МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №5
по дисциплине «Алгоритмы и структуры данных»
Тема: Бинарные деревья

Студент гр. 6382 _____ Вайгачёв А.О.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2017

Задание № 6

Декодирование: динамическое Хаффмана

Решение задачи.

Имеем исходные данные: алфавит и строка состоящая из 0 и 1(Код). Алфавит загружается из файла data.txt, в каждой строчке которого содержится бинарный код символа и через пробел сам символ. В программе он представлен в виде бинарного дерева. На листьях этого дерева символы, а путь к нему это бинарный код(Функции для БинДер в bt_implementation.cpp). Код загружается из файла code.txt. Он сохраняется в программе в виде двунаправленного линейного списка(Функции для линейного списка в list_implementation.cpp). Ответ сохраняется тоже в линейном списке. Для удобства храним код, алфавит и ответ в структуре struct code_n_alph. Динамическое декодирование Хаффмана предполагает специально упорядоченное дерево, такое что при порядке обхода от левого нижнего до вершины слева направо образуется не убывающая последовательность весов. Для удобной сортировки используем бинарное дерево на базе вектора(Функции для БинДер на векторе в vecbt_implementation.cpp). Сам алгоритм записан в файле huff_implementation.cpp.

Алгоритм устроен следующем способом: считываем первый символ из кода и инициализируем первое дерево. Справа символ с большим весом, слева специальный символ NCH = «#», необходимый для вхождения нового символа. После изменения веса или добавления нового символа последовательность может нарушится. Тогда необходимо делать перестановки узлов по правилу: меняем местами лист/узел A(нарушивший упорядоченность) и B(последний из меньших по весу узлов при перечислении и не «отец» A) и пересчитываем веса узлов. Делаем это до тех пор пока упорядоченность не восстановится. Об успешном окончании говорит создание нового символа с кодом завершения.

Программа выводит в консоль загруженный код, ошибки, ответ и сообщение о корректном завершении программы. Вся информацию о ходе работы записана в файле log.txt.

Описание функций и глобальных переменных

Btree.h

```
    typedef char base;
    struct node {
        base info;
        node *lt;
        node *rt;
            // constructor
            node () {lt = NULL; rt = NULL;}
    };

    typedef node *binTree; // "представитель" бинарного дерева

    binTree Create(void); //создает БТ. Возвращает БТ
```

```
    bool isNull(binTree); //проверяет БТ на нулевой указатель. На вход БТ
    base RootBT (binTree); // для непустого бин.дерева
    binTree Left (binTree);// для непустого бин.дерева
    binTree Right (binTree);// для непустого бин.дерева
    binTree ConsBT(const base &x, binTree &lst,  binTree &rst);
//Конструктор БТ. На вход атом, левое и правое поддерево
    void destroy (binTree&); //уничтожает БТ.
```

Huf.h

```
#include "Btree.h"
#include "list.h"
#include "vecbt.h"

#define MAX_SIZE 257      // posible asci character
#define HEAD MAX_SIZE - 1

struct code_n_alph{
    list *code;         //contains code
    binTree alph;       //contains alphavite
    list* answ;         //contains answer. Is used to hold decoded msg
    code_n_alph(){
        list *code = new list();
        binTree* alph = new binTree();
        list *answ = new list();
    }
};

code_n_alph *encoding(struct code_n_alph* cna);     //encoding finction
template <typename T> void huf_init(Node<T>*hf, struct code_n_alph* cna);
//initiating huf tree with first symbol

char nextSymbol(struct code_n_alph* cna);    //returns next symbol from code if it
was transmited
char nextByte(struct code_n_alph* cna);     // returns next byte from code

//Changes nodes (current <-> index)
template<typename T>void   HUF_swap(Node<T>* hf,size_t current, size_t index);

//Inserts new symbol in the tree with index from code cna
template<typename T> int   HUF_insert(Node<T>* hf,size_t index,struct code_n_alph
*cna);

//Makes tree -> huffman tree
template<typename T>int   HUF_update(Node<T>* hf);

//Adds mew symbol to answ from next bytes of code
template<typename T> int   HUF_add(Node<T>* hf,struct code_n_alph *cna);

//Recursiv rewight of hf
template<typename T>size_t HUF_wight(Node<T>* hf,size_t index);

//Changes HUFFTREE[index] with determined info
template<typename T>void   HUF_change(Node<T>* hf,size_t index,
                                      T a,size_t b,size_t c,size_t d,unsigned int e);
char EOC = '!';         //End Of Code
char NCH = '#';         //New CHaracter
char EMP = '@';         //EMpty node
size_t last_index;
size_t PTRNULL = 1000; // Обозначает нулевой указатель.
```

