Digital Design Verification

FYP/Internship Program

Lab 06- Link List

Name: Abdul Hadi Afzal

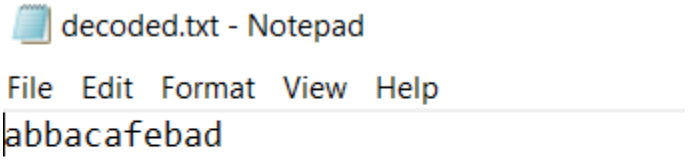
Date: 9 April 2025

Contents

Task1:

SS:

```
HP@haaadi MSYS /c/Users/HP/Desktop/LCDC/LAB6/LAB6
# ./decode
inserted symbol: a
inserted symbol: b
inserted symbol: c
inserted symbol: d
inserted symbol: e
inserted symbol: f
inserted symbol: g
Decoding completed. Output saved in decoded.txt
```



CODE:

Decode.c

```
/**
 * Author: Abdul Hadi Afzal
 * Task: Lab 6 – Huffman Decoder (Task 1: Decompress encoded file)
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_SYMBOLS 255
#define MAX_LEN     10

// Huffman tree node structure
struct tnode {
    struct tnode* left;
    struct tnode* right;
```

```
    int isleaf;
    char symbol;
};

// Global root of the Huffman tree
struct tnode* root = NULL;

/**
 * @function talloc
 * @desc Allocates a new tree node
 */
struct tnode* talloc() {
    struct tnode* p = (struct tnode*)malloc(sizeof(struct tnode));
    if (p != NULL) {
        p->left = p->right = NULL;
        p->symbol = 0;
        p->isleaf = 0;
    }
    return p;
}

/**
 * @function build_tree
 * @desc Builds the Huffman tree from symbol-code mapping in code.txt
 */
void build_tree(FILE* fp) {
    char symbol;
    char strcode[MAX_LEN];
    int items_read, i, len;
    struct tnode* curr = NULL;

    while (!feof(fp)) {
        items_read = fscanf(fp, " %c %s\n", &symbol, strcode);
        if (items_read != 2) break;

        curr = root;
        len = strlen(strcode);

        for (i = 0; i < len; i++) {
            if (strcode[i] == '0') {
                if (curr->left == NULL)
                    curr->left = talloc();
                curr = curr->left;
            } else if (strcode[i] == '1') {
                if (curr->right == NULL)
                    curr->right = talloc();
                curr = curr->right;
            }
        }

        curr->isleaf = 1;
        curr->symbol = symbol;
        printf("inserted symbol: %c\n", symbol);
    }
}

/**
```

```
                                * @function decode
 * @desc Decodes binary input using the Huffman tree and writes to output
                                */
                    void decode(FILE* fin, FILE* fout) {
                                    char c;
                         struct tnode* curr = root;

                       while ((c = getc(fin)) != EOF) {
                                  if (c == '0')
                                curr = curr->left;
                                else if (c == '1')
                               curr = curr->right;
                                     else
      continue; // ignore unexpected characters like newline, space

                                if (curr->isleaf) {
                              fputc(curr->symbol, fout);
                                     curr = root;
                                          }
                                          }
                                          }

                                        /**
                                * @function freetree
                           * @desc Recursively frees memory for Huffman tree
                                        */
                               void freetree(struct tnode* root) {
                                         if (root == NULL)
                                              return;
                                     freetree(root->left);
                                    freetree(root->right);
                                          free(root);
                                          }

                                        /**
                                     * @function main
  * @desc Entry point: builds tree from code.txt, decodes encoded.txt to
                                     decoded.txt
                                          */
                                    int main() {
                          const char* IN_FILE = "encoded.txt";
                           const char* CODE_FILE = "code.txt";
                         const char* OUT_FILE = "decoded.txt";

                                     FILE* fout;
                                      FILE* fin;

                        root = talloc(); // allocate root node

                                // Build tree from code.txt
                               fin = fopen(CODE_FILE, "r");
                                         if (!fin) {
                           perror("Error opening code.txt");
                                       return 1;
                                          }
                                   build_tree(fin);
                                     fclose(fin);
```

```
    // Decode encoded.txt
    fin = fopen(IN_FILE, "r");
    fout = fopen(OUT_FILE, "w");
    if (!fin || !fout) {
perror("Error opening input/output files");
    return 1;
    }

    decode(fin, fout);
    fclose(fin);
    fclose(fout);

    // Free memory
    freetree(root);

printf("Decoding completed. Output saved in decoded.txt\n");
    getchar(); // pause the program
    return 0;
    }
```

