ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

УПРАВЛЕНИЕ ДИСТАНЦИОННОГО ОБУЧЕНИЯ И ПОВЫШЕНИЯ КВАЛИФИКАЦИИ

Кафедра «Информационные технологии»

СБОРНИК УПРАЖНЕНИЙ

по дисциплине

«Теория алгоритмов»

Авторы
Рашидова Е.В.,
Борисова Е.В.,
Зубарева Е.Г.

Ростов-на-Дону, 2014

Аннотация

Методические указания предназначены для проведения лабораторных работ по дисциплине «Теория алгоритмов» (для студентов 2 – 3 курса специальностей дневного и заочного отделений).

Каждая лабораторная работа включает набор заданий, методические указания к ним и контрольные вопросы по изучаемой теме. В конце методических указаний приведен перечень рекомендуемой литературы.

Методические указания могут быть использованы для самостоятельной работы.

Авторы

к.ф.-м.н. доц. Рашидова Е.В.,
ст. преп.Борисова Е.В.,
ст. преп.Зубарева Е.Г.

Оглавление

ЛАБОРАТОРНАЯ РАБОТА № 1
«ОПРЕДЕЛЕНИЕ МАШИНЫ ТЬЮРИНГА. ПРИМЕНЕНИЕ МАШИНЫ ТЬЮРИНГА К СЛОВАМ. КОНСТРУИРОВАНИЕ МАШИН ТЬЮРИНГА»

Цель работы: получить практические навыки в реализации алгоритмов с использованием машин Тьюринга.

Теоретический материал

Символьные конструкции

Алфавитом будем называть любое конечное множество попарно различных знаков, называемых **буквами** (символами) этого алфавита. Алфавит будем обозначать заглавными буквами, например:

$$A = \{a, b, \ldots, z\};\ B = \{0,1\};\ C = \{\Delta, +, !, 0\}.$$

Символом λ будем обозначать пустой символ.

Словом в данном алфавите называется любая конечная (в том числе и пустая) последовательность букв этого алфавита. Слова будем обозначать малыми греческими буквами.

Например: α = алгоритм – слово в алфавите *A*; β = 1010100 – слово в алфавите *B*; $\gamma = +0\Delta$ – слово в алфавите *C*.

Пустое слово будем обозначать Λ.

Длина слова α (обозначается $|\alpha|$) – это количество букв в слове.

Определим некоторые отношения и операции над словами.

Равенство слов в алфавите *A* определяется индуктивно:

а) пустые слова равны

б) если слово α равно слову β, то $\alpha b = \beta b$, где *b* – буква в алфавите A.

Если слово α является частью слова β, то говорят, что имеет место **вхождение** слова α в слово β (слово α называется подсловом слова β). Это можно записать следующим образом: $\exists \gamma, \delta : \gamma\alpha\delta = \beta$, где γ, δ – слова в алфавите *A*.

Слово α называется **началом слова** β, если $\exists \gamma : \alpha\gamma = \beta$; **концом** _слова β, если $\exists \gamma : \gamma\alpha = \beta$. Слово длины *n*, составленное из буквы *a*, повторенной *n* раз, будем обо-

значать a^n, например *xyxxxyyyy* = xyx^3y^4.

Операция (и результат) приписывания слов α и β друг к другу называется **конкатенацией** (обозначается $\alpha//\beta$). Например, если

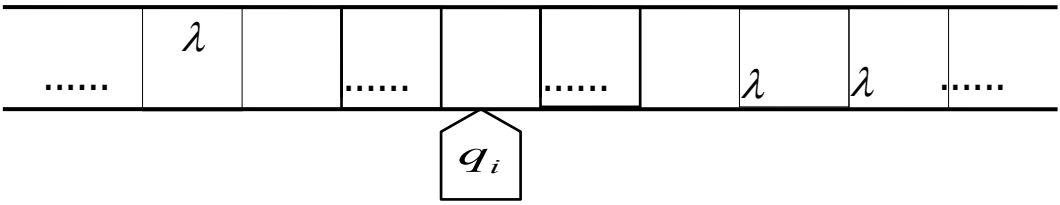
$$\alpha = aabbcc \quad \| \beta = abc \quad \Rightarrow \quad \alpha||\beta = aabbccabc.$$

Определение машины Тьюринга (МТ)

Под машиной Тьюринга понимается некоторая гипотетическая (абстрактная) машина, состоящая из следующих частей:

1) бесконечной в обе стороны ленты, разбитой на ячейки, в каждой из ячеек может быть записан только один символ из алфавита $A = \{a_1, a_2, \ldots a_n\}$, а также пустой символ λ;

2) рабочей головки или управляющего устройства (УУ), которое в каждый момент времени может находиться в одном из состояний множества $Q = \{q_1, q_2, \ldots q_n\}$. В каждом из состояний головка размещается напротив ячейки и может считывать (обозревать) или записывать в нее букву из алфавита *A*.



Машина Тьюринга

Функционирование МТ состоит из последовательности элементарных шагов (тактов). На каждом шаге выполняются следующие действия:

1) управляющее устройство считывает (обозревает) символ a_j;

2) в зависимости от своего состояния q_i и обозреваемого символа a_j

УУ вырабатывает символ $a'_j \in A$ и записывает его в обозреваемую ячейку (возможно $a'_j = a_j$);

3) головка перемещается на одну ячейку вправо *(R)*, влево *(L)* или остается на месте *(E)*;

4) головка переходит в другое внутреннее состояние q_i' (возможно $q_i' = q_i$).

Состояние q_1 называется начальным, q_Z – заключительным. При переходе в заключительное состояние машина останавливается.

Полное состояние МТ называется *конфигурацией*. Это распределение букв по ячейкам ленты, состояние рабочей головки и обозреваемая ячейка. Конфигурация в такте t записывается в виде: $K_t = \alpha q_i a_j \beta$, где α – подслово слева от обозреваемой ячейки, a_j – буква в обозреваемой ячейке, β – подслово справа. Начальная конфигурация $K_1 = q_1\alpha$ и конечная $K_Z = q_Z\alpha$ называются стандартными.

Для описания работы МТ существует 3 способа:

1) система команд вида $q_i a_j \to q_i' a_j' s, s \in \{R, L, E\}$;

2) функциональная таблица;

3) граф (диаграмма) переходов.

С помощью МТ можно описывать выполнение арифметических операций над числами. При этом числа представляются на ленте, как слова в алфавите, состоящем из цифр какой-нибудь системы счисления, и разделяющихся специальным знаком, не входящем данный алфавит, например, "*".

Наиболее употребительной является унарная система, состоящая из одного символа – |. Число *X* в унарной системе счисления на ленте записывается словом $\underbrace{|||||\ldots|||||}_{x}$, (сокращенно $|^x$) в алфавите *A={ | }*.

Пример 1. Операция сложения двух чисел в унарном коде.

Начальная конфигурация: $q_1 |^a * |^b$. Конечная конфигурация: $q_z |^a |^b$, т.е. сложение фактически сводится к приписыванию числа *b* к числу *a* . Для этого первый символ | стирается, а *

заменяется на $|$.

Система команд при $A=\{|,\lambda,*\}$ и $Q=\{q_1,q_2,q_3,q_z\}$.

1. $q_1|\to q_2\lambda R$
2. $q_1*\to q_z\lambda R$
3. $q_2|\to q_2|R$
4. $q_2*\to q_3|L$
5. $q_3|\to q_3|L$
6. $q_3\lambda\to q_z\lambda R$

Комментарий к системе команд

1. $q_1|\to q_2\lambda R$ – стирание первого символа $|$.

Если в обозреваемой ячейке записан символ $|$ и МТ находится в состоянии q_1, тогда состояние изменяется на q_2, обозреваемый символ заменяется на пустой, УУ сдвигается вправо.

2. $q_1*\to q_z\lambda R$ – стирание символа $*$, первый аргумент равняется 0.

Если в обозреваемой ячейке записан символ $*$ и МТ в состоянии q_1 (первый аргумент равняется 0), тогда состояние изменяется на q_z, обозреваемый символ заменяется на пустой, УУ сдвигается вправо.

3. $q_2|\to q_2|R$ – сдвиг вправо.

Если в обозреваемой ячейке записан символ, записан символ $|$ и МТ находится в состоянии q_2, тогда состояние и обозреваемый символ не изменяются, УУ сдвигается вправо.

4. $q_2*\to q_3|L$ – стирание символа $*$.

Если в обозреваемой ячейке записан символ $*$ и МТ находится в состоянии q_2, тогда состояние изменяется на q_3, и обозреваемый символ заменяется на $|$, УУ сдвигается влево (конец первого аргумента).

5. $q_3 | \to q_3 | L$ – сдвиг влево.

Если в обозреваемой ячейке записан символ | и МТ находится в состоянии q_3, тогда состояние и обозреваемый символ не изменяются, УУ сдвигается влево.

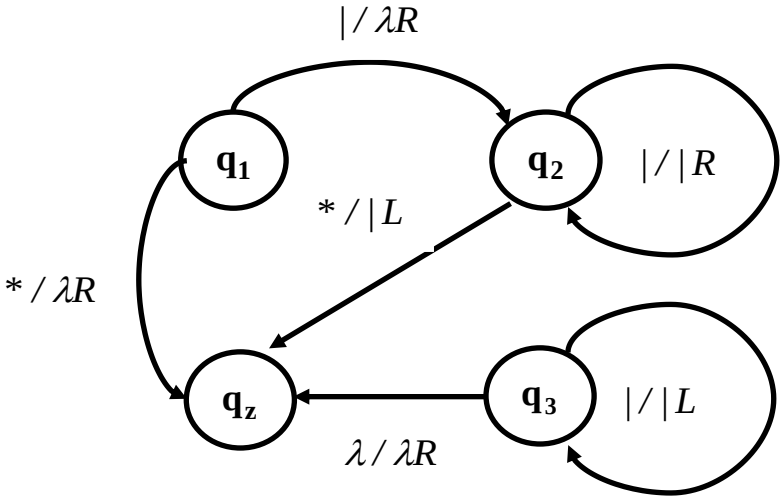
6. $q_3 \lambda \to q_z \lambda R$ –

Если в обозреваемой ячейке записан символ λ и МТ находится в состоянии q_3, тогда состояние изменяется на q_z, обозреваемый символ не изменяется, УУ сдвигается вправо (конец алгоритма, УУ расположено в начале рабочей зоны).

Описание МТ в виде функциональной таблицы:

$q_i \backslash a_j$	|	*	λ
q_1	$q_2 \lambda R$	$q_Z \lambda R$	-
q_2	$q_2 \| R$	$q_3 \| L$	-
q_3	$q_3 \| L$	-	$q_Z \lambda R$

Описание МТ в виде диаграммы переходов



Вычисление на МТ словарной функции f будем понимать следующим образом. Пусть в начальной конфигурации на ленте записано слово α. Если значение $f(\alpha)$ определено, то конечного числа шагов (тактов) машина должна перейти в заключительную конфигурацию, в которой на ленте записано слово $\beta = f(\alpha)$. В противном случае МТ должна работать бесконечно.

Числовая функция $f(x_1,\ldots,x_n)$ **правильно вычислима** (или просто вычислима) **по Тьюрингу**, если существует МТ, которая переводит конфигурацию $q_1 1^{x_1} * 1^{x_2} * \ldots * 1^{x_n}$ в конфигурацию $q_Z 1^y$, когда $f(x_1,\ldots,x_n)=y$, или работает бесконечно, когда $f(x_1,\ldots,x_n)$ не определена.

Задание на лабораторную работу

1.Описать системой команд, функциональной таблицей и диаграммой переходов работу машины Тьюринга, реализующую заданный вариантом алгоритм. Начальная и конечная конфигурации стандартны.

2. Проверить модель алгоритма на множестве тестовых примеров. Привести последовательности конфигураций машины Тьюринга, заданной в предыдущем пункте, для различных тестовых исходных слов.

Варианты заданий

1. Реализовать функцию арифметическое вычитание $x \dot{-} y$ в унарном коде.

2. Реализовать функцию выбор максимального из двух чисел $Max(x, y)$ над числами в унарном коде.

3. Реализовать функцию $Min(x, y)$ над числами в унарном коде.

4. Реализовать функцию $x \bmod y$ над числами в унарном коде.

5. Реализовать функцию $x \operatorname{div} y$ над числами в унарном коде.

6. Реализовать функцию $|x - y|$ над числами в унарном

коде.

7. Реализовать функцию выбор аргумента $I_m^n(x_1,x_2,\ldots,x_n)=x_m;(m\le n)$ над числами в унарном коде.

8. Реализовать вычисление предиката $X>Y$ в унарном коде с сохранением (восстановлением) исходных данных.

9. Реализовать вычисление предиката $X=Y$ в унарном коде с сохранением (восстановлением) исходных данных.

10. Реализовать вычисление предиката "x - четное число" в двоичном коде.

11. Реализовать алгоритм в алфавите $A=\{0,1\}$, меняющий местами первую и последнюю буквы слова.

12. Реализовать алгоритм над алфавитом $A=\{0,1\}$, меняющий местами первый ноль и последнюю единицу.

13. Реализовать операцию копирование в алфавите $\{0,1\}$, то есть получить из слова α слово $\alpha*\alpha$.

14. Реализовать алгоритм над алфавитом $A=\{0,1\}$, который выдает единицу, если в исходном слове только парные нули и ноль в противном случае.

15. Реализовать алгоритм в алфавите $A=\{0,1\}$, который переставляет буквы в слове α так, чтобы сначала шли все нули, потом – единицы.

16. Реализовать алгоритм, конструирующий в алфавите $A=\{0,1\}$ слова вида $\alpha=11010^2 10^3\ldots 10^n 1$, где n - произвольное натуральное число.

17. Реализовать алгоритм, реализующий функцию циклический сдвиг двоичного числа на одну ячейку.

18. Реализовать алгоритм в алфавите $A=\{1,2,3\}$, анализирующий последовательность цифр в слове и выдающий «+», если цифры образуют неубывающую последовательность, и «–» в противном случае.

