SCC0220 - Laboratório Introdução à Ciência da Computação II

Relatório de execução do trabalho prático 7

Alunos	**NUSP**
Alec Campos Aoki	15436800
Juan Henrique Passos	15464826

Trabalho 7 – De volta às raízes

Radix Sort

Comentário

O trabalho prático de volta às raízes consistia em um baralho especial de *k* cartas, sendo *k* um número inteiro entre 0 e 10^6, que possuem valores de tamanho *n*, sendo 0 < n < 150, sendo esses valores, do menor para o maior, *4, 5, 6, 7, Q, J, K, A, 2, 3*. O baralho é formado por 4 naipes, ouro, espadas, copas e paus. Logo, será fornecido *k* e *n*, e por conseguinte, *k* cartas com seus determinados valores de tamanho *n*, além de seu naipe. O objetivo deste trabalho é ordenar as cartas de acordo com seus valores e naipes (os naipes sendo mais significativos, e os valores mais à esquerda sendo os próximos mais significativos).

Para modelar o problema, foram utilizados dois algoritmos de ordenação, o *radixsort* e o *stoogesort*. O *radixsort* consiste em armazenar elementos ordenadamente com base no valor de seus dígitos, ao invés de compará-los. Dessa forma, é possível chegar em uma complexidade linear, pois não há comparações entre os elementos, prática que exige uma complexidade mínima de $O(n)$.

O algoritmo funciona da seguinte forma:

1. Encontrar o maior elemento do vetor;
2. Descobrir quantos dígitos formam esse elemento;
3. Iterar por todos os dígitos de um número a começar pela unidade, ordenando os elementos com base no dígito atual;
4. Atualizar o vetor principal a cada casa de dígito ordenada.

Para o problema proposto, foi criado um vetor de cartas, que é uma *struct* formada pelo naipe e pelos valores da carta. Agora, para a implementação do *radixsort*, preferiu-se a implementação com um TAD fila, podendo ser criada 10 filas, que representam todos os valores possíveis por dígitos das cartas, e adicioná-los na fila em ordem crescente, ou seja, a posição 0 para o valor 4, até a posição 9 para o valor 3. Assim, a ordenação dos dígitos é realizada pelas filas, e atualizados no vetor principal ao esvaziá-las. A interação pelos valores é dada por um laço *for* externo a todas as operações de inserções e remoções das filas.

Esse algoritmo tem uma complexidade de tempo de $O(n.(k+b))$, onde *n* é o número de dígitos (valores), *k* é o número de elementos (cartas) e *b* é a base do sistema numérico que está sendo usado (possíveis valores das cartas).

