Relatorio de Laboratório de Introdução à Ciência da Computação II – Avaliativo 7

Leonardo Kenzo Tanaka e Pedro Teidi de Sá Yamacita

18 de novembro de 2025

1 Análise da Busca Binária

```c
void QuickSort(int *datas_inteiras, int inf, int sup) {
    // codigo de particionamento e recursao omitido para brevidade) ...
    // Divide o vetor e ordena subvetores recursivamente
    if (j > inf) QuickSort(datas_inteiras, inf, j);
    if (i < sup) QuickSort(datas_inteiras, i, sup);
}

int busca_binaria(int *datas_inteiras, int tamanho, int alvo) {
    int esquerda = 0;
    int direita = tamanho - 1;
    while (esquerda <= direita) {
        int meio = esquerda + (direita - esquerda) / 2;
        if (datas_inteiras[meio] == alvo) return meio;
        if (datas_inteiras[meio] < alvo) esquerda = meio + 1;
        else direita = meio - 1;
    }
    return 0; // Nao encontrou
}
```

O código acima apresenta a implementação da Busca Binária combinada com o algoritmo de ordenação Quick Sort. A busca binária exige, como pré-condição, que o conjunto de dados esteja ordenado. Portanto, o código primeiramente converte as datas para inteiros e utiliza o Quick Sort para organizá-las.

A lógica da busca consiste em "dividir para conquistar": o algoritmo verifica o elemento central do vetor. Se o valor buscado for menor que o central, a busca continua na metade esquerda; se for maior, na metade direita. Esse processo reduz o espaço de busca pela metade a cada iteração.

Complexidade:A ordenação prévia com Quick Sort tem complexidade média de $O(N \log N)$. A busca binária em si tem complexidade $O(\log N)$. Para Q consultas, a complexidade total do algoritmo é $O(N \log N + Q \log N)$. É altamente eficiente para grandes volumes de dados, desde que o custo da ordenação inicial seja amortizado pelas múltiplas buscas.

2 Análise da Busca com Hashing

```c
int hash(int chave, int tamanho_table) {
    unsigned int posicao = chave % tamanho_table;
    return posicao;
}

// Trecho da funcao main (Caso 2)
NO **table = calloc(tamanho_table, sizeof(NO *));
for (int i = 0; i < N; i++) {
    // ... Insercao na tabela ...
    int index = hash(chave, tamanho_table);
    novo_no->proximo = table[index];
    table[index] = novo_no;
}

```

```c
for (int i = 0; i < Q; i++) {
    // ... Busca na tabela ...
    int index = hash(alvo, tamanho_table);
    NO *atual = table[index];
    while (atual != NULL) {
        if (atual->data == alvo) { encontrado = 1; break; }
        atual = atual->proximo;
    }
}
```

A implementação acima utiliza uma Tabela Hash com tratamento de colisões por encadeamento externo (listas ligadas). A função de hashing utiliza o método da divisão (chave % tamanho), mapeando as datas (convertidas para inteiros) diretamente para índices de um vetor.

Ao contrário da busca binária, este método não requer ordenação. O algoritmo constrói a tabela inserindo todos os N elementos. Para buscar, calcula-se o índice da chave e percorre-se a lista encadeada naquela posição apenas se houver colisões.

Complexidade: A construção da tabela leva tempo $O(N)$. A busca média é $O(1)$ (constante), assumindo uma distribuição uniforme das chaves. No pior caso (muitas colisões), a busca degrada para $O(N)$, mas isso é raro com uma boa função hash. A complexidade total para Q buscas é $O(N + Q)$, tornando este o método teoricamente mais rápido para grandes volumes de consultas.

