ЛАБОРАТОРНАЯ РАБОТА № 8

«Введение в алгоритмы и структуры данных Java»

Цель: приобретение навыков работы с рекурсивными методами, знакомство с динамическими структурами данных, приобретение навыков создания и использования простейшей динамической структуры.

Учебные вопросы:

1. Рекурсивные алгоритмы;
2. Динамические структуры данных;
3. Задания для самостоятельной работы;
4. Описание результата выполнения лабораторной работы.

1. РЕКУРСИВНЫЕ АЛГОРИТМЫ

Понятие рекурсии и простейшие примеры ее использования

В математике под рекурсией понимается способ организации вычислений, при котором функция вызывает сама себя с другим аргументом. Большинство современных языков высокого уровня поддерживают механизм рекурсивного вызова. Ввиду отсутствия в языке Java понятия функции рассматриваются рекурсивно вызываемые методы.

Таким образом, в языке **Java рекурсия** – это вызов метода из самого себя непосредственно (**простая рекурсия**) или через другие методы (**сложная**, или **косвенная рекурсия**). Пример сложной рекурсии: метод $m1$ вызывает метод $m2$, а метод $m2$ – метод $m1$.

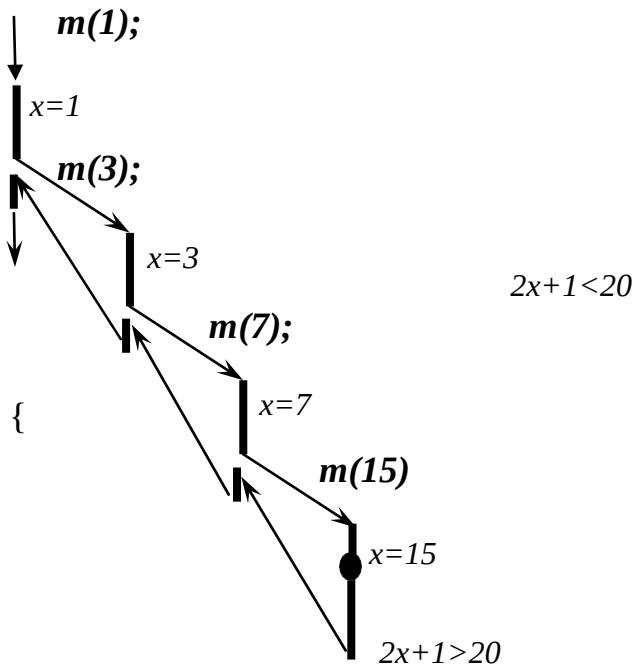
Рассмотрим несколько примеров простой рекурсии.

Пример 1. Для заданного параметра ***x*** вывести последовательность значений элементов числового ряда в соответствии со следующими требованиями:

– очередной элемент $x = 2*x+1$ (новое значение вычисляется с использованием старого);

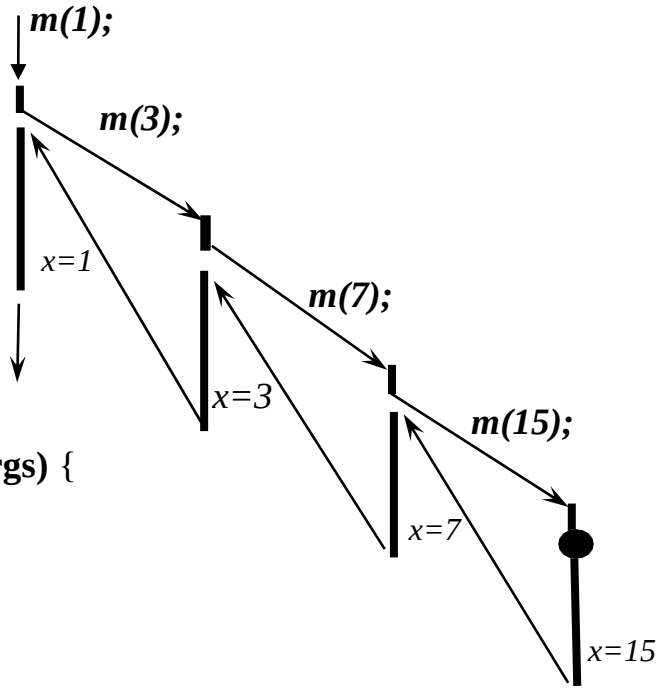
– $0 \le x < 20$.

```
public class Rec1 {
   public static void m(int x) {
      System.out.println("x="+x);
      if ( (2*x+1) <20) {
        m(2*x+1);
      }
   }
public static void main(String[] args) {
     m(1);
   }
}
```