19. Реализовать выделение подстроки, заключенной между двумя символами * (первая пара) в алфавите $\{a,b,*\}$. Если последовательность "* . . .*" отсутствует на ленте, стереть все.

20. В слове α в алфавите $\{a,b\}$ стереть все, кроме $\beta=aabb$. Если такой последовательности нет, все стереть.

21. Реализовать алгоритм над алфавитом $\{a,b\}$, переставляющий буквы в обратном порядке.

22. Реализовать предиката «в слове α в алфавите $A=\{x,y,z\}$ есть пара букв '*yy*' » .

23. Реализовать алгоритм в алфавите $A=\{a,b,c\}$, производящий в слове α алфавита замену всех вхождений буквы *а* на букву *б*.

24. Реализовать алгоритм в алфавите $A=\{0,1\}$ для вычисления логической функции $x \wedge y \vee z$, где *x,y,z* принимают значение 0 или 1.

25. Реализовать алгоритм в алфавите $A=\{0,1\}$ для вычисления логической функции $\overline{x \wedge y}$, где *x,y,z* принимают значение 0 или 1.

Контрольные вопросы

1. Дать определение машины Тьюринга (МТ) и ее составляющим.
2. Перечислить и определить способы описания МТ.
3. Какие операции выполняются в каждом такте работы МТ?
4. Дать определение конфигурации МТ.
5. Какие начальные и конечные конфигурации называют стандартными и как они обозначаются?
6. Что такое функция, правильно вычислимая по Тьюрингу?
7. Какие способы композиции МТ существуют, как они применяются и обозначаются?
8. Формулировка тезиса Тьюринга; можно ли его доказать строго?

ЛАБОРАТОРНАЯ РАБОТА №2
«КОМПОЗИЦИЯ МАШИН ТЬЮРИНГА. СПОСОБЫ КОМПОЗИЦИИ МАШИН ТЬЮРИНГА. АЛГОРИТМЫ С ИСПОЛЬЗОВАНИЕМ КОМПОЗИЦИИ»

Цель работы: получить практические навыки в записи алгоритмов с использованием композиции машин Тьюринга.

Теоретический материал

Вышеперечисленные способы описания МТ практически можно использовать только для несложных алгоритмов, в противном случае описание становится слишком громоздким. Машины Тьюринга для сложных алгоритмов могут строиться с использованием уже имеющихся элементарных МТ и такое построение называется **композицией МТ**.

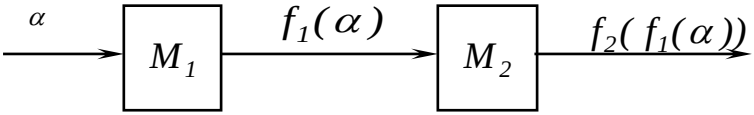
Опишем 4 основных способа композиции МТ:

- последовательная композиция (суперпозиция);
- параллельная композиция;
- разветвление
- цикл

1. Последовательная композиция машин Тьюринга

Последовательной композицией или **суперпозицией** машин Тьюринга M_1 и M_2, вычисляющих словарные функции $f_1(\alpha)$ и $f_2(\alpha)$ в алфавите *A*, называется машина *M*, вычисляющая функцию $f(\alpha) = f_2(f_1(\alpha))$.

Последовательная композиция изображается следующим образом:



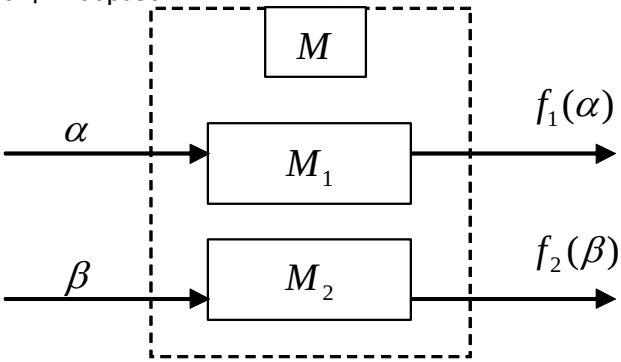
и обозначается $M = M_1 \circ M_2$ или $M = M_2(M_1)$.

2. Параллельная композиция машин Тьюринга

Параллельной композицией машин M_1 и M_2, вычисляющих словарные функции $f_1(\alpha)$ и $f_2(\beta)$ в алфавитах *A* и *B*, соответственно, называется машина *M*, вычисляющая сло-

варную функцию $f(\alpha||\beta) = f_1(\alpha)||f_2(\beta)$. Здесь знак || используется для разделения слов при параллельной композиции МТ.

Параллельная композиция МТ M_1 и M_2 изображается следующим образом:



и обозначается: $M = M_1 * M_2$.

Фактически параллельная композиция двух МТ получает на вход слово, состоящее из 2-х слов в разных алфавитах, и на выходе выдает слово, также состоящее из 2-х слов, т.е. представляет собой две одновременно и независимо работающие машины. Для реализации параллельной композиции используется машина с двухэтажной лентой.

Машина с двухэтажной лентой работает следующим образом:

1) слово β переписывается на второй этаж ленты и стирается на первом,

2) вычисляется $f_1(\alpha)$ на первом этаже,

3) вычисляется $f_2(\beta)$ на втором этаже

4) $f_2(\beta)$ переписывается на первый этаж, возможно, со сдвигом.

Команда МТ с двухэтажной лентой записывается следующим образом:

$$q_i \frac{b_k}{a_j} \to q_i' \frac{b_k'}{a_j'},$$

где a_j, b_k – буквы, записанные соответственно на первом

и втором этажах. Обозначим длины слов α,β, соответственно, n,m.

Продемонстрируем работу машины Тьюринга с двухэтажной лентой. В общем случае длины слов α,β и $f_1(\alpha),f_2(\beta)$ не совпадают между собой, но для простоты изображения принимаем, что они равны. Тогда реализация пунктов 1)-4) на МТ с двухэтажной лентой выполняется таким образом:

λ	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ
λ	a_1	a_2	$\cdots$	a_n	$\Vert$	b_1	b_2	$\cdots$	b_m	λ

λ	b_1	b_2	$\cdots$	b_m	λ
λ	a_1	a_2	$\cdots$	a_n	λ

λ	$f_2(\beta)$	λ
λ	$f_1(\alpha)$	λ

λ	$\lambda\lambda\lambda\lambda..\lambda$	λ
λ	$f_1(\alpha)\Vert f_2(\beta)$	λ

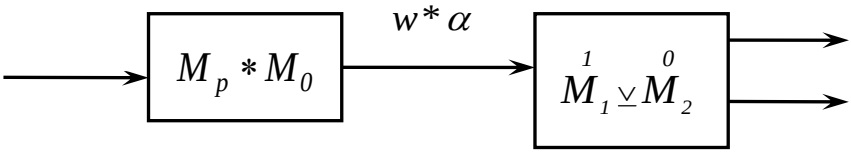
Для реализации параллельной композиции *n* машин Тьюринга используется *n*–этажная лента.

3. Разветвление или условный переход в композиции машин Тьюринга

Если заданы машины Тьюринга M_1 и M_2, вычисляющие словарные функции $f_1(\alpha)$ и $f_2(\alpha)$, и машина M_P, вычисляющая некоторый предикат $P(\alpha)$ с восстановлением (т.е. без стирания слова α), то для реализации разветвления может быть построена машина Тьюринга M, вычисляющая функцию:

$$f(\alpha)=\begin{cases} f_1(\alpha), & P(\alpha)\equiv''true'' \\ f_2(\alpha), & P(\alpha)\equiv'' false''. \end{cases}$$

Разветвление машин Тьюринга на схемах композиции изображается следующим образом:



и обозначается $M = M_1 \veebar M_2$, здесь w – результат работы машины M_P, принимающий значения «1», если предикат *P(α)="true"* и «0», если предикат *P(α)="false"*, M_0 – машина Тьюринга, реализуюшая копирование входного слова $\alpha \to \alpha$.

4. Цикл в композиции машин Тьюринга

Цикл в композиции МТ реализуется по тем же принципам, что и разветвление.

Циклическим будем считать следующий алгоритм M :

« пока *P(α)= "true"*, выполнять $f(\alpha)$ »,

где α – слово на ленте перед первым выполнением M и после очередного выполнения.

Для изображения цикла введем некоторые обозначения, пусть:

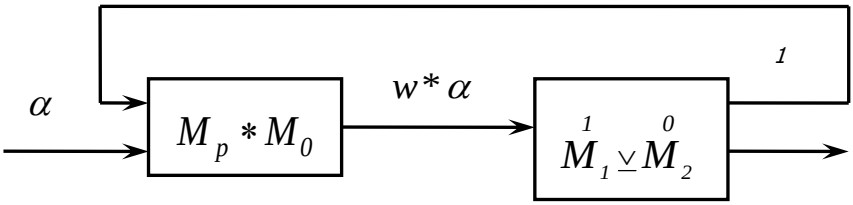
M_p – машина Тьюринга, реализующая вычисление предиката *P(α);*

M_0 – МТ, реализующая копирование входного слова $\alpha \to \alpha$;

M_1 – МТ, выполняемая в цикле и реализующая $f_1(\alpha)$;

M_2 – МТ, выполняемая при выходе из цикла и реализующая $f_2(\alpha)$.

Тогда, циклическая композиция машин Тьюринга или цикл, может быть изображена следующим образом:



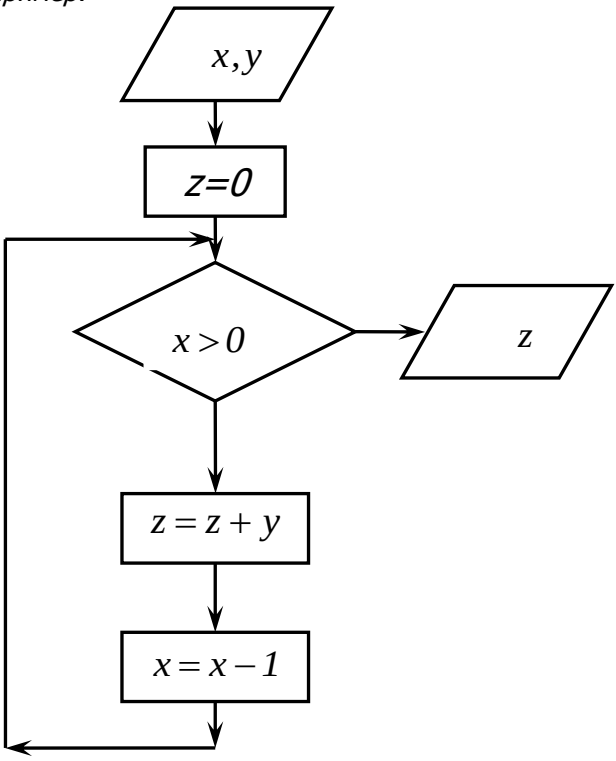
Программирование с помощью композиций машин Тьюринга:

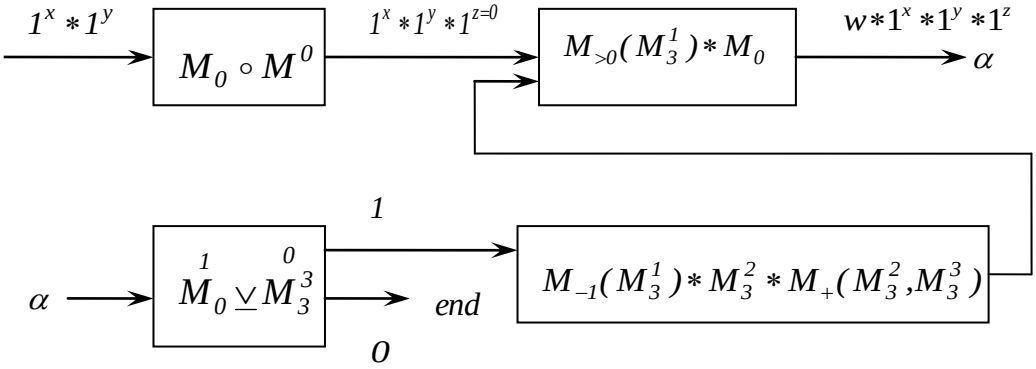
1) построение блок-схем сложных алгоритмов такой степени детализации, что их блоки соответствуют элементарным МТ;

2) построение элементарных МТ, реализующих простые блоки;

3) объединение элементарных МТ в композицию МТ.

Пример.





M_0 – машина Тьюринга, реализующая копирование входного слова;

M^0 – МТ, реализующая функцию установки константы ноль;

$M_{>0}$ – МТ, вычисляющая предикат с восстановлением $x > 0$;

M_n^i – МТ, реализующая функцию выбора i-того аргумента из n аргументов;

M_{-1} – МТ, реализующая функцию уменьшение аргумента x на 1 в унарном коде (вытирает крайний левый символ $|$);

M_+ – МТ, выполняющая сложение двух чисел в унарном коде.

Следует отметить, что в любом случае необходимо в начале выполнения алгоритма выполнить проверку входных данных на корректность (например, равенство 0 аргумента при делении).

ЗАДАНИЕ

Построить машину Тьюринга, вычисляющую функцию $f(n)$. Функцию выбрать из вариантов заданий перечисленных ниже. Машину Тьюринга представить, как композицию элементарных МТ, выполняющих операции: копирование аргумента, сложение, умножение, арифметическое вычитание, нахождение целой части и остатка от деления, сравнения чисел, выделение аргумента. Недостающие элементарные МТ описать любым известным способом.