#include "Btree.h"

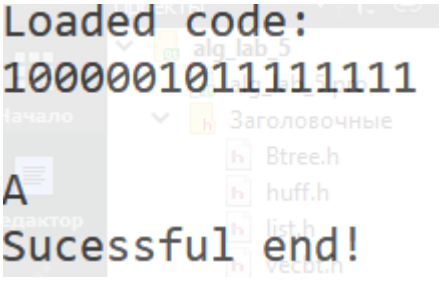
```
#include "list.h"
#include "vecbt.h"

#define MAX_SIZE 257      // posible asci character
#define HEAD MAX_SIZE - 1

struct code_n_alph{
    list *code;         //contains code
    binTree alph;       //contains alphavite
    list* answ;         //contains answer. Is used to hold decoded msg
    code_n_alph(){
        list *code = new list();
        binTree* alph = new binTree();
        list *answ = new list();
    }
};

code_n_alph *encoding(struct code_n_alph* cna);     //encoding finction
template <typename T> void huf_init(Node<T>*hf, struct code_n_alph* cna);
//initiating huf tree with first symbol

char nextSymbol(struct code_n_alph* cna);    //returns next symbol from code if it
was transmited
char nextByte(struct code_n_alph* cna);     // returns next byte from code

//Changes nodes (current <-> index)
template<typename T>void   HUF_swap(Node<T>* hf,size_t current, size_t index);

//Inserts new symbol in the tree with index from code cna
template<typename T> int   HUF_insert(Node<T>* hf,size_t index,struct code_n_alph
*cna);

//Makes tree -> huffman tree
template<typename T>int   HUF_update(Node<T>* hf);

//Adds mew symbol to answ from next bytes of code
template<typename T> int   HUF_add(Node<T>* hf,struct code_n_alph *cna);

//Recursiv rewight of hf
template<typename T>size_t HUF_wight(Node<T>* hf,size_t index);

//Changes HUFFTREE[index] with determined info
template<typename T>void   HUF_change(Node<T>* hf,size_t index,
                                      T a,size_t b,size_t c,size_t d,unsigned int
e);
```

Тестирование.

Для алфавита:

```
01000001 A
01000010 B
01000011 C
01000100 D
01000101 E
01000110 F
01000111 G
01001000 H
01001001 I
01001010 J
00000000 _
01001011 K
01001100 L
01001101 M
01001110 N
01001111 O
01010000 P
01010001 Q
01010010 R
01010011 S
01010100 T
01010101 U
01010110 V
01010111 W
01011000 X
01011001 Y
01011010 Z
11111111 !
```

Пример работы.



Корректные тесты

№	Входные данные	Результат
1	01000001011111111	A Sucessful end!
2	0100000100100001000010000111011000100010011 0110011011111110011111111	ABCCDDDDBB Sucessful end!
3	0100000100100001000010100100100010010110110 0010001000110110010001111111	ABRACADABRA Sucessful end!

№	Входные данные	Результат
4	0101001100100100000010001011000100110000000000000011000101010110000101010010001000100111011100010011111100010010111100101000101000010111000110001000010010101111110011001001011111111010110110001001001111101111010000001000001101000011101111000110111110101000101001011001101001010111011100010011111111	SHEL_UTENOK_PO_BOLOTU_ON_ISKAL_SEBE_RABOTU Sucessful end!