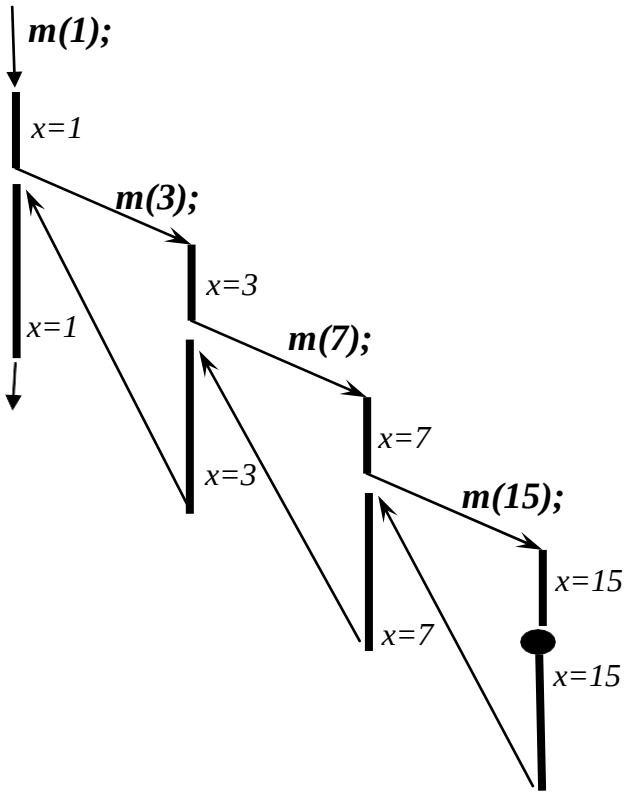
Пример 2. Вывести последовательность, представленную в предыдущем примере, в обратном порядке.

```
public class Rec2 {
   public static void m(int x) {
      if ( (2*x+1) <20) {
        m(2*x+1);
      }
      System.out.println("x="+x);
    }
   public static void main(String[] args) {
     m(1);
   }
}
```



Пример 3. Для вышеописанного задания сделать вывод пара- метра перед вхождением в рекурсивный вызов и после него.

```
public class Rec3 {
   private static int step=0;
   public static void m(int x) {
       space();
       System.out.println(""+x+"-> ");
       step++;
      if ((2*x+1) <20) {
        m(2*x+1);
      }
       step - -;
       space();
      System.out.println(""+x+ " <-");
   }
   public static void space() {
      for (int i = 0; i < step; i++) {
           System.out.print("  ");
      }
   }
   public static void main(String[] args) {
     m(1);
   }
}
```



Количество вложенных вызовов методов называется **глубиной рекурсии**.

Реализация рекурсивных вызовов опирается на механизм стека вызовов. Адрес возврата и локальные переменные метода записываются в стек, благодаря чему каждый следующий рекурсивный вызов этого метода пользуется своим набором локальных переменных и за счёт этого работает корректно.

На каждый рекурсивный вызов требуется некоторое количество оперативной памяти компьютера, и при чрезмерно большой глубине

рекурсии может наступить переполнение стека. Будет сгенерирована исключительная ситуация **StackOverflowError** (переполнение стека).

Вследствие этого рекомендуется избегать рекурсивных программ, которые приводят к слишком большой глубине рекурсии.

Также следует отметить, что рекурсию можно заменить циклом. Далее приведем наиболее часто используемые в учебных материалах примеры демонстрации работы рекурсии – вычисление факториала и чисел Фибоначчи.

Пример 4. Вычислить факториал числа *n* с использованием рекурсии.