Варианты заданий

1. Сумма всех четных делителей числа n.
2. Количество всех нечетных делителей числа n.
3. Количество нулей в двоичной записи n.
4. Сумма цифр в двоичной записи n.
5. Количество взаимно-простых с n чисел, $\le n$.
6. Максимальная цифра в 8-ричной записи числа n.
7. Минимальная цифра в 8-ричной записи числа n.
8. Количество четных цифр в 8-ричной записи числа n.
9. Количество нечетных цифр в 8-ричной записи числа n.
10. Сумма простых делителей числа n.
11. Количество простых делителей числа n.
12. Количество простых чисел, $\le n$.
13. Количество чисел, являющихся полными квадратами, $\le n$.
14. Сумма чисел, являющихся степенью двойки, $\le n$.
15. Максимальная цифра в 16-ричной записи числа n.
16. Минимальная цифра в 16-ричной записи числа n.
17. Ближайшее к n простое число.
18. Произведение делителей числа n.
19. Произведение простых делителей числа n.
20. Произведение взаимно-простых с n чисел, $\le n$.
21. Наименьшее общее кратное двух чисел, $K(x,y)$, $K(x,0)=K(0,y)=0$.
22. Наибольший общий делитель двух чисел, $D(x,y), D(0,0)=0$.
23. Функция, отличная от нуля в конечном числе точек.
24. Номер наибольшего простого делителя числа n.
25. Функция, вычисляющая целую часть квадратного корня от аргумента, $y=\left[\sqrt{x}\right]$.

Контрольные вопросы

1. Композиции машин Тьюринга и область их применения?
2. Дать определение и привести обозначение суперпозиции или последовательной композиции машин Тьюринга.
3. Дать определение и привести обозначение паралелльной

композиции машин Тьюринга.

4. Двухэтажная и n — этажная ленты, использование их в паралельной композиции машин Тьюринга.

5. Дать определение и привести обозначение разветвления или условного перехода в композиции машин Тьюринга.

6. Дать определение и привести обозначение цикла в композиции машин Тьюринга.

ЛАБОРАТОРНАЯ РАБОТА № 3 «СОРТИРОВКА МАССИВОВ. МЕТОДЫ СОРТИРОВОК»

Цель: изучить особенности основных алгоритмов сортировки массивов, получить практические навыки алгоритмизации задач сортировки массива.

Теоретический материал

В данной лабораторной работе представлены принципиальные алгоритмы внутренней сортировки. Задача сортировки состоит в упорядочивании элементов в неубывающем (невозрастающем) порядке. Основным условием, предъявляемым к алгоритмам сортировки, является экономное использование доступной памяти. Это предполагает, что перестановки элементов с целью их упорядочивания должны выполняться *на том же месте,* т. е. методы, в которых элементы из массива *a* передаются в результирующий массив *b*, представляют существенно меньший интерес.

Эффективность алгоритмов сортировки массива можно оценить по таким критериям, как число необходимых сравнений элементов (*C*) и число перестановок элементов (*M*).

Эти числа фактически являются функциями от *n* – числа сортируемых элементов. Хотя хорошие алгоритмы сортировки требуют порядка $n * log(n)$ сравнений, рассмотрим несколько простых и очевидных методов, называемых *прямыми,* где требуется порядка n^2 сравнений элементов. Разбор прямых методов целесообразен ввиду следующих причин:

1. Прямые методы особенно удобны для объяснения характерных черт основных принципов большинства сортировок.
2. Программы этих методов легко понимать, и они коротки. Следует помнить, что сами программы также занимают память.
3. Усложненные 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:

- Сортировки с помощью включения (by insertion).
- Сортировки с помощью выбора (by selection).
- Сортировки с помощью обмена (by exchange).

Все рассматриваемые алгоритмы будут оперировать массивом *a*, в котором будут храниться переставляемые на месте элементы. Данный массив объявляется следующим образом: *int a*[*n*];

1.1. Сортировка с помощью прямого включения

При сортировке элементов массива с помощью прямого включения массив делят на две части: отсортированную или «готовую» a_1, …, a_{i-1} и неотсортированную или «исходную» a_i, …, a_n.

В начале работы алгоритмы в качестве отсортированной части массива принимается только один первый элемент, а в качестве неотсортированной части – все остальные элементы.

На каждом шаге, начиная с $i = 2$ и увеличивая i каждый раз на единицу, из неотсортированной последовательности извлекается i-й элемент и вставляется в отсортированную так, чтобы не нарушить в ней упорядоченности элементов. Каждый шаг алгоритма включает четыре действия:

1. Взять i-й элемент массива, он же является первым элементом в неотсортированной части, и сохранить его в дополнительной переменной.
2. Найти позицию j в отсортированной части массива, куда будет вставлен i-й элемент, значение которого теперь хранится в дополнительной переменной. Вставка этого элемента не должна нарушить упорядоченности элементов отсортированной части массива.
3. Сдвиг элементов массива с $i - 1$ позиции по j-ю на один элемент вправо, чтобы освободить найденную позицию для вставки.
4. Вставка взятого элемента в найденную j-ю позицию.

На рис.1 приведены два первых шага работы алгоритма сортировки с помощью прямого включения. Отсортированная и неотсортированная части отделены вертикальной чертой.

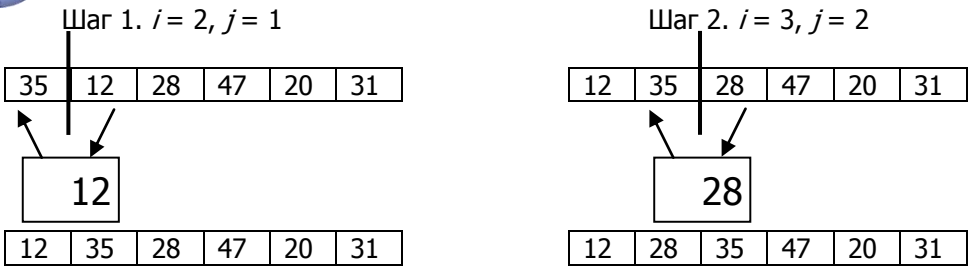


Рис. 1. Первые два шага алгоритма сортировки с помощью прямого включения

В табл. 1 показан пример работы алгоритма сортировки массива из шести элементов. Отсортированная часть массива подчеркнута.

Таблица1

Пример сортировки с помощью прямого включения

	35	12	28	47	20	31
$i = 2$	12	35	28	47	20	31
$i = 3$	12	28	35	47	20	31
$i = 4$	12	28	35	47	20	31
$i = 5$	12	20	28	35	47	31
$i = 6$	12	20	28	31	35	47

Анализ алгоритма сортировки с прямым включением. Число сравнений ключей (C_i) при i-м проходе самое большее равно $i - 1$, самое меньшее – 1. Если предположить, что все перестановки из n элементов равновероятны, то среднее число сравнений – $i / 2$. Число же перестановок M_i равно $C_i + 2$.

Данный алгоритм показывает наилучшие результаты работы в случае уже упорядоченной исходной части массива, наихудшие – когда элементы первоначально расположены в обратном порядке. Алгоритм с прямым включением можно легко улучшить, если обратить внимание на то, что готовая последовательность, в которую надо вставить новый элемент, сама уже упорядочена. Естественным является остановиться на двоичном поиске, при котором делается попытка сравнения с серединой готовой последовательности, а затем процесс деления пополам идет до тех пор, пока не будет найдена точка включения. Такой модифицированный алгоритм сортировки называется *методом с двоичным включением* (binary insertion).

К несчастью, улучшения, порожденные введением двоичного поиска, касаются лишь числа сравнения, а не числа необходимых перестановок. А поскольку движение элемента, т. е. самого элемента, и связанной с ним информации занимает значительно больше времени, чем сравнение двух ключей, то фактически улучшения не столь уж существенны, ведь важный член M так и продолжает оставаться порядка n^2. И на самом деле, сортировка уже отсортированного массива потребует больше времени, чем в случае последовательной сортировки с прямыми включениями.

Этот пример показывает, что «очевидные улучшения» часто дают не столь уж большой выигрыш, как это кажется на первый взгляд, а в некоторых случаях (случающихся на самом деле) эти «улучшения» могут фактически привести к ухудшениям. После всего сказанного сортировка с помощью включения уже не кажется столь удобным методом для цифровых машин: включение одного элемента с последующим сдвигом на одну позицию целой группы элементов не экономно. Остается впечатление, что лучший результат дадут методы, где передвижка, пусть и на большие расстояния, будет связана лишь с одним-единственным элементом.

1.2. Сортировка с помощью прямого выбора

При сортировке с помощью прямого выбора массив также делится на две части: отсортированную или «готовую» последовательность a_1, …, a_{i-1} и неотсортированную или «исходную» – a_i, …, a_n.

Метод прямого выбора в некотором смысле противоположен методу прямого включения. При прямом включении на каждом шаге рассматриваются только *один* (первый) элемент исходной последовательности и *все* элементы готовой последовательности. При этом отыскивается точка включения этого элемента в «готовую» последовательность так, чтобы при вставке не нарушить ее упорядоченности.

При прямом же выборе происходит поиск *одного* элемента из исходной последовательности, который обладает наименьшим (наибольшим) значением, и уже найденный элемент помещается в конец готовой последовательности.

На момент начала сортировки методом прямого выбора готовая последовательность считается пустой, соответственно, исходная последовательность включает в себя все элементы массива.

Алгоритм сортировки с помощью прямого выбора можно описать следующим образом:

1. Из всего массива выбирается элемент с наименьшим значением.
2. Он меняется местами с первым элементом a_1.
3. Затем этот процесс повторяется с оставшимися $n - 1$ элементами,

$n -$ 2 элементами и т. д. до тех пор, пока не останется один элемент с наибольшим значением.

Рассмотрим несколько первых шагов алгоритма сортировки с помощью прямого выбора на примере упорядочивания по возрастанию следующего массива:

35 12 28 47 20 31

На первом шаге готовая последовательность не содержит ни одного элемента. Выбираем наименьший элемент из исходной последовательности, куда пока что входят все элементы массива. Таким элементом является второй элемент массива, значение которого – 12. Он меняется местами с первым элементом.

Теперь готовая последовательность включает в себя один элемент – 12. На рис. 2, иллюстрирующем текущее состояние массива, готовая последовательность подчеркнута сплошной линией. Наименьший элемент исходной последовательности – 20, он должен занять место второго элемента. Оба эти элемента выделены на рис. 11 полужирным шрифтом.

12 **35** 28 47 **20** 31

Рис. 2. Первый шаг алгоритма сортировки с помощью прямого выбора

На втором шаге готовая последовательность состоит уже из двух элементов: 12 и 20. Наименьший элемент исходной последовательности – 28.

Он является третьим элементом массива и именно эту позицию он должен занять. Поэтому на рис. 3 полужирным шрифтом выделен только один данный элемент.

12 20 **28** 47 35 31

Рис. 3. Второй шаг алгоритма сортировки с помощью прямого выбора

В табл. 3 наглядно показаны все шаги алгоритма сортировки с помощью прямого выбора: отмечены готовые последовательности и переставляемые элементы.

Таблица 2

Пример сортировки с помощью прямого выбора

35	**12**	28	47	20	31	
i = 2	12	**35**	28	47	**20**	31
i = 3	12	20	**28**	47	35	31
i = 4	12	20	28	**47**	35	**31**
i = 5	12	20	28	31	**35**	47
i = 6	12	20	28	31	35	47

Анализ алгоритма сортировки с прямым выбором. При работе данного алгоритма число сравнений элементов (*C*) не зависит от начального порядка элементов. Можно сказать, что в этом смысле поведение данного метода менее естественно, чем поведение прямого включения. Для *C* имеем

$$C = (n^2 - n) / 2.$$

В случае изначально упорядоченных элементов число перестановок *M* минимально:

$$M_{min} = 3 * (n - 1).$$

Если же первоначально элементы располагались в обратном порядке, число перестановок будет максимальным:

$$M_{max} = n^2 / 4 + 3 * (n - 1).$$

В общем случае алгоритм с прямым выбором, как правило, предпочтительнее алгоритма прямого включения. Однако если элементы в начале упорядочены или почти упорядочены, алгоритм с прямым включением выполнит сортировку быстрее.

1.3. Сортировка с помощью прямого обмена

Алгоритм прямого обмена основывается на сравнении и перестановке пары соседних элементов и продолжении этого процесса до тех пор, пока не будут упорядочены все элементы. Обмен местами двух элементов представляет собой характерную особенность данного метода, хотя, конечно, в обоих рассматривавшихся до этого методах переставляемые элементы также «обменивались» местами.

1.3.1. Пузырьковая сортировка

Как и в методе прямого выбора при сортировке данным методом выполняется ряд проходов по массиву, сдвигая каждый раз наименьший элемент оставшейся последовательности к началу массива. Однако в отличие от рассмотренных ранее методов при пузырьковой сортировке происходит сравнивание и перестановка только соседних двух элементов.

Если расположить элементы массива вертикально, то за первый проход элемент с наименьшим значением, т. е. самый «легкий» элемент, поднимется на самый верх массива и станет его первым элементом. В результате следующего прохода второй по «легкости» элемент поднимется к началу массива и станет его вторым элементом. Такое продвижение элементов по массиву вызывает ассоциации с пузырьком воздуха, всплывающим в воде. Отсюда и название данного метода – метод «пузырька» или «пузырьковая сортировка».

Описать алгоритм сортировки методом пузырька можно следующим образом. На *i*-ом проходе алгоритма ($1 \le i \le n$) рассматриваются в обратном порядке элементы массива с *n*-го по *i*-й включительно. Сравниваются только соседние элементы. При этом, если производится сортировка по возрастанию и элемент $a[j]$ оказывается меньше предшествующего ему $a[j-1]$, то они меняются местами.

Если говорить терминами «готовой» и «исходной» последовательностей, то элементы до *i*-го представляют собой готовую последовательность, а с *i*-го по *n*-й – исходную. В начале готовая последовательность пуста. По мере работы алгоритма элементы «всплывают» из исходной последовательности и занимают место в конце готовой.

В табл. 3 показана работа алгоритма сортировки методом пузырька. Перемещение элементов к началу массива показано стрелками.

