Relatório Tarefa 4 INF1010 - Compressão de arquivo utilizando o algoritmo de Huffman

João Gabriel Peixoto Drumond Lopes - 2210168

Introdução.....3
Módulos e Funções..... 3
priorityqueue.c:.....3
definições de tipos:.....3
funções de acesso:..... 3
funções internas:.....4
arrayoc.c.....5
definições de tipos:.....5
funções de acesso:..... 5
funções internas:.....5
filehandler.c.....5
funções de acesso:..... 5
funções internas:.....5
Descrição da Solução..... 6
Saída da execução do programa..... 10
Saída da execução do Valgrind.....11
Conclusão..... 11

Introdução

- Implementar um algoritmo de compressão de arquivos, baseado no algoritmo de Huffman, de forma a reduzir o espaço de memória ocupado.
- Realizar a leitura de uma cadeia de caracteres de um arquivo e determinar o número de ocorrências de cada um.
- Armazenar esse número de ocorrências, relacionado ao caractere específico.
- Construir uma fila de prioridades ordenada de forma crescente, baseada no número de ocorrências.
- Transformar a fila de prioridades em uma árvore binária, seguindo o algoritmo proposto.
- Realizar um percurso da árvore binária, determinando o valor comprimido da representação de cada caractere.
- Armazenar essas representações em um guia de referência da codificação.
- Imprimir o resultado da codificação do arquivo inserido tanto no terminal quanto em um arquivo binário, criado pelo programa.

Módulos e Funções

priorityqueue.c:

definições de tipos:

- typedef struct priorityqueue pQ: tipo estruturado de cabeça de fila de prioridades (utilizado também como de árvore binária).
- typedef struct node Node: tipo estruturado de nó de fila de prioridade mas, também, de árvore binária.
- typedef struct codedict cdDict: tipo estruturado de dicionário de codificação. Armazena o carácter e o código correspondente.

funções de acesso:

- pQ* create_pq(void): aloca espaço de memória para tipo de cabeça de fila de prioridades e retorna esse espaço.
- void insert_pq(pQ* pq, int val, char let): insere um nó na fila de prioridades preenchendo os campos ‘valor’ e ‘letra’, ‘val’ e ‘let’, além do ponteiro para o ‘prox’, a depender de: lista vazia, elemento no final, elemento no início, etc. Insere de forma ordenada.

- Node* create_node(int val, char let, Node* pai, Node* right, Node* prox, Node* left): aloca espaço de memória para um tipo de nó de fila de prioridade/árvore binária e retorna esse espaço.
- void insert_pq_node(pQ* pq, Node* node): em vez de inserir um valor na lista de prioridades, insere um nó diretamente (para manter os campos não definidos normalmente na inserção, como 'left' e 'right').
- Node* remove_pq(pQ* pq): remove o primeiro nó da fila de prioridades e o retorna.
- void print_node(Node* node): imprime os campos de um nó de fila de prioridade.
- void free_pq(pQ* pq): libera toda a memória alocada de uma fila de prioridade, tanto dos nós internos quanto da cabeça.
- pQ* convert_ao_pq(arrayOc* oc): converte um array de ocorrências em uma fila de prioridade e a retorna.
- void print_pq(pQ* pq): imprime os conteúdos de todos os nós de uma fila de prioridades.
- void pq_to_bt(pQ* pq): converte uma fila de prioridades para uma árvore binária seguindo o algoritmo explicado em sala. Altera diretamente a cabeça da fila de prioridade para servir como cabeça de árvore binária.
- void print_head(pQ* pq): imprime todo o conteúdo de uma árvore binária, a partir da cabeça.
- void free_head(pQ* pq): libera toda a memória alocada por uma estrutura de árvore binária.
- void print_cdDict(cdDict** cddict): imprime todo o conteúdo de um array de estruturas cdDict.
- cdDict** percurso_head(pQ* pq): percorre toda uma árvore binária, passando os valores para a codificação.
- void free_cdDict(cdDict** cddict): libera toda a memória alocada de um array de estruturas cdDict.
- void access_cdDict(cdDict** cddict, int index, char* code);

<u>funções internas:</u>

- void print_btree(Node* node): função que efetivamente imprime os conteúdos da árvore binária. Mesma lógica para todos as outras funções internas desse módulo. Serve para não destruir completamente o encapsulamento da estrutura (poderia, também, ser utilizada uma função de acesso para extrair o 'first')
- void free_btree(Node* node): vide acima, mas para liberação de memória.
- void percurso_btree(cdDict** dict, Node* node, char* code): vide acima, mas para a codificação.

