SCC0220 - Laboratório Introdução à Ciência da Computação II

Relatório de execução da aula prática 3

Alunos	**NUSP**
Felipe Camargo Cerri	15451119
Gabriel Campanelli Iamato	15452920

Exercício 3 – Seleção

Item a

Comentário

Foram implementadas duas formas de ordenação de vetores, o Selection Sort e o Shell Sort. O Shell Sort passou em todos os testes do RunCodes, enquanto o Selection Sort teve tempo de execução excedido em alguns testes.

O algoritmo de ordenação por seleção (Selection Sort) se baseia na ideia de percorrer o vetor a ser ordenado, buscando sempre o menor (ou maior) elemento e inserindo-o no início desse mesmo vetor ou em uma estrutura de dados auxiliar. Para implementá-lo, utilizamos dois loops: o primeiro itera sobre todos os elementos do vetor, armazenando em uma pilha o menor deles, o segundo reinicia as iterações até que todos os elementos estejam ordenados. Por simplicidade, optamos por marcar um item como “removido” ao invés de efetivamente retirá-lo do array principal. Além disso, como solicitado, a pilha utilizada foi implementada como um tipo abstrato de dado (TAD). Optamos por implementá-la de forma sequencial e dinâmica. Esse TAD possui as funções básicas de: criar, empilhar, desempilhar, verificar o topo da pilha, verificar o tamanho, checar se está cheia ou vazia, imprimi-la e apagá-la. Aproveitamos as características de um TAD para evitar vazamentos de memória e manipular os dados de forma segura.

Por outro lado, o algoritmo Shell Sort é uma implementação mais eficiente do Insertion Sort, que também foi implementado como base de estudos (comentado no código oficial), mas não analisado no relatório devido à proposta do exercício. No Insertion Sort, o primeiro elemento do array considera-se ordenado, e os próximos elementos são realocados no array de acordo com essa primeira posição já ordenada. Se o elemento a ser comparado for trocado de posição, realiza-se um “Shift Right” (move-se os elementos de índice maior para a direita). O Shell Sort utiliza uma lógica similar, porém divide o array em sublistas (*gaps*) e aplica o Insertion Sort, até que o *gap* seja 1 e aplica-se a ordenação no vetor inteiro.

Aqui pode-se visualizar o processo do Shell Sort: https://opendsa-server.cs.vt.edu/embed/shellsortAV .

Código

TAD Pilha Encadeada

```c
#ifndef PILHA_E
    #define PILHA_E
    #define ERRO -1
    #include <stdbool.h>

    typedef struct pair_ {
        char nome[51];
        int nota;
    }pair;

    typedef struct pilha_ PILHA;
    PILHA* pilha_criar(int tam);
    void pilha_apagar(PILHA** pilha);
    bool pilha_vazia(PILHA* pilha);
    bool pilha_cheia(PILHA* pilha);
    int pilha_tamanho(PILHA* pilha);
    pair pilha_topo(PILHA* pilha);
    bool pilha_empilhar(PILHA* pilha, pair dado);
    pair pilha_desempilhar(PILHA* pilha);
    void pilha_print(PILHA* p);
#endif
```

Selection Sort

```c
void selection_sort(pair *elenco, int n){
    PILHA *pilha = pilha_criar(n);
    pair min = {"\0", INT_MAX};
    int posicao , i, j;
    for (j =0; j<n; j++) {
        for (i =0; i<n; i++) {
            if ((min.nota > elenco[i].nota) || (min.nota == elenco[i].nota && strcmp(min.nome, elenco[i].nome) < 0)) { //encontra a menor nota
                min = elenco[i];
                posicao = i;
            }
        }
        elenco[posicao].nota = INT_MAX; //"remove" menor nota do vetor evitando notas iguais nos subsequentes minimos
        pilha_empilhar(pilha, min);

        strcpy(min.nome, "\0");
        min.nota = INT_MAX;
    }
    i = 0;
    while(!pilha_vazia(pilha)){
        elenco[i] = pilha_desempilhar(pilha);
        //printf("\n%s %d", elenco[i].nome, elenco[i].nota);
        //pilha_print(pilha);
        i++;
    }
    pilha_print(pilha);
    pilha_apagar(&pilha);
}
```

Shell Sort

```c
void shell_sort(pair *elenco, int tamanho_elenco){
    // No sheel sort, o algoritmo divide em sublistas(gaps) e aplica o insertion sort, até que o gap seja 1 e aplique-se o insertion sort no vetor inteiro
    int gap = tamanho_elenco / 2;
    int j;
    pair elemento_aux;
    while(gap>0){
        // começa o loop a partir do gap e vai até o final do array
        for(int i = gap; i <tamanho_elenco; i++){
            elemento_aux = elenco[i]; // variável auxiliar para pegar o elemento atual
            j = i;
            // faz a verificação se o elemento_aux é maior que o elemento comparado pelo gap. Se for maior, atribui ao elemento j o elemento da distância de gap.
            while((j>=gap) && ((elemento_aux.nota > elenco[j-gap].nota) || ((elemento_aux.nota == elenco[j-gap].nota) && (strcmp(elemento_aux.nome, elenco[j-gap].nome)<0)))){
                elenco[j] = elenco[j-gap];
                j -= gap;
             }
            //Atribui-se o elemento original a sua posição correta (para que ele não seja perdido)
        elenco[j]= elemento_aux;
    }
    //repete-se esse processo
    gap /= 2;
    }
}
```

Saída

Para comprovar a eficiência dos dois métodos de ordenação, foi utilizada a biblioteca <time.h> da linguagem C para medir o tempo de execução do algoritmo. Como os resultados foram testados no RunCodes, serão apresentados apenas os tempos de execução de cada algoritmo, com casos de teste do RunCodes:

```
11
Alisson 89
Danilo 81
Marquinhos 87
Magalhaes 85
Arana 79
André 81
Guimaraes 84
Paqueta 82
Rodrygo 85
Vini-Jr. 90
Hendrick 68
Tempo de Execução Selection Sort: 0.000004
Tempo de Execução Shell Sort: 0.000001(base)
```

Tempo de execução mais curto (11 casos) : Shell Sort

```
Rivaldinho 64
Quaresma 72
Cunha 78
Volpi 67
Muniz 69
Lodi 79
Nunes 81
Pedrao 70
Morato 75
Dante 78
Becao 77
Silva 68
Tempo de Execução Selection Sort: 0.000405
Tempo de Execução Shell Sort: 0.000066(base)
```

Tempo de execução mais curto (386 casos): Shell Sort

```
Toporkiewicz 55
Florentino 80
Qenaj 59
Gavi 83
Dennis 65
Arigoni 63
Bykowski 58
Singh 60
Magnin 56
Amin 62
Davis 73
Diale 69
Tempo de Execução Selection Sort: 0.665433
Tempo de Execução Shell Sort: 0.005830(base)
```

Tempo de execução mais curto (17326 casos): Shell Sort

A partir dos testes, percebe-se que Shell Sort é um algoritmo mais eficiente do que o Selection Sort no que tange tempo de execução, e apresentou eficiência mais rápida nos casos testados. A diferença de processamento ficou mais clara ao passo que o tamanho dos casos teste aumentava.