Факториал числа *n* (обозначается *n!*) – произведение всех натуральных чисел от ***1*** до ***n*** включительно: *n!=1*2*...*n*. Пример 5!=1*2*3*4*5 =4!*5. Можно записать *n!=(n-1)!*n*.

```
public static int fact(int n){
    int result;
    if (n==1)
       return 1;
    else{
       result=fact(n-1)*n;
       return result;
    }
}
```

Пример 5. Вывести число Фибоначчи, заданное его номером в последовательности.

Последовательность Фибоначчи формируется так: нулевой член последовательности равен нулю, первый – единице, а каждый следующий – сумме двух предыдущих.

№ числа	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*...*	*20*
число	*0*	*1*	*1*	*2*	*3*	*5*	*8*	*13*	*21*	*...*	*6765*

Графическое представление порождаемой данным алгоритмом цепочки рекурсивных вызовов называется деревом рекурсивных вызовов. Для рассматриваемого алгоритма оно показано на рис. 2.1.

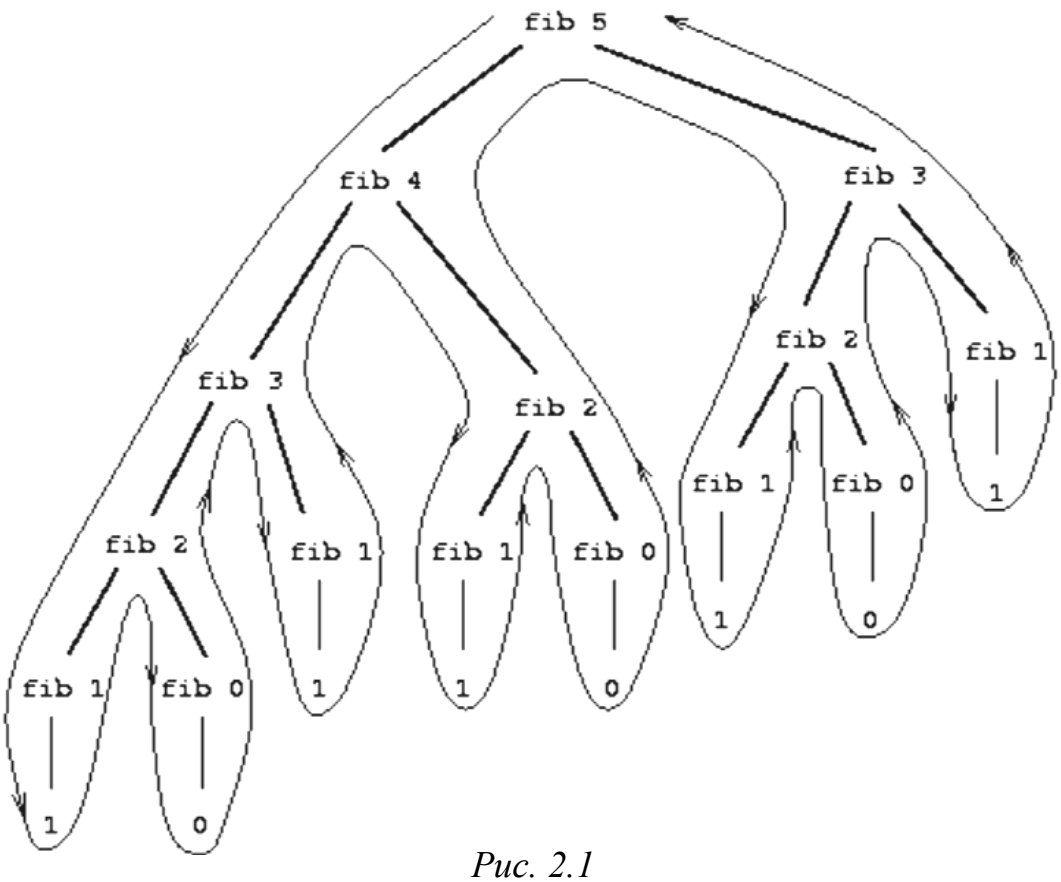


Рис. 2.1

```
public static int f(int n){
   if (n==0){
       return 0;
   }else
     if (n==1){
         return 1;
     } else {
         return f(n-2)+f(n-1);
   }
}
```

Трудоемкость рекурсивных реализаций алгоритмов зависит как от количества операций, выполняемых при одном вызове функции, так и от количества таких вызовов.

