UNIVERSIDADE FEDERAL ALFENAS (UNIFAL)
Bacharelado em Ciência da Computação

Disciplina	Método de realização	Data de apresentação
DCE792 - AEDS 2	Código	23/10/2024 às 23h59
Professor		
Iago Augusto de Carvalho (iago.carvalho@unifal-mg.edu.br)		

Trabalho prático 3 - Ordenação de *structs*

Alunos:

Thallysson Luis Teixeira Carvalho - 2024.1.08.022 Luis Renato Goulart - 2023.1.08.049 Renan Catini Amaral -2024.1.08.042

Arquivo “funçõesAux.c”

O arquivo “funções.aux” possui as seguintes funções:

- **troca(tipo void):** função para trocar a posição de dois ponteiros em um vetor.
- **normaliza_caracter(tipo wchar_t):** função para padronizar caracteres que possuem algum tipo de acento ou outra formatação para regular a ordem em que deve se ordenar o vetor(ordem alfabética dos nomes dos jogadores).
- **ComparaSemAcentoMinus(tipo int):**realiza a comparação entre duas strings (`str1` e `str2`) com base na ordem alfabética. Ela normaliza os caracteres para desconsiderar diferenças de acentuação e trabalha com caracteres padronizados. Ela compara duas strings em ordem alfabética ignorando acentuação e diferenças entre maiúsculas e minúsculas, retornando um número positivo se a primeira string vir depois da segunda em ordem alfabética ou um número negativo se a primeira string vir antes da segunda em ordem alfabética ou zero se as strings forem consideradas iguais.

Arquivo “funcoesOrdenacao.c”

O arquivo “funcoesordenacao.c” possui as seguintes funções:

- **mediana(tipo void):** esta função tem como objetivo selecionar a mediana de três elementos em um vetor de ponteiros para objetos(struct) do tipo “jogador”. Ela faz isso comparando os elementos localizados nos índices **inf** e **sup** e o valor médio **mid** = (inf + sup) / 2. Após identificar qual desses três elementos é a mediana, coloca-o na posição sup. Além disso, ela também conta o número de comparações e trocas realizadas durante o processo, atualizando os valores apontados por comparações e trocas. Em resumo a função possui como entrada rês elementos do vetor (vet[inf], vet[sup], vet[mid]) e saída é o vetor reorganizado de forma que o menor valor está em vet[inf],A mediana está em vet[sup] e O maior valor está em vet[mid] (em um caso alternativo).
- **particao(tipo int):** é um complemeto ao algoritmo de ordenação “QuickSort”. Sua função é particionar o vetor em duas partes em torno de um pivô, de forma que todos os

elementos menores que o pivô fiquem à esquerda e todos os maiores fiquem à direita. No final, a posição correta do pivô no vetor ordenado é retornada. Além disso, a função contabiliza o número de comparações e trocas realizadas durante o particionamento. Em resumo essa função divide o vetor em duas partes(elementos menores que o pivô à esquerda e maiores à direita), move o pivô para sua posição correta no vetor ordenado e retorna o índice do pivô após a divisão.

- **InsertionSort(tipo Void):**implementa o algoritmo de ordenação por inserção (Insertion Sort) para um vetor de ponteiros para objetos do tipo `jogador`. Além de realizar a ordenação, a função conta o número de comparações e trocas realizadas durante o processo, armazenando esses valores em variáveis externas. Divide vetor em duas partes,no início do vetor está a parte ordenada e no restante a parte não ordenada .A cada iteração, o algoritmo pega um elemento da parte não ordenada (chave) e insere na posição correta da parte ordenada.

- **QuickSort(tipo void):** implementa o algoritmo "QuickSort", que é um método de ordenação baseado na estratégia de divisão e conquista. Ela organiza os elementos de um vetor de ponteiros para estruturas do tipo "jogador" de forma recursiva, enquanto também conta o número de comparações e trocas realizadas durante o processo. Ela verifica se o intervalo é válido (inf < sup), escolhe o pivô usando a mediana de três elementos, particiona o vetor em torno do pivô (elementos menores a esquerda e maiores a direita) e ordena as partições. O número de comparações e trocas realizadas é contabilizado diretamente nas funções auxiliares "mediana"e "particao".

