



МИНОБРНАУКИ РОССИИ
федеральное государственное бюджетное образовательное
учреждение высшего образования
«Московский государственный технологический университет «СТАНКИН»
(ФГБОУ ВО МГТУ «СТАНКИН»)

Институт

информационных систем и технологий

Кафедра

информационных систем

Отчёт по лабораторной работе

по дисциплине **«Теория информационных систем и процессов»**

на тему: **Алгоритм сжатия Хаффмана**

Студент

группа ИДБ-16-07

_____ Махмудов Б.Н.

подпись

Преподаватель

_____ Охотников В.А.

подпись

Глава 1 Программа сжатия данных используя алгоритм Хаффмана



Рис. 1.1 Выбор режима работы программы

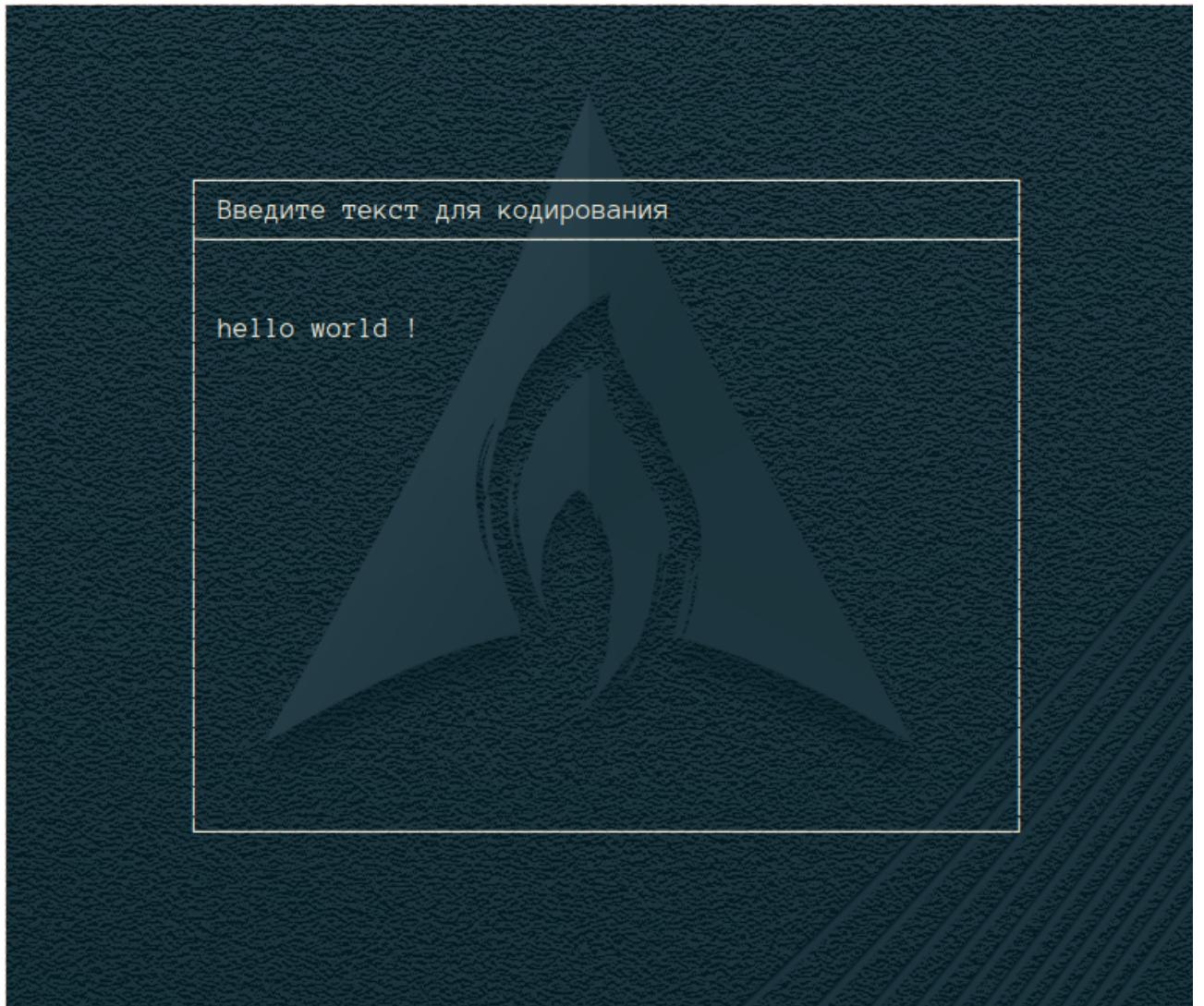


Рис. 1.2 Ввод данных в интерактивном режиме

Таблица префиксных кодов (Хаффман)
Общее количество символов: 13
Количество различных символов: 9

10-ное	Символ	Частота	Префикс код
104	'h'	0.07692	1000
101	'e'	0.07692	1001
108	'l'	0.23077	01
111	'o'	0.15385	101
32	' '	0.15385	00
119	'w'	0.07692	1110
114	'r'	0.07692	1111
100	'd'	0.07692	1100
33	'!'	0.07692	1101

Введенный вами текст следующий:
hello world !

