МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Тульский государственный университет»

Институт прикладной математики и компьютерных наук

Кафедра информационной безопасности

Утверждено на заседании кафедры
«Информационная безопасность»
«__» _____ 2021 г., протокол № __

Заведующий кафедрой

_____А.А. Сычугов

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
по выполнению лабораторных работ по дисциплине (модулю) «Алгоритмы и структуры данных в информационных системах»

основной профессиональной образовательной программы высшего образования – программы специалитета

по направлению подготовки
09.03.02 Информационные системы и технологии

с профилем
Информационные системы

Форма обучения: *очная*

Идентификационный номер образовательной программы: 090302-01-21

Тула 2021 год

Разработчик(и) методических указаний

Двоенко С.Д., профессор, д. ф.- м.н., доцент _____

(ФИО, должность, ученая степень, ученое звание) *(подпись)*

_____ _____

(ФИО, должность, ученая степень, ученое звание *(подпись)*

Содержание

Лабораторная работа №1 Генерация случайных чисел

1. Цель и задачи работы

Изучение способов генерации последовательности случайных чисел. Написание программы, демонстрирующей изученные принципы.

2. Теоретические сведения

Случайной величиной называется величина, которая в результате опыта может принять то или иное значение, причем неизвестно заранее, какое именно.

Но в некотором смысле такого объекта, как случайное число, просто нет. Скажем, двойка – это случайное число? Скорее можно говорить о последовательности независимых случайных чисел с определенным законом распределения, и это означает, грубо говоря, что каждое число было получено самым произвольным образом, без всякой связи с другими членами последовательности, и что у него есть определенная вероятность оказаться в любом заданном интервале.

Равномерным называется такое распределение, при котором каждое число равновероятно. Обычно, если специально не оговорено что-либо иное, имеют в виду равномерные распределения.

Линейный конгруэнтный метод

Схема данного метода была предложена Д.Х. Лемером в 1948 г. Для генерации случайных чисел с помощью линейного конгруэнтного метода необходимо выбрать четыре числа:

X_0 — начальное значение; $X_0 \geq 0$;
a — множитель; $a \geq 0$;
c — приращение; $c \geq 0$;
m — модуль; $m > X_0$, $m > a$, $m > c$.

Тогда искомая последовательность случайных чисел $\langle X_n \rangle$ получается из соотношения

$$X_{n+1} = (aX_n + c) \bmod m, \quad n \geq 0. \qquad (1)$$

Она называется линейной конгруэнтной последовательностью.

Например, при $X_0 = a = c = 7$, $m = 10$ последовательность выглядит так:

7,6,9,0,7,6,9,0, ...

Как видно из приведенного примера, последовательность не всегда оказывается «случайной», если выбирать X_0, a, c, m произвольно.

Этот пример иллюстрирует тот факт, что конгруэнтные последовательности всегда «зацикливаются», т.е., в конце концов, числа образуют цикл, который повторяется бесконечное число раз. Это свойство присуще всем последовательностям, имеющим общий вид $X_{n+1}=f(X_n)$. Повторяющийся цикл называется периодом. Длина периода у последовательности в данном примере равна 4. Реальные последовательности, которыми пользуются, имеют, конечно, сравнительно большой период.

Специального упоминания заслуживает частный случай c=0, когда процесс выработки случайных чисел происходит несколько быстрее. Ограничение c=0 уменьшает длину периода последовательности, но при этом все еще можно получить относительно большой период. В первоначальном методе Лемера было принято c = 0, хотя автор и упомянул возможность использования $c\neq0$. Идея получения более длинных последовательностей за счет обобщения $c\neq0$ принадлежит Томсону и независимо Ротенбергу.

Чтобы упростить многие формулы, оказывается полезным определить b=a – 1.

Можно сразу же отбросить случай a=1, так как при этом $X_n=(X_0+nc) \bmod m$, и очевидно, что последовательность не случайная. Вариант a=0 еще хуже. Следовательно, для практических целей можем предположить, что $a\geq2$, $b\geq1$.

Теперь можно обобщить соотношение (1),

$$X_{n+k}=(a^kXn+(a^k–1)c/b) \bmod m,\ k\geq0,\ n\geq0,$$

выразив (n+k)-й член прямо через n-й. Последовательность, составленная из каждого k-го члена нашей последовательности, образует другую линейную конгруэнтную последовательность с множителем a^k и приращением $((a^k–1)c/b)$.

Выбор параметров генератора

Так как длина периода не может быть больше m, то значение m должно быть достаточно большим. Чтобы получить длину периода последовательности равную m, нужно выполнить условия теоремы 1.

Теорема 1. *Длина периода линейной конгруэнтной последовательности равна m тогда и только тогда, когда*

1) c и m — взаимно простые числа;

2) b=a–1 кратно p для любого простого p, являющегося делителем m;

3) b кратно 4, если m кратно 4.

Таким образом, если выбрать числа a, c, и m так, чтобы они удовлетворяли условию теоремы 1, то период вырабатываемой таким генератором последовательности случайных чисел будет максимален и равен m.

Квадратичный конгруэнтный метод

Линейный конгруэнтный метод — не единственный метод, использующийся для выработки равномерно распределенных случайных чисел с помощью ЭВМ.

Линейный конгруэнтный метод можно обобщить, превратив его в квадратичный конгруэнтный метод:

$$X_{n+1}=(dX_n^2+aX_n+c) \bmod m.$$

Необходимые и достаточные условия для a, c и d, такие, чтобы данная последовательность имела бы максимальный период m, устанавливаются теоремой 2.

Теорема 2. *Квадратичная конгруэнтная последовательность имеет период длины m тогда и только тогда, когда выполняются следующие условия:*

1) *c и m — взаимно простые числа;*
2) *d и a кратны p — всем нечетным простым делителям m;*
3) *d — четное и d тождественно равно остатку от деления a–1 на 4, если m кратно 4, d тождественно равно остатку от деления a–1 на 2, если m кратно 2;*
4) *или d≡0, или a≡1 и c·d при делении на 9 всегда дает остаток 6, если m кратно 9.*

Метод Р. Ковэю

Для случая, когда m представляется степенью двойки, интересный квадратичный метод предложил Р. Ковэю:

$X_0 \bmod 4=2$, $X_{n+1}=X_n(X_n+1) \bmod 2^e$, $n\geq 0$.

Данный метод гарантирует достаточно большой период.

Аддитивный генератор чисел

Можно рассматривать генераторы случайных чисел вида

$$X_{n+1}=(X_n+X_{n-k}) \bmod m,$$

где k — достаточно большое число. Такие генераторы были предложены Грином, Смитом и Клемом. При соответствующем выборе X_0, X_1, ..., X_k эта формула может стать источником хороших случайных чисел.

Длина периода такого генератора не намного больше m. В статье Грина, Смита и Клема говорится, что при $k\leq 15$ последовательность не удовлетворяет тесту «проверка интервалов» (один из тестов, оценивающий

критерии случайности последовательности, т.е. показывает — достаточно ли случайна последовательность), хотя при k=16 тест проходит нормально.

Комбинированные генераторы случайных чисел

Другой важный класс методов сводится к комбинации датчиков случайных чисел для получения «еще более случайных» последовательностей.

Предположим, что мы имеем две последовательности X_0, X_1, ..., и Y_0, Y_1, ..., случайных чисел, расположенных между нулем и m–1(m — модуль последовательности), полученные двумя независимыми способами. Одно из предположений сводится к тому, чтобы складывать числа попарно по модулю m, получая последовательность $Z_n=(X_n+Y_n) \bmod m$. В этом случае желательно, чтобы длины периодов $<X_n>$ и $<Y_n>$ были взаимно простыми числами.

Другой метод, предложенный Маклареном и Марсальей значительно лучше.

При заданных методах выработки двух последовательностей $<X_n>$ и $<Y_n>$ этот метод позволяет генерировать «значительно более случайную последовательность». Мы используем вспомогательную таблицу V[0], V[1], ..., V[k], где k — некоторое число, выбираемое обычно для удобства равным примерно 100. Сначала V-таблица заполняется первыми k значениями X-последовательности. Затем выполняются следующие шаги:

1) присвоить X и Y значения очередных членов последовательностей $<X_n>$ и $<Y_n>$ соответственно;

2) вычислить $j=kY/m$, где m — модуль, использующийся в последовательности $<Y_n>$; таким образом, j принимает случайное значение, определяемое с помощью Y; $0 \le j < k$;

3) вывести V[j] — очередной член новой последовательности, и присвоить V[j] значение X.

Этот метод позволяет получать чрезвычайно большие периоды, если периоды последовательностей $<X_n>$ и $<Y_n>$ взаимно простые. И даже если длина периода не очень существенна, соседние члены последовательности почти не коррелируют друг с другом. Причиной того, что этот метод намного превосходит другие методы, является достаточная случайность последовательностей X_n и Y_n, которые не могут вырождаться.

3. Пример программы на языке C/C++

Программа генерации равномерно распределенных случайных чисел с использованием линейного конгруэнтного метода.

```
#include <iostream.h>
#include <conio.h>

const c=15,a=13;
```

```
void main()
{
   unsigned long Num,
                 m=65536,
                 x0=13;
   clrscr();
   cout<<”Генерация равномерно распределенных случайных чисел”;
   cout<<” с использованием линейного конгруэнтного метода ”;
   cout<<”Для продолжения нажмите любую клавишу, для выхода-Esc”;
   while(getch()!=27)
      {
         Num=(a*x0+c)%m;
         cout<<(float(Num)/float(m+1))<<'\n';
         x0=Num;
      }
}
```

4. Варианты заданий

1. Написать программу, генерирующую случайные числа в диапазоне [a,b], используя линейный конгруэнтный метод.
2. Написать программу, генерирующую случайные числа, используя квадратичный конгруэнтный метод.
3. Написать программу, генерирующую случайные числа, используя метод Р. Ковэю.
4. Написать программу, генерирующую случайные числа, используя аддитивный генератор чисел.
5. Написать программу, генерирующую случайные числа, используя комбинированный генератор случайных чисел.
6. Написать программу, находящую при генерации случайных чисел повторяющиеся значения и подсчитывающую количество шагов между ними. Для генерации чисел использовать линейный конгруэнтный метод.
7. Выполнить задание варианта 6 с использованием квадратичного конгруэнтного метода.
8. Выполнить задание варианта 6 с использованием метода Р. Ковэю.
9. Выполнить задание варианта 6 с использованием аддитивного генератора чисел.
10. Выполнить задание варианта 6 с использованием комбинированного генератора случайных чисел.
11. Написать программу, определяющую, какое число чаще других встречается в последовательности целых случайных чисел. Для генерации чисел использовать линейный конгруэнтный метод.
12. Выполнить задание варианта 11 с использованием квадратичного конгруэнтного метода.
13. Выполнить задание варианта 11 с использованием метода Р. Ковэю.
14. Выполнить задание варианта 11 с использованием аддитивного генератора чисел.

15. Выполнить задание варианта 11 с использованием комбинированного генератора случайных чисел.
16. Написать программу, демонстрирующую равномерность распределения случайных чисел на примере линейной конгруэнтной последовательности.
17. Выполнить задание варианта 16 с использованием квадратичного конгруэнтного метода.
18. Выполнить задание варианта 16 с использованием метода Р. Ковэю.
19. Выполнить задание варианта 16 с использованием аддитивного генератора случайных чисел.
20. Выполнить задание варианта 6 с использованием комбинированного генератора случайных чисел.

5. Указания по составлению отчета

Отчет должен содержать постановку и описание задачи, алгоритм решения задачи, текст составленной программы на языке программирования С++, краткие выводы.

Лабораторная работа №2 Генерация нормального и заданного распределений

1. Цель и задачи работы

Изучить генерацию случайных величин по заданному и нормальному законам распределения. Написать программы, иллюстрирующие изученные принципы.

2. Теоретические сведения

Для чего вообще нужны случайные числа? Оказывается, что они весьма полезны для самых различных целей: 1) моделирование реальных процессов, включающих в себя случайные величины; 2) выборка, когда практически нельзя проверить все варианты; 3) численный анализ, используемый при решении сложных задач вычислительной математики; 4) программирование вычислительных машин, когда случайные значения служат хорошим источником данных при испытании эффективности различных алгоритмов; 5) принятие решений; 6) игры.

Метод обратной функции

Остановимся на моделировании реальных процессов. Например, в физике и микроэлектронике многие явления носят вероятностный характер и описываются функциями распределения, а не точными формулами. Таким образом, при моделировании таких явлений возникает необходимость генерации числовых последовательностей, элементы которых распределены по заданному закону.

Обычно для этого используют генератор равномерно распределенной величины, преобразуя выданную им последовательность в нужную. Одним из наиболее универсальных методов преобразования случайной величин, равномерно распределенной в интервале *(0,1)*, в случайную величину с произвольным законом распределения, является метод обратной функции.

Пусть требуется получить случайную величину X, характеризуемую функцией распределения $F(x)=P(X<x)$. Согласно определению, функция распределения любой случайной величины, во-первых, принимает

значения в интервале (0,1), т.е. $0 \le F(x) \le 1$ для всех x, и, во-вторых, является неубывающей функцией, т.е. $F(x') < F(x'')$ при $x' < x''$.

Образуем функцию $x = \varphi(y)$, обратную, по отношению к $F(x)$, т.е. так, чтобы для всякого значения аргумента $0 \le y \le 1$ значение функции определялось, как решение уравнения $F(x) = y$. Тогда, если случайная величина Y равномерно распределена в интервале (0,1), то случайная величина X, полученная из нее преобразованием $x = \varphi(y) = F^{-1}(y)$, характеризуется функцией распределения $F(x)$ и, соответственно плотностью распределения $f(x) = F'(x)$.

Действительно, в любой конкретной точке x значение функции распределения случайной величины X определяется как вероятность $P(X<x) = P(0 \le Y \le F(x))$. Поскольку Y равномерно распределена в интервале (0,1), то $P(0 \le Y \le F(x)) = F(x)$, откуда получим $P(X<x) = F(x)$.

Пусть, например, необходимо получить случайную величину X экспоненциальной плотностью распределения

$$f(x) = \begin{cases} \lambda e^{-\lambda x}, & 0 \le x < \infty \\ 0, & x < 0 \end{cases} \qquad (1)$$

Соответствующая функция распределения выражается формулой

$$F(x) = \int_{-\infty}^{x} f(\xi) d\xi = \lambda \int_{0}^{x} e^{-\lambda \xi} d\xi = 1 - e^{-\lambda x}$$

Обратная функция $x = \varphi(y) = F^{-1}(y)$ может быть получена как решение уравнения $1 - e^{-\lambda x} = y$, т.е. определяется формулой $x = \varphi(y) = -\frac{1}{\lambda} \ln(1 - y)$. Эту формулу можно еще упростить, приняв во внимание,, что если случайная величина Y распределена равномерно в интервале $(0,1)$, то случайная величина $1 - Y$ распределена точно так же. Окончательно получим:

$$x = \varphi(y) = -\frac{1}{\lambda} \ln y$$

Методом обратной функции удобно пользоваться в том случае, если уравнение $F(x) = y$ можно решить аналитически в виде некоторой формулы $x = \varphi(y)$. Если же желаемая функция распределения $F(x)$ не выражается через элементарные функции, как, например, в случае нормального распределения, то использование метода обратной связи затруднено, поскольку при каждом акте генерирования случайной величины необходимо прибегать к интерполированию по таблицам и к численным методам решения уравнения.

Получение нормальных случайных величин

Рассмотрим способ генерирования нормальных случайных величин, основанный на предельном свойстве сумм независимых случайных величин, установленном центральной предельной теоремой. Плотность нормального распределения с математическим ожиданием m и дисперсией σ^2 выражается формулой

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2\sigma^2}(x-m)^2} \qquad (2)$$

В одной из самых простых форм центральная предельная теорема может быть сформулирована следующим образом.

Теорема. *Если* $Y_1,....Y_n$ *независимые случайные величины, имеющие один и тот же закон распределения с математическим ожиданием* m *и дисперсией* σ^2*, то при неограниченном увеличении* n *закон распределения суммы* $X = \sum_{i=1}^{n} Y_i$*, неограниченно приближается к нормальному с математическим ожиданием* $m \cdot n$ *и дисперсией* $n \cdot \sigma^2$**.**

В частности, в качестве случайных величин Y, удобно использовать независимые случайные величины, равномерно распределенные в интервале *(0,1)*. Поскольку математическое ожидание и дисперсия такой случайной величины соответственно равны 1/2 и 1/12, то нормальная случайная величина X может быть получена из n равномерных Y_i по формуле:

$$X = \frac{12\sigma}{n}(\sum_{i=1}^{n} Y_1 - \frac{n}{2}) + m$$

Формально распределение такой случайной величины приближается к нормальному лишь при n, однако, уже начиная с $n = 10$, отличие становится практически несущественным. Обычно берут $n = 12$, чтобы избежать лишнего умножения:

$$X = \sigma(\sum_{i=1}^{n} Y_1 - 6) + m$$

3. Пример аналитического расчета.

С помощью метода обратной функции получить случайную величину с заданной плотностью распределения *f*(*x*), приведенной на рис. 1. Построить нормальное распределение с заданными математическим ожиданием и дисперсией. Полученную в результате генерирования плотность вероятности сравнить с теоретической.

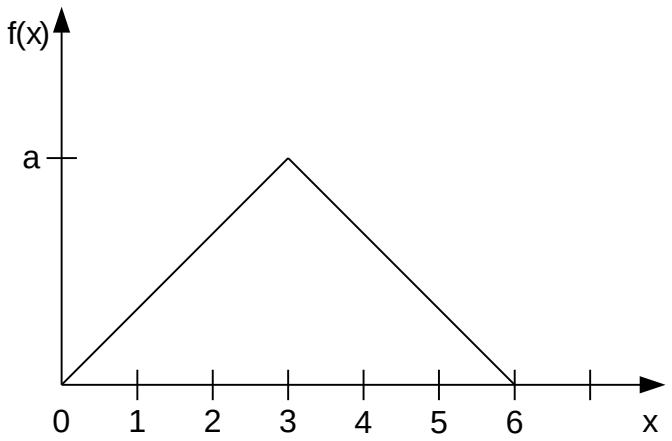


Рис. 1 Плотность распределения

Прежде всего, определим постоянную a, используя условие нормировки

$$\int_{-\infty}^{\infty} f(x)dx = 1.$$

Получим, что a=1/3.

Запишем теперь функцию распределения, учтя константу a.

$$f(x) = \begin{cases} \frac{x}{9}, & 0 < x \le 3 \\ \frac{2}{3} - \frac{x}{9}, & 3 < x \le 6 \end{cases}$$

Далее найдем функцию распределения $F(x)$.

$$F(x) = \int_{-\infty}^{x} f(x)dx$$

$$F(x) = \begin{cases} \frac{x^2}{18}, & 0 < x \le 3 \\ \frac{2}{3}x - \frac{x^2}{18} - 1, & 3 < x \le 6 \end{cases}$$

Теперь найдем функцию обратную к $F(x)$.

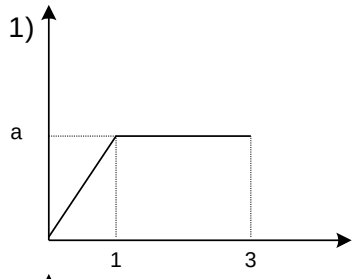
$$x = \varphi(y) = F^{-1}(y) = \begin{cases} 3\sqrt{2}\sqrt{y}, & 0 < y \le 0.5 \\ 6 - 3\sqrt{2-2y}, & 0.5 < y \le 1 \end{cases}$$

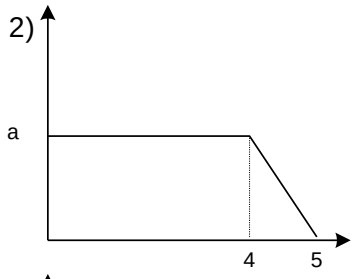
Найдем математическое ожидание и дисперсию для нормального распределения.

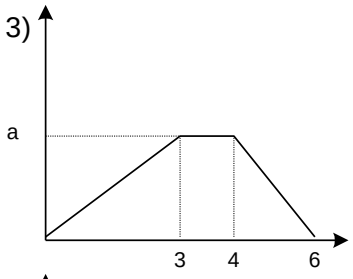
$$m = \int_{-\infty}^{\infty} xf(x)dx = 3$$

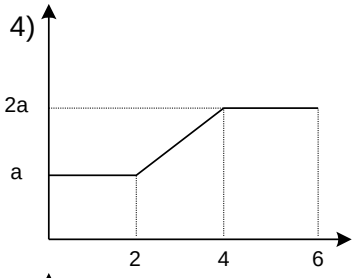
$$D = \int_{-\infty}^{\infty} (x-m)^2 f(x)dx = 1.5$$

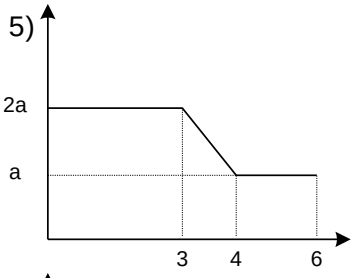
4. Варианты заданий

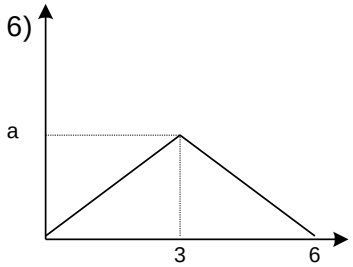
1)


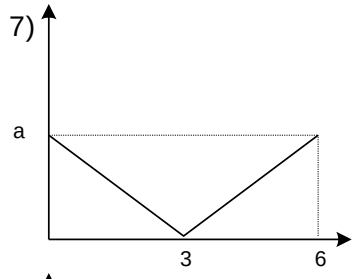
2)


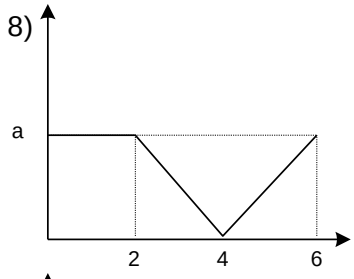
3)


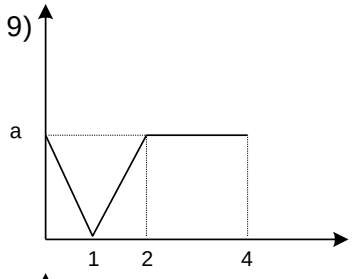
4)


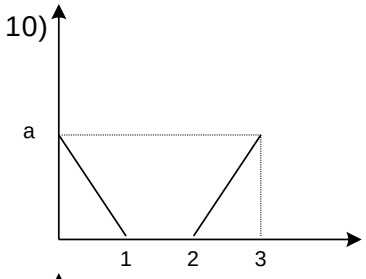
5)


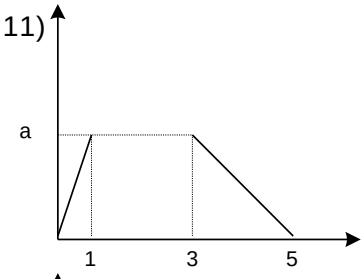
6)


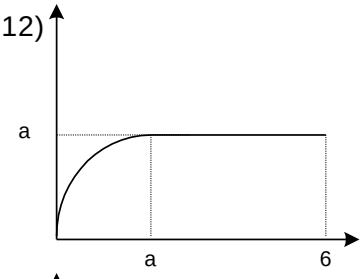
7)


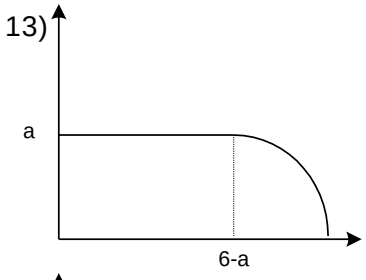
8)


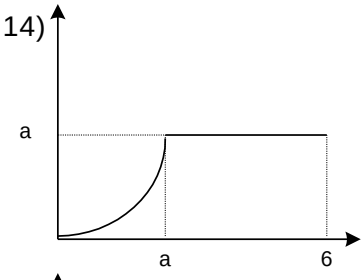
9)


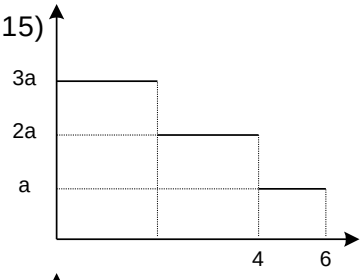
10)


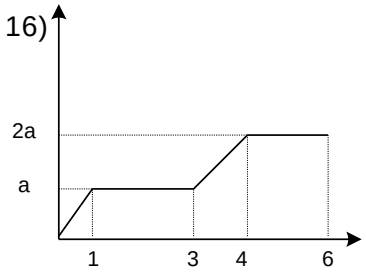
11)


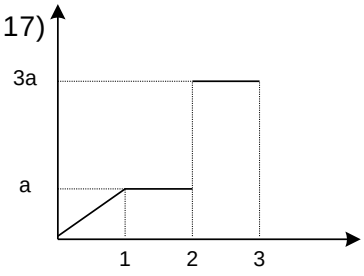
12)


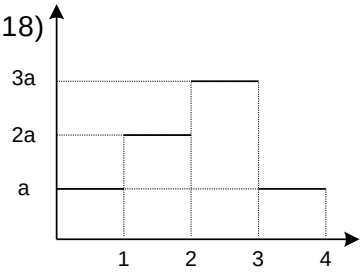
13)


14)


15)


16)


17)


18)


5. Указания по составлению отчета

Отчет должен содержать постановку и описание задачи, алгоритм решения задачи, текст составленной программы на языке программирования C++, краткие выводы.

Лабораторная работа №3 Сортировка прямым обменом

1. Цель и задачи работы

Изучение метода сортировки массивов прямым обменом. Написание программы, демонстрирующей изученные принципы.

2. Теоретические сведения

Сортировка — это процесс, позволяющий упорядочить множество подобных данных в возрастающем или убывающем порядке. Сортировка представляет собой один из наиболее приемлемых с интеллектуальной точки зрения алгоритмов, поскольку процесс хорошо определен. Алгоритмы сортировки хорошо исследованы и изучены. К сожалению, по этой причине сортировку иногда принимают как некую данность.

В общем, сортировку следует понимать как процесс перегруппировки заданного множества объектов в некотором определенном порядке. Цель сортировки — облегчить последующий поиск элементов в таком отсортированном множестве. Это почти универсальная, фундаментальная деятельность. Отсортированные объекты можно встретить на каждом шагу: в телефонных книгах, в списках подоходных налогов, в оглавлениях книг, в библиотеках, в словарях, на складах – почти везде, где надо искать хранимые объекты. Даже малышей учат держать вещи «в порядке», и они уже сталкиваются с некоторыми видами сортировок задолго до того, как познакомятся с азами арифметики. Между сортировкой и арифметикой нет никакой видимой связи, да, по-видимому, и невидимой нет. Сортировка — некий «первичный» процесс человеческой деятельности.

Таким образом, разговор о сортировке вполне уместен и важен, если речь идет об обработке данных. Что может легче сортироваться, чем данные? Тем не менее, первоначальный интерес к сортировке основывается на том, что при построении алгоритмов часто сталкиваются со многими фундаментальными приемами. Почти не существует методов, с которыми не приходится встречаться при обсуждении этой задачи. В частности, сортировка — это идеальный объект для демонстрации огромного разнообразия алгоритмов, все они изобретены для одной и той же задачи, многие в некотором смысле оптимальны, большинство имеют достоинства. Поэтому это еще и идеальный объект, демонстрирующий необходимость анализа производительности алгоритмов. К тому же на примерах сортировок можно показать, как путем усложнения алгоритма,

хотя под рукой и есть уже очевидные методы, можно добиться значительного выигрыша в эффективности.

Вот некоторые из наиболее важных применений сортировки:

1. Решение задачи «группировки», когда нужно собрать вместе все элементы с одинаковым значением некоторого признака. Допустим, имеется 10000 элементов, расположенных в случайном порядке, причем значения многих из них равны; и предположим, нам нужно переупорядочить файл так, чтобы элементы с равными значениями занимали соседние позиции в файле. Это, по существу, задача «сортировки» в широком смысле слова, и она легко может быть решена путем сортировки файла в. узком смысле слова, а именно расположением элементов в неубывающем порядке $v_1 \le v_2 \le \ldots \le v_{10000}$.

2. Если два или более файла отсортировать в одном и том же порядке, то можно отыскать в них все общие элементы за один последовательный просмотр всех файлов, без возвратов. Оказывается, что, как правило, гораздо экономнее просматривать список последовательно, а не перескакивая с места на место случайным образом, если только список не настолько мал, что он целиком помещается в оперативной памяти. Сортировка позволяет использовать последовательный доступ к большим файлам в качестве приемлемой замены прямой адресации.

Теперь четко определим задачу и введем соответствующую терминологию. Пусть надо упорядочить N элементов

$$R_1, R_2, \ldots, R_N.$$

Назовем их *записями,* а всю совокупность N записей назовем *файлом.* Каждая запись R_j имеет *ключ* K_j, который и управляет процессом сортировки. Помимо ключа, запись может содержать дополнительную «сопутствующую информацию», которая не влияет на сортировку, но всегда остается в этой записи.

Отношение порядка «<» на множестве ключей вводится таким образом, чтобы для любых трех значений ключей a, b, c выполнялись следующие условия:

1) справедливо одно и только одно из соотношений: $a<b$, $a=b$, $b<a$ (закон трихотомии);

2) если $a<b$ и $b<c$, то $a<c$ (закон транзитивности).

Эти два свойства определяют математическое понятие *линейного упорядочения,* называемого еще *совершенным упорядочением.* Любое множество с отношением «<», удовлетворяющим вышеперечисленным свойствам, поддается сортировке большинством методов.

Задача сортировки — найти такую перестановку записей $p(1), p(2),\ldots,p(N)$, после которой ключи расположились бы в неубывающем порядке:

$$K_{p(1)}, K_{p(2)}, \ldots, K_{p(N)}.$$

Сортировка называется *устойчивой,* если она удовлетворяет дополнительному условию, что записи с одинаковыми ключами остаются в прежнем порядке, т. е.

$$p(i)<p(j),\ \text{если}\ K_{p(i)}=K_{p(j)}\ \text{и}\ i<j.$$

Обычно сортировку подразделяют на два класса: *внутреннюю,* когда все записи хранятся в быстрой оперативной памяти, и *внешнюю,* когда все они там не помещаются. При внутренней сортировке имеются более гибкие возможности для построения структур данных и доступа к ним, внешняя же показывает, как поступать в условиях сильно ограниченного доступа.

Достаточно хороший общий алгоритм затрачивает на сортировку N записей время порядка $N\log(N)$; при этом требуется около $\log(N)$ «проходов» по данным. Это минимальное время. Так, если удвоить число записей, то и время при прочих равных условиях возрастет немногим более чем вдвое.

Сортировка массивов

Основное условие: выбранный метод сортировки массивов должен экономно использовать доступную память. Это предполагает, что перестановки, приводящие элементы в порядок, должны выполняться *на том же месте*, то есть методы, в которых элементы из массива a передаются в результирующий массив b, представляют существенно меньший интерес.

Хорошей мерой эффективности может быть C — число необходимых сравнений ключей и M — число пересылок (перестановок) элементов. Эти числа суть функции от числа сортируемых элементов. Как говорилось выше, если сортируется N элементов, то хорошим считается такой алгоритм сортировки, который требует порядка $N\log(N)$ сравнений. Для начала стоит рассмотреть самые простые алгоритмы, которые называют *прямыми*, требующими порядка N^2 сравнений. Прямые методы особенно удобны для объяснения характерных черт основных принципов большинства сортировок, программы этих методов легко понимать, и они коротки. Усложненные же методы требуют небольшого числа операций, но эти операции обычно сами более сложны, и поэтому для достаточно малых N прямые методы оказываются быстрее, хотя при больших N их использовать, конечно, не следует.

Методы сортировки *«на том же месте»* можно разбить в соответствии с определяющими их принципами на три основные категории:

1. Сортировки с помощью включения (by insertion)
2. Сортировки с помощью выделения (by selection)
3. Сортировка с помощью обменов (by exchange)

Сортировка с помощью прямого обмена

Пожалуй, наиболее очевидный способ обменной сортировки это сравнивать K_1 с K_2, меняя местами R_1 и R_2, если их ключи не упорядочены, затем проделать то же самое с R_2 и R_3, R_3 и R_4 и т.д. При выполнении этой последовательности операций записи с большими ключами будут продвигаться вправо, и на самом деле запись с наибольшим ключом займет положение R_N. При многократном выполнении этого процесса соответствующие записи попадут в позиции R_{N-1}, R_{N-2} и т.д., так что в конце концов, все записи будут упорядочены.

Данный метод назван «методом пузырька», т.к. если рассматривать массивы как вертикальные, а не горизонтальные построения, то большие элементы, подобно пузырькам, «всплывают» на соответствующую позицию. Метод пузырька известен также под более прозаическими именами, такими, как «обменная сортировка с выбором» или метод «распространения».

Можно предложить несколько путей улучшения метода пузырька. Заметим, что за один просмотр элемент не может переместиться более чем на одну позицию влево, так что если наименьший элемент вначале находится в правом конце, то нужно выполнить максимальное число сравнений. Это наводит на мысль о «шейкерной» сортировке (ShakerSort), когда просмотр осуществляется попеременно в обоих направлениях. При таком подходе среднее число сравнений несколько сокращается. Это происходит потому, что если R_j и R_{j+1} не меняются местами при двух последовательных просмотрах в противоположных направлениях, то записи R_j и R_{j+1} должны занимать свои окончательные позиции, и они могут не участвовать в последующих сравнениях. Например, просматривая перестановку 4 3 2 1 8 6 9 7 5 слева направо, получаем 3 2 1 4 6 8 7 5 9: записи R_4 и R_5 не поменялись местами. При просмотре последней перестановки справа налево R_4 все еще меньше (новой) записи R_5; следовательно, можно сразу же сделать вывод о том, что записи R_4 и R_5 могут и не участвовать ни в одном из последующих сравнений.

Анализ пузырьковой и шейкерной сортировок

Число сравнений в строго обменном алгоритме $C=(N^2-N)/2$, а минимальное, среднее и максимальное число перемещений элементов (присваиваний) равно соответственно $M_{\min}=0$, $M_{avg}=3*(N^2-N)/4$, $M_{\max}=3*(N^2-N)/2$. Анализ же улучшенных методов, особенно шейкерной сортировки, довольно сложен. Минимальное число сравнений $C_{\min}=N-1$. Кнут считает, что для улучшенной пузырьковой сортировки среднее число проходов пропорционально $N-k_1N^{1/2}$, а среднее число проходов пропорционально $1/2(N^2-N(k_2+\ln N))$. Следует обратить внимание на то, что все перечисленные выше усовершенствования не влияют на число

перемещений, они лишь сокращают число излишних двойных проверок. Обмен местами двух элементов — чаще всего более дорогостоящая операция, чем сравнение ключей. «Обменная» сортировка и ее небольшие усовершенствования представляют собой нечто среднее между сортировками с помощью включения и с помощью выбора. Фактически в пузырьковой сортировке нет ничего привлекательного, кроме названия. Шейкерная сортировка используется в случаях, когда известно, что элементы почти упорядочены — на практике это бывает весьма редко.

Сравнение методов сортировки

Как можно оценить относительные преимущества одного метода сортировки над другими при наличии их значительного разнообразия? Разумеется, скорость исполнения сортировки важна, но многие методы сортировки обладают уникальными характеристиками, влияющими на их применимость в том или ином случае. Таким образом, иногда среднюю скорость выполнения сортировки приходится сопоставлять с другими факторами. Можно назвать четыре критерия, которые следует использовать при выборе метода сортировки: средняя скорость сортировки, скорость в наилучшем и наихудшем случаях, естественно ли его поведение, сортировка элементов с совпадающими ключами.

Приведем оценку эффективности различных методов сортировки в соответствии с данными Н. Вирта (таблица 1). Как и раньше, N — число сортируемых элементов, а C и M соответственно число необходимых сравнений ключей и число обменов. Для всех прямых методов сортировки можно дать точные аналитические формулы. Столбцы *Min*, *Avg*, *Max* определяют соответственно минимальное, усредненное и максимальное по всем $N!$ перестановкам из N элементов значения.

Таблица 1. Сравнение прямых методов сортировки

Метод		*Min*	*Avg*	*Max*
Прямое включение	$C =$	$n-1$	$(n^2+n-2)/4$	$(n^2-n)/2-1$
	$M =$	$2(n-1)$	$(n^2-9n-10)/4$	$(n^2-3n-4)/2$
Прямой выбор	$C =$	$(n^2-n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
	$M =$	$3(n-1)$	$n(\ln n+0.57)$	$n^2/4+3(n-1)$
Прямой обмен	$C =$	$(n^2-n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
	$M =$	0	$0.75(n^2-n)$	$1.5(n^2-n)$

Приведем также время работы различных модификаций прямых методов сортировки.

В таблице 2 собраны времена работы различных вариаций методов сортировки в секундах, реализованных в системе Модула-2 на ЭВМ Lilith. Три столбца содержат времена сортировки уже упорядоченного массива,

случайной перестановки и массива, расположенного в обратном порядке. В начале приводятся цифры для 256 элементов, а потом — для 2048. Кроме того, заслуживают внимания следующие особенности:

1. Улучшение двоичного включения по сравнению с прямым включением действительно почти не дает, а в случае упорядоченного массива даже получается отрицательный эффект.

