МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ВОЛОГОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра автоматики и вычислительной техники

ПОСТРОЕНИЕ И АНАЛИЗ АЛГОРИТМОВ

Методические указания к лабораторным занятиям и курсовой работе

Факультет электроэнергетический

Направления:

09.03.02 – Информатика и вычислительная техника

27.03.04 – Управление в технических системах

Вологда
2014

УДК 621.374.1

Построение и анализ алгоритмов: методические указания к лабораторным занятиям и курсовой работе. – Вологда: ВоГТУ, 2014. – 35 с.

Методические указания предназначены для выполнения лабораторных работ по дисциплине «Построение и анализ алгоритмов» и служат для успешного освоения данного предмета и получения навыков решения разных типов задач. Для каждой лабораторной работы приводятся краткие теоретические сведения, разбираются примеры и предлагаются задания для самостоятельного выполнения. Также методические указания содержат пояснения к выполнению курсовой работы.

Утверждено редакционно-издательским советом ВоГУ

Составитель И.А. Андрианов, канд. техн. наук, доцент

Рецензент А.Н. Швецов, д-р техн. наук, профессор

Введение

Методические указания предназначены для студентов направлений 09.03.02– Информатика и вычислительная техника и 27.03.04 – Управление в технических системах, но могут быть полезны студентам и других специальностей, где изучается аналогичная дисциплина.

Методические указания разработаны для поддержки курса «Построение и анализ алгоритмов» (второй семестр) и содержат 9 четырёхчасовых лабораторных работ, которые предназначены для приобретения умений и навыков по разработке эффективных алгоритмов и структур данных для решения различных типов задач, а также по выполнению анализа алгоритмов.

Следует понимать, что полноценное освоение предмета невозможно без серьезной самостоятельной работы. Поэтому в дополнение к лабораторному практикуму студенты выполняют курсовую работу, которая состоит в разработке полноценного приложения с графическим интерфейсом, демонстрирующего работу выбранного алгоритма или структуры данных. Тематика заданий курсовых работ может быть несколько шире, чем рассматриваемый на лабораторных работах материал, и требует самостоятельной работы с литературой.

Лабораторная работа 1

Алгоритмы сортировки данных

Сортировкой последовательности называется расположение элементов этой последовательности согласно определённому линейному отношению порядка. Наиболее привычным является отношение "≤", в этом случае говорят о сортировке по неубыванию.

Существует большое число различных методов сортировки. В данной лабораторной работе рассматриваются два метода – сортировка простыми вставками и "быстрая" сортировка (метод Хоара).

Сортировка вставками относится к целому классу методов, которые чрезвычайно просты в понимании и реализации, но имеют более высокий порядок сложности, чем более совершенные алгоритмы. Пусть первые (i-1) элементов последовательности уже отсортированы. На очередном шаге возьмём элемент, стоящий на i-м месте, и поместим его в нужное место отсортированной части последовательности. Будем делать так до тех пор, пока вся последовательность не окажется отсортированной.

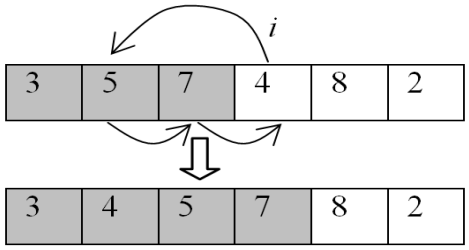

Рис. 1. Принцип работы сортировки вставками

Один из способов вставки элемента будет выглядеть так. Запомним i-й элемент в переменной t. Будем двигаться от (i-1)-го элемента к началу массива, сдвигая элементы на 1 вправо, пока они больше t. Наконец, поместим элемент t на освободившееся место. Иллюстрация показана на рисунке 1.

Быстрая сортировка Хоара является реализацией принципа “разделяй и властвуй”. Элементы сортируемого массива переставляются так, чтобы массив условно разделился на две части – левую и правую, причём никакой элемент из левой части не должен превосходить никакого элемента из правой.

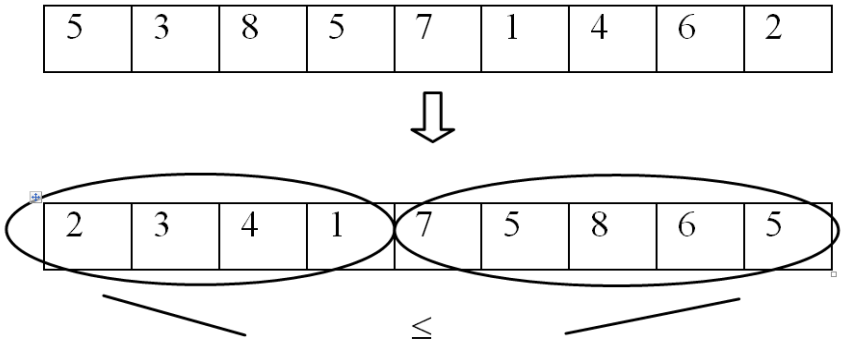

Рис.2. Принцип работы быстрой сортировки Хоара

Для получения такого разбиения можно действовать следующим образом. Пусть дан массив a[p..r]. Элемент m=a[(p+r) div 2] выбирается в качестве граничного (опорного).

Разбиение производится следующим образом. Двигаясь от начала массива к концу, найдём элемент ≥ x. Затем, двигаясь от конца к началу, найдём элемент ≤ x. Поменяем эти элементы местами. Будем продолжать действовать таким образом, пока не встретимся где-то внутри массива.

После этого процедура сортировки вызывается рекурсивно для левой и правой части, в результате чего массив будет отсортирован.

Пример программной реализации приведён ниже.

```
var a : array [1..N] of integer;

procedure QSort(left, right : integer);
var m, i, j, t : integer;
begin
  m := a[(left+right) div 2];
  i := left; j := right;
  repeat
    while a[i] < m do inc(i);
    while a[j] > m do dec(j);
    if i <= j then
    begin
      t := a[i]; a[i] := a[j]; a[j] := t;
      inc(i); dec(j);
    end;
  until i > j;

  if j > left then QSort(left, j);
```

```
  if i < right then QSort(i, right);
end;

begin
  ...
  QSort(1, N);
end.
```

Самостоятельные задания

1. Объявите константу MAXLEN = 1000000 – это будет максимально возможная длина массива. Объявите новый тип данных – массив целых чисел в диапазоне от 1 до MAXLEN.

2. Напишите процедуру, которая заполняет массив случайными числами в заданном диапазоне. Аргументы процедуры – массив, его длина, левая и правая границы диапазона. *Примечание.* При написании этой и последующих процедур сами определите, какие аргументы передавать по значению, какие по ссылке, а какие – по константной ссылке.

3. Напишите процедуру, которая выводит массив на консоль. Аргументы процедуры – массив и его длина.

4. Напишите основную часть программы, в которой вызываются эти процедуры. Проверьте, что всё работает верно. Попробуйте создавать массивы разного размера (10, 100, 1000, … 1000000 элементов).

Примечание: для больших массивов не надо выводить на консоль весь массив целиком – достаточно вывести его первые 50 элементов.

5. Напишите процедуру, выполняющую сортировку массива с помощью алгоритма сортировки простыми вставками. Процедура имеет 2 параметра: массив и его длина.

Добавьте в основную часть программы сортировку массива и вывод результата, проверьте работу программы.

6. Замерьте время выполнения сортировки на случайных массивах из 1000, 10000, 30000, 100000 элементов. Пример программы с замером времени лежит в папке «Построение и анализ алгоритмов». Разберитесь с этим примером и по аналогии сделайте то же в своей программе. Результаты замеров сохраните в файле с отчётом.

7. Постройте график времени выполнения сортировки случайного массива от размера входных данных.

8. Рассчитайте, сколько времени ваша программа будет сортировать массив из одного миллиона элементов и 10 миллионов элементов.

Примечание. Для этого оцените, как зависит количество действий от размера входных данных и во сколько раз увеличится количество действий (и, соответственно, время работы программы), если размер массива увеличился в 10 раз.

Подсказка: общий вид зависимости можно понять из графика (особенно, если взять большое число измерений).

9. Напишите процедуру, выполняющую сортировку массива методом Хоара («быстрая сортировка»).

10. Замерьте время выполнения быстрой сортировки на случайных массивах из 1000, 10000, 300000, 100000 элементов. Результаты замеров сохраните. Постройте график.

11. Рассчитайте, сколько времени теперь будет сортироваться случайный массив из одного миллиона элементов и 10 миллионов элементов. Проверьте это, запустив программу.

12. Замерьте теперь время сортировки обоими методами для изначально упорядоченного массива. В отчёте укажите соответствующие выводы.

13. Начертите блок-схемы всех алгоритмов

14*. Решите задачу №4 из проверяющей системы.

Лабораторная работа 2

Двоичный поиск

Пусть дан упорядоченный по неубыванию массив a[left..right]. Требуется найти в нём позицию элемента k (если их несколько, то любую). Это можно эффективно сделать следующим образом. Возьмём элемент из середины массива (он найдётся как a[(left+right) div 2]) и сравним его с x. Если элементы равны, то поиск закончен. Если x меньше элемента из середины, то очевидно, что ответ может быть только левее (ведь массив отсортирован), а в противном случае - только правее. То есть мы можем отбросить половину массива из рассмотрения и продолжить поиск в оставшейся части, используя те же самые рассуждения. Описанный алгоритм несложно реализовать программно.

```
const
  NMAX = 1000;

type
  TKey = longint;
  TArray = array[1..NMAX] of TKey;

function bsearch(const a: TArray; left, right: longint; k:
TKey): longint;
var
  m: TKey;
begin
  while left <= right do
  begin
    m := (left + right) div 2;
    if a[m] = k then begin
```

```
      bsearch := m;
      exit;
    end;
    if k < a[m] then
      right := m - 1
    else
      left := m + 1;
  end;
  bsearch := -1;
end;
```

Выполним оценку вычислительной сложности. Пусть изначально размер массива равен N элементов. На каждой итерации цикла размер уменьшается примерно вдвое, пока не останется один элемент, то есть $N / 2^x \approx 1$. Нам нужно число итераций x, оно найдётся как $x \approx \log_2(N)$, то есть асимптотическая сложность алгоритма составляет O(log(N)).

