[INF1010] Lab 6 - Huffman

Ana Caroline Judice - 1710238 Daniel Zimmer - 1810589

PUC-Rio - 15/11/2020

Não conseguimos fazer o código funcionar até o fim. Entendemos como deveriamos fazer e todas as estruturas necessárias para fazer funcionar, porém nos enrolamos em alguma das opereções com bits e nem 4 horas de gdb resolveu nosso problema.

Apesar disso, segue como estamos fazendo/fizemos (uma parte):

Explicação

Encoding

Inicialmente criamos uma lista de uma estrutura que chamamos de Character, nela, guardamos se ela está em uma folha da árvore binária ou não, o caractere que ela representa caso ela esteja numa folha e a contagem daquele caractere no arquivo de entrada.

Em seguida contamos todas ocorrências dos caracteres no arquivo de entrada.

Depois, com todos para todos os caracteres em que existe pelo menos uma ocorrência no arquivo, inserimos ele em um min heap.

Com o min heap criado, partimos para a criação da árvore binária.

Nos removemos sempre os 2 menores elementos do heap e juntamos ele como filhos de um mesmo nó da árvore, em seguida inserimos esse nó de volta no heap. Repetimos isso até que só restasse um elemento no heap. Esse elemento é nossa árvore pronta.

Para gerar os bits para encodar a árvore, nós percorremos a árvore com algoritmo pós. Então, dependendo do caminho percorrido até a folha, gerávamos os bits que representa cada caractere.

Então, utilizamos esses bits para encodar o arquivo. Também colocamos padding no início dele para ter um número exato de bytes. Para determinar quando o arquivo realmente começa utilizamos um bit 1.

Para gerar o header do arquivo de saída, percorremos a árvore novamente da mesma maneira salvando o bit 1 seguido do caractere ascii caso o nó seja uma folha e, caso não seja, inserimos o caractere 0. Depois adicionamos padding para ter o número exato de bytes.

Por fim, salvamos o tamanho do header nos seus primeiros 4 bytes e o salvamos junto com o arquivo encodado no arquivo de saída.

Decoding

Para decodar o arquivo primeiro precisamos parsar o header para gerar a árvore. Para fazer isso lemos bit a bit do header. Quando o bit 1 é lido, em seguida se le um caractere ascii e o insere numa pilha. Quando o caractere 0 é lido, removemos dois itens da pilha e os usamos como filhos de um mesmo nó, que é em seguida colocado de volta na pilha. Quando só resta um nó na pilha, a árvore binária está pronta.

A partir daí precisamos apenas percorrer a árvore seguindo os bits do arquivo de entrada e, sempre que encontrarmos uma folha, escrever aquele caractere no arquivo de saída.

Infelizmente, Existia algum bit mal posicionado na hora de parsar a árvore, portando não conseguimos terminar a decodação

