**SCC0220 - Laboratório Introdução à Ciência da Computação II**

**Relatório de execução da aula prática 6**

**Alunos NUSP**

Juan Henriques Passos 15464826

Alec Campos Aoki 15436800

**Trabalho prático 6 – Ordenação**

**Heapsort**

**🡺 Comentário**

O exercício consistia em ordenar pratos, conforme a sua prioridade e tempo de preparo, sendo que deveriam estar ordenados de forma que o prato com a maior prioridade estivesse no início, e caso pratos tivessem a mesma prioridade, tem mais prioridade aquele que tiver o menor tempo e preparo. Foi usado o *heapsort* para a ordenação dos pratos. *Heapsort* é uma técnica de ordenação baseada na estrutura de dados heap.

A estrutura da heap consiste de uma árvore binária (se divide em no máximo 2 elementos por vez) com ordem (pai > filhos (heap máxima) ou pai < filhos (heap mínima)) e forma (se o último nível da árvore não estiver completo, todos seus elementos devem estar o máximo à esquerda possível). Podemos representar a heap como um vetor.

O algoritmo do *heapsort* funciona da seguinte forma:

1. Constuir uma heap máxima com os elementos do vetor;
2. Trocar a raiz da heap (maior elemento, primeiro elemento do vetor) pelo menor (última posição do vetor);
3. Diminuir o tamanho da heap por um;
4. Rearranjar a heap;
5. Repetir o processo até que o vetor esteja ordenado.

O processo de construir a heap tem complexidade , já que percorre de forma ascendente os primeiros nós executando o rearranjar\_heap (estamos tornando as sub-árvores da heap em heaps máximas, e ao chegarmos na raiz, tornamos a árvore inteira uma heap máxima). O processo rearranjar\_heap tem complexidade e consiste de verificar se a ordem da árvore está coerente com aquela da heap máxima (e fazendo swap entre pais e fihos caso não esteja). Temos, então, que a complexidade do *heapsort* é .

O exercício consistia em ordenar pratos, conforme a sua prioridade e tempo de preparo, sendo que deveriam estar ordenados de forma que o prato com a maior prioridade estivesse no início, e caso pratos tivessem a mesma prioridade, tem mais prioridade aquele que tiver o menor tempo e preparo. Foi usado o *heapsort* para a ordenação dos pratos. *Heapsort* é uma técnica de ordenação baseada na estrutura de dados Heap Binária. Pode ser visto como uma otimização sobre o *selection sort*, onde primeiro encontramos o elemento máximo (ou mínimo) e o trocamos pelo último (ou primeiro). Repetimos o mesmo processo para os elementos restantes. No *Heapsort*t, usamos a Heap Binária para que seja possível encontrar e mover rapidamente o elemento máximo ( em vez de ) e, portanto, atingir a complexidade de tempo .

**🡺 Código** (página seguinte)

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<stdbool.h>

#include<time.h>

typedef struct prato\_{

int prioridade;

int tempo;

char \*prato;

}PRATO;

// Modularização

PRATO \*leitura\_dados(int tam);

bool menos\_prioritario(PRATO prato1, PRATO prato2);

void desalocacao\_memoria(PRATO \*\*vet, int tam);

void heapsort(PRATO \*vet, int n);

void swap(PRATO \*i, PRATO \*j);

void rearranjar\_heap(PRATO \*vet, int i, int tam\_heap);

int main(){

int quant\_pratos;

PRATO \*pratos;

// Quantidade de pratos;

scanf("%d", &quant\_pratos);

// Captura o tempo inicial

clock\_t inicio = clock();

pratos = leitura\_dados(quant\_pratos);

heapsort(pratos, quant\_pratos);

// Captura o tempo final

clock\_t fim = clock();

// Calcula o tempo de execução em segundos

double tempo\_execucao = (double)(fim - inicio) / CLOCKS\_PER\_SEC;

// Exibe o tempo de execução

printf("Tempo de execução: %f segundos\n", tempo\_execucao);

for(int i = 0; i < quant\_pratos; i++){

printf("%s\n", pratos[i].prato);

}

desalocacao\_memoria(&pratos, quant\_pratos);

return(0);

}

// Funcao que le os dados e aloca espaço necessario, sem desperdicio de memoria.