- **ContingSort(tipo void):** função auxiliar para o algoritmo RadixSort que implementa o algoritmo de ordenação "Counting Sort" adaptado para ordenar
  um vetor de ponteiros para estruturas do tipo jogador. Ela é usada principalmente para ordenar os jogadores com base em um índice específico (iChar) do nome de cada jogador, desconsiderando acentos, diferenciação entre maiúsculas e minúsculas, e normalizando caracteres especiais. Além disso, a função contabiliza o número de trocas realizadas durante o processo. O "Counting Sort" é um algoritmo estável e eficiente para casos onde o domínio dos valores (neste caso, os caracteres normalizados) é limitado e conhecido. Ele mantém a estabilidade de dois jogadores com o mesmo caractere no índice "iChar" que aparecem na mesma ordem relativa no vetor ordenado.
- **RadixSort(tipo void):** implementa o algoritmo "Radix Sort" adaptado para ordenar um vetor de ponteiros para estruturas jogador com base nos nomes. Ela combina o algoritmo "Counting Sort" para ordenar os caracteres de cada posição (coluna) dos nomes, de forma incremental, e usa o "Insertion Sort" no final para finalizar a ordenação de uma lista já quase ordenada. Além disso, a função rastreia o número de comparações, trocas e a memória adicional utilizada durante o processo. Tal algoritmo ordena os nomes com base nos caracteres individuais, começando pelos menos significativos mantendo a estabilidade do processo (elementos com caracteres iguais mantêm a ordem original) e é eficiente para conjuntos grandes de strings com tamanhos uniformes, pois combina a eficiência de O(n) do "Counting Sort" com uma etapa final linear (quase ordenada).Em resumo a função "RadixSort" ordena os nomes dos jogadores em ordem alfabética, tratando a normalização de caracteres (maiúsculas, acentos, e caracteres especiais) e, para o tamanho variável dos

nomes, ela usa “Counting Sort” para ordenar progressivamente por cada caractere e finaliza com “Insertion Sort” para melhorar a eficiência. O rastreamento de trocas, comparações e memória permite avaliar o desempenho do algoritmo.

Arquivo “ordenacao.c”

É a função principal (int main) que realiza a leitura de um arquivo CSV com dados de jogadores, ordena os jogadores usando diferentes algoritmos de ordenação implementados em outros arquivos e exibe o resultado ordenado junto com informações sobre o desempenho do algoritmo escolhido

(tempo de execução, número de comparações, trocas e memória adicional utilizada). Em resumo a função principal lê os dados dos jogadores de um arquivo CSV., escolhe um algoritmo de ordenação com base no argumento fornecido,ordena os jogadores de acordo com seus nomes,exibe os jogadores ordenados e as métricas de desempenho do algoritmo e libera toda a memória alocada dinamicamente.

Arquivo “ordenacao.h” e Estruturas de Dados Utilizadas

Arquivo cabeçalho onde possui as funções utilizadas e a “struct” **jpgador** que é separa da seguinte forma:

- wchar_t nome[41]: Nome do jogador
- wchar_t posicao[41]: Posição em que o jogador atua (e.g., goleiro, atacante). • wchar_t naturalidade[41]: Local de nascimento ou nacionalidade do jogador.
- wchar_t clube[41]: Clube em que o jogador atua.
- int idade: Idade do jogador.

A estrutura central do programa é baseada em **vetores (arrays ou listas estáticas/sequenciais)** de ponteiros para a struct **jogador**, armazenando as informações dos jogadores em um formato que permite fácil manipulação e ordenação. Cada posição do vetor armazena um ponteiro para um jogador, reduzindo o custo de operações como troca, já que manipula apenas ponteiros e não a struct completa.