Task2:

SS:

```
# ./encode.exe
built code: g -> 0
built code: a -> 10000
built code: b -> 10001
built code: c -> 1001
built code: d -> 101
built code: e -> 110
built code: f -> 111
original: abba cafe bad
encoded: 10000100011000110000100110000111110100011000010 1
```

encoded.txt - Notepad

File Edit Format View Help

```
10000100011000110000100110000111110100011000010 1
```

code.txt - Notepad

File Edit Format View Help

```
a 10000
b 10001
c 1001
d 101
e 110
f 111
g 0
```

CODE:

Encoded.c

```
/**
* Author: Abdul Hadi Afzal
```

```
 * Task: Lab 6 – Huffman Encoder (Task 2: Encode sample string using fixed
                                frequencies)
                                     */

                                #include <stdio.h>
                                #include <stdlib.h>
                                #include <string.h>

                               #define MAX_SYMBOLS 255
                                #define MAX_LEN     10

                                    struct tnode {
                                   struct tnode* left;
                                  struct tnode* right;
                                 struct tnode* parent;
                                  struct tnode* next;
                                      float freq;
                                      int isleaf;
                                     char symbol;
                                        };

                                // Global variables
                           char code[MAX_SYMBOLS][MAX_LEN];
                              struct tnode* root = NULL;
                             struct tnode* qhead = NULL;

                                // Allocate new node
                      struct tnode* talloc(int symbol, float freq) {
                 struct tnode* p = (struct tnode*)malloc(sizeof(struct tnode));
                                      if (p) {
                        p->left = p->right = p->parent = p->next = NULL;
                                p->symbol = symbol;
                                  p->freq = freq;
                                   p->isleaf = 1;
                                        }
                                    return p;
                                      }

                                // Insert into priority queue
                              void pq_insert(struct tnode* p) {
                            struct tnode *curr = qhead, *prev = NULL;

                          if (qhead == NULL || p->freq < qhead->freq) {
                                p->next = qhead;
                                   qhead = p;
                                    return;
                                     }

                             while (curr && p->freq >= curr->freq) {
                                   prev = curr;
                                curr = curr->next;
                                      }

                                   prev->next = p;
                                   p->next = curr;
                                        }
```

```
// Pop from priority queue
struct tnode* pq_pop() {
    if (qhead == NULL)
        return NULL;
    struct tnode* p = qhead;
    qhead = qhead->next;
    p->next = NULL;
    return p;
}

// Generate code using parent pointer
void generate_code(struct tnode* node, int depth) {
    if (node->isleaf) {
        int symbol = node->symbol;
        code[symbol][depth] = '\0';

        struct tnode* p = node;
        int i = depth - 1;
        while (p->parent) {
            code[symbol][i--] = (p == p->parent->left) ? '0' : '1';
            p = p->parent;
        }

        printf("built code: %c -> %s\n", symbol, code[symbol]);
    } else {
        if (node->left)  generate_code(node->left, depth + 1);
        if (node->right) generate_code(node->right, depth + 1);
    }
}

// Write code table to file
void dump_code(FILE* fp) {
    for (int i = 0; i < MAX_SYMBOLS; i++) {
        if (code[i][0]) {
            fprintf(fp, "%c %s\n", i, code[i]);
        }
    }
}

// Encode string using generated code
void encode(const char* str, FILE* fout) {
    while (*str) {
        if (*str != ' ') {
            fprintf(fout, "%s", code[*str]);
        }
        str++;
    }
}

// Free Huffman tree
void freetree(struct tnode* node) {
    if (!node) return;
    freetree(node->left);
    freetree(node->right);
    free(node);
}
```

```
                        // Main function
                          int main() {
    float freq[] = {0.01, 0.04, 0.05, 0.11, 0.19, 0.20, 0.4}; // For a-g
                          int NCHAR = 7;
              const char* test_string = "abba cafe bad";
                 const char* CODE_FILE = "code.txt";
                const char* OUT_FILE = "encoded.txt";
                          FILE* fout = NULL;

                    memset(code, 0, sizeof(code));

                // Step 1: Insert characters into queue
                   for (int i = 0; i < NCHAR; i++) {
                    pq_insert(talloc('a' + i, freq[i]));
                                }

                    // Step 2: Build Huffman tree
                   while (qhead && qhead->next) {
                      struct tnode* lc = pq_pop();
                      struct tnode* rc = pq_pop();
          struct tnode* parent = talloc(0, lc->freq + rc->freq);
                          parent->isleaf = 0;
                           parent->left = lc;
                          parent->right = rc;
                 lc->parent = rc->parent = parent;
                           pq_insert(parent);
                                 }
                         root = pq_pop();

                   // Step 3: Generate codes
                    generate_code(root, 0);

                   // Step 4: Write code.txt
                  fout = fopen(CODE_FILE, "w");
                        dump_code(fout);
                          fclose(fout);

                  // Step 5: Write encoded.txt
                           fout = fopen
```