Метод бинарного поиска может использоваться и для других целей – например, для численного решения уравнений. Пусть дана непрерывная строго монотонная функция f(x), которая на отрезке [a, b] пересекается с осью OX. Требуется найти точку пересечения с осью, то есть решить уравнение f(x)=0.

Пусть c = (a+b)/2 - середина отрезка [a, b]. Вычислим f(c). Если $f(a) \cdot f(c) > 0$, т.е. значения функции на концах отрезка [a, c] одинаковы по знаку, то корень уравнения находится на отрезке [c, b]. В этом случае отрезок [a, c] можно исключить из дальнейшего рассмотрения и перенести точку a в точку c. В противном же случае точка b переносится в точку c. Описанный способ иллюстрируется рисунком 3.

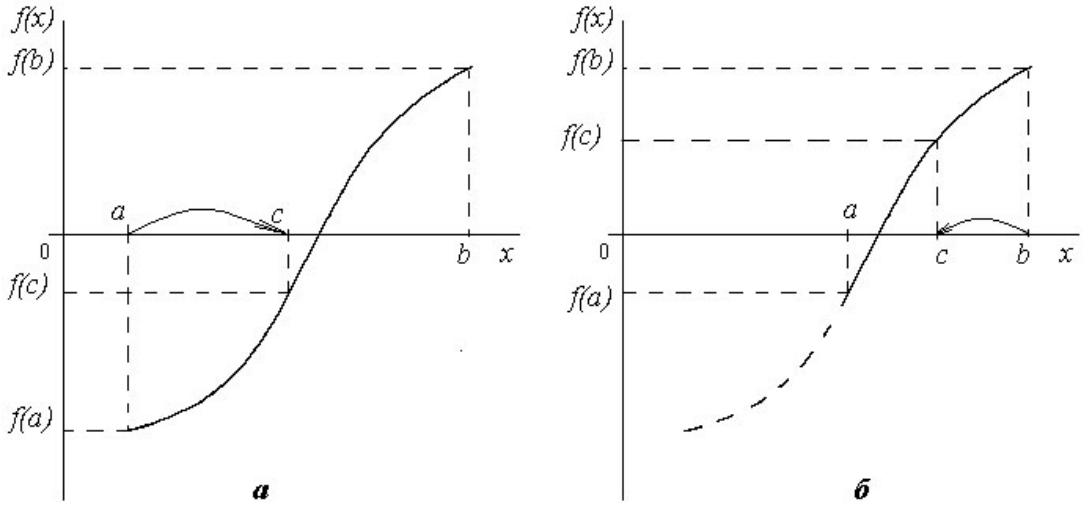

Рис. 3. Решение уравнения f(x)=0 методом дихотомии [11]

Процесс прекращается, когда длина отрезка [a, b] становится меньше требуемой точности.

Самостоятельные задания

1. Решите задачу №985 из проверяющей системы. *Подсказка*: временная сложность поиска должна быть по-прежнему O(log(N)).

2. Решите задачу №984 из проверяющей системы.

3. Решите задачу №868 из проверяющей системы. *Пояснения к решению*. В качестве левой границы искомой длины можно взять 0, в качестве правой – длину самой большой доски. Получив очередную серединку, подсчитаем, сколько досок такой длины мы сможем выпилить (рекомендуется данный подсчёт реализовать отдельной функцией). Если полученное количество оказалось меньше K, то нужно сдвинуть правую границу к середине, а если больше – левую.

4*. Решите задачу №1613 с сайта acm.timus.ru.

5*. Решите задачу №1064 с сайта acm.timus.ru.

Лабораторная работа 3

Связные списки

Структура данных (в узком смысле) – это способ представления данных и связей между данными в памяти машины. В *динамической* структуре данные и связи могут создаваться и изменяться динамически, то есть по ходу работы программы.

Чаще всего элементами данных являются *записи* (record), а связи между записями реализуются с помощью *указателей*.

Простейшим вариантом динамической структуры данных является односвязный список – см. рисунок.

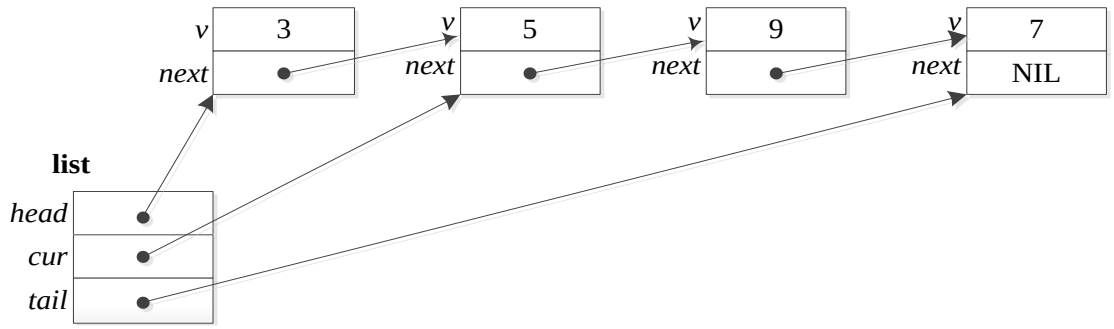

Рис. 4. Связный список

Список состоит из некоторого количества записей, где каждая предыдущая с помощью указателя ссылается на следующую. В последней записи указатель содержит значение NIL (нулевой адрес).

Для получения доступа к элементам списка создана переменная-запись list, содержащая 3 указателя: head, tail и cur, где:

- head – адрес первого элемента списка
- tail – адрес последнего элемента списка
- cur – адрес текущего элемента списка

Подобная структура находит применение в достаточно многих задачах. В данной лабораторной работе вы выполните её машинную реализацию и примените для решения задач в проверяющей системе.

Часть 1. Объявление типов, создание и инициализация переменных

Создайте в Delphi (или другой версии языка Pascal) новое консольное приложение, сохраните проект.

Вначале объявим типы данных. Нам понадобится 2 типа записей – один тип для элементов списка и один для переменной list.

Теперь рассмотрим элемент списка. Эта запись должна иметь 2 поля – v (значение) и next – адрес следующего. Её тип создаётся следующим образом:

```
type
  PElem = ^TElem;
  TElem = record
    v: integer;
    next: PElem;
  end;
```

Запись для переменной list должна просто содержать 3 указателя – head, cur и tail:

```
  TLinkList = record
   head, cur, tail: PElem;
  end;
```

Нам потребуется пока один список, поэтому создадим одну глобальную переменную list:

```
 var
    list: TLinkList;
```

Delphi не гарантирует инициализацию глобальных переменных нулями, и поля переменной list при запуске программы теоретически могут содержать какие-то случайные адреса. Поэтому сразу после основного begin в программе вставьте следующую инициализацию:

```
list.head:=nil; list.tail:=nil; list.cur := nil;
```

Часть 2. Разработка процедур и функций, реализующих типовые действия над списком

Типичные действия над списком включают:

- вставить новый элемент после текущего
- удалить элемент, стоящий после текущего
- сделать текущим элементом тот, что стоит после текущего (т.е. сдвинуться к следующему).

Рассмотрим процедуру вставки нового элемента. Будем передавать процедуре 2 аргумента – список, в который будем вставлять, и вставляемое значение. Для универсальности напишем процедуру так, что если cur=NIL (то есть текущего элемента нет), то вставка пойдёт в начало списка. Приведём код процедуры.

```
//вставить элемент после текущего (если cur=nil, то в начало списка)
procedure ins(var list: TLinkList; v: integer);
var
  p: PElem;
begin
  with list do begin
    new(p);
    p^.v:=v;
    if cur=nil then begin //выполняем вставку в начало списка
      p^.next := head;
      head := p;
      if tail=nil then tail:=p; //это если список был пустой
      exit;
    end;
    //вставка после текущего элемента
    p^.next := cur^.next;
    cur^.next := p;
    if p^.next = nil then tail:=p; //это если вставляли после последнего
  end;
end;
```

Теперь рассмотрим процедуру удаления элемента после текущего. Процедуре передаётся один аргумент – список, из которого удаляем. Для большей универсальности напишем процедуру так, что если cur=NIL (текущего элемента нет), то удалится первый элемент списка. Приведём код процедуры.

```
//удалить элемент после текущего (а если cur=nil, то первый)
procedure del(var list: TLinkList);
var
```

```
  p: PElem;
begin
  with list do begin
    if head=nil then exit; //нечего удалять, список пуст
    if cur=nil then begin //выполняем удаление первого
элемента
       p:=head;
       head:=head^.next;
       dispose(p);
       if head=nil then tail:=nil;
       exit;
    end;
    //удаление элемента за текущим
    if cur^.next=nil then exit; //нечего удалять
    p:=cur^.next;
    cur^.next:=p^.next;
    dispose(p);
    if cur^.next=nil then tail:=cur;
  end;
end;
```

Самостоятельные задания

1. Напишите процедуру **print**, которая выводит список на экран, разделяя числа пробелами. Проверьте работу всех процедур: вставьте что-нибудь в список, удалите, выведите результат.

