UNIVERSIDADE FEDERAL DE JUIZ DE FORA
23/10/20 - Trabalho Prático (Parte 1)
25/11/20 - Trabalho Prático (Parte 2)

JONAS GABRIEL MAIA BRUNO NEPOMUCENO (*201476026)*
DAVI CARDOSO DE ALMEIDA

Atividades realizadas por cada membro

Davi:
Quicksort
Contadores de cópia e comparação
Execução de testes
Documentação dos Ordenadores e Árvore
Análise dos resultados obtidos
RB Tree

Jonas:
InsertionSort
Leitura e escrita em arquivos
Cacheamento de dados de Livros
Benchmark dos ordenadores
Cronômetro de execução
Hashing
Documentação da Aplicação e Hashing

Estruturas de dados implementadas

Separamos as estruturas entre o sistema de apoio à execução controlada dos ordenadores a fim de registrar seu desempenho em testes dos próprios ordenadores. Sendo assim, dividimos essa seção entre as estruturas da aplicação e os próprios ordenadores.

Aplicação

O objetivo da aplicação era de forma rápida ler de uma fonte de mais de um milhão de registros informações sobre livros, e ordená-los realizando comparações de desempenho. De modo a evitar retrabalhos, deixar o código reutilizável e manter um baixo acoplamento entre os métodos da aplicação, separamos suas responsabilidades por camadas e arquivos. Enquanto as camadas seguem um propósito geral, os arquivos se destinam a resolver uma única responsabilidade na resolução da aplicação.

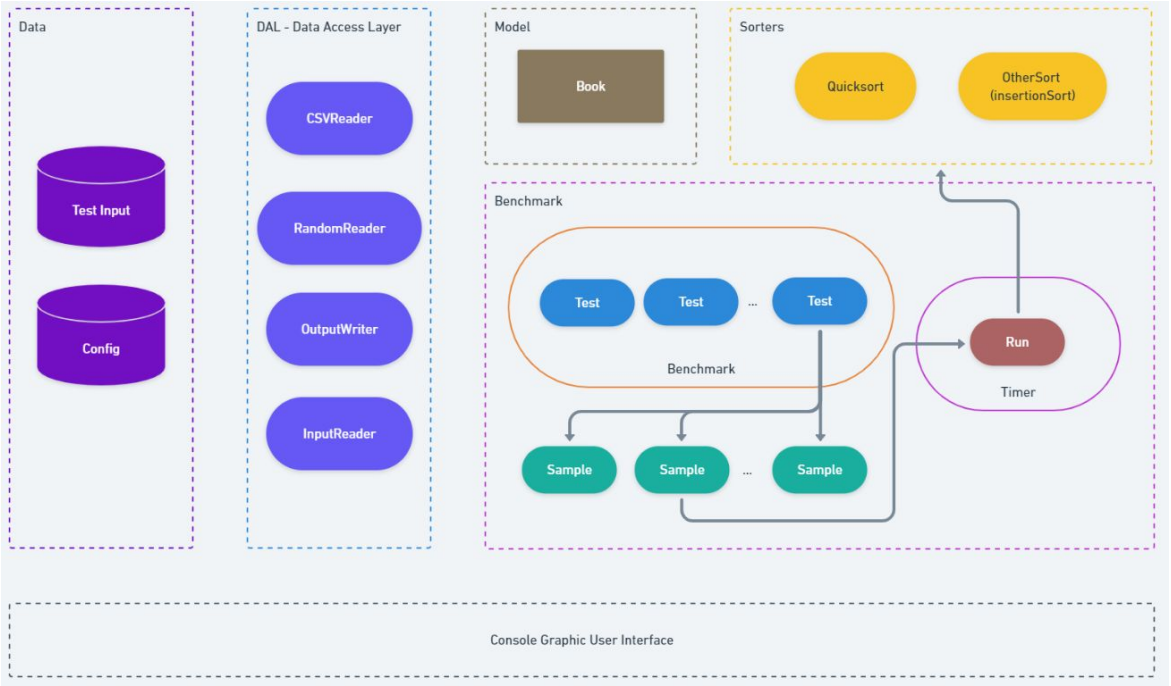


Figura 1: Arquitetura do projeto

A camada *Data (*Dados*)* se destina à camada onde os dados estão armazenados, responsável por prover os dados para a aplicação, tanto de configuração (Input) como de teste (Test Input). Os conjuntos de dados são arquivos de entrada, o de configuração sendo o *entrada.txt* e o de teste sendo o *dataset_simp_sem_descricao.csv*. A camada (*DAL - Data Access Layer*) cuida do acesso e manipulação dos dados. A fim de permitir o acesso rápido aos dados por parte das outras camadas do sistema, a camada de acesso a dados faz leitura dos dados uma única vez, mantendo um caching para amostragem. Essa camada trata também da transformação para um tipo significativo às operações, usando da camada dos modelos (*Model*) pra isso. Essa é responsável por guardar a representação virtual de entidades ao longo do ciclo de vida da aplicação.

