Relatório - REO 2

Arthur Henrique Sousa Cruz

Universidade Federal de Lavras

1 Execução

Para a geração de um arquivo de teste aleatório utilize os seguintes comandos: (Observação: Se novos testes forem gerados, testes que já existiam e tinham mesmo número de elementos serão excluídos.)

g++ -Wall -Wconversion gerador_de_testes.cpp -o gerador

./gerador

No arquivo **main.cpp** configure as variáveis **qtd_entradas**, **tamanhos_entradas**, **qtd_buscas** para o número de arquivos de entrada (padrão: 3, {10, 100, 1000}, 5, respectivamente), os tamanho de arquivos de entrada e o número de buscas a serem feitas. Após isso utilize os seguintes comandos para compilar e executar o código:

make

./busca-bin.app

2 Organização de arquivos

Os arquivos deste projeto foram organizados da seguinte forma:

- **busca.hpp/busca.cpp**: Implementação de uma classe para o vetor de buscas;
- **main.cpp**: Arquivo de testes;
- **gerador.cpp**: Arquivo gerador de testes;

3 Classe VetorBusca

Para facilitar o uso das estruturas foi criada a classe VetorBusca (ao invés do uso de funções). A classe contém como variável o tamanho máximo (definido como 1000), o tamanho atual e o vetor propriamente dito.

Seus principais métodos são:

- **VetorBusca()**: Equivalente à função *inicializaVetor(int vetor[], int *tamanhoAtual);*
- **VetorBusca()**: Desutor da classe;
- **inserir(int elemento)**: Equivalente à função *inserirNoVetor(int vetor[], int elemento, int *tamanhoAtual);*
- **inserir_ordenado(int elemento)**: Equivalente à função *inserirOrdenadoNoVetor(int vetor[], int elemento, int *tamanhoAtual);*
- **busca_sequencial(int elemento)**: Equivalente à função *inserirOrdenadoNoVetor(int vetor[], int elemento, int *tamanhoAtual);*
- **busca_binaria(int elemento)**: Equivalente à função *buscaBinaria(int vetor[], int element);*
- **get_primeiro_elemento()**: Método auxiliar criado para testar a busca binária do primeiro elemento;
- **get_ultimo_elemento()**: Método auxiliar criado para testar a busca binária do último elemento;
- **get_string()**: Método auxiliar criado para verificar o conteúdo do vetor.

4 Testes

Os testes utilizados (criados com base na lista de atividades) foram enviados junto ao código fonte. Caso tenha seja gerado um novo teste com mesmo tamanho, ele será excluído.

4.1 Vetor não ordenado

A busca sequencial custa um número de operações igual a posição do elemento no vetor. Assim sendo, para os testes realizados, o número de operações para primeiro elemento, último elemento e três elementos internos aleatórios foi respectivamente:

- 10 números: 1, 10, 4, 2, 5;
- 100 números: 1, 100, 15, 32, 4;
- 1000 números: 1, 1000, 53, 410, 994;

O grande número de passos se deve à complexidade de $O(n)$ da busca sequencial. Por outro lado a inserção no vetor teve custo de $O(1)$, já que basta inserir na "posição tamanhoAtual"do vetor.

4.2 Vetor ordenado

O número de verificações realizadas para encontrar um elemento no vetor ordenado utilizando a busca binária para o primeiro elemento, último elemento e cinco elementos internos aleatórios foi respectivamente:

- 10 números: 3, 4, 3, 3, 4, 2, 3;
- 100 números: 6, 7, 4, 3, 7, 7, 7;

– 1000 números: 9, 10, 10, 10, 8, 5, 10;

Comparado à busca sequencial, o número de verificações realizadas é bem menor para a maior parte dos casos. Isso se deve ao fato de que a busca binária verifica apenas o elemento do meio do vetor e, caso não encontre o elemento na posição, diminui seu espaço de busca pela metade a cada iteração. Isso também faz com que sua complexidade seja de $\log n$ e não $O(n)$ como a sequencial. Por outro lado, para manter o vetor ordenado o custo do pior caso é de $O(n)$. Isso se deve ao fato de que, na implementação feita, todos os elementos maiores que o novo elemento devem ter suas posições alteradas para a posição seguinte. No pior caso a inserção ocorre na primeira posição e todo o vetor deve ser remanejado.