Таблица 3

Пример пузырьковой сортировки

i = 1	*i* =2	*i* =3	*i* = 4	*i* = 5	*i* = 6	*i* = 7	*i* = 8
21	4	4	4	4	4	4	4
25	21	9	9	9	9	9	9
9	25	21	12	12	12	12	12
17	9	25	21	17	17	17	17
43	17	12	25	21	21	21	21
12	43	17	17	25	25	25	25
4	12	43	32	32	32	32	32
32	32	32	43	43	43	43	43

Улучшения этого алгоритма напрашиваются сами собой. На примере в табл. 3 видно, что три последних прохода не влияют

на порядок элементов поскольку, они уже отсортированы. Очевидный прием улучшения этого алгоритма – запоминать, были или не были перестановки в процессе некоторого прохода. Если в последнем проходе перестановок не было, то работа алгоритма может быть завершена.

Это улучшение однако, можно опять же улучшить, если запоминать не только сам факт, что обмен имел место, но и положение (индекс) последнего обмена. Ясно, что все пары соседних элементов выше этого индекса *k* уже упорядочены. Поэтому просмотр можно заканчивать на этом индексе, а не идти до заранее определенного нижнего предела *i*.

1.3.2. Шейкерная сортировка

Пример пузырьковой сортировки, представленный в табл. 3, отражает некоторую своеобразную асимметрию работы алгоритма: легкий пузырек всплывает сразу – за один проход, а тяжелый тонет очень медленно – за один проход на одну позицию. Например, массив

15 29 31 55 70 93 8

с помощью пузырьковой сортировки будет упорядочен за один проход, а для сортировки массива

93 8 15 29 31 55 70

потребуется шесть проходов. Это наводит на мысль о следующем улучшении: чередовать направление просмотра на каждом последующем проходе. Алгоритм, реализующий такой подход, называется «шейкерной сортировкой». Табл. 4 иллюстрирует сортировку данным методом тех же (табл. 3) восьми элементов. Переменные *L* и *R* содержат индексы элементов, до которых должен происходить просмотр при движении влево (или вверх) и вправо (или вниз) соответственно.

Таблица 4
Пример шейкерной сортировки

L =	1	2	2	3	3
R =	8	8	7	7	6
направление	↑	↓	↑	↓	↑
	21	4	4	4	4
	25	21	21	9	9
	9	25	9	21	12
	17	9	17	12	17
	43	17	25	17	21
	12	43	12	25	25
	4	12	32	32	32
	32	32	43	43	43

Если ввести переменную, которая будет отвечать за то, были ли перестановки элементов, то алгоритм шейкерной сортировки выполнит сортировку за пять проходов, в то время как для пузырьковой сортировки без улучшений потребовалось бы семь проходов.

Если же к алгоритму шейкерной сортировки добавить переменную *k*, содержащую индекс последнего перестановленного элемента, то число проходов сократится до четырех, как показано на примере в табл. 4.

Анализ пузырьковой и шейкерной сортировок. Число сравнений в базовом обменном алгоритме – алгоритме пузырьковой сортировке – равно

$$C = (n^2 - n) / 2,$$

а минимальное и максимальное число перестановок элементов равно

$$M_{min} = 0, \quad M_{max} = 3 * (n^2 - n) / 4.$$

Анализ же улучшенных методов, особенно шейкерной сортировки, довольно сложен. Стоит отметить, что все перечисленные выше усовершенствования не влияют на число перемещений, они лишь сокращают число излишних проверок. К несчастью, обмен местами двух элементов – чаще всего более дорогостоящая операция, чем их сравнение. Поэтому очевидные на первый взгляд улучшения дают не такой уж большой выигрыш, как ожидалось.

Шейкерная сортировка с успехом используется лишь в тех случаях, когда известно, что элементы почти упорядочены, что на

практике бывает весьма редко. Анализ показывает, что «обменная» сортировка и ее усовершенствования фактически оказываются хуже сортировок с помощью включений и с помощью выбора.

1.4. Сортировка Шелла

Все методы прямой сортировки фактически передвигают каждый элемент на всяком элементарном шаге на одну позицию. Поэтому они требуют порядка n^2 таких шагов. Следовательно, в основу любых улучшений алгоритмов сортировки должен быть положен принцип перемещения на каждом такте элементов на большие расстояния. Можно показать, что среднее расстояние, на которое должен продвигаться каждый из n элементов во время сортировки, равно n / 3 позиций. Эта цифра является целью в поиске улучшений, т. е. в поиске более эффективных методов сортировки.

Рассмотрим улучшенный метод, основанный на методе прямого включения – сортировке с помощью включений с уменьшающимися расстояниями. В 1959 г. Д. Шеллом было предложено усовершенствование алгоритма сортировки с помощью прямого включения. Рассмотрим его работу на примере следующего массива:

35 28 49 79 45 11 89 70 91 67 54 19 13 24

Сначала отдельно группируются элементы, отстоящие друг от друга на расстоянии 4. Таких групп будет четыре, они показаны на рис. 4 (а – г). Элементы, принадлежащие одной группе, объединены дугами.

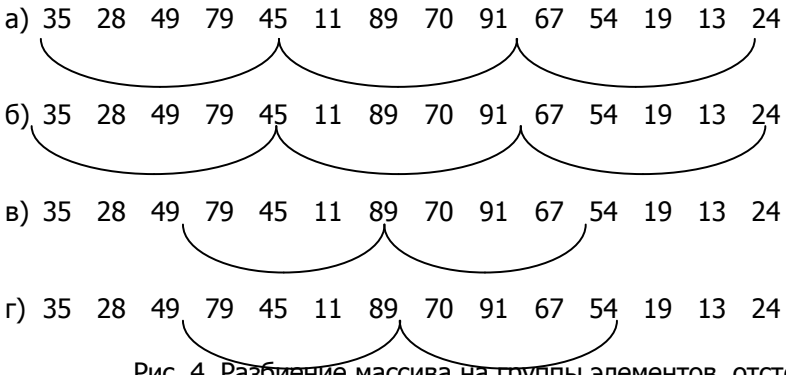


Рис. 4. Разбиение массива на группы элементов, отстоящих друг от друга на четыре

Следующим шагом выполняется сортировка внутри каждой из групп методом прямого включения. Сначала сортируются элементы 35, 45, 91, 13, затем 28, 11, 67, 24, следом 49, 89, 54, и, наконец, 79, 70, 19. В результате получаем массив:

13 11 49 19 35 24 54 70 45 28 89 79 91 67

Такой процесс называется четверной сортировкой. Следует отметить, что алгоритм сортировки Шелла также является алгоритмом сортировки «на месте». Поэтому все перестановки происходят в одном и том же массиве.

Следующим проходом элементы группируются так, что теперь в одну группу входят элементы, отстоящие друг от друг на две позиции. Таких групп две, они показаны на рис. 5 (а – б).

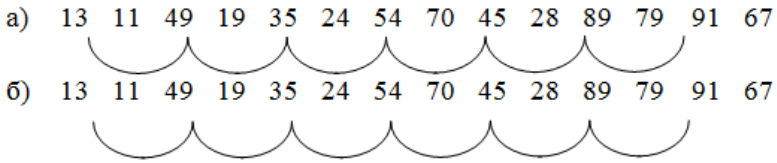


Рис. 5. Разбиение массива на группы элементов, отстоящих друг от друга на два

Вновь в каждой группе выполняется сортировка с помощью прямого включения. Это называется двойной сортировкой, ее результатом будет массив:

13 11 35 19 45 24 49 28 54 67 89 70 91 79

И наконец, на третьем проходе идет обычная или одинарная сортировка с помощью прямого включения (рис. 6).

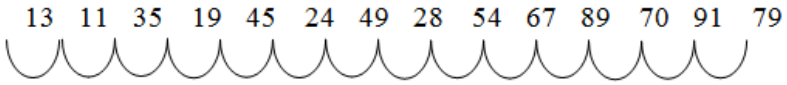


Рис. 6. Группа элементов, отстоящих друг от друга на один

Результатом работы алгоритма сортировки Шелла является отсортированный массив

11 13 19 24 28 35 45 49 54 67 70 79 89 91

На первый взгляд можно засомневаться: если необходимо несколько процессов сортировки, причем и каждый включаются все элементы, то не добавят ли они больше работы, чем

сэкономят? Однако на каждом этапе либо сортируется относительно мало элементов, либо элементы уже довольно хорошо упорядочены и требуется сравнительно немного перестановок.

Очевидно, что такой метод в результате дает упорядоченный массив. Видно также, что каждый проход от предыдущих только выигрывает (т. к. каждая i-сортировка объединяет две группы, уже отсортированные $2i$-сортировкой).

Расстояния в группах можно уменьшать по-разному, лишь бы последнее было единичным, ведь в самом плохом случае последний проход и сделает всю работу. Однако совсем не очевидно, что такой прием «уменьшающихся расстояний» может дать лучшие результаты, если расстояния не будут степенями двойки. При выполнении лабораторных работ рекомендуется начальный шаг взять равным $h_{нач} = n / 3$, где n – количество элементов массива, и уменьшать его на каждом проходе вдвое:

$$h_{i-1} = h_i / 2.$$

Как отмечалось выше $h_{конеч} = 1$.

Анализ сортировки Шелла. Анализ этого алгоритма поставил несколько весьма трудных математических проблем, многие из которых так еще и не решены. В частности, не известно, какие расстояния дают наилучшие результаты. Известно, однако, что выбор расстояний должен быть таким, чтобы взаимодействие различных цепочек проходило как можно чаще. В работе Кнут показал, что имеет смысл использовать следующую
последовательность (она записана в обратном порядке): 1, 4, 13, 40, 121, ..., где $h_{k-1} = 3h_k + 1$, h_t (или $h_{конеч}$) = 1 и $t = log_3(n) - 1$. Он рекомендует и другую последовательность: 1, 3, 7, 15, 31, ... где $h_{k-1} = 2h_k + 1$, $h_t = 1$ и $t = log_2(n) - 1$. Математический анализ показывает, что в последнем случае для сортировки n элементов методом Шелла затраты пропорциональны $n^{1.2}$, что значительно лучше n^2, необходимых для методов прямой сортировки.

ЗАДАНИЕ

1. Реализовать блок-схемы алгоритмов следующих сортировок:

- Сортировка с помощью прямого включения;
- Сортировка с помощью прямого выбора;
- Пузырьковая сортировка;

- Шейкерная сортировка;
- Сортировка Шелла.

Задание необходимо оформить в виде отчета и сдать преподавателю.

2. Запрограммировать методы сортировок, указанные выше (см. задание 1).

**3. Сравнить эффективность реализованных алгоритмов по числу перестановок. (задание на дополнительные баллы)

Контрольные вопросы

1. Что такое сортировка?
2. Перечислите виды сортировок.
3. Перечислите *общие критерии* оценки алгоритмов сортировки.
4. Перечислите достоинства и недостатки изученных методов сортировок.

ЛАБОРАТОРНАЯ РАБОТА № 4
«СОРТИРОВКА ЭЛЕМЕНТОВ ДВУМЕРНОГО МАССИВА»

Цель работы: изучить особенности применения алгоритмов сортировок и перестановок в двумерных массивах, научиться решать задачи сортировок и перестановок в двумерных массивах на языке С.

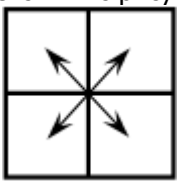
При выполнении лабораторной работы для каждого задания требуется написать программу на языке С, которая получает на входе числовые данные (в зависимости от постановки задачи), выполняет генерацию и *вывод* двумерного массива указанного типа. В каждой задаче необходимо выполнить обработку двумерного массива. Для этого необходимо разработать *алгоритм* (сортировок или перестановок в двумерных массивах) и реализовать его в виде отдельной функции. Ввод данных осуществляется с клавиатуры с учетом требований к входным данным, содержащихся в постановке задачи. Ограничениями на *входные данные* является *диапазон* используемого числового типа данных в языке С и максимально допустимый размер объявляемого двумерного массива.

Задания к лабораторной работе

1. Объявите двумерный вещественный массив, в котором ***n x m*** элементов. Заполните его числами, полученными по закономерности: $a_{ij} = \sum_{n=0}^{i} \frac{(i+1)(j+5)}{i+j+1}$. Отсортируйте каждую строку массива по убыванию. Распечатайте его в виде таблицы с точностью до 3 знаков после запятой дважды – до и после сортировки. Оформите генерацию, вывод массива и сортировку строк с помощью функций.
2. Объявите двумерный целочисленный массив, в котором ***n*** строк по ***m*** элементов. Выполните генерацию массива случайными целыми числами из промежутка ***[a;b).*** Переставьте столбцы массива так, чтобы их максимальные элементы образовали возрастающую последовательность. Выведите массив на экран в виде таблицы дважды – до и после перестановки. Оформите генерацию, вывод массива и перестановку столбцов с помощью функций.
3. Объявите двумерный вещественный массив, в котором ***n x m*** элементов. Выполните генерацию массива слу-

чайными *вещественными числами* из промежутка ***[a; b).*** Отсортируйте каждый столбец массива по возрастанию. Распечатайте его в виде таблицы с точностью до 2 знаков после запятой дважды – до и после сортировки. Оформите генерацию, вывод массива и сортировку столбцов с помощью функций.

4. Дана квадратная матрица размера ***2n x 2n***. Получите новую матрицу, переставляя ее блоки размера ***n x n*** в соответствии с рисунком.



5. Приведите квадратную целочисленную матрицу ***n x n*** к треугольному виду. Способ генерации матрицы выберите самостоятельно.

Указания к выполнению работы

Каждое задание необходимо решить в соответствии с изученными методами объявления, генерации и вывода двумерных массивов в языке С. Обработку данных необходимо выполнить, используя алгоритмы сортировок или перестановок данных в двумерных массивах.

Следует реализовать каждое задание в соответствии с приведенными этапами:

- изучить словесную постановку задачи, выделив при этом все виды данных;
- сформулировать математическую постановку задачи;
- выбрать метод решения задачи, если это необходимо;
- разработать графическую схему алгоритма;
- записать разработанный алгоритм на языке С;
- отладить программу.