Некорректные тесты

№	Ошибки	Входные данные с ошибками	Результат и ошибка
1	Нет знака окончания декодирования	010000010	Erorr nextByte! nor 0 1! Exiting decoding... A Sucessful end!
2	Нет знака окончания декодирования	0100000100100001000010000111011000100010011011001101111110011111111	Erorr nextByte! nor 0 1! Exiting decoding... ABCCDDDDBB Sucessful end!
3	Без окончания(без вызова символа окончания)	0100000100100001000010100100100010010110110001000100011011 00	Erorr nextByte! nor 0 1! Exiting decoding... ABRACADABRA Sucessful end!
4	Код, в который входят символы, которых нет в алфавите.	0101001100100100000ooooor100010110001001100000000000000110001010101100001010100100010001001110111000100111111100001001010111100101000101000010111000110001000010010101111100110010010111111111010110110001001111101111010000001000001101000011101111000110111110101000101001011001101001010111011100010011111111	Erorr nextByte! nor 0 1! Exiting decoding... Erorr nextByte! nor 0 1! Exiting decoding... SH0 Sucessful end!
5	Символ, которого нет в алфавите	0100000101100001000010100100100010010110110001000100011011 00	Error: RootBT(null) Exiting decoding... Error: RootBT(null) Exiting decoding... Error: RootBT(null) Exiting decoding... Error: RootBT(null) Exiting decoding... Erorr nextByte! nor 0 1! Exiting decoding... A B B A Sucessful end!

№	Ошибки	Входные данные с ошибками	Результат и ошибка
6	Шумы	0100000101100001100001010000100101001001011011000100100010001101100	Error: RootBT(null) Exiting decoding... Error: RootBT(null) Exiting decoding... Error: RootBT(null) Exiting decoding... Error: RootBT(null) Exiting decoding... Erorr nextByte! nor 0 1! Exiting decoding... A AA AA A A AA A AA A AA Sucessful end!
7	Шумы	0101001100100100000100010110001001100000000000001100010101011000010010100100010001001111011100010010111100101000101000010111000110001000001001010111100100100101111111101011011000100100111110111101000000100000110100001110111100011011111101010001010010110011010010101110111001000101111111	Error: RootBT(null) Exiting decoding... Erorr nextByte! nor 0 1! Exiting decoding... SHEL_UTENOK_PO_BOL_P _K___ HKPL_KTO _OLLE_K___KLL_OP SKH Sucessful end!