Рис. 1.3 Вывод таблицы префикс кодов после обработки ввода (интерактивный режим)

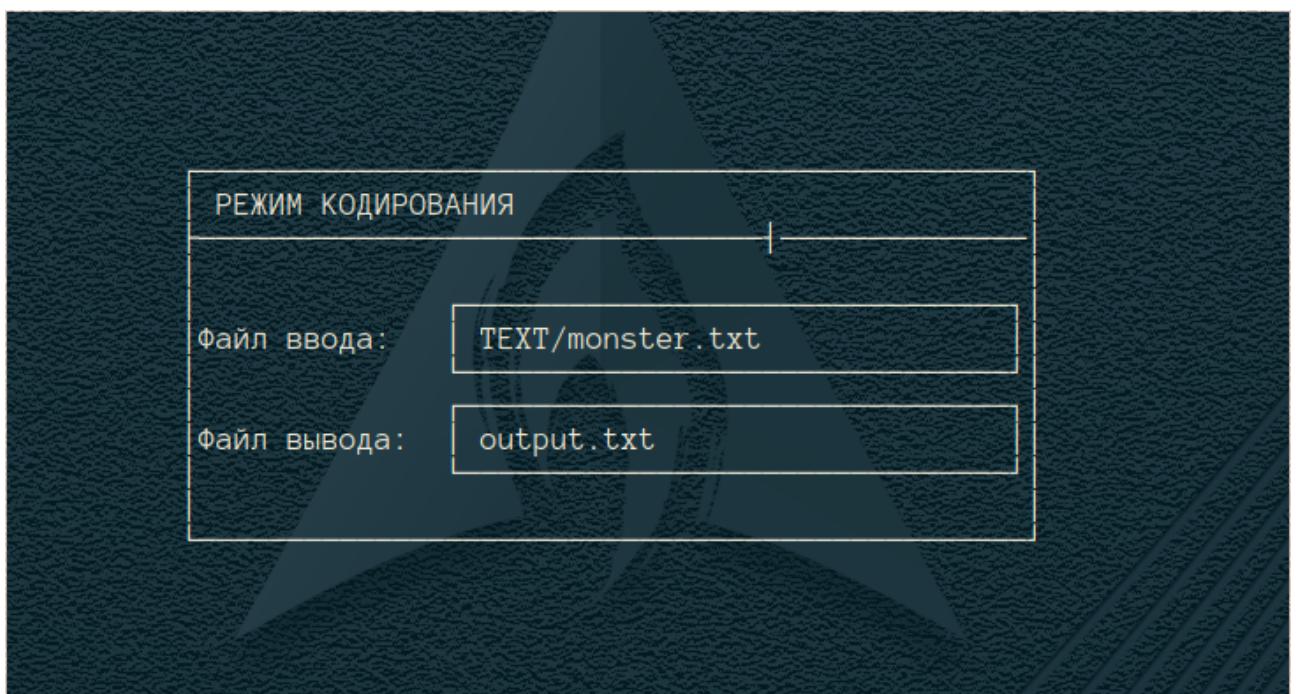


Рис. 1.4 Запрос файла для кодирования и декодирования или наоборот (зависит от режима)

Таблица префиксных кодов:

1054	-	0	-	0.0005647	-	101011111011
1073	-	б	-	0.0057324	-	0001100
1088	-	р	-	0.0156092	-	100010
1072	-	а	-	0.0276065	-	01010
1090	-	т	-	0.0207741	-	111101
1085	-	н	-	0.0219929	-	00000
1103	-	я	-	0.0067710	-	0011111
32	-	-	-	0.1555918	-	110
1089	-	с	-	0.0181663	-	101110
1086	-	о	-	0.0386970	-	11101
1087	-	п	-	0.0086618	-	1010110
1091	-	у	-	0.0094945	-	1011110
1080	-	и	-	0.0225898	-	00010
13	-	#	-	0.0052491	-	11111010
10	-	#	-	0.0161477	-	100110
1082	-	к	-	0.0108161	-	1111111