2. Напишите функцию **count**, которая возвращает количество элементов в списке. Придумайте, как сделать, чтобы функция count работала всего за несколько машинных операций независимо от длины списка

3. Напишите процедуру **reverse**, которая переворачивает список (то есть последний элемент становится первым, предпоследний – вторым и т.д.) Проверьте её работу с помощью процедуры **print**.

4. Проиллюстрируйте работу процедуру **ins** с помощью схематических рисунков. Вначале показывается исходное состояние переменных (сразу после входа в процедуру). Далее пишется очередная строчка процедуры и показывается на рисунке, что именно изменилось. Заметим, что у нас имеется 3 отдельных случая:

1) вставка в пустой список;

2) вставка в непустой список, list.cur=NIL (т.е. вставка в начало);

3) вставка в непустой список, list.cur<>NIL (общий случай).

Ниже показан пример выполнения для первого случая.

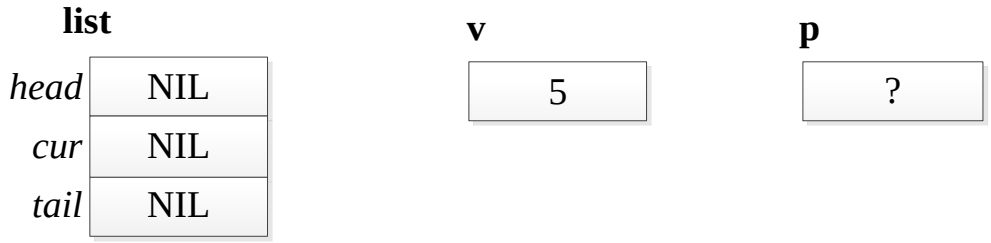

Рис.5. Состояние переменных при входе в процедуру ins

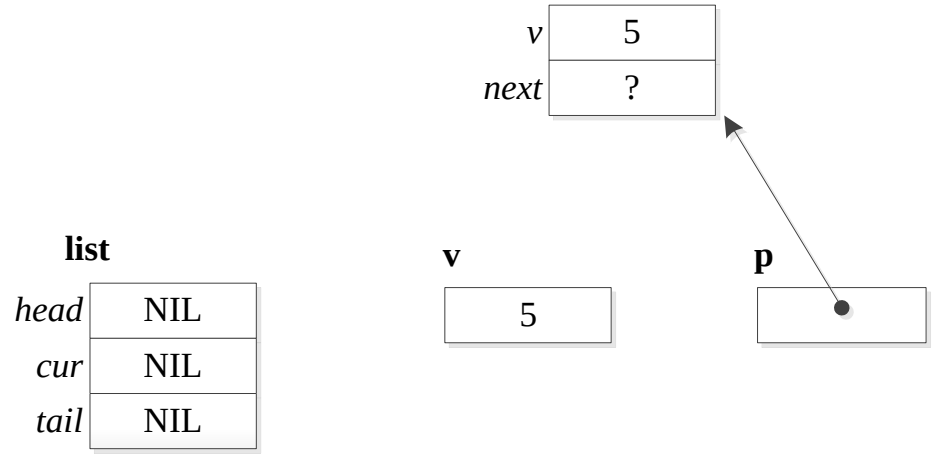

Рис.6. Состояние переменных после строк new(p); p^.v:=v;

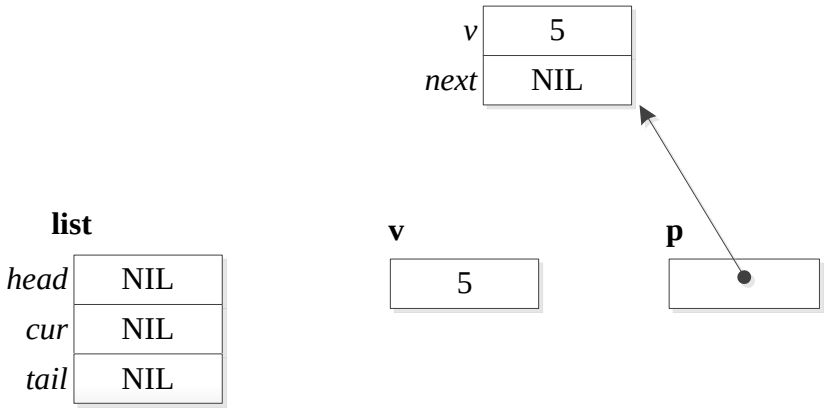

Рис.7. Состояние переменных после строки p^.next := head;

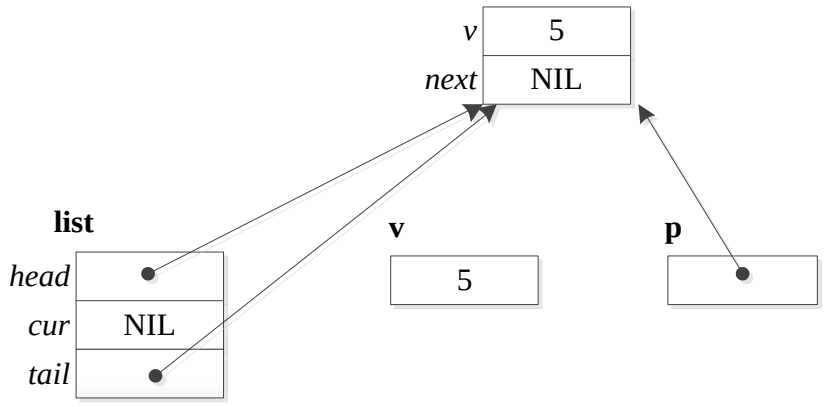

Рис.8. Состояние переменных после строк
head := p; if tail=nil then tail:=p;

Наконец, по команде exit вышли из процедуры, при этом локальная переменная p и переменная-параметр v уничтожились. Конечный результат:

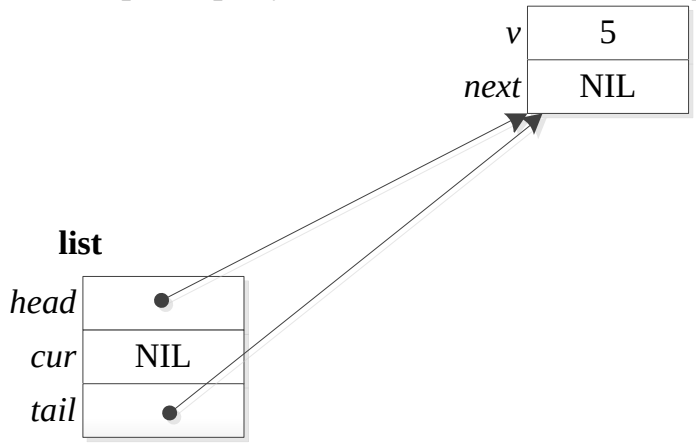

Рис. 9. Связный список после выхода из процедуры ins

Аналогичным образом изобразите 2 оставшихся случая.

5. Изобразите, как работает процедура del. Отдельно рассмотрите частный случай удаления первого элемента (cur=NIL).

6. Начертите блок-схему для задачи из задания 4.

Лабораторная работа 4

Стеки и очереди

Основные понятия. *Стек* представляет собой структуру данных – контейнер, в котором данные помещаются и извлекаются с одного и того же конца (вершины стека – англ. top), выполняется принцип LIFO (last in – first out), то есть помещенный последним элемент извлечён будет первым.

Очередь представляет собой структуру данных – контейнер, в котором данные помещаются с одного конца (англ. back), а извлекаются с другого (голова очереди – англ. front), выполняется принцип FIFO (first in – first out), то есть помещенный первым элемент извлечён будет также первым.

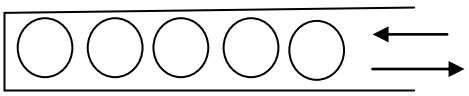

Рис. 10. Схема работы стека

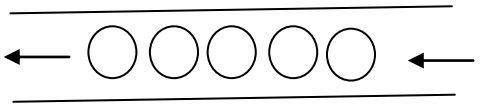

Рис. 11. Схема работы очереди

Реализация стека. Рассмотрим возможные способы реализации стека. Если максимально возможное количество элементов в стеке ограничено каким-либо значением, можно предложить самый простой способ реализации − массив. В этом случае для стека выделяется непрерывная область памяти ограниченных размеров. Рис. 12 поясняет работу со стеком на основе массива. Идея

проста − достаточно завести дополнительный индекс `top`, ссылающийся на вершину стека. Если стек пуст, то `top` может хранить значение 0.

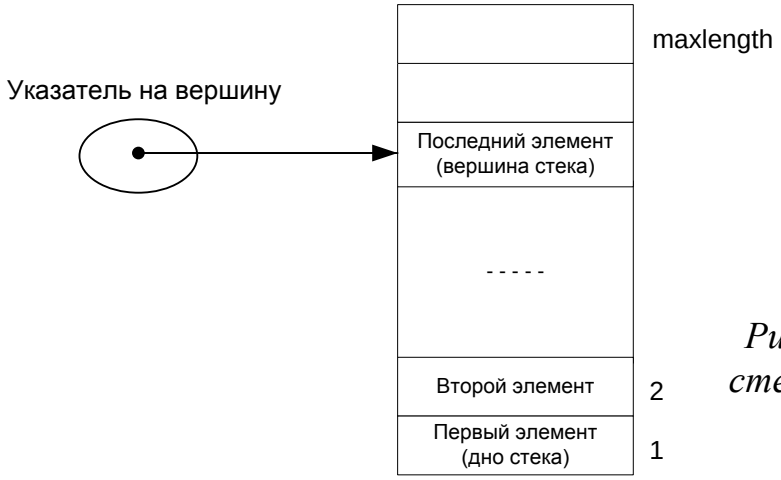

Рис. 12. Представление стека с помощью массива

Алгоритмы для реализации операций удаления и вставки элементов весьма просты и сводятся к изменению значения указателя top на единицу. При этом, конечно, нельзя забыть о крайних случаях, связанных с добавлением элемента в переполненный стек или попыткой извлечения элемента из пустого стека.

