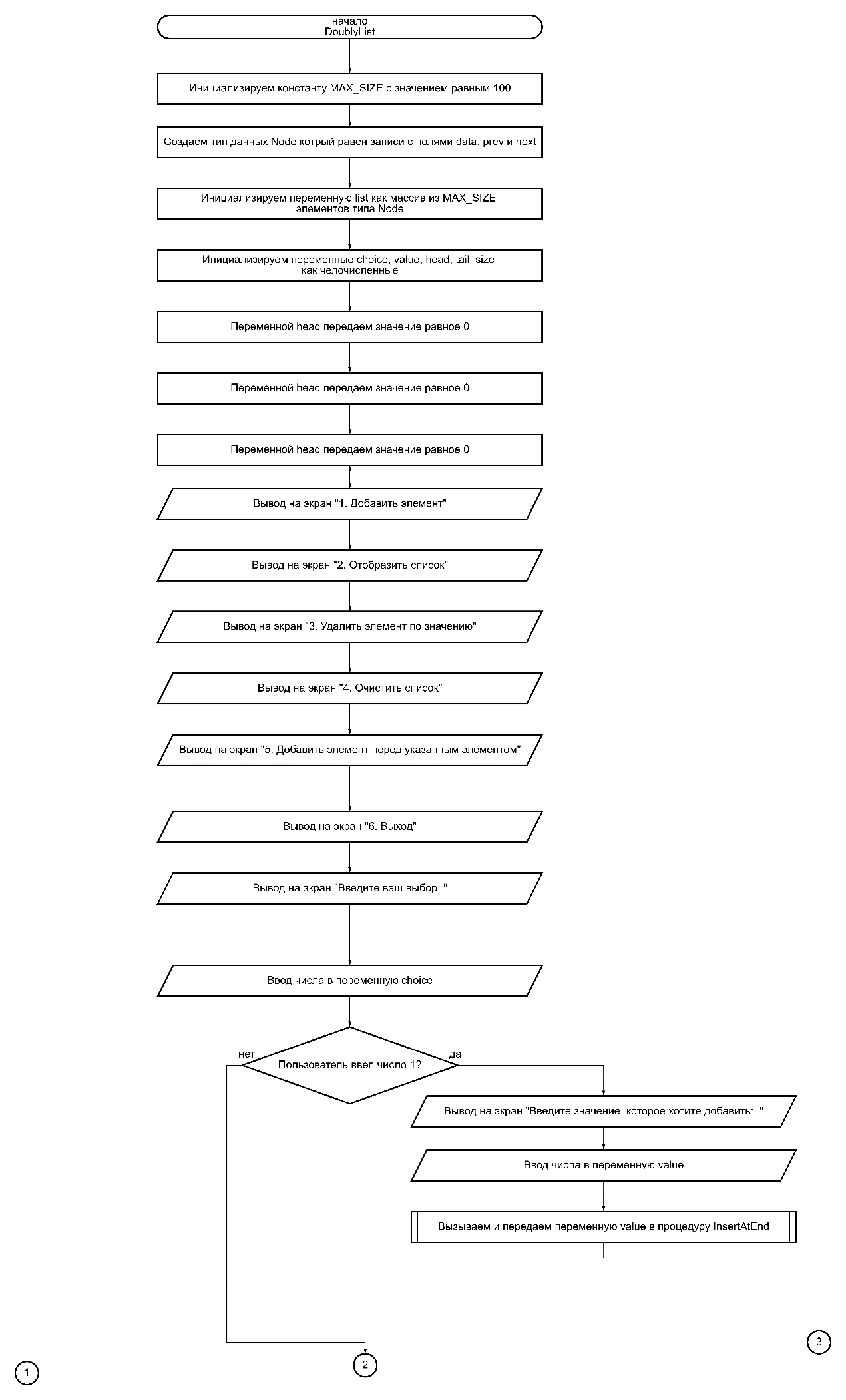


Рисунок 14.1 – Схема алгоритма работы процедуры DoublyList

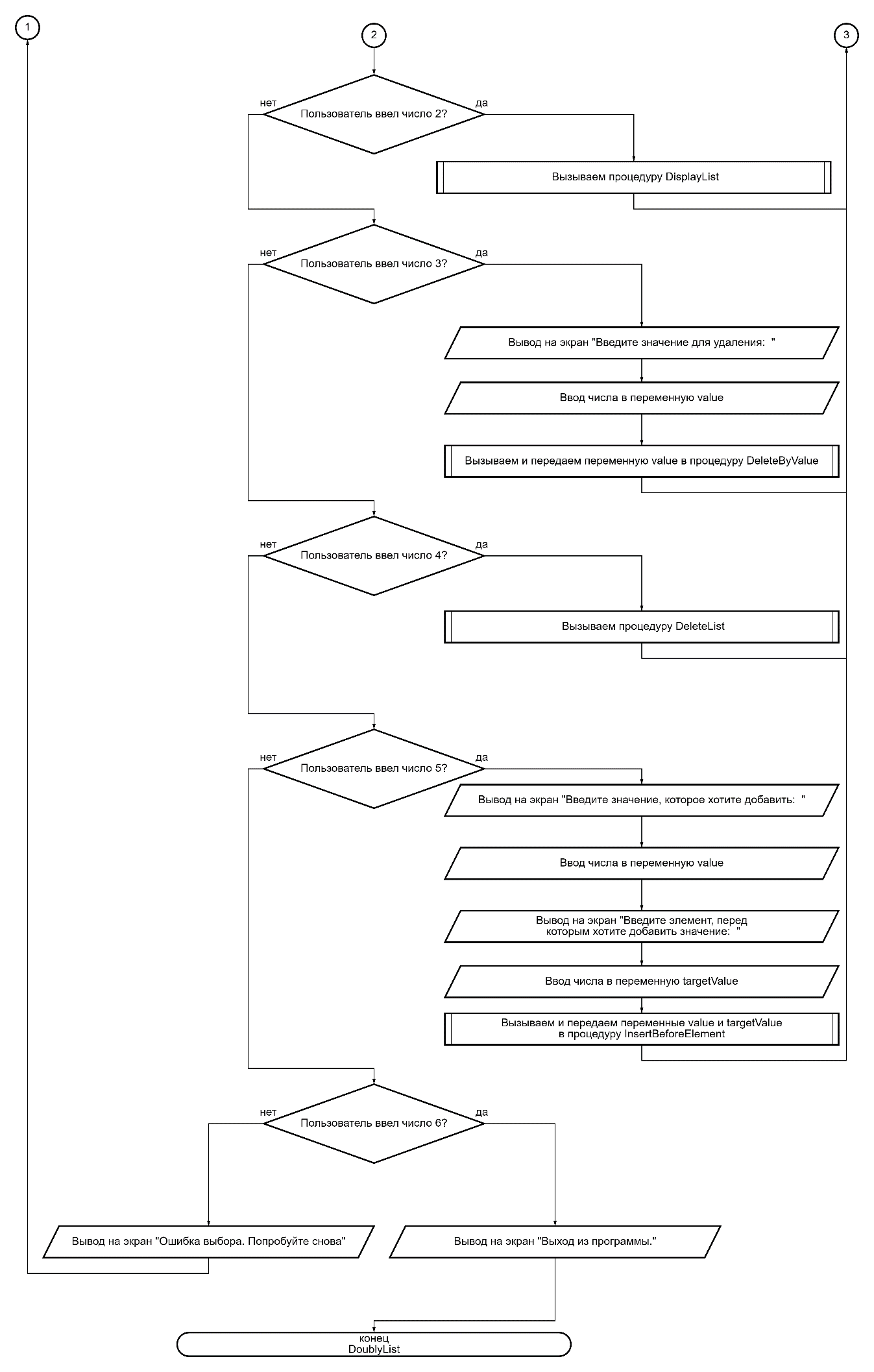


Рисунок 14.2 – Схема алгоритма работы процедуры DoublyList

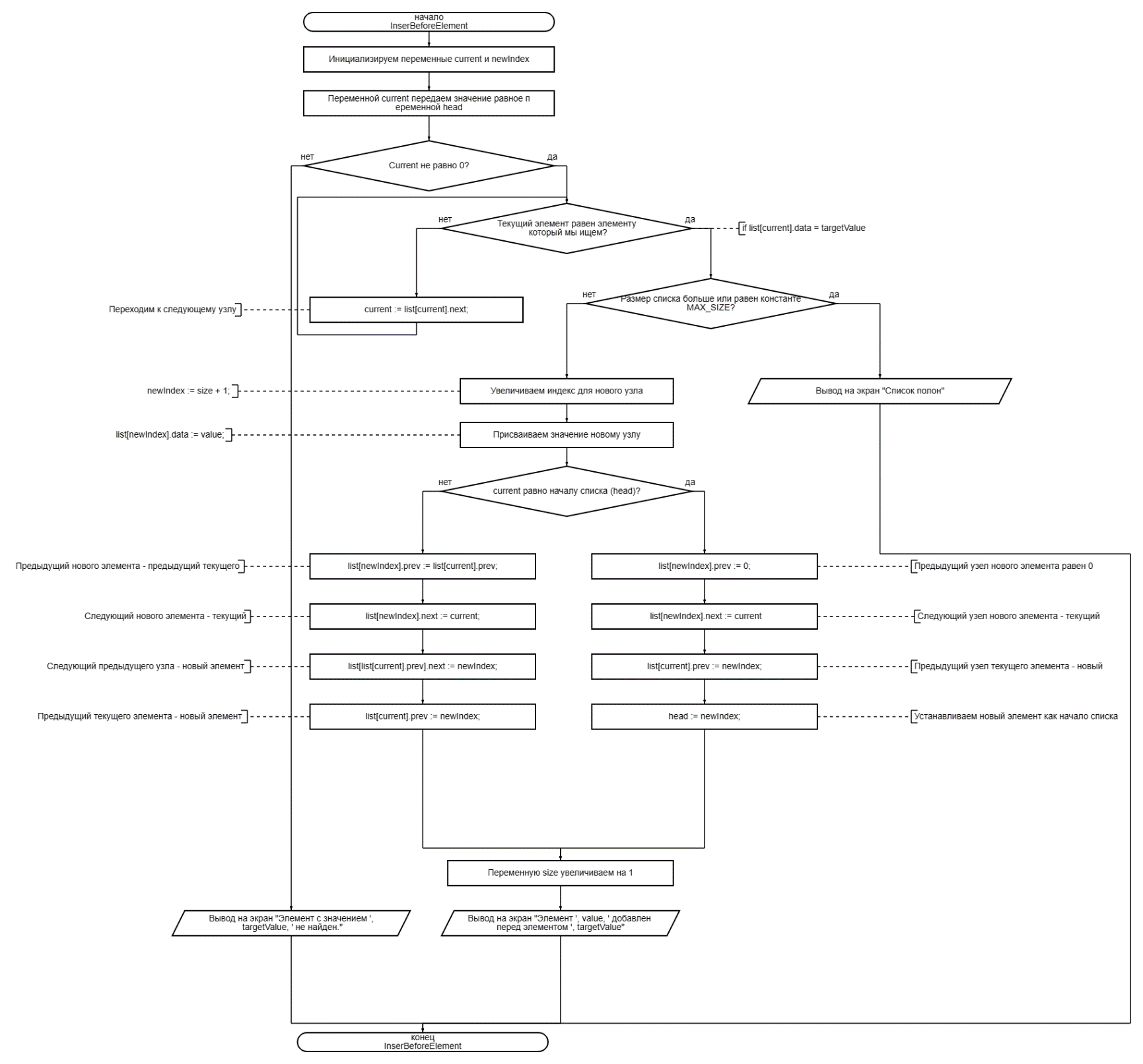


Рисунок 14.2 – Схема алгоритма работы процедуры InsertBeforeElement

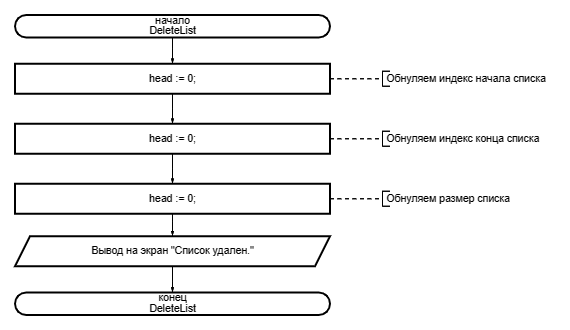


Рисунок 14 – Схема алгоритма работы процедуры DeleteList

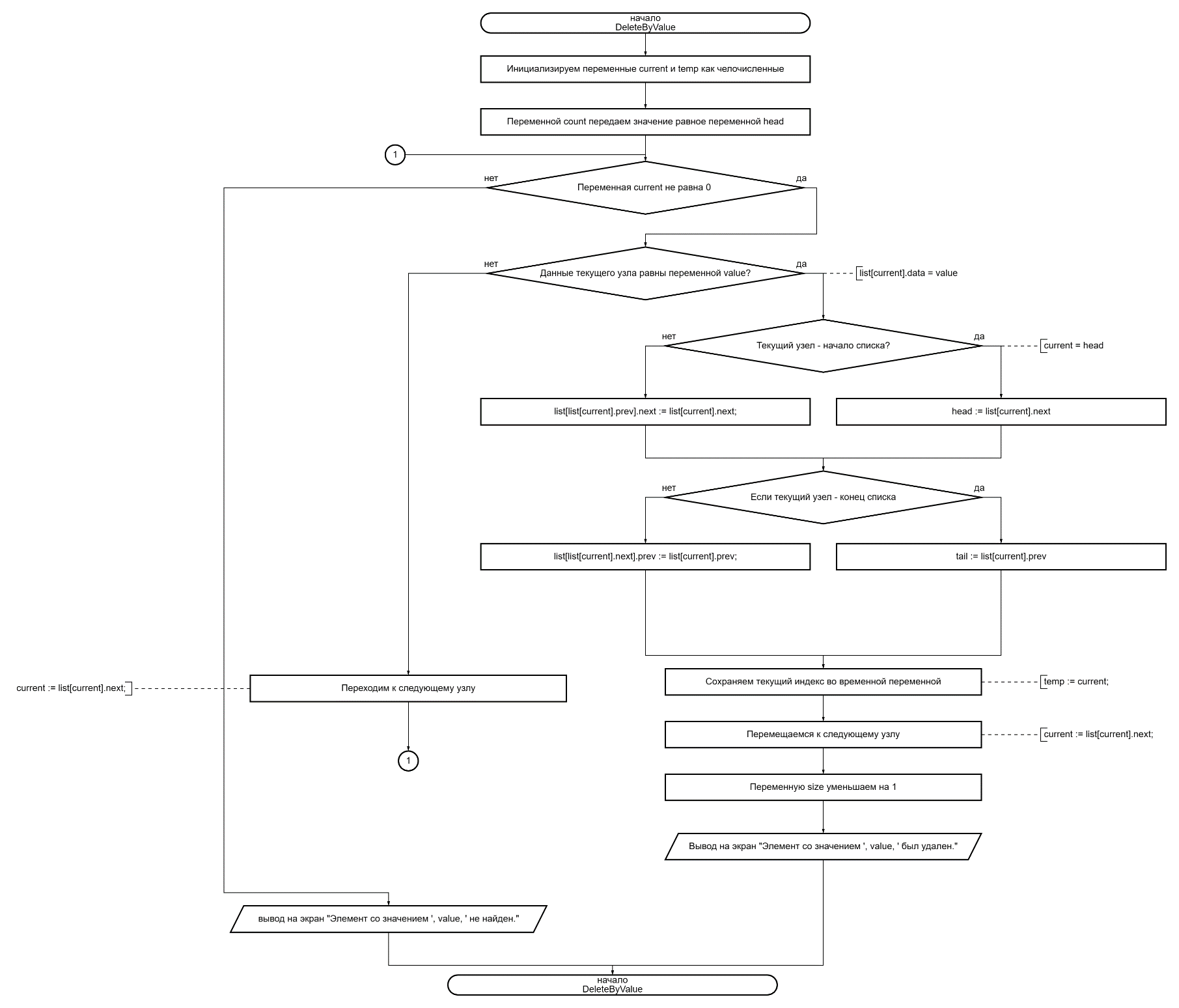


Рисунок 15 – Схема алгоритма работы процедуры DeleteByValue

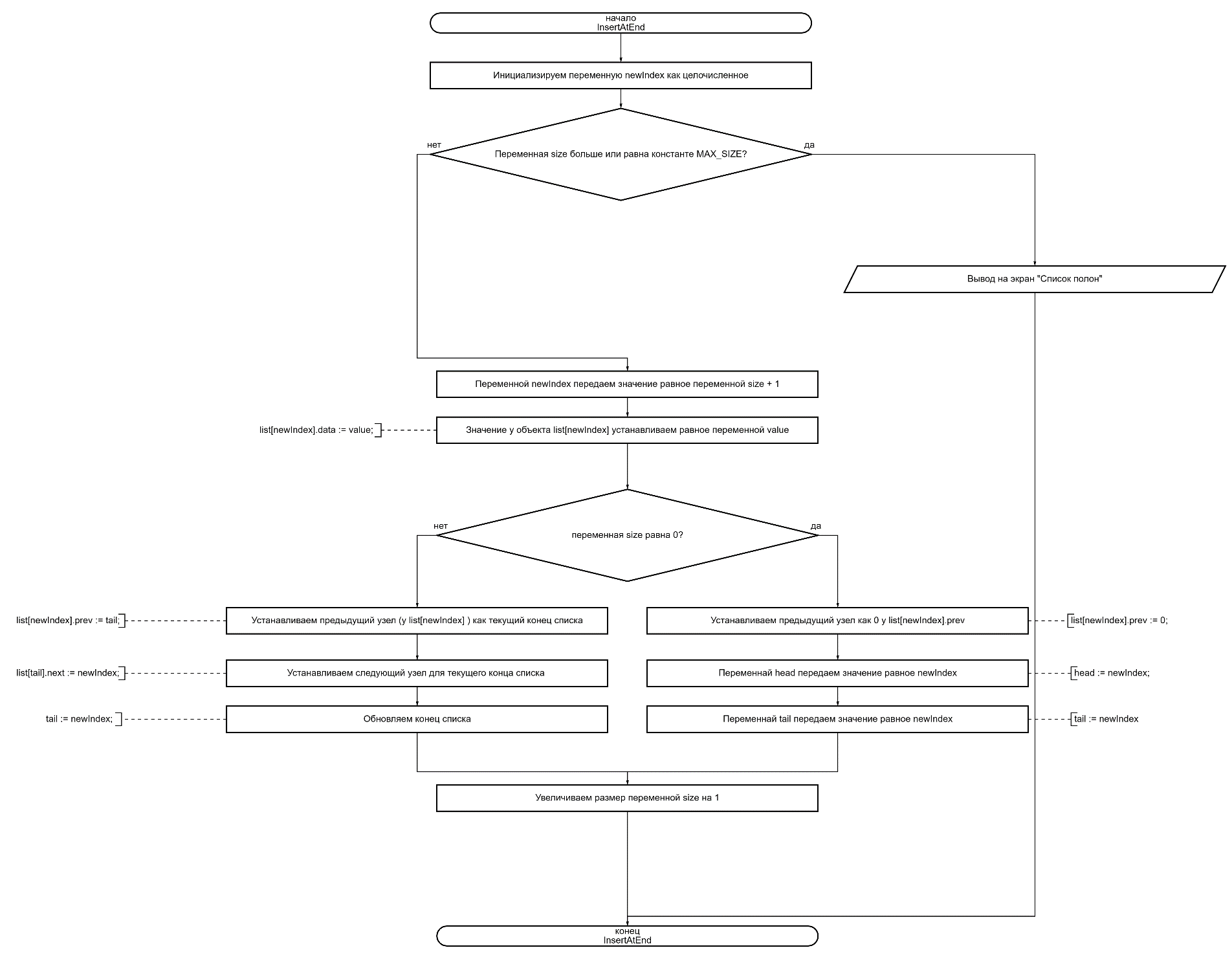


Рисунок 16 – Схема алгоритма работы процедуры InsertAtEnd