Task3:

CODE:

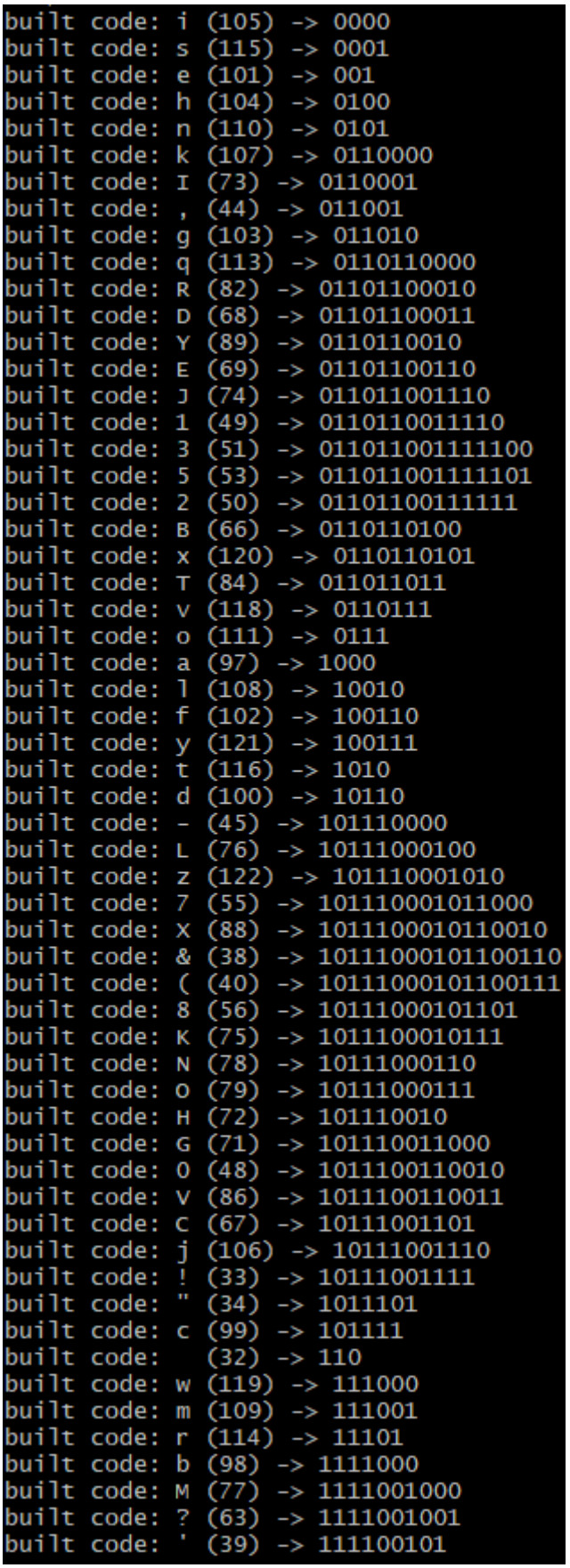
```
                 * Author: Abdul Hadi Afzal
              * Task: Lab 6 – Huffman Encoder (Task 3)
                              */

                     #include <stdio.h>
                    #include <stdlib.h>
                    #include <string.h>

                   #define MAX_SYMBOLS 256
                    #define MAX_LEN     64

                       struct tnode {
                      struct tnode* left;
                     struct tnode* right;
```

```
                        struct tnode* parent;
                         struct tnode* next;
                             float freq;
                             int isleaf;
                            char symbol;
                               };

                    char code[MAX_SYMBOLS][MAX_LEN];
                       struct tnode* root = NULL;
                      struct tnode* qhead = NULL;
                   int char_freq[MAX_SYMBOLS] = {0};

                         // Allocate a tree node
          struct tnode* talloc(int symbol, float freq) {
    struct tnode* p = (struct tnode*)malloc(sizeof(struct tnode));
                              if (p) {
            p->left = p->right = p->parent = p->next = NULL;
                          p->symbol = symbol;
                            p->freq = freq;
                             p->isleaf = 1;
                                 }
                             return p;
                               }

                      // Insert into priority queue
                    void pq_insert(struct tnode* p) {
                  struct tnode *curr = qhead, *prev = NULL;

                   while (curr && p->freq >= curr->freq) {
                               prev = curr;
                            curr = curr->next;
                                   }

                               p->next = curr;
                         if (prev) prev->next = p;
                               else qhead = p;
                                   }

                        // Pop from priority queue
                         struct tnode* pq_pop() {
                           if (!qhead) return NULL;
                           struct tnode* p = qhead;
                             qhead = qhead->next;
                               p->next = NULL;
                                  return p;
                                    }

              // Generate code strings using parent pointers
           void generate_code(struct tnode* node, int depth) {
                             if (node->isleaf) {
                    int symbol = (unsigned char)node->symbol;
                           code[symbol][depth] = '\0';
                             struct tnode* p = node;
                               int i = depth - 1;
                              while (p->parent) {
                             if (p == p->parent->left)
                               code[symbol][i--] = '0';
```