Контрольные вопросы

1. В чем принципиальное отличие задач сортировок двумерных и одномерных массивов?
2. Каким образом оформляется прототип функции, чтобы изменения, выполненные с элементами массива, были сохранены после завершения работы функции?

3. Приведите возможные обращения к элементу трехмерного массива, аналогичные обращению mas[i][j][k].

4. В чем причина неудобства использования массивов слишком больших измерений в программах?

5. При решении каких прикладных задач используются многомерные массивы? Отдельно приведите примеры для массивов с измерением два и более.

ЛАБОРАТОРНАЯ РАБОТА № 5 «ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ. ЛИНЕЙНЫЙ СПИСОК. ОДНОНАПРАВЛЕННЫЙ И ДВУНАПРАВЛЕННЫЙ. СОРТИРОВКА НА ОСНОВЕ ЛИНЕЙНЫХ СПИСКОВ»

Цель: изучить понятия, классификацию и объявление списков, особенности доступа к данным и работу с памятью при использовании однонаправленных и *двунаправленных* списков, научиться решать задачи с использованием списков на языке С.

Теоретический материал

Понятие списка хорошо известно из жизненных примеров: *список* студентов учебной группы, *список* призёров олимпиады, *список*(перечень) документов для представления в приёмную комиссию, *список* почтовой рассылки, *список* литературы для самостоятельного чтения и т.п.

Списком называется упорядоченное множество, состоящее из переменного числа элементов, к которым применимы *операции включения*, исключения. *Список*, отражающий отношения соседства между элементами, называется линейным.

Длина списка равна числу элементов, содержащихся в списке, *список* нулевой длины называется пустым списком. Списки представляют собой способ организации структуры данных, при которой элементы некоторого типа образуют цепочку. Для связывания элементов в списке используют систему указателей. В минимальном случае, любой элемент линейного списка имеет один *указатель*, который указывает на следующий элемент в списке или является пустым указателем, что интерпретируется как конец списка.

Структура, элементами которой служат записи с одним и тем же форматом, связанные друг с другом с помощью указателей, хранящихся в самих элементах, называют связанным списком. В связанном списке элементы линейно упорядочены, но порядок определяется не номерами, как в массиве, а указателями, входящими в состав элементов списка. Каждый *список* имеет особый элемент, называемый указателем начала списка (головой списка), который обычно по содержанию отличен от остальных элементов. В *поле* указателя *последнего элемента списка* находится специальный признак NULL, свидетельствующий о

конце списка.

Линейные связные списки являются простейшими *динамическими структурами данных*. Из всего многообразия связанных списков можно выделить следующие основные:

- однонаправленные (односвязные) списки;
- *двунаправленные* (*двусвязные*) списки;
- циклические (кольцевые) списки.

В основном они отличаются видом взаимосвязи элементов и/или допустимыми операциями.

Однонаправленные (односвязные) списки

Наиболее простой динамической структурой является однонаправленный *список*, элементами которого служат объекты *структурного типа*.

Однонаправленный (односвязный) список – это *структура данных*, представляющая собой последовательность элементов, в каждом из которых хранится *значение* и *указатель* на следующий *элемент списка* (рис.1). В последнем элементе *указатель* на следующий элемент равен NULL.

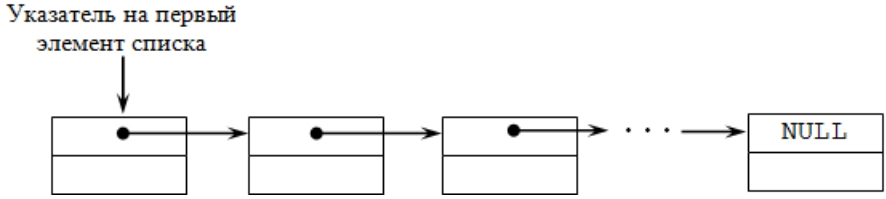


Рис. 1. Линейный однонаправленный список

Описание простейшего элемента такого списка выглядит следующим образом:

struct имя_типа { информационное *поле*; адресное *поле*; };

где информационное поле – это *поле* любого, ранее объявленного или стандартного, типа;

адресное поле – это *указатель* на *объект* того же типа, что и определяемая структура, в него записывается *адрес* следующего элемента списка.

Например:

```
struct Node {
      int key;//информационное поле
      Node*next;//адресное поле
     };
```

Информационных полей может быть несколько.

Например:

```
struct point {
    char*name;//информационное поле
    int age;//информационное поле
    point*next;//адресное поле
    };
```

Каждый *элемент списка* содержит *ключ*, который идентифицирует этот элемент. *Ключ* обычно бывает либо целым числом, либо строкой.

Основными операциями, осуществляемыми с однонаправленными списками, являются:

- создание списка;
- печать (просмотр) списка;
- вставка элемента в список;
- удаление элемента из списка;
- поиск элемента в списке
- проверка пустоты списка;
- удаление списка.

Двунаправленные (двусвязные) списки

Для ускорения многих операций целесообразно применять переходы между элементами списка в обоих направлениях. Это реализуется с помощью *двунаправленных* списков, которые являются сложной динамической структурой.

Двунаправленный (двусвязный) список – это *структура данных*, состоящая из последовательности элементов, каждый из которых содержит информационную часть и два указателя на соседние элементы (рис. 2). При этом два соседних элемента должны содержать взаимные ссылки друг на друга.

В таком списке каждый элемент (кроме первого и последнего) связан с предыдущим и следующим за ним элементами. Каждый элемент *двунаправленного* списка имеет два поля с указателями: одно *поле* содержит ссылку на следующий элемент, другое *поле*– ссылку на предыдущий элемент и третье *поле* – информационное. Наличие ссылок на следующее звено и на предыдущее позволяет двигаться по списку от каждого звена в любом направлении: от звена к концу списка или от звена к началу списка, поэтому такой *список* называют двунаправленным.

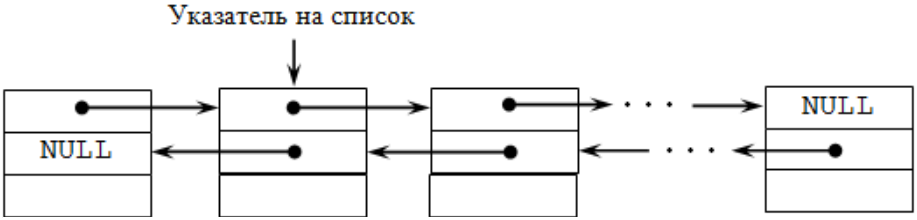


Рис. 2 Двунаправленный список

Описание простейшего элемента такого списка выглядит следующим образом:

```
struct имя_типа {
        информационное поле;
        адресное поле 1;
        адресное поле 2;
       };
```

где информационное поле – это *поле* любого, ранее объявленного или стандартного, типа;

адресное поле 1 – это *указатель* на *объект* того же типа, что и определяемая структура, в него записывается *адрес* следующего элемента списка ;

адресное поле 2 – это *указатель* на *объект* того же типа, что и определяемая структура, в него записывается *адрес* предыдущего элемента списка.

Например:

```
struct list {
      type elem ;
      list *next, *pred ;
      }
list *headlist ;
```

где type – тип информационного поля элемента списка;

*next, *pred – указатели на следующий и предыдущий элементы этой структуры соответственно.

Переменная-указатель headlist задает *список* как единый программный *объект*, ее *значение* – *указатель* на первый (или заглавный) *элемент списка*.

Основные *операции*, выполняемые над двунаправленным списком, те же, что и для однонаправленного списка. Так как *двунаправленный список* более гибкий, чем однонаправленный, то при включении элемента в *список*, нужно использовать *указатель* как на элемент, за которым происходит включение, так и *указатель* на элемент, перед которым происходит включение.

При исключении элемента из списка нужно использовать как *указатель* на сам исключаемый элемент, так и указатели на предшествующий или следующий за исключаемым элементы. Но так как элемент *двунаправленного* списка имеет два указателя, то при выполнении *операций включения*/исключения элемента надо изменять больше связей, чем в однонаправленном списке.

Рассмотрим основные *операции*, осуществляемые с двунаправленными списками, такие как:

- создание списка;
- печать (просмотр) списка;
- вставка элемента в список;
- удаление элемента из списка;
- поиск элемента в списке;
- проверка пустоты списка;
- удаление списка.

ЗАДАНИЕ

I. **Создание простого упорядоченного по невозрастанию списка неотрицательных целых чисел** (Например, 12->10->9->7->3->NULL)

1. Для описания узлов списка создать структуру Node с двумя полями:
   а) значение элемента списка (целое число),
   б) указатель на следующий узел.
2. С помощью typedef определить обозначения NODE и pNODE соответственно для узла и указателя на него.
3. Для описания списка создать структурный тип List с двумя полями:
   а) begin - указатель на первый элемент списка;
   б) len - количество элементов в списке.
4. Определить синонимы LIST и pLIST для типов список и указатель на список.
5. Написать функцию создания нового (пустого) списка
   pLIST createList(void);

При успешном создании списка функция возвращает указатель на созданную структуру типа LIST, иначе NULL. Поля созданной структуры обнуляются.

6. Написать функцию-предикат
   int isEmpty(pLIST pL);
   используемую для определения, список пуст (1) или нет (0).

7. Написать функцию

pNODE getPointer(pLIST pL, int date);

для поиска в упорядоченном по невозрастанию списке места для вставки узла со значением date.

Функция возвращает указатель на предшествующий найденному месту узел.

Если в список пуст - возвращается NULL.

Для узлов, стоящих па первом и втором местах, возвращается pL->begin.

8. Написать функцию

int addNodeAfter(pLIST pL, pNODE pN,int newdate);

для добавления в список нового элемента newdate после узла, на который

указывает pN, исключение составляют случаи,

1) когда pN == pL->begin!=NULL;
2) когда pN ==NULL.

При успешном добавлении возвращается 1, в противном случае - 0.

9. Написать функцию

pNODE findNode(pLIST pL, int date);

для поиска в списке узла со значением date. Функция возвращает

указатель на предшнествующий найденному узел. Если в списке нет элемента со значением date - возвращается NULL. Для узлов, стоящих па первом и втором местах, возвр pL->begin. Функция используется для удаления узла.

10. Написать функцию

int delNode(pLIST pL, pNODE pN);

для удаления из списка элемента, на который указывает pN.

11. Написать функцию

void clearList(pLIST pL);

для удаления всех узлов списка (очистки списка).

12. Написать функцию

void showSList(pLIST pL);

которая последовательно просматривает весь список и для каждого узла выводит на экран его адрес, находящееся число и адрес следующего узла. В случае пустого списка выводится соответствующее сообщение.

13. Написать функцию

void deleteList(pLIST pL);

для удаления списка.

14. В функции main

а) создать пустой список;

б) работа со списком осуществляется в режиме диалога:

вывести меню для выбора одной из возможных операций

-добавления элемента в список (список должен быть упорядочен по значениям целых чисел!);

-поиск числа в списке;

-удаление числа из списка;

-просмотр списка;

-очистка списка;

-конец работы.

Контрольные вопросы

1. Любой ли список является связным? Обоснуйте ответ.
2. В чем отличие первого элемента однонаправленного (*двунаправленного*) списка от остальных элементов этого же списка?
3. В чем отличие последнего элемента однонаправленного (*двунаправленного*) списка от остальных элементов этого же списка?
4. В чем принципиальные отличия выполнения основных операций в однонаправленных и *двунаправленных* списках?

ЛАБОРАТОРНАЯ РАБОТА № 6 «ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ. СТЕК И ОЧЕРЕДЬ»

Цель работы: изучить понятия, объявления, особенности доступа к данным и работы с памятью в стеках и очередях, научиться решать задачи с использованием стеков и очередей в языке С.

Краткие теоретические сведения

Стеки

В списках *доступ* к элементам происходит посредством адресации, при этом *доступ* к отдельным элементам не ограничен. Но существуют также и такие *списковые структуры* данных, в которых имеются *ограничения доступа* к элементам. Одним из представителей таких *списковых структур* является стековый *список* или просто *стек*.

Стек (англ. stack – стопка) – это *структура данных*, в которой новый элемент всегда записывается в ее начало (вершину) и очередной читаемый элемент также всегда выбирается из ее начала (рис. 1). В стеках используется *метод доступа* к элементам LIFO (Last Input – First Output, "последним пришел – первым вышел"). Чаще всего принцип работы стека сравнивают со стопкой тарелок: чтобы взять вторую сверху, нужно сначала взять верхнюю.

Стек – это *список*, у которого доступен один элемент (одна позиция). Этот элемент называется вершиной стека. Взять элемент можно только из *вершины стека*, добавить элемент можно только в *вершину стека*. Например, если записаны в *стек* числа 1, 2, 3, то при последующем извлечении получим 3,2,1.

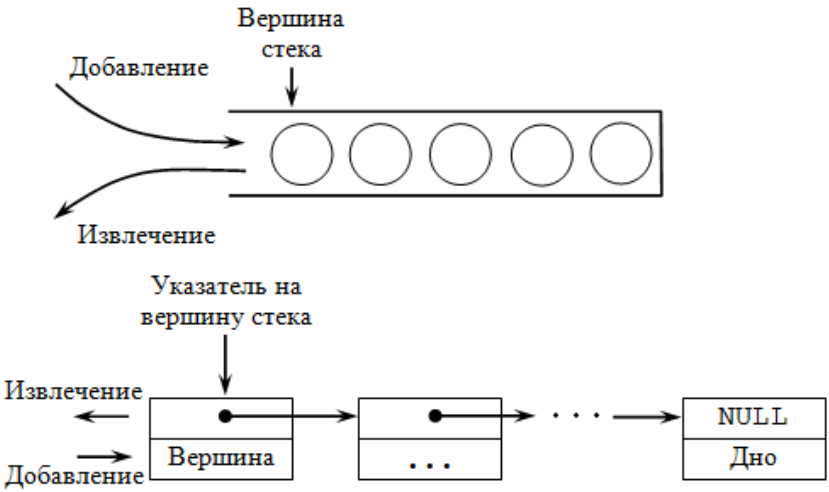


Рис. 1. Стек и его организация

Описание стека выглядит следующим образом:

```
struct имя_типа {
        информационное поле;
        адресное поле;
       };
```

где информационное поле – это *поле* любого ранее объявленного или стандартного типа;

адресное поле – это *указатель* на *объект* того же типа, что и определяемая структура, в него записывается *адрес* следующего элемента стека.

