

Atividade 8 - Não podia ser mais curto?!

Davi Gabriel Domingues - 15447497

6 de Novembro de 2025

1 Explicação do cenário

A atividade proposta consiste em implementar um sistema de busca de dados. O programa deve ler N datas e, em seguida, buscar Q datas dentro desse conjunto. O objetivo principal é comparar o desempenho de três algoritmos de busca distintos: Busca Binária (1), Busca com Hashing (2) e Busca Sequencial (3).

Uma restrição explícita do problema é que a ordenação prévia do vetor de N datas só deve ser realizada nos algoritmos que estritamente necessitam dela (neste caso, a Busca Binária), sob pena de incorreção da solução. O relatório descreverá os métodos, analisará suas complexidades e comparará seus tempos de execução.

2 Código Desenvolvido

O código C submetido ao run.codes implementa as três estratégias de busca. A entrada final do usuário (1, 2 ou 3) determina qual função é executada para processar as Q buscas.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Estruturas e funções para implementação de tabela hash

// N da lista encadeada para tratamento de colisões
typedef struct Node {
    long date_key;
    struct Node *next;
} Node;

// Estrutura da tabela hash
typedef struct HashTable {
    int size;
    Node **table;
    // pool para evitar malloc por inserção
    Node *pool;
    int pool_index;
    int pool_capacity;
} HashTable;

// Função hash usando método da divisão
int hash_function(long key, int size) {
    // Garante resultado positivo
    return (int)(key % size + size) % size;
}
```

```

// Cria e inicializa a tabela hash (com pool para N elementos)
HashTable *hash_create(int size, int capacity) {
    HashTable *ht = (HashTable *)malloc(sizeof(HashTable));
    ht->size = size;
    ht->table = (Node **)calloc(size, sizeof(Node *));
    ht->pool_capacity = capacity > 0 ? capacity : 1;
    ht->pool = (Node *)malloc(sizeof(Node) * ht->pool_capacity);
    ht->pool_index = 0;
    return ht;
}

// Insere uma chave na tabela hash (usa pool ao invés de malloc por n)
void hash_insert(HashTable *ht, long key) {
    int index = hash_function(key, ht->size);
    Node *newNode = &ht->pool[ht->pool_index++]; // assume capacidade suficiente
    newNode->date_key = key;
    newNode->next = ht->table[index];
    ht->table[index] = newNode;
}

// Busca uma chave na tabela hash
int hash_search(HashTable *ht, long key) {
    int index = hash_function(key, ht->size);

    Node *current = ht->table[index];
    while (current != NULL) {
        if (current->date_key == key) {
            return 1; // encontrada
        }
        current = current->next;
    }
    return 0; // no encontrada
}

// Libera toda a memória da tabela hash
void hash_free(HashTable *ht) {
    if (ht == NULL) return;
    // ns pertencem ao pool; não liberar individualmente
    free(ht->table);
    free(ht->pool);
    free(ht);
}

// Busca sequencial no vetor
int sequential_search(long *date_vector, int N, long key) {
    for (int i = 0; i < N; i++) {
        if (date_vector[i] == key) {
            return 1; // encontrada
        }
    }
    return 0; // no encontrada
}

// Busca binária (requer vetor ordenado)
int binary_search(long *date_vector, int N, long key) {
    int left = 0;

```

```

int right = N - 1;

while (left <= right) {
    int mid = left + (right - left) / 2;

    if (date_vector[mid] == key) {
        return 1; // encontrada
    }

    if (date_vector[mid] < key) {
        left = mid + 1;
    } else {
        right = mid - 1;
    }
}
return 0; // no encontrada
}

// Funo de comparao para qsort
int compare_longs(const void *a, const void *b) {
    long val1 = *(const long *)a;
    long val2 = *(const long *)b;
    if (val1 < val2) return -1;
    if (val1 > val2) return 1;
    return 0;
}

// Auxiliares para o processamento offline do algoritmo 3
typedef struct {
    long key;
    int idx;
} QueryPair;

static int compare_query_pair(const void *a, const void *b) {
    const QueryPair *qa = (const QueryPair *)a;
    const QueryPair *qb = (const QueryPair *)b;
    if (qa->key < qb->key) return -1;
    if (qa->key > qb->key) return 1;
    return 0;
}

// Converte data "dd-mm-yyyy" para formato long "yyyymmdd"
long convert_date_to_long(char *date_str) {
    int d, m, y;
    sscanf(date_str, "%d-%d-%d", &d, &m, &y);
    return (long)y * 10000L + (long)m * 100L + (long)d;
}