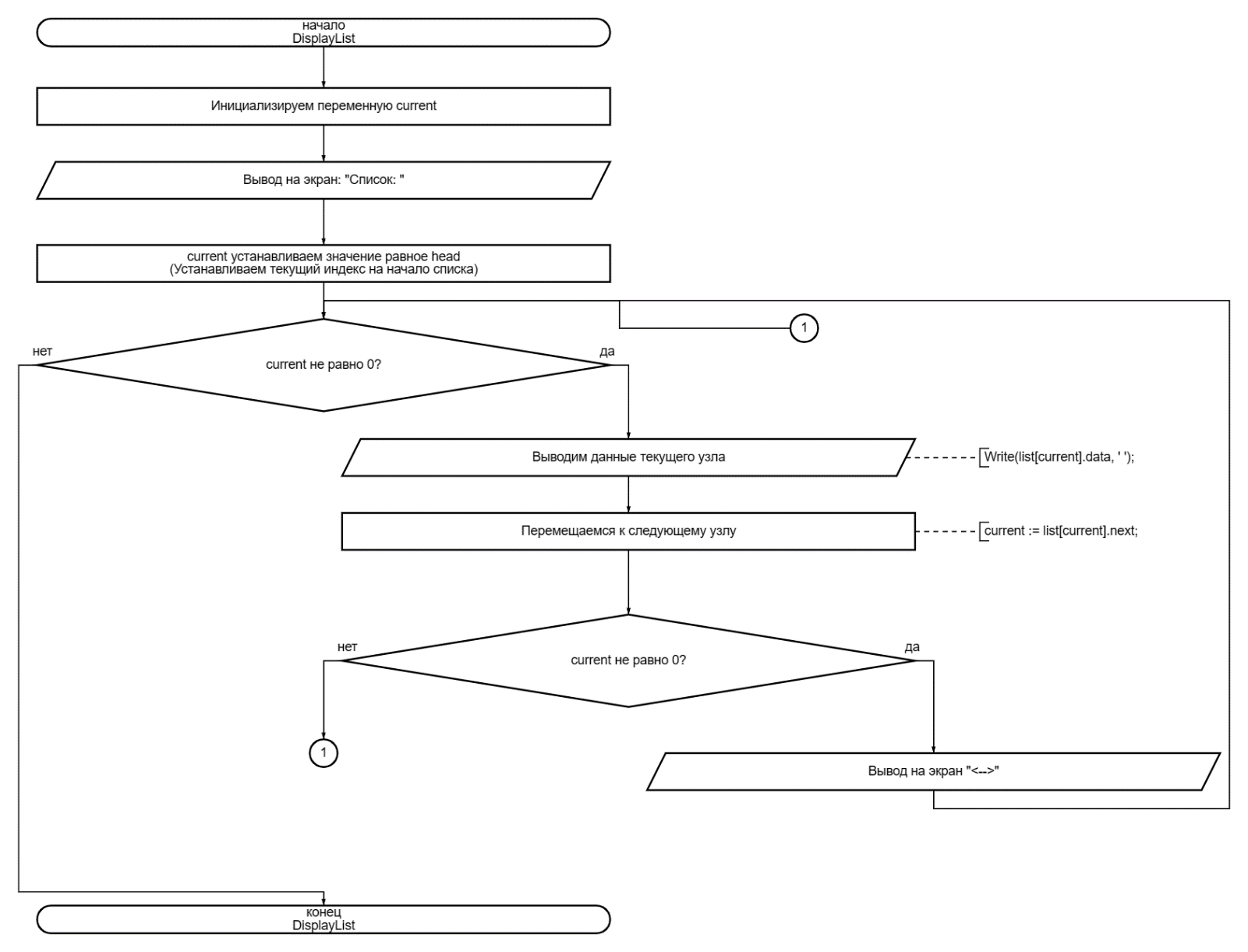


Рисунок 17 – Схема алгоритма работы процедуры DisplayList

Код программы:

**program** Главная\_программа;

**uses** kr, kr\_array;

**procedure** StaticMemory;

**begin**

WriteLn('Используем статическую память');

DoublyList;

**end**;

**procedure** DynamicMemory;

**begin**

WriteLn('Используем динамическую память');

Main;

**end**;

**var**

choice: Integer;

**begin**

WriteLn('Выберите тип памяти: ');

WriteLn('1. Статическая память');

WriteLn('2. Динамическая память');

Write('Выш выбор:');

Read(choice);

writeln(' ');

**case** choice **of**

1: StaticMemory;