Более детальное рассмотрение рекурсий при расчете их трудоемкости приводит к необходимости учета затрат как на организацию вызова функции и передачи параметров, так и на возврат вычисленных значений и передачу управления в точку вызова.

Можно заметить, что некоторая ветвь дерева рекурсивных вызовов обрывается при достижении такого значения передаваемого параметра, при котором функция может быть вычислена непосредственно. Таким образом, рекурсия эквивалентна конструкции цикла, в котором каждый проход есть выполнение рекурсивной функции с заданным параметром.

2 ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

Понятие динамических и статических структур данных

Используемые в программировании данные можно разделить на две большие группы: данные статической структуры и данные динамической структуры.

В предыдущих лабораторных работах для хранения однотипных элементов: числовых, строковых, объектов собственных разработанных классов использовались массивы. Массив– это одна из базовых статических структур данных.

Данные статической структуры – это данные, взаиморасположение и взаимосвязи элементов которых всегда остаются постоянными. Количество элементов – величина постоянная.

Данные динамической структуры – это данные, внутреннее строение которых формируется по какому-либо закону, но количество элементов, их взаиморасположение и взаимосвязи могут динамически изменяться во время выполнения программы согласно закону формирования.

Динамическая структура построена как цепочка взаимосвязанных элементов одного типа, обычно называемых узлами, каждый из которых включает:

– **информационные поля (поле)** – где содержатся необходимые данные. Информационное поле само может являться интегрированной структурой – массивом, другой динамической структурой и т.п.;

– **адресные поля (поле)** – содержат один или несколько указателей, связывающий данный элемент с другими элементами структуры.

Количество информационных и адресных полей определяется классом структуры.

К динамическим структурам относят:

– однонаправленные (односвязные) списки;

– двунаправленные (двусвязные) списки;
– кольцевые (циклические) списки;
– стеки;
– деки;
– очереди;
– бинарные деревья и др.

Они отличаются способом связи отдельных элементов и/или допустимыми операциями.

Однонаправленный список – линейная структура, где каждый элемент (кроме последнего) имеет указатель на следующий элемент. У последнего элемента указатель равен NULL.

Двунаправленный список – линейная структура, где каждый элемент (кроме первого и последнего) имеет указатели на следующий элемент и на предыдущий элемент. У первого элемента указатель на предыдущий элемент равен NULL, а у последнего – указатель на следующий равен NULL.

Кольцевые списки – циклическая структура. Они могут быть однонаправленными и двунаправленными. Особенностью является то, что ссылка последнего элемента указывает на первый элемент, а в двунаправленном списке у первого элемента дополнительно присутствует ссылка на последний элемент списка.

В **однонаправленных**, **двунаправленных** и **циклических списках** добавление, включение и исключение элементов производится в любом месте.

Очередь – линейный список, в котором все включения элементов производятся на одном конце списка, а все исключения (и обычно всякий доступ) – на другом по принципу FIFO (First In – First Out, «первым пришёл – первым вышел»).

Стек – линейная структура данных, в которой добавление и удаление элементов осуществляется в одной стороны списка (с конца) по принципу LIFO (Last In – First Out, «последним пришёл – первым вышел»).

Дек – линейная структура данных, в которой можно добавлять и удалять элементы с обоих концов списка.

Бинарные деревья – древовидная структура данных, в которой каждый элемент (родительский узел) имеет ссылки на два других элемента (дочерних), называемых левым и правым наследниками (потомками).

Средством доступа к динамическим структурам и их элементам является указатель (адрес) на место их текущего расположения в памяти. Как правило, это указатель на первый элемент структуры.

Достоинства связного представления данных:

– размер структуры ограничивается только доступным объемом машинной памяти;

– при изменении логической последовательности элементов структуры требуется не перемещение данных в памяти, а только коррекция указателей;

– большая гибкость структуры, возможность обеспечить ее изменчивость.

Основные недостатки:

– на поля связок расходуется дополнительная память;

– доступ к элементам связной структуры может быть менее эффективным по времени.