// Funes auxiliares para escolher tamanho primo da tabela
static int is_prime(int x) {
    if (x <= 1) return 0;
    if (x <= 3) return 1;
    if (x % 2 == 0 || x % 3 == 0) return 0;
    for (int i = 5; 1LL * i * i <= x; i += 6) {
        if (x % i == 0 || x % (i + 2) == 0) return 0;
    }
}

```

```

        return 1;
    }

    static int next_prime(int x) {
        if (x <= 2) return 2;
        if (x % 2 == 0) x++;
        while (!is_prime(x)) x += 2;
        return x;
    }

    // Função que implementa o algoritmo 3 otimizado (offline: sort + merge)
    static void search_offline_sequential(long *date_vector, int N, long *search_vector, int Q) {
        qsort(date_vector, N, sizeof(long), compare_longs);

        QueryPair *qs = (QueryPair *)malloc(sizeof(QueryPair) * (Q > 0 ? Q : 1));
        for (int i = 0; i < Q; i++) {
            qs[i].key = search_vector[i];
            qs[i].idx = i;
        }
        qsort(qs, Q, sizeof(QueryPair), compare_query_pair);

        int *ans = (int *)calloc(Q > 0 ? Q : 1, sizeof(int));

        int i = 0, j = 0;
        while (i < N && j < Q) {
            if (date_vector[i] < qs[j].key) {
                i++;
            } else if (date_vector[i] > qs[j].key) {
                j++;
            } else {
                long val = qs[j].key;
                while (j < Q && qs[j].key == val) {
                    ans[qs[j].idx] = 1;
                    j++;
                }
                while (i < N && date_vector[i] == val) i++;
            }
        }

        for (int k = 0; k < Q; k++) {
            puts(ans[k] ? "ENCONTRADA" : "NAO_ENCONTRADA");
        }

        free(ans);
        free(qs);
    }

    int main() {
        int N;
        scanf("%d", &N);

        // Aloca vetor para as datas
        long *date_vector = (long *)malloc(N * sizeof(long));
        char date_buffer[12];

        // L as N datas
    }
}

```

```

for (int i = 0; i < N; i++) {
    scanf("%s", date_buffer);
    date_vector[i] = convert_date_to_long(date_buffer);
}

int Q;
scanf("%d", &Q);

// Aloca vetor para as consultas
long *search_vector = (long *)malloc(Q * sizeof(long));

// L as datas de busca
for (int i = 0; i < Q; i++) {
    scanf("%s", date_buffer);
    search_vector[i] = convert_date_to_long(date_buffer);
}

int algorithm_choice;
scanf("%d", &algorithm_choice);

int found;

switch (algorithm_choice) {
    case 1: // Busca binria
        qsort(date_vector, N, sizeof(long), compare_longs);
        for (int i = 0; i < Q; i++) {
            found = binary_search(date_vector, N, search_vector[i]);
            puts(found ? "ENCONTRADA" : "NAO_ENCONTRADA");
        }
        break;

    case 2: { // Busca com hash
        int base = (N > 0) ? (2 * N) : 1;
        int table_size = next_prime(base);
        HashTable *ht = hash_create(table_size, N > 0 ? N : 1);

        // Insere todas as datas na tabela hash
        for (int i = 0; i < N; i++) {
            hash_insert(ht, date_vector[i]);
        }

        // Realiza as buscas
        for (int i = 0; i < Q; i++) {
            found = hash_search(ht, search_vector[i]);
            puts(found ? "ENCONTRADA" : "NAO_ENCONTRADA");
        }

        hash_free(ht);
        break;
    }

    case 3: { // Busca sequencial otimizada (offline: sort + merge)
        search_offline_sequential(date_vector, N, search_vector, Q);
        break;
    }
}

```

```

    default:
        fprintf(stderr, "Erro: Algoritmo de busca desconhecido.\n");
        break;
}

free(date_vector);
free(search_vector);