Código

- **radixsort.c**

```c
#include"queue.h"

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
//Modularização
void radixsort(CARTA *baralho,int tam_baralho, int valor);
CARTA *ler_baralho(int tam_baralho, int carta);
int pos_carta(char valor);

int main(){
    int tam_baralho, valores; // Quantidade de quartas e quantidade de valores em uma carta.
    scanf("%d %d", &tam_baralho, &valores);

    CARTA *baralho = ler_baralho(tam_baralho, valores);

    for(int i = 0; i < tam_baralho; i++){
        printf("%s %s;", baralho[i].naipe, baralho[i].valor);
    }
    printf("\n");

    //Orderna o vetor pelos valores e naipes.
    radixsort(baralho, tam_baralho, valores);

    return 0;
}

CARTA *ler_baralho(int tam_baralho, int carta){
    CARTA *baralho = (CARTA *) malloc(tam_baralho*sizeof(CARTA));
    char naipe[10];
    for(int i = 0; i < tam_baralho; i++){
        // Alocar espaço para o valor + o \0.
        baralho[i].valor = (char*) malloc((carta + 1)*sizeof(char));
        scanf("%s", naipe);
        baralho[i].naipe = (char*) malloc((strlen(naipe) + 1)*sizeof(char));
        strcpy(baralho[i].naipe, naipe);
        scanf(" %s", baralho[i].valor);
    }
```

```
    return baralho;

}

void radixsort(CARTA *baralho,int tam_baralho, int valor){
    // Valor equivale ao numero de valores em cada carta.
    // Filas para guardar as cartas pelos seus valores.
    QUEUE **queue = (QUEUE**) malloc(10*sizeof(QUEUE*));
    for(int i = 0; i < 10; i++){
        queue[i] = queue_generate();
    }

    // Adicionar os numeros as filas pelos valores, começando pelo menos significativo.
    for(int i = 0; i < valor; i++){
        for(int j = 0; j < tam_baralho; j++){
                    // Adiciona a carta na fila do seu valor correspondente na posicao valor-i-1(LSD).
                   // Note que i-1 é o mais significativo. logo o valor-i-1 é o menos significativo.
            queue_push(queue[pos_carta((baralho[j].valor)[valor-i-1])], baralho[j]);
        }
        // Variavel para atualizar o vetor ao descarregar as filas.
        int index = 0;
        for(int j = 0; j < 10; j++){
            while(!queue_empty(queue[j])){
                baralho[index] = queue_pop(queue[j]);
                index++;
            }
        }

        printf("Após ordenar o %d° dígito dos valores:\n", valor-i);
        for(int i = 0; i < tam_baralho; i++){
            printf("%s %s;", baralho[i].naipe, baralho[i].valor);
        }
        printf("\n");
    }
    // Ordenação dos naipes
```

```
    for(int i = 0; i < tam_baralho; i++){
        int posicao;
        if(strcmp(baralho[i].naipe, "♦") == 0)
            posicao = 0;
        else if(strcmp(baralho[i].naipe, "♠") == 0)
            posicao = 1;
        else if(strcmp(baralho[i].naipe, "♥") == 0)
            posicao = 2;
        else
            posicao = 3;

        queue_push(queue[posicao], baralho[i]);
    }

    // Variavel para atualizar o vetor ao descarregar as filas.
    int index = 0;
    for(int j = 0; j < 4; j++){
        while(!queue_empty(queue[j])){
            baralho[index] = queue_pop(queue[j]);
            index++;
        }
    }
    printf("Após ordenar por naipe:\n");
    for(int i = 0; i < tam_baralho; i++){
        printf("%s %s;", baralho[i].naipe, baralho[i].valor);
    }

    //Desalocação de memoria.
    for(int i = 0; i < 10; i++){
        queue_erase(&queue[i]);
    }
    free(queue);
    queue = NULL;
}
// Função que decide a posição dos valores no vetor de filas.
int pos_carta(char valor){
    switch(valor){
        case '4':
```

```
            return 0;
        case '5':
            return 1;
        case '6':
            return 2;
        case '7':
            return 3;
        case 'Q':
            return 4;
        case 'J':
            return 5;
        case 'K':
            return 6;
        case 'A':
            return 7;
        case '2':
            return 8;
        case '3':
            return 9;
    }
}
```

- TAD queue.h

```
#ifndef QUEUE_H
    #define QUEUE_H

    typedef struct queue_ QUEUE;

    #include<stdbool.h>

    typedef struct carta_{
        char *naipe;
        char *valor;
    }CARTA;

    QUEUE *queue_generate();
    void queue_push(QUEUE *queue, CARTA carta);
    CARTA queue_pop(QUEUE *queue);
    bool queue_empty(QUEUE *queue);
```

```
    void queue_erase(QUEUE **queue);
    int queue_size(QUEUE *queue);
    bool queue_full(QUEUE *queue);
    void queue_print(QUEUE *queue);

#endif
```