Como o objetivo do programa é comparar o desempenho de ordenadores, existe uma camada que detém as execuções e métricas para mensurar o desempenho, aqui chamada de *Benchmark.* Essa recebe um conjunto de dados e, utilizando dos métodos de leitura aleatorizada da camada de acesso aos dados, retira amostras para testes. A partir desses testes, executa de forma cronometrada os ordenadores, registrando métricas de desempenho de ordenação. Cada benchmark executa testes para cada tamanho de amostra recuperada do arquivo de *Input*, gerando 5 conjuntos amostrais por tamanho de amostra. A partir da amostra, inicia a execução do código que gerencia (método Run) a ordenação cronometrada (método Timer). O cronômetro se baseia nas funções da biblioteca sys/time.h para iniciar o tempo de execução antes da ordenação iniciar e ao fim da ordenação. Ela também retorna quais foram o número de cópias no vetor e o número de comparações de elementos feito pelos ordenadores.

Ao fim do benchmark, quando todos os testes tiverem registrado os resultados das ordenações, terão sido agrupadas as seguintes informações da execução:

- Tamanho da amostra
    - Número da execução atual
    - Algoritmo executado de Ordenação:
        - Número de cópias realizadas
        - Número de comparações utilizadas
        - Tempo total da ordenação

Os resultados serão compilados em um arquivo de saída, saida.txt, que poderá ser encontrado na raíz do projeto.

Ordenadores

InsertionSort

Pior caso	Caso médio	Melhor caso	Estável
$O(n^2)$	$O(n^2)$	$O(n)$	Sim

É um algoritmo eficiente para problemas com uma entrada pequena, é mais eficiente que o bubble sort e normalmente mais que o selection sort, é indicado quando o arquivo está praticamente todo ordenado e possui poucas inserções.

A ideia principal é, dado um array de entrada, este é dividido em dois subarrays, o primeiro a esquerda formado pelo primeiro elemento e que se mantêm ordenado, o segundo são o restante dos elementos desordenados que vão sendo a cada iteração, selecionados como pivot e posicionados no primeiro subarray na posição correta até que todos os elementos do subarray desordenado estejam na posição correta.

```c
#include "sorters.h"

#include <string.h>

void insertionsort(Book **to_sort, int size, int copy_count, int compare_count)
{
    int i, j;
    Book *pivot;

    for (j = 1; j < size; j++)
    {
        pivot = to_sort[j];
        i = j - 1;

        while (i >= 0 && (strcmp(to_sort[i]->title, pivot->title) > 0))
        {
            compare_count++;
            to_sort[i + 1] = to_sort[i];
            copy_count++;
            i--;
        }

        to_sort[i + 1] = pivot;
        copy_count++;
    }
}
```

QuickSort

Pior caso	Caso médio	Melhor caso	Estável
O(n²)	O(nlogn)	O(nlogn)	Não

É um algoritmo que utiliza da abordagem de divisão e conquista assim como o merge sort, ele é composto basicamente por dois métodos, sort, partition.

A ideia principal do algoritmo é, através da divisão, um pivot é escolhido, o array de entrada é particionado em dois subarrays, na qual a primeira metade possui elementos menores ou igual ao pivot e a segunda maiores ou igual, na conquista, os dois subarrays são ordenados recursivamente. Segue os métodos abaixo:

Partition

```c
int partition(Book **to_sort, int low, int high, int copy_count, int compare_count)
{
    Book *pivot = to_sort[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)
    {
        int compare = strcmp(to_sort[j]->title, pivot->title);
        if (compare < 0 || compare == 0)
        {
            compare_count++;
            i++;
            swap(to_sort[i], to_sort[j], copy_count);
        }
    }
    swap(to_sort[i + 1], to_sort[high], copy_count);
    return (i + 1);
}
```

Escolhe o último elemento como pivot, este é colocado na posição correta do array, então todos os elementos menores que o pivot são postos à esquerda do pivot e todos maiores a direita.

Quick Sort

```c
void quicksort(Book **to_sort, int size, int copy_count, int compare_count)
{
    sort(to_sort, 0, size - 1, copy_count, compare_count);
}
```

Inicia o algoritmo com a chamada do sort a primeira vez passando o índice do primeiro e do último elemento do array de entrada.