2: DynamicMemory;

**end**;

**end**.

**unit** kr;

**interface**

**type**

NodePtr = ^Node;

Node = **record**

data: Integer;

prev, next: NodePtr;

**end**;

**procedure** InitList;

**procedure** AddNode(data: Integer);

**procedure** AddNodeAtBeginning(data: Integer);

**procedure** AddBeforeNumber(data, beforeNum: Integer);

**procedure** RemoveLastNode;

**procedure** RemoveFirstNode;

**procedure** RemoveByValue(value: Integer);

**procedure** DisplayList;

**procedure** Menu;

**procedure** Main;

**implementation**

**var**

head, tail, current: NodePtr;

choice, num, beforeNum: Integer;

**procedure** InitList;

**begin**

head := nil;

tail := nil;

**end**;

**procedure** AddNode(data: Integer);

**var**

newNode: NodePtr;

**begin**

**New**(newNode);

newNode^.data := data;

newNode^.prev := nil;

newNode^.next := nil;

**if** head = nil **then**

head := newNode

**else**

**begin**

tail^.next := newNode;

newNode^.prev := tail;

**end**;

tail := newNode;

**end**;

**procedure** AddNodeAtBeginning(data: Integer);

**var**

newNode: NodePtr;

**begin**

**New**(newNode);

newNode^.data := data;

newNode^.prev := nil;

newNode^.next := head;

**if** head <> nil **then**

head^.prev := newNode.