В случае, когда максимальное число элементов в стеке заранее неизвестно, его можно реализовать с помощью односвязного списка. Данный способ показан на рисунке 13.

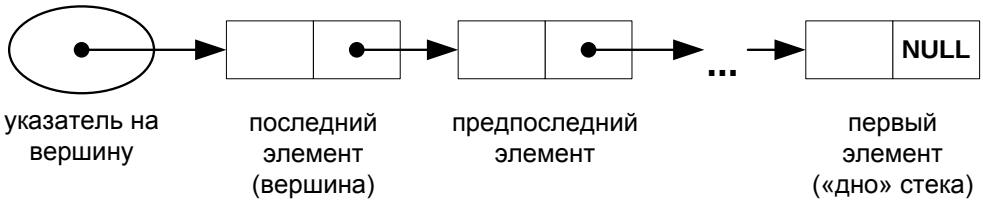

Рис. 13. Представление стека с помощью списка

Пример применения стека. Стек используется в большом числе алгоритмов. Одним из них является вычисление выражений в постфиксной записи. Смысл постфиксной (обратной польской) записи арифметического выражения состоит в том, что вначале пишутся аргументы, а после них – знак операции. Например, запись "2 3 + 4 *" означает, что нужно сложить 2 и 3, затем полученный результат умножить на 4. Плюсы постфиксной записи – удобство машинного вычисления, не требуются скобки.

Рассмотрим, каким образом вычисляется выражение в постфиксной записи с применением стека. Алгоритм выглядит так. Берём очередную входную лексему. Если это число, то кладём его в стек. Если же это знак операции, то извлекаем два элемента из стека, выполняем операцию, результат кладём в стек. По окончании входных данных на вершине стека будет лежать ответ.

Реализация очереди. Реализация очереди при помощи массива немного сложнее, чем реализация стека. Вспомним, что вставка и удаление элементов выполняются с разных концов. Поэтому используют два индекса `head` и `tail`, ссылающиеся на голову и хвост очереди.

При вставках и удалениях элементов очередь как бы передвигается по массиву, постепенно приближаясь к его границе. При этом в начале массива появляется свободное пространство за счет освободившихся при удалении элементов. Как только хвост очереди достигнет верхней границы массива, следующие элементы будут добавляться уже в свободные ячейки в начале массива. Такую реализацию очереди называют *кольцевой* или *циклической*.

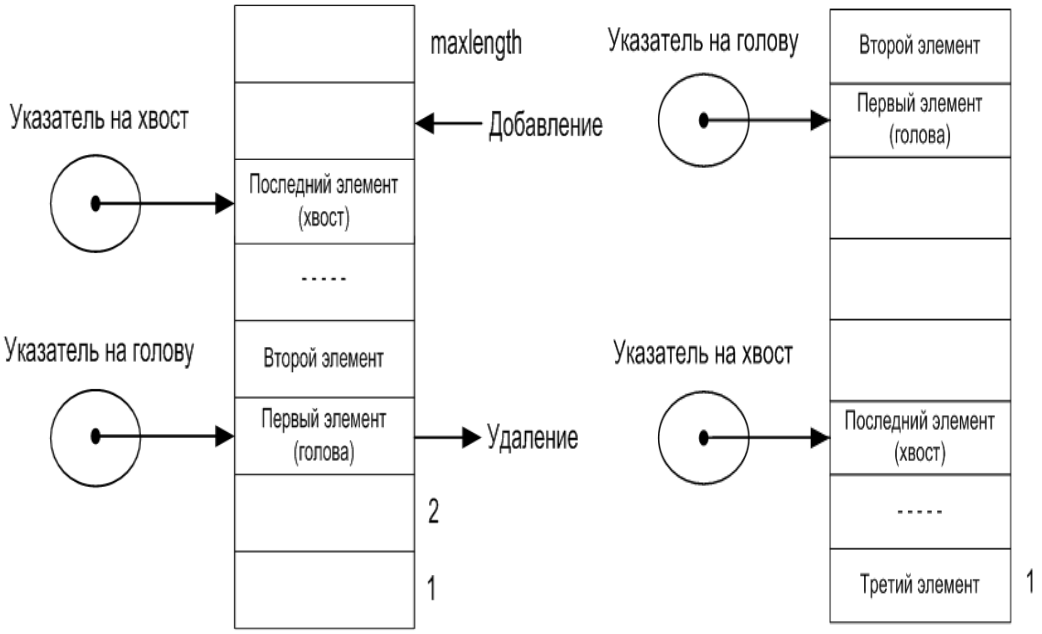

Рис. 14. Положение головы и хвоста очереди в разные моменты времени

Заметим, что только по двум индексам head и tail невозможно отличить пустую очередь от полностью заполненной. Чтобы устранить данный недостаток, достаточно создавать массив хотя бы на один элемент больше, чем максимально возможная длина очереди (либо просто хранить в отдельной переменной текущее число элементов).

В случае, когда максимальное число элементов в очереди заранее неизвестно, её можно реализовать с помощью односвязного списка. Поскольку вставка и удаление выполняются на разных концах, то для эффективной реализации нужно хранить два дополнительных указателя − на хвост и голову очереди.

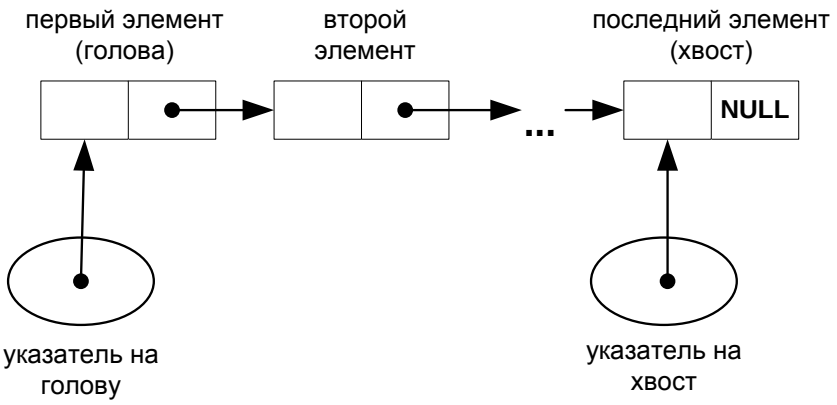

Рис. 15. Реализация очереди на односвязном списке

Применение очередей. Очереди, как и стеки, используются весьма широко − например, очередь сообщений в операционной системе Windows, очередь сетевых пакетов в маршрутизаторе и др.

Самостоятельные задания

1. Реализуйте стек на базе массива.
2. Реализуйте стек на базе односвязного списка.
4. Реализуйте очередь на базе циклического массива.
4. Реализуйте очередь на базе односвязного списка.
5. Решите задачу №862 из проверяющей системы.
6. Решите задачу №863 из проверяющей системы.
7. Решите задачу №248 из проверяющей системы.
8. Решите задачу №226 из проверяющей системы.
9. Решите задачу №244 из проверяющей системы.
10*. Решите задачу №246 из проверяющей системы.

Лабораторная работа 5

Бинарные деревья поиска

Примером иерархической структуры является бинарное поисковое дерево – см. рисунок 16. Каждый узел дерева имеет от 0 до двух детей (левого сына и правого сына). Значения (числа v) в узлах дерева расположены так, что для любого узла r выполняется следующее требование: значения во всех узлах левого поддерева меньше, чем значение в r, а значения в узлах правого поддерева – больше, чем в r. Такое дерево применяется для представления множеств в памяти ЭВМ. Рассмотрим особенности его машинной реализации.

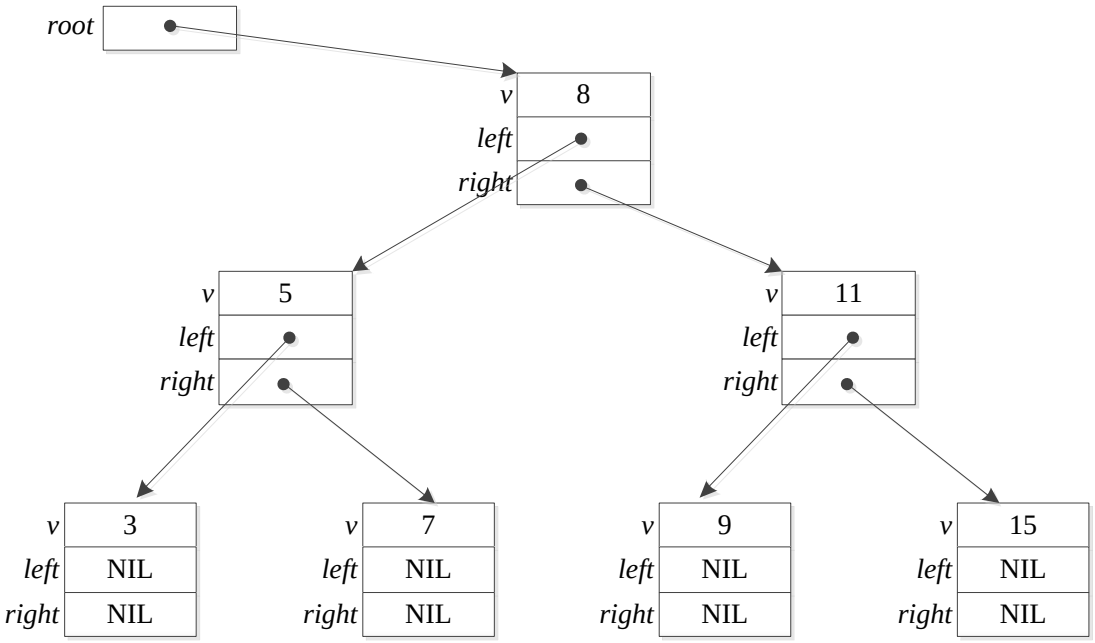

Рис. 16. Двоичное дерево поиска

Типы данных и инициализация переменных. Узлом дерева будет запись из трех полей – одно числовое и 2 указателя. Начало программы на данном этапе будет выглядеть так:

```
type
  PNode = ^TNode;
  TNode = record
    v: integer;
    left, right: PNode;
  end;

var
  root: PNode;

begin
  root:=NIL;
end.
```

Основные операции. ***Вставка*** элемента x в дерево начинается с корневого узла дерева (его адрес записан в root). Сравниваем x со значением v в этом узле. Если они равны, то мы нашли x и делать ничего не надо – выходим из процедуры. Если x меньше v, то переходим по левому указателю (left), если больше – по правому (right). Продолжаем делать так до тех пор, пока указатель, по которому надо переходить, не окажется NIL. В этом случае создаём новый узел и его адрес помещаем в этот указатель.