Como o código inclui funções que utilizam recursividade, também é possível destacar, de forma implícita, o uso da estrutura de dados **pilha**. Essa estrutura é gerenciada pela memória do computador e é essencial para controlar as chamadas de funções recursivas, garantindo a execução correta da lógica dos algoritmos.

Arquivo "MAKEFILE"

É o arquivo usado para automatizar o processo de compilação de um programa em C. Ele organiza a compilação e a linkagem dos arquivos fonte, além de limpar os arquivos temporários gerados no processo. Ele define como os arquivos ".c" serão transformados em arquivos executáveis, garantindo que tudo seja compilado corretamente.
Para executar os comandos é necessário realizar os seguintes passos:

- Abra o terminal e navegue até o diretório do projeto.
  - **cd /caminho/para/o/projeto** • Compile o projeto com o Makefile.
  - **make all**
- Execute o programa usando os argumentos:
  - 1 para InsertionSort
  - 2 para QuickSort
  - 3 para RadixSort
  - Use o comando "**./ordenacao n**"**,** onde n é dos 3 argumentos para escolher o método de ordenação previamente definido.

COMPLEXIDADE DOS ALGORITMOS UTILIZADOS

InsertionSort

- No melhor caso, quando os elementos já estão ordenados, sua complexidade é **O(n)**, pois o algoritmo realiza apenas uma comparação por elemento. No pior caso, quando os elementos estão ordenados de forma inversa, a complexidade é **$O(n^2)$**, devido à necessidade de realizar comparações e deslocamentos para cada elemento no vetor. Sua complexidade de memória é **O(1)** pois não usa memória adicional significativa além do array de entrada.

QuickSort

- Sua complexidade no melhor caso possui **O(n*log n)**, quando as partições são bem balanceadas. No pior caso possui complexidade **$O(n^2)$**, quando as partições são muito desbalanceadas. Sua complexidade de memória é **O(1)** pois não usa memória adicional significativa além do array de entrada.

ContingSort(auxiliar)

- Sua complexidade é **O(n + k)**, onde **n** representa o número de elementos no vetor a ser ordenado e **k** é o valor máximo do domínio, ou seja, o maior valor que um caractere pode assumir após a normalização. Essa complexidade reflete o fato de que o algoritmo percorre o vetor de entrada (**n**) e também realiza operações baseadas no tamanho do domínio (**k**), como a inicialização e atualização do vetor de contagem.

RadixSort

- Sua complexidade é O(R*(n+M)), onde R é o comprimento do menor nome, `n` é o número de jogadores (tamanho do vetor a ser ordenado) e M é o tamanho do alfabeto. O algoritmo realiza um **CountingSort** para cada caractere (até o menor comprimento, **R**), e cada execução do **CountingSort** tem complexidade **O(n + M)**. Por isso, a complexidade total do **RadixSort** é determinada pelo produto de **R** e **(n + M)**. Sua complexidade de memória é **O(n)** pois usa arrays auxiliares para contagens e buckets."

Desempenho de cada algoritmo

Primeira linha refere-se ao tempo de execução(em milisegundos)
Segunda linha refere-se ao número de comparações executadas
Terceira linha refere-se ao número de trocas executadas
Quarta linha refere-se a memória adicional gasta

InsertionSort

```
0.013133
318323.000000
318323.000000
0.000000
```

QuickSort

```
0.001384
13114.000000
7342.000000
0.000000
```

RadixSort

```
0.000818
1961.000000
6557.000000
9684.000000
```

Conclusão

Conclui-se que o método mais rápido(gasta menos tempo de execução) é o RadixSort, entretanto ele também é o que mais consome memória auxiliar para ser executado. Em relação ao número de operações de comparação e troca realizadas, a ordem decrescente (do maior para o menor) dos algoritmos é: **InsertionSort**, **QuickSort** e, por último, o **RadixSort**, que realiza o menor número dessas operações.