В языке Java существует достаточное количество стандартных классов, реализующих динамическое взаимодействие элементов. Это, прежде всего, классы коллекций. Среди них списки, например ArrayList, LinkedList, очереди ArrayDeque и много других. В данной лабораторной работе рассмотрено создание и использование динамической структуры данных – однонаправленного списка с элементами в виде объектов собственного класса.

Формирование и вывод односвязного списка

Линейный однонаправленный список (его еще называют односвязным списком) – это простейшая динамическая структура данных. Ее схема представлена на рисунке

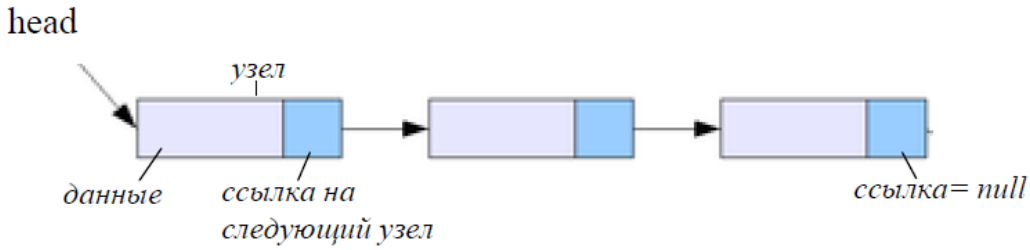


Рисунок 1

Он состоит из узлов (звеньев), каждый из которых содержит как, собственно, данные, так и ссылку на следующий узел списка. В односвязном списке можно передвигаться только в сторону от начала (головы списка **head**) до конца списка (хвоста – **tail**). Узнать адрес предыдущего элемента, опираясь на содержимое текущего узла, невозможно. Программисту должен быть доступен указатель на начало списка (голову – head). Зная его, можно получить доступ к любому элементу, пройдя последовательно по ссылкам.

Пример 1. Построить однонаправленный список из предварительно созданных независимых элементов (узлов), у которых значением поля будет целое число, равное номеру элемента, и вывести значения полей на экран, пример показан на рисунке 2.

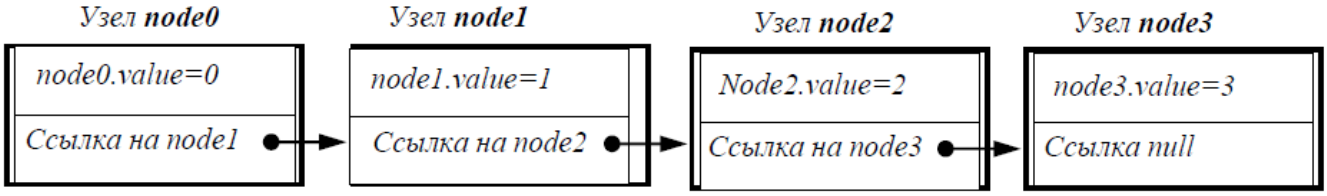


Рисунок 2

В начале пример построения списка с «головы»:

```
package dsd_manual;

class Node { // КЛАСС – СТРУКТУРА ЭЛЕМЕНТА СПИСКА
   public int value; // значение
   public Node next;    // поле – ссылка (указатель) на следующий узел

   Node(int value, Node next) { // конструктор класса
      this.value = value;
      this.next = next;
}}

public class DSD_manual { // ГЛАВНЫЙ КЛАСС
   public static void main(String[] args) {
   // создание несвязанных узлов с помощью конструктора
      Node node0 = new Node(0, null); // 0-й узел – будет головой в списке
      Node node1 = new Node(1, null);
      Node node2 = new Node(2, null);
      Node node3 = new Node(3, null); // последний узел – будет хвостом в списке

   // связывание узлов в список с помощью ссылок
      node0.next = node1;
      node1.next = node2;
      node2.next = node3;

   // вывод списка с использованием вспомогательной переменной ref,
   // соответствующей текущему значению ссылки при прохождении по списку
      Node ref = node0; // для перемещения по списку достаточно помнить голову
      while (ref != null) {
          System.out.print(" " + ref.value);
          ref = ref.next;
      }  }}
```

Вышеописанный пример приведен только для демонстрации расположения элементов в списке. В реальности элементы сразу при создании связываются в список и запоминаются в одной переменной, в которой хранится ссылка на первый элемент. В вышеприведенном примере – это **node0**, а переменные **node1, node2** и **node3 –** излишние.