2. Пузырьковая сортировка определенно наихудшая из всех сравниваемых. Ее усовершенствованная версия, шейкерная сортировка, продолжает оставаться плохой по сравнению с прямым включением и прямым выбором (за исключением патологического случая уже упорядоченного массива).

3. Quicksort лучше в 2-3 раза, чем Heapsort. Она сортирует массив, расположенный в обратном порядке, практически с той же скоростью, что и уже упорядоченный.

Таблица 2. Время работы различных программ сортировки

Метод	Разновидность	Упорядоченный	Случайный	В обратном порядке
n=256				
Прямое включение	StraightInsertion	0.02	0.82	1.64
	BinaryInsertion	0.12	0.70	1.30
Прямой выбор	StraightSelection	0.94	0.96	1.18
Прямой обмен	BubbleSort	1.26	2.04	2.80
	ShakerSort	0.02	1.66	2.92
Улучшенный пр. включения	ShellSort	0.10	0.24	0.28
Улучшенный пр. выбора	HeapSort	0.20	0.20	0.20
Улучшенные пр. обмена	QuickSort	0.08	0.12	0.08
	NonRecQuickSort	0.08	0.12	0.08
	StraightMerge	0.18	0.18	0.18
n=2048				
Прямое включение	StraightInsertion	0.22	50.74	103.80
	BinaryInsertion	1.16	37.66	76.06
Прямой выбор	StraightSelection	58.18	58.34	73.46
Прямой обмен	BubbleSort	80.18	128.84	178.66
	ShakerSort	0.16	104.44	187.36
Улучшенный пр. включения	ShellSort	0.80	7.08	12.34
Улучшенный пр. выбора	HeapSort	2.32	2.22	2.12
Улучшенные пр. обмена	QuickSort	0.72	1.22	0.76
	NonRecQuickSort	0.72	1.32	0.80

Следует заметить, что если прямые методы сортировки обменом считаются самыми плохими, то улучшенные версии обменной сортировки являются самыми лучшими.

3. Схемы алгоритмов пузырьковой и шейкерной сортировок

На рисунке 1 представлена схема алгоритма пузырьковой сортировки, а на рисунке 2 — шейкерной сортировки.

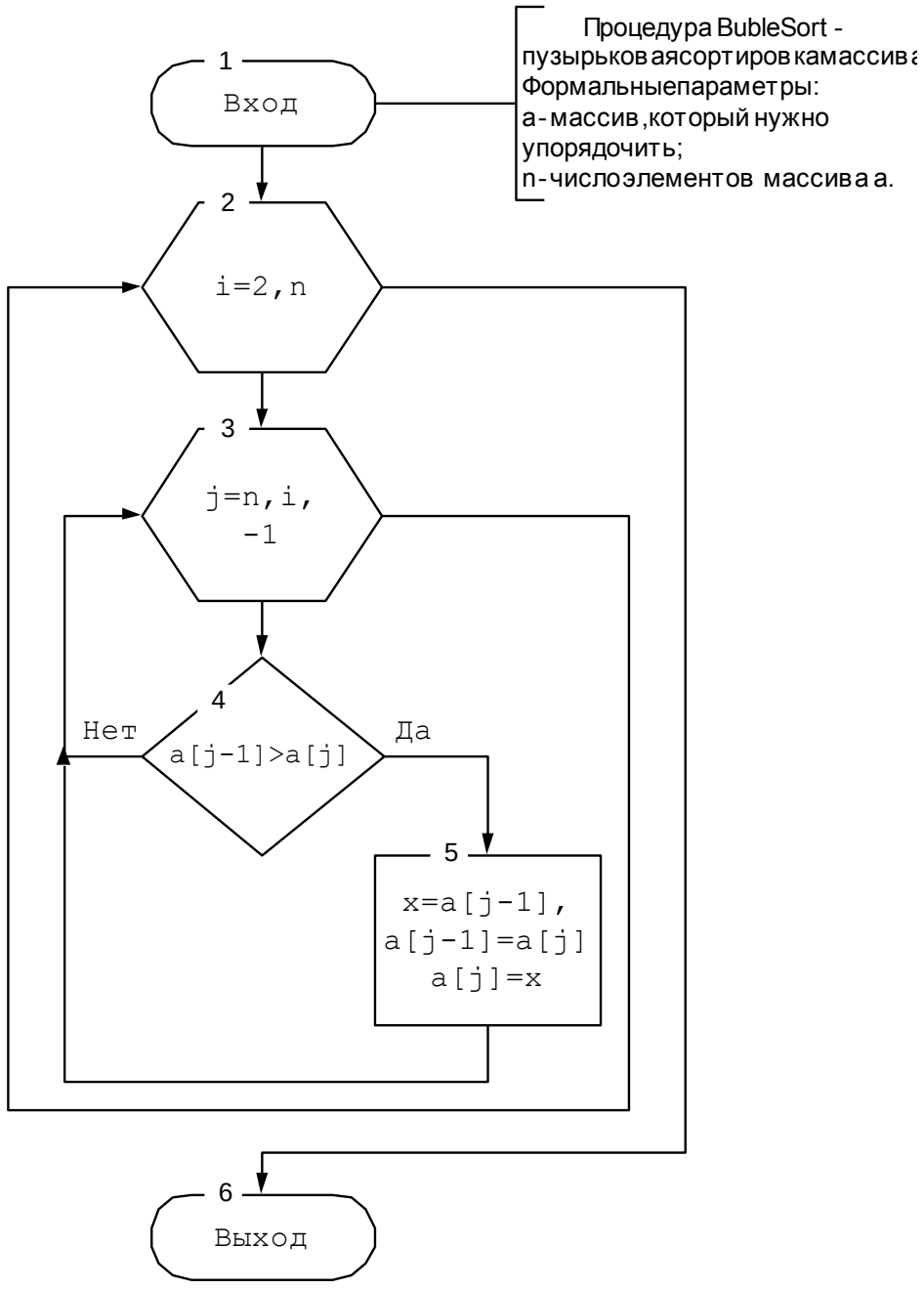


Рис. 1. Схема алгоритма пузырьковой сортировки.

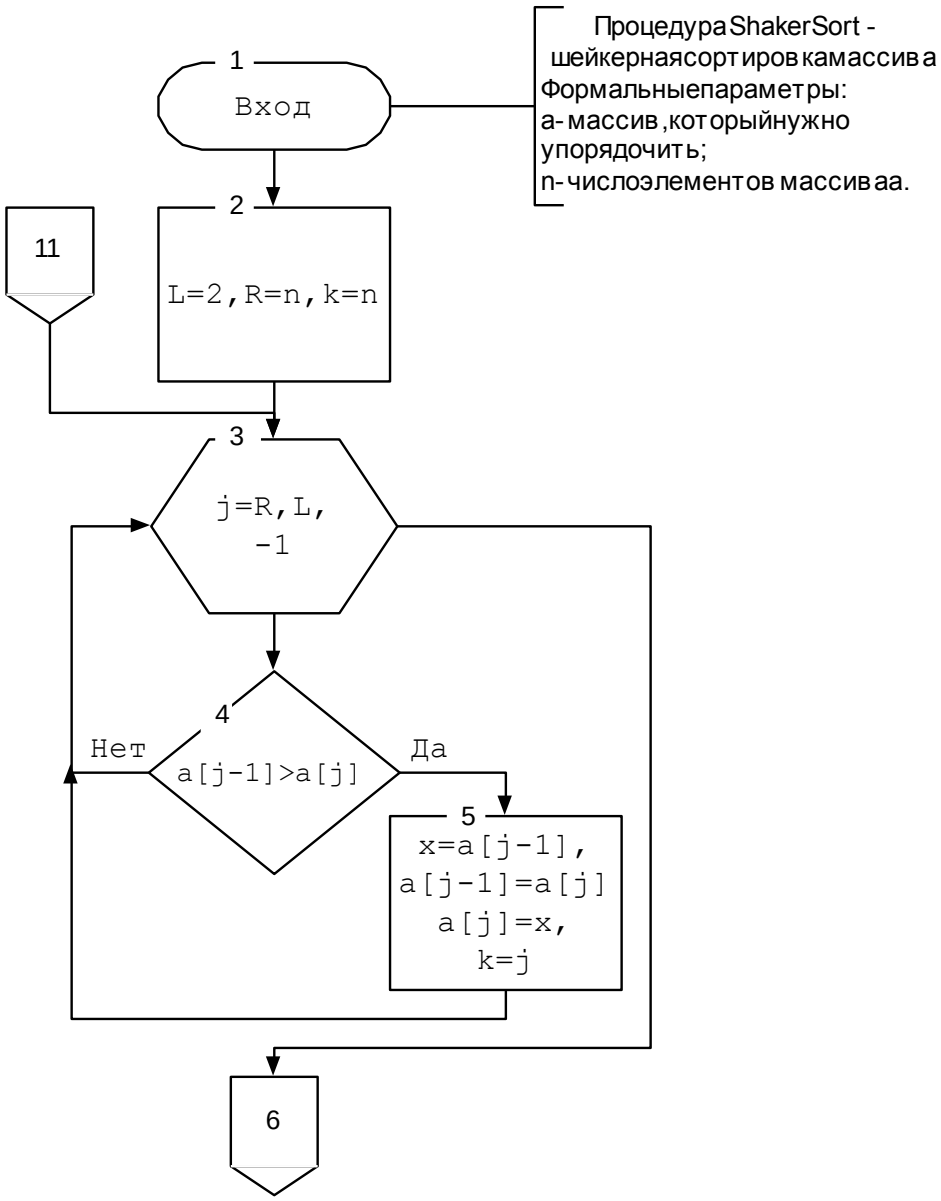
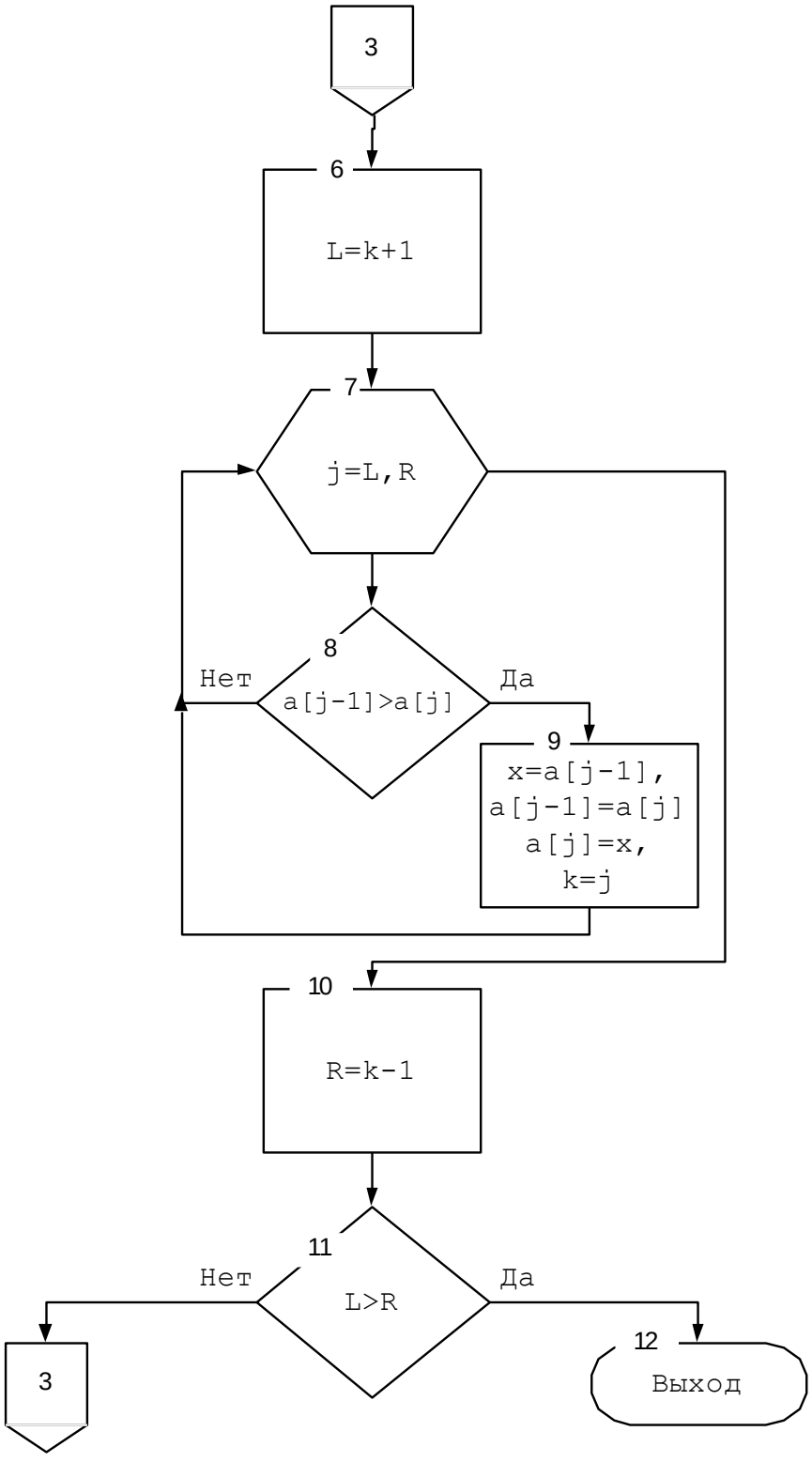


Рис. 2. Схема алгоритма шейкерной сортировки.



4. Пример программы на языке C/C++

Данная программа демонстрирует метод пузырьковой сортировки массива.

```
#include <iostream.h>
#include <stdlib.h>

main(void)
{
 const int n=20;
 float a[n];
 int i,j;
 float x;

 cout << "Упорядочивание последовательности, состоящей из 20";
 cout << "случайных чисел, с помощью пузырьковой сортировки";
 cout << "Несортированная последовательность:\n";
 for (i=0; i<n; i++)
  {
   a[i]=random(100);
   cout << a[i] << " ";
  }

 for (i=1; i<n; i++)
  {
   for (j=n-1; j>i-1; j--)
    {
     if (a[j-1]>a[j])
      {
       x=a[j-1];
       a[j-1]=a[j];
       a[j]=x;
      }
    }
  }

 cout << "\nОтсортированная последовательность:\n";
 for (i=0; i<n; i++)
  {
   cout << a[i] << " ";
  }

 return 0;
}
```

5. Варианты заданий

1. Реализовать метод пузырьковой сортировки и определить число необходимых перестановок и сравнений для уже упорядоченного массива, случайной перестановки и массива, расположенного в обратном порядке.

2. Реализовать метод шейкерной сортировки элементов и определить число необходимых перестановок и сравнений для уже упорядоченного массива, случайной перестановки и массива, расположенного в обратном порядке.
3. Реализовать два прямых метода сортировки с помощью обмена (пузырьковая и шейкерная) и сравнить число необходимых перестановок для уже упорядоченного массива, случайной перестановки и массива, расположенного в обратном порядке.
4. Упорядочить массив методом прямой обменной сортировки, состоящий из n^2 элементов (где n=1, 2, 3, ...) и вывести его в виде спирали (матрицы n на n).
   Пример:

   исходный массив для n=3:
   1 9 8 2 3 7 6 4 5
   после упорядочивания:
   1 2 3 4 5 6 7 8 9
   вывод:
   1 2 3
   8 9 4
   7 6 5
5. Задается массив, упорядоченный по возрастанию. В него добавляются три произвольных элемента. Используя метод пузырьковой сортировки, упорядочить массив по возрастанию.
6. Задается массив, упорядоченный по убыванию. В него добавляются пять произвольных элементов. Используя метод шейкерной сортировки, упорядочить массив по возрастанию.
7. Даны два массива. Построить третий массив, который включает элементы двух предыдущих. Отсортировать его по возрастанию, используя метод прямой сортировки обменом. Если встречаются два и более одинаковых элемента, удалить их, оставив один.
8. Даны два массива A и B, каждый из которых состоит из n элементов (n=1, 2, 3, ...). Построить третий массив C, таким образом, что $c_i=a_i+b_i$ (i=1,…,n). Отсортировать его по возрастанию, используя метод прямой обменной сортировки.
9. Реализовать метод «пузырька» прямой обменной сортировки, используя ввод данных из файла.
10. Реализовать шейкерный метод прямой обменной сортировки, используя ввод данных из файла.
11. Реализовать метод «пузырька» прямой обменной сортировки, используя вывод результатов в файл.
12. Реализовать шейкерный метод прямой обменной сортировки, используя вывод результатов в файл.
13. Дан массив, состоящий более чем из десяти элементов. Используя методы прямой сортировки обменом, упорядочить первую его половину по возрастанию, а вторую – по убыванию.

14. Даны два массива A и B, каждый состоит из n элементов (n=1,2,3, ...). Построить третий массив C, при условии, что $c_i=a_i \cdot b_i$ (i=1,...,n). Отсортировать его по убыванию, используя метод прямой обменной сортировки.
15. Даны два массива. Построить третий массив, который включает элементы двух предыдущих. Отсортировать его по убыванию, используя прямой метод сортировки обменом. Если встречаются два и более одинаковых элемента, оставить только один из них. Данные берутся из файла и записываются в файл.
16. Даны два массива A и B, каждый состоит из n элементов (n=1,2,3,...). Построить третий массив C, при условии $c_i=a_i-b_i$ (i=1,...,n). Отсортировать его по возрастанию, используя метод прямой обменной сортировки.
17. Реализовать метод пузырьковой сортировки и определить число необходимых перестановок и сравнений для уже упорядоченного массива, случайной перестановки и массива, расположенного в обратном порядке. Данные вводятся из файла, а результаты по выбору пользователя — либо в файл, либо на экран.
18. Реализовать метод шейкерной сортировки элементов и определить число необходимых перестановок и сравнений для уже упорядоченного массива, случайной перестановки и массива, расположенного в обратном порядке. Данные вводятся из файла, а результаты по выбору пользователя: либо в файл, либо на экран.
19. Реализовать два прямых метода сортировки с помощью обмена (пузырьковая и шейкерная) и сравнить число необходимых перестановок для уже упорядоченного массива, случайной перестановки и массива, расположенного в обратном порядке. Данные вводятся из файла, а результаты по выбору пользователя: либо в файл, либо на экран.
20. Реализовать метод прямой обменной сортировки и упорядочить по убыванию массив, содержащий суммы оценок студентов группы. На экран вывести список студентов и их средний балл, а в файл – список студентов, их оценки, сумму и средний балл.

6. Указания по составлению отчета

Отчет должен содержать постановку и описание задачи, алгоритм решения задачи, текст составленной программы на языке программирования C++, краткие выводы.

Лабораторная работа №4 Улучшенные методы сортировки обменом

1. Цель и задачи работы

Изучение улучшенных методов сортировки обменом, выделение основных достоинств и недостатков данных методов. Написание программы с использованием изученного материала.

2. Теоретические сведения

Необходимость улучшенных методов

Одной из основных задач программирования является сортировка массивов, которая используются во многих задачах. При изучении методов сортировки было установлено, что все методы можно разбить на 3 основных категории:

- сортировка с помощью включения (by insertion)
- сортировка с помощью выделения (by selection)
- сортировка с помощью обменов (by exchange)

Сортировки с помощью обменов являются наиболее простыми с точки зрения программирования. Они также наиболее понятны для пользователя. Алгоритм прямого обмена основывается на сравнении и смене мест пары соседних элементов и продолжении этого процесса до тех пор, пока не будут упорядочены все элементы. К данным методам относятся шейкер-сортировка и сортировка методом пузырька. Эти два метода называются простыми методами сортировки с помощью обменов.

Простые методы сортировки обмена достаточно эффективны при небольшом количестве элементов. Но уже при их увеличении время работы значительно увеличивается. Это связано с тем, что простые методы перемещают каждый элемент на одну позицию на каждом элементарном шаге, поэтому они требуют порядка n^2 таких шагов. Следовательно, любое улучшение таких методов должно основываться на принципе пересылки элементов за один цикл на большое расстояние.

Необходимость нахождения методов сортировки, которые за наименьшее возможное время способны упорядочить массив – основная задача при изучении сортировки массивов. Несомненно, данные алгоритмы должны быть и будут сложнее, чем алгоритмы простых методов, но в современном мире скорость имеет огромное значение и получение более быстрой информации является одним из основных, если не основным фактором в наши дни. Следовательно, ставилась задача получить новые улучшенные методы сортировки.

И такие методы сортировки были получены. Улучшенным методом сортировки с помощью обмена является метод *быстрой сортировки (Quicksort)* .

Быстрая сортировка (Quicksort)

Из трех алгоритмов прямой (строгой) сортировки метод пузырька являлся самым неэффективным. Конечно, следовало ожидать, что улучшенный метод должен работать быстрее, но результат превзошел все ожидания. Данный метод оказался самым быстрым из ныне известных методов сортировки массивов. Его производительность столь впечатляюща, что изобретатель данного метода Ч. Хоар назвал его *быстрой сортировкой* (Quicksort).

В Quicksort исходят из того соображения, что для достижения наилучшей эффективности сначала лучше производить перестановки на большие расстояния. Рассмотрим следующий алгоритм. Пусть мы имеем одномерный массив A. Выберем наугад какой-либо элемент (назовем его x) и будем просматривать слева наш массив, пока не обнаружим элемент $a_i > x$, затем будем просматривать массив справа, пока не встретим $a_j < x$. Теперь поменяем местами два этих элемента и продолжим наш процесс просмотра и обмена, пока оба просмотра не встретятся где-то в середине массива. В результате массив окажется разбитым на левую часть, с ключами меньше (или равными x), и правую – с ключами большими (или равными x).

Для примера возьмем массив:

44 55 12 42 94 06 18 67

В качестве x в данном примере выберем 42. В результате мы получим два обмена: 44 и18, затем 55 и 06.

18 06 12 42 94 55 44 67

Последние значения индексов: i=5; j=3. Ключи $a_1 \ldots a_{i-1}$ меньше или равны ключу x=42, а ключи $a_{j+1} \ldots a_n$ больше или равны x. Следовательно, существуют две части, а именно

Ak: $1<=k<i : a_k<=x$

Ak: $j<k<=n : x <= a_k$

Описанный алгоритм очень прост и эффективен. Но напомним, что наша цель — не только провести разделение на части исходного массива элементов, но и отсортировать его. Сортировку от разделения отличает лишь небольшой шаг: нужно применить этот процесс к получившимся двум частям, затем к частям частей, и так до тех пор, пока каждая из частей не будет состоять из одного единственного элемента.

Анализ быстрой сортировки

Для того чтобы проанализировать свойства быстрой сортировки, мы должны разобраться, как идет процесс разбиения. После выбора границы *x* разбиению подвергается весь массив. Таким образом, выполняется ровно *n*

сравнений. Число обменов можно оценить при помощи следующего вероятностного рассуждения.

При заданной границе значений *x* ожидаемое число операций обмена равно числу элементов в левой части разделяемой последовательности, т.е. n–1, умноженному на вероятность того, что при обмене каждый такой элемент попадает на своё место. Обмен происходит, если этот элемент находился в правой части. Вероятность этого равна (n – x + 1) / n. Поэтому ожидаемое число обменов есть среднее этих ожидаемых значений для всех возможных границ x.

$$M = \frac{1}{n}\sum_{x=1}^{n}\frac{x-1}{n}(n-x+1) = \frac{n}{6} - \frac{1}{6n}$$

Если повезет, и мы всегда выбираем в качестве границы медиану (*медианой* для n элементов называется элемент меньший или равный половине из n элементов и больший или равный другой половине), то в этом случае процесс разделения расщепляется на две половины и для сортировки требуется всего *log n* проходов. В результате общее число сравнений равно *n*log n* , а общее число обменов *n*log (n)/6*. Нельзя, конечно, ожидать, что мы каждый раз будем выбирать медиану. Вероятность такого выбора составляет только 1/n. Но средняя производительность Quicksort при случайном выборе границы отличается от упомянутого оптимального варианта лишь коэффициентом *2*ln(2)*.

Однако Quicksort имеет и недостатки. Главный из них — недостаточно высокая производительность при небольших n. Однако этот недостаток присущ всем усовершенствованным методам. Но перед другими усовершенствованными методами этот имеет то преимущество, что для обработки небольших частей в него можно легко включить какой-либо из прямых методов сортировки.

Также существует вопрос о самом плохом случае. Как в этом случае поведет себя Quicksort? К несчастью, ответ на этот вопрос не утешителен. Разберем случай, когда каждый раз выбирается наибольшее из всех значений в указанной части. Тогда на каждом этапе сегмент будет расщепляться на левую часть, состоящую из n-1 элемента и правую часть, состоящую из одного элемента. В результате потребуется n (а не log(n)) разделений и наихудшая производительность метода будет порядка n^2.

Явно видно, что главное заключается в выборе элемента для сравнения – x . Мы предложили выбирать средний элемент, хотя с тем же успехом можно выбирать первый или последний элемент. В этих случаях хуже всего, если массив был первоначально упорядочен. Для среднего элемента упорядочение массива является оптимальным вариантом. Средняя производительность при выборе среднего элемента чуточку улучшается. Сам Хоар предлагает x выбирать случайно, а для небольших выборок, вроде всего трех ключей, останавливаться на медиане. Такой разумный выбор мало влияет на среднюю производительность Quicksort, но зато значительно ее улучшает в наихудших случаях.

Нахождение медианы

Медианой для n элементов называется элемент, меньший или равный половине из n элементов и больший или равный другой половине. Например, медиана для элементов 16 12 99 95 18 87 10 равна 18. Задача поиска медианы тесно связана проблемой сортировки, поскольку очевидный метод определения медианы заключается в том, чтобы отсортировать n элементов, а затем выбрать средний элемент. Однако существует алгоритм, позволяющий отыскивать среди n элементов k-ое наименьшее число. В этом случае поиск медианы – просто частный случай k =n/2. Этот алгоритм был предложен Ч.Хоаром и состоит в следующем.

Сначала применяется операция разделения из Quicksort с L=1 и R=n, в качестве разделяющего значения x берётся a_k. В результате получаем индексы i и j, удовлетворяющие таким условиям:

1. $a_h < x$ для всех h<i
2. $a_h > x$ для всех h>j
3. i > j

При этом мы сталкиваемся с одним из трёх случаев:

1. Разделяющее значение x было слишком мало, и граница между двумя частями лежит ниже нужной величины k. Процесс разделения повторяется для элементов $a_i ... a_R$.
2. Выбранная граница x была слишком большой. Операции разделения следует повторить для элементов $a_L ... a_j$.
3. j < k < i : элемент a_k разделяет массив на две части в нужной пропорции, следовательно, это то, что нужно.

Процессы разделения повторяются до тех пор, пока не возникнет третий случай.

Сравнение методов сортировки массивов

Здесь представлена таблица, в которой получено время работы улучшенных методов различных сортировок, полученных Николасом Виртом при n=2048.

Таблица 1. Сравнение различных методов сортировки

Метод	Упорядоченный	Случайный	В обрат. порядке
Улучшенный ShellSort пр. Включения	0.80	7.08	12.34
Улучшенный HeapSort пр. Выбора	2.32	2.22	2.12
Улучшенные QuickSort пр. Обмена	0.72	1.22	0.76

Как видно из таблицы, улучшенный метод прямого обмена является самым эффективным.

3. Пример программы на языке C/C++

Данная программа демонстрирует сортировку массива не рекурсивным методом быстрой сортировки

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>
#include <iostream.h>;


void main()
{
 const max=10;

 double arr[max];
 int k,st,end,num,n,s;
 int i,j,x;
 double buf,xbuf;

 clrscr();
 randomize();


 for (k=0;k<max;k++)
  {
   arr[k]=0;
   arr[k]=random(256);
  }
 for (k=0;k<max;k++)
  {
   cout<<' '<<arr[k];
  }
 cout<<'\n';

 st=0;
 end=max-1;
 num=0;
 n=1;

 do
  {
   for (k=0;k<n;k++)
    {
     //--------------------------------
       i=st,j=end,x=int((st+end)/int(2));
       xbuf=arr[x];

       do
     {
      while (arr[i]<xbuf)
       {i++;};
```

```
      while (arr[j]>xbuf)
       {j--;};
      if (i<=j)
       {
        buf=arr[i];
        arr[i]=arr[j];
        arr[j]=buf;
        i++;j--;

       }
     }
       while (i<j);

     //------------------------------

     st=end;
     if ((max-end-num)==2)
      {end=end+num+1;}
     else if ((max-end-num)==0)
        {end=end+num-1;}
       else
        {end=end+num;};
     if (end>(max-1)) {end=max;};
    }
   n++;
   st=0;
   end=int(max/n);
   num=end;
  }
 while (n<=max);