Не следует забывать ещё один нюанс. Если элемент вставляется в пустое дерево, то адрес созданного узла нужно записать в переменную root.

Приведём пример реализации без использования рекурсии.

```
//вставка элемента в двоичное дерево
//root - адрес корневого узла, x - вставляемый элемент
procedure ins(var root: PNode; x: integer);
var
  r: PNode; //адрес  текущего  узла дерева
begin
  //вначале проверим частный случай, когда дерево пусто
  if root=NIL then begin
    new(root);
    root^.left:=NIL; root^.right:=NIL;  root^.v:=x;
    exit;
  end;
  r:=root; //начинаем спускаться от корня
  repeat
    if r^.v=x then exit; //элемент уже есть в дереве
    if x<r^.v then  //надо идти влево
    begin
      if r^.left<>NIL then
        r:=r^.left  //узел слева есть - идём в него
      else
      begin
        //слева ничего нет - создаём новый узел
        new(r^.left);
        r:=r^.left;
        r^.left:=NIL; r^.right:=NIL; r^.v:=x;
        exit;
      end
    end
    else
    begin //надо идти вправо
      if r^.right<>NIL then
        r:=r^.right  //узел справа есть - идём в него
      else
      begin
        //справа ничего нет - создаём новый узел
        new(r^.right);
        r:=r^.right;
        r^.left:=NIL; r^.right:=NIL; r^.v:=x;
        exit;
      end;
    end;
  until false;
```

```
end;
```

Удаление – несколько более сложная процедура, чем вставка. Пусть нам надо удалить элемент x. Найдём узел r, который его содержит. Если такого узла найти не удалось, то и делать ничего не надо.

Если у узла r нет сыновей, то всё хорошо: уничтожаем узел и пишем NIL в указатель, который на него ссылался.

Если у узла только один сын, то тоже всё несложно: уничтожаем узел, а в указатель, который на него ссылался, заносим адрес сына.

Наконец, что делать, если у удаляемого узла r два сына? В этом случае нам надо найти в его правом поддереве узел с самым маленьким элементом. Это несложно: берём правого сына и идём от него влево, пока не найдём узел q, у которого левого сына нет. Он и будет искомым. Теперь значение из q запишем в узел r, а удалять будем уже не r, а q. Как это делается, см. выше.

Приведём пример реализации (для удобства он сделан в виде рекурсивной процедуры. Для экономии места в стеке внутренняя рекурсивная процедура вложена во внешнюю и используется её параметр x и локальные переменные):

```
//Удалить элемент x из двоичного дерева
//root-корень дерева, x - удаляемый элемент
procedure del(var root: PNode; x: Integer);
var
  p,q: PNode;

  //удалить элемент x из поддерева с корнем в r
  procedure del_rec(var r: PNode);
  begin
    if r=NIL then exit; //элемента x в множестве не нашлось
    if x<r^.v then begin del_rec(r^.left); exit; end;
    if x>r^.v then begin del_rec(r^.right); exit; end;
    //если у узла 0 или 1 ребёнок
    if (r^.left=NIL) or (r^.right=NIL) then
    begin
      p:=r; //запомним адрес узла во временной переменной
      //в указатель из родителя запишем адрес единственного
ребёнка
      if r^.left=NIL then r:=r^.right else r:=r^.left;
      dispose(p); //освободим память, занимаемую узлом
      exit;
    end;
    //у узла есть оба сына
    p:=r^.right;
    //частный случай: у узла, на который указывает p, нет ле-
вого сына
    if p^.left=nil then
```

```
    begin
      r^.v:=p^.v;
      r^.right:=p^.right;
      dispose(p);
      exit;
    end;
    //идём влево, пока есть куда
    q:=p^.left;
    while q^.left<>NIL do begin
      p:=q; q:=q^.left;
    end;
    //значение из q^ кладём в r^, и удалять будем уже не r^,
а q^
    r^.v:=q^.v;
    p^.left:=q^.right;
    dispose(q);
  end;
begin
  del_rec(root);
end;
```

Вопросы для самопроверки

1. Почему в процедуре ins аргумент root передаётся через var? Как называется такой способ передачи? В чем будет ошибка и как она проявится, если var убрать? Аналогичный вопрос для процедуры del.
2. Объясните использование указателей p и q в конце процедуры del.
3. Если в дереве N элементов, какая у него может быть максимальная и минимальная высота (высота – это количество рёбер на пути от корня до самого дальнего листа)
4. Если у дерева высота h, какое у него может быть минимальное и максимальное число узлов?
5. Если у дерева высота h, каково максимальное количество рекурсивных вызовов может быть в процедуре del?
6. Если в дереве N листьев, сколько в нём максимум и минимум может быть узлов?
7. Если убрать вызов dispose в процедуре del, что изменится в работе программы?

Самостоятельные задания

1. Напишите функцию проверки наличия элемента x в дереве. Заголовок функции должен иметь следующий вид:

```
function existsInTree(root: PNode; x:integer): boolean;
```

2. Напишите процедуру, которая выводит на консоль все элементы дерева по возрастанию через пробел. Заголовок должен иметь следующий вид:

```
procedure printTree(root: PNode);
```

3. Напишите функцию, которая возвращает количество элементов в дереве. Заголовок функции должен иметь следующий вид:

```
function count(root: PNode): integer;
```

Подумайте, как сделать, чтобы эта функция работала всего за несколько машинных операций независимо от размера множества.

4. Пользуясь написанным кодом, решите задачу №215 в проверяющей системе.

5. С помощью рисунков проиллюстрируйте, как работает процедура вставки элемента в дерево (см. предыдущую работу). Достаточно рассмотреть случай, когда дерево не пусто.

6. С помощью рисунков проиллюстрируйте, как работает процедура удаления элемента из дерева. Рассмотрите случаи, когда удаляемый элемент находится в узле, у которого:

- меньше 2-х детей,
- 2 сына,
- 2 сына и он - корень дерева.

Лабораторная работа 6

Рандомизированные деревья поиска

Основной проблемой простых бинарных деревьев поиска является возможность вырожденных случаев. Достаточно, чтобы входные данные были изначально отсортированы, и полученное дерево, по сути, превратится в связный список. При этом вычислительная сложность возрастёт от O(log(n)) до O(n), что для многих практических задач будет неприемлемо.

Существуют различные способы, обеспечивающие поддержку бинарного дерева поиска в хорошем состоянии. Один из самых простых подходов состоит в том, чтобы просто случайно перемешать входные данные перед построением дерева. При росте количества входных данных вероятность получения "плохого" дерева стремится к нулю.

Однако, этот вариант годится лишь для случая, когда все данные известны изначально. Во многих практически важных задачах данные поступают поэлементно, и каждый элемент должен быть вставлен в дерево сразу после поступления.

Чуть более сложный подход заключается в том, чтобы немного усложнить процедуру вставки элемента в дерево. Вначале элемент вставляется абсолютно таким же образом, как описано в предыдущей работе. При этом подсчитывается, на какой глубине h он оказался. После этого берётся случайное число k от 0 до h, и элемент поднимается на k уровней вверх с помощью так называемых *вращений*. Вращением называется изменение структуры дерева, при ко-

тором заданный элемент оказывается на один уровень выше, но при этом дерево остаётся по-прежнему корректным. На следующих двух рисунках показана схема малого правого вращения, а также конкретный пример.

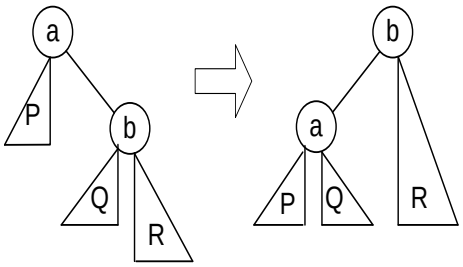

Рис.17. Схема малого правого вращения

На рисунке 17 b - это элемент, который нужно поднять, a - его родитель, P, Q и R - поддеревья.