Существует два основных способа построения односвязного

списка. Первый способ – изначально создается «голова», а новые элементы помещаются в конец списка, образуя хвост. Второй способ – изначально создается «хвост» и «голова» меняет свое положение при наращивании списка. В нижеприведенном примере показаны оба варианта.

Пример 2. Построить односвязный список из 10 элементов, у которых значением поля будет целое число, равное номеру элемента, и вывести значения полей на экран.

```
package dsd_create;

class Node { // описание элемента
   public int value;
   public Node next;

   Node(int value, Node next) {    // конструктор
      this.value = value;
      this.next = next;
 }}

public class DSD_create{    // главный класс
   public static void main(String[] args) {
   // создание 1-го узла, который изначально является и головой, и хвостом списка
     Node head=new Node(0, null);
     Node tail=head;
   // добавление элементов с наращиванием хвоста
     for (int i = 0; i <9; i++) {
       tail.next=new Node(i+1, null);
       tail=tail.next;              // указатель на созданный элемент запоминается
      }                             // как указатель на новый хвост

   // вывод элементов на экран
      Node ref = head;              // ref – рабочая переменная для текущего узла
      while (ref != null) {
         System.out.print(" " + ref.value);
         ref = ref.next;
      }
 }}
```

Ниже приведен второй вариант метода **main**, в котором список строится изначальным созданием «хвоста», причем для него создавать дополнительную переменную не нужно.

Узел «головы» получается путем создания нового узла, в котором старая ссылка на «голову» становится полем **next**, т.е. «хвостом».

```
public static void main(String[] args) {
  // добавление элементов с перемещением головы (наращивание с головы)
   Node head=null;     // начальное значение ссылки на голову
   for (int i =9; i>=0; i--) {
      head=new Node(i, head);
    }
  // вывод элементов на экран
     Node ref = head;
     while (ref != null) {
        System.out.print(" " + ref.value);
        ref = ref.next;
     }
  }
```

Изменение линейного односвязного списка

Как указывалось ранее, основным достоинством динамических списков по сравнению, например с массивами, является легкость добавления и удаления их элементов.

Рассмотрим алгоритмы изменения списков, создание которых приведено в примере 2 данной лабораторной работы.

Добавление элемента в начало списка (с головы). Данная задача решается в одно действие: создается новый узел, который становится новой «головой» списка, а старая ссылка на «голову» становится ссылкой на следующий элемент (рисунок 3)

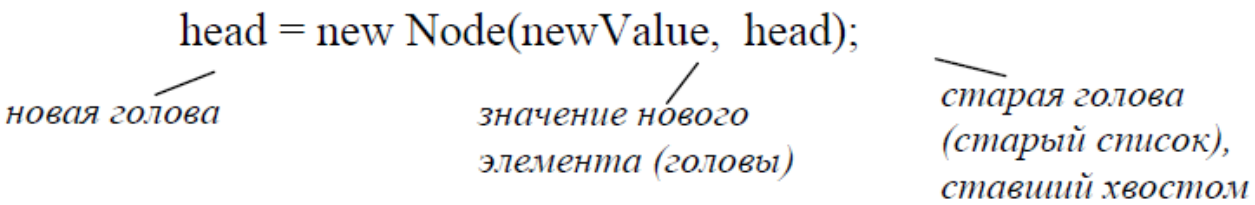


Рисунок 3

Добавление элемента в конец списка. Для того чтобы добраться до последнего элемента, необходимо пройтись по всему списку с «головы», пока не будет получен элемент, у которого ссылка **next** на следующий элемент равна **null**. Далее добавляется новый элемент. Ниже представлен фрагмент программы.

```
// создается новый элемент со значением 123 – будущий хвост
    Node newtail=new Node(123, null);

// для перемещения по списку используется вспомогательная переменная ref,
// которой в качестве начальной ссылки передается указатель на «голову»
    ref = head;
    while (ref.next != null) { // пока не последний элемент
      ref = ref.next;
    }
// указателю последнего элемента присваиваем новый «хвост» (элемент)
    ref.next=newtail;
```

Добавление элемента в список в указанное место (вставка элемента). Рассмотрим пример вставки второго элемента со значением. Схематично вставка элемента показана на рис. 3. Изначально полю **next** нового элемента присваивается ссылка на узел, который был ранее под номером 2, а потом полю **next** первого элемента присваивается ссылка на новый узел (именно в такой последовательности, иначе список будет разорван).