 for (k=0;k<max;k++)
  {
   cout<<' '<<arr[k];
  }
}
```

4. Варианты заданий

1. Вводится три массива с одинаковым количеством элементов: первый упорядоченный по возрастанию, второй — по убыванию, третий в произвольном порядке. Отсортировать их по возрастанию, используя метод быстрой сортировки. Сделать вывод о скорости работы (считать количество перестановок элементов). В качестве ключа взять медиану массива.
2. Упорядочить массив методом быстрой сортировки, состоящий из n^2 элементов (где n=1 2 3 ...). Вывести его в виде спирали (матрица n на n).

Пример:
N=3

7 2 –2 -1 0 1 9 –9 5
упорядоченный
-9 -2 –1 0 1 2 5 7 9
Вывод:
-9 –2 -1
7 9 0
5 2 1

3. Даны два массива с одинаковым количеством элементов. Отсортировать их по возрастанию, используя методы быстрой сортировки. Для первого массива — рекурсивный метод, для второго — нерекурсивный. Сделать вывод о скорости работы при разных методах (считать количество перестановок элементов).
4. Задается массив, упорядоченный по возрастанию. В него добавляются три произвольных элемента. Используя метод быстрой сортировки, упорядочить массив по возрастанию.
5. Задается массив, упорядоченный по убыванию. В него добавляются пять произвольных элементов. Используя метод быстрой сортировки, упорядочить массив по возрастанию.
6. Вводится три массива с одинаковым количеством элементов: первый – упорядоченный по возрастанию, второй – по убыванию, третий – в произвольном порядке. Отсортировать их по возрастанию, используя метод быстрой сортировки. Сделать вывод о скорости работы (считать количество перестановок элементов). В качестве ключа взять первый элемент массива.
7. Даны два массива. Построить третий массив, который включает элементы двух предыдущих. Отсортировать его по возрастанию, используя метод быстрой сортировки. Если встречаются два и более одинаковых элемента, удалить их, оставив один.
8. Даны два массива a и b, каждый состоит из n элементов (n=1 2 3 ...). Построить третий массив c, при условии $c_i=a_i+b_i$. Отсортировать его по возрастанию, используя метод быстрой сортировки.
9. Вводятся три массива с одинаковым количеством элементов: первый – упорядоченный по возрастанию, второй — по убыванию, третий – в произвольном порядке. Отсортировать их по возрастанию, используя метод быстрой сортировки. Сделать вывод о скорости работы (считать количество перестановок элементов). В качестве ключа взять последний элемент массива.
10. Дан массив. Отсортировать его по убыванию, используя методы быстрой сортировки. Для первого случая — рекурсивный метод, для второго — нерекурсивный. Сделать вывод о скорости работы при разных методах (считать количество перестановок элементов).
11. Реализовать рекурсивный метод быстрой сортировки, используя ввод данных из файла.
12. Реализовать нерекурсивный метод быстрой сортировки, используя ввод данных из файла. Результат записать в другой файл.

13. Отсортировать в первом случае по возрастанию, во втором — по убыванию номера телефонов студентов твоей группы. Использовать методы быстрой сортировки. Сделать вывод о скорости работы метода (считать количество перестановок элементов).
14. Дан массив из более, чем 15 элементов. Используя методы быстрой сортировки, упорядочить первую его половину по возрастанию, а вторую — по убыванию.
15. Отсортировать в первом случае по возрастанию, во втором — по убыванию номера домов студентов твоей группы. Использовать методы быстрой сортировки. Сделать вывод о скорости работы метода (считать количество перестановок элементов).
16. Дан массив, упорядоченный по возрастанию. Написать программу, которая позволяет пользователю заменить три элемента. Используя методы быстрой сортировки, упорядочить, получившийся массив по возрастанию и по убыванию. Сделать вывод о скорости работы метода (считать количество перестановок элементов).
17. Дан массив, упорядоченный по убыванию. Написать программу, которая позволяет пользователю добавить три элемента. Используя методы быстрой сортировки, упорядочить, получившийся массив по возрастанию и по убыванию. Сделать вывод о скорости работы метода (считать количество перестановок элементов).
18. Даны два массива a и b, каждый состоит из n элементов (n=1 2 3 ...). Построить третий массив c , при условии $c_i=a_i*b_i$. Отсортировать его по возрастанию, используя метод быстрой сортировки.
19. Даны два массива. Построить третий массив, который включает элементы двух предыдущих. Отсортировать его по возрастанию, используя метод быстрой сортировки. Если встречаются два и более одинаковых элемента, удалить их. Данные берутся из файла и записываются в файл.
20. Даны два массива a и b, каждый состоит из n элементов (n=1 2 3 ...). Построить третий массив c , при условии $c_i=a_i–b_i$. Отсортировать его по возрастанию, используя метод быстрой сортировки.

5. Указания по составлению отчета

Отчет должен содержать постановку и описание задачи, алгоритм решения задачи, текст составленной программы на языке программирования C/C++, краткие выводы.

Лабораторная работа №5 Сортировка вставками

1. Цель и задачи работы

Изучение метода сортировки массивов методом включений. Написание программы, демонстрирующей изученные принципы.

2. Теоретические сведения

Сортировка — это процесс, позволяющий упорядочить множество подобных данных в возрастающем или убывающем порядке. Сортировка представляет собой один из наиболее приемлемых с интеллектуальной точки зрения алгоритмов, поскольку процесс хорошо определен. Алгоритмы сортировки хорошо исследованы и изучены. К сожалению, по этой причине сортировку иногда принимают как некую данность.

В общем, сортировку следует понимать как процесс перегруппировки заданного множества объектов в некотором определенном порядке. Цель сортировки — облегчить последующий поиск элементов в таком отсортированном множестве. Это почти универсальная, фундаментальная деятельность. Отсортированные объекты можно встретить на каждом шагу: в телефонных книгах, в списках подоходных налогов, в оглавлениях книг, в библиотеках, в словарях, на складах – почти везде, где надо искать хранимые объекты. Даже малышей учат держать вещи «в порядке», и они уже сталкиваются с некоторыми видами сортировок задолго до того, как познакомятся с азами арифметики. Между сортировкой и арифметикой нет никакой видимой связи, да, по-видимому, и невидимой нет. Сортировка — некий «первичный» процесс человеческой деятельности.

Таким образом, разговор о сортировке вполне уместен и важен, если речь идет об обработке данных. Что может легче сортироваться, чем данные? Тем не менее, первоначальный интерес к сортировке основывается на том, что при построении алгоритмов часто сталкиваются со многими фундаментальными приемами. Почти не существует методов, с которыми не приходится встречаться при обсуждении этой задачи. В частности, сортировка — это идеальный объект для демонстрации огромного разнообразия алгоритмов, все они изобретены для одной и той же задачи, многие в некотором смысле оптимальны, большинство имеют достоинства. Поэтому это еще и идеальный объект, демонстрирующий необходимость анализа производительности алгоритмов. К тому же на примерах сортировок можно показать, как путем усложнения алгоритма, хотя под рукой и есть уже очевидные методы, можно добиться значительного выигрыша в эффективности.

Вот некоторые из наиболее важных применений сортировки:

1. Решение задачи «группировки», когда нужно собрать вместе все элементы с одинаковым значением некоторого признака. Допустим, имеется 10000 элементов, расположенных в случайном порядке, причем значения многих из них равны; и предположим, нам нужно переупорядочить файл так, чтобы элементы с равными значениями занимали соседние позиции в файле. Это, по существу, задача «сортировки» в широком смысле слова, и она легко может быть решена путем сортировки файла в. узком смысле слова, а именно расположением элементов в неубывающем порядке $v_1 \le v_2 \le \ldots \le v_{10000}$.

2. Если два или более файла отсортировать в одном и том же порядке, то можно отыскать в них все общие элементы за один последовательный просмотр всех файлов, без возвратов. Оказывается, что, как правило, гораздо экономнее просматривать список последовательно, а не перескакивая с места на место случайным образом, если только список не настолько мал, что он целиком помещается в оперативной памяти. Сортировка позволяет использовать последовательный доступ к большим файлам в качестве приемлемой замены прямой адресации.

Теперь четко определим задачу и введем соответствующую терминологию. Пусть надо упорядочить N элементов

$$R_1, R_2, \ldots, R_N.$$

Назовем их *записями,* а всю совокупность N записей назовем *файлом.* Каждая запись R_j имеет *ключ* K_j, который и управляет процессом сортировки. Помимо ключа, запись может содержать дополнительную «сопутствующую информацию», которая не влияет на сортировку, но всегда остается в этой записи.

Отношение порядка «<» на множестве ключей вводится таким образом, чтобы для любых трех значений ключей a, b, c выполнялись следующие условия:

1) справедливо одно и только одно из соотношений: $a<b$, $a=b$, $b<a$ (закон трихотомии);

2) если $a<b$ и $b<c$, то $a<c$ (закон транзитивности).

Эти два свойства определяют математическое понятие *линейного упорядочения,* называемого еще *совершенным упорядочением.* Любое множество с отношением «<», удовлетворяющим вышеперечисленным свойствам, поддается сортировке большинством методов.

Задача сортировки — найти такую перестановку записей $p(1), p(2),\ldots,p(N)$, после которой ключи расположились бы в неубывающем порядке:

$$K_{p(1)}, K_{p(2)}, \ldots, K_{p(N)}.$$

Сортировка называется *устойчивой,* если она удовлетворяет дополнительному условию, что записи с одинаковыми ключами остаются в прежнем порядке, т. е.

$$p(i)<p(j), \text{ если } K_{p(i)}=K_{p(j)} \text{ и } i<j.$$

Обычно сортировку подразделяют на два класса: *внутреннюю,* когда все записи хранятся в быстрой оперативной памяти, и *внешнюю,* когда все они там не помещаются. При внутренней сортировке имеются более гибкие возможности для построения структур данных и доступа к ним, внешняя же показывает, как поступать в условиях сильно ограниченного доступа.

Достаточно хороший общий алгоритм затрачивает на сортировку N записей время порядка $N\log(N)$; при этом требуется около $\log(N)$ «проходов» по данным. Это минимальное время. Так, если удвоить число записей, то и время при прочих равных условиях возрастет немногим более чем вдвое.

Сортировка массивов

Основное условие: выбранный метод сортировки массивов должен экономно использовать доступную память. Это предполагает, что перестановки, приводящие элементы в порядок, должны выполняться *на том же месте*, то есть методы, в которых элементы из массива a передаются в результирующий массив b, представляют существенно меньший интерес.

Хорошей мерой эффективности может быть C — число необходимых сравнений ключей и M — число пересылок (перестановок) элементов. Эти числа суть функции от числа сортируемых элементов. Как говорилось выше, если сортируется N элементов, то хорошим считается такой алгоритм сортировки, который требует порядка $N\log(N)$ сравнений. Для начала стоит рассмотреть самые простые алгоритмы, которые называют *прямыми*, требующими порядка N^2 сравнений. Прямые методы особенно удобны для объяснения характерных черт основных принципов большинства сортировок, программы этих методов легко понимать, и они коротки. Усложненные же методы требуют небольшого числа операций, но эти операции обычно сами более сложны, и поэтому для достаточно малых N прямые методы оказываются быстрее, хотя при больших N их использовать, конечно, не следует.

Методы сортировки *«на том же месте»* можно разбить в соответствии с определяющими их принципами на три основные категории:

1. Сортировки с помощью включения (by insertion)
2. Сортировки с помощью выделения (by selection)
3. Сортировка с помощью обменов (by exchange)

Сортировка методом включений

Разделим условно все записи на готовую последовательность:

$$R_1, R_2, \ldots, R_{i-1};$$

и входную (исходную) последовательность:

$R_i, R_{i+1}, \ldots, R_n$.

На каждом шаге, начиная с $i=2$ и увеличивая i на единицу, будем извлекать очередной i-й элемент из входной последовательности, и вставлять его в готовую последовательность, при этом элемент вставляется на нужное место.

В таблице 1 представлен пример процесса сортировки с помощью включений восьми случайно выбранных чисел.

Таблица 1. Пример сортировки с помощью вставок

№ итерации	*Последовательность*							
Исходный	*44*	*55*	*12*	*42*	*94*	*18*	*06*	*67*
2	*44*	*55*	*12*	*42*	*94*	*18*	*06*	*67*
3	*12*	*44*	*55*	*42*	*94*	*18*	*06*	*67*
4	*12*	*42*	*44*	*55*	*94*	*18*	*06*	*67*
5	*12*	*42*	*44*	*55*	*94*	*18*	*06*	*67*
6	*12*	*18*	*42*	*44*	*55*	*94*	*06*	*67*
7	*06*	*12*	*18*	*42*	*44*	*55*	*94*	*67*
8	*06*	*12*	*18*	*42*	*44*	*55*	*67*	*94*

При поиске подходящего места для текущего элемента удобно чередовать сравнения и движения по последовательности, т.е. текущий элемент сравнивается с очередным элементом R_j, а затем либо текущий элемент вставляется на свободное место, либо R_i сдвигается вправо, и процесс «уходит» влево.

При этом, пересылая текущий элемент с места l на место k, мы заполняем место l элементом с индексом $(l-1)$, а пустое место элемента $(l-1)$ элементом с индексом $(l-2)$, и так далее, до бывшего элемента с индексом k, который становится на место $(k+1)$. Другими словами сдвигаем последовательность вправо.

Данное «просеивание» может закончиться в двух случаях:

1. Найден элемент R_i с ключом, меньшим, чем ключ текущего элемента.

2. Достигнут левый край готовой последовательности.

Эти два условия окончания цикла дает возможность рассмотрения фиктивного элемента — барьера. Этот элемент может быть не включен в сортировку, что приведет к ошибке алгоритма или к ошибочной сортировке. Для решения этой проблемы достаточно расширить диапазон индексов на одну ячейку в описании элементов R_i.

Алгоритм сортировки простыми включениями легко можно улучшить, пользуясь тем, что готовая последовательность, в которую нужно вставить новый элемент, уже упорядочена. Поэтому место включения можно найти гораздо быстрее. Очевидно, что здесь можно применить бинарный поиск, который исследует средний элемент готовой последовательности и продолжает деление пополам, пока не будет найдено

место включения. Модифицированный алгоритм сортировки называется *сортировкой бинарными включениями*.

Анализ сортировки прямыми и бинарными включениями

При сортировке прямыми включениями число сравнений ключей (C_i) при *i-м* просеивании составляет самое большое *(i-1)*, самое меньшее — 1; если предположить, что все перестановки из N ключей равновероятны, то среднее число сравнений — *i/2*. Число пересылок (присваиваний элементов) M_i равно (C_i+2) (учитывая барьер). Поэтому общее число сравнений и пересылок можно найти по формулам:

$$C_{min} = N-1 \qquad M_{min} = 3(N-1)$$
$$C_{avg} = \tfrac{1}{4}(N^2+N-2) \qquad M_{avg} = \tfrac{1}{4}(N^2+9N-10)$$
$$C_{max} = \tfrac{1}{4}(N^2+N-4) \qquad M_{max} = \tfrac{1}{2}(N^2+3N-4)$$

Наименьшие числа появляются, если элементы с самого начала упорядочены, а наихудший (наибольшее число) случай встречается, если элементы расположены в обратном сортировке порядке. Заметим, что данный алгоритм описывает так называемую устойчивую сортировку, т.е. оставляет неизменным порядок следования элементов с одинаковыми значениями.

При сортировке бинарными включениями в конце поиска интервал должен быть единичной длины; значит, деление его пополам производится $i*\log_2(i)$ раз. Таким образом:

$$C = \sum_{i=1}^{n} [\log_2 i].$$

Количество сравнений не зависит от исходного порядка элементов. Но из-за округления при делении интервала поиска пополам действительное число сравнений для i элементов может быть на 1 больше ожидаемого. Природа этого «перекоса» такова, что в результате места′ включения в нижней части находятся в среднем несколько быстрее, чем в верхней части. Это дает преимущество в тех случаях, когда элементы изначально далеки от правильного порядка. На самом же деле минимальное число сравнений требуется, если элементы вначале расположены в обратном порядке, а максимальное — если они уже упорядочены. Следовательно, это случай неестественного поведения алгоритма сортировки:

$$C = N(\log_2 N - \log_2 e \pm 0{,}5).$$

К сожалению, улучшение, которое мы получаем, используя метод бинарного поиска, касается только числа сравнений, а не числа необходимых пересылок. В действительности, поскольку пересылка элементов, т.е. ключей и сопутствующей информации, обычно требует значительно больше времени, чем сравнение двух ключей, то это

улучшение ни в коей мере не является решающим: важный показатель M (число пересылок элементов) по-прежнему остается порядка N^2. И в самом деле, пересортировка уже рассортированного массива занимает больше времени, чем при сортировке простыми включениями с последовательным поиском! Этот пример показывает, что «очевидное улучшение» часто оказывается намного менее существенным, чем кажется вначале, и в некоторых случаях может на самом деле оказаться ухудшением.

Сравнение методов сортировки

Как можно оценить относительные преимущества одного метода сортировки над другими при наличии их значительного разнообразия? Разумеется, скорость исполнения сортировки важна, но многие методы сортировки обладают уникальными характеристиками, влияющими на их применимость в том или ином случае. Таким образом, иногда среднюю скорость выполнения сортировки приходится сопоставлять с другими факторами. В качестве критериев, которые следует использовать при выборе метода сортировки, можно назвать следующие четыре: какова средняя скорость сортировки? Какова скорость в наилучшем и наихудшем случаях? Естественно или нет его поведение? Выполняет ли он сортировку элементов с совпадающими ключами?

Приведем оценку эффективности различных методов сортировки в соответствии с данными Н. Вирта (таблица 2). Как и раньше, N — число сортируемых элементов, а C и M соответственно число необходимых сравнений ключей и число обменов. Для всех прямых методов сортировки можно дать точные аналитические формулы. Столбцы *Min*, *Avg*, *Max* определяют соответственно минимальное, усредненное и максимальное по всем $N!$ перестановкам из N элементов значения.

Таблица 2. Сравнение прямых методов сортировки

Метод		*Min*	*Avg*	*Max*
Прямое включение	$C =$	$n-1$	$(n^2+n-2)/4$	$(n^2-n)/2-1$
	$M =$	$2(n-1)$	$(n^2-9n-10)/4$	$(n^2-3n-4)/2$
Прямой выбор	$C =$	$(n^2-n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
	$M =$	$3(n-1)$	$n(\ln n+0.57)$	$n^2/4+3(n-1)$
Прямой обмен	$C =$	$(n^2-n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
	$M =$	0	$0.75(n^2-n)$	$1.5(n^2-n)$

Приведем также время работы различных модификаций прямых методов сортировки.

В таблице 3 собраны времена работы различных вариаций методов сортировки в секундах, реализованных в системе Модула-2 на ЭВМ Lilith. Три столбца содержат времена сортировки уже упорядоченного массива, случайной перестановки и массива, расположенного в обратном порядке. В

начале приводятся цифры для 256 элементов, а потом — для 2048. Кроме того, заслуживают внимания следующие особенности.

1. Улучшение двоичного включения по сравнению с прямым включением действительно почти не дает, а в случае упорядоченного массива даже получается отрицательный эффект.

2. Пузырьковая сортировка определенно наихудшая из всех сравниваемых.

Ее усовершенствованная версия, шейкерная сортировка, продолжает оставаться плохой по сравнению с прямым включением и прямым выбором (за исключением патологического случая уже упорядоченного массива).

3. Quicksort лучше в 2-3 раза, чем Heapsort. Она сортирует массив, расположенный в обратном порядке, практически с той же скоростью, что и уже упорядоченный.

Таблица 3. Время работы различных программ сортировки

Метод	Разновидность	Упорядоченный	Случайный	В обратном порядке
n=256				
Прямое включение	StraightInsertion	0.02	0.82	1.64
	BinaryInsertion	0.12	0.70	1.30
Прямой выбор	StraightSelection	0.94	0.96	1.18
Прямой обмен	BubbleSort	1.26	2.04	2.80
	ShakerSort	0.02	1.66	2.92
Улучшенный пр. включения	ShellSort	0.10	0.24	0.28
Улучшенный пр. выбора	HeapSort	0.20	0.20	0.20
Улучшенные пр. обмена	QuickSort	0.08	0.12	0.08
	NonRecQuickSort	0.08	0.12	0.08
	StraightMerge	0.18	0.18	0.18
n=2048				
Прямое включение	StraightInsertion	0.22	50.74	103.80
	BinaryInsertion	1.16	37.66	76.06
Прямой выбор	StraightSelection	58.18	58.34	73.46
Прямой обмен	BubbleSort	80.18	128.84	178.66
	ShakerSort	0.16	104.44	187.36
Улучшенный пр. включения	ShellSort	0.80	7.08	12.34
Улучшенный пр. выбора	HeapSort	2.32	2.22	2.12
Улучшенные пр. обмена	QuickSort	0.72	1.22	0.76
	NonRecQuickSort	0.72	1.32	0.80

Следует заметить, что, если прямые методы сортировки обменом считаются самыми плохими, то улучшенные версии обменной сортировки являются самыми лучшими.

3. Схемы алгоритмов сортировок прямым и бинарным включением

На рисунке 1 представлена схема алгоритма сортировки прямым включением, а на рисунке 2 — бинарным включением.

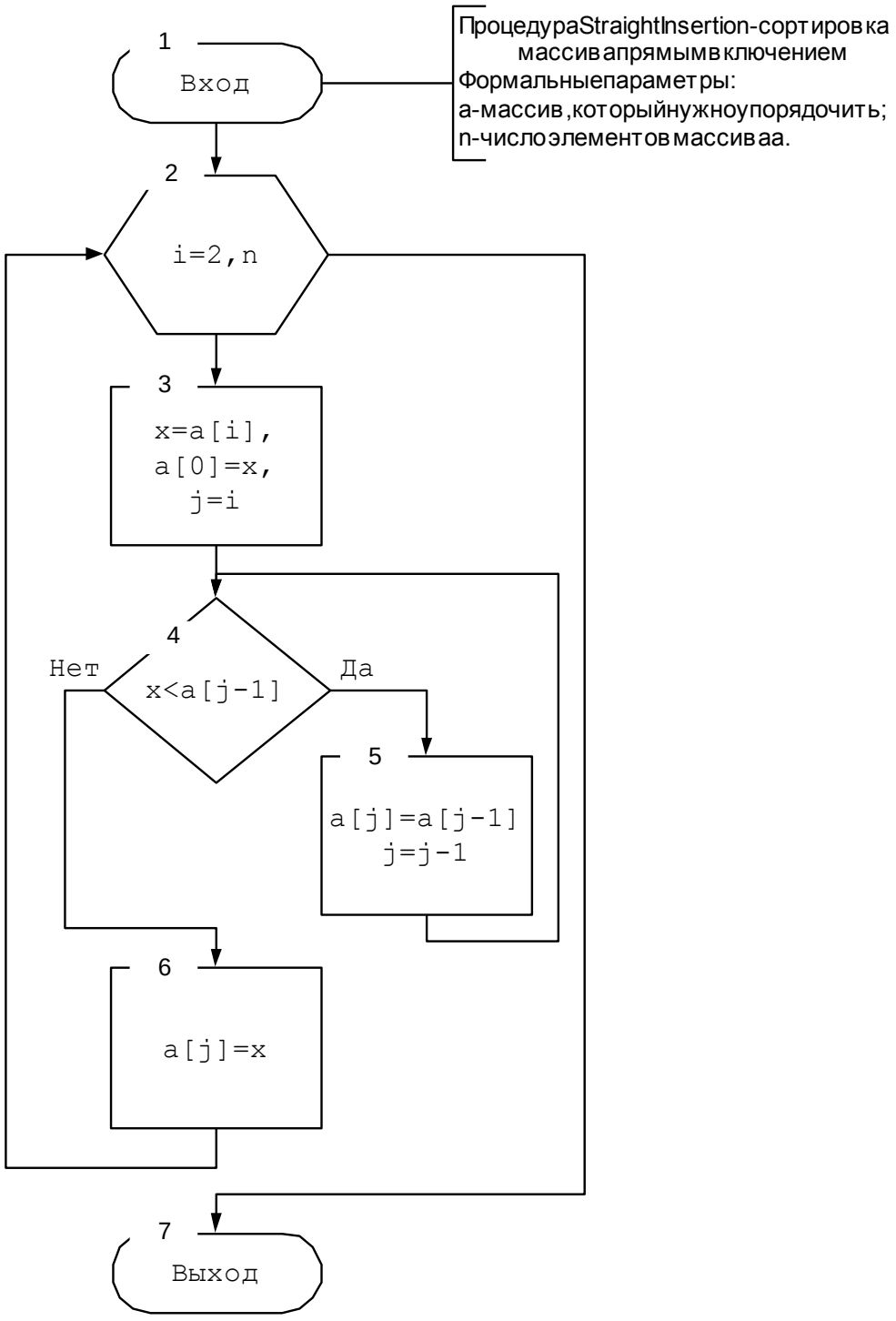


Рис. 1. Схема алгоритма сортировки прямым включением.

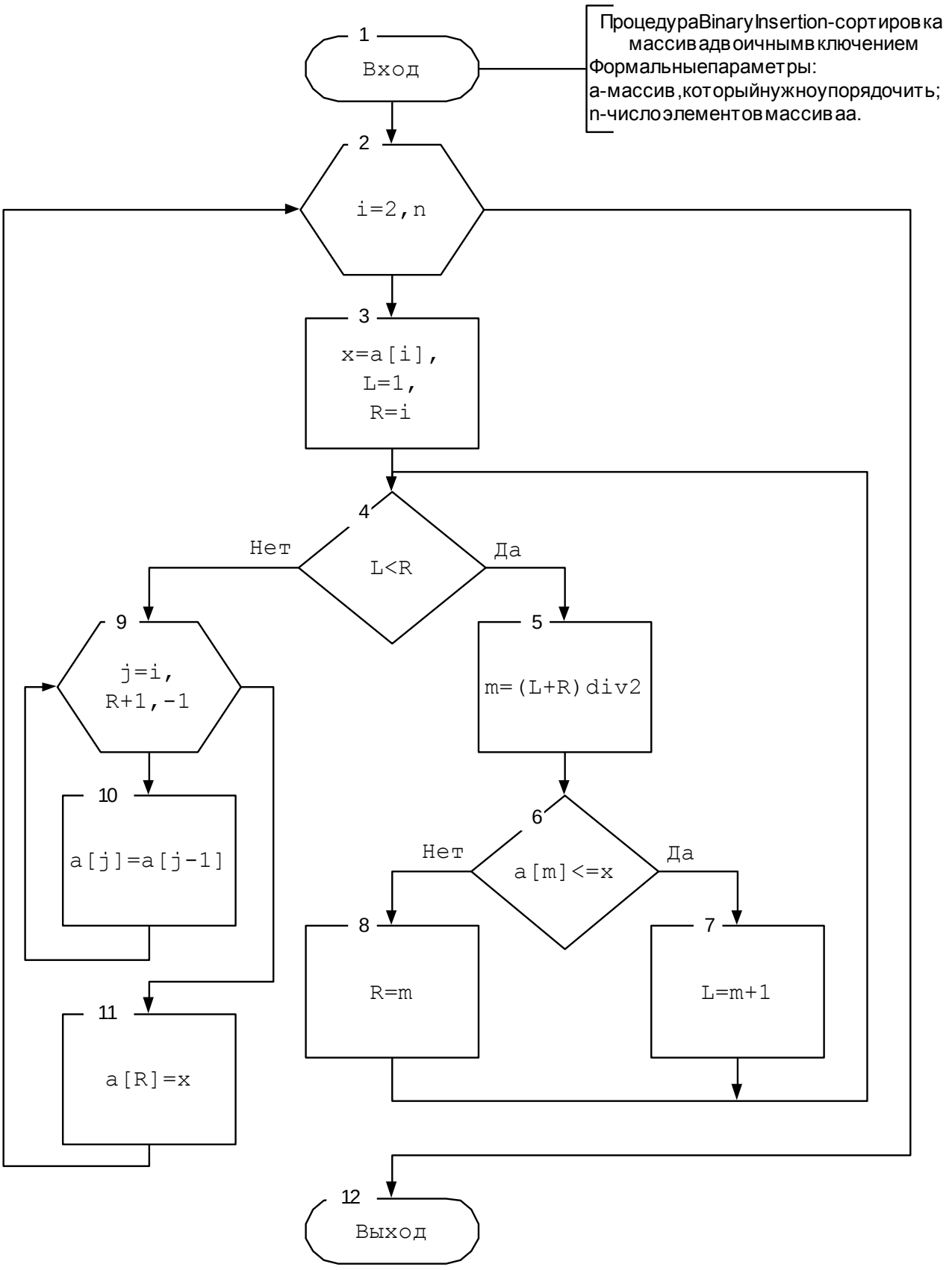


Рис. 2. Схема алгоритма сортировки бинарным включением.

4. Пример программы на языке C/C++

Данная программа демонстрирует метод сортировки прямым включением.

```
#include <iostream.h>
#include <stdlib.h>

main(void)
{
 const int n=20;
 float a[n+1];
 int i,j;
 float x;

 cout << "Сортировка последовательности, состоящей из 20";
 cout << "случайных чисел, методом прямого включения";
 cout << "Несортированная последовательность:\n";
 for (i=1; i<n+1; i++)
  {
   a[i]=random(100);
   cout << a[i] << " ";
  }

 for (i=2; i<n+1; i++)
  {
   x=a[i];
   a[0]=x;
   j=i;
   while (x<a[j-1])
    {
     a[j]=a[j-1];
     j--;
    }
   a[j]=x;
  }

  cout << "\nОтсортированная последовательность:\n";
  for (i=1; i<n+1; i++)
   {
    cout << a[i] << " ";
   }

 return 0;
}
```

5. Варианты заданий

1. Отсортировать массив, состоящий из 2^5 элементов.
2. Отсортировать натуральные числа (1…9), заданные в произвольном порядке.
3. Отсортировать массив, состоящий из положительных чисел.
4. Отсортировать массив, состоящий из отрицательных чисел.

5. Отсортировать массив, состоящий из положительных двузначных чисел.
6. Произвести сортировку массива, состоящего из 0 и1, записанных в произвольном порядке.
7. Отсортировать клавиши на клавиатуре в порядке убывания их ASCII–кодов.
8. Отсортировать клавиши на клавиатуре в порядке возрастания их ASCII–кодов.
9. Отсортировать по возрастанию массив, состоящий букв вашего имени и фамилии, считая что, а=1, б=2, в=3, …я=33.
10. Отсортировать буквы вашего отчества по схеме, указанной в п. 9.
11. Отсортировать массив, состоящий из (2n+1) элементов. Сортировку до (n+1)-го элемента осуществлять по возрастанию, а после (n+1)-го — по убыванию, т.е. «горкой».

$$\underbrace{1\ \ 2\ \ 3\ \ 4}_{n}\ \ \underset{n+1}{5}\ \ \underbrace{4\ \ 3\ \ 2\ \ 1}_{n}$$

12. Отсортировать массив, состоящий из (2n+1) элементов «оврагом».

$$\underbrace{5\ \ 4\ \ 3\ \ 2}_{n}\ \ \underset{n+1}{1}\ \ \underbrace{2\ \ 3\ \ 4\ \ 5}_{n}$$

13. Отсортировать последовательности, состоящие из натуральных чисел 1…9 и 9…1 и сравнить характеристики сортировки в первом и во втором случаях.
14. Провести сортировку массивов, состоящих из 2^5 и 2^6 чисел и сравнить характеристики сортировки.
15. Отсортировать массив, состоящий из отрицательных двузначных чисел в порядке убывания.
16. Реализовать метод сортировки прямым включением, используя ввод данных из файла.
17. Реализовать метод сортировки бинарным включением, используя ввод данных из файла.
18. Реализовать метод сортировки прямым включением, используя вывод результатов в файл.
19. Реализовать метод сортировки бинарным включением, используя вывод результатов в файл.
20. Даны два массива. Построить третий массив, который включает элементы двух предыдущих. Отсортировать его по убыванию, используя метод сортировки бинарным включением. Данные берутся из файла и записываются в файл.

6. Указания по составлению отчета

Отчет должен содержать постановку и описание задачи, алгоритм решения задачи, текст составленной программы на языке программирования C/C++, краткие выводы.

Лабораторная работа № 6 Распределяющая сортировка массивов

1. Цель и задачи работы

Изучение метода распределяющей сортировки массивов. Написание программы, демонстрирующей изученные принципы.

2. Теоретические сведения

Сортировка — это процесс, позволяющий упорядочить множество подобных данных в возрастающем или убывающем порядке. Сортировка представляет собой один из наиболее приемлемых с интеллектуальной точки зрения алгоритмов, поскольку процесс хорошо определен. Алгоритмы сортировки хорошо исследованы и изучены. К сожалению, по этой причине сортировку иногда принимают как некую данность.

В общем, сортировку следует понимать как процесс перегруппировки заданного множества объектов в некотором определенном порядке. Цель сортировки — облегчить последующий поиск элементов в таком отсортированном множестве. Это почти универсальная, фундаментальная деятельность. Отсортированные объекты можно встретить на каждом шагу: в телефонных книгах, в списках подоходных налогов, в оглавлениях книг, в библиотеках, в словарях, на складах – почти везде, где надо искать хранимые объекты. Даже малышей учат держать вещи «в порядке», и они уже сталкиваются с некоторыми видами сортировок задолго до того, как познакомятся с азами арифметики. Между сортировкой и арифметикой нет никакой видимой связи, да, по-видимому, и невидимой нет. Сортировка — некий «первичный» процесс человеческой деятельности.

Таким образом, разговор о сортировке вполне уместен и важен, если речь идет об обработке данных. Что может легче сортироваться, чем данные? Тем не менее, первоначальный интерес к сортировке основывается на том, что при построении алгоритмов часто сталкиваются со многими фундаментальными приемами. Почти не существует методов, с которыми не приходится встречаться при обсуждении этой задачи. В частности, сортировка — это идеальный объект для демонстрации огромного разнообразия алгоритмов, все они изобретены для одной и той же задачи, многие в некотором смысле оптимальны, большинство имеют достоинства. Поэтому это еще и идеальный объект, демонстрирующий необходимость анализа производительности алгоритмов. К тому же на примерах сортировок можно показать, как путем усложнения алгоритма,

хотя под рукой и есть уже очевидные методы, можно добиться значительного выигрыша в эффективности.

Вот некоторые из наиболее важных применений сортировки:

1. Решение задачи «группировки», когда нужно собрать вместе все элементы с одинаковым значением некоторого признака. Допустим, имеется 10000 элементов, расположенных в случайном порядке, причем значения многих из них равны; и предположим, нам нужно переупорядочить файл так, чтобы элементы с равными значениями занимали соседние позиции в файле. Это, по существу, задача «сортировки» в широком смысле слова, и она легко может быть решена путем сортировки файла в. узком смысле слова, а именно расположением элементов в неубывающем порядке $v_1 \le v_2 \le \ldots \le v_{10000}$.

2. Если два или более файла отсортировать в одном и том же порядке, то можно отыскать в них все общие элементы за один последовательный просмотр всех файлов, без возвратов. Оказывается, что, как правило, гораздо экономнее просматривать список последовательно, а не перескакивая с места на место случайным образом, если только список не настолько мал, что он целиком помещается в оперативной памяти. Сортировка позволяет использовать последовательный доступ к большим файлам в качестве приемлемой замены прямой адресации.

Теперь четко определим задачу и введем соответствующую терминологию. Пусть надо упорядочить N элементов

$$R_1, R_2, \ldots, R_N.$$

Назовем их *записями,* а всю совокупность N записей назовем *файлом.* Каждая запись R_j имеет *ключ* K_j, который и управляет процессом сортировки. Помимо ключа, запись может содержать дополнительную «сопутствующую информацию», которая не влияет на сортировку, но всегда остается в этой записи.

Отношение порядка «<» на множестве ключей вводится таким образом, чтобы для любых трех значений ключей a, b, c выполнялись следующие условия:

1) справедливо одно и только одно из соотношений: $a<b$, $a=b$, $b<a$ (закон трихотомии);

2) если $a<b$ и $b<c$, то $a<c$ (закон транзитивности).

Эти два свойства определяют математическое понятие *линейного упорядочения,* называемого еще *совершенным упорядочением.* Любое множество с отношением «<», удовлетворяющим вышеперечисленным свойствам, поддается сортировке большинством методов.

Задача сортировки — найти такую перестановку записей $p(1), p(2),\ldots,p(N)$, после которой ключи расположились бы в неубывающем порядке:

$$K_{p(1)}, K_{p(2)}, \ldots, K_{p(N)}.$$

Сортировка называется *устойчивой,* если она удовлетворяет дополнительному условию, что записи с одинаковыми ключами остаются в прежнем порядке, т. е.

$$p(i)<p(j),\ \text{если}\ K_{p(i)}=K_{p(j)}\ \text{и}\ i<j.$$

Обычно сортировку подразделяют на два класса: *внутреннюю,* когда все записи хранятся в быстрой оперативной памяти, и *внешнюю,* когда все они там не помещаются. При внутренней сортировке имеются более гибкие возможности для построения структур данных и доступа к ним, внешняя же показывает, как поступать в условиях сильно ограниченного доступа.

Достаточно хороший общий алгоритм затрачивает на сортировку N записей время порядка $N\log(N)$; при этом требуется около $\log(N)$ «проходов» по данным. Это минимальное время. Так, если удвоить число записей, то и время при прочих равных условиях возрастет немногим более чем вдвое.

Сортировка массивов

Основное условие: выбранный метод сортировки массивов должен экономно использовать доступную память. Это предполагает, что перестановки, приводящие элементы в порядок, должны выполняться *на том же месте*, то есть методы, в которых элементы из массива a передаются в результирующий массив b, представляют существенно меньший интерес.

Хорошей мерой эффективности может быть C — число необходимых сравнений ключей и M — число пересылок (перестановок) элементов. Эти числа суть функции от числа сортируемых элементов. Как говорилось выше, если сортируется N элементов, то хорошим считается такой алгоритм сортировки, который требует порядка $N\log(N)$ сравнений. Для начала стоит рассмотреть самые простые алгоритмы, которые называют *прямыми*, требующими порядка N^2 сравнений. Прямые методы особенно удобны для объяснения характерных черт основных принципов большинства сортировок, программы этих методов легко понимать, и они коротки. Усложненные же методы требуют небольшого числа операций, но эти операции обычно сами более сложны, и поэтому для достаточно малых N прямые методы оказываются быстрее, хотя при больших N их использовать, конечно, не следует.

Прямые методы сортировки можно разбить в соответствии с определяющими их принципами на следующие основные категории:

1. Сортировки с помощью включения;
2. Сортировки с помощью выделения;
3. Сортировка с помощью обменов;
4. Сортировка с помощью подсчета.

Сортировка методом подсчета

Данный метод сортировки основан на том, что j-й ключ в окончательно упорядоченной последовательности превышает ровно (j-1) из остальных ключей. Иначе говоря, если известно, что некоторый ключ превышает ровно N других, то после сортировки соответствующая запись должна занять (N+1)-е место.

Таким образом, идея данного метода сортировки состоит в том, чтобы сравнить попарно все ключи и подсчитать, сколько из них меньше каждого отдельного ключа.

Очевидный способ выполнить сравнения – сравнить K_j с K_i при $1 \le j \le N$, но легко видеть, что более половины этих действий излишни, поскольку не нужно сравнивать ключ сам с собой и после сравнения K_a с K_b уже не надо сравнивать K_b с K_a.

В результате приходим к различным алгоритмам сортировки методом подсчета.

Распределяющая сортировка

Распределяющая сортировка – это разновидность сортировки посредством подсчета, которая весьма важна с точки зрения эффективности; она применима в основном в том случае, когда имеется много равных ключей, и все они попадают в диапазон $U \le K_j \le V$, где значение (V-U) невелико. Эти предварительные сведения представляются весьма ограничительными, но на самом деле существует немало приложений этой идеи. Например, если применить этот метод лишь к старшим цифрам ключей, а не ко всем ключам целиком, то файл окажется частично отсортированным, и будет сравнительно просто довести дело до конца.

Чтобы понять принцип, предположим, что все ключи лежат между 1 и 100. При первом просмотре файла можно подсчитать, сколько имеется ключей, равных 1,2,…, 100, а при втором просмотре можно уже располагать записи в соответствующих местах области вывода. В приведенном ниже алгоритме все это описано более подробно.

Алгоритм распределяющей сортировки

Этот алгоритм в предположении, что все ключи – целые числа в диапазоне $U \le K_j \le V$ при $1 \le j \le N$, сортирует записи $R_1,\ldots,R_N$, используя вспомогательную таблицу **COUNT**[U],…, **COUNT**[V]. В конце работы алгоритма все записи в требуемом порядке переносятся в область вывода $S_1,\ldots,S_N$.

1. [Сбросить счетчики.] Установить **COUNT**[U],…, **COUNT**[V] равными нулю.

2. [Цикл по j.] Выполнить шаг 3. при $1 \le j \le N$, затем перейти к шагу 4.
3. [Увеличить **COUNT**[K_j].] Увеличить значение **COUNT**[K_j] на единицу.
4. [Суммирование.] (К этому моменту значение **COUNT**[i] есть число ключей, равных i.) Установить **COUNT**[i]= **COUNT**[i]+ **COUNT**[i-1] при i=U+1,U+2,…,V.
5. [Цикл по j.] (К этому моменту значение **COUNT**[i] есть число ключей, меньших или равных i, в частности **COUNT**[V]=N.) Выполнить шаг 6. при j=N,N-1,…,1 и завершить работу алгоритма.
6. [Вывод R_j .] Установить i= **COUNT**[K_j], $S_i=R_j$ и **COUNT**[K_j]=i-1.

Для того чтобы лучше понять данный алгоритм, промоделируем его работу, выделяя при этом основные шаги.

Пусть дан массив a[5]={ 3 , 5 , 2 , 1 , 4 }.

Шаг №1. Установим счетчики **COUNT**[i]=0,при i=3, 5, 2, 1, 4. Получим следующую таблицу значений счетчиков:

I	3	5	2	1	4
COUNT[i]	0	0	0	0	0

Шаг №2 и шаг №3. Подсчитаем сколько имеется ключей, равных 3, 5, 2, 1, 4. Получим следующую таблицу значений счетчиков:

I	3	5	2	1	4
COUNT[i]	1	1	1	1	1

Шаг №4. Суммирование значений счетчиков, индексы которых меньше или равны заданному, т.е. **COUNT**[2]=**COUNT**[2]+**COUNT**[1], **COUNT**[3]= **COUNT**[3]+**COUNT**[2], где **COUNT**[2]=**COUNT**[2]+**COUNT**[1], и т.д. В результате таблица значений счетчиков будет иметь следующий вид:

I	3	5	2	1	4
COUNT[i]	3	5	2	1	4

Шаги №5 и №6. В соответствии со значениями счетчиков расположим элементы массива в требуемом порядке. В результате получим отсортированный массив:

a[5]={ 1, 2, 3, 4, 5 }.

Анализ распределяющей сортировки

Распределяющая сортировка применима в основном в тех случаях, когда выполняются следующие условия:

1) все ключи – целые числа;
2) имеется много равных ключей;
3) все ключи попадают в диапазон $U \le K_j \le V$, где значение (V-U) невелико.

Преимущества метода распределяющей сортировки:

1) при выполнении сформулированных выше условий это очень быстрая процедура сортировки;
2) метод устойчив (не изменяется порядок записей с равными ключами).

Недостатком метода распределяющей сортировки является неэкономное использование памяти: требуется память для счетчиков и 2*N записей. Существуют модификации этого метода, экономящие N из этих записей, но они теряют устойчивость.

3. Схема алгоритма распределяющей сортировки

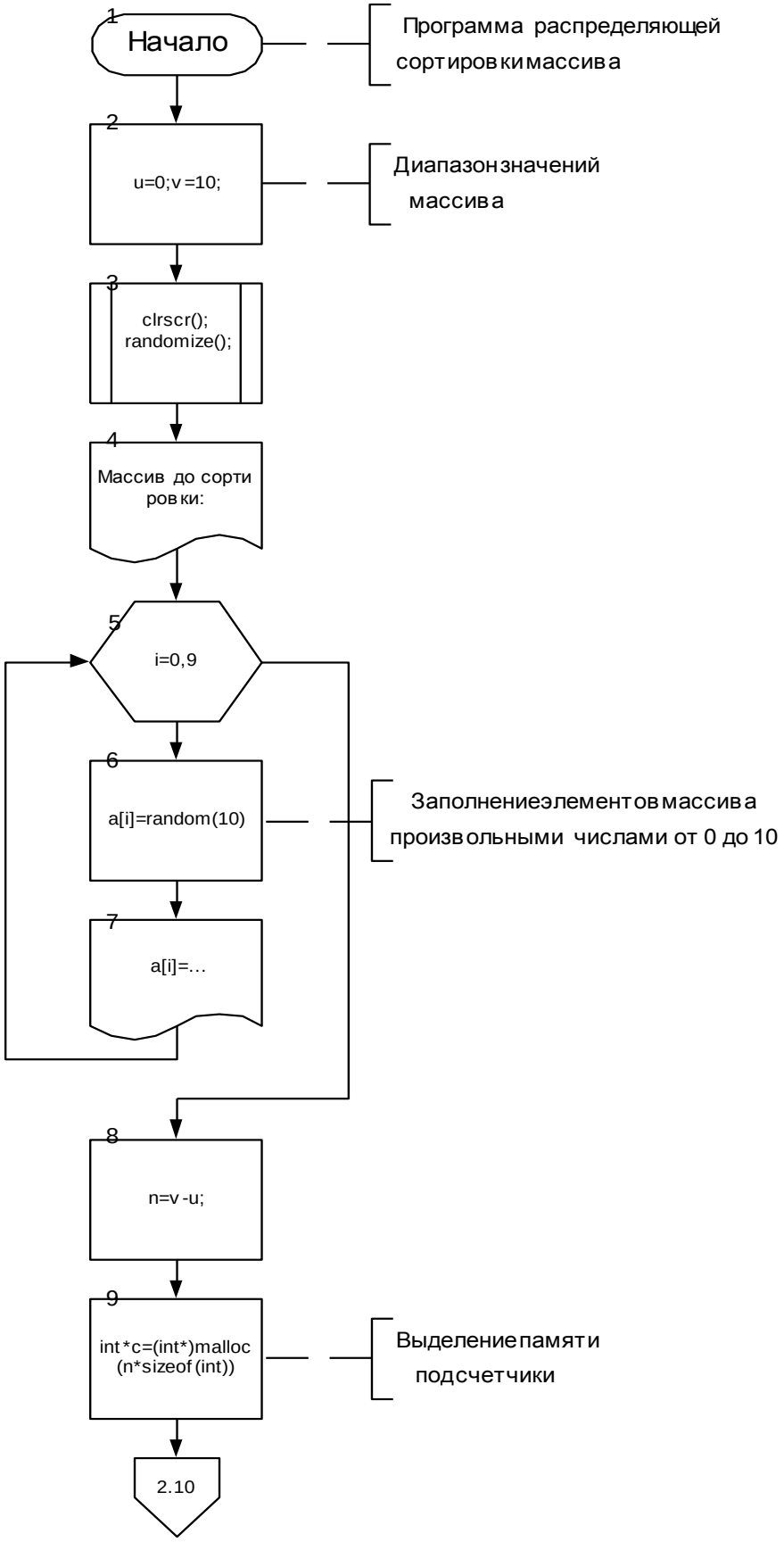
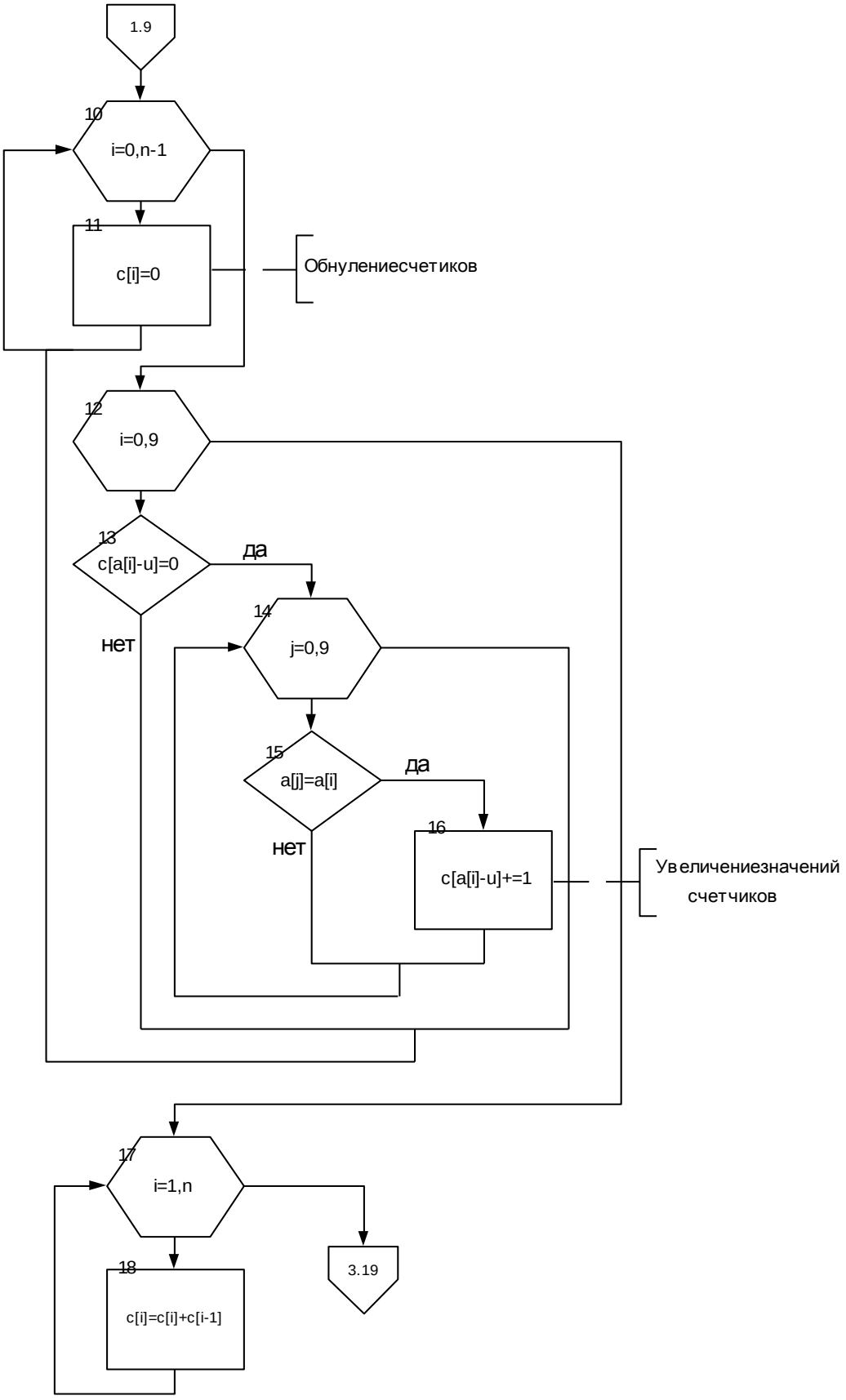
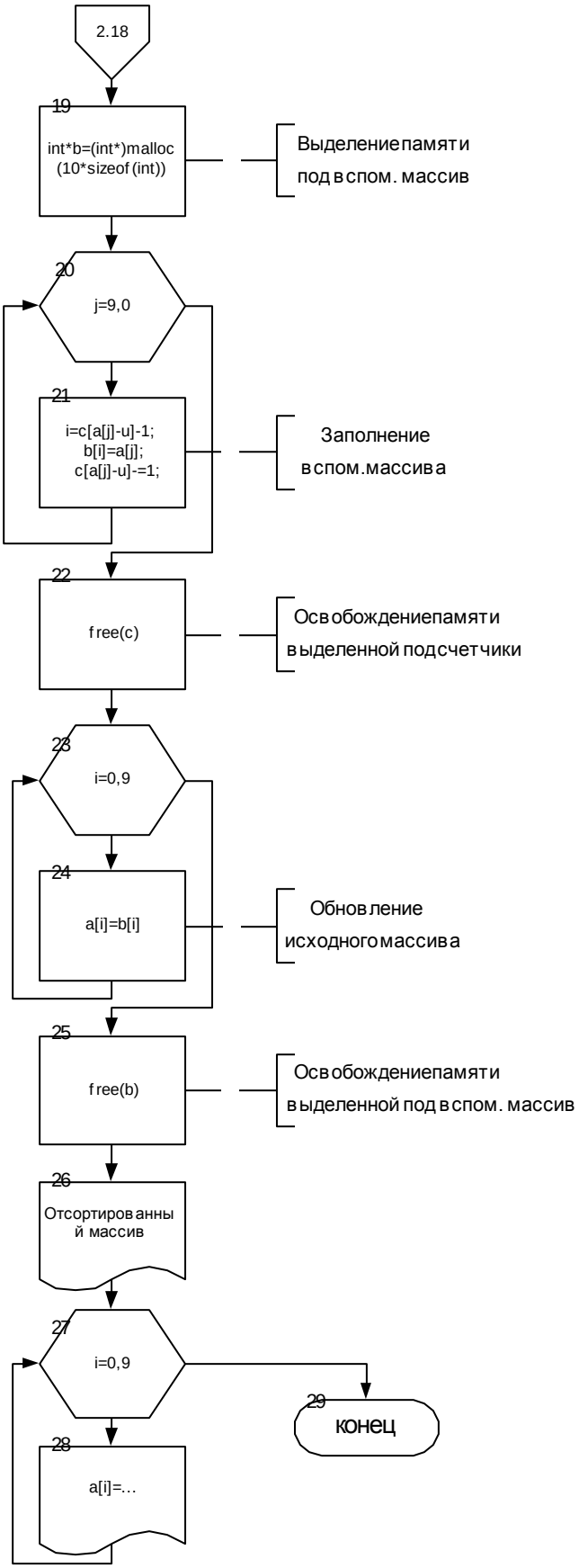


Рис. 1. Схема алгоритма распределяющей сортировки.


1.9
10
i=0,n-1
11
c[i]=0
Обнулениесчетиков
12
i=0,9
13
c[a[i]-u]=0
да
нет
14
j=0,9
15
a[j]=a[i]
да
нет
16
c[a[i]-u]+=1
Увеличениезначений
счетчиков
17
i=1,n
18
c[i]=c[i]+c[i-1]
3.19


2.18
19
int*b=(int*)malloc
(10*sizeof(int))
Выделение памяти
под вспом. массив
20
j=9,0
21
i=c[a[j]-u]-1;
b[i]=a[j];
c[a[j]-u]-=1;
Заполнение
вспом.массива
22
free(c)
Освобождение памяти
выделенной под счетчики
23
i=0,9
24
a[i]=b[i]
Обновление
исходного массива
25
free(b)
Освобождение памяти
выделенной под вспом. массив
26
Отсортированны
й массив
27
i=0,9
29
конец
28
a[i]=...

3. Пример программы на языке C/C++