Например:

```
struct list {
      type pole1;
      list *pole2;
     } stack;
```

Стек как *динамическую структуру данных* легко организовать на основе линейного списка. Поскольку работа всегда идет с заголовком стека, то есть не требуется осуществлять просмотр элементов, удаление и вставку элементов в середину или конец списка, то достаточно использовать экономичный по памяти линейный однонаправленный *список*. Для такого списка достаточно хранить *указатель вершины стека*, который указывает на первый *элемент списка*. Если *стек* пуст, то списка не существует, и *указатель* принимает *значение* NULL.

Описание элементов сте- ка аналогично описанию

ментов линейного однонаправленного списка. Поэтому объявим *стек* через объявление линейного однонаправленного списка:

```
struct Stack {
    Single_List *Top;//вершина стека
    };
. . . . . .
Stack *Top_Stack;//указатель на вершину стека
```

Основные *операции*, производимые со стеком:

- создание стека;
- печать (просмотр) стека;
- добавление элемента в *вершину стека*;
- извлечение элемента из *вершины стека*;
- проверка пустоты стека;
- очистка стека

Задание 1. СОЗДАНИЕ СТЕКА НА ОСНОВЕ ЛИНЕЙНОГО СПИСКА

1. Для описания узлов стека создать структуру Node с двумя полями:

а) значение элемента стека (символьный тип),

б) указатель на следующий узел.

2. С помощью typedef определить обозначения NODE и pNODE соответственно для узла и указателя на него.

3. Для описания стека создать структурный тип Stack с двумя полями:

а) begin - указатель на первый элемент списка;

б) len - количество элементов в списке.

4. Определить синонимы STACK и pSTACK для типов список и указатель на список.

5. Написать функцию создания нового (пустого) списка

pSTACK createStack(void);

При успешном создании списка функция возвращает указатель на созданную структуру типа STACK, иначе NULL.

Поля созданной структуры обнуляются.

6. Написать функцию-предикат

int isEmpty(pLIST pL);

используемую для определения, список пуст (1) или нет (0).

7. Написать функцию для добавления нового элемента в стек.

void push (pSTACK pS, char ch)

8. Написать функцию для извлечения элемента из стека

char pop (pSTACK pS)

9. В функции main

Разработать программу работы с символьным стеком, реализующую операции добавления и удаления элементов из стека и отображения текущего состояния стека.

Задание 2. Реализовать стек списком. Все стандартные операции со стеком должны быть оформлены отдельными подпрограммами.

1. Создать стек из случайных целых чисел, лежащих в диапазоне –50 до +50 и преобразовать его в два стека. Первый должен содержать только положительные числа, а второй – только отрицательные. Порядок следования чисел должен быть сохранен.
2. Создать стек из случайных целых чисел и удалить из него записи с четными числами.
3. Создать стек из случайных целых чисел и поменять местами крайние элементы.
4. Создать стек из случайных целых чисел и удалить из него каждый второй элемент.

Очереди

Очередь – это *структура данных*, представляющая собой последовательность элементов, образованная в порядке их поступления. Каждый новый элемент размещается в конце очереди; элемент, стоящий в начале очереди, выбирается из нее первым. В очереди используется принцип доступа к элементам FIFO (First Input – First Output, "первый пришёл – первый вышел") (рис.2). В очереди доступны два элемента (две позиции): начало очереди и конец очереди. Поместить элемент можно только в конец очереди, а взять элемент только из ее начала. Примером может служить обыкновенная *очередь* в магазине.

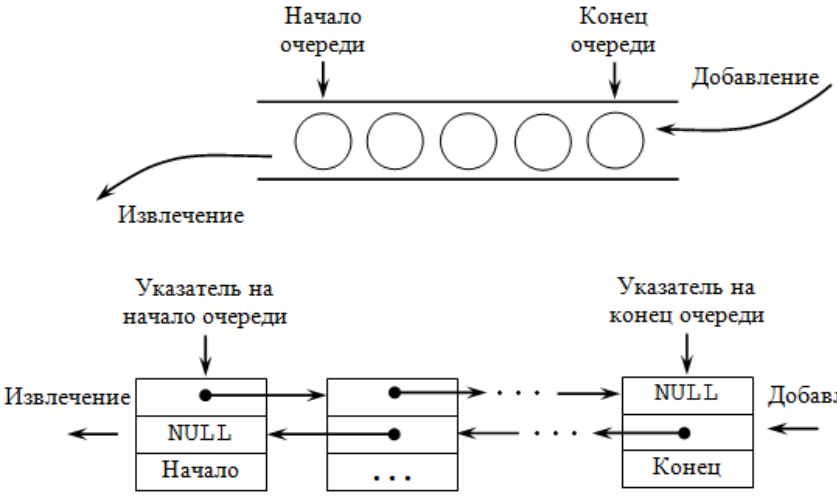


Рис. 2. Очередь и ее организация

Описание очереди выглядит следующим образом:

```
struct имя_типа {
                информационное поле;
                адресное поле1;
                адресное поле2;
               };
```

где информационное поле – это *поле* любого, ранее объявленного или стандартного, типа;

адресное поле1, адресное поле2 – это указатели на объекты того же типа, что и определяемая структура, в них записываются адреса первого и следующего элементов очереди.

Например:

1 способ: адресное *поле* ссылается на объявляемую структуру.

```
struct list2 {
            type  pole1;
            list2 *pole1, *pole2;
           }
```

2 способ: адресное *поле* ссылается на ранее объявленную структуру.

```
struct list1 {
            type  pole1;
            list1  *pole2;
           }
struct ch3 {
```

```
        list1 *beg, *next ;
      }
```

Очередь как *динамическую структуру данных* легко организовать на основе линейного списка. Поскольку работа идет с обоими концами очереди, то предпочтительно будет использовать линейный двунаправленный *список*. Хотя для работы с таким списком достаточно иметь один *указатель* на любой *элемент списка*, здесь целесообразно хранить два указателя – один на начало списка (откуда извлекаем элементы) и один на конец списка (куда добавляем элементы). Если *очередь* пуста, то списка не существует, и указатели принимают *значение* NULL.

Описание элементов очереди аналогично описанию элементов линейного *двунаправленного* списка. Поэтому объявим *очередь* через объявление линейного *двунаправленного* списка:

```
struct Queue {
        Double_List *Begin;//начало очереди
        Double_List *End; //конец очереди
       };
. . . . . .
Queue *My_Queue;//указатель на очередь
```

Основные *операции*, производимые с очередью:

- создание очереди;
- печать (просмотр) очереди;
- добавление элемента в конец очереди;
- извлечение элемента из начала очереди;
- проверка пустоты очереди;
- очистка очереди.

Задание 3

Опишите очередь с вещественным информационным полем, и заполните ее элементами с клавиатуры. Выполните циклический сдвиг элементов в очереди так, чтобы в ее начале был расположен наибольший элемент.

Контрольные вопросы

1. В чем преимущества и недостатки организации структур в виде стека?
2. В чем преимущества и недостатки организации структур в виде очереди?
3. Для моделирования каких реальных задач удобно использовать стек? А для каких очередь?

4. Какое значение хранит указатель на стек?
5. Какое значение хранит указатель на очередь?
6. С какой целью в программах выполняется проверка на пустоту стека и очереди?

ЛАБОРАТОРНАЯ РАБОТА №7 «ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ. БИНАРНЫЕ ДЕРЕВЬЯ»

Цель работы: изучить понятие, формирование, особенности доступа к данным и работы с памятью в бинарных деревьях, научиться решать задачи с использованием *рекурсивных функций* и алгоритмов обхода бинарных деревьев в языке С.

Теоретический материал

Дерево является одним из важнейших и интересных частных случаев графа. *Древовидная модель* оказывается довольно эффективной для представления динамических данных с целью быстрого *поиска информации*.

Деревья являются одними из наиболее широко распространенных структур данных в *информатике* и программировании, которые представляют собой иерархические структуры в виде набора связанных узлов.

Дерево – это *структура данных*, представляющая собой совокупность элементов и отношений, образующих иерархическую структуру этих элементов (рис.1). Каждый элемент дерева называется вершиной (узлом) дерева. Вершины дерева соединены направленными дугами, которые называют ветвями дерева. Начальный узел дерева называют корнем дерева, ему соответствует нулевой уровень. Листьями дерева называют вершины, в которые входит одна *ветвь* и не выходит ни одной ветви.

Каждое *дерево* обладает следующими свойствами:

1. существует узел, в который не входит ни одной дуги (корень);
2. в каждую вершину, кроме корня, входит одна дуга.

Деревья особенно часто используют на практике при изображении различных иерархий. Например, популярны генеалогические деревья.

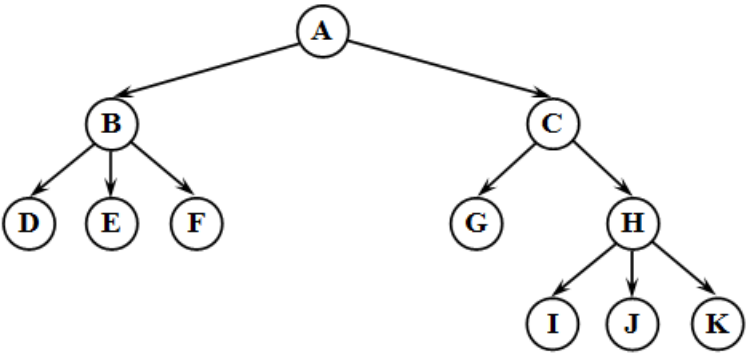


Рис. 1. Дерево

Все вершины, в которые входят ветви, исходящие из одной общей вершины, называются потомками, а сама *вершина* – предком. Для каждого предка может быть выделено несколько. Уровень потомка на единицу превосходит уровень его предка. *Корень дерева* не имеет предка, а *листья дерева* не имеют потомков.

Высота (глубина) дерева определяется количеством уровней, на которых располагаются его вершины. *Высота* пустого дерева равна нулю, *высота дерева* из одного корня – единице. На первом уровне дерева может быть только одна *вершина* – *корень дерева*, на втором – потомки корня дерева, на третьем – потомки потомков корня дерева и т.д.

Поддерево – часть древообразной структуры данных, которая может быть представлена в виде отдельного дерева.

Степенью вершины в дереве называется количество дуг, которое из нее выходит. Степень дерева равна *максимальной степени* вершины, входящей в *дерево*. При этом листьями в дереве являются вершины, имеющие степень нуль. *По* величине степени дерева различают два типа деревьев:

- двоичные – степень дерева не более двух;
- сильноветвящиеся – степень дерева произвольная.

Упорядоченное дерево – это *дерево*, у которого ветви, исходящие из каждой вершины, упорядочены *по* определенному критерию.

Деревья являются рекурсивными структурами, так как каждое *поддерево* также является деревом. Таким образом, *дерево* можно определить как рекурсивную структуру, в которой каждый элемент является:

- либо пустой структурой;

- либо элементом, с которым связано конечное число поддеревьев.

Действия с рекурсивными структурами удобнее всего описываются с помощью рекурсивных алгоритмов.

Списочное *представление деревьев* основано на элементах, соответствующих вершинам дерева. Каждый элемент имеет *поле*данных и два поля указателей: *указатель* на начало списка потомков вершины и *указатель* на следующий элемент в списке потомков текущего уровня. При таком способе представления дерева обязательно следует сохранять *указатель* на вершину, являющуюся *корнем дерева*.

Для того, чтобы выполнить определенную операцию над всеми вершинами дерева необходимо все его вершины просмотреть. Такая задача называется обходом дерева.

Обход дерева – это упорядоченная последовательность вершин дерева, в которой каждая *вершина* встречается только один раз.

При обходе все вершины дерева должны посещаться в определенном порядке. Существует несколько способов обхода всех вершин дерева. Выделим три наиболее часто используемых способа *обхода дерева* (рис. 2):

- прямой;
- симметричный;
- обратный.

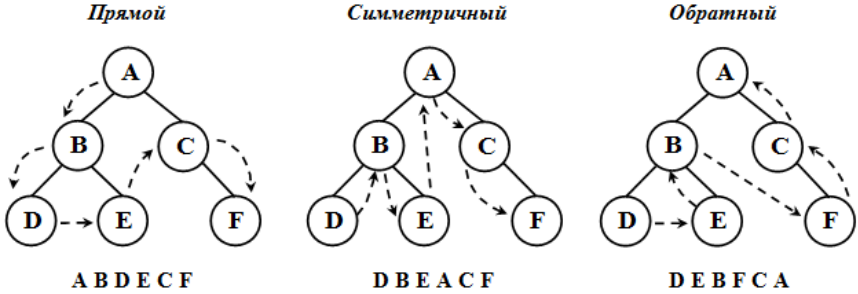


Рис. 2. Обходы деревьев

Существует большое многообразие древовидных структур данных. Выделим самые распространенные из них: бинарные (двоичные) деревья, красно-черные деревья, B-деревья, *АВЛ-деревья*, матричные деревья, смешанные деревья и т.д.

Бинарные деревья

Бинарные деревья являются деревьями со степенью не более двух.

Бинарное (двоичное) дерево – это *динамическая структура данных*, представляющее собой *дерево*, в котором каждая *вершина* имеет не более двух потомков (рис.3). Таким образом, *бинарное дерево* состоит из элементов, каждый из которых содержит информационное *поле* и не более двух ссылок на различные бинарные поддеревья. На каждый элемент дерева имеется ровно одна *ссылка*.