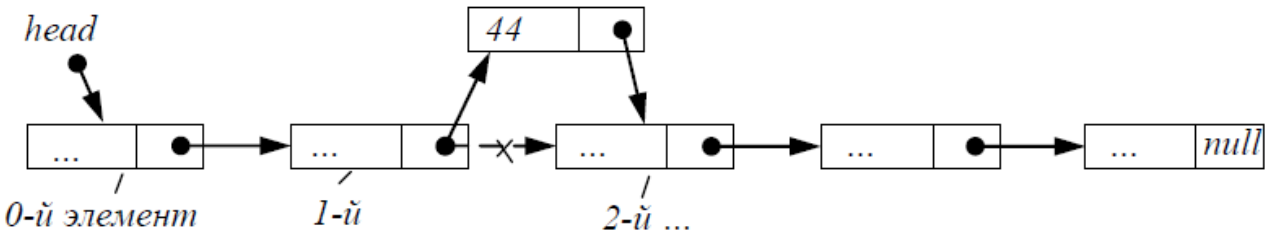


Рисунок 3

Фрагмент кода вставки элемента в список.

```
// создается новый элемент со значением 44 – для вставки
  Node newNode=new Node(44, null);
```

```
    ref = head;   // используем временную переменную
    int k=1;      // счетчик элементов
// поиск нужного положения узла для вставки
   while (ref.next!= null && (k<2 )) {
      ref = ref.next;
      k++;
    }

// переброска ссылок при вставке элемента
   newNode.next=ref.next.next;
   ref.next=newNode;
```

Удаление элемента с начала списка (с головы). Эта задача такая же простая, как и добавление элемента с головы, так как указатель находится на начальном элементе. Новой головой становится ее ссылка на следующий элемент.

```
    head=head.next;
```

Удаление последнего элемента списка (с хвоста). Для его удаления необходимо переместиться в конец списка на предпоследний элемент. Ниже приведен фрагмент кода.

```
// создаем вспомогательную переменную
    ref = head;
// перемещаемся на предпоследний элемент
    while (ref.next.next != null) {
      ref = ref.next;
    }
// полю next предпоследнего элемента присваиваем null
   ref.next=null;
```

Удаление элемента с заданным номером из списка. Для этого ссылку элемента, предшествующего удаляемому, необходимо перебросить на элемент, расположенный после удаляемого.

Рассмотрим пример удаления второго элемента. Схематично удаление элемента показано на рис. 4, где полю **next** первого элемента задается указатель на третий элемент.

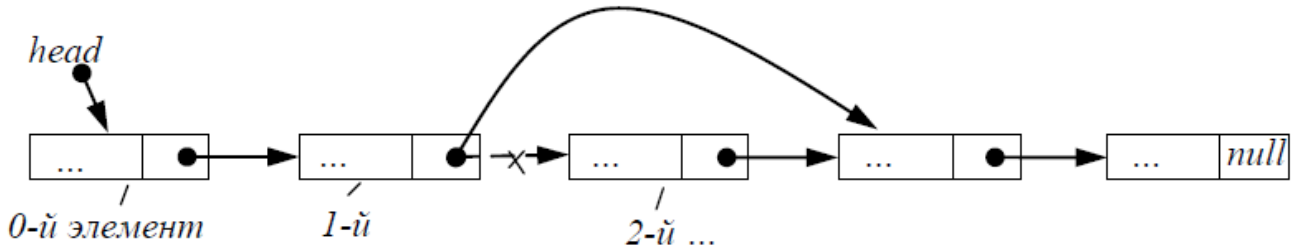


Рисунок 4

Ниже приведен фрагмент кода удаления второго элемента.

```
ref = head; // создаем вспомогательную переменную
k=1;
 // поиск положения узла, предшествующего удаляемому
while (ref.next!= null && (k<2 )) {
   ref = ref.next;
   k++;
}
 // переброска ссылки для исключения ненужного элемента из списка
 ref.next=ref.next.next;
```

Одним из достоинств языка Java является то, что очищать память после потери ссылки на объект не нужно – это будет сделано автоматически.