```
//Распределяющая сортировка массива
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<alloc.h>
void main()
{
 int a[10];
 int i,j,temp,u,v,n;
 u=0;v=10;
 clrscr();
 randomize();
 printf("\nМассив до сортировки:\n");
 for(i=0;i<10;i++)       //Формирование
 {                       //исходного
  a[i]=random(10);       //массива
  printf("%d ",a[i]);
 }
 getch();
 n=v-u;//Количество счетчиков
 int *c=(int*)malloc(n*sizeof(int));//Память под счетчики
 for(i=0;i<n;i++) c[i]=0;//Обнуление счетчиков
 for(i=0;i<10;i++)
 {
  if (c[a[i]-u]==0)//Проверка: был ли подсчет для ключа
  {
  for(j=0;j<10;j++)
  if (a[j]==a[i]) c[a[i]-u]+=1;//Подсчет кол-ва элементов,
                               //равных ключу
  }
 }
 for(i=1;i<n+1;i++)
   c[i]=c[i]+c[i-1];//Суммирование
 int *b=(int*)malloc(10*sizeof(int));//Выделение памяти под
                                     //вспом. массив
 for(j=9;j>-1;j--)
  {
   i=c[a[j]-u]-1;
   b[i]=a[j];
   c[a[j]-u]-=1;
  }
 free(c);//Освобождение памяти, выделенной для счетчиков
 for(i=0;i<10;i++) a[i]=b[i];//Обновление исходного массива
 free(b);//Освобождение памяти, выделенной для вспом. массива
 printf("\nОтсортированный массив:\n");
 for(i=0;i<10;i++)
 printf("%d ",a[i]);
 getch();//Задержка до нажатия клавиши
}
```

4. Контрольные вопросы

1. Что представляет собой процесс сортировки?
2. Какое основное условие накладывается на сортировку массивов?
3. На чем основана сортировка методом подсчета?
4. В чем суть распределяющей сортировки?
5. Является ли алгоритм распределяющей сортировки алгоритмом «устойчивой» сортировки? Почему?
6. Будет ли алгоритм распределяющей сортировки работать правильно, если в шаге 5 значение j будет изменяться от 1 до N, а не от N до 1?

5. Варианты заданий

1. Отсортировать натуральные числа (1…9), заданные в произвольном порядке.
2. Отсортировать массив, состоящий из положительных двузначных чисел по возрастанию.
3. Отсортировать массив, состоящий из отрицательных двузначных чисел в порядке убывания.
4. Дано два массива a и b, каждый состоит из n элементов (n = 1,2,3 ...). Построить третий массив c при условии $c_i = a_i + b_i$. Отсортировать его по возрастанию.
5. Дано два массива a и b, каждый состоит из n элементов (n = 1,2,3 ...). Построить третий массив c , при условии $c_i = a_i * b_i$. Отсортировать его по убыванию.
6. Отсортировать по возрастанию массив, состоящий букв вашего имени и фамилии, считая, что а = 1, б = 2, в = 3, …я = 33.
7. Отсортировать буквы вашего отчества, считая, что а = 1, б = 2, в = 3, …я = 33.
8. Дан массив, состоящий более чем из десяти элементов. Упорядочить первую его половину по возрастанию, а вторую – по убыванию.
9. Реализовать распределяющую сортировку массива, используя ввод данных из файла.
10. Реализовать распределяющую сортировку массива, используя вывод данных в файл.
11. Упорядочить по убыванию массив, содержащий суммы оценок студентов группы. На экран вывести список студентов и их средний балл, а в файл – список студентов, их оценки, сумму и средний балл.
12. Задается массив, упорядоченный по возрастанию. В него добавляются три произвольных элемента. Упорядочить массив по возрастанию.
13. Задается массив, упорядоченный по возрастанию. В него добавляются три произвольных элемента. Упорядочить массив по убыванию.

14. Упорядочить по возрастанию массив, содержащий суммы оценок студентов группы. На экран вывести список студентов и их средний балл, а в файл – список студентов, их оценки, сумму и средний балл.
15. Задается массив, упорядоченный по возрастанию. В него добавляются три произвольных элемента. Упорядочить массив по убыванию

6. Указания по составлению отчета

Отчет должен содержать постановку и описание задачи, алгоритм решения задачи, текст составленной программы на языке программирования С++, краткие выводы.

Библиографический список

1. Подбельский В.В. Программирование на языке Си. М.: Финансы и статистика, 2000.

Лабораторная работа №7 Сортировка слиянием

1. Цель и задачи работы

Изучить метод сортировки слиянием файлов и массивов в памяти. Написать программы, иллюстрирующие изученные принципы.

2. Теоретические сведения

Сортировка слиянием

Слияние означает объединение одного или более упорядоченных файлов в один упорядоченный файл. Можно, например, слить два подфайла – 503 703 765 и 087 512 677, получив 087 503 512 677 703 765. Простой способ сделать это – сравнить два наименьших элемента, вывести наименьший из них и повторить процедуру. Начав с

$$\begin{cases} 503\ 703\ 765 \\ 087\ 512\ 677 \end{cases}$$

получим

$$087 \begin{cases} 503\ 703\ 765 \\ 512\ 677 \end{cases}$$

затем

$$087\ 503 \begin{cases} 703\ 765 \\ 512\ 677 \end{cases}$$

и т.д. Необходимо позаботиться о действиях на случай, когда исчерпается один из файлов.

Эта простая процедура, по существу, наилучший из возможных способов слияния на традиционной ЭВМ, если $m \approx n$.

Общий объем работы, выполняемой алгоритмом, по существу, пропорционален m+n, поэтому ясно, что слияние - более простая задача, чем сортировка. Однако задачу сортировки можно свести к слияниям, сливая все более длинные подфайлы до тех пор, пока не будет отсортирован весь файл. Такой подход можно рассматривать как развитие идеи сортировки вставками: вставка нового элемента в упорядоченный файл – частный случай слияния, при n=1! Если нужно ускорить процесс вставок, то можно рассмотреть вставку нескольких элементов за раз, группируя их, а это естественным образом приводит к общей идеи сортировки слиянием. С исторической точки зрения метод слияний – один из самых первых методов, предназначенных для сортировки на ЭВМ; он был предложен Джоном фон Нейманом еще в 1945 году.

Таблица 1.

503|	087|	512|	061|	908|	170|	897|	275|	|653	|426	|154	|509	|612	|677	|765	|703
503	703|	512	677|	509	908|	426	897|	|653	275	|170	154	|612	061	|765	087
087	503	703	765|	154	170	509	908|	|897	653	426	275	|677	612	512	061
061	087	503	512	612	677	703	765|	|908	897	653	509	426	275	170	154
061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Таблица 1 иллюстрирует сортировку слиянием, когда «свечка сжигается с обоих концов». Мы анализируем исходный файл слева и справа, двигаясь к середине. Рассмотрим переход от второй строки к третьей. Справа видим возрастающий отрезок 503 703 765, а справа, если читать справа налево, имеем отрезок 087 512 677. Слияние этих двух последовательностей дает подфайл 087 503 512 677 703 765, который помещается слева в третьей строке. Затем ключи 061 612 908 в строке 2 сливаются с 170 509 897, и результат (061 170 509 612 897 908) записывается *справа* в строке 3. Наконец, 154 275 426 653 сливается с 653 (перекрытие обнаруживается прежде, чем оно может привести к вредным последствиям), и результат записывается слева. Точно также строка 2 получилась из исходного файла в строке 1.

Вертикальными линиями в таблице 1 отмечены границы между отрезками. Это так называемые «ступеньки вниз», где меньший элемент следует за большим. Такой метод называют «естественным» слиянием, потому что он использует отрезки, которые естественно образуются в исходном файле. Если файл случаен, то в нем около 1/2N возрастающих отрезков. При каждом просмотре число отрезков сокращается вдвое. Таким образом, число просмотров составляет примерно около $\log_2 N$.

Простое двухпутевое слияние

В данном алгоритме границы между отрезками полностью определены «ступеньками вниз». Такой подход обладает тем возможным преимуществом, что исходные файлы с преобладанием возрастающего или убывающего расположения элементов могут обрабатываться очень быстро, но при этом замедляется основной цикл вычислений. Вместо проверок ступенек вниз можно принудительно установить длину отрезков, считая, что все отрезки исходного файла имеют длину 1. После первого просмотра все отрезки, кроме, возможно, последнего, имеют длину 2 и т.д.

Таблица 2.

503|	087|	512|	061|	908|	170|	897|	275|	|653	|426	|154	|509	|612	|677	|765	|703
503	703|	512	677|	509	908|	426	897|	|653	275	|170	154	|612	061	|765	087
087	503	703	765|	154	170	509	908|	|897	653	426	275	|677	612	512	061
061	087	503	512	612	677	703	765|	|908	897	653	509	426	275	170	154
061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Пример работы алгоритма в таблице 2. Данный метод справедлив и тогда, когда N не является степенью числа 2; сливаемые отрезки не все имеют длину 2^k.

Рассмотрим оба алгоритма с точки зрения структур данных. Почему нам необходима память под 2n, а не под N записей? Причина проста: мы работаем с двумя списками. Но в любой момент времени половина памяти не используется, и очевидно, что в данном случае, следовало бы воспользоваться связным распределением памяти.

Порядок выполнения работы

1) Задать файлы, содержащие случайные числовые значения.
2) Произвести сортировку в соответствии с индивидуальным заданием.
3) Результат работы программы вывести в файл и на дисплей.

3. Варианты заданий

1. Написать программу сортировки слиянием по возрастанию двух файлов разного размера, используя метод естественного двух путевого слияния.
2. Написать программу сортировки слиянием по убыванию двух файлов разного размера, используя метод естественного двух путевого слияния.
3. Написать программу сортировки слиянием массива в памяти по возрастанию, используя метод естественного двух путевого слияния.
4. Написать программу сортировки слиянием массива в памяти по убыванию, используя метод естественного двух путевого слияния.
5. Написать программу сортировки слиянием двух файлов по возрастанию, обрабатывать заданное число записей в каждом из файлов. Использовать метод естественного двух путевого слияния.
6. Написать программу сортировки слиянием двух файлов по убыванию, обрабатывать заданное число записей в каждом из файлов. Использовать метод естественного двух путевого слияния.
7. Написать программу сортировки слиянием по возрастанию массива в памяти, обрабатывая указанное число записей от начала массива. Использовать метод естественного двух путевого слияния.
8. Написать программу сортировки слиянием по убыванию массива в памяти, обрабатывая указанное число записей от начала массива. Использовать метод естественного двух путевого слияния.
9. Написать программу сортировки слиянием по возрастанию массива в памяти, обрабатывая указанное число записей от концов массива. Использовать метод естественного двух путевого слияния.
10. Написать программу сортировки слиянием по убыванию массива в памяти, обрабатывая указанное число записей от концов массива. Использовать метод естественного двух путевого слияния.

11. Выполнить задание варианта 1), используя метод простого двух путевого слияния
12. Выполнить задание варианта 2), используя метод простого двух путевого слияния
13. Выполнить задание варианта 3), используя метод простого двух путевого слияния
14. Выполнить задание варианта 4), используя метод простого двух путевого слияния
15. Выполнить задание варианта 5), используя метод простого двух путевого слияния
16. Выполнить задание варианта 6), используя метод простого двух путевого слияния
17. Выполнить задание варианта 7), используя метод простого двух путевого слияния
18. Выполнить задание варианта 8), используя метод простого двух путевого слияния
19. Выполнить задание варианта 9), используя метод простого двух путевого слияния

4. Указания по составлению отчета

Отчет должен содержать постановку и описание задачи, алгоритм решения задачи, текст составленной программы на языке программирования C/C++, краткие выводы.

Лабораторная работа №8 Сортировка методом простого выбора

1. Цель и задачи работы

Исследовать метод сортировки массивов методом простого выбора, провести анализ метода сортировки массива, составить схему алгоритма для сортировки, написать программу на языке программирования C/C++.

2. Теоретические сведения

Описание метода

Одно важное семейство методов сортировки основано на идее многократного выбора. Простейшая сортировка посредством выбора сводится к следующему: алгоритм находит элемент с наименьшим значением и выполняет перестановку, меняя его местами с первым элементом массива. После этого из оставшихся n-1 элементов ищется наименьший, после чего осуществляется его перестановка со вторым элементов и так далее. Перестановки будут осуществляться до тех пор, пока не поменяются местами последние два элемента. Например, если бы строка «dcab» сортировалась методом выбора, то каждый из проходов давал бы следующий результат:

Исходный массив	d c a b
проход 1	a c d b
проход 2	a b d c
проход 3	a b c d

Анализ метода

К сожалению, сортировка методом отбора является n-квадратичным алгоритмом. Внешний цикл исполняется n-1 раз, а внутренний – n/2 раз. В результате сортировка методом отбора требует ½(n2-n) сравнений, что сильно замедляет работу метода при большом количестве элементов. Количества перестановок, требующиеся в наилучшем и наихудшем случаях для метода сортировки отбором, будут следующими:

- наилучший случай: 3(n-1),
- наихудший случай: n2/4+3(n-1).

В наилучшем случае, если список уже упорядочен, требуется сравнить только (n-1) элемент, и каждое сравнение требует трех промежуточных шагов. Наихудший случай аппроксимирует количество сравнений.

Средний случай для определения труден, и вывод его формулы выходит за рамки данной дисциплины. Однако, формулу для его вычисления мы приведем: n(log n + y), где y — константа Эйлера, примерно равная 0.577216.

Хотя количество сравнений для пузырьковой сортировки и сортировки методом отбора одинаковы, однако, для сортировки отбором показатель количества перестановок в среднем случае намного лучше. Тем не менее, существуют более совершенные методы сортировки.

Алгоритм хорошо работает на массивах малых размеров (до 100).

3. Схема алгоритма сортировки методом простого выбора

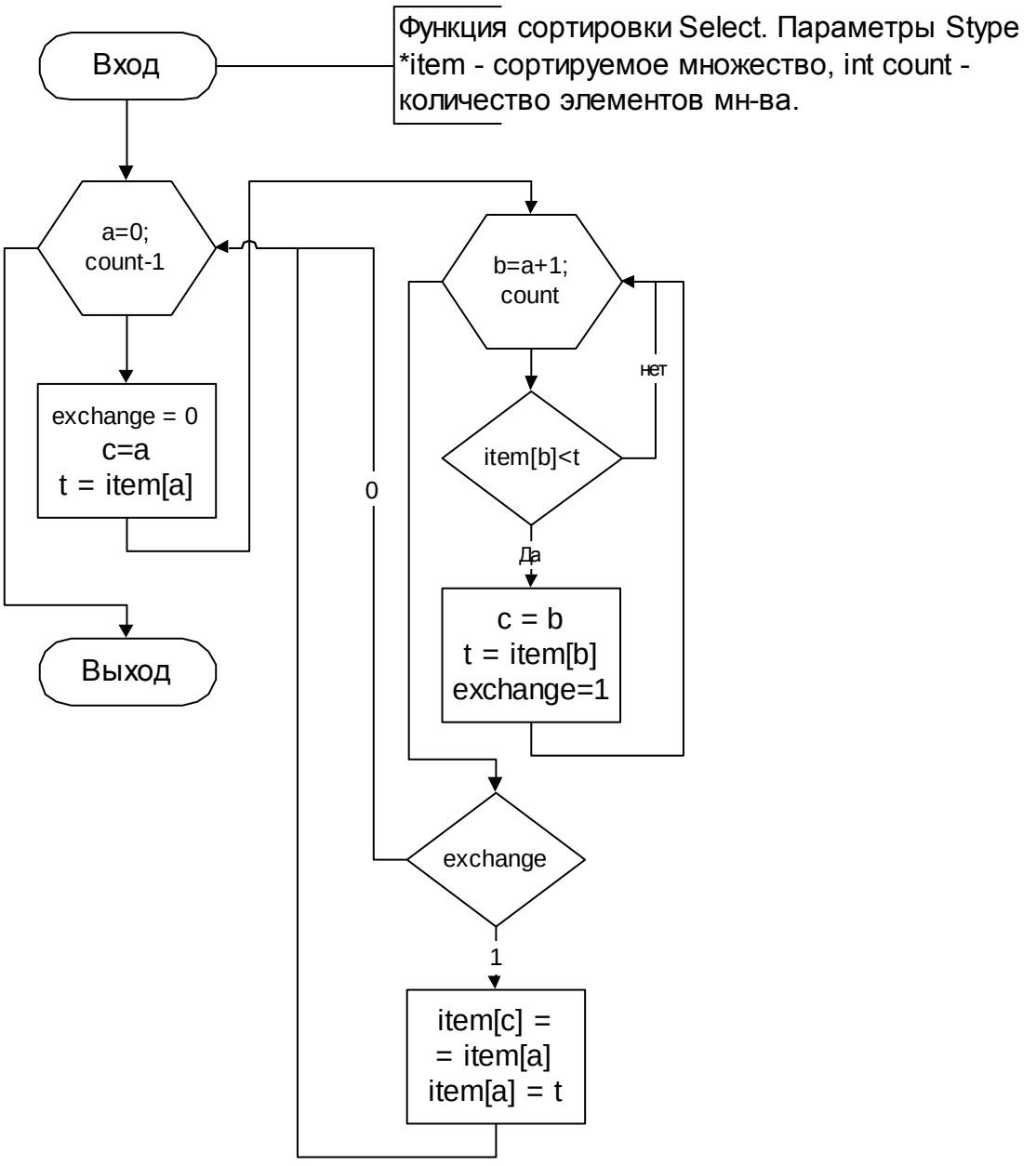


Рис. 1. Схема алгоритма сортировки методом простого выбора

4. Пример программы на языке C/C++

```
#include <iostream.h>
#include <string.h>
template <class Stype> void select (Stype *item, int count);
// Примеры использования метода сортировки
int main()
{ // сортировка массива символов
   char str[] = "dcab";
   select (str, (int) strlen(str)) ;
   cout << "Отсортированный массив: " << str << "\n";
// сортировка массива целых чисел
   int nums[] = {5, 1, 3, 9, 5, 1, 8};
   int i;
   select (nums, 7);
   cout << "Отсортированный массив: ";
   for(i=0; i<7; i++) cout << nums[i] << " ";
   return 0;
}
// Сам метод сортировки
template <class Stype> void select(Stype *item, int count)
{  int a, b, c;
   int exchange;
   Stype t;
   for (a=0; a<count-1; ++a)
   {   exchange = 0;
       c=a;
       t = item[a] ;
       for (b=a+1;b<count; ++b)
       {  if(item[b]<t)
          {    c = b;
               t = item[b] ;
               exchange = 1 ;
          }
       }
       if (exchange)
       {   item[c] = item[a];
           item[a] = t;
          }
    }
}
```

5. Варианты заданий

1. Отсортировать символьный массив, состоящий из 30 элементов.
1. Отсортировать числа (0…9), заданные в произвольном порядке.
2. Отсортировать массив, состоящий из положительных чисел, по возрастанию.
3. Отсортировать массив, состоящий из отрицательных чисел, по убыванию.
4. Отсортировать массив, состоящий из положительных двузначных чисел.

5. Произвести сортировку массива, состоящего из 0, 1 и 2, записанных в произвольном порядке (количество элементов n не менее 6).
6. Отсортировать текст, расположив слова в алфавитном порядке, и вывести результат в новый файл (считать, что a=1, b=2, c=3, …, z=26, а = 27, б = 28, в = 29, …я = 59).
7. Отсортировать клавиши на клавиатуре в порядке возрастания их ASCII –кодов.
8. Отсортировать по возрастанию массив, состоящий букв вашего имени, отчества и фамилии, считая, что а = 1, б = 2, в = 3, …я = 33.
9. Отсортировать буквы вашего отчества по убыванию по схеме, указанной в задании 9.
10. Отсортировать массив, состоящий из (2n + 1) элементов. Сортировку до (n + 1) –го элемента осуществлять по убыванию, а после (n + 1)-го – по возрастанию,.

$$\underbrace{4\ \ 3\ \ 2\ \ 1}_{n}\ \ 5\ \ \underbrace{1\ \ 2\ \ 3\ \ 4}_{n+1\quad n}$$

11. Отсортировать по возрастанию массив, состоящий букв вашего имени, отчества и фамилии, учитывая пробелы между ними по схеме из задания 9, и считая <пробел>=0.
12. Отсортировать массив, состоящий из отрицательных двузначных чисел в порядке возрастания.
13. Отсортировать массив, состоящий из студентов вашей группы (расположенных в массиве произвольно), по возрастанию, по схеме из задания 9.
14. Отсортировать массив из задания 13 по убыванию.
15. Отсортировать массив, состоящий из трехзначных чисел по возрастанию.
16. Отсортировать массив, состоящий из трехзначных чисел по убыванию.
17. Отсортировать массив, состоящий из положительных чисел по убыванию.
18. Отсортировать массив, состоящий из отрицательных чисел, по возрастанию.
19. Отсортировать клавиши с кириллицей на клавиатуре в порядке возрастания их ASCII – кодов.

6. Указания по составлению отчета

Отчет должен содержать постановку и описание задачи, алгоритм решения задачи, текст составленной программы на языке C/C++, краткие выводы.

Библиографический список

1. Шилдт Г. Теория и практика C++: пер. с англ. СПб.: БХВ – Петербург, 2001.

Лабораторная работа №9 Генерация произвольных распределений

1. Цель и задачи работы

Изучение генерации случайных последовательностей по произвольно заданному закону распределения. Написание программы, иллюстрирующей изученные принципы.

2. Теоретические сведения

Наряду с генерацией равномерного и нормального распределений возникает необходимость генерации случайных последовательностей с произвольным законом распределения (законом распределения случайной величины называется соотношение, устанавливающее связь между возможными значениями случайной величины и соответствующими им вероятностям). Ситуация относительно проста, если для заданной функции распределения можно найти аналитически обратную функцию. Как поступают в таких случаях, описано в предыдущей работе. А если нельзя этого сделать? Или очень сложно и практически не осуществимо?

Тогда можно использовать табличный способ задания функций и некоторые предварительные сведения, адекватные решаемой задаче. В таком случае легко построить функцию распределения, если задана функция плотности распределения. Пусть есть сетка, в узлах которой задана функция плотности распределения. И пусть эта сетка равномерна (что, вообще говоря, не обязательно, но такое предположение упрощает рассуждения, которые можно будет обобщить и на сетку с неравномерным шагом). Тогда достаточно знать x_0 — координату начала сетки, ее шаг, количество заданных узлов и значения в них функции плотности вероятности. Для построения функции распределения, которая также будет представлена таблично, используем смысл функции плотности распределения, которая показывает приращение функции распределения на каждом следующем ее шаге. Т.е. алгоритм построения функции распределения будет следующим. На первом шаге устанавливаем в узле X_0 значение, равное 0. А на каждом следующем шаге получаем новое значение путем суммирования предыдущего значения со значением функции распределения в соответствующем узле.

Затем, для простоты, предположим, что случайная величина между узлами распределена равномерно, т.о. получим функцию распределения в виде непрерывной кусочной функции, т.е. для каждого кусочка легко получить обратную функцию. Тогда для получения случайной величины с заданным законом распределения необходимо получить значение от

генератора равномерного распределения (от 0 до 1) и, используя график функции распределения, получить случайную величину.

Кроме того, необходимо учесть, что интеграл от плотности распределения должен быть равен 1, т.е. необходимо провести нормировку. Это делается следующим образом. Вычисляем сумму всех значений заданной плотности вероятности («пробегая» по всем узлам сетки). А далее делим каждое из значений этой функции (повторяя «пробег» по узлам) на полученную сумму. Таким образом, получим нормированную плотность вероятности.

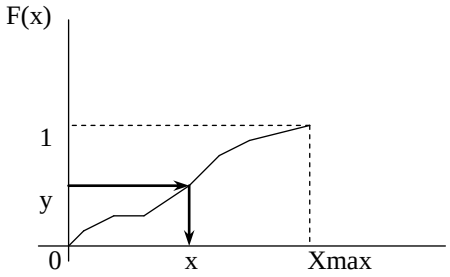


Рис. 1. Графическое изображение метода нахождения случайной величины с заданным законом распределения с помощью этого закона и генератора равномерного распределения (от 0 до 1).

3. Порядок выполнения работы

1) Задать массив, содержащий значения заданной функции плотности вероятности с приемлемым шагом дискретизации.
2) Пронормировать полученную функцию, заданную своими значениями в узлах сетки.
3) По полученной отнормированной функции плотности вероятности построить функцию распределения, которую также задать массивом значений в узлах той же сетки, что и функция плотности вероятности.
4) По полученной функции распределения найти последовательность из 50000 случайных величин и отобразить полученный результат на экране в виде гистограммы (отобразить надо плотность распределения полученной последовательности). Для этого можно использовать модуль gist.h

Существует также ещё один алгоритм, позволяющий сгенерировать дискретную случайную величину, закон распределения которой задан в виде таблицы:

X	x_1	x_2	…	x_n
P	p_1	p_2	…	p_n

Для этого необходимо:

1) Разбить интервал (0,1) на n частичных интервалов: Δ_1 - (0 , p_1); Δ_2 - (p_1 , p_1+p_2); ...; Δ_n - ($p_1+ p_2+...+p_{n-1}$, 1);

2) Получить от генератора равномерного распределения (от 0 до 1) случайное число r.
3) Если значение r попало в частичный интервал Δ_i, то генерируемая случайная величина приняла возможное значение x_i.

4. Пример программы на языке C/C++

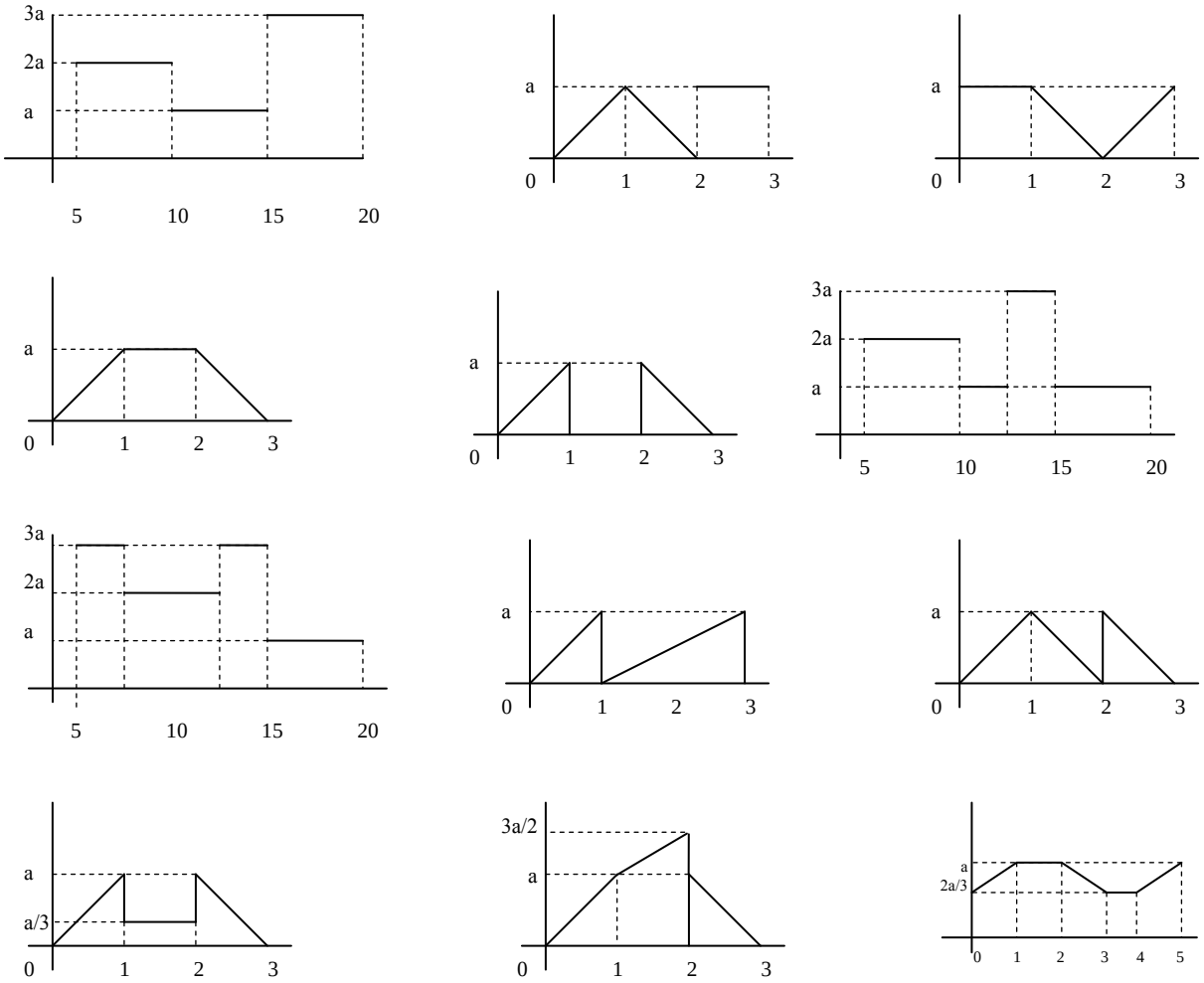
Программа демонстрирует вывод гистограммы на экран

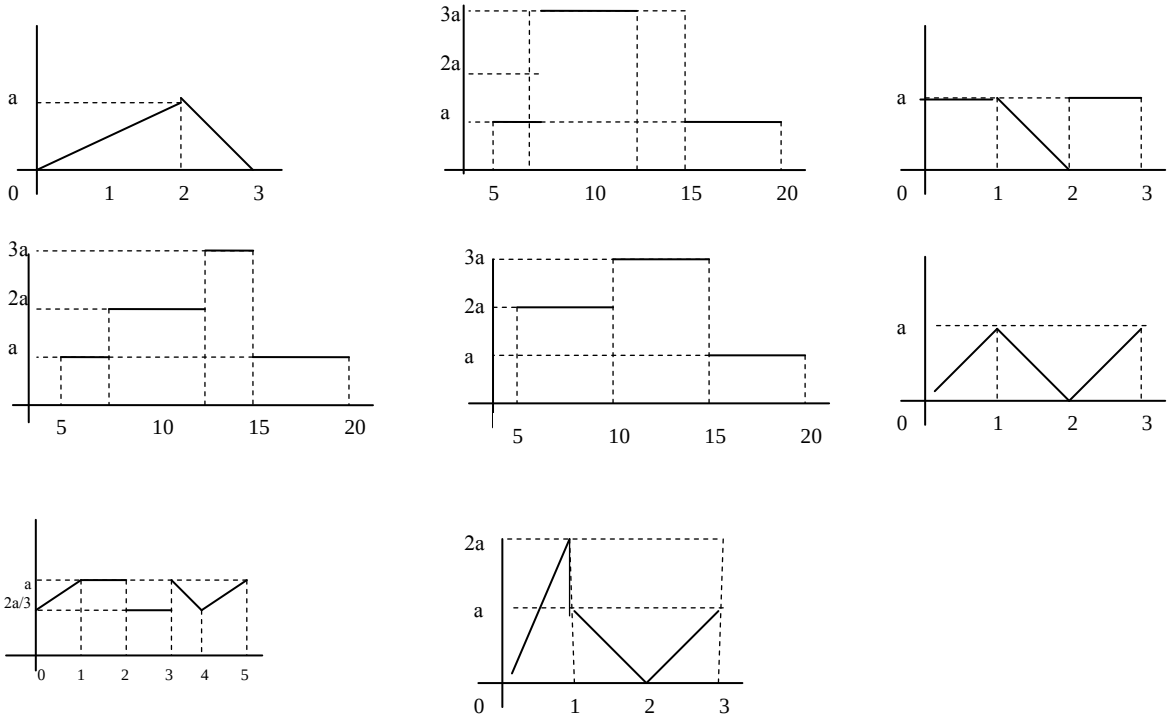
```
#include <graphics.h>
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
void DrawGist(double xmin,double xmax, int N, double* data,
char* head)
{int a,b,err;
 detectgraph(&a,&b);
 initgraph(&a,&b,"");
 err=graphresult();
 if (err!=grOk)
   {
     cout<<endl<<"Error of graphic initialization";
     } else {
     setcolor(MAGENTA);
     outtextxy(320-strlen(head)*4,5,head);
     setcolor(15);
     line(50,10,50,470);
     line(10,450,630,450);
     double nums[100],tmp,maximum;
     int i;
     maximum=data[0];
     for (i=1; i<N; i++)
     if (maximum<data[i]) maximum=data[i];
     line(40,50,60,50);
     setcolor(LIGHTBLUE);
     int dx=int(550/N);
     char *string;
     gcvt(maximum,4,string);
     outtextxy(5,40,string);
     nums[0]=xmin;
     nums[N]=xmax;
     tmp=(xmax-xmin)/N;
     for (i=1; i<N; i++) nums[i]=i*tmp+xmin;//X столбцов
     setlinestyle(SOLID_LINE,0,3);
     for (i=0; i<N; i++) {//draw columns
     setcolor(WHITE);
     bar(i*dx+50,450,(i+1)*dx+50,450-400.0*data[i]/maximum);
     setcolor(BROWN);
     rectangle(i*dx+50,450,(i+1)*dx+50,450-400.0*
     data[i]/maximum);
     gcvt(nums[i], 4, string);
```