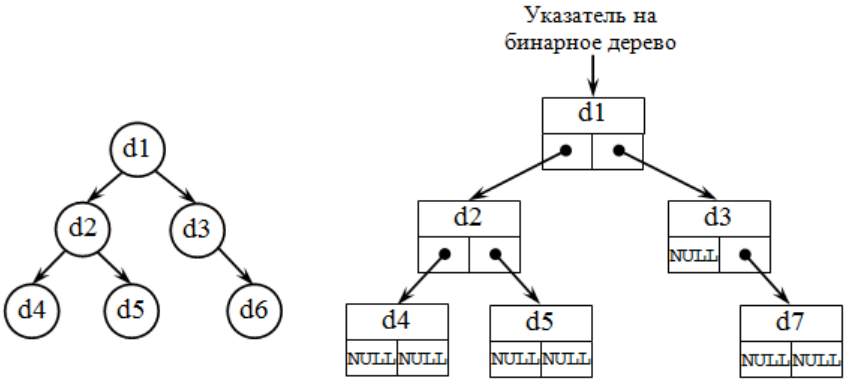


Рис. 3. Бинарное дерево и его организация

Каждая *вершина бинарного дерева* является структурой, состоящей из четырех видов полей. Содержимым этих полей будут соответственно:

- информационное поле (ключ вершины);
- служебное поле (их может быть несколько или ни одного);
- указатель на *левое поддерево*;
- указатель на правое *поддерево*.

По степени вершин бинарные деревья делятся на (рис. 4):

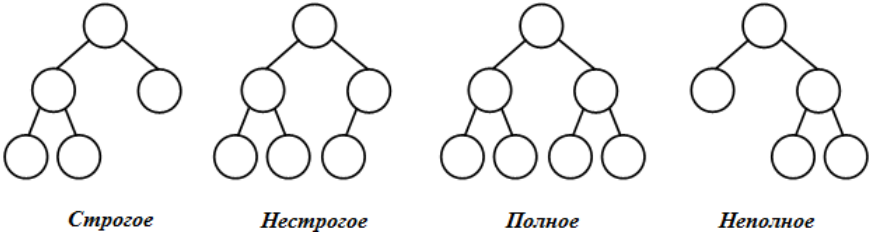


Рис. 4.

- строгие – вершины дерева имеют степень ноль (у листьев) или два (у узлов);
- нестрогие – вершины дерева имеют степень ноль

(у листьев), один или два (у узлов).

В общем случае у *бинарного дерева* на k -м уровне может быть до 2^{k-1} вершин. *Бинарное дерево* называется полным, если оно содержит только полностью заполненные уровни. В противном случае оно является неполным.

Дерево называется сбалансированным, если длины всех путей от корня к внешним вершинам равны между собой. *Дерево* называется почти сбалансированным, если длины всевозможных путей от корня к внешним вершинам отличаются не более, чем на единицу.

Бинарное дерево может представлять собой *пустое множество*. *Бинарное дерево* может выродиться в *список* (рис. 5).

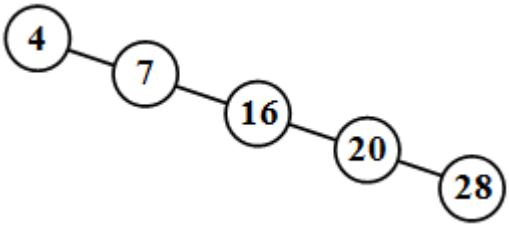


Рис. 5. Список как частный случай бинарного дерева

Структура дерева отражается во входном потоке данных так: каждой вводимой пустой связи соответствует условный символ, например, '*' (звездочка). При этом сначала описываются левые потомки, затем, правые. Для структуры *бинарного дерева*, представленного на следующем рисунке 6, *входной поток* имеет вид: ABD*G***CE**FH**J**.

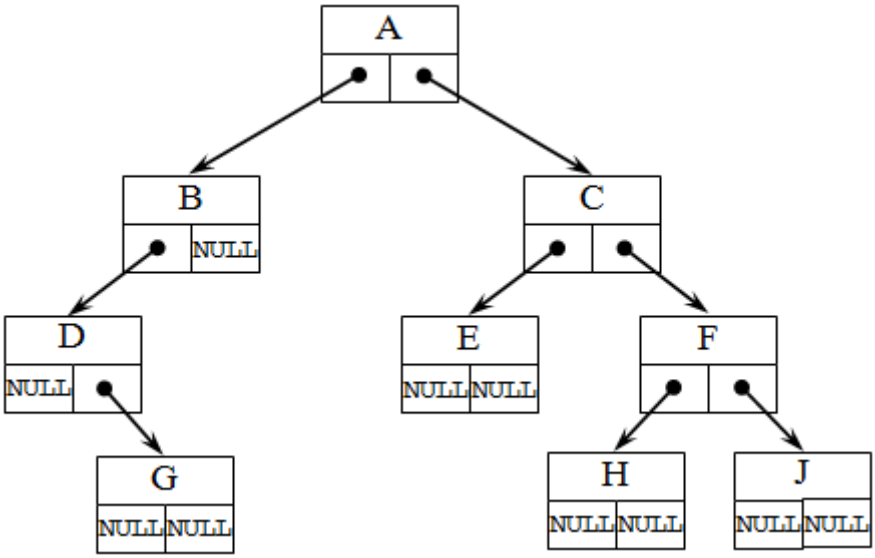


Рис. 6. Адресация в бинарном дереве

Бинарные деревья могут применяться для поиска данных в специально построенных деревьях (*базы данных*), сортировки данных, вычислений *арифметических выражений*, кодирования (метод Хаффмана) и т.д.

Описание *бинарного дерева* выглядит следующим образом:

```
struct имя_типа {
                информационное поле;
                [служебное поле;]
                адрес левого поддерева;
                адрес правого поддерева;
               };
```

где информационное поле – это *поле* любого ранее объявленного или стандартного типа;

адрес левого (правого) *поддерева* – это *указатель* на *объект* того же типа, что и определяемая структура, в него записывается *адрес* следующего элемента левого (правого) *поддерева*.

Например:

```
struct point {
        int data;//информационное поле
        int count; //служебное поле
        point *left;//адрес левого поддерева
        point *right;//адрес правого поддере-
```

ва

```
};
```

Основными операциями, осуществляемыми с бинарными деревьями, являются:

- создание *бинарного дерева*;
- печать *бинарного дерева*;
- обход *бинарного дерева*;
- вставка элемента в *бинарное дерево*;
- удаление элемента из *бинарного дерева*;
- проверка пустоты *бинарного дерева*;
- удаление *бинарного дерева*.

Для описания алгоритмов этих основных операций используется следующее объявление:

```
struct BinaryTree{
    int Data; //поле данных
    BinaryTree* Left; //указатель на левый потомок
    BinaryTree* Right; /указатель на правый потомок
};
. . . . . .
BinaryTree* BTree = NULL;
```

Задания к лабораторной работе

При выполнении лабораторной работы для каждого задания требуется написать программу на языке С, в которой выполнено формирование бинарных деревьев в соответствии с постановкой задачи, ввод данных элементов деревьев с учетом типа информационного поля, их обработка и *вывод* на экран в указанном формате. Для хранения данных бинарных деревьев следует использовать ресурсы динамической памяти. Ввод данных осуществляется с клавиатуры с учетом требований к входным данным, содержащихся в постановке задачи. Ограничениями на *входные данные* являются максимальный размер строковых данных, диапазоны числовых типов полей структуры и допустимый размер области динамической памяти в языке С.

Выполните приведенные ниже задания.

1. Реализуйте программу, в которой выполняются все основные операции с бинарным деревом.
2. Найдите количество четных элементов *бинарного дерева*. Укажите эти элементы и их уровни.
3. Найдите сумму элементов *сбалансированного дерева*, находящихся на уровне k.

4. Оператор мобильной связи организовал базу данных *абонентов*, содержащую сведения о телефонах, их владельцах и используемых тарифах, в виде *бинарного дерева*. Составьте программу, которая:

- обеспечивает начальное формирование базы данных в виде *бинарного дерева*;
- производит вывод всей базы данных;
- производит поиск владельца по номеру телефона;
- выводит наиболее востребованный тариф (по наибольшему числу *абонентов*).

Указания к выполнению работы

Выполнение работы следует начать с решения задачи 1, реализовав алгоритмы основных операций над бинарным деревом. Каждое из заданий 2, 3 и 4 необходимо решить в соответствии с изученными методами и реализованными алгоритмами формирования, вывода и обработки данных бинарных деревьев в языке С. Обработку бинарных деревьев следует выполнить на основе базовых алгоритмов: *поиск по* дереву, вставка элемента в *дерево*, балансировка дерева, *удаление элемента* из дерева, удаление всего дерева.

Контрольные вопросы

1. С чем связана популярность использования деревьев в программировании?
2. Можно ли список отнести к деревьям? Ответ обоснуйте.
3. Какие данные содержат адресные поля элемента *бинарного дерева*?
4. Может ли *бинарное дерево* быть строгим и неполным? Ответ обоснуйте.
5. Может ли *бинарное дерево* быть нестрогим и полным? Ответ обоснуйте.
6. Каким может быть почти сбалансированное *бинарное дерево*: полным, неполным, строгим, нестрогим? Ответ обоснуйте.
7. Куда может быть добавлен элемент в *бинарное дерево* в зависимости от его вида (полное, неполное, строгое, нестрогое)? Вид дерева при этом должен сохраниться.
8. Куда может быть добавлен элемент в сбалансированное *бинарное дерево*? Вид дерева при этом должен со-

храниться.

9. Чем отличаются, с точки зрения реализации алгоритма, прямой, симметричный и *обратный обходы бинарного дерева*?

ЛАБОРАТОРНАЯ РАБОТА №8
«АЛГОРИТМЫ ХЕШИРОВАНИЯ ДАННЫХ»

Цель работы: изучить построение функции *хеширования* и алгоритмов *хеширования* данных и научиться разрабатывать алгоритмы открытого и закрытого *хеширования* при решении задач на языке С.

Теоретический материал

Процесс поиска данных в больших объемах информации сопряжен с временными затратами, которые обусловлены необходимостью просмотра и сравнения с ключом поиска значительного числа элементов. Сокращение поиска возможно осуществить путем *локализации* области просмотра. Например, отсортировать данные по ключу поиска, разбить на непересекающиеся блоки по некоторому групповому признаку или поставить в соответствие реальным данным некий код, который упростит процедуру поиска.

В настоящее время используется широко распространенный метод обеспечения быстрого доступа к информации, хранящейся во внешней памяти – *хеширование*.

Хеширование (или хэширование, англ. hashing) – это преобразование входного массива данных определенного типа и произвольной длины в выходную битовую строку фиксированной длины. Такие преобразования также называются хеш-функциямиили функциями *свертки*, а их результаты называют хешем, хеш-кодом, хеш-таблицей или дайджестом сообщения (англ. message digest).

Хеш-таблица – это *структура данных*, реализующая *интерфейс* ассоциативного массива, то есть она позволяет хранить пары вида "*ключ- значение*" и выполнять три *операции*: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу. Хеш-таблица является массивом, формируемым в определенном порядке *хеш-функцией*.

Принято считать, что хорошей, с точки зрения практического применения, является такая *хеш-функция*, которая удовлетворяет следующим условиям:

- функция должна быть простой с вычислительной точки зрения;
- функция должна распределять ключи в хеш-таблице наиболее равномерно;
- функция не должна отображать какую-либо связь

между значениями ключей в связь между значениями адресов;

- функция должна минимизировать число *коллизий* – то есть ситуаций, когда разным ключам соответствует одно значение *хеш-функции* (ключи в этом случае называются синонимами).

При этом первое свойство хорошей *хеш-функции* зависит от характеристик компьютера, а второе – от значений данных.

Если бы все данные были случайными, то *хеш-функции* были бы очень простые (например, несколько битов ключа). Однако на практике случайные данные встречаются достаточно редко, и приходится создавать функцию, которая зависела бы от всего ключа. Если *хеш-функция* распределяет совокупность *возможных ключей* равномерно по множеству индексов, то *хеширование* эффективно разбивает множество ключей. Наихудший случай – когда все ключи хешируются в один *индекс*.

При возникновении *коллизий* необходимо найти новое *место* для хранения ключей, претендующих на одну и ту же ячейку хеш-таблицы. Причем, если *коллизии* допускаются, то их количество необходимо минимизировать. В некоторых специальных случаях удается избежать *коллизий* вообще. Например, если все ключи элементов известны заранее (или очень редко меняются), то для них можно найти некоторую инъективную хеш-функцию, которая распределит их по ячейкам хеш-таблицы без *коллизий*. Хеш-таблицы, использующие подобные *хеш-функции*, не нуждаются в механизме разрешения *коллизий*, и называются хеш-таблицами с прямой адресацией.

Хеш-таблицы должны соответствовать следующим свойствам.

- Выполнение операции в хеш-таблице начинается с вычисления *хеш-функции* от ключа. Получающееся хеш-значение является индексом в исходном массиве.
- Количество хранимых элементов массива, деленное на число возможных значений *хеш-функции*, называется коэффициентом заполнения хеш-таблицы (load factor) и является важным параметром, от которого зависит среднее время выполнения операций.
- Операции поиска, вставки и удаления должны выполняться в среднем за время O(1). Однако при такой оценке не учитываются возможные аппаратные затраты на перестройку индекса хеш-таблицы, связанную с увеличением значения размера массива и добавлением в хеш-таблицу новой пары.
- Механизм разрешения *коллизий* является важной состав-

ляющей любой хеш-таблицы.

Хеширование полезно, когда широкий *диапазон* возможных значений должен быть сохранен в малом объеме памяти, и нужен способ быстрого, практически произвольного доступа. *Хэш-таблицы* часто применяются в базах данных, и, особенно, в *языковых процессорах* типа компиляторов и *ассемблеров*, где они повышают скорость обработки таблицы идентификаторов. В качестве использования *хеширования* в повседневной жизни можно привести примеры распределение книг в библиотеке по тематическим каталогам, упорядочивание в словарях по первым буквам слов, *шифрование* специальностей в вузах и т.д.