Рис. 1.5 Вывод таблицы префикс кодов после обработки файла (режима ввода/вывода из файла)

```

#define CODIFY

#include <huffman.h>

//функция для вычисления префикс кодов

void codify(CODETABLE table)

{

//размер очереди (используется в дальнейшем)

int qsz = table->distcharc;

int mintreedepth = 1;

CODEVALUE sorted[table->distcharc];

int min = 1, bp; //bp - для вставки в очередь нового эл-та


/*после окончания цикла получим

полностью отсортированную очередь*/

for (int i = 0; i < qsz; ) {

for (int j = 0; j < qsz; ++j)

if (table->codes[j].occurrence == min) {

sorted[i].value = table->codes[j].value;

sorted[i++].occurrence = table->codes[j].occurrence;

}

++min;

}

CODENODE *queue = malloc(sizeof(CODENODE) * qsz); //очередь

/*baseline имеет точную копию очереди до начала работы алгоритма,

* также каждый элемент baseline имеет ссылку на элемент из массива

* codes в таблице кодов*/



CODENODE baseline = malloc(sizeof(struct codenode) * qsz);

CODENODE temp; //указатель на побегушках

for (int i = 0; i < table->distcharc; ++i) {

```

```

baseline[i].weight = sorted[i].occurrence;
baseline[i].origin = &sorted[i];
queue[i] = &baseline[i];
}

/*начало работы алгоритма, вся суть которого получить бинарное дерево
 * алгоритм использует сортируемую очередь для получения оптимальных
 * префикс кодов*/
while (qsz > 1) {
    temp = malloc(sizeof(struct codenode));
    temp->weight = queue[0]->weight + queue[1]->weight;
    temp->parent = NULL;
    queue[0]->parent = queue[1]->parent = temp;
    queue[0]->dirctn = '0';
    queue[1]->dirctn = '1';

    for (int i = 0; i < qsz - 2; ++i) {
        queue[i] = queue[i + 2]; //двигаем элементы на 2 влево
    }
    //сжимаем очередь
    queue = realloc(queue, sizeof(CODENODE) * (--qsz));
    //находим место для вставки нового элемента
    for (bp = 0; (bp < qsz - 1) &&
        (queue[bp]->weight < temp->weight); ++bp);
    for (int i = 0; i < (qsz - bp - 1); ++i)
        queue[qsz - 1 - i] = queue[qsz - 2 - i];//подвиньтесь!
    queue[bp] = temp; //вставляем
}

/*сборка кодов, начинаем снизу с каждого элемента baseline,
 * оттуда движемся вверх используя указатели до корня,

```

```

* в результате получим перевернутый префикс код*/
char t[100] = {0}; //временный буффер для сборки кода
for (int i = 0, j = 0; i < table->distcharc; ++i) {
temp = &baseline[i];
for (j = 0; temp->parent; j++) {
t[j] = temp->dirctn;
temp = temp->parent;
}
t[j] = 0;
/*mintreedepth мета информация касательно минимальной глубины дерева,
 * данная информация существенно ускоряет процесс декодирования в
 * дальнейшем*/
mintreedepth = (mintreedepth < j) ? mintreedepth : j;
strreverse(t); //переворачиваем строку
baseline[i].origin->prefixc = malloc(strlen(t) + 1);
//codelen мета-информация для записи таблицы в файл
baseline[i].origin->codelen = strlen(t);
/*кладем полученный код в таблицу */
strcpy(baseline[i].origin->prefixc, t);
memset(t, 0, 100);
}
for (int i = 0, j = 0; i < table->distcharc; ++i) {
while (sorted[j++].value != table->codes[i].value);
table->codes[i].prefixc = sorted[--j].prefixc;
table->codes[i].codelen = sorted[j].codelen;
j = 0;
}
table->mintreedepth = mintreedepth;
free(queue);

```

```
free(baseline);  
}  
  
/*pathnode используется при декодирования  
 * является узлом дерева поиска, которое  
 * формируется используя таблицу кодов  
 * данный подход снизил время декодирования  
 * в 5 раз*/
```

```
typedef struct pathnode {  
wchar_t value;  
struct pathnode *child[2];  
} *PATHNODE;
```

```
/*Основная структура содержащие данные о кодах  
 * и символах*/  
typedef struct codevalue {  
wchar_t value;  
unsigned int occurrence;  
double distribution;  
short codelen;  
char *prefixc;  
} CODEVALUE;
```

```
/*Мастер структура*/  
typedef struct codetable {  
unsigned int totcharc;  
unsigned int distcharc;  
unsigned int tableszie;
```

```

unsigned int mintreedepth;
CODEVALUE codes[] ;
} *CODETABLE;

/*Используется при формировании префиксного дерева*/
typedef struct codenode {
unsigned int weight;
unsigned char dirctn;
struct codenode *parent;
CODEVALUE *origin;
} *CODENODE;

#include <huffman.h>

/*Кладёт таблицу префикс кодов
 * со всем содержим в файл, кладет как есть
 * в бинарном виде*/
void filedump(CODETABLE table, FILE *out)
{
fwrite(table, table->tablesiz, 1, out);
for (int i = 0; i < table->distcharc; ++i)
fwrite(table->codes[i].prefixc, table->codes[i].codelen, 1, out);
}

/*Вынимает таблицу которую положил filedump*/
CODETABLE fileextract(FILE *in)
{