```
        setcolor(LIGHTBLUE);
        outtextxy(int(i*dx+50),454,string);
        };
        gcvt(nums[N], 4, string);
        setcolor(LIGHTBLUE);
        outtextxy(int(N*dx+50),454,string);
        getch();// пауза после вывода гистограммы
        closegraph();
        };
};
```

5. Варианты заданий

Сгенерировать распределения:





6. Указания по составлению отчета

Отчет должен содержать постановку и описание задачи, алгоритм решения задачи, текст составленной программы на языке программирования С++, краткие выводы.

Библиографический список

1. Вентцель Е.С. Теория вероятностей. М.: Высшая школа, 2001.
2. Гмурман В.Е. Руководство к решению задач по теории вероятностей и математической статистике. М.: Высшая школа, 2001.

Лабораторная работа №10 Генерация случайных чисел в С

1. Цель и задачи работы

Изучение генератора случайных чисел в языке С. Написание программы с использованием изученных функций.

2. Теоретические сведения

Встроенные функции

В языке С имеется встроенный генератор случайных чисел. Вызов генератора осуществляется с помощью функций

```
int rand(void)  и
int random(int num)
```

Описание этих функций содержится в файле `stdlib.h`.

Первая функция возвращает псевдослучайные числа в диапазоне от 0 до RAND_MAX (RAND_MAX — константа, определенная в файле `stdlib.h`). Вторая функция возвращает случайные числа в диапазоне от 0 до num–1.

```
//Пример использования функции rand().
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   int i;

   printf("Десять случайных чисел от 0 до 99\n\n");
   for(i=0; i<10; i++)
      printf("%d\n", rand() % 100);
   return 0;
}
```

Результат:

Десять случайных чисел от 0 до 99
46
30
82
90
56
17
95
15
48
26

//Пример использования функции random.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
int main(void)
{
  printf
  ":Случайное число в диапазоне 0-99:%d\n",random (100));
   return 0;
}
```

Результат:
Случайное число в диапазоне 0-99: 42

Также в языке С имеется две вспомогательные функции для работы с генератором случайных чисел. Они также описаны в файле stdlib.h. Это функции

`void randomize(void)` и
`void srand(unsigned seed).`

Первая функция инициализирует случайным значением генератор случайных чисел, вызываемый с помощью функции `rand()`. При своей работе `randomize` использует функции, описанные в модуле `time.h`, поэтому при ее использовании нужно подключать этот модуль.

```
//Пример использования функции randomize.
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

  int main(void)
  {
     int i;

     randomize();
     printf("Десять случайных чисел от 0 до 99\n\n");
     for(i=0; i<10; i++)
         printf("%d\n", rand() % 100);
     return 0;
  }
```

Функция `void srand(unsigned seed)` инициализирует генератор случайных чисел `rand()` с помощью значения параметра `seed`. Благодаря этой функции, в С имеется возможность повторять генерацию случайных чисел, каждый раз получая при этом одинаковую последовательность.

```
//Пример использования функции srand.
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
```

```
int main(void)
{
   int i;
   time_t t;

   srand((unsigned) time(&t));
   printf("Десять случайных чисел от 0 до 99\n\n");
   for(i=0; i<10; i++)
       printf("%d\n", rand() % 100);
   return 0;
}
```

Генерация случайных чисел в некотором диапазоне

Для того, чтобы сгенерировать случайную величину в некотором диапазоне с помощью встроенных функций языка C, можно воспользоваться следующим приемом. Если требуется получить случайное число X в диапазоне от 0 до n, то это можно сделать, используя формулу:

X=rand()%(n+1),

где rand() — функция, о которой говорилось выше, % — операция деления по модулю.

Формула работает следующим образом. Функция rand(), как уже говорилось выше, возвращает число в диапазоне от 0 до RAND_MAX. После выполнения операции деления по модулю (n+1), получаем число в диапазоне от 0 до n (деление по модулю – это остаток от обычного деления на данное число, а остаток не может быть больше или равен делителю, т.е. он меньше n+1. n – максимальное значение результата, 0 – минимальное).

Если требуется получить случайное число X в диапазоне от некоторого m до n, то это можно сделать, используя формулу:

X=rand()%(n–m+1)+m.

После выполнения rand()%(n–m+1) получается число в диапазоне от 0 до n–m. После прибавления m получаем число в диапазоне от 0+m=m до n–m+m=n, что и было нужно.

Также можно сгенерировать случайное число в некотором диапазоне с помощью функции random(int num). Эта функция возвращает число в диапазоне от 0 до num–1.

Для генерации с помощью функции random(int num) случайного числа X в диапазоне от m до n можно использовать формулу:

X=random(n–m+1)+m.

Принцип действия аналогичной формулы был описан выше.

Генерация случайных чисел в диапазоне от 0 до 1 ($0 \le x_i \le 1$)

Для генерации случайных чисел в диапазоне от 0 до 1 удобно использовать функцию `random(int num)`. Чтобы получить число X в диапазоне от 0 до 1, можно воспользоваться формулой: X= float(random(n+1))/float(n).

Очевидно, если $0 \le k \le n$, то дробь $0 \le k/n \le 1$ всегда. В данном случае роль числа k играет random (n+1) и $0 \le$ random (n+1) $\le n$, т.е. $0 \le$ random (n+1)/n ≤ 1.

Определение нахождения случайной величины в одном из некоторых состояний

В различных инженерных задачах часто требуется определить нахождение случайной величины $0 \le x \le 1$ в одном из нескольких состояний $p(i)>0$, i=1,2,... n, $\Sigma p(i)=1$(проще говоря, попадание случайной величины в некоторый диапазон). Функция ниже показывает один из методов решения этой задачи.

3. Пример программы на языке C/C++

```
int rnd_diapazon(int i,float *mas)
{  //i-количество отрезков p(1)+p(2)+...+p(i)=1
   //Возвращаемое значение-номер отрезка,
   //если 0-произошла ошибка в данных
   //mas-массив из i элементов p(k), 0<=k<=(i-1),
   //mas[i] - состояние p(i+1)
   float interval; int n=1000;
   randomize();
   interval= float(random(n+1))/float(n);
   float x=0;
   switch(i)
   {  //если интервал один, случайная величина попадает
      //в него в любом случае
      case 1:return 1;

      //Случай с двумя интервалами
      case 2:{ if(interval<mas[0]) return 1;
               return 2;
             }
      //Если интервалов больше двух
      default:for(int k=0;k<=i;k++)
      {  if((interval>x)&&(interval<x+mas[k])) return k;
         x=x+mas[k];
      };
   };
   return 0;//ошибка в данных
}
```

4. Варианты заданий

1. Написать демонстрационную программу, которая использует функцию, генерирующую случайные числа в диапазоне от a до b.
2. Написать функцию, генерирующую случайные вещественные числа в диапазоне от a до b.
3. Написать функцию, генерирующую случайные двоичные числа.
4. Написать функцию, генерирующую случайные двоичные числа в диапазоне от a до b (a и b — двоичные числа).
5. Выполнить задание варианта 4 при условии, что a и b — десятичные числа.
6. Написать функцию, генерирующую случайную последовательность нулей и единиц.
7. Написать программу, определяющую повторяющиеся значения в последовательности случайных чисел, генерируемых функцией rand.
8. Написать программу, определяющую повторяющиеся значения в последовательности случайных чисел, генерируемых функцией random.
9. Написать программу генерации случайных чисел в диапазоне от –k до k.
10. Написать программу, генерирующую вещественные случайные числа в диапазоне от –k до k.
11. Написать программу, выводящую на экран «звездное небо» в текстовом режиме.
12. Написать программу, выводящую на экран «звездное небо» в графическом режиме.
13. Написать программу, изображающую на экране движение точки со случайными приращениями координат («броуновское движение»).
14. Написать программу «заливки» экрана отдельными точками, координаты которых определяются случайно.
15. Написать программу, изображающую линии случайной длины, соединенные последовательно друг с другом под случайным углом, кратным 90°.
16. Написать функцию, генерирующую случайные простые числа.
17. Написать функцию, генерирующую случайные простые числа в диапазоне от a до b.
18. Написать функцию, генерирующую случайные чётные числа.
19. Написать функцию, генерирующую случайные нечётные числа.
20. Написать программу, «заливки» области экрана с координатами левой верхней вершины — x_1, y_1 и правой нижней — x_2, y_2, отдельными точками, координаты которых определяются случайно.

5. Указания по составлению отчета

Отчет должен содержать постановку и описание задачи, алгоритм решения задачи, текст составленной программы на языке программирования C/C++, краткие выводы.

Лабораторная работа №11 Сортировка методом Шелла

1. Цель и задачи работы

Исследовать метод сортировки массивов методом Шелла, провести анализ метода сортировки массива, составить схему алгоритма для сортировки, написать программу на языке С/С++.

2. Теоретические сведения

Описание метода

Этот метод назван по имени его изобретателя (D.L. Shell). Он построен на основе метода вставки с минимизацией промежуточных шагов. Рассмотрим иллюстрацию на рис. 1. Сначала выполняется сортировка элементов, отстоящих друг о друга на три позиции. После этого сортируются элементы, отстоящие друг от друга на две позиции. Наконец, выполняется сортировка смежных элементов.[1]

В таблице на рис. 2 проиллюстрирована следующим образом идея, лежащая в основе метода Шелла. Сначала делим 16 записей на 8 групп по две записи в каждой группе: *(R1, R9), (R2,R10), ... ,(R8,R16)*. В результате сортировки каждой группы записей по отдельности приходим ко второй строке таблицы рис. 2., это называется «первым просмотром». Заметим, что элементы 154 и 512 поменялись местами, а 908 и 897 переместились вправо. Разделим теперь записи на четыре группы по четыре в каждой: *(R1, R5, R9, R13), ... , (R4, R8, R13, R16)* — и опять отсортируем каждую группу в отдельности; этот второй просмотр приводит к третьей строке таблицы. При третьем просмотре сортируются две группы по восемь записей; процесс завершается четвертым просмотром, во время которого сортируются все 16 записей. В каждом из этих промежуточных процессов сортировки участвуют либо сравнительно короткие файлы, либо уже сравнительно хорошо упорядоченные файлы, поэтому на каждом этапе можно пользоваться простыми вставками; записи довольно быстро достигают своего конечного сведения.[2]

Первый взгляд на этот алгоритм не дает понимания того, почему этот кажущийся столь сложным метод дает столь хорошие результаты. Непонятно даже, как он вообще сортирует массив. Однако метод работает. Каждый промежуточный шаг задействует относительно небольшое количество элементов, которые к тому же могут уже находиться в нужном порядке. Поэтому метод Шелла эффективен, и упорядоченность массива возрастает после каждого прохода.

---

[1] Шилдт Г. Теория и практика С++: пер. с англ. – СПб.: БХВ – Петербург, 2001. С. 31.

[2] Д. Кнут. Искусство программирования для ЭВМ. Т. 3. Издательство «Мир». М., 1977.

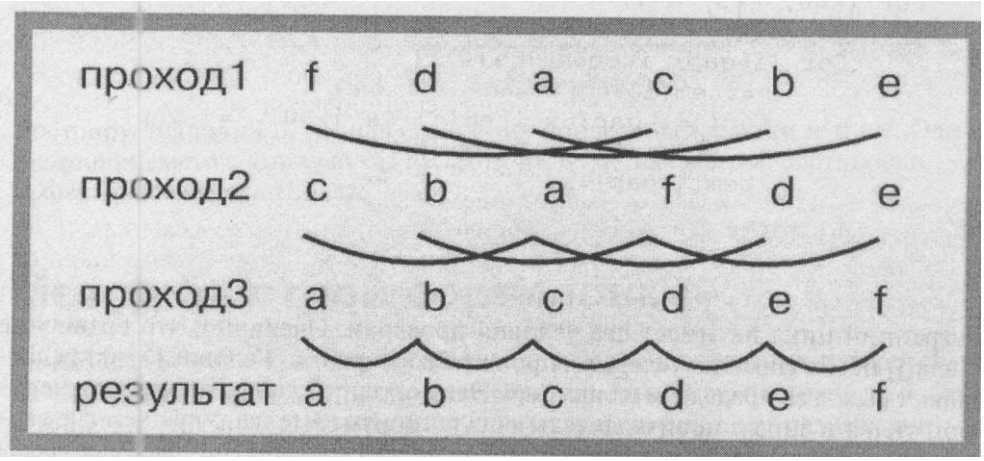


Рис. 1. Сортировка методом Шелла (по Шилдту)

Рис. 2. Сортировка методом Шелла (по Кнуту)

Анализ метода

Скорость работы данного метода зависит от числа групп, на которые разбивается основной массив. Эти числа должны выбираться с осторожностью, потому что они оказывают влияние ни количество итераций, выполняемых программой. Как ранее отмечалось, хорошо себя зарекомендовала последовательность 9, 5, 3, 2, 1. По некоторым данным наилучший результат дает разбивка на ... 3280, 1093, 364, 121, 40, 13, 4, 1, где каждое предыдущее число вычисляется умножением последующего на 3 и добавлением единицы. Надо начинать с числа максимального, но меньшего, чем размер массива. Например, сортировать массив из 5000 элементов надо с числа 3280. Нет полной гарантии, что это лучший выбор. Можно попробовать использовать другую последовательность, может быть, получится лучший результат. Единственным требованием остается равенство последнего приращения 1.

Кроме того, время выполнения алгоритма пропорционально n1.2 при сортировке n элементов. Это – существенный прогресс по сравнению с n-квадратными методами сортировки. Например, сортировка массива из

3600 элементов выполняется менее чем за одну секунду на процессоре 133MHz Pentium.

Алгоритм хорошо работает на массивах средних по размерам и больших массивах.

3. Схема алгоритма сортировки методом Шелла

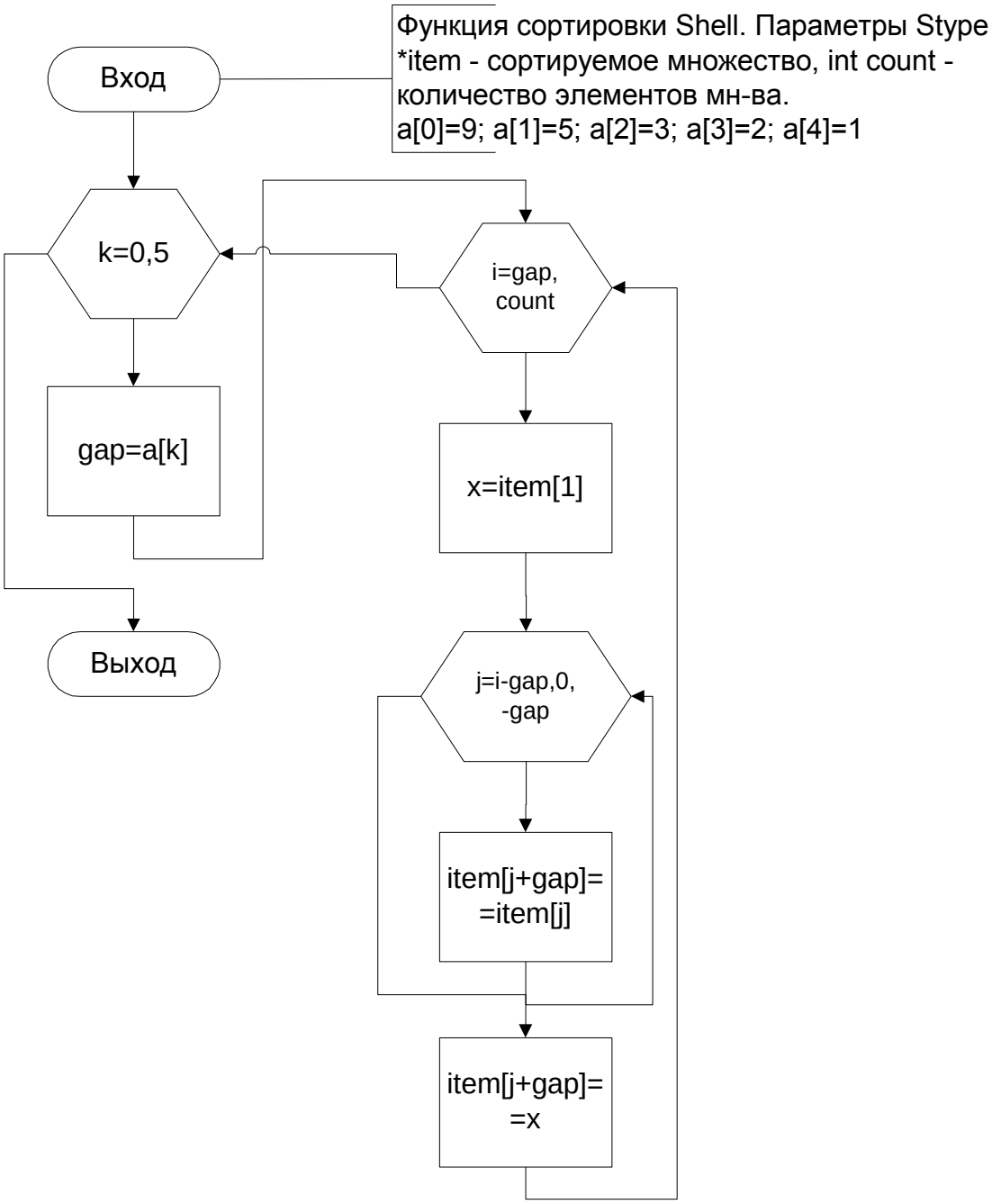


Рис. 3. Схема алгоритма сортировки методом Шелла.

4. Пример программы на языке C/C++

```
#include <iostream.h>
#include <string.h>
template <class Stype> void shell (Stype *item, int count);
int main ()
{
//сортировка массива символов
    char str[] = "dcab";
    shell (str, strlen(str)) ;
    cout << "Отсортированная строка: " << str << "\n";
// сортировка массива целых чисел
     int nums[] = {5, 7, 3, 9, 5, 1, 8};
    int i;
    shell (nums, 7);
    cout << "Отсортированный массив: ";
     for(i=0; i<7; i++) cout << nums[i] << " ";
     return 0;
}
// Сам метод сортировки
template <class Stype> void shell (Stype *item, int count)
{
     int i, j, gap, k;
     Stype x;
     int a[5] = {9,5,3,2,1};
      for (k=0; k<5; k++)
    {
         gap=a[k];
         for (i=gap; i<count; ++i)
         {
            x=item[1];
            for(j=i-gap; x<item[j] && j>=0; j=j-gap)
                  item[j+gap]=item[j];
            item[j+gap]=x;
         }
      }
     }
```

5. Варианты заданий

1. Отсортировать символьный массив, состоящий из 30 элементов.
2. Отсортировать числа (0…9), заданные в произвольном порядке.
3. Отсортировать массив, состоящий из положительных чисел, по возрастанию.
4. Отсортировать массив, состоящий из отрицательных чисел, по убыванию.
5. Отсортировать массив, состоящий из положительных двузначных чисел.
6. Произвести сортировку массива, состоящего из 0, 1 и 2, записанных в произвольном порядке (количество элементов n не менее 6).
7. Отсортировать текст, расположив слова в алфавитном порядке, и вывести результат в новый файл (считать, что a=1, b=2, c=3, …, z=26, а = 27, б = 28, в = 29, …я = 59).

8. Отсортировать клавиши букв латинского алфавита на клавиатуре в порядке возрастания их ASCII –кодов.
9. Отсортировать по возрастанию массив, состоящий букв вашего имени, отчества и фамилии, считая, что а = 1, б = 2, в = 3, …я = 33.
10. Отсортировать буквы вашего отчества по убыванию по схеме, указанной в задании 9.
11. Отсортировать массив, состоящий из (2n + 1) элементов. Сортировку до (n + 1) –го элемента осуществлять по убыванию, а после (n + 1)-го – по возрастанию,.

$$\underline{4\ \ 3\ \ 2\ \ 1}\ \ 5\ \ \underline{1\ \ 2\ \ 3\ \ 4}$$
$$\text{n}\qquad\text{n+1}\qquad\text{n}$$

12. Отсортировать по возрастанию массив, состоящий букв вашего имени, отчества и фамилии, учитывая пробелы между ними по схеме из задания 9, и считая <пробел>=0.
13. Отсортировать клавиши с кириллицей на клавиатуре в порядке возрастания их ASCII – кодов.
14. Отсортировать массив, состоящий из отрицательных двузначных чисел в порядке возрастания.
15. Отсортировать массив, состоящий из студентов вашей группы (расположенных в массиве произвольно), по возрастанию, по схеме из задания 9.
16. Отсортировать массив из задания 13 по убыванию.
17. Отсортировать массив, состоящий из трехзначных чисел по возрастанию.
18. Отсортировать массив, состоящий из трехзначных чисел по убыванию.
19. Отсортировать массив, состоящий из положительных чисел по убыванию.
20. Отсортировать массив, состоящий из отрицательных двузначных чисел в порядке возрастания.

6. Указания по составлению отчета

Отчет должен содержать постановку и описание задачи, алгоритм решения задачи, текст составленной программы на языке С/С++, краткие выводы.

Библиографический список

1. Шилдт Г. Теория и практика С++: пер. с англ. СПб.: БХВ – Петербург, 2001.

Лабораторная работа №12 Основы рекурсии

1. Цель и задачи работы

Ознакомиться с понятием рекурсии, основными видами рекурсии, примерами ее применения. Научиться описывать и использовать рекурсивные функции. Написать программу, обрабатывающую данные рекурсивным способом.

2. Теоретические сведения

Понятие рекурсии

Объект называется *рекурсивным,* если он полностью или частично определен через самого себя. Таким образом, под *рекурсией* понимают наличие у некоторого объекта свойства рекурсивности. Заметим, что рекурсия встречается не только в математике, но и в повседневной жизни. Примером может служить рекламная картинка, которая содержит свое собственное изображение.

Рассмотрим явление рекурсии на примере некоторой функции S(n), которая вычисляет сумму первых n положительных целых - задачу, решаемую путем многократного сложения.

$$S(n) = \sum_{1}^{n} i = 1 + 2 + 3 + ... + n - 1 + n$$

Например, для S(10) необходимо сложить первые 10 целых, чтобы получить ответ 55:

$$S(n) = \sum_{1}^{10} i = 1 + 2 + 3 + ... + 9 + 10 = 55$$

Если применить этот же алгоритм для вычисления S(11), процесс повторит все эти сложения. Более практичным подходом было бы использовать предыдущий результат для S(10), а затем добавить к нему 11, чтобы получит ответ S(ll) = 66:

$$S(n) = \sum_{1}^{11} i = S(10) + 11 = 55 + 11 = 66$$

При таком подходе для получения результата используется предыдущий результат вычислений. Это называется *рекурсивным процессом*.

Теперь рассмотрим степенную функцию x^n, где x — действительное число, а n - неотрицательное целое. Во-первых, исходя из определения натуральной степени числа, типичным будет следующая последовательность действий:

$$x^n = \underbrace{x * x * x * ... * x * x}_{n \text{ множителей}}$$

Второй способ вытекает из следующих соображений:

$$2^0 = 1;$$

$$2^1 = 2;$$

$$2^2 = 2 * 2 = 4;$$

$$2^3 = 2 * 4 = 8;$$

$$2^4 = 2 * 8 = 16.$$

Попробуем представить степенную функцию с помощью рекурсивного процесса. Будем опираться на то, что начальная степень двойки $2^0 = 1$, а последующие ее степени есть всего лишь удвоение предыдущего значения. Процесс использования меньшей степени для вычисления очередной приводит к рекурсивному определению степенной функции. Для действительного x значение x^n определяется как

$$x^n = \begin{cases} 1, & при \quad n = 0, \\ x * x^{(n-1)}, & при \quad n > 0. \end{cases}$$

Похожее рекурсивное определение описывает функцию S(n), дающую сумму первых n целых чисел. Для простого случая S(1) сумма равна 1. Сумма S(n) может быть получена из S(n-1):

$$S(n) = \begin{cases} 1, & при \quad n = 1, \\ n + S(n-1), & при \quad n > 1. \end{cases}$$

Рекурсия имеет место при решении какой-то задачи посредством разбиения ее на меньшие подзадачи, выполняемые с помощью одного и того же алгоритма. Процесс разбиения завершается при достижении простейших возможных решаемых подзадач. Эти простейшие задачи называются *условиями останова*. Можно сказать, что рекурсия действует по принципу «разделяй и властвуй».

Алгоритм определен рекурсивно, если это определение состоит из:

- одного или нескольких условий останова, которые могут быть вычислены для определенных параметров;
- шага рекурсии, в котором текущее значение в алгоритме может быть определено в терминах предыдущего значения. В конечном итоге шаг рекурсии должен приводить к условиям останова.

Например, рекурсивное определение степенной функции имеет единственное условие останова для случая n = 0 ($x^0 = 1$). Шаг рекурсии описывает общий случай $x^n = x * x^{n-1}$, при n > 0.

Рекурсия позволяет определить бесконечное множество объектов с помощью конечного высказывания. Точно так же бесконечные вычисления можно описать с помощью конечной рекурсивной программы, даже если эта программа не содержит явных циклов. В общем виде рекурсивную программу *P* можно изобразить как композицию ρ базовых, операторов S_i (не содержащих *P*) и самой *P*: $P \equiv \rho[S_i, P]$.

Виды рекурсии

Необходимое и достаточное средство для рекурсивного представления программ — это описание процедур, или подпрограмм, так как оно позволяет присваивать какому-либо оператору имя, с помощью которого можно вызывать этот оператор. Если подпрограмма *P* содержит явное обращение к самой себе, то она называется *прямо рекурсивной;* если *P* содержит обращение к подпрограмме *Q*, которая содержит (прямо или косвенно) обращение к *P*, то *P* называется *косвенно рекурсивной.*

Рекурсивные алгоритмы можно классифицировать следующим образом:

1. *Обычная* рекурсия: рекурсивный вызов подпрограммы происходит из тела самой подпрограммы.
2. *Параллельная* рекурсия: из тела подпрограммы происходит несколько параллельных вызовов этой подпрограммы.
3. *Взаимная* рекурсия: вызов 1-ой подпрограммы происходит из тела другой подпрограммы, которая, в свою очередь, была вызвана из 1-ой подпрограммы.
4. Рекурсия *более высокого порядка*: при этом может происходить сколь угодно сложный перекрестный рекурсивный вызов большого числа подпрограмм.

С подпрограммой принято связывать некоторое множество локальных объектов, т. е. переменных, констант, типов и подпрограмм, которые определены локально в этой подпрограмме, а вне ее не существуют или не имеют смысла. Каждый раз, когда такая подпрограмма рекурсивно вызывается, обычно для нее создается новое множество локальных переменных. Хотя они имеют те же имена, что и соответствующие элементы множества локальных переменных, созданного при предыдущем обращении к этой же подпрограмме, их значения различны. Это важно иметь в виду, когда объем памяти, отводимый под работу рекурсивной программы, критичен.

Если рекурсивный алгоритм имеет конечное число вычислений, то в нем всегда можно выделить *входную часть* и *циклическую*. Под входной частью алгоритма подразумевается та его часть, которая уже не будет содержать вызова этой подпрограммы в том или ином виде, тем самым как бы начиная рекурсивное движение в обратную сторону. Так, например, в алгоритме вычисления факториала $f(n)$ входной частью будет условие $if\ n=1\ then\ f=n$. Циклической же частью рекурсивного алгоритма называется та его часть, которая содержит рекурсивный вызов этой подпрограммы в том или ином виде. При этом максимальное количество рекурсивных вызовов подпрограммы называется *глубиной рекурсии* или глубиной рекурсивного спуска.

Подобно операторам цикла, рекурсивные подпрограммы могут привести к бесконечным вычислениям. Поэтому очень важной является проблема *окончания работы* рекурсивного алгоритма. Очевидно, что для того, чтобы работа когда-либо завершилась, необходимо, чтобы рекурсивное обращение к подпрограмме *P* подчинялось условию *B,* которое в какой-то момент перестает выполняться. Поэтому более точно схему рекурсивных алгоритмов можно представить так:

$$P \equiv if\ B\ then\ \rho[S_i, P] \quad или \quad P \equiv \rho[S_i, if\ B\ then\ P].$$

Основной способ доказать, что выполнение операторов цикла когда-либо заканчивается - определить функцию $f(x)$ (*x* - множество переменных программы), такую, что $f(x) \le 0$ удовлетворяет условию окончания цикла (с предусловием или с постусловием), и доказать, что при каждом повторении $f(x)$ уменьшается. Точно так же можно доказать, что выполнение рекурсивной процедуры *P* когда-либо завершится, показав, что каждое выполнение *P* уменьшает $f(x)$. Наиболее надежный способ обеспечить окончание подпрограммы - связать с *P* параметр (значение), скажем *n,* и рекурсивно вызывать *P* со значением этого параметра *n*-1. Тогда замена условия *B* на *n*>0 гарантирует окончание работы. Это можно изобразить следующими схемами программ:

$$1.\ P \equiv if\ n > 0\ then\ \rho[S_i, P];$$

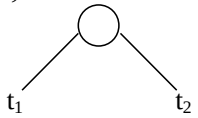
$$2.\ P \equiv \rho[S_i, if\ B\ then\ P].$$

На практике нужно обязательно убедиться, что наибольшая глубина рекурсии не только конечна, но и достаточно мала. Дело в том, что при каждом рекурсивном вызове процедуры *P* выделяется некоторая память для размещения ее переменных. Кроме этих локальных переменных нужно еще сохранять текущее состояние вычислений, чтобы вернуться к нему, когда закончится выполнение новой активации *P* и нужно будет вернуться к старой.

Примеры рекурсивных задач

Рекурсия является особенно мощным средством в математических определениях. Рассмотрим примеры рекурсивных определений натуральных чисел, древовидных структур и некоторых функций:

1. Натуральные числа:
   (a) 1 есть натуральное число;
   (b) целое число, следующее за натуральным, есть натуральное число.
2. Древовидные структуры:
   (a) О есть дерево (называемое пустым деревом);
   (b) если t_1 и t_2 — деревья, то



есть дерево (нарисованное корнем вверх).

3. Функция факториал n! для неотрицательных целых чисел;

(a) $0!=1$, (b) *если* $n>0$, *то* $n!=n*(n-1)!$

Сила рекурсии обеспечивает весьма простые и изящные решения ряда проблем. Рассмотрим некоторые из них.

Ханойская башня. Одной из самых интересных задач, имеющих ярко выраженное рекурсивное решение, является задача о Ханойской башне. Согласно легенде, у жрецов храма Брахмы есть медная платформа с тремя алмазными шпилями. На одном шпиле А нанизано 64 золотых диска, каждый из которых немного меньше того, что под ним. Конец света наступит, когда жрецы переместят все диски со шпиля А на шпиль С. Однако задача имеет весьма специфические условия. За один раз можно перемещать только один диск, и при этом ни разу диск большего размера не должен лечь на диск меньшего размера. В таком случае, жрецам необходимо было бы выполнить $2^{64}-1$ хода. Если тратить по одной секунде на ход, то на это потребуется 500 миллиардов лет!

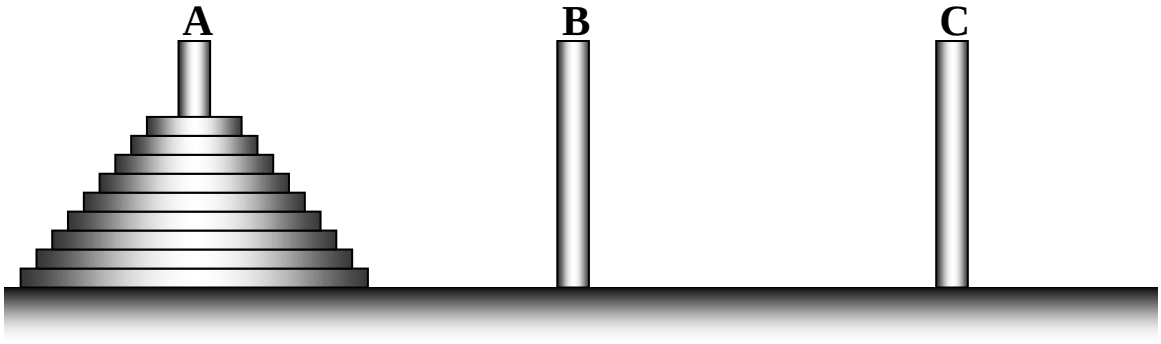


Рис. 1. Ханойская башня - Начальное положение.

Задача о Ханойской башне является достаточно тяжелой, если не заметить тех преимуществ, которые дает рекурсивное решение. Рассмотрим решение этой задачи на примере 9 дисков. Для начала сосредоточимся на перемещении верхних 8 дисков на шпиль *B* и последующем перемещении самого большого диска на шпиль *C*.