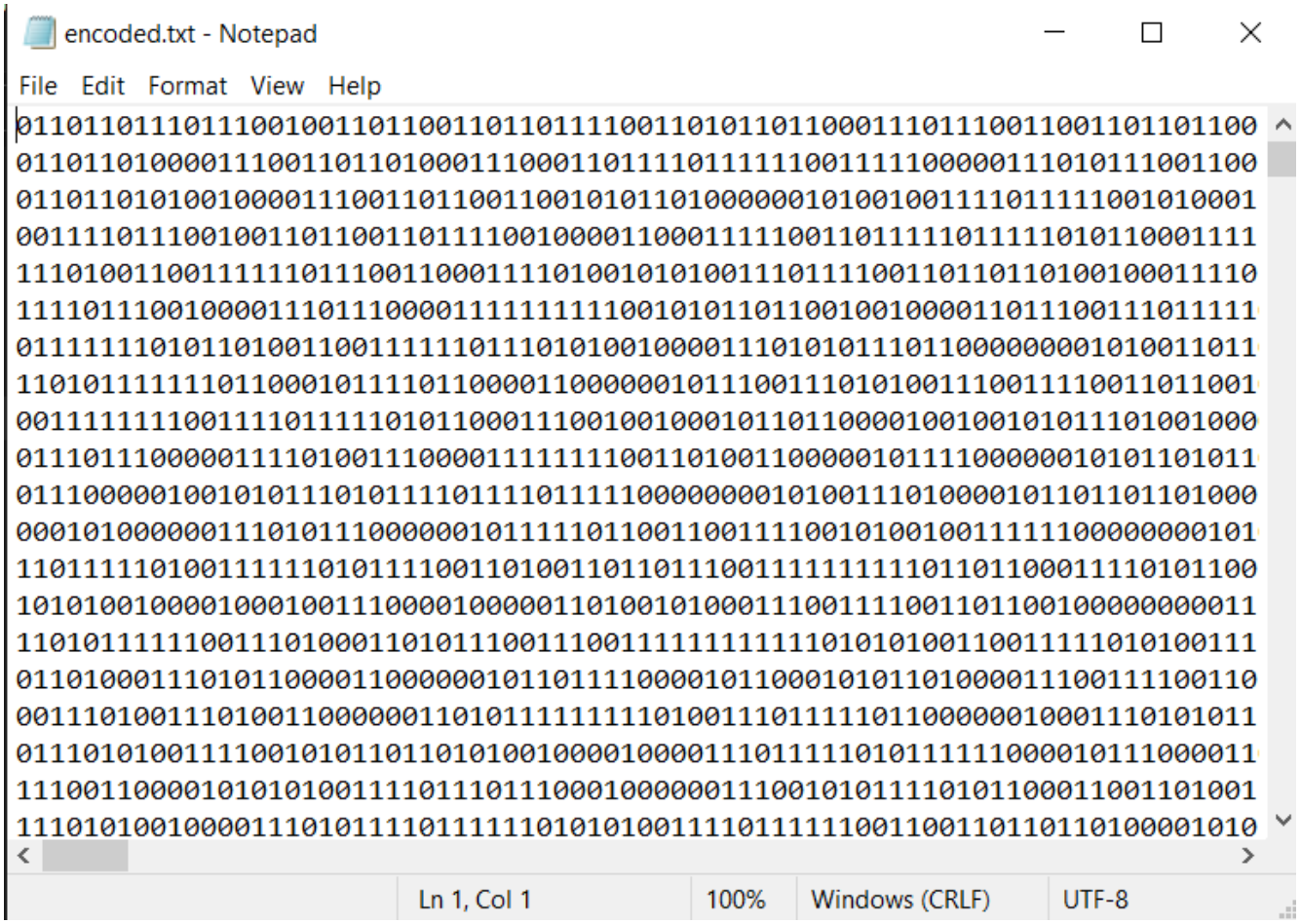
```
                                else
                            code[symbol][i--] = '1';
                               p = p->parent;
                                }
        printf("built code: %c (%d) -> %s\n", symbol, symbol,
                        code[symbol]);
                             } else {
       if (node->left)  generate_code(node->left, depth + 1);
      if (node->right) generate_code(node->right, depth + 1);
                                }
                               }

                    // Output codes to file
                  void dump_code(FILE* fp) {
               for (int i = 0; i < MAX_SYMBOLS; i++) {
                            if (code[i][0]) {
                     fprintf(fp, "%c %s\n", i, code[i]);
                                }
                              }
                             }

            // Encode input file into binary codes
             void encode(FILE* fin, FILE* fout) {
                            int c;
                while ((c = fgetc(fin)) != EOF) {
                            if (code[c][0]) {
                            fputs(code[c], fout);
                                }
                              }
                             }

                  // Free the Huffman tree
             void freetree(struct tnode* node) {
                           if (!node) return;
                        freetree(node->left);
                       freetree(node->right);
                            free(node);
                               }

                            int main() {
                           FILE *fin, *fout;
                const char* IN_FILE = "book.txt";
              const char* CODE_FILE = "code.txt";
             const char* OUT_FILE = "encoded.txt";

                 memset(code, 0, sizeof(code));
              memset(char_freq, 0, sizeof(char_freq));

            // Step 1: Read file and count frequencies
                    fin = fopen(IN_FILE, "r");