Código

huffman.c

```c
#include <stdio.h>
#include <stdlib.h>

#include "heap.h"
#include "stack.h"
#include "btree.h"

typedef struct character {
    int count;
    char isLeaf;
    char c;
} Character;

typedef struct encoding {
    unsigned char len;
    unsigned char val;
} Encoding;

int HeapCompFunc(void *a, void *b) {
    Character *charA = (Character *) ((Node *) a)->data;
    Character *charB = (Character *) ((Node *) b)->data;

    int countA = charA->count;
    int countB = charB->count;

    if (countA == countB) {
        if (!charA->isLeaf) {
            if (charB->isLeaf) {
                countA++;
            }
        } else { // charA.isLeaf
            if (!charB->isLeaf) {
                countB++;
            } else { // charA.isLeaf && charB.isLeaf
                return charA->c < charB->c;
            }
        }
    }
```

```
    return countA < countB;
}

int traverse(Node *n, Encoding *encs, char val, char len) {
    int sizeLeft, sizeRight;

    if (n->left) {
        sizeLeft = traverse(n->left, encs, val<<1, len+1);
    }
    if (n->right) {
        sizeRight = traverse(n->right, encs, (val<<1) | 1, len+1);
    }

    Character *c = n->data;

    if (c->isLeaf) {
        encs[c->c].val = val;
        encs[c->c].len = len;

        return c->count * len;
    }

    return sizeLeft + sizeRight;
}

void headerTraverse(Node *n, Encoding *encs, char* header, int *headerIdx, char *bitIdx) {
    if (n->left) {
        headerTraverse(n->left, encs, header, headerIdx, bitIdx);
    }
    if (n->right) {
        headerTraverse(n->right, encs, header, headerIdx, bitIdx);
    }

    Character *c = n->data;

    if (c->isLeaf) {

        header[*headerIdx] <<= 1;
        header[*headerIdx] |= 1;
        (*bitIdx)++;

        if (*bitIdx == 8) {
            (*headerIdx)++;
            header[*headerIdx] = 0;
            *bitIdx = 0;
        }

        union {
            char c[2];
            short s;
        } u;
        u.c[0] = c->c;

        int i = 8;
        while(i--) {
```

```
            u.s <<= 1;
            header[*headerIdx] <<= 1;
            header[*headerIdx] |= (u.c[1] & 1);
            (*bitIdx)++;

            if (*bitIdx == 8) {
                (*headerIdx)++;
                header[*headerIdx] = 0;
                *bitIdx = 0;
            }
        }

    } else {
        header[*headerIdx] <<= 1;
        (*bitIdx)++;

        if (*bitIdx == 8) {
            (*headerIdx)++;
            header[*headerIdx] = 0;
            *bitIdx = 0;
        }
    }

}

int main(int argc, char **argv) {

    if (argc != 4) {
        printf("usage: huffman -e|-d [input file] [output file]\n");
        return 1;
    }


    if (argv[1][1] == 'e') {
        FILE *input = fopen(argv[2], "r");
        fseek(input, 0, SEEK_END);
        long inputSize = ftell(input);
        rewind(input);

        char *inputData = malloc(inputSize);
        fread(inputData, inputSize, 1, input);

        Character c[256];

        for (int i = 0; i < 256; i++) {
            c[i].c = i;
            c[i].count = 0;
            c[i].isLeaf = 1;
        }

        for (int i = 0; i < inputSize; i++) {
            c[(int) inputData[i]].count++;
        }

        Heap *h = HEAP_create(HeapCompFunc);
```

```
for (int i = 0; i < 256; i++) {
    if (c[i].count != 0) {
        Node *n = BTREE_create(&c[i]);
        h = HEAP_insert(h, n);
    }
}

while (HEAP_len(h) > 1) {
    Node *right = HEAP_remove(h);
    Node *left  = HEAP_remove(h);

    Character *ch = malloc(sizeof(Character));
    ch->isLeaf = 0;
    ch->count =
        ((Character *) left )->count +
        ((Character *) right)->count;

    Node *p = BTREE_create(ch);
    p->left  = left;
    p->right = right;

    h = HEAP_insert(h, p);
}

Encoding encs[256];

Node *n = HEAP_remove(h);

int bitSize = traverse(n, encs, 0, 0) + 1;

int bitOffset = ((8 - (bitSize%8))%8);
// round up bits to multiple of 8
int byteSize = (bitSize+bitOffset)/8;

printf("---%d\n", byteSize);
printf("---%d\n", bitOffset);

char *outputData = malloc(byteSize);

char len = bitOffset;
int outputIdx = 0;

outputData[outputIdx] = 1;
len++;
for (int i = 0 ; i < inputSize; i++) {
    Encoding enc = encs[inputData[i]];
    printf("%c - %x - %d\n", inputData[i], enc.val, enc.len);
    union {
        char c[2];
        short s;
    } u;
    u.c[0] = enc.val << (8 - enc.len);

    while (enc.len--) {

        u.s <<= 1;
```

```
            outputData[outputIdx] <<= 1;
            outputData[outputIdx] |= (u.c[1] & 1);
            len++;

            if (len == 8) {
                len = 0;
                outputIdx++;
                outputData[outputIdx] = 0;
            }
        }
    }

    char *header = malloc(512);

    int headerIdx = 4;
    char bitIdx = 0;

    header[0] = 0;
    headerTraverse(n, encs, header, &headerIdx, &bitIdx);

    header[headerIdx] <<= 1;
    (bitIdx)++;

    if (bitIdx == 8) {
        (headerIdx)++;
        header[headerIdx] = 0;
        bitIdx = 0;
    }

    for (int i = bitIdx; i < 8; i++) {
        header[headerIdx] <<= 1;
    }
    union {
        char c[4];
        int i;
    } u;
    u.i = headerIdx;

    header[0] = u.c[0];
    header[1] = u.c[1];
    header[2] = u.c[2];
    header[3] = u.c[3];


    FILE *output = fopen(argv[3], "w+");

    fwrite(header, headerIdx, 1, output);
    fwrite(outputData, byteSize, 1, output);

} else if (argv[1][1] == 'd'){

    FILE *input = fopen(argv[2], "r");
    fseek(input, 0, SEEK_END);
    long inputSize = ftell(input);
    rewind(input);
```

```
char *inputData = malloc(inputSize);
fread(inputData, inputSize, 1, input);

Stack *s = STACK_create();

union {
    char c[4];
    int i;
} uu;
uu.c[0] = inputData[0];
uu.c[1] = inputData[1];
uu.c[2] = inputData[2];
uu.c[3] = inputData[3];

union {
    char c[2];
    short s;
} u;

int bits = 0;
char bit = 0;
int inputIdx = 4;

u.c[0] = inputData[inputIdx];

for (int i = 0; i < uu.i - 4; i++) {
    if (bits == 8) {
        inputIdx++;
        u.c[0] = inputData[inputIdx];
        bits = 0;
    }

    u.s <<= 1;
    bit = u.c[1] & 1;
    bits++;
    if (bit == 1) {
        char ascii = 0;
        for (int i = 0; i < 8; i++) {

            if (bits == 8) {
                inputIdx++;
                u.c[0] = inputData[inputIdx];
                bits = 0;
            }

            u.s <<= 1;
            bit = (u.c[1] & 1);
            bits++;

            ascii <<= 1;
            ascii |= bit;
        }

        Character *ch = malloc(sizeof(Character));
        ch->isLeaf = 1;
        ch->c = ascii;
```

```
                Node *n = BTREE_create(ch);
                s = STACK_push(s, n);

            } else {
                if (STACK_len(s) == 1) {
                    break;
                }
                Node *right = STACK_pop(s);
                Node *left  = STACK_pop(s);

                Character *ch = malloc(sizeof(Character));
                ch->isLeaf = 0;
                Node *p = BTREE_create(ch);
                p->left = left;
                p->right = right;

                s = STACK_push(s, p);
            }
        }


        Node *tree = STACK_pop(s);

        printf("%p\n", tree);



    } else {

        printf("usage: huffman -e|-d [input file] [output file]\n");
        return 1;

    }

    return 0;
}
```