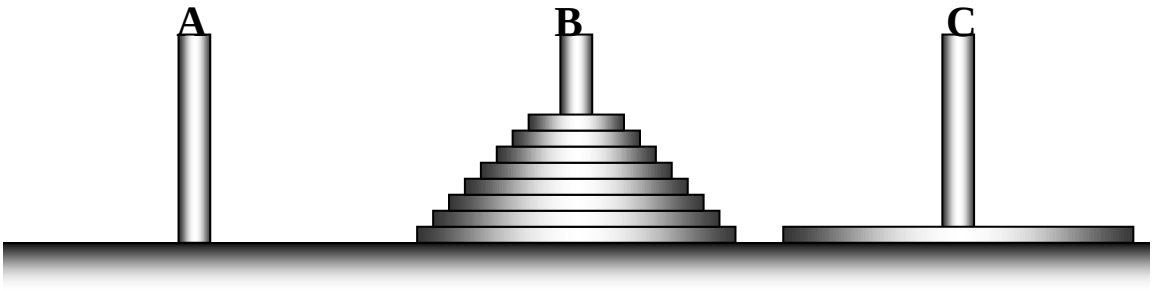


Рис. 2. Ханойская башня - Самый большой диск на шпиле С.

Осталась более простая задача перемещения только 8 дисков со шпиля В на шпиль С. Применяя тот же самый алгоритм, сосредоточим свое внимание на верхних 7 дисках и вытащим их из стопки. Затем перенесем самый большой диск из оставшихся со шпиля В на шпиль С.

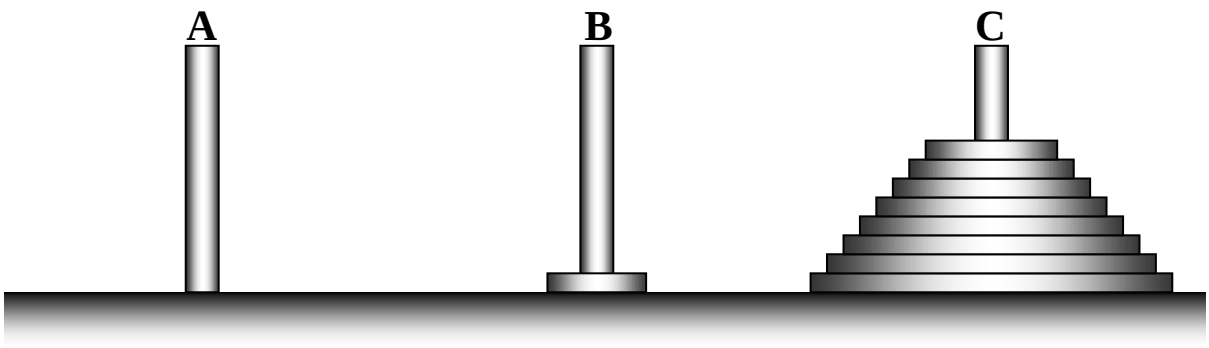


Рис. 3. Ханойская башня - Условие останова.

Очевидно, что данное решение носит рекурсивный характер. Проблема разбивается на последовательность меньших подзадач одного и того же типа, Условием останова является простая задача перемещения одного диска.

Комбинаторика. Рекурсия находит широкое применение в комбинаторике. Предположим, бросаются три игральные кости и записывается общий итог. Одним из вопросов комбинаторики является количество различных способов, которыми можно выбрать 8 очков из бросания трех игральных костей. Рекурсивный подход пытается свести проблему к постепенно упрощающимся задачам и использовать эти результаты для решения более сложной проблемы. В этом случае три игральные кости считаются сложной проблемой, и внимание фокусируется на более простом случае бросания двух игральных костей. Тем самым предполагается, что можно взять две игральные кости и любой результат N в диапазоне от 2 до 12 и определить все различные способы выпадения N. В случае трех костей, дающих в сумме 8, бросается первую кость и записывается значение в таблице вместе со значением N, представляющим оставшееся количество очков, которое должно быть набрано двумя следующими костями. Например, если на первой кости выпало 3, следующие две должны дать в сумме 5. Опираясь на результаты опытов с двумя игральными костями, определяется, что есть четыре возможных исхода бросания двух костей, дающих 5. Объединяя эти исходы с тройкой на первой кости, получается, как минимум, четыре способа выбросить восьмерку на трех костях. Комбинаторика имеет огромное множество подобных задач, имеющих рекурсивное решение.

Лабиринт. Еще одним примером рекурсивного алгоритма служит решение задачи о выходе из лабиринта. Пусть некоторым образом задан лабиринт, а также начальное положение, откуда необходимо найти путь к выходу. При прохождении этого лабиринта возникает бесконечное число альтернатив, приводящих в тупики и окончательному запутыванию. Такая задача является достаточно сложной и для ее решения необходимо создание специальных алгоритмов. При этом данная задача является ярким

примером рекурсии с возвратом, которая будет более подробно рассмотрена в следующей лабораторной работе.

Применение рекурсии

Область применения рекурсии при решении различных задач очень широка, т.к. огромное количество итеративных алгоритмов можно определить в рекурсивных терминах. Обычно рекурсивные задачи имеют достаточно красивые решения с точки зрения искусства программирования, однако, в применении рекурсии есть и свои подводные камни. Таким, например, является большой расход памяти компьютера в большинстве рекурсивных алгоритмов по сравнению с аналогичными решениями подобных задач без использования рекурсии.

Рекурсивные алгоритмы наиболее пригодны в случаях, когда поставленная задача или используемые данные определены рекурсивно. Но это не значит, что при наличии таких рекурсивных определений лучшим способом решения задачи непременно является рекурсивный алгоритм. В действительности из-за того, что обычно понятие рекурсивных алгоритмов объяснялось на неподходящих примерах, в основном и возникло широко распространенное предубеждение против использования рекурсии в программировании и приравнивание ее к неэффективности.

Программы, в которых следует избегать использования рекурсии, можно охарактеризовать следующей схемой, изображающей их строение. Это схема (1) и эквивалентная ей (2):

$$P \equiv if\ B\ then\ \rho[S; P], \quad (1)$$

$$P \equiv \rho[S; if\ B\ then\ P]. \quad (2)$$

Эти схемы естественно применять в тех случаях, когда вычисляемые значения определяются с помощью простых рекуррентных соотношений. Такими, например, являются задачи вычисления факториала и последовательности Фибоначчи.

Итак, вывод таков: следует избегать рекурсии, когда имеется *очевидное* итеративное решение поставленной задачи.

Но это не означает, что всегда нужно избавляться от рекурсии любой ценой. Во многих случаях она вполне применима, как будет показано в следующих разделах этой главы и в последующих главах. Тот факт, что рекурсивные процедуры можно реализовать на *нерекурсивных* по сути машинах, говорит о том, что для практических целей любую рекурсивную программу можно преобразовать в чисто итеративную. Но это требует явного манипулирования со стеком рекурсий, и эти операции до такой степени заслоняют суть программы, что понять ее становится очень трудно. Следовательно, алгоритмы, которые по своей природе скорее рекурсивны, чем итеративны, нужно представлять в виде рекурсивных процедур.

3. Контрольные вопросы

1. Что такое *рекурсия*?
2. Где встречается рекурсия?
3. Приведите примеры рекурсии?
4. Назовите признаки рекурсивности объекта.
5. Перечислите виды рекурсии.
6. Что такое *обычная* рекурсия?
7. Что такое *параллельная* рекурсия?
8. Что такое *взаимная* рекурсия?
9. Что такое рекурсия *более высокого уровня*?
10. Из каких частей состоит рекурсивная подпрограмма?
11. Что содержится во *входной* части рекурсии?
12. Что содержится в *циклической* части рекурсии?
13. Что называется глубиной *рекурсии*?
14. Как решается *проблема остановки* рекурсивного алгоритма?
15. В каких математических задачах применяют понятие рекурсии?
16. Можно ли дать определение множеству натуральных чисел с помощью рекурсии?
17. Можно ли дать определение множеству целых чисел с помощью рекурсии?
18. Можно ли дать определение множеству действительных чисел с помощью рекурсии?
19. В чем заключается задача о *ханойской башне*?
20. В чем заключается задача о *лабиринте*?
21. Придумайте комбинаторные задачи, для решения которых можно использовать рекурсивные алгоритмы.
22. Назовите *достоинства* применения рекурсивных алгоритмов для решения практических задач.
23. Назовите *недостатки* применения рекурсивных алгоритмов.
24. Всегда ли возможно реализовать рекурсивный алгоритм без использования рекурсии?
25. В каких случаях рекомендуется избегать применения рекурсии?

4. Варианты заданий

1. «Ханойская башня». Имеются три колышка A,B,C и N дисков разного диаметра на A, пронумерованных от 1 до N в порядке возрастания их размеров. Требуется перенести все диски с колышка A на колышек C, соблюдая при этом следующие условия: диски можно переносить только по одному, больший диск нельзя ставить на меньший.
2. Вычислить все члены последовательности Фибоначчи от 1 до n.

3. Вычислить X в степени Y путем многократного умножения, где X и Y - целые числа. Вычисление производить по алгоритму половинного деления степени.
4. Описать рекурсивную функцию, которая методом деления отрезка пополам находит на интервале от A до B корень уравнения с некоторой заданной точностью.
5. Написать функцию $C(m,n)$, где $0 \le m \le n$, для вычисления биноминальных коэффициентов: $C(m,n) = \frac{n!}{m!*(n-m)!}$.
6. Отсортировать массив методом «выбора».
7. Перевести целое число из одной системы счисления в другую.
8. Дано N различных натуральных чисел. Напечатать все возможные сочетания этих чисел.
9. Построение треугольника Паскаля.
10. Вычислить функцию $y = \sin(x)$, разложенную в степенной ряд, с заданной степенью точности.
11. Нахождение НОД для двух натуральных чисел с использованием алгоритма Евклида.
12. Найти количество чисел меньше числа N (1000<N<100000), составленных из некоторых цифр A, B, C.
13. Найти значение X в степени Y, где X - вещественное число, а Y - целое. Вычислять согласно формуле:

$$x^Y = \begin{cases} 1 & при \quad Y = 0 \\ 1/x^Y & при \quad Y < 0 \\ x * x^{Y-1} & при \quad Y > 0 \end{cases}$$

14. Вычислить функцию $y = \ln(x)$, разложенную в степенной ряд, с заданной степенью точности.
15. Вычислить все простые числа на интервале от 2 до n.
16. Задана последовательность символов. Описать функцию проверяющую, является ли симметричной последовательность символов в этой последовательности, начиная с i-того и кончающегося на j-ом элементе
17. Вычислить рекурсивно функцию вида:

$$y = \cos(x) + \cos(x^2) + \cos(x^3) + ... + \cos(x^n).$$

18. Написать функцию, вычисляющую остаток от деления нацело для двух целых чисел. Не использовать аналогичные стандартные функции и операции умножения, деления и возведения в степень.
19. Написать функцию, выполняющую деление нацело одного числа на другое. Не использовать аналогичные стандартные функции и операции умножения, деления и возведения в степень.

20. Задана последовательность вещественных чисел, описать функцию, которая ищет минимальный элемент в этой последовательности.

5. Указания по составлению отчета

Отчет должен содержать постановку и описание задачи, алгоритм решения задачи, текст составленной программы на языке программирования C++, краткие выводы.

Лабораторная работа №13 Рекурсия с возвратом

1. Цель и задачи работы

Ознакомление с понятием «рекурсия с возвратом», изучение принципов организации рекурсивно-возвратных алгоритмов. Написание программы рекурсивно-возвратного решения задачи в соответствии с вариантом задания.

2. Теоретические сведения

Особенно интересный раздел программирования — это задачи из области искусственного интеллекта. Обычно в таких задачах ставятся вопросы-задания, например, «Сколько существует способов…», «Подсчитайте количество элементов…» и т.п. Ответы на них, как правило, требуют исчерпывающего поиска в некотором множестве M всех возможных вариантов, среди которых находятся решения конкретной задачи. Существует два общих метода организации исчерпывающего поиска: перебор с возвратом и его естественное логическое дополнение – метод решета.

Соединение метода перебора с возвратом с рекурсией определяет специфический способ реализации рекурсивных вычислений и заслуживает своего отдельного названия. Рекурсию, в которой используется метод перебора с возвратом, мы будем называть *возвратной рекурсией.* Ценность метода возвратной рекурсии несомненна. Во-первых, программы решения многих задач строятся по единой схеме, а во-вторых, они компактны и тем самым просты для понимания и усвоения соответствующих идей.

Опишем некоторую совокупность комбинаторных задач, к которым заведомо применим алгоритм перебора с возвратом.

Пусть $M_0, M_1, \ldots, M_{n-1}$ — n конечных линейно упорядоченных множеств, а G – совокупность ограничений (условий), ставящих в соответствие векторам вида

$$v = (v_0, \ldots, v_k)^T \quad (v_j \in M_j;\ j = 0, 1, \ldots, k;\ j \le n-1)$$

булево значение $G(v) \in \{$*истина*, *ложь*$\}$. Векторы $v = (v_0, \ldots, v_k)^T$, для которых $G(v)$ = *истина*, назовем *частичными решениями*. Пусть существует конкретное правило P, в соответствии с которым некоторые из частичных решений могут объявляться *полными решениями*. Тогда возможна постановка двух следующих поисковых задач:

- найти все полные решения или установить отсутствие таковых;
- найти хотя бы одно полное решение или установить его отсутствие.

Наиболее часто подобные задачи формулируются и решаются при следующем задании правила *P* на частичных решениях [1, с.108-114], [2-5]:

$$P(v_0,...,v_k)=\begin{cases} полное \quad решение \quad при \quad k=n-1, \\ неполное \quad решение \quad при \quad k<n-1. \end{cases}$$

Общий метод решения приведенных задач состоит в последовательном покомпонентном наращивании вектора *v* слева направо, начиная с v_0, и последующих испытаниях его ограничениями *G* и правилом *P*.

В п.3 на некотором паскалеподобном псевдокоде приведены три общие схемы решения задач методом перебора с возвратом: нерекурсивный вариант поиска всех решений (схема 1), рекурсивный вариант поиска всех решений (схема 2), рекурсивный вариант поиска одного решения (схема 3).

По схеме 1 находятся все решения задачи, если они есть, и работа завершается при $j = 0$. Ее рекурсивный вариант, представленный на схеме 2, существенно более прост и очевиден.

Генерирование всех решений по схеме 2, если они есть, организуется вызовом backtracking(0). Нахождение одного решения по схеме 3, если оно есть, проводится вызовом: backtracking(0,’решение не найдено’).

В общем случае этот метод перебора с возвратом приводит к алгоритмам с экспоненциальной временной сложностью. Применяется он в основном к классу так называемых *Np*-полных задач (задача коммивояжера, задача о рюкзаке и т.д.), эквивалентных друг другу в том смысле, что все они разрешимы недетерминированными алгоритмами полиномиальной сложности. Далее для них известно, что, либо все они разрешимы, либо ни одна из них неразрешима детерминированными алгоритмами полиномиальной сложности. Иными словами, если хотя бы для одной из этих задач не существует детерминированного алгоритма, имеющего в худшем случае полиномиальную трудоемкость, то таких алгоритмов не должно существовать и для остальных задач этого класса. Наоборот, если хотя бы для одной из этих задач удалось найти детерминированный алгоритм, имеющий в худшем случае полиномиальную трудоемкость, то подобные алгоритмы существовали бы и для остальных задач этого класса и, более того, их можно было бы построить.

Несколько слов об упомянутых выше недетерминированных алгоритмах. Поясним смысл этого понятия. Пусть алгоритм выполняется до тех пор, пока не доходит до места, с которого должен быть сделан выбор из нескольких альтернатив. Детерминированный алгоритм

однозначно осуществит выбор конкретной альтернативы и продолжит работать в соответствии с этим выбором. Недетерминированный алгоритм исследует все возможности одновременно, как бы копируя себя для реализации вычислений по всем альтернативам одновременно (оператор *выбор)*.

Затем все копии работают независимо друг от друга и по мере необходимости продолжают создавать новые копии. Копия, сделавшая неправильный или безрезультатный выбор, прекращает свою работу (оператор *неуспех*). Копия, нашедшая решение задачи, объявляет об этом (оператор *успех*), давая тем самым сигнал другим копиям о прекращении вычислений. Недетерминированные алгоритмы, являясь весьма полезной и продуктивной абстракцией, рекурсивны по сути, ибо при реализации оператора выбора они фактически обращаются сами к себе.

Далее мы будем рассматривать только детерминированные рекурсивные алгоритмы.

Обход шахматной доски конем

Рассмотрим конкретное воплощение абстрактной схемы 3 для известной задачи об обходе конем шахматной доски размера n×n. Будем считать, что строки и столбцы доски пронумерованы соответственно сверху вниз и слева направо цифрами от 0 до $n - 1$, а конь перемещается по доске согласно обычным шахматным правилам. Найти обход доски с конкретного поля означает, что нужно найти последовательность позиций перемещения коня такую, что каждое поле посещается им ровно один раз. Для доски 8×8 эта задача впервые была поставлена Л. Эйлером.

Постановка задачи. На шахматной доске размера $n \times n$ в позиции (x, y) находится конь (см. рис. 1). Составить рекурсивную программу-функцию, находящую методом перебора с возвратом обход доски конем, если он существует. При отсутствии обхода должно быть выдано соответствующее сообщение.

Решение. На рис.1 приведен фрагмент доски. Конь K стоит в позиции (x, y). Клетки с цифрами вокруг K – это поля, на которые конь может переместиться из (x, y) ($0 \le x, y \le n - 1$) за один ход. Множество этих полей обозначим через $M(x,y)$. Для фиксированного поля (x, y) множество $M(x, y)$ может содержать от двух до восьми элементов. Попробуем упорядочить эти множества. Рассмотрим вспомогательную матрицу приращений

$$D = \begin{bmatrix} 1 & -1 & -2 & -2 & -1 & 1 & 2 & 2 \\ 2 & 2 & 1 & -1 & -2 & -2 & -1 & 1 \end{bmatrix} \qquad (1)$$

Для поля (x, y) построим последовательность полей

$$(x + D_{0,k}, y + D_{1,k}) \quad (k = 0,1,\ldots,7) \qquad (2)$$

и отберем из них те, для которых

$$0 \le x + D_{0,k} \le n - 1 \quad \text{и} \quad 0 \le y + D_{1,k} \le 0 \ (\text{k} = 0, 1, \ldots, 7) \qquad (3)$$

Именно эти поля и составляют множество $M(x, y)$. Каждому элементу $(a, b) \in M(x,y)$ припишем номер K столбца D, элементы которого в соответствии с (2) и (3) задают приращения по осям для перемещений коня из (x, y) в (a, b). Таким образом, возможные ходы коня из (x, y) упорядочены по полученным ими номерам. На рис.1 эти номера проставлены в соответствующих клетках. Заметим, что иные варианты матрицы D, а всего их 8! = 40320, дают иные способы упорядочивания $M(x, y)$.

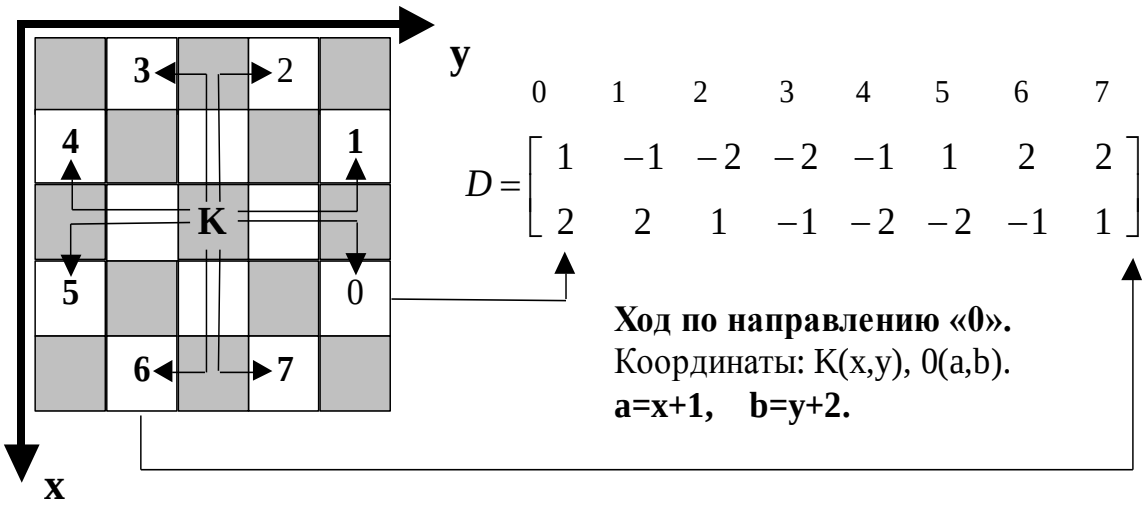


Рис.1. Схема упорядочивания множества возможных ходов коня.

Решая поставленную задачу с помощью общей схемы 3 перебора с возвратом, мы должны последовательно формировать вектор длины n. Нам удобней строить не вектор, а матрицу размера $n \times n$, отмечая в её клетках номера ходов коня от единицы до n^2. В приложении приведено определение функций *HTour* и *HMain* на языке C++, реализующих данный алгоритм.

В каждом рекурсивном вызове глубины j делается попытка поместить коня в очередное j-е поле доски. Позицию, из которой коня продвинуть дальше нельзя, назовем тупиком. Попадание в тупик приводит к завершению текущего рекурсивного вызова, то есть к возврату к предыдущему полю и продолжению работы с ним. Иных случаев завершения рекурсивных вызовов не существует.

Поэтому базой рекурсии нужно считать совокупность всех тупиков. Элементы базы заранее, до вычислений, неизвестны. Если в очередном рекурсивном вызове продвинуть коня удается, то осуществляется переход к следующему рекурсивному вызову. Если удается продвинуть коня на последнюю из незанятых клеток n, то осуществляются последовательные возвраты на первый рекурсивный уровень ($j = 0$) и возвращается решение. Если осуществлен возврат на первый рекурсивный уровень при незаполненной доске, и на этом уровне уже испытаны все допустимые ходы, то выдается сообщение об отсутствии решений.

Следствие. Попытки решать задачу при $n > 8$ с помощью функций программы *Horse.cpp* наталкиваются на сложности, связанные с недостатком памяти или продолжительностью вычислений. Поэтому для решения задачи даже для обычной шахматной доски ($n = 8$) требуются какие-либо иные подходы.

Правило Варнсдорфа

В 1823 году Варнсдорф в брошюре «Простейшее и наиболее общее решение задачи о ходе коня» предложил следующее правило обхода доски размером 8×8.

На каждом ходу ставь коня на такое поле, из которого можно совершить наименьшее число ходов на еще не пройденные поля. Если таких полей несколько, разрешается выбирать любое из них.

Долгое время о его справедливости не было известно каких-либо конкретных утверждений. Опровержение правила Варнсдорфа приведено в [6, с. 28, 29]. Иными словами, с какого бы поля конь ни начал движение, следуя правилу Варнсдорфа, его можно завести в тупик до полного обхода доски. Добавлением дополнительных ограничений к правилу Варнсдорфа можно получать те или иные алгоритмы перемещения коня по доске (возможно, и не приводящие к её обходу). Делать это можно фиксированием конкретного вида матрицы приращений D, используемой для получения кандидатов на очередной ход, и правила однозначного выбора хода в неопределенных ситуациях. Как мы уже отмечали, зафиксировать D можно 40320 способами. На рис. 2 указан один из них. При наличии нескольких допустимых ходов условимся выбирать тот из них, который сформирован с помощью самого левого столбца D. При фиксированной матрице D соответствующий алгоритм перемещения коня назовем *D-алгоритмом*.

Обход доски конем D-алгоритмом

Постановка задачи. Пусть D зафиксировано, и на шахматной доске размера $n \times n$ в позиции (x, y) находится конь. Составить рекурсивную

программу-функцию, находящую D-алгоритмом обход доски конем, если он существует. При отсутствии обхода должна быть выдана «тупиковая позиция» – матрица-доска с уже зафиксированным неполным обходом.

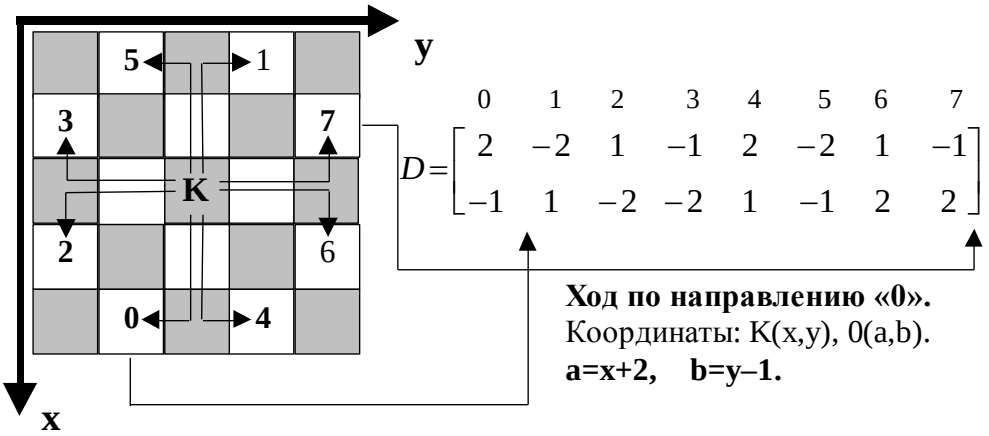


Рис. 2. Схема предпочтения ходов в неопределенных ситуациях

Решение. Функции *ClockW* и *Vans* решают поставленную задачу (см. Примеры
программ).

3. Примеры программ

Обход шахматной доски конём:

```
#define N 8
typedef int Matrix[N][N];
static int D[2][8]={{1,-1,-2,-2,-1,1,2,2},
                    {2,2,1,-1,-2,-2,-1,1}};

int HTour (int n, int x, int y, Matrix &H, int boo, int j)
{
   int k=0, a, b;
   while ( (k<=7)&&(boo==0) )
        { a=x+D[0][k];  b=y+D[1][k];
          if ( ( (a>=0)&&(a<=(n-1) ) )&&
             ( (b>=0)&&(b<=(n-1) ) )&&
             (H[a][b]==0) )
             {
               H[a][b]=j;
               boo=1;
               if (j<n*n) { boo=HTour (n,a,b,H,0,j+1);
                 if (boo == 0) H[a][b]=0; }
             }
          k=k+1;
        }
   return boo;
}
```

```
int HMain (int n, int x, int y, Matrix &M)
{
 int boo;  Matrix H;
 for (int i=0;i<N;i++)
    for (int j=0;j<N;j++) H[i][j]=0;
 H[x][y]=1;
 boo=HTour (N,x,y,H,0,2);
 for (int i=0;i<n;i++)
    for (int j=0; j<n; j++) M[i][j]=H[i][j];
 return boo;
}

void Vans(int n, int x, int y, Matrix &H, int j)
{ int k, a, b, A, B, numk, nums, s, boo;
   int u, v;
   numk=8;
  for (k=0;k<=7;k++)
     { a=x+D[0][k]; b=y+D[1][k]; nums=0;
       if (((a>=0)&&(a<=(n-1)))&&( (b>=0)&&(b<=(n-1)))&&
            (H[a][b]==0) )
       { for (s=0;s<=7;s++)
            { u=a+D[0][s]; v=b+D[1][s];
              boo=( (u>=0)&&(u<=(n-1)) )&&
              ( (v>=0)&&(v<=(n-1)) )&&
              (H[u][v]==0);
              if (boo) nums++;
            }
         if (nums<numk) { numk=nums; A=a; B=b; }
       }
     }
  if (numk<8) { H[A][B]=j; Vans(n, A, B, H, j+1); }
}

void ClockW(int n, int x, int y, Matrix &M)
{ Matrix H;
 for (int i=0;i<N;i++)
    for (int j=0; j<N; j++) H[i][j]=0;
 H[x][y]=1;
 Vans(N, x, y, H, 2);
 for (i=0;i<n;i++)
    for (int j=0;j<n;j++) M[i][j]=H[i][j];
}
```

Нерекурсивный вариант схемы перебора с возвратом для нахождения всех решений (Схема 1):

```
begin (* Нерекурсивный вариант схемы перебора с возвратом.*)
      (* Нахождение всех решений.                          *)
  j := 0;
  while j ≥ 0 do
    if просмотрены не все элементы M_j then
      begin
```

```
        w := очередной непросмотренный элемент Mj ;
        Элемент w объявляется просмотренным ;
        if G(v0, v1, …, vj - 1, w) = true then (* если частичное
                                                  решение *)
         if (v0, v1, …, vj - 1, w)^T – полное решение задачи then
                   write ( (v0, v1, …, vj - 1, w)^T );
         else
          begin
               vj := w;
               if  j < n – 1  then  j := j + 1;
             end
      end
    else
      j:=j-1(* Возврат к более короткому частичному решению *)
            (* Все элементы Mj становятся непросмотренными *)
end
```

Рекурсивный вариант схемы перебора с возвратом для нахождения всех решений (Схема 2):

```
procedure  backtracking ( j );
(* Рекурсивный вариант схемы перебора с возвратом *)
(* Нахождение всех решений расширением (v0, v1, …, vj - 1)^T *)
(* Массив v – глобальный *)
begin
  for  w ∈ Mj  do
    if G(v0, v1, …, vj - 1, w) = true then
    begin
      vj := w;
      if v ≡ (v0,v1,…,vj - 1, w)^T–решение задачи then write (v);
      else  if j < n – 1 then backtracking ( j + 1);
    end
end
```

Рекурсивный вариант схемы перебора с возвратом для нахождения одного решения (Схема 3):

```
procedure  backtracking ( j,  pri );
(* Рекурсивный вариант схемы перебора  с возвратом.            *)
(* Нахождение одного решения расширением (v0, v1, …, vj - 1)^T .  *)
(* Массив v – глобальный.                                      *)
       begin
          while ( просмотрены не все элементы Mj )  and
               ( pri = 'решение не найдено')    do
             if   G(v0, v1, …, vj - 1, w) = true   then
                begin
                  vj := w;
                  if v ≡ (v0, v1, …, vj - 1, w)^T – решение задачи
                   then
                     begin
```

```
                    write (v);
                    pri := 'решение найдено';
                  end
                else
                  if j<n-1 then backtracking (j+1, pri);
              end
  end
```

4. Варианты заданий

1. Пусть на шахматной доске обход конем уже начат, и конь уже побывал на нескольких клетках. Определить, можно ли завершить этот обход. Если да, то указать последовательность ходов.
2. Обойти шахматную доску конем, побывав в каждой клетке не более одного раза, и вернуться в исходную клетку.
3. Задан изограф. Надо обойти все вершины графа, в каждой вершине побывав не более одного раза. Если это невозможно, то вывести соответствующее сообщение. Граф задается двоичной матрицей смежности. Матрица считывается из файла. Первая строка файла – количество N вершин графа, далее указывается сама матрица N×N.
4. Условие предыдущей задачи. Пусть на графе обход уже начат. Определить, можно ли завершить этот обход. Если да, то указать последовательность обхода.
5. «Лабиринт». Лабиринт задается двумерным массивом. Необходимо найти выход из начальной клетки. Элементы, по которым можно пройти, обозначаются «1», а по которым нельзя проходить – «0». Исходная позиция обозначается цифрой «2». Если проход возможен, то координаты каждого шага вывести на экран. Если пройти нельзя, то выдать соответствующее сообщение. Проход по диагонали невозможен.
6. Построить латинский квадрат.
7. «Рюкзак». Вместимость рюкзака задана как максимальная масса вещей, которые можно в него положить (в кг). Также заданы различные вещи, для каждой из которых известна их масса и ценность. Необходимо узнать, какова максимально возможная ценность предметов, которые можно уложить в этот рюкзак.
8. Из состава участников конференции, на которой присутствует N человек надо избрать делегацию, состоящую из M человек. Сколькими способами это можно сделать?
9. Какое максимальное число коней можно расставить на шахматной доске, при условии, что они не будут бить друг друга?
10. Вывести на экран все возможные способы расстановки белых фигур (два коня, два слона, две ладьи, ферзь и король) на первой линии шахматной доски (без соблюдения обычных правил расстановки).

11. Вывести на экран все возможные способы расстановки 8 ферзей на шахматной доске, при условии, чтобы они друг друга не били (без учета симметрии).
12. Вывести на экран все возможные способы расстановки 8 ферзей на шахматной доске, при условии, чтобы они друг друга не били (с учетом симметрии).
13. Вывести на экран все возможные способы расстановки 5 ферзей на кубической шахматной доске размером 5×5×5, при условии, чтобы они друг друга не били (без учета симметрии). Считать, что ферзи в пространстве бьют не только по вертикалям и диагоналям в каждой плоскости, но и вдоль диагоналей куба.
14. Вывести на экран все возможные способы расстановки 5 ферзей на кубической шахматной доске размером 5×5×5, при условии, чтобы они друг друга не били (без учета симметрии). Считать, что ферзи в пространстве бьют не только по вертикалям и диагоналям в каждой плоскости, но и вдоль диагоналей куба.
15. Задан изограф. Надо найти самую длинную цепочку (путь) в графе между какими-либо двумя вершинами. В цепочке не должно быть повторяющихся вершин. Граф задается двоичной матрицей смежности. Матрица считывается из файла. Первая строка файла — количество N вершин графа, далее указывается сама матрица N×N.
16. Дан массив из n целых положительных чисел a[1]…a[n] и число s. Требуется узнать, может ли число s быть представлено как сумма некоторых из чисел массива a. (Каждое число можно использовать не более чем по одному разу.)
17. Перечислить все последовательности из *n* нулей, единиц и двоек, в которых никакая группа цифр не повторяется два раза подряд (нет участка вида *XX*).
18. Перечислить все последовательности из *n* нулей, единиц и двоек, в которых никакая группа цифр не повторяется три раза подряд (нет участка вида *XXX*).
19. Сколько существует вариантов расстановки пяти ферзей на шахматной доске так, чтобы они держали под контролем все поля доски?
20. Сколько существует вариантов расстановки пяти ферзей на шахматной доске размером 11×11 так, чтобы они держали под контролем все поля доски?

5. Указания по составлению отчета

Отчет должен содержать постановку и описание задачи, алгоритм решения задачи, текст составленной программы на языке программирования C/C++, краткие выводы.

Лабораторная работа №14 Линейные списки

1. Цель и задачи работы

Ознакомиться с понятием линейных списков, основными видами линейных списков, примерами их применения. Научиться описывать и использовать линейные списки. Написать программу, реализующую работу со списками.

2. Теоретические сведения

Понятие линейного списка

Программы обычно оперируют информацией, которая содержится в таблицах. В большинстве случаев эти таблицы представляют собой не просто бесформенную массу численных значений, а элементы данных с важными *структурными взаимосвязями*.

В простейшем виде таблица может выглядеть так, как простой *линейный список* элементов, а сведения о ее структурных свойствах можно получить, ответив на следующие вопросы. Какой элемент располагается в списке первым? Какой последним? Какие элементы предшествуют данному элементу, а какие следуют за ним? Сколько элементов содержится в списке? Ответив на такие вопросы, даже в этом простом случае можно получить богатую информацию о структуре данных.

Линейный список представляет собой последовательность $n \geq 0$ узлов X[1], X[2], ..., X [n], структурной особенностью которой является такое расположение элементов списка один относительно другого, как будто они находятся на одной линии. Иначе говоря, в такой структуре должно соблюдаться следующее условие: если $n>0$ и X[1] является первым узлом, а X[n] — последним, то k-й узел X[k] следует за X [k–1] и предшествует узлу X [k+1] для всех $1<k<n$.

С линейными списками могут выполняться следующие операции.

1) Получение доступа к k-му узлу списка для проверки и/или изменения содержимого его полей.

2) Вставка нового узла сразу после или до k-го узла.

3) Удаление k-го узла.

4) Объединение в одном списке двух (или более) линейных списков.

5) Разбиение линейного списка на два (или более) списка.

6) Создание копии линейного списка.

7) Определение количества узлов в списке.

8) Сортировка узлов в порядке возрастания значений в определенных полях этих узлов.

9) Поиск узла с заданным значением в некотором поле.

В операциях (1)-(3) очень важны особые случаи, когда $k=1$ и $k=n$, поскольку доступ к первому и последнему элементам линейного списка можно организовать гораздо проще, чем к любому другому элементу списка.

Виды линейных списков

В одном компьютерном приложении редко используются сразу все девять типов операций в общей их формулировке. Поэтому линейные списки могут иметь самые разные представления в зависимости от класса операций, которые наиболее часто должны с ними выполняться. Достаточно трудно создать единое представление линейных списков, при котором эффективно выполнялись бы все эти операции. Например, сравнительно сложно организовать доступ к k-му узлу длинного списка для произвольно выбранного k, если одновременно необходимо выполнять вставку и удаление элементов в середине списка. Поэтому нужно различать разные типы линейных списков в зависимости от выполняемых с ними основных операций так же, как следует различать типы памяти компьютера, которые определяются ее конкретным назначением.

Линейные списки, в которых операции вставки, удаления и доступа к значениям чаще всего выполняются в первом или последнем узле, получили следующие специальные названия.

Стек — это линейный список, в котором все операции вставки и удаления (и, как правило, операции доступа к данным) выполняются только на одном из концов списка (рис. 1, (а)).

Очередь или *односторонняя очередь* — это линейный список, в котором все операции вставки выполняются на одном из концов списка, а все операции удаления (и, как правило, операции доступа к данным) — на другом (рис. 1, (б)).

Дек или *двусторонняя очередь* — это линейный список, в котором все операции вставки и удаления (и, как правило, операции доступа к данным) выполняются на обоих концах списка (рис. 1, (с)).