- TAD fila.c

```
#include"queue.h"

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

typedef struct node_ NODE;

struct node_{
    NODE *next;
    CARTA carta;
};

struct queue_{
    NODE *front;
    NODE *back;
    int size;
};

QUEUE *queue_generate(){
    QUEUE *q = (QUEUE*) malloc(sizeof(QUEUE));
    if(q != NULL){
        q->front = NULL;
        q->back = NULL;
        q->size = 0;
    }
    return(q);
}

void queue_push(QUEUE *queue, CARTA carta){
    if(!queue_full(queue)){
```

```
        NODE *node = (NODE*) malloc(sizeof(NODE));
        node->next = NULL;
        if(queue->front == NULL){
            queue->front = node;
            queue->back = node;
        }
        else{
            queue->back->next = node;
            queue->back = node;
        }
        node->carta = carta;
        queue->size++;
    }
}

CARTA queue_pop(QUEUE *queue){
    CARTA carta;
    if(!queue_empty(queue)){
        NODE *erase_node;

        erase_node = queue->front;
        carta = erase_node->carta;
        queue->front = erase_node->next;

        free(erase_node);
        erase_node = NULL;
        queue->size--;

        if(queue->front == NULL)
            queue->back = NULL;

    }
    return carta;
}

bool queue_empty(QUEUE *queue){
    if(queue != NULL){
        return(queue->size == 0);
```

```
    }
    return true;
}

bool queue_full(QUEUE *queue){
    if(queue != NULL){
        NODE *node = (NODE*) malloc(sizeof(NODE));

        if(node == NULL) return true;

        free(node);
        node = NULL;

        return false;
    }
    return true;
}

void queue_erase(QUEUE **queue){
    if(!queue_empty(*queue)){
        NODE *queue_erase = (*queue)->front;
        while(queue_erase != NULL){
            (*queue)->front = (*queue)->front->next;
            free(queue_erase);
            queue_erase = (*queue)->front;
        }
        free(*queue);
        *queue = NULL;
    }
}

int queue_size(QUEUE *queue){
    if(queue != NULL){
        return(queue->size);
    }
    return(-1);
}
```

```c
void queue_print(QUEUE *queue){
    if(!queue_empty(queue)){
        NODE *node = queue->front;
        printf("Queue: \n");
        while(node != NULL){
            printf("%s %s\n", node->carta.naipe, node->carta.valor);
            node = node->next;
        }
        printf("\n");
    }
}
```

Saída

Caso 6 do Runcodes.

```
♣ 326K222A554K;♣ 32767J3QQ36K;♣ 32Q64QQ66QA7;♣ 32QK4A34575A;♣ 32QK657JQ32K;♣ 32J45JA7K4J3;♣ 32J4334AK32Q;♣ 32J6JKK25467;♣ 32J
336A6;♣ 32K2K2KKJJ54;♣ 32A57JJA2KK5;♣ 32A3456343A2;♣ 32227A2Q644J;♣ 323533374Q4J;♣ 323657J7Q5K7;♣ 3237J66J33AJ;♣ 323J7535A747
342K7246354;♣ 33563A74K7AQ;♣ 336KQAQ53J3Q;♣ 336A55JQ25QA;♣ 3363A733Q262;♣ 33766576A656;♣ 33J45A5AQQ56;♣ 33J7JJ6K2J73;♣ 33JQ7A
27;♣ 332424QJ7JQ5;♣ 332524KA5J6Q;♣ 3326KJK52JQ7;♣ 332766342J47;♣ 33232JK35525;♣ 3334QQ46K7AQ;♣ 333QQ576Q777;♣ 333QAQ77AA42;
Tempo de execucao: 0.047897s
```

Stooge Sort

Comentário

Stoogesort é um algoritmo de comparação recursivo que consiste de:

1. Comparar o primeiro e último elemento do vetor, ordenando-os;
2. Chamar a função recursivamente para os primeiros dois terços do vetor;
3. Chamar a função recursivamente para os últimos dois terços do vetor;
4. Chamar a função recursivamente novamente para os primeiros dois terços do vetor .

Note que o caso base é quando o vetor sendo analisado tem tamanho 2. A equação de recorrência desse algoritmo é $T(n) = T(\frac{2}{3}n) + c,\ n \geq 3$. Resolvendo pelo método da árvore de recorrência, obtemos $T(n) = O(n^{log(3)\frac{3}{2}}) = O(n^{2,7})$. Percebemos que o tempo de execução do *stoogesort* é muito maior que o do *radixsort*, pois ao invés de comparar uma grande quantidade de elementos, o *radixsort* não realiza comparações e analisa o valor de cada dígito/valor do elemento, sendo, portanto, mais adequado para ordenar cartas, onde o que importa é o valor total, e não o relativo a outra carta.