                            if (!fin) {
                 perror("Error opening book.txt");
                            return 1;
                              }

                           int total = 0, c;
                while ((c = fgetc(fin)) != EOF) {
```

```
                char_freq[c]++;
                    total++;
                    }
                fclose(fin);

            // Step 2: Build priority queue
        for (int i = 0; i < MAX_SYMBOLS; i++) {
                if (char_freq[i] > 0) {
    pq_insert(talloc(i, (float)char_freq[i] / total));
                    }
                }

            // Step 3: Build Huffman tree
           while (qhead && qhead->next) {
              struct tnode* lc = pq_pop();
              struct tnode* rc = pq_pop();
  struct tnode* parent = talloc(0, lc->freq + rc->freq);
                parent->isleaf = 0;
                 parent->left = lc;
                parent->right = rc;
         lc->parent = rc->parent = parent;
                 pq_insert(parent);
                    }
                root = pq_pop();

              // Step 4: Generate codes
               generate_code(root, 0);

              // Step 5: Save code.txt
           fout = fopen(CODE_FILE, "w");
                  if (!fout) {
           perror("Error writing code.txt");
                      return 1;
                    }
                dump_code(fout);
                  fclose(fout);

        // Step 6: Encode book.txt to encoded.txt
                fin = fopen(IN_FILE, "r");
               fout = fopen(OUT_FILE, "w");
                if (!fin || !fout) {
           perror("Error during encoding");
                      return 1;
                    }
             encode(fin, fout);
                fclose(fin);
                fclose(fout);

    printf("✅ book.txt compressed successfully.\n");
  printf("→ Output files: encoded.txt and code.txt\n");

                freetree(root);
                   getchar();
                   return 0;
                    }
```