Дек, следовательно, является более общим вариантом стека или очереди. Следует различать деки с *ограниченным выводом* и с *ограниченным вводом,* в которых операции удаления и вставки элементов соответственно выполняются только на одном из концов.

При выполнении операции удаления в стеке прежде всех удаляется «младший» объект списка, т. е. объект, который был вставлен в список позже других. В очереди удаление выполняется наоборот: раньше всех удаляется «старший» объект, поскольку узлы покидают очередь в том же порядке, в котором они вставлялись.

Многие исследователи независимо пришли к выводу о важности стеков и очередей, а потому присвоили им иные собственные имена. Так,

стеки часто называют магазинными списками, реверсивными хранилищами, магазинами, вложенными хранилищами, кучами, дисциплинами обслуживания в обратном порядке (LIFO) и даже флюгерными списками. Очереди часто называют циклическими хранилищами или дисциплинами обслуживания в порядке поступления (FIFO). Еще один термин, «стеллаж», применялся для деков с ограниченным выводом. Аналогично деки с ограниченным вводом назывались также свитками. Такое разнообразие имен само по себе интересно уже тем, что оно свидетельствует о важности этих концепций.

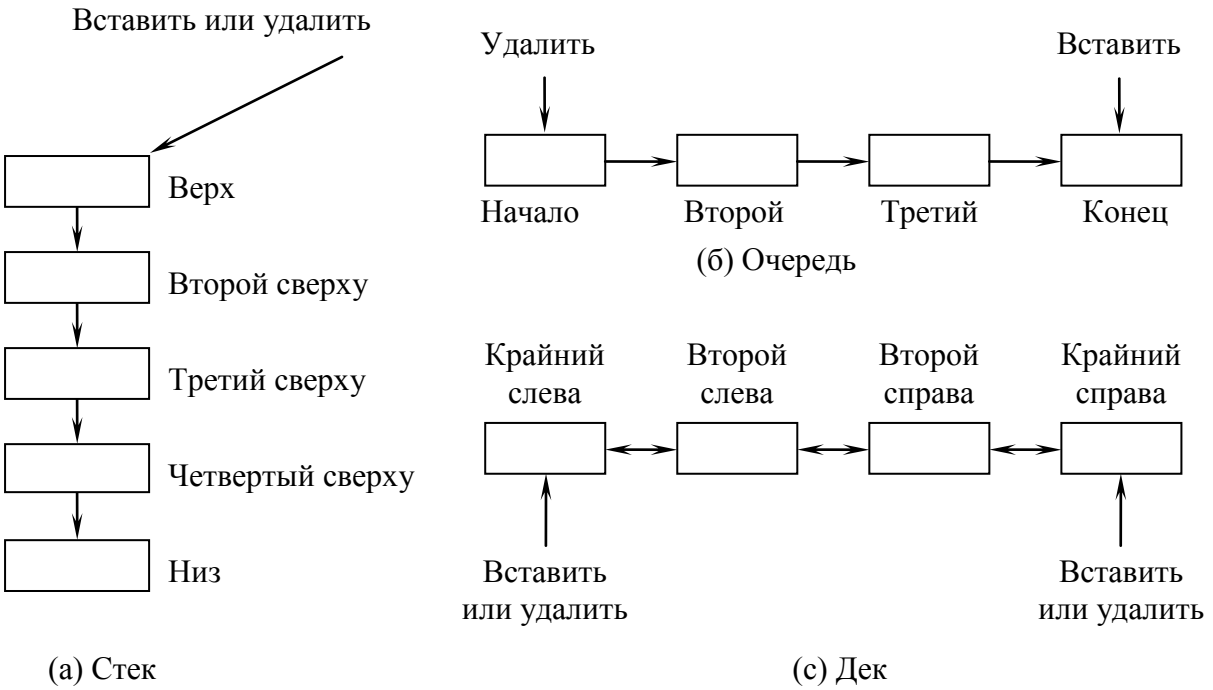


Рис. 1. Три наиболее важных класса линейных списков.

Для описания алгоритмов обработки этих структур обычно используется специальная терминология. Например, объект кладут на *верх* стека или снимают верхний элемент стека. Доступ к элементу, находящемуся *внизу* стека, наиболее затруднителен, и его нельзя удалить до тех пор, пока не будут удалены все остальные элементы стека. (Часто говорят, что объект *проталкивается вниз* стека и *выталкивается* на *верх* стека при удалении самого верхнего элемента. На самом деле физически ничто никуда не проталкивается, а объекты всего лишь добавляются сверху стека). По отношению к очередям применяют понятия *начало* и *конец* очереди. Объекты вставляются в конце очереди и проталкиваются по ней до тех пор, пока не достигнут начала очереди. При работе с деками используют понятия *левый* и *правый* концы. Концепции верха, низа, начала

и конца иногда применяют по отношению к декам, которые используются в качестве стеков или очередей, но нет никаких стандартных соглашений в отношении того конца, с которого они должны располагаться: с правого или левого.

При работе со стеками и очередями удобно использовать некоторые дополнительные обозначения. Например, обозначение

$$A \Leftarrow x$$

указывает, что элемент *x вставлен* сверху стека *A* (если *A* — это стек) или элемент *x вставлен* в конце очереди (если *A* — это очередь). Аналогично

$$x \Leftarrow A$$

используется для обозначения того, что переменная *x* приравнивается к значению верхнего элемента стека *A* или начального элемента очереди *A*, т. е. это значение, которое *удаляется* из *A*. Это обозначение не имеет смысла, если *A* пусто, т. е. когда *A* не содержит значений.

Способы распределения элементов в линейном списке

Последовательное распределение

Наиболее простой и естественный способ хранения линейного списка в памяти компьютера заключается в расположении элементов в последовательных ячейках, при котором один узел списка следует сразу же за другим. Тогда

$$\mathrm{LOC}(\,X[j+1]\,) = \mathrm{LOC}(\,X[j]\,) + c,$$

где *c* – количество слов в одном узле (Обычно c = 1). Вообще,

$$\mathrm{LOC}(\,X[j]\,) = L_0 + cj,$$

где L_0 — константа, которая называется *базовым адресом,* т. е. является адресом некоего узла X [0].

Последовательное распределение очень удобно при работе со *стеком.* Для этого достаточно иметь переменную T, которая называется *указателем стека.* Если стек пуст, то T=0. Для ввода нового элемента Y в стек следует выполнить такие действия:

$$T \leftarrow T+1; \quad X[T] \leftarrow Y.$$

А если стек не пуст, можем положить Y равным верхнему узлу и удалить этот узел, выполнив обратные действия:

$$Y \leftarrow X[T]; \quad T \leftarrow T-1.$$

Представление *очереди* или *дека* организовано несколько сложнее. Очевидным решением могло бы быть применение двух указателей, F и R (для начала и конца очереди), где F=R=0, если очередь пуста. Тогда

вставка элемента с конца очереди может быть выполнена с помощью следующих действий:

$$R \leftarrow R+1; \quad X[R] \leftarrow Y.$$

Удаление же узла в начале очереди (F указывает на ячейку, расположенную перед начальным элементом очереди) может быть выполнено с помощью других действий:

$$F \leftarrow F+1; \quad Y \leftarrow X[F]; \quad \text{если } F = R, \text{ то } F \leftarrow R \leftarrow 0.$$

Следует отметить, что, если при такой организации работы очереди R всегда опережает F (т. е. в очереди всегда имеется хотя бы один узел), элементы таблицы X[1], X[2], ..., X[1000], ... будут последовательно заниматься вплоть до бесконечности, что приведет к расточительному использованию памяти. Следовательно, простой метод на основе вышеупомянутых операций нужно использовать только тогда, когда известно, что F настигает R достаточно регулярно, например, если все удаления выполняются скачкообразно с полным опустошением очереди.

Чтобы разрешить задачу переполнения памяти при такой организации очереди, можно выделить M узлов X [1] ,..., X [M], неявно образующих замкнутое кольцо, в котором за X[M] следует X[1]. Тогда описанные выше действия будут выглядеть так:

если R=M, то $R \leftarrow 1$, в противном случае $R \leftarrow R+1$; $X[R] \leftarrow Y$;
если F=M, то $F \leftarrow 1$, в противном случае $F \leftarrow F+1$; $Y \leftarrow X[F]$.

Очевидно, при использовании метода на основе этих действий предполагается, что в очереди может быть не более M узлов.

Довольно часто на практике используются программы с несколькими стеками, причем размер каждого из них может динамически изменяться. В таком случае не следует накладывать какое-либо ограничение на максимальный размер каждого стека, поскольку его размер непредсказуем. Даже если для каждого стека будет определен его максимальный размер, вряд ли возникнет ситуация, когда одновременно переполнятся все стеки.

Сосуществование всего двух списков с изменяемой длиной можно довольно просто организовать за счет роста их навстречу друг другу, как показано на рис. 2.

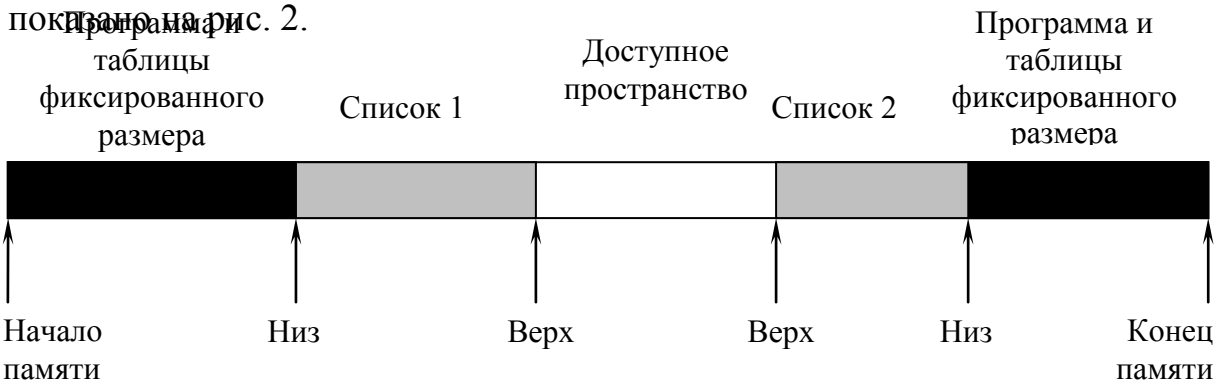


Рис. 2. Организация двух списков навстречу друг другу.

Связанное распределение

Вместо того чтобы хранить линейный список в последовательных ячейках памяти, можно использовать более гибкую схему, в соответствии с которой каждый узел содержит связь со следующим узлом списка.

В программе со связанным распределением необходимо использовать переменную или константу связи, которая будет указывать на адрес первого элемента, причем все другие элементы списка могут быть найдены с помощью этого адреса.

На рисунке 3 переменной связи, указывающей на первый узел списка, является FIRST.

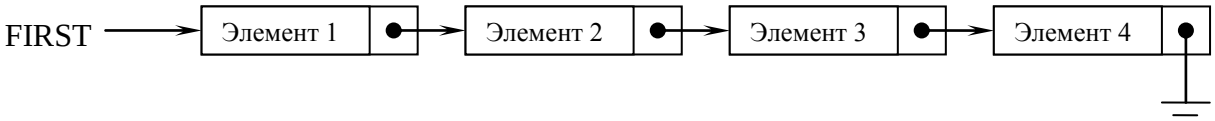


Рис. 3. Связанное распределение.

При организации связанного распределения в памяти потребуется выделить дополнительное пространство для размещения связей. В некоторых ситуациях этот фактор может оказаться доминирующим. Но часто бывает так, что хранимая в узле информация все равно не занимает все слово целиком, поэтому место для связи всегда найдется. Кроме того, во многих случаях несколько элементов комбинируются в одном узле, а потому связь может использоваться сразу для нескольких элементов данных. Однако важнее то, что с помощью связанного распределения памяти *выигрыш* можно получить неявно за счет частичного перекрытия таблиц, совместно использующих некоторые области памяти. Часто последовательное распределение не так рационально, как связанное, поскольку для его эффективной работы необходимо очень большое количество дополнительных вакантных ячеек памяти.

Операция удаления элемента в связанном списке выполняется проще. Например, для удаления элемента 3 необходимо только изменить связь с ним в элементе 2. А при последовательном выделении памяти такое удаление в общем случае подразумевает перемещение значительной части списка в другие ячейки. Также проще выполняется и вставка элемента в середину списка. Например, для вставки элемента 5 между 2 и 3 потребуется изменить только две связи.

С другой стороны, при последовательном выделении памяти обращения к произвольным элементам списка осуществляются гораздо быстрее.

3. Пример программы на языке C/C++

Программа демонстрирует, как можно задать линейный список и его элементы.

```
#include <iostream.h>
#include <conio.h>

struct atom {float *item; atom *prev; atom *next;};
struct list {atom *head; atom *current;};

void main()
{
 int n,i;
 list l;
 atom *a;

 l.head=0;     //Инициализация
 l.current=0; //списка
 cout << "\nВведите количество элементов списка: ";
 cin >> n;
 for (i=0; i<n; i++) //Ввод элементов списка
 {
  a=new atom;
  a->item=new float;
  if (l.head==0) {a->next=0;}
  else
  {
    a->next=l.head;
    l.head->prev=a;
  }
  a->prev=0;
  l.head=l.current=a;
  cout << "Введите " << i+1 << "-й" << " элемент: ";
  cin >> *(l.current->item);
 }
 cout << "Введенный список:\n";
 l.current=l.head;
 for (i=0; i<n; i++) // Вывод списка на экран
 {
   cout << *(l.current->item) << " ";
   l.current=l.current->next;
 }
 for (i=0; i<n; i++) // Удаление списка
 {
   l.current=l.head;
   a=l.current;
   l.head=l.current->next;
   l.head->prev=0;
   a->next=0;
   delete a->item;
   delete a;
```

```
 }
 cout << "\nНажмите любую клавишу.";
 while (!kbhit());
}
```

4. Варианты заданий

1. Даны два упорядоченных по возрастанию дека. Объединить их в третий дек, упорядоченный по убыванию.
2. Построить линейный список из n динамических переменных, содержащих вводимые вещественные числа. Между n/2-й и (n/2)+1-й переменной вставить новую переменную со значением ноль. Вывести полученный список.
3. Даны действительные числа. Построить линейный список из указанных элементов. Вычислить $y = \prod_{i=1}^{n} (x_i + x_{n-i+1})$
4. Построить линейный список из целых чисел. Если среди них имеются совпадающие, то исключить их из последовательности.
5. Указать количество совпадающих чисел в последовательности, организованной как линейный список.
6. Из последовательности символов, состоящей из n элементов и организованной как линейный список, получить последовательность $c_n, c_{n-1}, \ldots, c_1, c_2, \ldots, c_m$. m<n.
7. Последовательность чисел, организованную как линейный список, упорядочить по убыванию.
8. Из последовательности, состоящей из 2n элементов и организованной как линейный список, получить число $\min\{a_1+a_{n+1}, a_2+a_{n+2}, \ldots, a_n+a_{2n}\}$.
9. Вычислить $\max |x_i - \bar{x}|$, где $\bar{x}$ — среднее арифметическое чисел, организованных как линейная последовательность.
10. Даны две последовательности символов. Определить те символы, которые одновременно входят в обе последовательности.
11. Даны действительные числа. Построить линейный список из указанных элементов. Если эта последовательность упорядочена по возрастанию, то оставить ее без изменения. Иначе получить последовательность в обратном порядке.
12. Последовательность чисел, организованную как линейный список, преобразовать, расположив вначале отрицательные члены, а затем неотрицательные. При этом порядок отрицательных остается прежним, а порядок неотрицательных изменяется на обратный.
13. Даны n действительных чисел, организованных как линейный список. Определить новую последовательность $\{x_j - x_n\}_{j=1}^{n-1}$.
14. Даны действительные числа. Построить линейный список из указанных элементов. Вычислить $\sum_{i=1}^{n} x_i x_{n-i+1}$.

15. Даны символы. Построить линейный список из указанных элементов. Вывести те из них, которые входят в эту последовательность по одному разу.
16. Из последовательности $\{a\}_{i=1}^{2n}$, организованной в линейный список, получить число $\max\{\min\{a_1, a_{2n}\}, \min\{a_2, a_{2n-1}\},\ldots, \min\{a_n, a_{n+1}\}\}$.
17. Последовательность чисел, организованную как линейный список, упорядочить по убыванию.
18. Если в последовательности целых чисел, организованной как линейный список, есть хотя бы один член меньше 20, то все отрицательные члены заменять их квадратами, в противоположном случае умножить все члены на 3.
19. Определить индексы членов последовательности, организованной как линейный список, для которых выполняется соотношение
$$x_{n-i+1} < x_i \le x_{n-i}.$$
20. Даны два линейных списка $\{x_i\}_{i=1}^{k}$ и $\{y_i\}_{i=1}^{k}$. Получить новую последовательность $x_1, y_1, x_2, y_2,\ldots, x_k, y_k$.

5. Указания по составлению отчета

Отчет должен содержать постановку и описание задачи, алгоритм решения задачи, текст составленной программы на языке программирования C/C++, краткие выводы.

Лабораторная работа №15 Сильноветвящиеся деревья

1. Цель работы

Целью работы является знакомство с сильноветвящимися деревьями, изучение методов их представления, создания, работы, изменения вниз-направленных сильноветвящихся деревьев. Написание программы, демонстрирующей изученные принципы.

2. Теоретические сведения

Понятие о сильноветвящихся деревьях

На прошлых занятиях были рассмотрены деревья, которые имели всего лишь 2 ветви (бинарные деревья). Теперь рассмотрим деревья, которые будут иметь много ветвей. Такие деревья называются сильноветвящимися.

Сильноветвящиеся деревья могут содержать в своих узлах более чем один ключ.

Разрешим тройные и четверные узлы, которые могут содержать два или три ключа соответственно. У тройного узла есть три выходящие из него ветви. Одна ветвь для всех записей, ключи которых меньше, чем оба его ключа, одна для всех записей, которые больше либо равны первому ключу, но меньше второго, и одна для всех записей, которые больше его ключей или равны второму ключу. Аналогично, 4-ной узел имеет 4 ветви выходящие из него: по одной для каждого интервала определенного его 3 ключами. (Узлы в обычном бинарном дереве можно, таким образом, называть двойными узлами: один ключ, две ветви.)

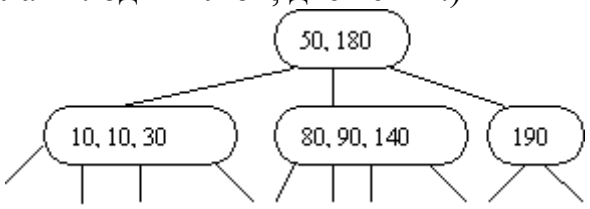


Рис. 1. Пример 2-3-4 дерева

Дерево, содержащее двойные, тройные и четверные узлы называется деревом 2-3-4.

Чтобы вставить новый узел в 2-3-4 дерево нужно произвести поиск и затем «нацепить» наш узел. Легко видно, что делать, если узел, в котором обрывается наш поиск, двойной: просто превращаем его в тройной. Например, число 240 мы могли бы добавить в дерево, изображенное на рисунке 1, просто добавив его (и другую ветвь) к узлу содержащему 190. Аналогично 3-ной узел можно легко превратить в 4-ной. Но как нам быть, если новый узел должен быть вставлен в 4-ной узел? Например, как нам

вставить 70 в дерево рисунка? Одна из возможностей - нацепить 70 как новый самый левый узел узла содержащего 80, 90 и 140. Однако есть лучшее решение этой проблемы: сначала расщепим 4-ной узел в два двойных и передадим один из его ключей его родителями. Так, узел, содержащий 80, 90 и 140, делится на два двойных (один содержащий 80, другой содержащий 140), и его центральный ключ 90 передается в 3-ной узел, содержащий 50 и 180, превращая его в 4-ной узел. После таких манипуляций у нас освобождается место для 70 в узле, содержащем 80.

Рис. 2. Вставка числа 70 в 2-3-4 дерево

Расщепление четверных узлов.

Но что если нам нужно расщепить 4-ной узел, родитель которого тоже является 4-ным узлом? Один из способов - расщепить также и его родителя, и продолжать так до самого верха. Однако более легкий путь гарантирования того, что родитель расщепляемого узла никогда не будет 4-ным узлом - это расщеплять любой четверной узел, попадающийся нам, пока мы просматриваем наше дерево сверху донизу.

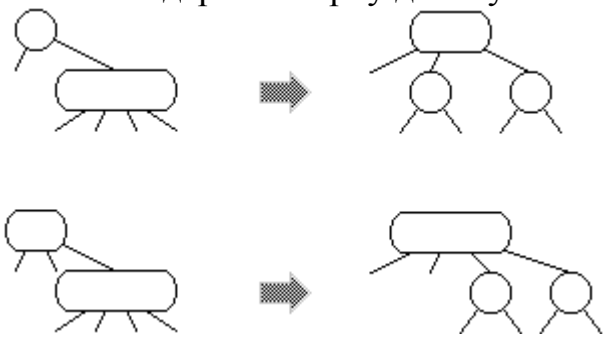


Рис. 3. Расщепление 4-ного узла

Вышеприведенный пример показывает, что мы легко можем вставлять новые узлы, просматривая дерево сверху вниз и расщепляя 4-ные узлы по мере продвижения вниз. В частности, как показано на рисунке, каждый раз как мы наталкиваемся на 2-ной узел, соединенный с 4-ным, мы преобразуем его в 3-ной узел соединенным с двумя 2-ными, и каждый как мы встречаем 3-ной узел, соединенный с 4-ным, мы преобразуем его в 4-ной узел соединенным с двумя 2-ными.

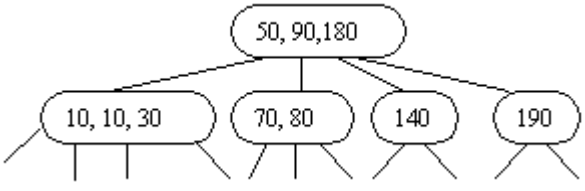


Рис. 4.

Рис. 5.

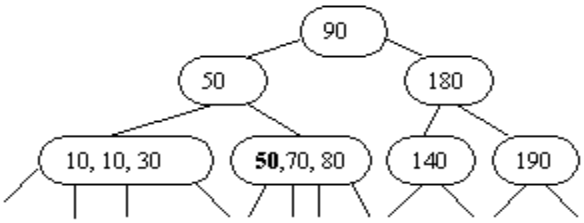


Рис. 6.

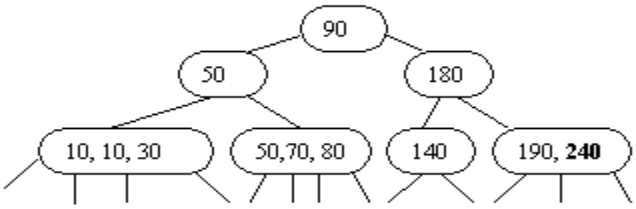


Рис. 7.

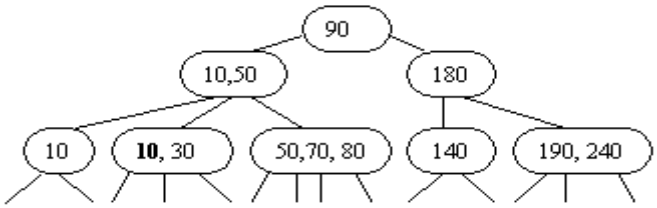


Рис. 8.

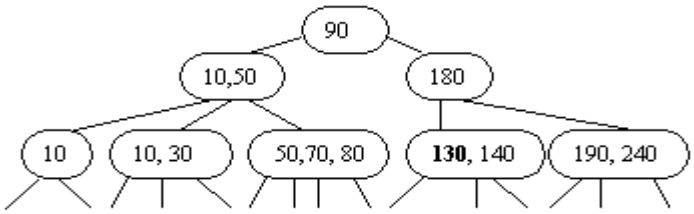


Рис. 9.

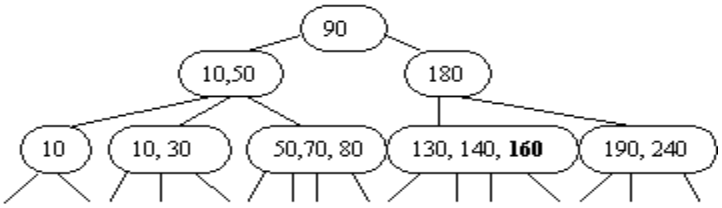


Рис. 10.

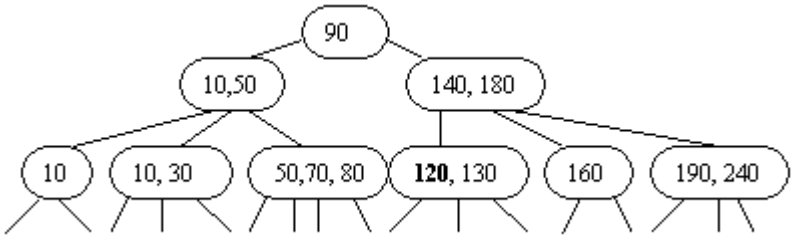


Рис. 11.

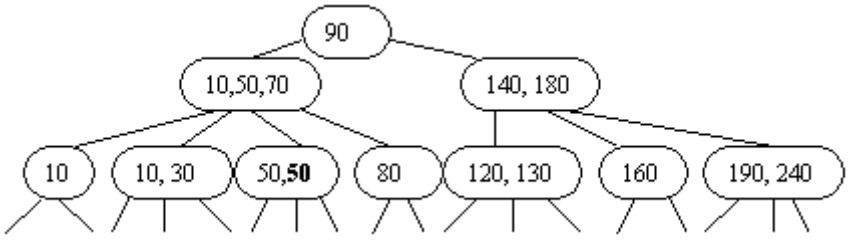


Рис. 4 – 11. Последовательность расщепления 4-ных узлов

Эта операция «расщепления» работает благодаря тому, что мы передвигаем не только ключи, но и указатели. Так 2 двойных узла имеют

тоже количество указателей (четыре), что и один 4-ной узел, благодаря чему расщепление можно выполнить, не переиначивая того, что находится ниже узла расщепления. Тройной узел не может быть преобразован в четверной простым добавлением в него ключа: необходимо также добавить и указатель (в этом случае этот указатель обеспечивается из расщепляемого узла). Критическая точка этого метода - это то, что все трансформации чисто «локальные»: не требуется проверять или изменять никакие части дерева кроме тех, что изображены на рисунке. Каждая из таких трансформаций передает вверх один из ключей 4-ного узла его родителю по дереву и переставляет указатели. Заметьте, что нам нет нужды волноваться, что родитель узла окажется 4-ным узлом так, как трансформация гарантирует, что любой узел, в который мы добрались не 4-ной. В частности, когда мы достигнем дна дерева, мы будем иметь дело в 2-ным или 3-ным узлом, и потому можем вставлять в него напрямую. Как только корень дерева становится 4-ным узлом, мы делим его. Это приводит к тому, что наше дерево «подрастает» на уровень. Только превращение корня дерева в 4-ной узел может заставить наше дерево подрасти на один уровень.

Алгоритм, показанный выше, показывает так же, как производить поиск в 2-3-4 деревьях; вследствие того, что 4-ный узлы расщепляются по мере того, как мы продвигаемся сверху вниз, деревья были названы вниз-направленные 2-3-4 деревья. Что самое интересное во всем этом - это то, что, несмотря на то, что мы вовсе не волновались о том, чтобы сбалансировать это дерево, оно получилось идеально сбалансированным.

При рассмотрении сильноветвящихся деревьев мы ограничимся только рассмотрением деревьев 2-3-4, т.к. любое сильноветвящееся дерево можно свести к дереву 2-3-4.

Представление сильноветвящихся деревьев

В целом ряде задач используются сильноветвящиеся деревья. Каждый элемент для представления бинарного дерева должен содержать как минимум три поля – значение или имя узла, указатель левого поддерева, указатель правого поддерева. Произвольные деревья могут быть бинарными или сильноветвящимися. Причем число потомков различных узлов не ограничено и заранее не известно.

Тем не менее, для представления таких деревьев достаточно иметь элементы, аналогичные элементам списковой структуры бинарного дерева. Элемент такой структуры содержит минимум три поля: значение узла, указатель на начало списка потомков узла, указатель на следующий элемент в списке потомков текущего уровня. Также как и для бинарного дерева необходимо хранить указатель на корень дерева. При этом дерево представлено в виде структуры, связывающей списки потомков различных вершин. Такой способ представления вполне пригоден и для бинарных деревьев.

Представление деревьев с произвольной структурой в виде массивов

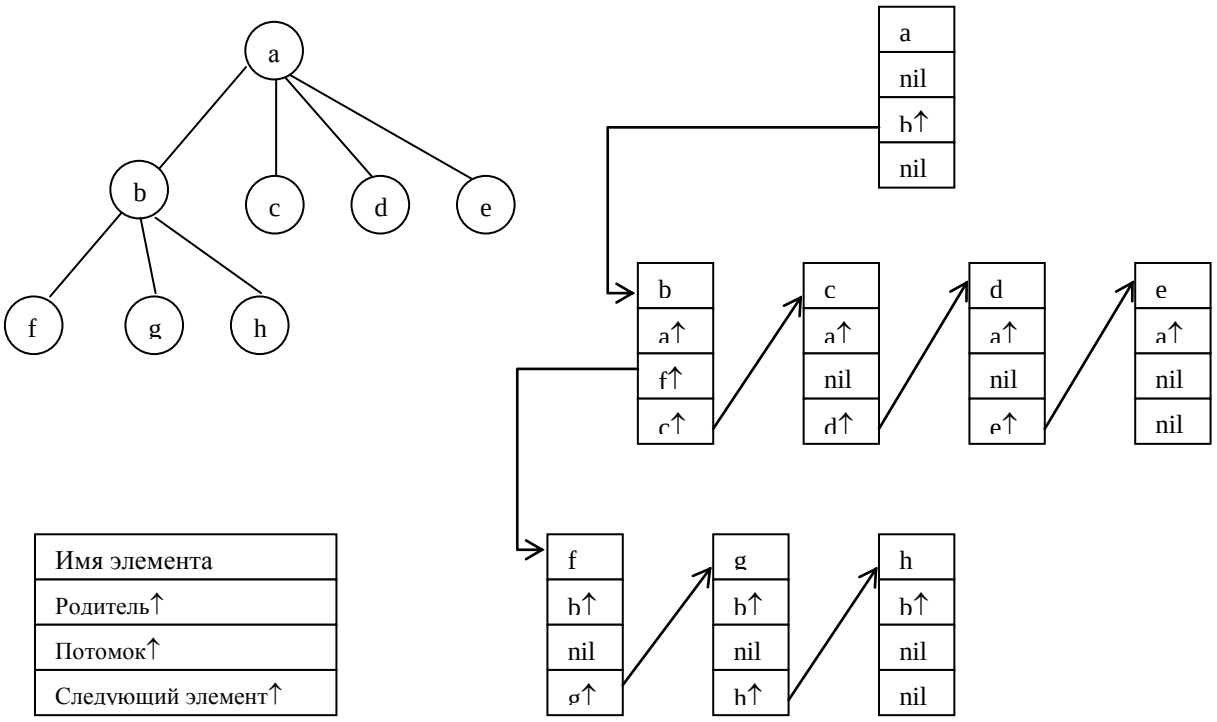


Рис. 12. Представление сильноветвящихся деревьев в виде списков

может быть основано на матричных способах представления графов.

1. Пример программы

Рассмотрим использование таких деревьев для представления иерархической структуры каталогов файловой системы. Во многих файловых системах структура каталогов и файлов, как правило, представляет собой одно или несколько сильноветвящихся деревьев. В файловой системе MS DOS корень дерева соответствует логическому диску. Листья дерева соответствуют файлам и пустым каталогам, а узлы с ненулевой степенью - непустым каталогам.

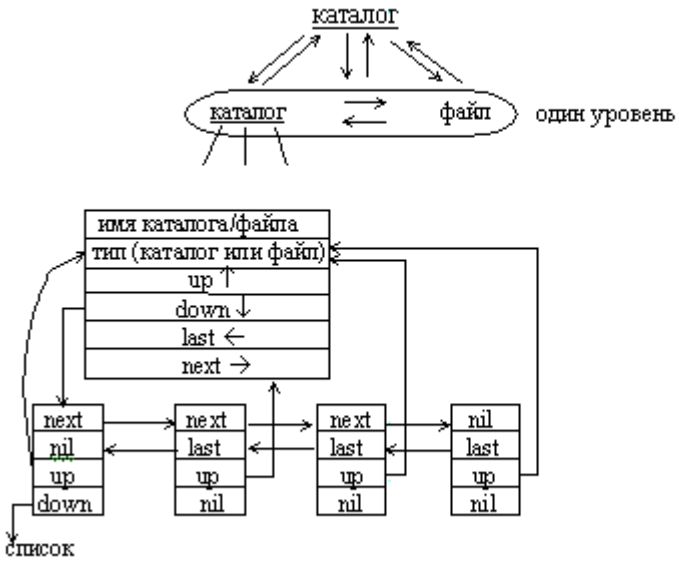


Рис. 13. Представление логической структуры каталогов и файлов в виде сильноветвящегося дерева.

Для представления такой структуры используем расширение спискового представления сильноветвящихся деревьев. Способы представления деревьев, рассмотренные ранее, являются предельно экономичными, но не очень удобными для перемещения по дереву в разных направлениях. Именно такая задача встает при просмотре структуры каталогов. Необходимо осуществлять “навигацию” – перемещаться из текущего каталога в каталог верхнего или нижнего уровня, или от файла к файлу в пределах одного каталога.

Для облегчения этой задачи сделаем списки потомков двунаправленными. Для этого достаточно ввести дополнительный указатель на предыдущий узел ”last”. С целью упрощения перемещения по дереву от листьев к корню введем дополнительный указатель на предок текущего узла “up”. Общими с традиционными способами представления являются указатели на список потомков узла “down” и следующий узел “next”.

Для представления оглавления диска служат поля имя и тип файла/каталога. Рассмотрим программу, которая осуществляет чтение структуры заданного каталога или диска, позволяет осуществлять навигацию и подсчет места занимаемого любым каталогом.

```
program dir_tree;
uses dos;
  type node=record
       name: string[50]; {Имя каталога/файла}
       size: longint; {Размер файла (байт) }
```

```
        node_type: char; {Тип узла (файл –'f' / каталог-'c')}
        up,down: pointer; {Указатели на предка и список потомков}
        last,next: pointer; {Указатели на соседние узлы}
  end;
var
   n,i,l:integer;
   root, current_root: pointer;
   pnt, current:^node;
   s : searchrec;
   str: string;

Procedure create_tree(local_root:pointer);
{Отображение физического оглавления диска в логическую структуру}
var
   local_node, local_r_node, local_last : ^node;

Procedure new_node;
{Создание нового узла в дереве каталогов и файлов}
begin
    new(local_node);
    local_node^.last:=local_last;
        if not(local_last=nil) then local_last^.next:=local_node;
     local_node^.next:=nil;
     local_node^.down:=nil;
     local_node^.up:=local_r_node;
        if local_r_node^.down =nil then local_r_node^.down:=local_node;
     local_node^.name:=local_r_node^.name+'\'+s.name;
        if s.attr and Directory = 0 then local_node^.node_type:='f'
                                    else local_node^.node_type:='c';
     local_node^.size:=s.size;
     local_last:=local_node;
end;
begin {Собственно процедура}
local_r_node:=local_root;
local_last:=nil;
findfirst(local_r_node^.name+'\*.*',anyfile,s);
    if doserror = 0 then
        begin
            if (s.name<>'.') and (s.name<>'..') and (s.attr and VolumeID = 0)
                then new_node;
            while doserror=0 do begin
             findnext(s);
                if (doserror = 0) and (s.name<>'.') and (s.name<>'..') and (s.attr and
        VolumeID = 0) then new_node;
                end
```

```
        end;
  if not (local_r_node^.down=nil) then
     begin
      local_node:=local_r_node^.down;
        repeat
           if local_node^.node_type='c' then create_tree(local_node);{Рекурсия}
           local_node:=local_node^.next
        until local_node=nil
     end
end;

Procedure current_list;
{Вывод оглавления текущего каталога}
begin
   current:=current_root;
   writeln('текущий каталог - ', current^.name);
        if current^.node_type='c' then
           begin
              pnt:=current^.down;
              i:=1;
              repeat {Проходим каталог в дереве}
                     writeln (i:4,'-',pnt^.name);
                      pnt:=pnt^.next;
                      i:=i+1
              until pnt=nil
           end;
end;

Procedure down;
{Навигация в дереве каталогов. Перемещение на один уровень вниз}
begin
   current:=current_root;
      if not (current^.down=nil) then
         begin
            current:= current^.down;
            writeln('номер в оглавлении'); readln; read(l);
             i:=1;
             while (i<l) and not (current^.next=nil) do
                   begin
                      current:=current^.next;
                      i:=i+1
                    end;
             if (current^.node_type='c') and not (current^.down=nil)
             then current_root:= current;
         end;
```

```
end;

Procedure up;
{Навигация в дереве каталогов. Перемещение на один уровень вверх}
begin
    current:=current_root;
     if not (current^.up=nil) then current_root:=current^.up;
end;

Procedure count;
{Расчет числа файлов и подкаталогов иерархической структуры каталога}
var
   n_files, n_cats :integer;
   procedure count_in (local_root : pointer);
     var
        local_node, local_r_node: ^node;
        begin
           local_r_node:=local_root;
           if not (local_r_node^.down=nil) then
                 begin
                    local_node:=local_r_node^.down;
                    repeat
                        if local_node^.node_type='f'
                           then  n_files:=n_files+1
                           else begin
                                    n_cats:=n_cats+1;
                                    count_in (local_node)
                                end;
                           local_node:=local_node^.next
                    until local_node=nil
                 end
        end;
begin {Собственно процедура}
         n_files:=0; n_cats:=0;
         count_in (current_root);
         writeln ('файлы : ',n_files, ' каталоги: ', n_cats);
end;

Procedure count_mem;
{Расчет физического объема иерархической структуры каталога}
var
   mem :longint;
procedure count_m_in (local_root : pointer);
  var
     local_node, local_r_node: ^node;
```

```
   begin
      local_r_node:=local_root;
         if not (local_r_node^.down=nil) then
              begin
                  local_node:=local_r_node^.down;
                  repeat
                      if local_node^.node_type='f'
                         then mem:=mem+local_node^.size
                         else
                            count_m_in (local_node);
                     local_node:=local_node^.next
                  until local_node=nil
              end
   end;
begin {Собственно процедура}
        mem:=0;
        count_m_in (current_root);
        writeln ('mem ', mem, ' bytes');
end;

{----------основная программа----------}
begin
        new(current);
        {Инициализация корня дерева каталогов и указателей для
        навигации}
        root:=current; current_root:=current;
        writeln('каталог ?'); read(str); writeln(str);
        current^.name:=str;
        current^.last:=nil; current^.next:=nil;
        current^.up:=nil; current^.down:=nil;
        current^.node_type:='c';
        {Создание дерева каталогов}
        create_tree(current);
           if current^.down=nil then current^.node_type:=' ';
                 repeat
                { Интерактивная навигация }
                   writeln ('1-список');
                   writeln('2-вниз');
                   writeln('3-вверх');
                   writeln('4-число файлов');
                   writeln('5-объем');
                   readln(n);
                       if n=1 then current_list;
                       if n=2 then down;
                       if n=3 then up;
```

```
                if n=4 then count;
                if n=5 then count_mem;
        until n=0
end.
```

Для чтения оглавления диска в данной программе используются стандартные процедуры findfirst и findnext, которые возвращают сведения о первом и последующих элементах в оглавлении текущего каталога.