Приложение А. (Код программы)

```
#ifndef bt_implementation_cpp
  #define bt_implementation_cpp
  #include <iostream>
  #include <cstdlib>
  #include "Btree.h"
  using namespace std ;

  //-------------------------------------
      binTree Create()
      {   return NULL;
      }
  //-------------------------------------
      bool isNull(binTree b)
      {   return (b == NULL);
      }
  //-------------------------------------
      base RootBT (binTree b)
      {   if (b == NULL) {
              cerr << "Error: RootBT(null) \n";
              cerr << "Exiting decoding...\n";
              return NULL;
             // exit(1);
          }
          else return b->info;
      }
  //-------------------------------------
      binTree Left (binTree b)
      {   if (b == NULL) { cerr << "Error: Left(null) \n"; exit(1); }
          else return b ->lt;
      }
```

```
//-------------------------------------
    binTree Right (binTree b)
    {   if (b == NULL) { cerr << "Error: Right(null) \n"; exit(1); }
        else return b->rt;
    }
//-------------------------------------
    binTree ConsBT(const base &x, binTree &lst,  binTree &rst)
    {   binTree p;
        p = new node;
        if ( p != NULL) {
            p ->info = x;
            p ->lt = lst;
            p ->rt = rst;
            return p;
        }
        else {cerr << "Memory not enough\n"; exit(1);}
    }
//-------------------------------------
    void destroy (binTree &b)
    {   if (b != NULL)  {
            destroy (b->lt);
            destroy (b->rt);
            delete b;
            b = NULL;
        }
    }

#endif //bt_implementation_cpp

#ifndef huff_implementation_cpp
#define huff_implementation_cpp

//huff_implementation_cpp

#include "huff.h"

#include "vecbt_implementation.cpp"

char EOC = '!';         //End Of Code
char NCH = '#';         //New CHaracter
char EMP = '@';         //EMpty node
size_t last_index;
size_t PTRNULL = 1000;

struct code_n_alph * encoding(struct code_n_alph *cna){

    Node<char> HUF[MAX_SIZE];

    int i = 1;

    log << "-------" << "Step #" << i++ << "------"<< std::endl;

    huf_init(HUF,cna);
    vec_displayBT(0,HUF,0);

    last_index =  2;                    //max index
    log << "-------" << "Step #" << i++ << "------"<< std::endl;
    while(HUF_add(HUF,cna) == 0){
        HUF_wight(HUF,0);
            log << "\n before update:\n" ;
        vec_displayBT(0,HUF,0);
```

```
            log << "\n after update:\n" ;

        while(HUF_update(HUF) == 1);
        vec_displayBT(0,HUF,0);
        log << "\n-------" << "Step #" << i++ << "------" << std::endl;
    }
    return cna;
}

template<typename T>size_t HUF_wight(Node<T>* hf,size_t index){
    //log << "\nRewighting...\n";
    if(vec_RootBT(hf,index) == EMP){
        hf[index].wight =
                HUF_wight(hf,vec_Left(hf,index))
                + HUF_wight(hf,vec_Right(hf,index));
        return hf[index].wight;
    } else if(vec_RootBT(hf,index) == NCH) {
        hf[index].wight = 0;
        return 0;
    } else {
        return hf[index].wight;;
    }
}

template<typename T>int HUF_update(Node<T>* hf){
    size_t source = 0;
    size_t destination = 0;
    unsigned int dest_wight;  //wight of destination
    size_t d = 1;  // means delta - distance between compareble wight (!) is used
when next is parent;
    size_t pr;          //paren of source
    for(size_t i = last_index; i > 0; i--){
        pr = hf[i].pr;
        if(i-1 == pr) {d = 2;} else {d = 1;};
        if(hf[i].wight > hf[i-d].wight) {
            source = i;
            dest_wight = hf[source-d].wight;
            destination = source-d;
            break;
        }
    }
    if(source != 0){
        unsigned int bigger_wight;
        pr = hf[source].pr;
        for(size_t i = destination; i-2>= 0; i--){
            if(i-1 == pr) {d = 2;} else {d = 1;};
            bigger_wight = hf[i-d].wight;
            if(dest_wight < bigger_wight){
                if(i != pr){
                    destination = i;
                } else {
                    destination = i-d;
                }
                break;
            }
        }
        HUF_swap(hf,source,destination);
        HUF_wight(hf,0);
        return 1;
    }
    return 0;
}
```

```
template<typename T>void HUF_change(Node<T>* hf,size_t index,
                                    T a,size_t b,size_t c,size_t d,unsigned int e){
    hf[index].info  =a;
    hf[index].rt    =b;
    hf[index].lt    =c;
    hf[index].pr    =d;
    hf[index].wight =e;
    hf[index].number;
}

template<typename T>void HUF_swap(Node<T>* hf,size_t a, size_t b){

    log << " \nswitches " << vec_RootBT(hf,a) << "(" <<hf[a].wight << ")" ;