PRATO \*leitura\_dados(int tam){

PRATO \*pratos = (PRATO\*) malloc(tam\*sizeof(PRATO));

if(pratos == NULL){

printf("Erro ao alocar memoria\n");

exit(1);

}

for(int i = 0; i < tam; i++){

int prioridade, tempo;

char leitura[52], \*prato;

scanf("%d %d %s", &prioridade, &tempo, leitura);

pratos[i].prioridade = prioridade;

pratos[i].tempo = tempo;

// Guarda-se exatamente o espaço necessário para guardar nome.

prato = (char\*) malloc((strlen(leitura) + 1)\*sizeof(char));

strcpy(prato, leitura);

pratos[i].prato = prato;

}

return pratos;

}

// Função que analisa se o prato1 tem menos prioridade que o prato2.

bool menos\_prioritario(PRATO prato1, PRATO prato2){

if(prato1.prioridade == prato2.prioridade)

if(prato1.tempo > prato2.tempo) return true;

else return false;

else if(prato1.prioridade > prato2.prioridade) return false;

else return true; // Prato 1 possui menos prioridade que o prato 2.

}

// Libera memoria armazenada, tanto do campo prato da struct, quanto do vetor do tipo PRATO.

void desalocacao\_memoria(PRATO \*\*vet, int tam){

if(\*vet == NULL) return;

for(int i = 0; i < tam; i++){

free((\*vet)[i].prato);

(\*vet)[i].prato = NULL; // Boa prática

}

free(\*vet);

\*vet = NULL;

return;

}

void swap(PRATO \*i, PRATO \*j){

PRATO aux = \*i;

\*i = \*j;

\*j = aux;

}

void rearranjar\_heap(PRATO \*vet, int i, int tam\_heap){

int esq, dir, maior;

esq = 2\*i + 1;

dir = 2\*i + 2;

maior = i;

// Busca ajustar o filho da esquerda para ser menor que pai.

if(esq < tam\_heap && menos\_prioritario(vet[maior], vet[esq]))

maior = esq;

// Filho deve ser menor que o pai.

if(dir < tam\_heap && menos\_prioritario(vet[maior], vet[dir]))

maior = dir;

// Caso um dos filhos seja maior, sua posição é reposicionada, e busca-se averiguar se ela é a sua posição nessa árvore.

if(maior != i){

swap(&vet[maior], &vet[i]);

rearranjar\_heap(vet, maior, tam\_heap);

}

}

void heapsort(PRATO \*vet, int n){

int i;

// Começa em (n/2) - 1, pois a partir disso, são representadas as folhas das arvore binaria.

for(i = (n / 2) - 1; i >= 0; i--){

rearranjar\_heap(vet, i, n);

}

// Heap sort

for (int i = n - 1; i >= 0; i--) {

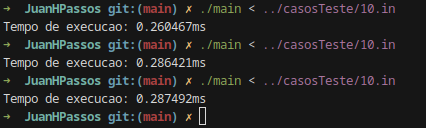
swap(&vet[0], &vet[i]);

rearranjar\_heap(vet, 0, i);

}

}

**🡺 Saída**

****

**Mergesort**

**🡺 Comentário**

Utilzamos também o algoritmo *Mergesort* para ordernar os pratos. O algoritmo *Mergesort* consiste de uma ordenação por intercalação e divisão e conquista. Suas equações de recorrência são (caso base) e . Utilizando o método da árvore de recorrência nessas equações, temos que a complexidade desse algoritmo é .Seu algoritmo consite de:

1. Quebrar o array na metade sucessivamente (recursivamente, no nosso caso) até que se obtenha arrays unitários;
2. Ordernar os dois arrays, intercalando-os e os colocando na ordem certa;
3. Repetir o passo anterior até retornarmos ao tamanho original do array.

Podemos perceber, pelo tempo de execução, que o *mergesort* acabando sendo, nessas implementações, mais rápido que o *heapsort*, provavelmente devido à sua implementação mais simples (o algoritmo *heapsort* exige várias operações custosas e repetitivas, enquanto as operações do *mergesort* são mais diretas e simples).