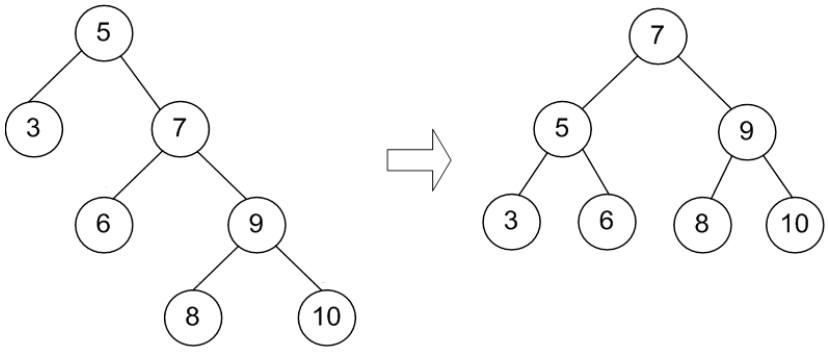

Рис. 18. Пример малого правого вращения

За счёт того, что количество вращений определяется случайно, с увеличением числа элементов вероятность "плохих" случаев стремится к нулю. Процедура вставки в рандомизированное дерево представлена ниже.

```
function add(var r: PNode; v: integer; level: integer = 0):
integer;
var
  rest, vtmp: integer;
  q, pa, pb, pc: PNode;
begin
  if r = nil then begin
    new(r);
    r^.left := nil;
    r^.right := nil;
    r^.v := v;
    add := random(level div 2);
    exit;
  end;
  if v = r^.v then begin
    add := 0;
```

```
    exit;
  end;
  if v < r^.v then begin
    rest := add(r^.left, v, level + 1);
    if rest > 0 then begin
      //левое вращение
     q := r^.left;
     pa := q^.left;
     pb := q^.right;
     pc := r^.right;
     vtmp := r^.v; r^.v := q^.v; q^.v := vtmp;
     r^.left := pa;
     r^.right := q;
     q^.left := pb;
     q^.right := pc;
    end;
    add := rest - 1;
  end else begin
    rest := add(r^.right, v, level + 1);
    if rest > 0 then begin
      //правое вращение
     q := r^.right;
     vtmp := r^.v; r^.v := q^.v; q^.v := vtmp;
     pa := r^.left;
     pb := q^.left;
     pc := q^.right;
     vtmp := r^.v; r^.v := q^.v; q^.v := vtmp;
     r^.left := q;
     q^.left := pa;
     q^.right := pb;
     r^.right := pc;
    end;
    add := rest - 1;
  end;
end;
```

Самостоятельные задания

1. Замерьте скорость вставки в рандомизированное и простое дерево для N от 1000 до 10000 с шагом 1000 на случайных и изначально отсортированных данных. Результаты занесите в таблицы.

2. Постройте диаграммы времени вставки для обоих вариантов дерева. Сравните полученные результаты и обоснуйте их.

3. Напишите процедуру, которая возвращает количество элементов в дереве, меньших заданного значения x. Процедура должна работать со сложно-

стью O(log(N)). Чтобы достичь такой сложности, подумайте, какие дополнения нужно внести в узлы дерева. Соответственно модифицируйте и процедуру вставки.

4. Добавьте в каждый узел дерева указатель на родителя. Напишите нерекурсивную процедуру вставки в рандомизированное дерево.

5. Решите задачу №251 из проверяющей системы.

6. Для двух заданных узлов дерева найдите их ближайшего общего предка. Достаточно получить сложность $O(\log^2(N))$, но при желании можно реализовать и более эффективный алгоритм.

Лабораторная работа 7

Хеш-таблицы

Основной идеей хеш-таблиц является использование ключа в качестве индекса элемента массива (таблицы). Например, при продаже билетов в кино или на концерт схему зрительного зала можно представить в виде двухмерной таблицы. Каждый ключ представляет собой пару из двух индексов (ряд и место) и однозначно задаёт элемент таблицы, в которой каждый элемент хранит логическое значение (занято/свободно). При хорошей наполняемости зрительного зала расходы памяти на хранение незанятых мест будут минимальны. Подобные таблицы называются *таблицами прямого доступа*.

На практике такие структуры встречаются редко. В большинстве случаев диапазон возможных значений ключей достаточно широк, поэтому таблица прямого доступа либо вообще не поместится в память, либо будет использовать память слишком неэффективно.

Однако сама идея использования ключа в качестве индекса элемента массива заслуживает самого пристального внимания, поскольку на ее основе может возникнуть другая − *преобразование значения ключа в индекс элемента массива* с использованием какой-либо функции, возвращающей результат в виде целого числа в заданном ограниченном диапазоне. В этом случае расход памяти становится управляемым и может быть достигнут разумный компромисс между скоростью и памятью.

Математическая функция h(K), которая преобразует значения ключей K в индексы элементов массива, называется *хеш-функцией*. Сами индексы иначе называются *хеш-адресами* и находятся в диапазоне от 0 до M-1, где M − некоторое положительное целое число. Массив размером M, в котором ведётся поиск, называется *хеш-таблицей* и обычно представляет собой массив записей (ключи и связанная информация). В частном случае элементами хеш-таблицы могут быть просто значения ключей.

Например, пусть входная последовательность ключей имеет вид: 3, 25, 7, 48, 71. Будем использовать простейшую хеш-функцию h(K) = K mod M (где операция mod означает остаток от деления). Поскольку входных данных немного, выберем M = 7. Тогда все хеш-адреса будут находиться в диапазоне от 0 до 6, а хеш-таблица будет почти заполнена (см. табл. 1)

Таблица 1. Хеш-таблица для последовательности 3 25 7 48 71
при применении хеш-функции h(k) = k mod 7

Хеш-адрес	0	1	2	3	4	5	6
Значение	7	71		3	25		48

При вставке в хеш-таблицу может возникнуть следующая проблема. Предположим, что нужно вставить в хеш-таблицу еще одно значение ключа, на этот раз равное 8. Подсчитываем значение хеш-функции: 8 mod 7 = 1. Однако ячейка с хеш-адресом 1 уже занята ключом 71. Такая ситуация иначе называется конфликтом или *коллизией.*

Существует два основных метода разрешения коллизий. Метод цепочек состоит в том, что все элементы с одинаковыми хеш-адресами объединяются в связный список (хеш-цепочку). Тогда хеш-таблица представляет собой массив из N указателей на хеш-цепочки. Такая структура показана на рис. 19.

Реализация хеш-таблицы с использованием метода цепочек довольно проста. Сами цепочки обычно реализуются в виде однонаправленных связных списков, для хранения указателей на цепочки используется обычный массив.

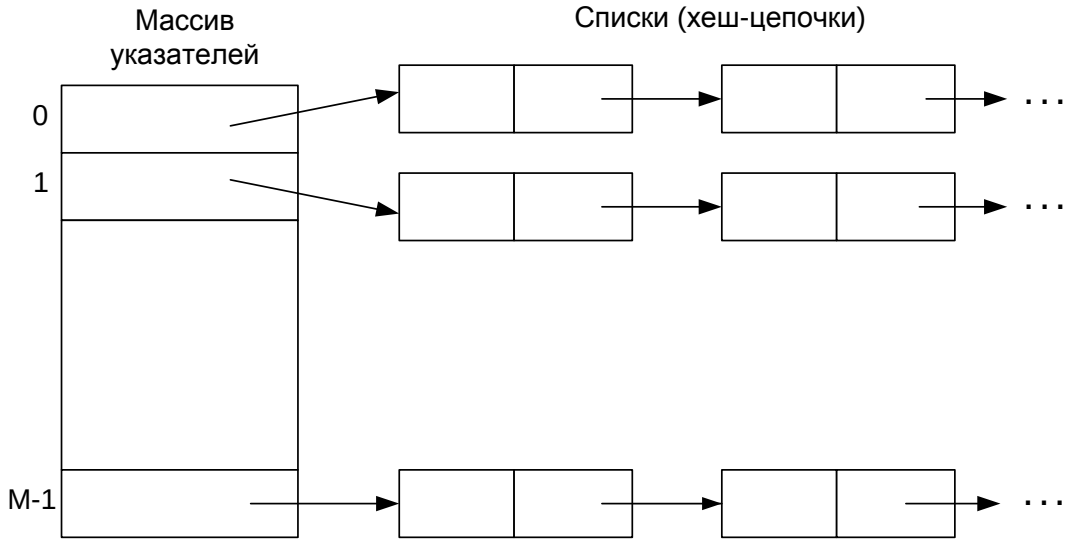

Рис. 19. Разрешение коллизий методом цепочек

Второй метод разрешения коллизий − хеширование с открытой адресацией (в некоторых источниках он называется закрытым хешированием, что вносит некоторую путаницу). Коллизии в этом случае разрешаются следующим образом. Если вычисленный по ключу хеш-адрес оказывается занятым, каким-либо способом находится другая незанятая позиция, куда и помещается новый элемент. Если все позиции заняты, то элемент вставить нельзя (место кончилось). Этот процесс поиска подходящей позиции называется *исследованием хеш-таблицы*.

Наиболее простым способом исследования хеш-таблицы является *линейное зондирование.* При этом $h_i(x)=(h(x)+i) \bmod N$. То есть если ячейка с номером i

уже занята, то проверяется ячейка с номером i+1,затем с номером i+2 и т.д., пока не найдётся свободная.

При поиске элемента x необходимо просмотреть всю последовательность, начиная с вычисленного хеш-адреса, пока не будет найден x, не встретится пустой элемент, или не будут просмотрены все элементы.

Заметим, что при удалении из хеш-таблицы освободившийся элемент необходимо помечать специальным образом, чтобы отличать от изначально пустого − иначе поиск будет работать неверно.

Самостоятельные задания

1. Реализуйте хеш-таблицу со следующими параметрами. Ключи − целые числа. Метод разрешения коллизий − цепочки переполнения. Хеш-функция h(x) = x mod M, где M – размер хеш-таблицы − простое число, примерно равное N, где N – количество ключей.

Замерьте скорость работы операций вставки, удаления и поиска для N = 1000, 10000, 100000, 1000000 для случайных и отсортированных данных, результаты занесите в таблицу.

2. Реализуйте хеш-таблицу, используя метод хеширования с закрытой адресацией. В качестве M возьмите простое число, превышающее N примерно в 2 раза. Выполните замеры времени работы, результаты занесите в таблицу.