heap.c

```
#include "heap.h"

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

#define INIT_CAPACITY 16

struct heap {
    int len;
    int capacity;
    int (*compFunc) (void *a, void *b);
    void *data[];
};

static void up(Heap *h, int i) {
    int p = (i - 1) / 2;
```

```
    // comp data[i] < data[p]
    if (p >= 0 && h->compFunc(h->data[i], h->data[p])) {
        void *aux = h->data[i];
        h->data[i] = h->data[p];
        h->data[p] = aux;
        up(h, p);
    }
}

static void down(Heap *h, int i) {
    int f1 = 2 * i + 1;
    if (f1 >= h->len) {
        return;
    }
    int f2 = f1 + 1;
    if (f2 < h->len) {
        if (h->compFunc(h->data[f2], h->data[f1]) && h->compFunc(h->data[f2], h->data[i])) {
            void *aux = h->data[i];
            h->data[i] = h->data[f2];
            h->data[f2] = aux;
            down(h, f2);
            return;
        }
    }
    if (h->compFunc(h->data[f1], h->data[i])) {
        void *aux = h->data[i];
        h->data[i] = h->data[f1];
        h->data[f1] = aux;
        down(h, f1);
    }
}

Heap *HEAP_create(int (*compFunc) (void *a, void *b)) {
    Heap* h = malloc(sizeof(Heap) + sizeof(void *) * INIT_CAPACITY);

    h->compFunc = compFunc;
    h->capacity = INIT_CAPACITY;
    h->len = 0;

    return h;
}

Heap *HEAP_insert(Heap *h, void *val) {
    if (h->len == h->capacity) {
        h->capacity <<= 2;
        h = realloc(h, sizeof(Heap) + sizeof(void*) * h->capacity);
    }

    h->data[h->len] = val;
    up(h, (h->len)++);

    return h;
}

void *HEAP_remove(Heap *h) {
    void *val = h->data[0];
```

```
    h->data[0] = h->data[--(h->len)];
    down(h, 0);

    return val;
}

Heap *HEAP_construct(void *vet[], int n, int (*compFunc) (void *a, void *b)) {
    Heap *h = HEAP_create(compFunc);

    for (int i = n / 2 - 1; i >= 0; i--) {
        down(h, i);
    }

    return h;
}

int HEAP_len(Heap *h) { return h->len; }
```