**else**

tail := newNode;

head := newNode;

**end**;

**procedure** AddBeforeNumber(data, beforeNum: Integer);

**var**

newNode, temp: NodePtr;

**begin**

**New**(newNode);

newNode^.data := data;

current := head;

**while** current <> nil **do**

**begin**

**if** current^.data = beforeNum **then**

**begin**

temp := current^.prev;

temp^.next := newNode;

newNode^.prev := temp;

newNode^.next := current ;

current^.prev := newNode;

**Exit**

**end**;

current := current^.next;

**end**;

WriteLn('Число ', beforeNum, ' не найдено в списке.');

**end**;

**procedure** RemoveLastNode;

**var**

temp: NodePtr;

**begin**

**if** tail <> nil **then**

**begin**

temp := tail^.prev;

Dispose(tail);

tail := temp;

**if** tail <> nil **then**

tail^.next := nil

**else**

head := nil;

**end**;

**end**;

**procedure** RemoveFirstNode;

**var**

temp: NodePtr;

**begin**

**if** head <> nil **then**

**begin**

temp := head^.next;

Dispose(head);

head := temp;

**if** head <> nil **then**

head^.prev := nil

**else**

tail := nil;

**end**;

**end**;

**procedure** RemoveByValue(value: Integer);

**var**

temp, toDelete: NodePtr;

**begin**

current := head;

**while** current <> nil **do**

**begin**

**if** current^.data = value **then**

**begin**

toDelete := current;

**if** toDelete = head **then**

**begin**

head := toDelete^.next;

**if** head <> nil **then**

head^.prev := nil

**else**

tail := nil;

**end**

**else if** toDelete = tail **then**

**begin**

tail := toDelete^.prev;

tail^.next := nil;

**end**

**else**

**begin**

temp := toDelete^.prev;

temp^.next := toDelete^.next;

toDelete^.next^.prev := temp

**end**;

Dispose(toDelete);

WriteLn('Элемент со значением ', value, ' удален из списка.');

**Exit**;

**end**;

current := current^.next;

**end**;

WriteLn('Элемент со значением ', value, ' не найден в списке.');

**end**;

**procedure** DisplayList;

**begin**

current := head;

Write('Содержимое списка: ');

Write('nil <- ');

**while** current <> nil **do**

**begin**

Write(current^.data, ' ');

current := current^.next;

**if** current <> nil **then**

Write('<- -> ');

**end**;

write('-> nil');

WriteLn;

**end**;

**procedure** Menu;

**begin**

WriteLn('Выберите действие:');

WriteLn('1. Добавить элемент в конец списка');

WriteLn('2. Добавить элемент перед указанным числом');

WriteLn('3. Добавить элемент в начало списка');

WriteLn('4. Удалить последний элемент');

WriteLn('5. Удалить первый элемент');

WriteLn('6. Удалить элемент по значению');

WriteLn('7. Отобразить список');

WriteLn('8. Выход');

**end**;

**procedure** Main;

**begin**

InitList;

**repeat**

Menu;

ReadLn(choice);

**case** choice **of**

1:

**begin**

Write('Введите число для добавления в конец списка: ');

ReadLn(num);

AddNode(num);

**end**;

2:

**begin**

Write('Введите число для добавления: ');

ReadLn(num);

Write('Введите число, перед которым нужно добавить: ');

ReadLn(beforeNum);

AddBeforeNumber(num, beforeNum);

**end**;

3:

**begin**

Write('Введите число для добавления в начало списка: ');

ReadLn(num);

AddNodeAtBeginning(num);

**end**;

4: RemoveLastNode;

5: RemoveFirstNode;

6:

**begin**

Write('Введите значение элемента для удаления: ');

ReadLn(num);

RemoveByValue(num);

**end**;

7: DisplayList;

8: WriteLn('Программа завершена.');

**else** WriteLn('Некорректный выбор.');

**end**;

ReadLn;

**until** choice = 8;

**end**;

**end**.

**unit** kr\_array;

**interface**

**const**

MAX\_SIZE = 100;

**type**

Node = **record**

data: Integer;

prev: Integer;

next: Integer;

**end**;

**var**

head, tail: Integer;

size: Integer;

list: **array**[1..MAX\_SIZE] **of** Node;

**procedure** InsertAtEnd(value: Integer); // Процедура для добавления элемента в конец списка

**procedure** DisplayList; // Процедура для отображения списка

**procedure** DeleteByValue(value: Integer); // Процедура для удаления элемента по значению

**procedure** DeleteList; // Процедура для удаления всего списка

**procedure** DoublyList; // Процедура для управления списком через case-menu

**procedure** InsertBeforeElement(value, targetValue: Integer); // Процедура для вставки элемента перед указанным элементом

**implementation**

**procedure** InsertBeforeElement(value, targetValue: Integer);

**var**

current, newIndex: Integer;

**begin**

current := head; // Устанавливаем текущий индекс на начало списка

**while** current <> 0 **do** // Пока не конец списка

**begin**

**if** list[current].data = targetValue **then** // Находим целевой элемент

**begin**

**if** size >= MAX\_SIZE **then** // Проверяем, не переполнен ли список

**begin**

WriteLn('Список полон'); // Выводим сообщение об ошибке

**Exit**; // Выходим из процедуры

**end**;

newIndex := size + 1; // Увеличиваем индекс для нового узла

list[newIndex].data := value; // Присваиваем значение новому узлу

**if** current = head **then** // Если целевой элемент - начало списка

**begin**

list[newIndex].prev := 0; // Предыдущий узел нового элемента равен 0

list[newIndex].next := current; // Следующий узел нового элемента - текущий

list[current].prev := newIndex; // Предыдущий узел текущего элемента - новый

head := newIndex; // Устанавливаем новый элемент как начало списка

**end**

**else** // Если целевой элемент не начало списка

**begin**

list[newIndex].prev := list[current].prev; // Предыдущий нового элемента - предыдущий текущего

list[newIndex].next := current; // Следующий нового элемента - текущий

list[list[current].prev].next := newIndex; // Следующий предыдущего узла - новый элемент

list[current].prev := newIndex; // Предыдущий текущего элемента - новый элемент

**end**;

Inc(size); // Увеличиваем размер списка

WriteLn('Элемент ', value, ' добавлен перед элементом ', targetValue); // Выводим сообщение об успешном добавлении

**Exit**; // Выходим из процедуры

**end**;

current := list[current].next; // Переходим к следующему узлу

**end**;

WriteLn('Элемент с значением ', targetValue, ' не найден.'); // Если элемент не найден