arrayoc.c

definições de tipos:

- typedef struct arrayoc arrayOc: tipo de array de ocorrências, armazena tamanho e um array de inteiros com a ocorrência de cada caracter da tabela ASCII.

funções de acesso:

- arrayOc* extract_characters(char* string): função para extrair os caracteres de uma “string” e inserir para cada caracter na posição adequada do array de ocorrências.
- int access_arrayOc(arrayOc* head, int position): acessa o valor de posição ‘position’ no array de ocorrências.
- void print_arrayOc(arrayOc* head): imprime todo o conteúdo do array de ocorrências.
- int get_length(arrayOc* oc): acessa o valor do tamanho do array de ocorrências.
- void free_arrayOc(arrayOc* ao): libera a memória alocada do array de ocorrências.

funções internas:

- void insert_arrayOc(arrayOc* head, char value): insere um valor no array de ocorrências.
- arrayOc* create_arrayOc(int length): cria uma estrutura de array de ocorrências.
- int* create_intarray(int length): aloca espaço de um array de inteiros de tamanho definido.

filehandler.c

funções de acesso:

- arrayOc* build_arrayOc(char* name, char* mode): constrói um array de ocorrências a partir de um arquivo.
- void print_binary(char* name, cdDict** codedict): imprime em um arquivo binário (e imprime para o terminal) o resultado da codificação do arquivo passado, de acordo com os valores presentes no array de cdDict.

funções internas:

- char* extract_lines(FILE* arq): extrai as linhas de um arquivo. Utiliza um macro para o tamanho máximo, mas poderia alocar tamanho bem maior.

- FILE* open_file(char* name, char* mode): abre um arquivo de nome ‘name’ no modo ‘mode’ e retorna um FILE pointer.
- void close_file(FILE* arq): fecha o arquivo passado por parâmetro.

Descrição da Solução

A descrição da solução é a descrição dos processos sucessivos realizados no arquivo main.c. Além das chamadas aqui citadas, também são realizadas impressões ao longo do programa para indicar a etapa atualmente sendo realizada.
Inicialmente, é chamada a função de construir um array de ocorrências (build_arrayOc()) a partir de um arquivo. O nome do arquivo deve ser passado como parâmetro para a função. Essa função inicia abrindo um arquivo (open_file()), por meio de uma função criada para tratar erros, principalmente. Depois, extrai as linhas desse arquivo para uma string (extract_lines()), que contém todo o conteúdo do arquivo, para mais fácil utilização. A partir dessa string, extrai os caracteres (extract_characters()), e insere num array de ocorrências (insert_arrayOc()), de acordo com o valor, incrementando em um para cada valor. Esse array de ocorrências é criado anteriormente, por meio de create_arrayOc(). O array de fato de ocorrências é criado por create_intarray(). O valor de cada ocorrência é inicializado com 0, para prevenir Undefined Behavior (UB) quando incrementar o valor. Por fim, o arquivo é fechado (close_file()).

```
arrayOc* build_arrayOc(char* name, char* mode) {
    if (name == NULL) {
        fprintf(stderr, "build_arrayOc(): Valor de nome do
arquivo invalido\n");
        exit(-1);
    }
    if (mode == NULL) {
        fprintf(stderr, "build_arrayOc(): Valor de modo de
abertura invalido\n");
        exit(-2);
    }
    FILE* char_arq = open_file(name, mode);
    char* lines = extract_lines(char_arq);
    arrayOc* occurrences = extract_characters(lines);
    free(lines);
```

```
    close_file(char_arq);
    return occurrences;
}
```

```
arrayOc* extract_characters(char* string) {
    if (string == NULL) {
        fprintf(stderr, "extract_characters(): Valor de
string invalida\n");
        exit(-1);
    }
    arrayOc* head = create_arrayOc(ASCII_LENGTH);
    for (int i = 0; i < strlen(string); i++)
        insert_arrayOc(head, string[i]);
    printf("extract_characters(): Array de ocorrencias
construído e retornado com sucesso\n");
    return head;
}
```

Depois disso, o array de ocorrências resultante é impresso (print_arrayOc()), para inspeção do resultado. Em seguida, é realizada a conversão de array de ocorrências para fila de prioridades ordenada crescentemente (convert_ao_pq()). Essa função, por sua vez, chama as funções de acesso de valor por posição no array de ocorrências (access_arrayOc()) e a função de inserção na fila de prioridades (insert_pq()). Essa função trata inserção de acordo com a necessidade, com tratamento diferente para: inserção no início, inserção no meio, inserção no final.