📄 Código

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <time.h>

#include "item.h"
#include "lista.h"

/*funções do timer*/
typedef struct{
   clock_t start;
   clock_t end;
}Timer;
void start_timer(Timer *timer);
double stop_timer(Timer *timer);

typedef struct carta_{
   char naipe[3];
   char *valores;
} CARTA;

void ler_cartas(CARTA baralho[], int quantCartas, int quantValores);
void imprimir_baralho(CARTA baralho[], int quantCartas);
bool comparar_cartas(CARTA carta1, CARTA carta2, char naipes[][4], char valores[], int
quantValores); //true: carta1 < carta2
void stoogeSort(CARTA baralho[], int inicio, int fim, int quantValores, char naipes[][4],
char valores[]);

int main(void){
   /*vetores auxiliares para acharmos os valores de uma carta*/
   char naipes[4][4];
   strcpy(naipes[0], "♦");
   strcpy(naipes[1], "♠");
   strcpy(naipes[2], "♥");
   strcpy(naipes[3], "♣");
   char valores[10];
```

```
    valores[0] = '4';
    valores[1] = '5';
    valores[2] = '6';
    valores[3] = '7';
    valores[4] = 'Q';
    valores[5] = 'J';
    valores[6] = 'K';
    valores[7] = 'A';
    valores[8] = '2';
    valores[9] = '3';

    int quantCartas, quantValores;
    scanf("%d %d", &quantCartas, &quantValores);

    CARTA baralho[quantCartas];
    ler_cartas(baralho, quantCartas, quantValores);

    Timer tempoTimer;
    double tempoExec;
    start_timer(&tempoTimer);

    stoogeSort(baralho, 0, quantCartas-1, quantValores, naipes, valores);

    tempoExec = stop_timer(&tempoTimer);

    imprimir_baralho(baralho, quantCartas);

    printf("\nTempo de execucao: %lfs\n", tempoExec);

    return 0;
}

void ler_cartas(CARTA baralho[], int quantCartas, int quantValores){
    for(int i=0; i<quantCartas; i++){
        baralho[i].valores = (char *)malloc((quantValores+1)*sizeof(char));
        if(baralho[i].valores == NULL) exit(1);
        scanf(" %s %s", baralho[i].naipe, baralho[i].valores);
    }
```

```
    return;
}

void imprimir_baralho(CARTA baralho[], int quantCartas){
    for(int i=0; i<quantCartas-1; i++){
        printf("%s %s;", baralho[i].naipe, baralho[i].valores);
    }
    printf("%s %s;\n", baralho[quantCartas-1].naipe, baralho[quantCartas-1].valores);

    return;
}

bool comparar_cartas(CARTA carta1, CARTA carta2, char naipes[][4], char valores[], int
quantValores){
    int naipeCarta1=0, naipeCarta2=0;
    while(strncmp(carta1.naipe, naipes[naipeCarta1], 3) != 0) naipeCarta1++;
    while(strncmp(carta2.naipe, naipes[naipeCarta2], 3) != 0) naipeCarta2++;

    if(naipeCarta1 < naipeCarta2) return true;
    else if(naipeCarta1 > naipeCarta2) return false;
    else{
        /*naipes iguais, checar os valores das cartas*/
        for(int i=0; i<quantValores; i++){
            int valorCarta1=0, valorCarta2=0;
            while(carta1.valores[i] != valores[valorCarta1]) valorCarta1++;
            while(carta2.valores[i] != valores[valorCarta2]) valorCarta2++;

            if(valorCarta1 < valorCarta2) return true;
            else if(valorCarta1 > valorCarta2) return false;
        }
    }

    return false;
}

void stoogeSort(CARTA baralho[], int inicio, int fim, int quantValores, char naipes[][4],
char valores[]){
```

```
    if(comparar_cartas(baralho[fim], baralho[inicio], naipes, valores, quantValores)){
        CARTA cartaAux = baralho[inicio];
        baralho[inicio] = baralho[fim];
        baralho[fim] = cartaAux;
    }

    if(inicio + 1 >= fim) return;
    //else:
    int t = ((fim-inicio+1)/3);
    stoogeSort(baralho, inicio, fim-t, quantValores, naipes, valores);
    stoogeSort(baralho, inicio+t, fim, quantValores, naipes, valores);
    stoogeSort(baralho, inicio, fim-t, quantValores, naipes, valores);

    return;
}
```

Saída

Caso 6 do runCodes.