3 Análise da Busca Sequencial

```c
    // Trecho da funcao main (Caso 3)
for (int i = 0; i < Q; i++) {
    int encontrado = 0;
    for (int j = 0; j < N; j++) {
        if (strcmp(datas[j], datas_alvo[i]) == 0) {
            encontrado = 1;
            break;
        }
    }
    // ... prints ...
}
```

O trecho acima descreve a Busca Sequencial (ou linear). É a abordagem mais simples (força bruta): para cada uma das Q datas buscadas, o algoritmo percorre o vetor de N datas do início ao fim, comparando as strings uma a uma usando a função strcmp.

Não há pré-processamento ou ordenação dos dados. Contudo, essa simplicidade custa caro em desempenho. Como as comparações são feitas com strings (char*) em vez de inteiros, há um custo adicional constante para cada comparação, além do custo algorítmico.

Complexidade: Para cada consulta, no pior caso (elemento não encontrado ou no final), o algoritmo faz N comparações. Para Q consultas, a complexidade total é $O(Q \times N)$. Se N e Q forem grandes (ex: 10^5), o número de operações torna-se proibitivo (10^{10}), tornando este método inviável para grandes bases de dados.

4 Comparação de Desempenho e Conclusão

Algoritmo	**Pré-processamento**	**Busca (1 item)**	**Total (Q buscas)**	**Uso de Memória**
Sequencial	Nenhum	$O(N)$	$O(N \times Q)$	Baixo ($O(1)$ extra)
Binária	Ordenação: $O(N \log N)$	$O(\log N)$	$O((N + Q) \log N)$	Baixo ($O(\log N)$)
Hashing	Construção: $O(N)$	$O(1)$ (médio)	$O(N + Q)$	Alto ($O(N)$ extra)

Tabela 1: Comparação de Complexidade e Desempenho dos Algoritmos de Busca

Caso	Busca Binária	Busca com Hashing	Busca Sequencial
1 a 3	0.0018 s	0.0018 s	0.0073 s
4 a 6	0.1493 s	0.1678 s	3.0008+ s
7 a 9	0.1125 s	0.1702 s	3.0007+ s

Tabela 2: Análise empírica de desempennho (Tempo de execução no Runcodes)

4.1 Discussão dos Resultados

1. **Mais Rápido:** O método de Busca com Hashing tende a ser o mais rápido globalmente, pois sua complexidade total é linear $O(N + Q)$. Ele elimina o fator logarítmico presente na busca binária e o fator linear multiplicativo da busca sequencial. No entanto, ele consome mais memória devido à estrutura da tabela e dos nós da lista encadeada. No entanto, nos casos disponíveis no Runcodes, seu tempo de execução é sempre maior ou igual ao da Busca Binária

2. **Intermediário:** A Busca Binária é extremamente eficiente e muito próxima do Hashing em termos de tempo para valores razoáveis de N. A desvantagem é a necessidade de ordenação prévia ($QuickSort$). Se Q (número de buscas) for muito pequeno (ex: buscar 1 data em 1 milhão), o custo de ordenar tudo ($N \log N$) pode fazer com que ela seja mais lenta que a sequencial. Mas para Q grande, ela é ordens de grandeza superior à sequencial.

3. **Mais Lento:** A Busca Sequencial é, inequivocamente, a mais lenta para grandes entradas, não conseguindo executar os casos 6 e 9 do Runcodes dentro do tempo de execução limite disponível. Ela realiza um produto cartesiano das entradas ($N \times Q$). Enquanto a busca binária divide o problema e o Hashing salta para a resposta, a busca sequencial verifica item por item. Além disso, na nossa implementação, a busca sequencial compara strings (strcmp), o que é computacionalmente mais custoso do que comparar inteiros, como feito nas implementações Binária e Hashing.

5 Conclusão

Para aplicações reais com grandes volumes de dados, a Busca com Hashing é a escolha ideal se houver memória disponível, devido ao seu tempo de acesso constante $O(1)$. Se a memória for restrita, a Busca Binária é a melhor alternativa, equilibrando velocidade e uso de recursos. A Busca Sequencial deve ser evitada, exceto para listas triviais (ex: $N < 100$).