**end**;

// Процедура добавления элемента в конец списка

**procedure** InsertAtEnd(value: Integer);

**var**

newIndex: Integer; // Индекс нового узла

**begin**

**if** size >= MAX\_SIZE **then** // Проверка, что список не переполнен

**begin**

WriteLn('Список полон'); // Вывод сообщения об ошибке

**Exit**; // Выход из процедуры

**end**;

newIndex := size + 1; // Увеличиваем индекс для нового узла

list[newIndex].data := value; // Присвоение значению нового узла переданного значения

**if** size = 0 **then** // Если список пуст

**begin**

list[newIndex].prev := 0; // Устанавливаем предыдущий узел как 0

head := newIndex; // Устанавливаем начало списка на новый узел

tail := newIndex; // Устанавливаем конец списка на новый узел

**end**

**else** // Если список не пуст

**begin**

list[newIndex].prev := tail; // Устанавливаем предыдущий узел как текущий конец списка

list[tail].next := newIndex; // Устанавливаем следующий узел для текущего конца списка

tail := newIndex; // Обновляем конец списка

**end**;

Inc(size); // Увеличиваем размер списка

**end**;

// Процедура для отображения списка

**procedure** DisplayList;

**var**

current: Integer; // Индекс текущего узла

**begin**

writeln('');

writeln('Список:');

current := head; // Устанавливаем текущий индекс на начало списка

**while** current <> 0 **do** // Пока не конец списка

**begin**

Write(list[current].data, ' '); // Выводим данные текущего узла

current := list[current].next; //// Перемещаемся к следующему узлу

**if** current <> 0 **then**

Write('<- -> ');

**end**;

writeln('');

**end**;

// Процедура для удаления элемента по значению

**procedure** DeleteByValue(value: Integer);

**var**

current, temp: Integer; // Индекс текущего узла и временный индекс

**begin**

current := head; // Устанавливаем текущий индекс на начало списка

**while** current <> 0 **do** // Пока не конец списка

**begin**

**if** list[current].data = value **then** // Если данные текущего узла равны переданному значению

**begin**

**if** current = head **then** // Если текущий узел - начало списка

head := list[current].next // Обновляем индекс начала списка (head := list[current].next - происходит присваивание переменной head значения list[current].next)

**else**

list[list[current].prev].next := list[current].next; // Связываем предыдущий узел текущего с последующим (list[list[current].prev].next обращается к полю next предыдущего узла и устанавливает его значение равным индексу следующего узла текущего узла.

//list[current].next обращается к полю next текущего узла, которое содержит индекс следующего узла в списке.)

**if** current = tail **then** // Если текущий узел - конец списка

tail := list[current].prev // присваивает переменной tail значение индекса узла, который является предыдущим узлом относительно текущего узла в списке.

**else**

list[list[current].next].prev := list[current].prev; // Связываем следующий узел текущего с предыдущим

temp := current; // Сохраняем текущий индекс во временной переменной

current := list[current].next; // Перемещаемся к следующему узлу

size := size - 1; // Уменьшаем размер списка

WriteLn('Элемент со значением ', value, ' был удален.'); // Выводим сообщение об успешном удалении

**Exit**; // Выход из процедуры

**end**;

current := list[current].next; // Переходим к следующему узлу

**end**;

WriteLn('Элемент со значением ', value, ' не найден.'); // Если элемент не найден

**end**;

// Процедура для удаления всего списка

**procedure** DeleteList;

**var** i:integer;

**begin**

**for** i := 1 **to** MAX\_SIZE **do**

**begin**

list[i].data := 0; // обнуляем данные узла

list[i].prev := 0; // обнуляем индекс предыдущего узла

list[i].next := 0; // обнуляем индекс следующего узла

**end**;

head := 0; // обнуляем индекс начала списка

tail := 0; // обнуляем индекс конца списка

size := 0; // обнуляем размер списка

WriteLn('Список удален.'); // Выводим сообщение об успешном удалении списка

**end**;

**procedure** DoublyList; // Процедура для управления списком через консольное меню выбора

**var**

choice, value, targetValue: Integer; // Переменные для выбора, значения и целевого значения

**begin**

head := 0; // Обнуляем индекс начала списка

tail := 0; // Обнуляем индекс конца списка

size := 0; // Обнуляем размер списка

**repeat**

WriteLn('1. Добавить элемент');

WriteLn('2. Отобразить список');

WriteLn('3. Удалить элемент по значению');

WriteLn('4. Очистить список');

WriteLn('5. Добавить элемент перед указанным элементом');

WriteLn('6. Выход');

Write('Введите ваш выбор: ');

ReadLn(choice); // Считываем выбор

**case** choice **of**

1:

**begin**

Write('Введите значение, которое хотите добавить: ');

ReadLn(value); // Считываем значение для добавления

InsertAtEnd(value); // Вызываем процедуру добавления элемента

**end**;

2:

DisplayList; // Вызываем процедуру отображения списка

3:

**begin**

Write('Введите значение для удаления: ');

ReadLn(value); // Считываем значение для удаления

DeleteByValue(value); // Вызываем процедуру удаления элемента по значению

**end**;

4:

DeleteList; // Вызываем процедуру удаления всего списка

6:

WriteLn('Выход из программы.'); // Выводим сообщение о выходе

5:

**begin**

Write('Введите значение, которое хотите добавить: ');

ReadLn(value); // Считываем значение для добавления

Write('Введите элемент, перед которым хотите добавить значение: ');

ReadLn(targetValue); // Считываем значение целевого элемента

InsertBeforeElement(value, targetValue); // Вызываем процедуру добавления элемента перед указанным элементом

**end**;

**else**

WriteLn('Ошибка выбора. Попробуйте снова'); // Выводим сообщение об ошибке выбора

**end**;

**until** choice = 6; // Повторяем цикл до выбора выхода

**end**;

**end**.

Работа программы:

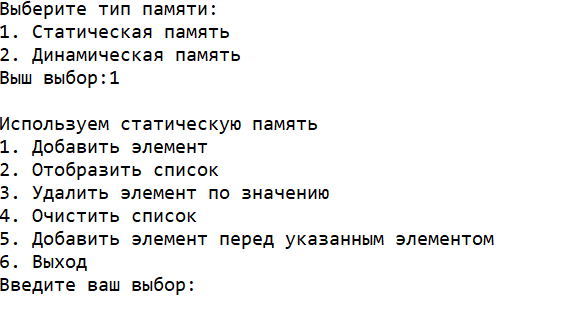
Если выбрать статическую память:

Рисунок 18 – Выбираем статический тип данных

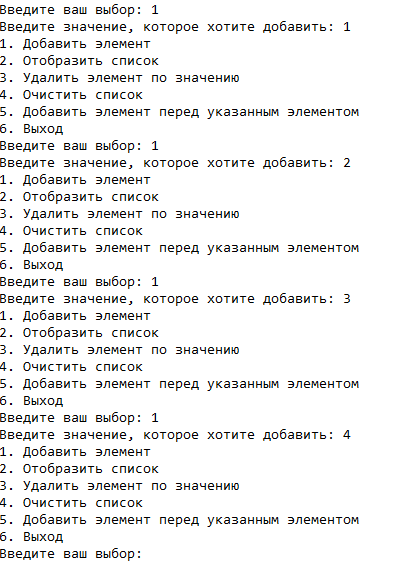
Добавляем элементы в список:

Рисунок 19 – Добавляем элементы в список

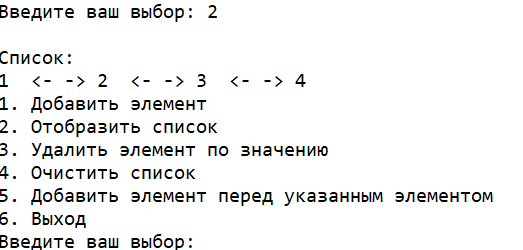
Выводим список на экран:

Рисунок 20 – Вывод элементов списка

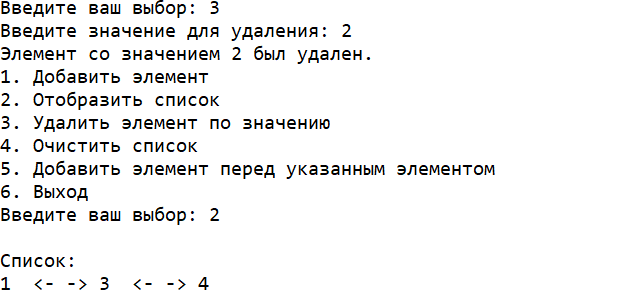
Удаляем элемент по значению:

Рисунок 21 – Удаление элемента по значению

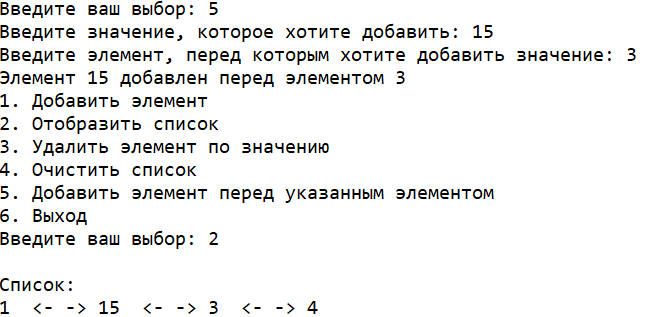
Добавление элемента перед указанным:

Рисунок 22 – Добавление элемента

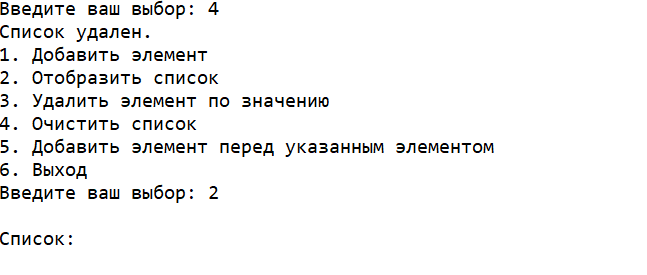
Очистка списка:

Рисунок 23 – Очистка списка

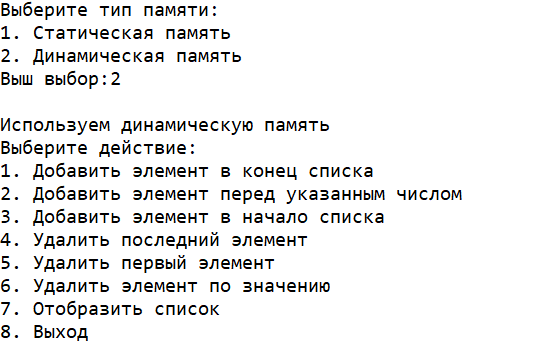
Если выбрать динамическую память:

Рисунок 24 – Динамическую память

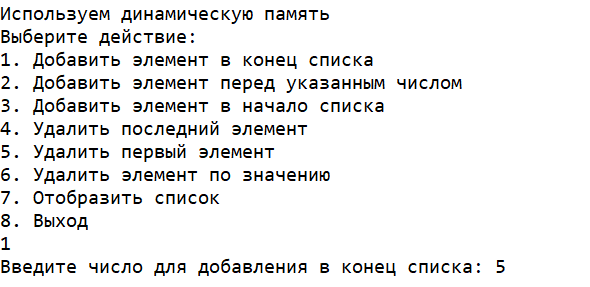
Добавляем элемент в конец списка:

Рисунок 25 – Добавляем элемент в конец списка

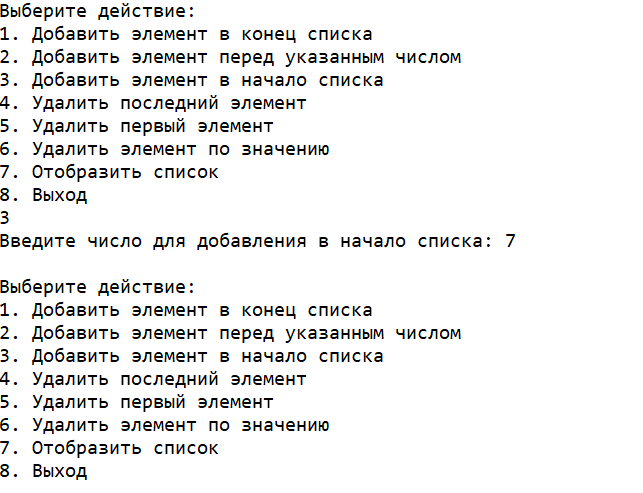
Добавление элемента в начало списка:

Рисунок 26 – Добавляем элемент в начало списка

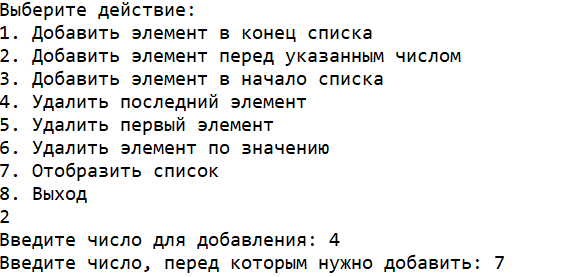
Добавляем элемент перед указанным:

Рисунок 27 – Добавляем элемент перед указанным элементом

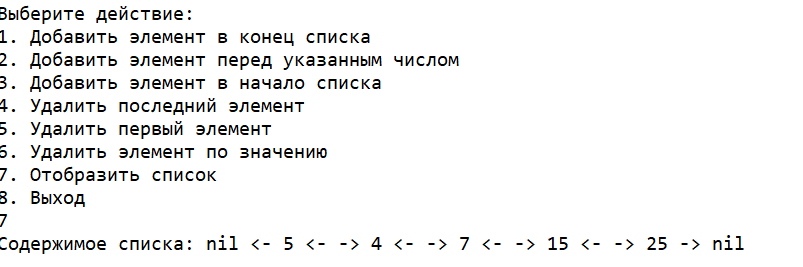
Выводим список:

Рисунок 28 – Вывод списка

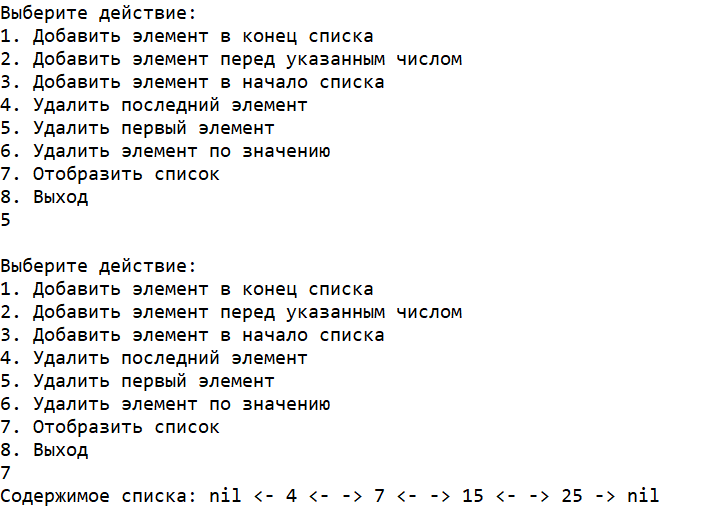
Удаляем первый элемент списка:

Рисунок 29 – Удаляем первый элемент списка

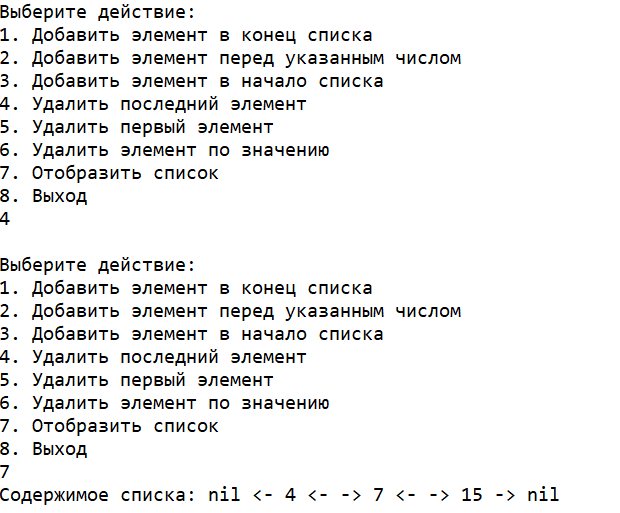
Удаляем последний элемент списка:

Рисунок 30 – Удаляем первый элемент списка

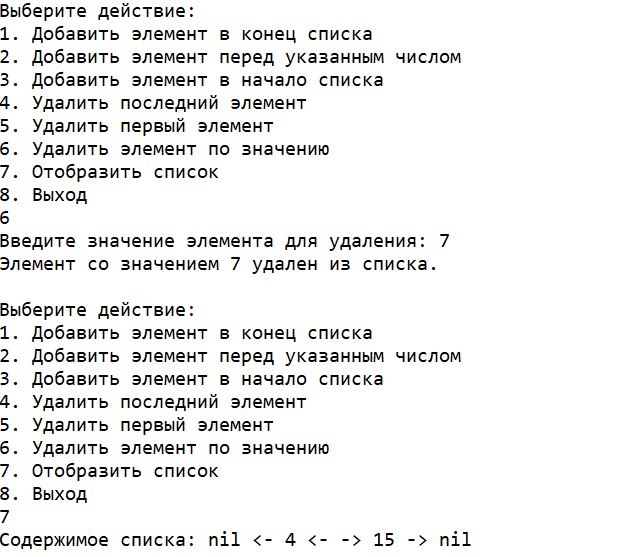
Удаляем элемент по значению:

Рисунок 31 – Удаляем элемент по значению

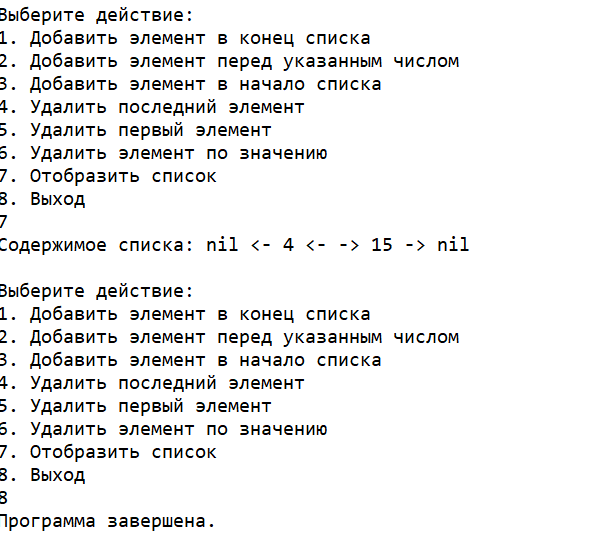
Вывод списка и выход из программы :

Рисунок 32 – Вывод списка на экран и выход из программы

Вывод:

Данная работа со структурой данных, двусвязный список, была достаточно сложной, а именно сложность возникла, из-за того, что нужно было уследить за обновлением связей у элементов списка.

Также сложность добавляла реализация двусвязного списка как со статической памятью, так и с динамической памятью.

Еще нужно было реализовать взаимодействие с программой через case-меню что также еще добавляло сложность к данной работе.

Поэтому пришлось данную программу разбить на 1 программу и 2 модуля.

Эта работа была довольно интересной и увлекательной, но признаться, довольно сложной. Взаимодействие с двусвязным списком и case-меню добавили мне новый уровень опыта.

Добавить возможность вывода списка как у динамической памяти

(nil<-5<->6<->7->nil)

Добавить возможность вставки элемента перед указанным