    log << " <-> " << vec_RootBT(hf,b) << "(" <<hf[b].wight << ")\n";

/*    if(a == b){
        if(hf[hf[a].pr].lt == a) {
            hf[a]
        }
    }*/
    size_t parent[2];
    parent[0] = hf[a].pr;
    parent[1] = hf[b].pr;
    Node<T> c[2];
    c[0]  = hf[a];
    c[1]  = hf[b];
    hf[b] =  c[0];
    hf[a] =  c[1];

    hf[a].pr = parent[0];
    hf[b].pr = parent[1];

    // to connect nodes with sons (lft & rgth) if it nesecery
    if(vec_RootBT(hf,a) == EMP){
        hf[vec_Right(hf,a)].pr = a;
        hf[vec_Left(hf,a)].pr  = a;
    }
    if(vec_RootBT(hf,b) == EMP){
        hf[vec_Right(hf,b)].pr = b;
        hf[vec_Left(hf,b)].pr  = b;
    }

}

template<typename T> int HUF_insert(Node<T>* hf,size_t index,struct code_n_alph *cna)
{

    char c = nextSymbol(cna);

    log << "\ninsertion...";

    HUF_change(hf,index+1,
               c, PTRNULL , PTRNULL , index , 1);

    HUF_change(hf,index+2,
               NCH, PTRNULL , PTRNULL , index , 0);

    HUF_change(hf,index,
               EMP, index+1 , index+2 , hf[index].pr , 1);
```

```
    Add(cna->answ,c);
    c = vec_RootBT(hf,index+1);

    if(c == EOC) return 1;

    last_index += 2;
    HUF_wight(hf,0);

    return 0;
}

template<typename T> int HUF_add(Node<T>* hf,struct code_n_alph *cna){
    char bit;
    size_t index = 0;
    log << "\nnext bytes:";
    char c = vec_RootBT(hf,index);
    while(c == EMP){
        bit = nextByte(cna);
        log << bit;
        if(bit == '0'){
            index = vec_Left(hf,index);
        } else if(bit == '1'){
            index = vec_Right(hf,index);
        } else {
            c = bit;
            break;
        }
        c = vec_RootBT(hf,index);
    };

    log << " - HUF code's symbol - " << c;
    if(c == NCH){
        log << " - new symbol\n";
        return HUF_insert(hf,index,cna);
    }

    if(c == EOC){
        log << " - Error add! not enough bytes...\n";
        log << " Exiting decoding...\n";
        Add(cna->answ,c);
        return 1;
    }

    Add(cna->answ,c);
    (hf[index].wight)++;
    HUF_wight(hf,0);
    return 0;
}

char nextByte(struct code_n_alph* cna){   // '0' or '1'
    char c = Item(cna->code);

    if(c != '0' && c != '1'){
            std::cerr << "Erorr nextByte! nor 0 1!\n";
            std::cerr << "Exiting decoding...\n";
            c = EOC;
            //exit(EXIT_FAILURE);
        }
    Next(cna->code);
    return c;
}
```

```
char nextSymbol(struct code_n_alph* cna){
    char code;
    char letter;
    log << "New code bytes:";
    code = nextByte(cna);
    log << code;
    binTree tmp = cna->alph;
    do{
        if(code == '0'){
            tmp = Left(tmp);
        } else if(code == '1'){
            tmp = Right(tmp);
        } else if(code == EOC){
            letter = code;
            break;
        };
        letter = RootBT(tmp);
        if(letter != '0' && letter != '1') break;
        code = nextByte(cna);
        log << code;
    } while(code == '0' || code == '1');
    log << " - symbol of " << letter  << std::endl;
    return letter;
}

template <typename T> void huf_init(Node<T>* hf,struct code_n_alph* cna){
    Start(cna->code);
    Start(cna->answ);
    char c = nextSymbol(cna);
    if(c == EOC){
        std::cerr << "Error huf_init! EOC at the start!";
        exit(EXIT_FAILURE);
    }
    Add(cna->answ,c);
    Next(cna->answ);

    //first node

    hf[0].info  =  EMP;
    hf[0].rt    =  1;
    hf[0].lt    =  2;
    hf[0].pr    = PTRNULL;
    hf[0].wight =  1;
    hf[0].number;

    //first character

    hf[1].info  =  c;
    hf[1].rt    = PTRNULL;  // means empty
    hf[1].lt    = PTRNULL;
    hf[1].pr    =  0;
    hf[1].wight =  1;
    hf[1].number;

    //New symbol

    hf[2].info  =  NCH;
    hf[2].rt    = PTRNULL;
    hf[2].lt    = PTRNULL;
    hf[2].pr    =  0;
    hf[2].wight =  0;
    hf[2].number;
```

```
}

////////////////////////////////////////////////////////////

#endif //huff_implementation_cpp

#include "vecbt.h"
using namespace std ;

int currentH = 0;
size_t lastIndex = 0;

template <typename T> bool vec_isNull(Node<T>* b,size_t index){
    if(index != PTRNULL){
        return b[index].info == NULL;
    } else {
        return NULL == NULL;
    }
}

template <typename T> T vec_RootBT (Node<T>* b, size_t index){
    if (b == NULL) {
        cerr << "Error: RootBT(null) \n";
        exit(1);
    }
    else
        return b[index].info;
}

template <typename T> size_t vec_Left (Node<T>* b,size_t index ){
    if (b == NULL){
        cerr << "Error: Left(null) \n";
        exit(1);
    }
    else
        return b[index].lt;
}

template <typename T> size_t vec_Right (Node<T>* b, size_t index){
    if (b == NULL) {
        cerr << "Error: Right(null) \n";