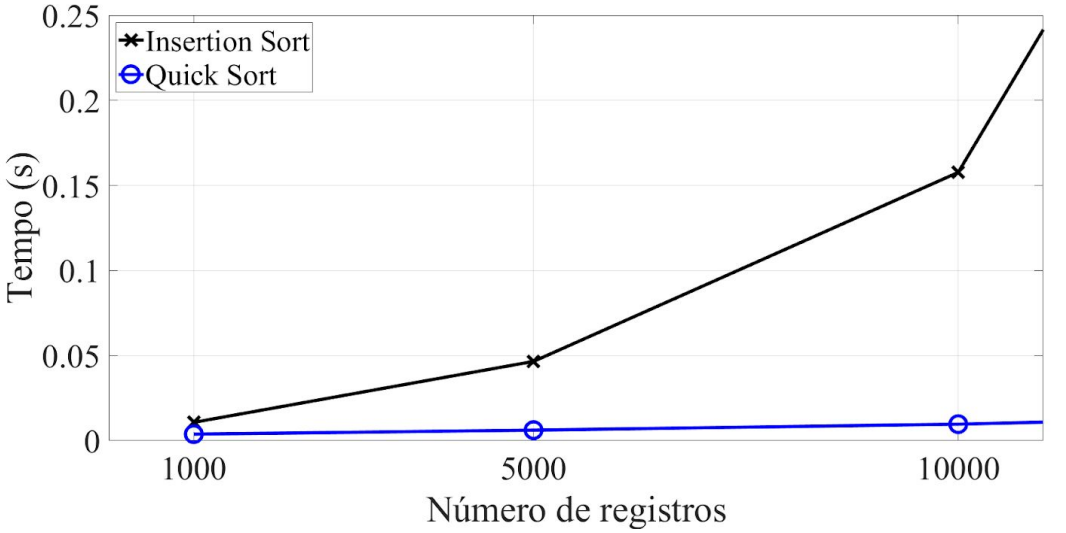
Sort

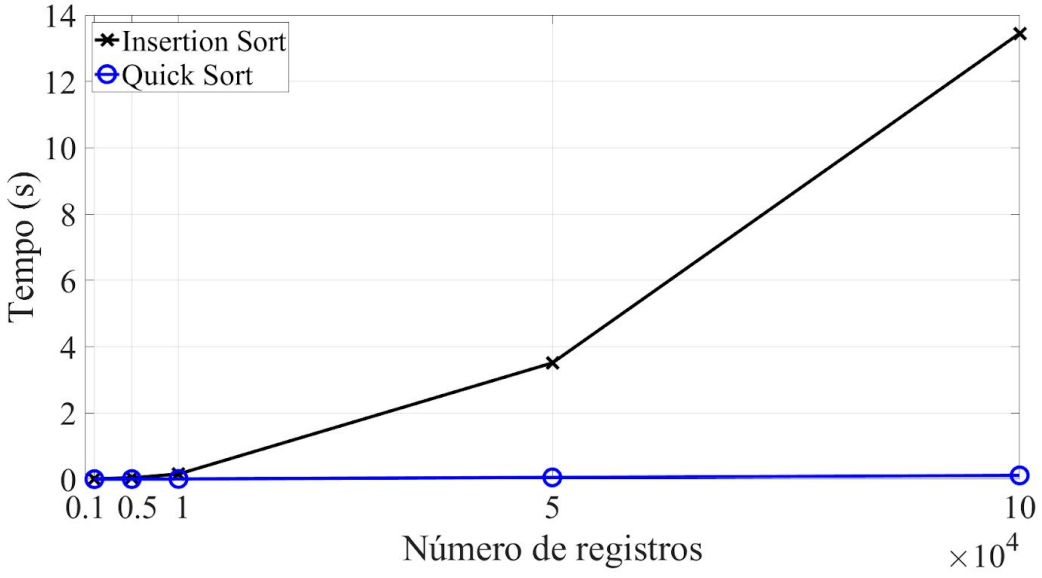
```c
void sort(Book **to_sort, int low, int high, int copy_count, int compare_count)
{
    if (low < high)
    {
        int pi = partition(to_sort, low, high, copy_count, compare_count);

        sort(to_sort, low, pi - 1, copy_count, compare_count);
        sort(to_sort, pi + 1, high, copy_count, compare_count);
    }
}
```

Método principal que é chamada recursivamente, chamadas primeiramente para o subarray a esquerda, e posteriormente a direta da posição do pivot ordenado previamente pelo método de partição.