Алгоритмы хеширования

Существует несколько типов функций *хеширования*, каждая из которых имеет свои преимущества и недостатки и основана на представлении данных. Приведем обзор и *анализ* некоторых наиболее простых из применяемых на практике хеш-функций.

Таблица прямого доступа

Простейшей организацией таблицы, обеспечивающей идеально быстрый поиск, является таблица прямого доступа. В такой таблице ключ является адресом записи в таблице или может быть преобразован в адрес, причем таким образом, что никакие два разных ключа не преобразуются в один и тот же адрес. При *создании таблицы* выделяется память для хранения всей таблицы и заполняется пустыми записями. Затем записи вносятся в таблицу – каждая на свое место, определяемое ее ключом. При поиске ключ используется как адрес и по этому адресу выбирается запись. Если выбранная запись пустая, то записи с таким ключом вообще нет в таблице. Таблицы прямого доступа очень эффективны в использовании, но, к сожалению, область их применения весьма ограничена.

Назовем пространством ключей множество всех теоретически возможных значений ключей записи. Назовем пространством записей множество тех ячеек памяти, которые выделяются для хранения таблицы. Таблицы прямого доступа применимы только для таких задач, в которых размер пространства записей может быть равен размеру пространства ключей. В большинстве реальных задач размер пространства записей много меньше, чем пространства ключей. Так, если в качестве ключа используется фамилия, то, даже ограничив *длину ключа* десятью символами кириллицы, получаем 3310 возможных значений ключей. Даже если ресурсы *вычислительной системы* и позволят выделить простран-

ство записей такого размера, то значительная часть этого пространства будет заполнена пустыми записями, так как в каждом конкретном заполнении таблицы фактическое множество ключей не будет полностью покрывать пространство ключей.

В целях экономии памяти можно назначать размер пространства записей равным размеру фактического множества записей или превосходящим его незначительно. В этом случае необходимо иметь некоторую функцию, обеспечивающую отображение точки из пространства ключей в точку в пространстве записей, то есть, преобразование ключа в адрес записи: a=h(k), где a – адрес,k – ключ.

Идеальной *хеш-функцией* является инъективная функция, которая для любых двух неодинаковых ключей дает неодинаковые адреса.

Метод остатков от деления

Простейшей *хеш-функцией* является деление по модулю числового значения ключа Key на размер пространства записиHashTableSize. Результат интерпретируется как адрес записи. Следует иметь в виду, что такая функция хорошо соответствует первому, но плохо – последним трем требованиям к *хеш-функции* и сама по себе может быть применена лишь в очень ограниченном диапазоне реальных задач. Однако операция деления по модулю обычно применяется как последний шаг в более сложных функциях *хеширования*, обеспечивая приведение результата к размеру пространства записей.

Если ключей меньше, чем элементов массива, то в качестве *хеш-функции* можно использовать деление по модулю, то есть остаток от деления целочисленного ключа Key на *размерность массива* HashTableSize, то есть:

Key % HashTableSize

Данная функция очень проста, хотя и не относится к хорошим. Вообще, можно использовать любую *размерность массива*, но она должна быть такой, чтобы минимизировать число *коллизий*. Для этого в качестве размерности лучше использовать простое число. В большинстве случаев подобный выбор вполне удовлетворителен. Для *символьной строки* ключом может являться остаток от деления, например, суммы *кодов символов* строки на HashTableSize.

На практике, метод деления – самый распространенный.

```
//функция создания хеш-таблицы метод деления по модулю
int Hash(int Key, int HashTableSize) {
```

```
//HashTableSize
    return Key % HashTableSize;
}
```

Метод функции середины квадрата

Следующей *хеш-функцией* является функция середины квадрата. Значение ключа преобразуется в число, это число затем возводится в квадрат, из него выбираются несколько средних цифр и интерпретируются как адрес записи.

Метод свертки

Еще одной *хеш-функцией* можно назвать функцию *свертки*. Цифровое представление ключа разбивается на части, каждая из которых имеет длину, равную длине требуемого адреса. Над частями производятся определенные арифметические или поразрядные *логические операции*, результат которых интерпретируется как адрес. Например, для сравнительно небольших таблиц с ключами – *символьными строками* неплохие результаты дает функция *хеширования*, в которой адрес записи получается в результате сложения *кодов символов*, составляющих строку-ключ.

В качестве *хеш-функции* также применяют функцию преобразования *системы счисления*. Ключ, записанный как число в некоторой *системе счисления* P, интерпретируется как число в *системе счисления* Q>P. Обычно выбирают Q=P+1. Это число переводится из системы Q обратно в систему P, приводится к размеру пространства записей и интерпретируется как адрес.

Открытое хеширование

Основная идея базовой структуры при открытом (внешнем) *хешировании* заключается в том, что потенциальное множество (возможно, бесконечное) разбивается на конечное *число классов*. Для B классов, пронумерованных от 0 до B-1, строится *хеш-функция* h(x) такая, что для любого элемента x исходного множества функция h(x) принимает целочисленное значение из интервала 0,1,...,B-1, соответствующее классу, которому принадлежит элемент x. Часто классы называют сегментами, поэтому будем говорить, что элемент x принадлежит сегменту h(x). Массив, называемый таблицей сегментов и проиндексированный номерами сегментов 0,1,...,B-1, содержит заголовки для B списков. Элемент x, относящийся к i -му списку – это элемент исходного множества, для которого h(x)=i.

Если сегменты примерно одинаковы по размеру, то в

этом случае списки всех сегментов должны быть наиболее короткими при данном числе сегментов. Если исходное множество состоит из N элементов, тогда средняя длина списков будет N/Вэлементов. Если можно оценить величину N и выбрать B как можно ближе к этой величине, то в каждом списке будет один или два элемента. Тогда время выполнения операторов словарей будет малой постоянной величиной, не зависящей от N.

Закрытое хеширование

При закрытом (внутреннем) *хешировании* в хеш-таблице хранятся непосредственно сами элементы, а не заголовки списков элементов. Поэтому в каждой записи (сегменте) может храниться только один элемент. При закрытом *хешировании* применяется методика повторного хеширования. Если осуществляется попытка поместить элемент x в сегмент с номером h(x), который уже занят другим элементом (коллизия), то в соответствии с методикой повторного *хеширования* выбирается последовательность других номеров сегментов h1(x),h2(x),..., куда можно поместить элемент x. Каждое из этих местоположений последовательно проверяется, пока не будет найдено свободное. Если свободных сегментов нет, то, следовательно, таблица заполнена, и элемент x добавить нельзя.

При поиске элемента x необходимо просмотреть все местоположения h(x),h1(x),h2(x),..., пока не будет найден x или пока не встретится пустой сегмент. Чтобы объяснить, почему можно остановить поиск при достижении пустого сегмента, предположим, что в хеш-таблице не допускается удаление элементов. Пусть h3(x) – первый пустой сегмент. В такой ситуации невозможно нахождение элемента x в сегментах h4(x),h5(x) и далее, так как при вставке элемент x вставляется в первый пустой сегмент, следовательно, он находится где-то до сегмента h3(x). Но если в хеш-таблице допускается удаление элементов, то при достижении пустого сегмента, не найдя элемента x, нельзя быть уверенным в том, что его вообще нет в таблице, так как сегмент может стать пустым уже после вставки элемента x. Поэтому, чтобы увеличить эффективность данной реализации, необходимо в сегмент, который освободился после *операции удаления элемента*, поместить специальную константу, которую назовем, например, *DEL*. В качестве альтернативы специальной константе можно использовать дополнительное поле таблицы, которое показывает состояние элемента. Важно различать константы *DEL* и NULL –

последняя находится в сегментах, которые никогда не содержали элементов. При таком подходе выполнение поиска элемента не требует просмотра всей хеш-таблицы. Кроме того, при вставке элементов сегменты, помеченные константой *DEL*, можно трактовать как свободные, таким образом, пространство, освобожденное после удаления элементов, можно рано или поздно использовать повторно. Но если невозможно непосредственно сразу после удаления элементов пометить освободившиеся сегменты, то следует предпочесть закрытому *хешированию* схему открытого *хеширования*.

Существует несколько методов повторного *хеширования*, то есть определения местоположений h(x),h1(x),h2(x),...:

- линейное опробование;
- квадратичное опробование;
- двойное *хеширование*.

Линейное опробование сводится к *последовательному перебору* сегментов таблицы с некоторым фиксированным шагом:

адрес=h(x)+ci,

где ***i*** – номер попытки разрешить коллизию;

c – константа, определяющая шаг перебора.

При шаге, равном единице, происходит *последовательный перебор* всех сегментов после текущего. Квадратичное опробование отличается от линейного тем, что шаг перебора сегментов нелинейно зависит от номера попытки найти свободный сегмент:

адрес=h(x)+ci+di^2,

где ***i*** – номер попытки разрешить коллизию,

c и ***d*** – *константы*.

Благодаря нелинейности такой адресации уменьшается число проб при большом числе ключей-синонимов. Однако даже относительно небольшое число проб может быстро привести к выходу за *адресное пространство* небольшой таблицы вследствие квадратичной зависимости адреса от номера попытки.

Еще одна разновидность метода открытой адресации, которая называется двойным хешированием, основана на нелинейной адресации, достигаемой за счет суммирования значений основной и дополнительной хеш-функций:

адрес=h(x)+ih2(x).

Очевидно, что по мере заполнения хеш-таблицы будут происходить *коллизии*, и в результате их разрешения очередной *адрес* может выйти за пределы *адресного пространства* таблицы. Чтобы это явление происходило реже, можно пойти на увеличение длины таблицы по сравнению с диапазоном адре-

сов, выдаваемым *хеш-функцией*. С одной стороны, это приведет к сокращению числа *коллизий* и ускорению работы с хеш-таблицей, а с другой – к нерациональному расходованию памяти. Даже при увеличении длины таблицы в два раза по сравнению с областью значений *хеш-функции* нет гарантии того, что в результате *коллизий адрес* не превысит длину таблицы. При этом в начальной части таблицы может оставаться достаточно свободных сегментов. Поэтому на практике используют циклический переход к началу таблицы.

Однако в случае многократного превышения *адресного пространства* и, соответственно, многократного циклического перехода к началу будет происходить просмотр одних и тех же ранее занятых сегментов, тогда как между ними могут быть еще свободные *сегменты*. Более корректным будет использование сдвига адреса на 1 в случае каждого циклического перехода к началу таблицы. Это повышает *вероятность* нахождения свободных сегментов.

В случае применения схемы закрытого *хеширования* скорость выполнения вставки и других операций зависит не только от равномерности распределения элементов по сегментам *хеш-функцией*, но и от выбранной методики повторного *хеширования*(опробования) для разрешения *коллизий*, связанных с попытками вставки элементов в уже заполненные *сегменты*. Например, методика линейного опробования для разрешения *коллизий* – не самый лучший выбор.

Как только несколько последовательных сегментов будут заполнены, образуя группу, любой новый элемент при попытке вставки в эти *сегменты* будет вставлен в конец этой группы, увеличивая тем самым длину группы последовательно заполненных сегментов. Другими словами, для поиска пустого сегмента в случае непрерывного расположения заполненных сегментов необходимо просмотреть больше сегментов, чем при случайном распределении заполненных сегментов. Отсюда также следует очевидный *вывод*, что при непрерывном расположении заполненных сегментов увеличивается *время выполнения* вставки нового элемента и других операций.

Задания к лабораторной работе

При выполнении лабораторной работы для каждого задания требуется написать программу на языке С, которая получает данные с клавиатуры или из входного файла, выполняет их обработку в соответствии с требованиями задания и выводит результат в

выходной *файл*. Для обработки данных необходимо реализовать функции алгоритмов *хеширования* данных. Ограничениями на *входные данные* являются максимальный размер строковых данных, допустимый *диапазон* значений используемых числовых типов в языке С.

Ознакомьтесь с теоретическим материалом лабораторной работы.

Выполните приведенные ниже задания.

1. Составьте хеш-таблицу, содержащую буквы и количество их вхождений во введенной строке. Вывести таблицу на экран. Осуществить поиск введенной буквы в хеш-таблице.
2. Постройте хеш-таблицу из слов произвольного текстового файла, задав ее размерность с экрана. Выведите построенную таблицу слов на экран. Осуществите поиск введенного слова. Выполните программу для различных размерностей таблицы и сравните количество сравнений. Удалите все слова, начинающиеся на указанную букву, выведите таблицу.
3. Постройте хеш-таблицу для зарезервированных слов, используемого языка программирования (не менее 20 слов), содержащую HELP для каждого слова. Выдайте на экран подсказку по введенному слову. Добавьте подсказку по вновь введенному слову, используя при необходимости реструктуризацию таблицы. Сравните эффективность добавления ключа в таблицу или ее реструктуризацию для различной степени заполненности таблицы.
4. В текстовом файле содержатся целые числа. Постройте хеш-таблицу из чисел файла. Осуществите поиск введенного целого числа в хеш-таблице. Сравните результаты количества сравнений при различном наборе данных в файле.

Указания к выполнению работы

Каждое задание необходимо решить в соответствии с изученным алгоритмами *хеширования* данных, реализовав программный код на языке С. Рекомендуется воспользоваться теоретическим материалом лабораторной работы, где подробно рассматривается описание используемых в работе алгоритмов.. Результаты обработки данных следует выводить в выходной *файл* и дублировать *вывод* на экране.

Контрольные вопросы

1. Каков принцип построения хеш-таблиц?
2. Существуют ли универсальные методы построения хеш-таблиц? Ответ обоснуйте.
3. Почему возможно возникновение *коллизий*?
4. Каковы методы устранения *коллизий*? Охарактеризуйте их эффективность в различных ситуациях.
5. Назовите преимущества открытого и закрытого *хеширования*.
6. В каком случае поиск в хеш-таблицах становится неэффективен?
7. Как выбирается метод изменения адреса при повторном *хешировании*?