3 ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. Создать приложения для демонстрации примеров 1 – 5 из раздела 1. Для примера 5 дополнительно вывести последовательность обхода дерева рекурсивных вызовов. Отработать код с помощью отладчика и привести скриншоты минимум трех точек обработанных отладчиком.

2. Создать приложение с использованием рекурсии для перевода целого числа, введенного с клавиатуры, в двоичную систему счисления.

3. Создать приложение, позволяющее ввести и вывести одномерный массив целых чисел. Для ввода и вывода массива разработать рекурсивные методы вместо циклов for.

4. Выполнить пример 1 из раздела 2. Отработать код с помощью отладчика и привести скриншоты минимум трех точек обработанных отладчиком.

5. Создать два проекта, в которых продемонстрировать два способа создания линейного однонаправленного списка (с головы и с хвоста) согласно примеру 2 из второго раздела. Отработать код с помощью отладчика и привести скриншоты минимум трех точек обработанных отладчиком.

6. Разработать проект, в котором для ввода, вывода и изменения односвязного линейного списка создать следующие методы:

а) с использованием цикла:

- ввод с головы **createHead**();
- ввод с хвоста **createTail**();
- вывод (возвращается строка, сформированная из элементов списка) **toString**();
- добавление элемента в начало списка **AddFirst**();
- добавление элемента в конец списка **AddLast**();
- вставка элемента в список с указанным номером **Insert**();
- удаление элемента с головы списка **RemoveFirst**();
- удаление последнего элемента списка **RemoveLast**();
- удаление из списка элемента с указанным номером **Remove**();

а) с использованием рекурсии:

– ввод с головы **createHeadRec**();
– ввод с хвоста **createTailRec**();
– вывод (возвращается строка, сформированная из элементов списка) **toStringRec**().

Дополнительное задание. Разработать проект согласно любому варианту (табл. 1). Для вычислений требуемых величин использовать методы.

Таблица 1

Вариант 1
Ввести с клавиатуры список из n целых чисел. а) Найти сумму, количество и среднее значение среди чисел, делящихся на 3.
б) Найти среди чисел, которые делятся на 3, минимальное и максимальное значения и их номера и поменять их местами.
Вариант 2
Ввести с клавиатуры список из n целых чисел. а) Найти сумму, количество и среднее значение среди чисел, которые не делятся на 5.
б) Найти среди отрицательных чисел, которые не делятся на 5, минимальное и максимальное значения и их номера и поменять их местами.
Вариант 3
Ввести с клавиатуры список из n целых чисел. а) Найти сумму, количество и среднее значение среди чисел, делящихся на 7.
б) Найти среди положительных чисел данного массива минимальное и максимальное значения и их номера и поменять их местами.
Вариант 4
Ввести с клавиатуры список из n целых чисел. а) Найти сумму, количество и среднее значение среди чисел, которые не делятся на 9.
б) Найти среди отрицательных чисел, которые не делятся на 7, минимальное и максимальное значения и их номера и поменять их местами.
Вариант 5
Ввести с клавиатуры список из n целых чисел. а) Найти сумму, количество и среднее значение среди четных чисел.
б) Найти среди четных положительных чисел данного массива минимальное и максимальное значения и их номера и поменять их местами.

4 ОПИСАНИЕ РЕЗУЛЬТАТА ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

В отчете по лабораторной работе должны быть представлены 6 проектов.

Лабораторная работа принимается при наличии отчета и шести работающих, верно, выполненных проектов.

Структура отчета по лабораторной работе:

1. Титульный лист;
2. Цель работы;
3. Описание задачи;
4. Ход выполнения (содержит код программы);
5. Вывод;

Оформление:

а) шрифт Times New Roman;

б) размер шрифта 12 или 14;

в) межстрочный интервал 1,5.

Отчет выполняется индивидуально и направляется по адресу электронной почты proverkalab@yandex.ru. В теле письма необходимо указать ФИО студента и номер группы.