Então, a memória do array de ocorrências é liberada, visto que não mais será utilizado (free_arrayOc()), prevenindo memory leaks.

A fila de prioridades oriunda da conversão é impressa (print_pq()), também para inspeção do resultado. Mostra a posição na fila, o valor do caractere e a quantidade de ocorrências. A partir dela, ocorre a conversão para uma árvore binária, conforme o algoritmo descrito em sala (pq_to_bt()). Essa função chama funções de criação de nó (create_node()), para definição de filhos direito e esquerdo, de inserção de nó em fila de prioridades (inserção de nó direto, sem valor, para manutenção dos campos não presentes na assinatura da função de inserção normal insert_pq_node()).

```
pQ* convert_ao_pq(arrayOc* oc) {
    pQ* new_pq = create_pq();
    for (int i = 0; i < get_length(oc); i++) {
```

```
        if (access_arrayOc(oc, i) != 0)
            insert_pq(new_pq, access_arrayOc(oc, i),
(char)i);
    }
    return new_pq;
}
```

```
void pq_to_bt(pQ* pq) {
    while (pq->tam > 1) {
        Node* left = remove_pq(pq);
        Node* right = remove_pq(pq);

        Node* novo = create_node(left->val + right->val, '+',
NULL, right, NULL, left);
        insert_pq_node(pq, novo);
    }
}
```

Seguindo, é realizado o percurso na árvore binária gerada para criar o dicionário de codificação. A partir da função percurso_head(), é chamada a função create_dict(), para criar cada estrutura do array de cdDict. Em sequência, a função recursiva percurso_btree() é chamada, e realiza o percurso até os nós folhas, atualizando o array de codificação quando adequado. É utilizada uma string e uma string auxiliar, concatenando '0' ou '1', a depender se for para esquerda ou para a direita. Ao chegar no nó folha, o valor é passado para a estrutura de codificação.
É impresso o cdDict** resultante (print_cdDict()) e, depois, a árvore binária criada anteriormente (print_head(), que chama print_btree()), função recursiva clássica de impressão de valores de árvore binária. São utilizados parênteses para melhor formatação e visualização da árvore.

```
void percurso_btree(cdDict** dict, Node* node, char* code) {
    if (node->left == NULL && node->right == NULL) {
        int index = (int)node->let;
        dict[index]->let = node->let;
        strcpy(dict[index]->code, code);
        strcpy(code, "");
        return;
    }
    char code_aux[8];
```

```
    strcpy(code_aux, "");
    strcpy(code_aux, code);
    strcat(code_aux, "0");
    percurso_btree(dict, node->left, code_aux);
    strcpy(code_aux, code);
    strcat(code_aux, "1");
    percurso_btree(dict, node->right, code_aux);
}
```

```
cdDict** percurso_head(pQ* pq) {
    if (pq == NULL)
        exit(-1);
    cdDict** new_cddictarray =
(cdDict**)malloc(sizeof(cdDict*) * 128);
    for (int i = 0; i < 128; i++)
        new_cddictarray[i] = create_cddict(i);
    char code[8] = "";
    percurso_btree(new_cddictarray, pq->first, code);
    return new_cddictarray;
}
```