        exit(1);
    }
    else
        return b[index].rt;
}

template <typename T> size_t vec_ConsBT(T x, size_t lst,  size_t rst, size_t pr,
unsigned int wight, unsigned int number,
                                    Node<T>* b, size_t index){
    if ( b != NULL) {
        b[index].info = x;
        b[index].lt = lst;
        b[index].rt = rst;
        b[index].pr = pr;
        b[index].wight = wight;
        b[index].number = number;
        if (lastIndex<rst)
            lastIndex = rst;
        return index;
    }
```

```
    else {
        cerr << "Memory not enough\n";
        exit(1);
    }
}

template <typename T> void vec_destroy (size_t index, Node<T>* b){
    if (b != NULL) {
        destroy (b[index].lt, b);
        destroy (b[index].rt, b);
        delete b;
        b = NULL;
    }
}
/*
template <typename T> size_t enterBT (size_t index, Node<T>* b, ifstream &fin){
    char ch;
    int p, q;
    fin >> ch;
    lastIndex = index;
    if (ch=='/'){
        b[index].info = NULL;
        return index;
    }
    else{
        p = enterBT(++index, b, fin);
        index = lastIndex;
        q = enterBT(++index, b, fin);
        index = q;
        return ConsBT(ch, p, q, b, index-(q-p+1));
    }
}
*/
template <typename T> size_t vec_readBT(size_t index, Node<T>* b){
    char ch;
    int p, q;
    cin >> ch;
    lastIndex = index;
    if (ch=='/'){
        b[index].info = NULL;
        return index;
    }
    else{
        p = readBT(++index, b);
        index = lastIndex;
        q = readBT(++index, b);
        index = q;
        return ConsBT(ch, p, q, b, index-(q-p+1));
    }
}

template <typename T> void vec_outBT(size_t index, Node<T>* b){
    if (b[index].info != NULL) {
        cout << RootBT(b,index);
        outBT(Left(b,index), b);
        outBT(Right(b,index), b);
    }
    else cout << '/';
}

template <typename T> void vec_displayBT (size_t index, Node<T>* b, size_t n){
    if (b!=NULL) {
        if(!vec_isNull(b, index)){
```

```
            log << " " << vec_RootBT(b, index) << "(" << b[index].wight << ")";
            if(!vec_isNull(b,vec_Right(b, index))){
                vec_displayBT (vec_Right(b, index), b,n+1);
            }
            else log << endl;
            if(!vec_isNull(b,vec_Left(b, index))) {
                for (int i=1;i<=n;i++) log << "      ";
                vec_displayBT (vec_Left(b, index), b,n+1);
            }
        }
        else{
            log << "(NULL)" << endl;
        }
    }
    else {
    };
}



#ifndef list_implementation_cpp

#define list_implementation_cpp

#include <iostream>
#include <windows.h>
#include <fstream>
#include <cstring>

//size_t PTRNULL = 1000;

#include "huff.h"

    list* Cre_List(){
        list *l = new list();
        if(l == NULL){
            std::cerr << "Error list! Not enough mem." << std::endl;
            exit(EXIT_FAILURE);
        }
        else return l;
    }

    bool IsNull(list *l){
        if(l == NULL){
            std::cerr << "Error list! List is empty." << std::endl;
            exit(EXIT_FAILURE);
        }
        return l == NULL;
    }

    void  Next(list* l){
        IsNull(l);
        if (l->cur == NULL){
            std::cerr << "Error next! Next is empty." << std::endl;
            exit(EXIT_FAILURE);
        }
        l->cur = l->cur->next;
    }

    void Prev(list* l){
        IsNull(l);
```

```
    if(l->cur->prev == NULL){
        std::cerr << "Error prev! Prev is empty." << std::endl;
        exit(EXIT_FAILURE);
    }
    l->cur = l->cur->prev;
}

void Start(list* l){
    IsNull(l);
    l->cur = l->begin;
}

void End(list* l){
    IsNull(l);
    l->cur = l->end;
}

void Add(list* l, char elem){
    IsNull(l);
    atom *n = new atom();
    n->elem = elem;
    if (n == NULL){
        std::cerr << "Error add! Not enough mem." << std::endl;
        exit(EXIT_FAILURE);
    }
    if(l->begin->elem == NULL){
        l->cur = n;
        l->begin = l->cur;
        l->end   = l->cur;
        l->cur   = l->begin;
        l->size += 1;
        return;
    }
    else{
        if(l->end->prev == NULL) {
            l->end = n;
            l->end->prev = l->begin;
            l->begin->next = l->end;
            Start(l);
        }
        else{
            l->end->next = n;
            n->prev = l->end;
            l->end = l->end->next;
            Start(l);
        }
        l->size += 1;
    }
}

char Item(list* l){
    IsNull(l);
    if(l->cur == NULL){return '!';}
    return l->cur->elem;
}

void destroy(list* l){
    IsNull(l);
    for(int i = 0; i < l->size; i++){
        End(l);
        l->end = l->cur->prev;
        delete l->cur;
```

```
        }
        delete l;
    }

#endif //list_implementation_cpp
```