Resultados

Número de registros de entrada	Estrutura de dados	Tempo de execução em segundos (Média de 5 amostras)	Número de comparações (Média de 5 amostras)	Número de cópias (Média de 5 amostras)
1000	InsertionSort	0,01077		
	QuickSort	0,00383		
5000	InsertionSort	0,04659		
	QuickSort	0,00619		
10000	InsertionSort	0,15770		
	QuickSort	0,00970		
50000	InsertionSort	3,51006		
	QuickSort	0,05683		
100000	InsertionSort	13,45133		
	QuickSort	0,11625		





Como pode ser observado na tabela acima, era esperado que o algoritmo do quick sort superasse o insertion sort para todos os tamanhos de entrada, uma vez que, para se mostrar o contrário, a entrada teria que ser bem menor que 1000 registros, dezenas a uma centena por exemplo. (foi feito um teste com uma entrada de 10 registros, o insertion sort levou a vantagem)

Um outro ponto a ser considerado é que quanto maior é a entrada, mais discrepante fica a diferença de tempo entre os dois algoritmos, revelando a enorme ineficiência do insertion para grandes quantidades de dados. Este ponto pode ser exemplificado observando o gráfico para a entrada de 1000, na qual os tempos não chegam a 1 segundo e são muito próximos, já para a entrada de 100000 tem uma diferença de 0 para 13 segundos.

Árvore Vermelho e Preta

	Caso médio	Pior Caso
Busca	O(logn)	O(logn)
Inserção	O(logn)	O(logn)
Remoção	O(logn)	O(logn)

São árvores binárias de busca auto-balanceadas, possui sua implementação relativamente complexa em relação a outras estruturas balanceadas como árvores B, AVL. Apesar disso sua ela possui um ótimo pior caso na prática para suas operações, inserção, deleção e busca, como é mostrado no caso acima. Sua complexidade deve-se principalmente a ideia principal do algoritmo, para manter-se “balanceada”, a cada operação feita é avaliada um conjunto 5 propriedades:

1. Cada nó é vermelho ou preto
2. A raiz é preta
3. Cada folha é preta
4. Se um nó é vermelho, seus filhos são pretos
5. Todo caminho até as folhas possui o mesmo número de nós pretos (altura negra)

Logo para cada inserção o método verify_properies abaixo é chamado para checar se alguma propriedade de 1 a 5 respectivamente foi violada:

```
void RBTree::verify_properties(rbtree t)
{
    verify_property_1 (t->root);
    verify_property_2 (t->root);
    verify_property_4 (t->root);
    verify_property_5 (t->root);
}
```

Segue os métodos abaixo de 1 ao 5:

```
/*
 * Verifying Property 1
 */
void RBTree::verify_property_1(node n)
{
    assert (node_color(n) == RED || node_color(n) == BLACK);
    if (n == NULL)
        return;
    verify_property_1(n->left);
    verify_property_1(n->right);
}
/*
 * Verifying Property 2
 */
void RBTree::verify_property_2(node root)
{
    assert (node_color(root) == BLACK);
}

/*
 * Verifying Property 4
 */
void RBTree::verify_property_4(node n)
{
    if (node_color(n) == RED)
    {
        assert (node_color(n->left) == BLACK);
        assert (node_color(n->right) == BLACK);
        assert (node_color(n->parent) == BLACK);
    }
    if (n == NULL)
        return;
    verify_property_4(n->left);
    verify_property_4(n->right);
}
/*
 * Verifying Property 5
 */
void RBTree::verify_property_5(node root)
{
    int black_count_path = -1;
    verify_property_5_helper(root, 0, &black_count_path);
}

void RBTree::verify_property_5_helper(node n, int black_count, int* path_black_count)
{
    if (node_color(n) == BLACK)
    {
        black_count++;
    }
    if (n == NULL)
    {
        if (*path_black_count == -1)
        {
            *path_black_count = black_count;
        }
        else
        {
            assert (black_count == *path_black_count);
        }
        return;
    }
    verify_property_5_helper(n->left,  black_count, path_black_count);
    verify_property_5_helper(n->right, black_count, path_black_count);
}
```