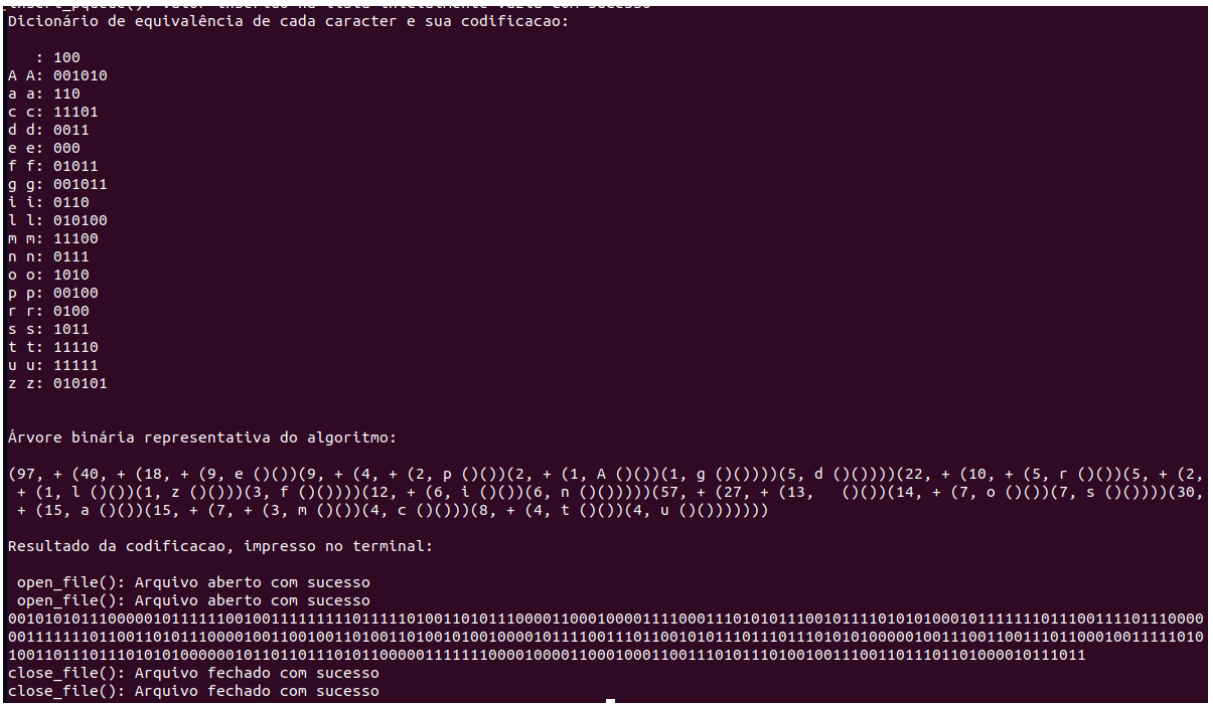
```
cdDict* create_cddict(int i) {
    cdDict* new_cddict = (cdDict*)malloc(sizeof(cdDict));
    if (new_cddict == NULL)
        exit(-1);
    strcpy(new_cddict->code, "");
    new_cddict->let = (char)i;
    return new_cddict;
}
```

A função de codificação de fato é chamada, print_binary(), que imprime no terminal o resultado da codificação. A memória da árvore e do array de codificação é liberada (free_head(), que chama free_btree(), função recursiva tradicional de liberação de memória de árvore binária), e o programa termina e execução.

```
void print_binary(char* name, cdDict** codedict) {
    FILE* char_arq = open_file(name, "r");
    FILE* char_bin = open_file("output.bin", "wb");
```

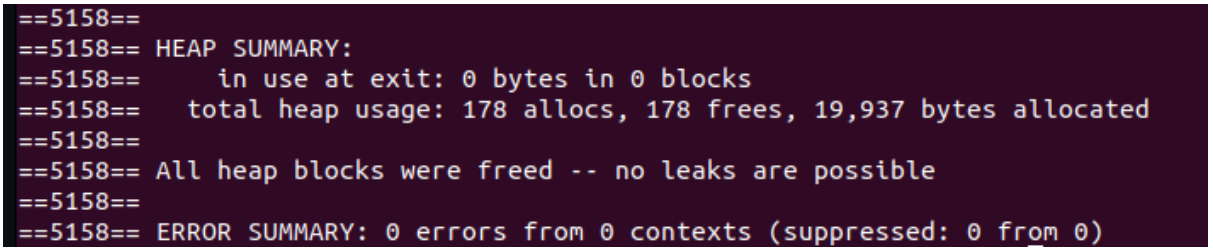
```
    char* code = (char*)malloc(sizeof(char) * 8);
    while (!feof(char_arq)) {
        char arq_char = fgetc(char_arq);
        if (arq_char != '\n') {
            accessDict(codedict, (int)arq_char, code);
            printf("%s", code);
            fwrite(code, sizeof(char), strlen(code),
char_bin);
        }
    }
    printf("\n");
    free(code);
    close_file(char_bin);
    close_file(char_arq);
}
```

Saída da execução do programa



Além disso, também é criado um arquivo chamado “output.bin”, a partir do arquivo de entrada, “input.txt”.

Saída da execução do Valgrind



Valgrind foi utilizado para constatar leaks de memória, e todas as memórias alocadas foram devidamente liberadas, vide imagem.

Conclusão

Em geral, o trabalho encontra-se funcional. Alguns módulos apresentaram coesão fraca, como o de fila de prioridades, que trata de vários conceitos. Poderia ter sido realizado mais tratamento de erro, com impressão do problema no stderr. Os caracteres aceitos foram somente aqueles da tabela ASCII tradicional (levando em conta um char com sinal, de 128 possíveis valores positivos (ou 0)). Caso exista algum valor fora desse escopo, ele é ignorado pelo programa, que não o conta, emite mensagem de erro e finaliza e execução (para não causar problemas de overflow). Foram realizadas tentativas de adaptar para funcionar até 255, mas não foram bem sucedidas.