main.cpp

```
#include <iostream>

#include <windows.h>
#include <fstream>
#include <cstring>

#include "huff.h"
using namespace std;

//Makes alphavite binTree
binTree alpha();
//Comstucts BT
binTree make_tree(binTree p);
// Reads from file characters and adds to the code list
void input_code(list* l);
//Shows list
void show_list(list* l);

void outBT(binTree b);
void displayBT(binTree b, int n);

ofstream log("log.txt");
int main()
{
    setlocale (0, "rus");
    code_n_alph* cna = new code_n_alph();
    cna->code = Cre_List();
    cna->answ = Cre_List();
    log << "loading code..." << endl;
    cout << "Loaded code:\n";
    input_code(cna->code);
    cout << endl;
    log << "Loading alphavite to BT..." << endl;
    cna->alph = alpha();
    log << "Structure of alphaBT:" << endl;
    outBT(cna->alph);
    log <<"\nDisplay:" <<endl;
    displayBT((cna->alph),0);
    cna = encoding(cna);
    //log << "Decoded message" << endl;
    //cout <<"Decoded message" << endl;
    show_list(cna->answ);
    log  << "\nSucessful end!";
    cout << "\nSucessful end!";

}

void show_list(list*l){
    Start(l);
    Next(l);
    for(int i = 0; i < l->size-1 ; i++){
        log << Item(l);
        cout << Item(l);
```

```
        Next(l);
    }
    log << endl;
    Start(l);
}

// Reads from file characters and adds to the code list
void input_code(list* l){
    ifstream code("code.txt");
    if(!code){
        log << "Erorr code! File not found!" << endl;
        exit(EXIT_FAILURE);
    }
    char cur;
    do{
        code.read(&cur,1);
        Add(l,cur);
    }while(cur != '\n');
    code.close();
    show_list(l);
}

ifstream data("data.txt");
binTree alpha()                 // constructing BT of alphavite
{
    if(!data){
        log << "Erorr data! File not found!" << endl;
        exit(EXIT_FAILURE);
    }
    binTree p = new node();
        p =  make_tree(p);
    return p;

}

binTree make_tree(binTree p){   //0 -> left, 1 -> right
    char cur;
    binTree x = Create();
    binTree tmp = p;
    if(isNull(tmp)) {
        tmp = new node();
    }
    log << "Alphavite:" << endl;

    do{                         // cur != '\0' && !data.eof()

        /*
         * Data must look like
         * [ bin code <space> symbol <enter> ]
         * do not use EOC NCH EMP symbols in data
         * for avoiding unpredictable circustamces
         */

        data.read(&cur,1);      // Loading from file

        log << cur;

        if(cur == '\n'){        // alternativ exit from cycle
            data.read(&cur,1);
            log << cur;
            if(cur == '\n'){
```

```
                data.read(&cur,1);
                log << cur;
                if(cur == '\n'){
                    return p;
                }
            }
        }
        if(cur == '0')
            if(!isNull(Left(tmp))) {
                tmp = Left(tmp);
                continue;
            }
            else{
                tmp->lt = ConsBT(cur,x,x);
                tmp = Left(tmp);
                continue;
            }
        if(cur == '1')
            if(!isNull(tmp->rt)) {
                tmp = Right(tmp);
                continue;
            }
            else{
                tmp->rt = ConsBT(cur,x,x);
                tmp = Right(tmp);
                continue;
            }
        if(cur == ' '){
            data.read(&cur,1);
            log << cur;
            tmp->info = cur;
            tmp = p;
            continue;
        }

     } while (cur != '\0' && !data.eof());

    return p;               //Apparently nevr goes to here

}

void outBT(binTree b)
{
    if (b!=NULL) {
        log << RootBT(b);
        outBT(Left(b));
        outBT(Right(b));
    }
    else log << '/';
}

void displayBT (binTree b, int n)
{       // n - уровень узла
    if (b!=NULL) {
        log << ' ' << RootBT(b);
        if(!isNull(Right(b))) {displayBT (Right(b),n+1);}
        else log << endl; // вниз
        if(!isNull(Left(b))) {
            for (int i=1;i<=n;i++) log << "  "; // вправо
            displayBT (Left(b),n+1);}
    }
    else {};}
```