No processo de inserção/remoção, existem 5 e 6 casos distintos de balanceamento/recoloração/rotações (simples a esquerda e à direita), segue abaixo os métodos da classe RBTree:

```
class RBTree
{
    public:
        typedef int (*compare_func)(void* left, void* right);
        rbtree rbtree_create();
        void* rbtree_lookup(rbtree t, void* , compare_func compare);
        void rbtree_insert(rbtree t, void* , void* , compare_func compare);
        void rbtree_delete(rbtree t, void* , compare_func compare);
        node grandparent(node n);
        node sibling(node n);
        node uncle(node n);
        void verify_properties(rbtree t);
        void verify_property_1(node root);
        void verify_property_2(node root);
        color node_color(node n);
        void verify_property_4(node root);
        void verify_property_5(node root);
        void verify_property_5_helper(node n, int , int*);
        node new_node(void* key, void* , color , node , node);
        node lookup_node(rbtree t, void* , compare_func compare);
        void rotate_left(rbtree t, node n);
        void rotate_right(rbtree t, node n);
        void replace_node(rbtree t, node oldn, node newn);
        void insert_case1(rbtree t, node n);
        void insert_case2(rbtree t, node n);
        void insert_case3(rbtree t, node n);
        void insert_case4(rbtree t, node n);
        void insert_case5(rbtree t, node n);
        node maximum_node(node root);
        void delete_case1(rbtree t, node n);
        void delete_case2(rbtree t, node n);
        void delete_case3(rbtree t, node n);
        void delete_case4(rbtree t, node n);
        void delete_case5(rbtree t, node n);
        void delete_case6(rbtree t, node n);
};
```

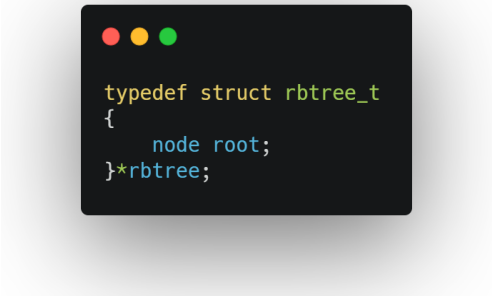
Nó

O nó se distingue basicamente das outras árvores binárias de busca pelo campo de cor. Ele possui as seguintes propriedades:

- **Color:** um enum com as cores Red e Black

- **Key:** chave, ID do livro
- **Value:** valor, ponteiro para o livro
- **Left, right, parent:** ponteiro para o nó filho a esquerda, direita e o pai respectivamente

```c
#include "color.h"

typedef struct rbtree_node
{
    color color;
    void *key;
    void *value;
    rbtree_node *left, *right, *parent;
}*node;
```



```c
typedef struct rbtree_t
{
    node root;
}*rbtree;
```

Autores mais frequentes

Abaixo é descrita a estratégia de implementação para encontrar os autores mais recentes.

Código desenvolvido

O código desenvolvido está na branch *"hash_feature"* e não foi integrado a branch principal para não causar conflitos com os demais códigos. Para executá-lo, basta inserir o arquivo de entrada "entrada.txt", executar o comando make e em seguida o comando ./main.

Entrada

A entrada consiste em um arquivo TXT contendo o número de autores mais frequentes desejados da base de livros. A quantidade especificada determina quantos serão impressos no arquivo de saída.

Estruturas

Para encontrar os autores mais frequentes, foi preciso ler o conjunto de livros disponíveis e mapear seus autores para um vetor de propriedades. Além disso, o programa lê o conjunto de autores e os mapeia para um vetor de propriedades. Os autores que não tem identificador foram considerados com id -1 e nome "UNKNOWN".

Em seguida, cria-se a hash table, uma estrutura de hash encadeada, com seus nós sendo alocados a partir da capacidade da hash. A estrutura dos nós armazena os valores de autor, uma chave que será usada nos cálculos de hashing, um contador de acertos que será usado posteriormente para frequência e um ponteiro para o próximo.

A lista de autores é então mapeada para a hash. Essa será usada para buscar todos os autores, e para cada nó é determinado o valor de frequência do valor. A partir da tabela hash com a frequência de autores, criou-se uma lista de frequência de autores, a qual ordenada com o Quicksort, entrega os autores mais frequentes no conjunto de livros.

Hashing

A Hashing proposta foi criada para esse trabalho, exclusivamente. A intenção era gerar uma quantidade pequena de esforço computacional e conseguir uma hash que causasse poucas colisões, mas respeitando as dimensões da tabela hash. Porém como utilizou-se uma estrutura hash com encadeamento linear, as colisões foram tratadas entre as posições da tabela hash independente do tamanho da mesma.

Segue o algoritmo de cálculo abaixo:

```
const string EMPTY_STR = "";

int HashTable::hash_int(int number, int capacity)
{
    int hash = 0;
    if (number < 0)
        number = -number;

    hash = (PRIME * number) % capacity;
    return hash;
}
```

Saída

A saída consiste em um arquivo de texto com a frequência de autores descrita como quantidade e nome.