В процессе чтения корневого каталога строится первый уровень потомков в списковой структуре дерева. Далее процедура построения поддерева вызывается для каждого узла в корневом каталоге. Затем процесс повторяется для всех непустых каталогов до тех пор, пока не будет построена полная структура оглавления.

Все операции по просмотру содержимого каталогов и подсчету занимаемого объема производятся не с физическими каталогами и файлами, а с созданным динамическим списковым представлением их логической структуры в виде сильноветвящегося дерева.

В данном примере программы для каждого файла или каталога хранится его полное имя в MS-DOS, которое включает имя диска и путь. Если программу несколько усложнить, то можно добиться более эффективного использования динамической памяти. Для этого потребуется хранить в узле дерева только имя каталога или файла, а полое имя – вычислять при помощи цепочки имен каталогов до корневого узла.

4. Варианты заданий

1. Построить 2-3-4 дерево пятого уровня.
2. Написать функцию подсчета узлов в дереве.
3. Построить генеалогическое дерево вашей семьи.
4. Написать функцию создания нового узла.
5. Заданы числа A1, A2 и A3. Определить к какому интервалу относится введенное число N.
6. Написать функцию расщепления 4-го узла на бинарные.
7. Дано выражение {(1-p)*A+B/(C-D)} представить его в виде дерева.
8. Построить дерево второго уровня, состоящее из элементов с четырьмя ветками.
9. Написать функцию подсчета листьев (нулевых узлов) в дереве.
10. Дана последовательность из N целых чисел, вывести эту последовательность на экран в виде 2-3-4 дерева.
11. Написать функцию поиска элемента с заданным ключом.
12. Отобразить структуру управления предприятием в виде сильноветвящегося дерева.
13. Написать функцию удаления узла.
14. Создать модель лабиринта: родительский узел 1-го уровня=вход, узлы последующих уровней=развилки, листья=тупики, листья со спец. признаком=выходы.

15. Написать функцию расщепления 4-го узла на тройные.
16. Создать дерево, используя узлы пятого порядка.
17. Создать функцию прохождения 2-3-4 деревьев.
18. Представить цепную реакцию в виде сильноветвящегося дерева.
19. Написать функцию объединения трех бинарных узлов в два тройных узла.
20. Организовать сортировку N чисел по следующей схеме: 1-вый уровень: число n > ключа k – правая ветвь, n < k – левая ветвь n = k – центральная. Уровни 2 и 3 аналогично.

5. Указания по составлению отчета

Отчет должен содержать постановку и описание задачи, алгоритм решения задачи, текст составленной программы на языке программирования C/C++, краткие выводы.

Лабораторная работа №16 Древовидные структуры

1. Цель работы

Изучить древовидные структуры, такие как деревья, бинарные и сбалансированные деревья. Знать основные понятия и определения древовидных структур (длина внутреннего/внешнего пути, степень узла, узел-потомок, узел-предок, терминальный элемент и т.д.), а также способы изображения древовидных структур. Научиться выполнять операции поиска, включения и удаления узлов, как с бинарными, так и со сбалансированными деревьями. Написать программу, демонстрирующую изученные принципы.

2. Теоретические сведения

Основные понятия и определения

Пусть древовидная структура определяется следующим образом: древовидная структура с базовым типом T – это либо:

1. пустая структура; либо
2. узел типа T, с которым связано конечное число древовидных структур с базовым типом T, называемых поддеревьями.

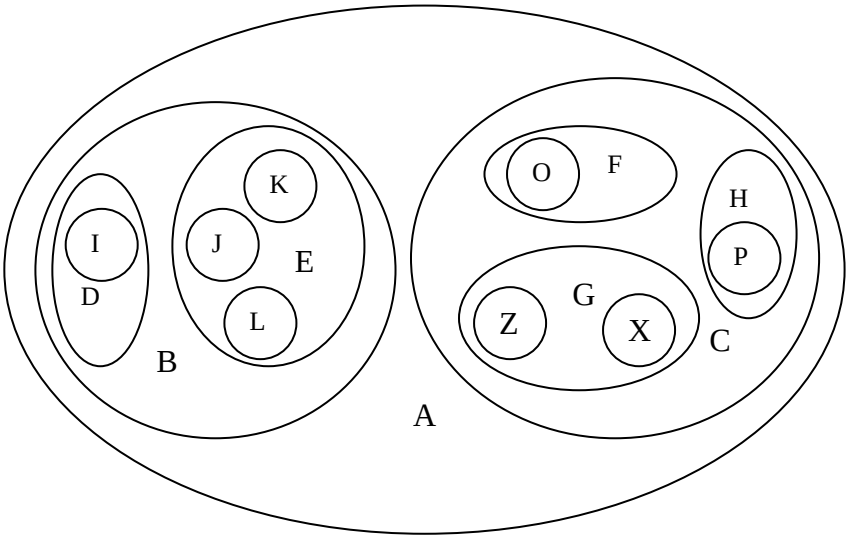
Если сравнивать списки и древовидные структуры, то можно отметить, что список есть древовидная структура, у которой каждый узел имеет не более одного «поддерева». Поэтому список называется также вырожденным деревом.

Существует несколько способов изображения древовидной структуры. Например, пусть базовый тип T есть множество букв; такая древовидная структура разными способами изображена на рис. 1.

Все эти представления демонстрируют одну и ту же структуру и поэтому эквивалентны. В дальнейшем древовидные структуры мы будем называть просто деревьями.

Упорядоченное дерево – это дерево, у которого ветви каждого узла упорядочены. Следовательно, два упорядоченных дерева на рис. 2 – это особые, отличные друг от друга деревья.

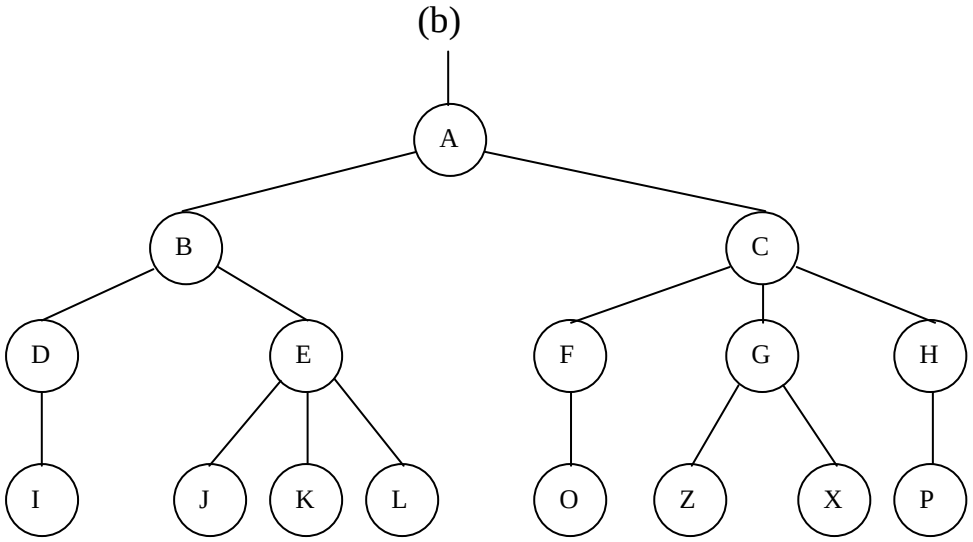
Узел y, который находится непосредственно под узлом x, называется (непосредственным) потомком x; если x находится на уровне i, то говорят, что y – на уровне i+1. Наоборот, узел x называется (непосредственным) предком y.



(a)

(A (B (D (I), E (J, K, L)), C (F (O), G (Z, X), H (P))))

(b)



(c)

Рис. 1. Представления древовидной структуры: (a) вложенные множества; (b) вложенные скобки; (c) граф.

Считается, что корень дерева расположен на уровне 1. Максимальный уровень какого-либо элемента дерева называется его глубиной или высотой.

Рис. 2. Два различных бинарных дерева.

Если элемент не имеет потомков, он называется терминальным элементом или листом, а элемент, не являющийся терминальным, называется внутренним узлом. Число (непосредственных) потомков внутреннего узла называется его степенью. Максимальная степень всех узлов есть степень дерева. Число ветвей, или ребер, которые нужно пройти, чтобы продвинуться от корня к узлу x, называется длиной пути к x. Корень имеет длину пути 1, его непосредственные потомки – длину пути 2 и т. д. Вообще, узел на уровне i имеет длину пути i. Длина пути дерева определяется как сумма длин путей всех его узлов. Она также называется длиной внутреннего пути. Например, длина внутреннего пути дерева, изображенного на рис. 1, равна 52. Очевидно, что средняя длина пути P_I есть

$$P_I = \frac{1}{n}\sum_i n_i \cdot i,$$

где n_i - число узлов на уровне i. Для того чтобы определить, что называется длиной внешнего пути, мы будем дополнять дерево специальным узлом каждый раз, когда в нем встречается нулевое поддерево. При этом мы считаем, что все узлы должны иметь одну и ту же степень – степень дерева. Следовательно, подобное расширение дерева предполагает заполнение пустых ветвей, разумеется, при этом специальные узлы не имеют дальнейших потомков. Дерево на рис. 1, дополненное специальными узлами, показано на рис. 3, где специальные узлы изображены квадратиками.

Длина внешнего пути теперь определяется как сумма длин путей всех специальных узлов. Если число специальных узлов на уровне i есть m_i, то средняя длина внешнего пути P_E равна

$$P_E = \frac{1}{m}\sum_i m_i \cdot i.$$

У дерева, приведенного на рис. 3, длина внешнего пути равна 153.

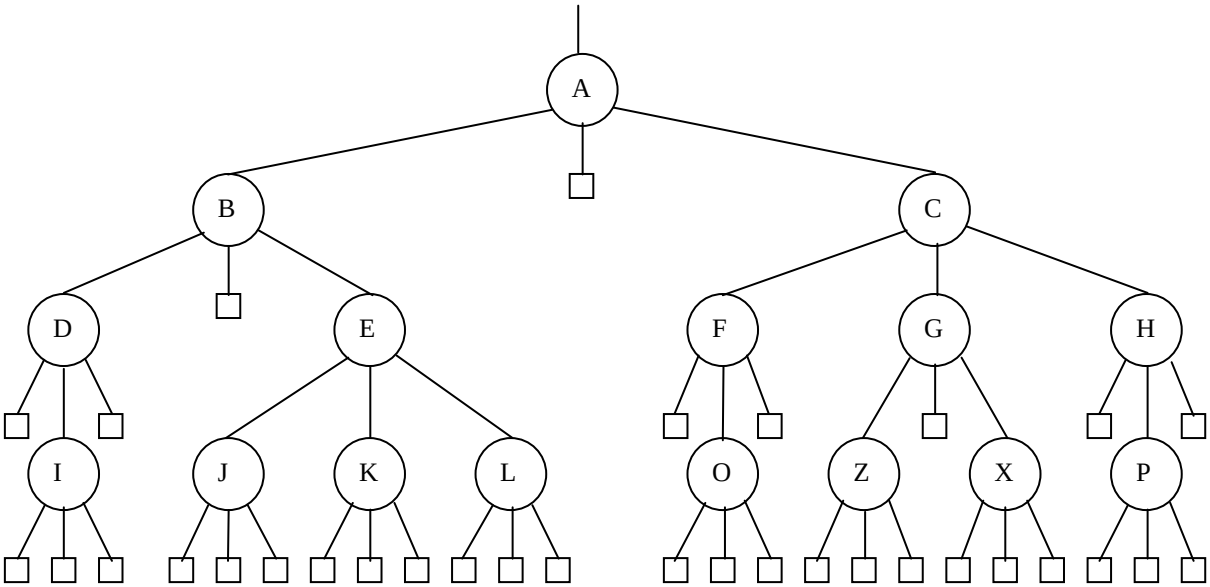


Рис. 3. Тернарное дерево со специальными узлами.

Число специальных узлов m, которые нужно добавить к дереву степени d, непосредственно зависит от числа n исходных узлов. Заметим, что на каждый узел указывает ровно одна ветвь. Следовательно, в расширенном поддереве имеется m+n ветвей. С другой стороны, из каждого исходного узла выходят d ветвей, а из специальных узлов – ни одной. Поэтому всего имеется dn+1 ветвей (1 дает ветвь, указывающую на корень). Из этих двух формул мы получаем следующее равенство между числом m специальных узлов и n исходных узлов: dn+1=m+n, или

$$m=(d-1)n+1.$$

Максимальное число узлов в дереве заданной высоты h достигается в случае, когда все узлы имеют d поддеревьев, кроме узлов уровня h, не имеющих ни одного. Тогда в дереве степени d первый уровень содержит 1 узел (корень), уровень 2 содержит d его потомков, уровень 3 содержит d^2 потомков d узлов уровня 2 и т. д. Это дает следующую величину:

$$N_d(h)=1+d+d^2+\ldots+d^{h-1}=\sum_{i=0}^{h-1}d^i$$

в качестве максимального числа узлов для дерева с высотой h и степенью d. При d=2 мы получаем

$$N_2(h)=\sum_{i=0}^{h-1}2^i=2^h-1.$$

Упорядоченные деревья степени 2 играют особо важную роль. Они называются бинарными деревьями. Мы определяем упорядоченное бинарное дерево как конечное множество элементов (узлов), каждый из которых либо пуст, либо состоит из корня (узла), связанного с двумя

различными бинарными деревьями, называемыми левым и правым поддеревом корня. Далее мы будем рассматривать в основном бинарные деревья. Деревья, имеющие степень больше 2, называются сильно ветвящимися деревьями.

Теперь мы обратимся к проблеме представления деревьев. Ясно, что изображение таких рекурсивных структур (точнее, рекурсивно определенных) с разветвлениями предполагает использование ссылок. Очевидно, что не имеет смысла описывать переменные с фиксированной древовидной структурой, вместо этого узлы определяются как переменные с фиксированной структурой, т. е. фиксированного типа, где степень дерева определяет число компонент-ссылок, указывающих на поддеревья данного узла. Ясно, что ссылка на пустое поддерево обозначается через **nil**. Следовательно, бинарное дерево будет состоять из компонент такого типа:

```
Type node = record
              op: char;
              left, right: ^node
            end.
```

Ясно, что существуют способы представления абстрактной древовидной структуры в терминах других типов данных, например таких, как массив:

```
t: array[1..n] of  record
                     op: char;
                     left, right: integer
                   end.
```

Основные операции с бинарными деревьями

Имеется много задач, которые можно выполнять на древовидной структуре; распространенная задача – выполнение заданной операции P с каждым элементом дерева. Здесь P рассматривается как параметр более общей задачи посещения всех узлов, или, как это обычно называют, обход дерева.

Если рассматривать эту задачу как единый последовательный процесс, то отдельные узлы посещаются в некотором определенном порядке и могут считаться расположенными линейно. В самом деле, описание многих алгоритмов существенно упрощается, если можно говорить о переходе к следующему элементу дерева, имея в виду некоторое упорядочение.

Существуют три принципа упорядочения, которые естественно вытекают из структуры дерева. Так же как и саму древовидную структуру, их удобно выразить с помощью рекурсии. Обращаясь к бинарному дереву

на рис. 4, где R обозначает корень, а A и B – левое и правое поддеревья, мы можем определить такие три упорядочения:
1. Сверху вниз: R, A, B (посетить корень до поддеревьев)
2. Слева направо: A, R, B
3. Снизу вверх: A, B, R (посетить корень после поддеревьев)

Обходя дерево на рис. 5 и выписывая символы, находящиеся в узлах, в том порядке, в котором они встречаются, мы получаем следующие последовательности:
1. Сверху вниз: * + a / b c – d * e f
2. Слева направо: a + b / c * d – e * f
3. Снизу вверх: a b c / + d e f * - *

Можно узнать три формы записей выражений: обход сверху вниз дает префиксную запись, обход снизу вверх – постфиксную запись, а обход слева направо дает привычную инфиксную запись, хотя и без скобок, необходимых для определения порядка выполнения операций

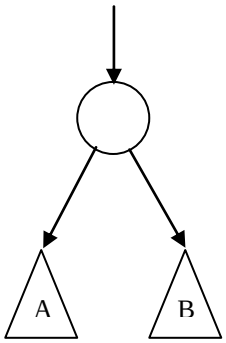


Рис. 4. Бинарное дерево.

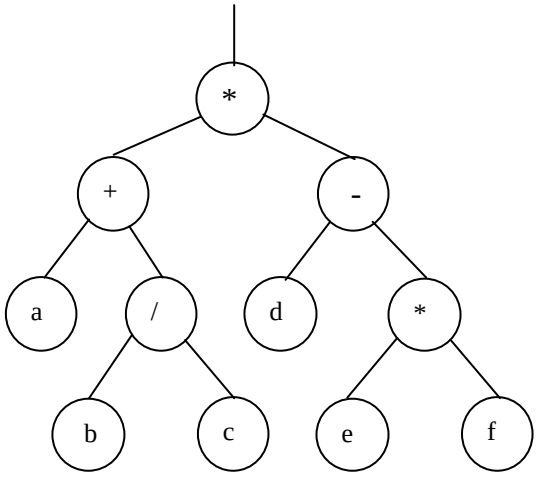


Рис. 5. Выражение (a+b/c)*(d –e*f), представленное в виде дерева.

Бинарные деревья часто используются для представления множеств данных, элементы которых ищутся по уникальному, только им присущему ключу. Если дерево организовано таким образом, что для каждого узла t_i все ключи в левом поддереве меньше ключа t_i, а ключи в правом

поддереве больше ключа t_i, то это дерево называется деревом поиска. В дереве поиска можно найти место каждого ключа, двигаясь, начиная от корня и переходя на левое или правое поддерево каждого узла, в зависимости от значения его ключа.

Если в процессе поиска в дереве не найдено ключа с нужным значением, то в конце дерева можно поместить барьер. Использование ссылок позволяет связать все терминальные узлы дерева с одним и тем же барьером. Полученная структура – это уже не просто дерево, а скорее, дерево, все листья которого прицеплены внизу к одному якорю. Барьер можно также считать общим представлением всех внешних (специальных) узлов, которыми дополняется исходное дерево (см. рис. 3). Процедура поиска принимает значение ссылки на барьер (на **nil**), если в дереве не найдено ключа с нужным значением.

Возможности техники динамического размещения переменных с доступом к ним через ссылки вряд ли полностью проявляются в тех задачах, где построенная структура данных остается неизменной. Более подходящими примерами служат задачи, в которых сама структура дерева изменяется, т.е. дерево растет и/или уменьшается во время выполнения программы. Это также случай, когда другие представления данных, такие, как массив, не подходят и когда дерево с элементами, связанными ссылками, как раз и есть подходящая структура.

Примером растущего, но никогда не убывающего дерева может служить задача построения частотного словаря. В этой задаче задана последовательность слов, и нужно установить число появлений каждого слова. Это означает, что, начиная с пустого дерева, каждое слово ищется в дереве. Если оно найдено, увеличивается его счетчик появлений, если нет – в дерево вставляется новое слово (с начальным значением счетчика, равным 1). Такая задача называется поиском по дереву с включением. Возможно использование следующих типов данных:

```
Type ref = ^word;
            word = record
                  key: integer;
                  count: integer;
                  left, right: ref
                end.
```

Задача, обратная включению, называется удалением из дерева. Удаление из дерева – задача, обычно более сложная, чем включение. Трудность заключается в удалении элементов с двумя потомками, поскольку мы не можем указывать одной ссылкой на два направления. В этом случае удаляемый элемент нужно заменить либо на самый правый элемент его левого поддерева, либо на самый левый элемент его правого поддерева. Ясно, что такие элементы не могут иметь более одного потомка.

Сбалансированные деревья

Сбалансированное дерево – это дерево, у которого для каждого узла высота его двух поддеревьев различается не более чем на 1.

Сбалансированные деревья часто называют АВЛ-деревьями, так как определение сбалансированности было дано Адельсоном-Вельским и Ландисом. Это определение не только простое, но также приводит к легко выполнимой балансировке, а средняя длина поиска остается практически такой же, как у идеально сбалансированного дерева.

Со сбалансированными деревьями можно выполнять следующие операции:

1. Найти узел с данным ключом
2. Включить узел с данным ключом
3. Удалить узел с данным ключом

Включение в сбалансированное дерево

Пусть дан корень r с левым и правым по ддеревьями L и R. Предположим, что в L включается новый узел, вызывая увеличение его высоты на 1. Возможны три случая:

1. $h_L = h_R$: L и R становятся неравной высоты, но критерий сбалансированности не нарушается

2. $h_L < h_R$: L и R приобретают равную высоту, т.е. сбалансированность даже улучшается.

3. $h_L > h_R$: критерий сбалансированности нарушается, и дерево нужно перестраивать.

Алгоритм включения и балансировки полностью определяется способом хранения информации о сбалансированности дерева. Крайнее решение состоит в хранении этой информации полностью неявно в самой структуре дерева. В этом случае показатель сбалансированности узла должен заново вычисляться каждый раз, когда узел затрагивается включением, что приводит к чрезвычайно высоким затратам. Другая крайность − явно хранить показатель сбалансированности в информации, связанной с каждым узлом. Тогда определение типа узла принимает следующий вид:

```
Type node = record
              key : integer;
            count : integer;
                  left, right : ref;
                  bal : -1 .. +1
          end
```

В общем случае процесс включения узла состоит из последовательности таких трех этапов:

1.Следовать по пути поиска, пока не окажется, что ключа нет в дереве.

2.Включить новый узел и определить новый показатель сбалансированности.

3.Пройти обратно по пути поиска и проверить показатель сбалансированности у каждого узла.

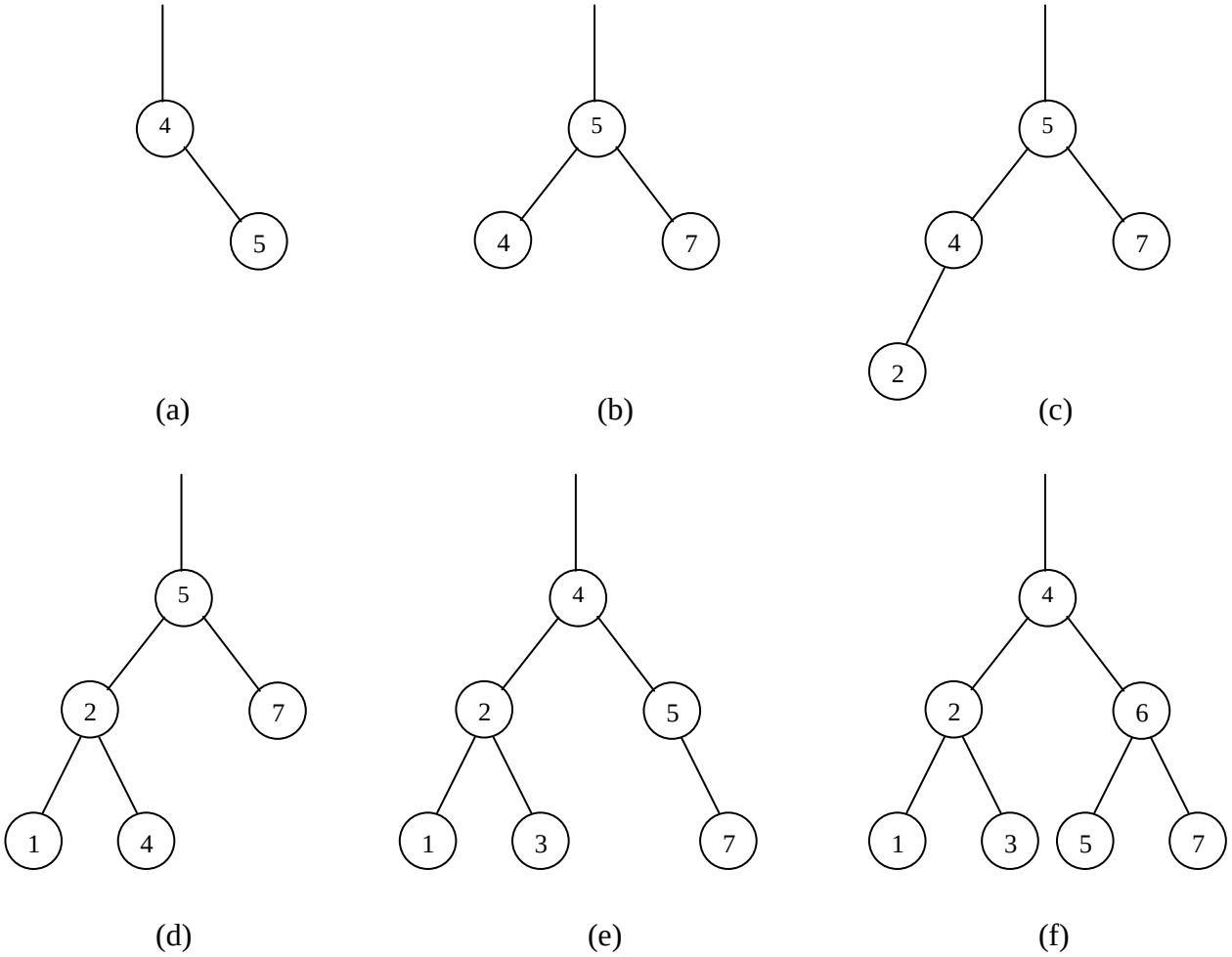


Рис. 6. Включение в сбалансированное дерево

Рассмотрим бинарное дерево (a), которое состоит только из двух узлов. Включение ключа 7 вначале дает несбалансированное дерево. Его балансировка требует однократного правого (RR) поворота, давая в результате идеально сбалансированное дерево (b). Последующее включение узлов 2 и 1 дает несбалансированное поддерево с левым корнем 4.

Это поддерево балансируется однократным левым (LL) поворотом (d). Далее включение ключа 3 сразу нарушает критерий сбалансированности в корневом узле 5. Сбалансированность теперь восстанавливается с помощью более сложного двукратного поворота налево и направо (LR); результатом является дерево (e). Теперь при следующем включении

потерять сбалансированность может лишь узел 5. Включение узла 6 приводит к четвертому виду балансировки – двукратному повороту направо и налево (RL). Окончательное дерево показано на рис. 5.6 (f).

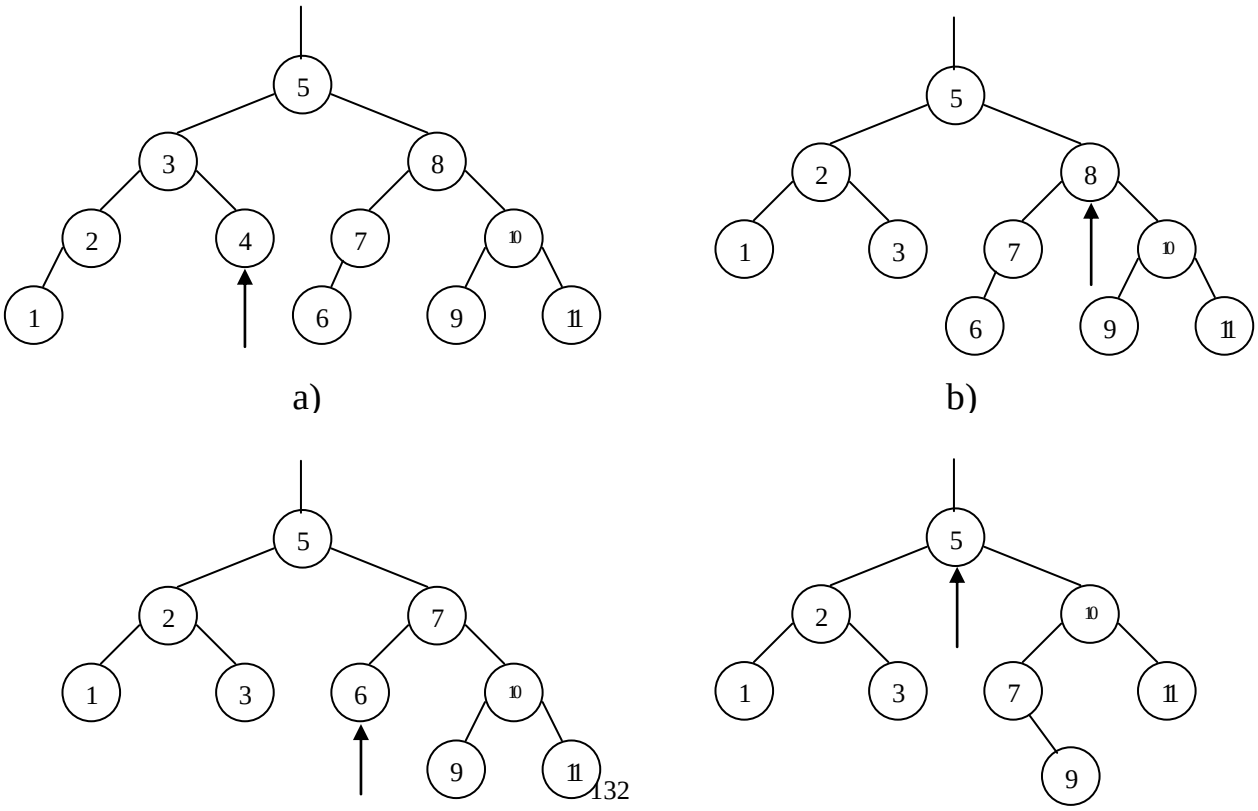
Из-за сложности операций балансировки рекомендуется использовать сбалансированные деревья лишь в том случае, когда поиск информации происходит значительно чаще, чем включение.

Удаление из сбалансированного дерева

Операция удаления узла из сбалансированного дерева является более сложной, чем включения. В тоже время операция балансировки остается в основном такой же, что и при включении, т.е. состоит из однократного или двукратного поворота узлов.

Рассмотрим алгоритм удаления из сбалансированного дерева. Этапы, а также результат удаления узлов из сбалансированного дерева показаны на рис. 7.

Удаление ключа 4 ведет к несбалансированности в узле 3. Его балансировка требует однократного поворота налево. Балансировка вновь становится необходимой после удаления узла 6. В этом случае правое поддерево корня балансируется однократным поворотом направо. Удаление узла 2, хотя само по себе просто, так как он имеет только одного потомка, вызывает сложный двукратный поворот направо и налево. Двукратный поворот направо и налево вызывается удалением узла 7, который прежде заменяется самым правым элементом левого поддерева, т.е. узлом с ключом 3.



Оптимальные деревья поиска

Известно, что существуют случаи, когда имеется информация о вероятности обращений к отдельным ключам. Для таких случаев обычно характерно, что ключи остаются постоянными, т.е. дерево поиска не подвергается ни включениям, ни удалениям, а сохраняет постоянную структуру. Типичным примером служит сканер транслятора, который для каждого слова (идентификатора) определяет, является ли оно зарезервированным (ключевым) словом. Статистические измерения, проведенные на сотнях транслируемых программ, могут в этом случае дать точные сведения о частоте появления отдельных ключей и, следовательно, о вероятности обращения к ним.

Предположим, что p_i — вероятность обращения к узлу i в дереве поиска :

$$\Pr\{x = k_i\} = p_i, \quad \sum_{i=1}^{n} p_i = 1.$$

Теперь мы хотим организовать дерево поиска так, чтобы общее число шагов поиска, подсчитанное для достаточно большого количества опытов, было минимальным. Для этого припишем каждому узлу в определении длины пути некоторый вес. Узлы, к которым часто обращаются, считаются тяжелыми, а посещаемые редко – легкими.

Взвешенная длина пути – сумма всех путей от корня к каждому узлу, умноженных на вероятность обращения к этому узлу:

$$P_I = \sum_{i=1}^{n} p_i h_i \ ,$$

h_i – уровень узла i (или его расстояние от корня +1). Наша цель – минимизировать взвешенную длину пути для данного распределения вероятностей.

Пример сканера в трансляторе наводит на мысль, что слова, встречающиеся в исходном тексте, не всегда являются зарезервированными словами; в действительности это скорее исключение, чем правило. Выяснение того, что данное слово не является ключом в дереве поиска, можно рассматривать как обращение к гипотетическому «специальному узлу», вставленному между меньшим и большим ключами и имеющему соответствующую длину внешнего пути. Если известна также вероятность q_i того, что аргумент поиска x лежит между двумя ключами k_i и k_{i+1}, то это может существенно повлиять на структуру оптимального дерева поиска. Обобщим задачу, учитывая и неудачные поиски.

Общая взвешенная длина имеет теперь следующий вид:

$$P = \sum_{i=1}^{n} p_i h_i + \sum_{j=0}^{m} q_j h_j^{'}, \quad \text{где } \sum_{i=1}^{n} p_i + \sum_{j=0}^{m} q_j = 1,$$

h_i – уровень внутреннего узла i, h_j – уровень внешнего узла j. Среднюю взвешенную длину пути можно назвать «ценой» дерева поиска, так как она является мерой ожидаемого количества затрат на поиск.

Оптимальное дерево поиска – дерево поиска, структура которого дает минимальную цену для всех деревьев с заданным множеством ключей k_i и вероятностями обращений p_i и q_j.

Необходимо отметить одно важное свойство оптимальных деревьев: все их поддеревья тоже являются оптимальными. Эта особенность предполагает алгоритм, который систематически находит все большие и большие деревья, начиная с отдельных узлов как наименьших возможных поддеревьев. Таким образом, дерево растет «от листьев к корню».

3. Варианты заданий

1. Создайте объект «Дерево». Напишите программу, демонстрирующую возможности данного объекта.
2. Напишите программу обхода дерева сверху вниз.
3. Напишите программу обхода дерева слева направо.
4. Напишите программу обхода дерева снизу вверх.
5. Напишите программу, которая находит элемент с заданным ключом X в бинарном дереве, используя обход дерева сверху вниз.
6. Напишите программу, которая находит элемент с заданным ключом X в бинарном дереве, используя обход дерева слева направо.
7. Напишите программу, которая находит элемент с заданным ключом X в бинарном дереве, используя обход дерева снизу вверх.
8. Напишите программу, реализующую операцию включения узла в бинарном дереве.
9. Напишите программу, реализующую операцию удаления узла в бинарном дереве.
10. Напишите программу, реализующую операцию включения узла в сбалансированном дереве.
11. Напишите программу, реализующую операцию удаления узла в сбалансированном дереве.
12. Разработайте объект «Дерево». Напишите программу, демонстрирующую возможности данного объекта в графическом режиме.
13. Используя алгоритм бинарного поиска написать программу поиска задаваемого числа в числовом массиве (находится в памяти).
14. Используя алгоритм бинарного поиска написать программу поиска задаваемого числа в числовом массиве (находится на внешнем носителе).
15. Напишите программу, представляющую арифметическое выражение в виде бинарного дерева (в прямом порядке).
16. Напишите программу, представляющую арифметическое выражение в виде бинарного дерева (в обратном порядке).

17. Напишите программу, представляющую арифметическое выражение в виде бинарного дерева (в концевом порядке).
18. Напишите программу, которая читает тексты программ, находит все определения и вызовы процедур подпрограмм и пытается установить топологическое упорядочение на подпрограммах. Пусть $P \prec Q$ выполняется, если P вызывается в Q.

4. Указания по составлению отчета

Отчет должен содержать постановку и описание задачи, алгоритм решения задачи, текст составленной программы на языке C/C++, краткие выводы.