3. Постройте диаграммы времени работы обоих вариантов хеш-таблиц. Сравните полученные результаты и обоснуйте их.

4. Выполните расчёт вычислительной сложности алгоритмов. Во сколько раз увеличивается время работы программы при увеличении размера входных данных в 10 раз? Совпадает ли теоретическая и полученная экспериментально оценки? Если нет, то в чём может быть причина?

5. Предложите хеш-функцию лучше, чем x mod M. Выполните замеры скорости вставки в оба вида хеш-таблиц с этой функцией, сравните с предыдущими результатами.

Лабораторная работа 8

Битовые массивы

Частным случаем описанных в начале предыдущей работе таблиц прямого доступа являются битовые массивы. Пусть требуется хранить в памяти компьютера множество целых чисел, элементы которого могут принимать значения от 0 до N-1 (где N – константа). Одним из простейших вариантов является создание массива a: array[0..N-1] of Boolean, где a[k]=false означает, что элемент k не принадлежит множеству, а a[k]=true – принадлежит.

Недостаток такого подхода заключается в неэффективном расходовании памяти. При больших значениях N массив в память может просто не поместиться. Однако, достаточно легко можно уменьшить требования памяти в 8 раз. Заметим, что одно значение типа Boolean занимает целый байт, тогда как на самом деле для представления логического значения достаточно одного

бита. Это связано с тем, что в языке Pascal (как и в большинстве других языков) любой тип данных не может занимать менее одного байта.

Однако в языке Pascal имеется набор операций для работы с битами, что позволяет решить данную проблему. Нам потребуются следующие операции: AND – побитовое "И", OR – побитовое "ИЛИ", XOR – побитовое "исключающее ИЛИ", NOT – побитовое инвертирование, SHL – побитовый сдвиг влево, SHR – побитовый сдвиг вправо.

Рассмотрим вначале следующую подзадачу. Пусть имеется байт x, требуется записать единицу в его бит с номером bitPos, при этом не затронув соседние биты. С этой целью нам потребуются две операции - SHL и OR. Вначале с помощью операции SHL сформируем битовую маску – двоичное число, в котором все разряды равны 0, и только в разряде с номером bitPos стоит 1. Затем выполняем побитовое "ИЛИ" данной маски и x.

```
mask := 1 SHL bitPos;
x := x OR mask;
```

Следующая таблица поясняет данные действия на примере x=73 (01001001), bitPos=2.

Таблица 2. Пример использования побитовых операций

x	01001001
mask	00000100
x OR mask	01001101

Теперь рассмотрим общий случай. Пусть имеется массив a: array[0..N-1] of Byte. Требуется установить в нём i-й по порядку бит в единицу. Каждый элемент данного массива занимает 8 битов. Номер байта, где лежит требуемый бит, найдётся как i div 8. Номер бита внутри байта найдётся как i mod 8. Далее применяем решение вышеописанной задачи.

```
const NMAX = 1000000;

type
  TArray = array[0..NMAX] of byte;

procedure Add(var a: TArray; i: longint);
var
  bytePos: longint;
  bitPos, mask: byte;
begin
  bytePos := i div 8;
  bitPos := i mod 8;
  mask := 1 shl bitPos;
  a[bytePos] := a[bytePos] or mask;
```

```
end;
```

Самостоятельные задания

1. Напишите процедуру сброса i-го бита в массиве в нуль.
2. Напишите процедуру инвертирования i-го бита в массиве.
3. Напишите функцию, возвращающую значение i-го бита в массиве.
4. Напишите процедуры, выполняющие пересечение и объединение первого множества со вторым. Используйте в решении примерно N/8 итераций циклов.
5. Напишите функцию, возвращающую количество элементов в множестве.
6*. Решите предыдущую задачу, используя в решении примерно N/8 итераций циклов.

Лабораторная работа 9

Исчерпывающий поиск

На практике встречается много таких задач, которые имеют конечное (но, возможно, очень большое) множество допустимых решений, при этом "качество" решения определяется значением какой-то функции. Требуется найти наилучшее (оптимальное) решение, при котором значение функции будет минимальным (или максимальным).

Например, пусть имеется несколько камней, известны их веса. Требуется разбить их на две кучи так, чтобы суммарные веса каждой кучи отличались как можно меньше.

Задачи такого рода называются *задачами дискретной оптимизации*. Одним из способов решения таких задач является перебор возможных вариантов решений и выбор среди них наилучшего. Однако подобные алгоритмы имеют крайне высокую вычислительную сложность и поэтому применимы лишь для очень маленьких входных данных − не более нескольких десятков элементов.

В качестве примера рассмотрим описанную выше задачу о камнях. По сути, нам требуется перебрать все способы получения первой группы, так как все предметы, не вошедшие в неё, попадают во вторую. То есть требуется выполнить перебор всех подмножеств множества из N элементов.

Часть 1. Полный двоичный перебор

Для представления подмножеств создадим массив a из N элементов, где a[i]=1, если i-й элемент входит в текущее подмножество, и 0 - если нет. Заметим, что массив a можно рассматривать как запись некоторого N-разрядного двоичного числа, где элемент a[i] содержит значение его i-го разряда. При этом каждому подмножеству из N элементов однозначно соответствует некоторое N-разрядное двоичное число. Получив последовательно в массиве a записи всех N-разрядных двоичных чисел, мы тем самым переберём все подмножества.

Будем получать числа в порядке возрастания. Для того чтобы перейти от предыдущего числа к следующему, необходимо прибавить к нему единицу. В случае двоичной записи это делается следующим образом. Движемся от

младших разрядов к старшим, пока все они равняются единице, и присваиваем им значение 0. Дойдя до первого нуля, присваиваем ему значение 1. Например, при прибавлении 1 к числу 010010111 получится число 010011000. Программная реализация не представляет сложности. Ниже представлен пример кода, который выводит на экран все перебираемые подмножества.

```
const
  NMAX = 1000;

type
  TArray = array[1..NMAX] of byte;

function NextSubset(var a: TArray; n: integer): boolean;
begin
  while a[n] = 1 do begin
    a[n] := 0;
    dec(n);
  end;
  if n < 1 then
    NextSubset := false
  else
  begin
    NextSubset := true;
    a[n] := 1;
  end;
end;

procedure printSubset(const a: TArray; n: integer);
var
  i: integer;
begin
  for i := 1 to n do write(a[i]);
  writeln;
end;


var
  a: TArray;
  n, i: longint;

begin
  read(n);
  for i := 1 to n do a[i] := 0;
  repeat
```

```
    printSubset(a, n);
  until not NextSubset(a, n);
end.
```

Поскольку количество всех подмножеств экспоненциально зависит от размера множества, то алгоритм имеет экспоненциальную сложность.

Вышеприведённый код несложно модифицировать для решения задачи о камнях. Значение a[i]=0 говорит о том, что камень идёт в кучу с номером 0, а значение a[i]=1 − в кучу с номером 1. Получив очередной вариант решения, просто сравниваем модуль разности весов куч с наилучшим найденным ранее.

Заметим, что веса обеих куч удобно не пересчитывать заново после генерации каждого подмножества, а модифицировать их непосредственно в процедуре NextSubset.

Часть 2. Ускорение полного перебора. Отсечения

Если в какой-то момент времени веса куч стали равными, то очевидно, перебор можно уже закончить. Это тривиальный пример *отсечения*. Рассмотрим теперь более сложный пример.

Пусть C0 − вес кучи с номером 0, C1 − вес кучи с номером 1. Предположим, что в какой-то момент значение (C1−C0) превышает текущую наилучшую разницу весов куч. Пусть начиная с какого-то k все элементы a[k], … , a[n] равны нулю.

Заметим, что нам нет никакого смысла перебирать следующие $2^{(n-k)} - 1$ вариантов, поскольку при этом будут изменяться только элементы a[k], …, a[n]. Изменение же этих нулей на единицы будет давать решения ещё хуже текущего, так как разность (C1−C0) при этом только вырастет.

Следовательно, мы спокойно можем данные варианты пропустить. Для этого достаточно в процедуре NextSubset начинать цикл не с n, а с (k−1).

Эта простая идея позволяет снизить время работы программы при N=40 с нескольких часов до менее одной секунды. Однако сложность алгоритма попрежнему остаётся экспоненциальной.

Часть 3. Рекурсивная реализация перебора (backrack)

Рекурсивную реализацию перебора подмножеств легко получить из следующих соображений. Все подмножества можно разбить на две группы − содержащие первый элемент и не содержащие его. Отметим, что мы не берём первый элемент, и рекурсивно переберём все подмножества из оставшихся элементов. Затем отметим факт взятия первого элемента и снова рекурсивно переберём все подмножества из оставшихся элементов. Ниже приведён пример реализации.

```
procedure subsets(var a: TArray; n: integer; i:integer = 1);
begin
  if i>n then
```

```
    printSubset (a, n)
  else
  begin
    a[i] := 0;
    subsets(a, n, i + 1);
    a[i] := 1;
    subsets(a, n, i + 1);
  end;
end;
```

Разумеется, при рекурсивной реализации также можно (и нужно) использовать отсечения.

Самостоятельные задания

1. Реализуйте решение для задачи с камнями, используя полный двоичный перебор.
2. Создайте копию решения и добавьте в него вышеописанные отсечения.
3. Реализуйте решение для задачи с камнями, используя рекурсивный перебор.
4. Создайте копию этого решения и добавьте в него отсечения.
5. Создайте текстовый файл с 30 камнями случайного веса. Замерьте время работы всех четырёх решений, сделайте выводы.
6. Создайте текстовый файл с 30 камнями с такими весами, чтобы поровну на две кучи камни разделить было невозможно (подумайте, как это сделать). Замерьте время работы всех четырёх решений, сделайте выводы.
7. Решите задачу №866 из проверяющей системы.
8. Решите задачу №869 из проверяющей системы.
9*. Решите задачу №680 из проверяющей системы.