heap.h

```
#ifndef HEAP_H
#define HEAP_H

#define INIT_CAPACITY 16

typedef struct heap Heap;

Heap *HEAP_create(int (*compFunc) (void *a, void *b));
Heap *HEAP_insert(Heap *h, void *val);
void *HEAP_remove(Heap *h);
Heap *HEAP_construct(void *vet[], int n, int (*compFunc) (void *a, void *b));
int   HEAP_len(Heap *h);

#endif // HEAP_H
```

stack.c

```
#include "stack.h"

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

#define INIT_CAPACITY 16

struct stack {
    int len;
    int capacity;
    void *data[];
};

Stack *STACK_create() {
    Stack *s = malloc(sizeof(Stack) + sizeof(void *) * INIT_CAPACITY);

    s->len = 0;
```

```
    s->capacity = INIT_CAPACITY;

    return s;
}

Stack *STACK_push(Stack *s, void* val) {
    if (s->len == s->capacity) {
        s->capacity <<= 1;
        s = realloc(s, sizeof(Stack) + sizeof(void *) * s->capacity);
    }

    s->data[s->len++] = val;

    return s;
}

void *STACK_pop(Stack *s) {
    return s->data[--s->len];
}

int STACK_len(Stack *s) { return s->len; };
```

stack.h

```
#ifndef STACK_H
#define STACK_H

typedef struct stack Stack;

Stack *STACK_create();
Stack *STACK_push(Stack *s, void* val);
void  *STACK_pop(Stack *s);
int    STACK_len(Stack *s);

#endif // STACK_H
```

btree.c

```
#include "btree.h"

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

Node *BTREE_create(void *val) {
    Node *n = malloc(sizeof(Node));

    n->left  = NULL;
    n->right = NULL;
    n->data = val;

    return n;
}

Node *BTREE_insertLeft(Node *n, void *val) {
    n->left = BTREE_create(val);
```

```
    return n->left;
}

Node *BTREE_insertRight(Node *n, void *val) {
    n->right = BTREE_create(val);

    return n->right;
}

void BTREE_traversePost(Node *n, void (*func) (void *)) {
    if (n->left) {
        BTREE_traversePost(n->left, func);
    }
    if (n->right) {
        BTREE_traversePost(n->right, func);
    }
    func(n->data);
}
```

btree.h

```
#ifndef BTREE_H
#define BTREE_H

struct node {
    struct node *left;
    struct node *right;
    void *data;
};

typedef struct node Node;

Node *BTREE_create(void *val);
Node *BTREE_insertLeft(Node *n, void *val);
Node *BTREE_insertRight(Node *n, void *val);
void  BTREE_traversePost(Node *n, void (*func) (void *));

#endif // BTREE_H
```

makefile

```
CC = gcc
CARGS = -Wall -Wno-char-subscripts -g

build: huffman

clean:
    rm huffman heap.o stack.o btree.o

huffman: huffman.c heap.o stack.o btree.o
    $(CC) $(CARGS) huffman.c \
        heap.o stack.o btree.o \
        -o huffman
```

```
heap.o: heap.c heap.h
    $(CC) $(CARGS) heap.c -c

stack.o: stack.c stack.h
    $(CC) $(CARGS) stack.c -c

btree.o: btree.c btree.h
    $(CC) $(CARGS) btree.c -c
```