SS:


built code: i (105) -> 0000
built code: s (115) -> 0001
built code: e (101) -> 001
built code: h (104) -> 0100
built code: n (110) -> 0101
built code: k (107) -> 0110000
built code: I (73) -> 0110001
built code: , (44) -> 011001
built code: g (103) -> 011010
built code: q (113) -> 0110110000
built code: R (82) -> 01101100010
built code: D (68) -> 01101100011
built code: Y (89) -> 0110110010
built code: E (69) -> 01101100110
built code: J (74) -> 011011001110
built code: 1 (49) -> 0110110011110
built code: 3 (51) -> 011011001111100
built code: 5 (53) -> 011011001111101
built code: 2 (50) -> 01101100111111
built code: B (66) -> 0110110100
built code: x (120) -> 0110110101
built code: T (84) -> 011011011
built code: v (118) -> 0110111
built code: o (111) -> 0111
built code: a (97) -> 1000
built code: l (108) -> 10010
built code: f (102) -> 100110
built code: y (121) -> 100111
built code: t (116) -> 1010
built code: d (100) -> 10110
built code: - (45) -> 101110000
built code: L (76) -> 10111000100
built code: z (122) -> 101110001010
built code: 7 (55) -> 1011100011011000
built code: X (88) -> 1011100010110010
built code: & (38) -> 10111000101100110
built code: ((40) -> 10111000101100111
built code: 8 (56) -> 1011100010110l
built code: K (75) -> 101110001011l
built code: N (78) -> 10111000110
built code: O (79) -> 10111000111
built code: H (72) -> 101110010
built code: G (71) -> 1011100110000
built code: 0 (48) -> 101110011001O
built code: V (86) -> 101110011001l
built code: C (67) -> 10111001101
built code: j (106) -> 10111001110
built code: ! (33) -> 10111001111
built code: " (34) -> 1011101
built code: c (99) -> 101111
built code: (32) -> 110
built code: w (119) -> 111000
built code: m (109) -> 111001
built code: r (114) -> 11101
built code: b (98) -> 1111000
built code: M (77) -> 1111001000
built code: ? (63) -> 1111001001
built code: ' (39) -> 111100101

```
built code: W (87) -> 1111001100
built code: A (65) -> 1111001101
built code: S (83) -> 1111001110
built code: P (80) -> 111100111100
built code: F (70) -> 111100111101
built code: Q (81) -> 1111001111110000
built code: 4 (52) -> 1111001111110001
built code: U (85) -> 11110011111001
built code: ) (41) -> 111100111111010000
built code: Γ (226) -> 1111001111101000100
built code: Φ (232) -> 1111001111101000101
built code: Z (90) -> 1111001111101000110
built code: / (47) -> 11110011111010001110
built code: α (224) -> 111100111111010001111
built code: Θ (233) -> 1111001111101001
built code: 9 (57) -> 1111001111101010
built code: 6 (54) -> 1111001111101011
built code: : (58) -> 11110011111011
built code: ; (59) -> 1111001111111
built code:
 (10) -> 111101
built code: . (46) -> 1111100
built code: p (112) -> 1111101
built code: u (117) -> 111111
Γ£à book.txt compressed successfully.
Γâ€ Output files: encoded.txt and code.txt
```



CONCLUSION:

In this lab, we successfully implemented Huffman encoding and decoding algorithms in C. The process began with building a Huffman tree from either a fixed set of frequencies (Task 2) or real character frequencies extracted from a large input file (Task 3). Using this tree, we generated optimal prefix codes for each character and encoded strings or entire files into binary form.