**🡺 Código**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <string.h>

#include <time.h>

typedef struct prato\_ PRATO;

struct prato\_{

char nome[51];

int prioridade;

int tempoPreparo;

};

/\*funções base\*/

bool comparar\_pratos(PRATO prato1, PRATO prato2);

void imprimir\_pratos(PRATO \*pontVetPratos, int tam);

/\*funcões mergesort\*/

void intercala(PRATO esquerda[], int tam\_esq, PRATO direita[], int tam\_dir, PRATO \*v);

void mergesort(PRATO \*v, int tam);

/\*funções do timer\*/

typedef struct{

clock\_t start;

clock\_t end;

}Timer;

void start\_timer(Timer \*timer);

double stop\_timer(Timer \*timer);

int main(void){

int quantPratos = 0;

scanf("%d", &quantPratos);

PRATO \*pontVetPratos = (PRATO \*)calloc(quantPratos, sizeof(PRATO));

if(pontVetPratos == NULL) exit(1);

/\*lendo a entrada\*/

for(int i=0; i<quantPratos; i++){

scanf("%d %d %s", &pontVetPratos[i].prioridade, &pontVetPratos[i].tempoPreparo, pontVetPratos[i].nome);

pontVetPratos[i].nome[strlen(pontVetPratos[i].nome)] = '\0';

}

mergesort(pontVetPratos, quantPratos);

imprimir\_pratos(pontVetPratos, quantPratos);

free(pontVetPratos);

return 0;

}

/\*mergesort\*/

/\*função para intercalar dois vetores ordenadamente (ordem crescente de prioridade)\*/

void intercala(PRATO esquerda[], int tam\_esq, PRATO direita[], int tam\_dir, PRATO \*v){

int i=0, e=0, d=0;

/\*escrevendo de volta no vetor v\*/

while((e < tam\_esq) && (d < tam\_dir)){

/\*vendo qual prato tem menor prioridade\*/

if(comparar\_pratos(esquerda[e], direita[d])){

v[i] = esquerda[e];

e++;

}

else{

v[i] = direita[d];

d++;

}

i++;

}

/\*caso não sobre elementos em um vetor para compararmos, escrevemos o restante no vetor v\*/

while(e < tam\_esq){

v[i] = esquerda[e];

e++;

i++;

}

while(d < tam\_dir){

v[i] = direita[d];

d++;

i++;

}

return;

}

void mergesort(PRATO \*v, int tam){

if(tam == 1){ //caso base

return;

}

int meio = tam/2;

PRATO \*esquerda, \*direita;

esquerda = (PRATO \*)calloc(meio, sizeof(PRATO));

direita = (PRATO \*)calloc((tam-meio), sizeof(PRATO));

/\*dividindo o vetor v[] em dois (esquerda e direita), e copiando os valores pra cada um\*/

int e=0, d=0;

for(int i=0; i<tam; i++){

if(i < meio){

esquerda[e] = v[i];

e++;

}

else{

direita[d] = v[i];

d++;

}

}

/\*chamando o mersort recursivamente, até tam == 1\*/

mergesort(esquerda, meio); //note que v[] é o vetor esquerda[]!

mergesort(direita, tam-meio);

/\*intercalando os vetores sucessivamente\*/

intercala(esquerda, meio, direita, tam-meio, v);

free(esquerda);

free(direita);

return;

}

//Retorna true se o prato1 deve ter maior prioridade que o prato2

bool comparar\_pratos(PRATO prato1, PRATO prato2){

if(prato1.prioridade < prato2.prioridade) return true;

else if(prato1.prioridade == prato2.prioridade){

if(prato1.tempoPreparo > prato2.tempoPreparo) return true;

}

return false;

}

void imprimir\_pratos(PRATO \*pontVetPratos, int tam){

for(int i=0; i<tam; i++){

//printf("%d %d %s\n", pontVetPratos[i].prioridade, pontVetPratos[i].tempoPreparo, pontVetPratos[i].nome);

printf("%s\n", pontVetPratos[i].nome);

}

return;

}

void start\_timer(Timer \*timer){

timer->start = clock();

return;

}

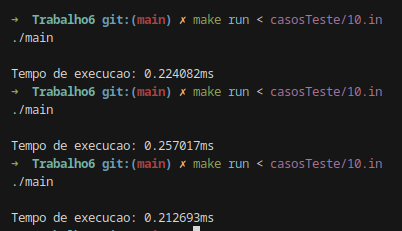
double stop\_timer(Timer \*timer){

timer->end = clock();

return((double)(timer->end - timer->start)) / CLOCKS\_PER\_SEC;

}

**🡺 Saída**

****