```

```

unsigned int tsz;

CODETABLE table = malloc(sizeof(struct codetable));

fread(table, sizeof(struct codetable), 1, in);

tsz = table->tablesiz;
free(table);
rewind(in);
table = malloc(tsz);
fread(table, tsz, 1, in);
for (int i = 0; i < table->distcharc; ++i) {
    table->codes[i].prefixc = malloc(table->codes[i].codelen + 1);
    fread(table->codes[i].prefixc, table->codes[i].codelen, 1, in);
    table->codes[i].prefixc[table->codes[i].codelen] = 0;
}
return table;
}

```

```

#define UTILITY

#include <huffman.h>

/*gettable берёт источник символов (файл или строка)
 * читает её заполняет codetable всей информацией кроме
 * кроме кодов*/
CODETABLE gettable(void *textsource, M mode)
{
CODETABLE table;
unsigned int dcount = 0, tcount = 0, tsz;
int val, occs[256] = {0}; //occurrences
wchar_t temp[256] = {0};
wchar_t *hit;
switch (mode) {

```

```

case FILE_S:
;
FILE *fptr = (FILE *)textsource;
while ((val = fgetwc(fptr)) != WEOF) {
if (!(hit = wcschr(temp, val))) {
temp[dcount] = val;
++occs[dcount];
++dcount;
}
else
++(occs[hit - temp]);
++tcount;
}
break;

case STRING_S:
;
wchar_t *cptr = (wchar_t *)textsource;
for (int i = 0; cptr[i] != '\0'; ++i) {
if (!(hit = wcschr(temp, cptr[i]))) {
temp[dcount] = cptr[i];
++occs[dcount];
++dcount;
}
else
++(occs[hit - temp]);
++tcount;
}
break;
}

```

```

table = malloc(tsz = (sizeof(*table) + sizeof(struct codevalue) * wcslen
table->totcharc = tcount;
table->distcharc = wcslen(temp);
table->tablesize = tsz;

for (int i = 0; i < table->distcharc; ++i) {
    table->codes[i].value = temp[i];
    table->codes[i].occurrence = occs[i];
    table->codes[i].distribution = occs[i] / (double)tcount;
}

return table;
}

/*спуш, споп, treedelete функции для поддержания чистоты*/
void spush(PATHNODE **stack, int *stsz, PATHNODE node)
{
    *stack = realloc(*stack, sizeof(PATHNODE) * (++(*stsz)));
    (*stack)[*stsz - 1] = node;
}

PATHNODE spop(PATHNODE **stack, int *stsz)
{
    PATHNODE temp = (*stack)[*stsz - 1];
    *stack = realloc(*stack, sizeof(PATHNODE) * (--(*stsz)));
    return temp;
}

```

```

void treedelete(PATHNODE root)
{
    unsigned int stsz = 1;
    PATHNODE *stack = malloc(sizeof(PATHNODE));
    PATHNODE node = *stack = root;

    while (stsz) {
        if (node->child[0])
            spush(&stack, &stsz, node->child[0]);
        if (node->child[1])
            spush(&stack, &stsz, node->child[1]);
        free(node);
        node = spop(&stack, &stsz);
    }
}

```

```

/*build_search_tree построение поискового дерева
 * используя префикс коды в таблице*/
PATHNODE build_search_tree(CODETABLE table)
{
    char buf[50] = {0};
    PATHNODE root = malloc(sizeof(struct pathnode));
    root->child[0] = root->child[1] = NULL;
    PATHNODE allocator = NULL;
    PATHNODE tracer = NULL;
    for (int i = 0; i < table->distcharc; ++i) {
        strcpy(buf, table->codes[i].prefixc);

```

```
tracer = root;

for (int j = 0; buf[j] != 0; ++j) {
    if (tracer->child[buf[j] - 48] == NULL) {
        allocator = malloc(sizeof(struct pathnode));
        allocator->child[0] = allocator->child[1] = NULL;
        tracer->child[buf[j] - 48] = allocator;
    }

    tracer = tracer->child[buf[j] - 48];
}

tracer->value = table->codes[i].value;
}

return root;
}
```