Курсовая работа

Цель курсовой работы состоит в разработке визуализатора заданного алгоритма или структуры данных. Визуализатор алгоритма – программа, наглядно демонстрирующая работу алгоритма в пошаговом режиме над заданным набором данных. Визуализатор разрабатывается как обычное приложение с графическим пользовательским интерфейсом в любой среде программирования. По желанию студента визуализатор может быть реализован с возможностью выполнения непосредственно в веб-браузере (как Java-апплет, элемент ActiveX, Web-приложение на языке JavaScript или VBScript и другие способы).

Разработанный визуализатор должен обеспечивать:

- наглядную графическую иллюстрацию всех особенностей работы алгоритма,
- вывод пояснения к каждому шагу алгоритма,
- работу в пошаговом и автоматическом режиме,
- регулировку скорости автоматического выполнения,
- возможность отката на любое количество шагов назад,

- работу как с предварительно заданными, так и со случайными и введёнными пользователем данными,
- корректную обработку частных и вырожденных случаев.

Пример возможного интерфейса визуализатора представлен на следующем рисунке.

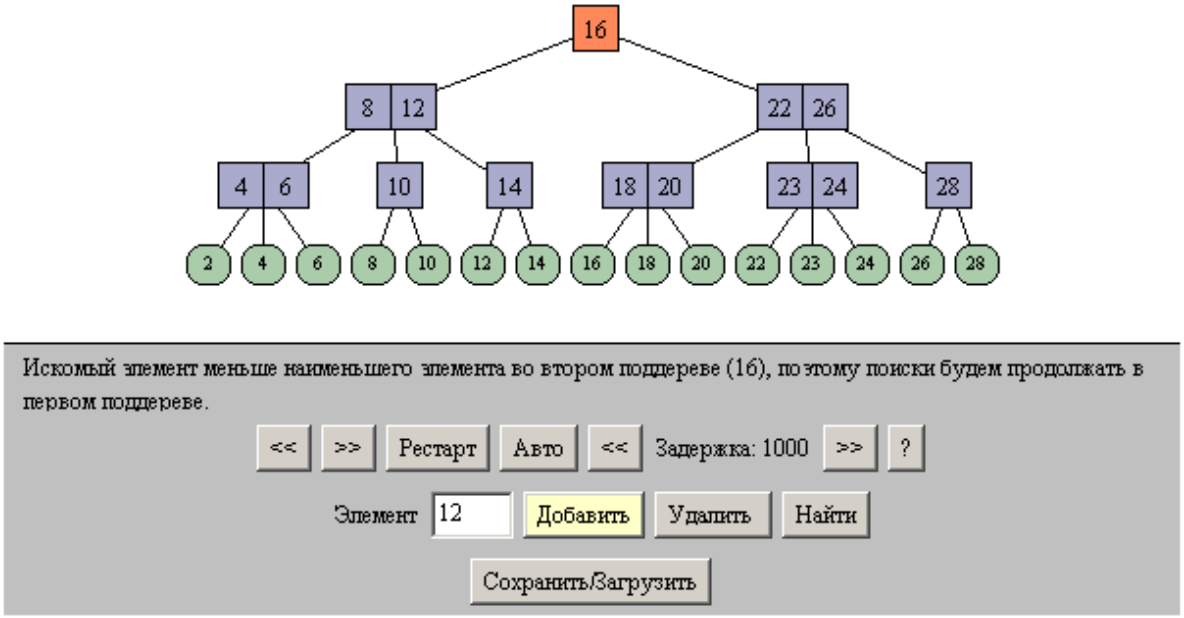

Рис.20. Пример интерфейса визуализатора

Примерная тематика/формулировки заданий (по вариантам):

№	Задание курсовой работы
1	Полный двоичный перебор и его ускорение путем отсечения заведомо неподходящих вариантов. Можно взять такой пример: дано N камней разных весов. Нужно разбить их на две кучи так, чтобы веса каждой кучи отличались как можно меньше.
2	AVL-деревья. Требуется показать, как выполняются большие и малые левые и правые вращения при вставке и удалении элементов, предусмотреть возможность вставки и удаления произвольных (задаваемых пользователем элементов)
3	Перебор с возвратом (backtracking) и отсечением заведомо неподходящих вариантов. Для визуализации можно рассмотреть задачу коммивояжера.
4	Алгоритм Краскала для построения минимального остовного дерева. В визуализаторе необходимо показать как графически построение дерева, так и структуры данных, позволяющие выполнить это эффективно
5	Алгоритм Прима. В визуализаторе необходимо показать как графически построение дерева, так и структуры данных, позволяющие выполнить это эффективно
6	Если взять все перестановки первых N натуральных чисел и отсортировать их по возрастанию, то каждой перестановке будет соответствовать уникальный номер. Задача состоит в эффективном получении переста-

	новки по её номеру и номера по заданной перестановке.
7	Поразрядная сортировка строк разной длины от старших разрядов (первых символов) к младшим (последним символам)
8	Поиск эйлеровых путей или цикла в графе. Необходимо рассмотреть как ориентированные, так и неориентированные графы.
9	Метод динамического программирования на примере задачи о рюкзаке: дано N предметов, для каждого из них известен вес и стоимость. Нужно выбрать некоторое число предметов на как можно большую сумму, но не превысить максимального веса W.
10	Алгоритм Дейкстры для поиска кратчайших путей от заданной вершины до всех остальных
11	Динамическое программирование на примере задачи об оптимальной триангуляции выпуклого N-угольника. Суть задачи: дан выпуклый N-угольник. Проведя диагонали, разбить его на треугольники, чтобы сумма длин диагоналей была минимальной.
12	Сортировка Шелла. Рассмотреть различные последовательности приращений.
13	Быстрая сортировка Хоара. В визуализаторе показать схематически в виде дерева, как выполнялись рекурсивные вызовы.
14	Сортировка двоичной кучей (пирамидой). В визуализаторе показать как графическое представление пирамиды, так и её хранение в массиве.
15	Поиск сильно связных компонент в графе на основе обхода в глубину
16	Динамическое программирование на примере задачи о перемножении цепочки матриц. Суть задачи: дано N матриц, которые нужно перемножить (перемножение идёт обычным методом «строка на столбец»). Требуется так расставить скобки, чтобы общий объём вычислений был минимальным.
17	Линейное хеширование.
18	Быстрая сортировка Хоара. В визуализаторе показать схематически в виде дерева, как выполнялись рекурсивные вызовы.
19	Сортировка слиянием. В визуализаторе показать схематически в виде дерева, как выполнялись рекурсивные вызовы.
20	Распределяющая сортировка натуральных чисел от младших разрядов к старшим
21	Рандомизированное поисковое дерево
22	Нерекурсивный вариант быстрой сортировки Хоара
23	Поиск мостов в графе
24	Поиск точек сочленения в графе
25	Обходы графов

Примерный объем пояснительной записки: 20 стр., шрифт 12, через 1,3 интервала. Примерный объем графической части: 2 листа формата А4.

Заключение

Тематика практических работ в данных методических указаниях была подобрана таким образом, чтобы в процессе их выполнения студенты получили навыки разработки и программной реализации алгоритмов и структур данных из всех разделов курса. Внимательное и добросовестное выполнение лабораторных работ является необходимым условием для успешного выполнения курсовой работы. Полученные при этом знания и навыки пригодятся в последующих учебных дисциплинах, а также будут полезны в профессиональной деятельности в области разработки программного обеспечения.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Ржеуцкая, С. Ю. Структуры данных и алгоритмы обработки: учеб. пособие: [в 2 ч.]. Ч. 1/ С. Ю. Ржеуцкая, И. А. Андрианов. - Вологда: ВоГТУ, 2005. - 223 с.
2. Ржеуцкая, С. Ю. Структуры данных и алгоритмы обработки: учеб. пособие: [в 2 ч.]. Ч. 2. / С. Ю. Ржеуцкая, И. А. Андрианов . - Вологда: ВоГТУ, 2007. - 128 с.
3. Кормен , Т. Алгоритмы: построение и анализ/ Т. Кормен , Ч. Лейзерсон, Р. Ривест; пер. с англ. - 3-е изд.. - М.: Вильямс, 2013. - 1328 с.
4. Задачи по программированию / С. Окулов, Т. Ашихмина, Н. Бушмелева, М. Корчемкин, Е. Разова, Р. Шарыгин - М.: Бином. Лаборатория знаний, 2014. - 824 с.
5. Скиена, С. Алгоритмы. Руководство по разработке / С. Скиена; пер. с англ. - СПб.: БХВ-Петербург, 2011. - 720 с.
6. Шень, А. Программирование. Теоремы и задачи /А. Шень. - М.: МЦНМО, 2013. - 296 с.
7. MAXimal: алгоритмы [Электронный ресурс]. - Режим доступа: http://e-maxx.ru/algo.
8. Школа программиста [Электронный ресурс]. – Режим доступа: http://acmp.ru.
9. Timus Online Judge [Электронный ресурс]. – Режим доступа: http://acm.timus.ru.
10. Дистанционный практикум по программированию (проверяющая система) кафедры АВТ ВоГУ [Электронный ресурс]. – Режим доступа: http://atpp.vstu.edu.ru/cgi-bin/arh_problems.pl.
11. Метод деления отрезка пополам [Электронный ресурс]. – Режим доступа: http://eco.sutd.ru/Study/Informat/mpd.html.

Подписано в печать 23.06.2014.	Усл. печ. л.	Тираж экз.
Печать офсетная.	Бумага писчая.	Заказ №____.

Отпечатано: РИО ВоГУ, г. Вологда, ул. Орлова, 6