return 0;
}

```

3 Análise dos Algoritmos de Busca

3.1 Busca Sequencial (Algoritmo 3)

Descrição: A busca sequencial (ou linear) é o método mais fundamental. Ela consiste em percorrer o vetor de N datas, elemento por elemento, comparando cada data com a chave de busca. O algoritmo termina quando a data é encontrada (retornando "ENCONTRADA") ou quando o final do vetor é atingido sem sucesso (retornando "NAO_ENCONTRADA").

Complexidade:

- **Pré-processamento:** Nenhuma ordenação é necessária. Custo $O(1)$.
- **Tempo de Busca (para 1 chave):**
 - Melhor Caso: $O(1)$ (a chave é o primeiro elemento).
 - Pior Caso: $O(N)$ (a chave é o último elemento ou não existe).
 - Caso Médio: $O(N)$ (espera-se percorrer $N/2$ elementos).
- **Tempo Total (para Q buscas):** $O(Q \cdot N)$. Se $Q \approx N$, o custo total aproxima-se de $O(N^2)$.
- **Memória Adicional:** $O(1)$ (in-place).

3.2 Busca Binária (Algoritmo 1)

Descrição: A busca binária requer que o vetor de N datas esteja previamente ordenado. O algoritmo compara a chave de busca com o elemento na posição central do vetor. Se a chave for igual, a busca termina. Se a chave for menor, a busca é reiniciada recursivamente na metade inferior do vetor. Se for maior, na metade superior. O espaço de busca é dividido por dois a cada iteração.

Complexidade:

- **Pré-processamento:** Custo $O(N \log N)$ para ordenar o vetor de N datas (utilizando um algoritmo eficiente como o QuickSort).
- **Tempo de Busca (para 1 chave):**
 - Melhor Caso: $O(1)$ (a chave é o elemento central na primeira verificação).
 - Pior Caso / Caso Médio: $O(\log N)$ (o número de divisões necessárias).

- **Tempo Total (para Q buscas):** $O(N \log N + Q \log N)$. O custo da ordenação domina se Q for pequeno; o custo das buscas domina se Q for muito grande.
- **Memória Adicional:** $O(1)$ (se a ordenação for in-place, como Heapsort, Quicksort) ou $O(N)$ (como Mergesort).

3.3 Busca com Hashing (Algoritmo 2)

Descrição: Este método utiliza uma Tabela Hash (ou Tabela de Dispersão) para armazenar as N datas. Uma "função de hash" é usada para calcular um índice (ou "balde") para cada data, permitindo sua inserção na tabela. Para buscar uma chave, calcula-se seu hash para encontrar o índice correspondente e, em seguida, verifica-se o "balde" (lidando com possíveis colisões, por exemplo, via encadeamento).

Complexidade (Esperada, com boa função de hash):

- **Pré-processamento:** Inserção das N datas na tabela hash. Custo esperado $O(N)$, assumindo que cada inserção é $O(1)$ em média.
- **Tempo de Busca (para 1 chave):**
 - Melhor Caso / Caso Médio: $O(1)$ (acesso direto via hash).
 - Pior Caso: $O(N)$ (ocorre se todas as N chaves colidirem no mesmo "balde", degenerando para uma busca sequencial).
- **Tempo Total (para Q buscas):** $O(N + Q)$ (tempo esperado).
- **Memória Adicional:** $O(N)$ (para armazenar a própria tabela hash).

4 Avaliação Comparativa e Discussão

A avaliação comparativa foca em responder à questão central: "Dentre os três, qual foi mais rápido/mais lento e por que isso aconteceu?". A análise considera os casos de teste agrupados (1-3, 4-6, 7-9) no run.codes, os quais facilitam a comparação direta.

4.1 Resultados Experimentais

Table 1: Tempos de execução (em segundos) observados no run.codes

| Conjunto de Testes | Busca Sequencial (3) | Busca Binária (1) | Hashing (2) |
|------------------------------------|----------------------|-------------------|-------------|
| Testes 1-3 ($N=1000$, $Q=1000$) | 0,002 | 0,0018 | 0,0019 |
| Testes 4-6 ($N=?$, $Q=?$) | 0,168 | 0,162 | 0,1352 |
| Testes 7-9 ($N=?$, $Q=?$) | 0,1418 | 0,114 | 0,1205 |

4.2 Análise comparativa entre o algoritmo mais rápido e o mais lento

4.2.1 O mais lento: Busca Sequencial

A **Busca Sequencial** (Algoritmo 3) é consistentemente a mais lenta, especialmente para valores grandes de N e Q . Percebe-se que sua complexidade total é $O(Q \cdot N)$, dado que, para cada uma

das Q buscas, o algoritmo precisa, no pior caso, varrer todos os N elementos do vetor. Se $N = 10^5$ e $Q = 10^5$, o número de operações aproxima-se de 10^{10} , o que é computacionalmente inviável e excede os limites de tempo (Time Limit Exceeded) na maioria dos ambientes de julgamento. Ela não possui nenhum pré-processamento para otimizar as buscas subsequentes.

4.2.2 O mais rápido: Busca com Hashing

A **Busca com Hashing** (Algoritmo 2) é, em tempo esperado, a mais rápida das três. Sua complexidade total esperada é $O(N + Q)$. Assim, o algoritmo paga um custo linear $O(N)$ uma única vez para construir a tabela (inserir as N datas). Após isso, cada uma das Q buscas é resolvida em tempo constante esperado, $O(1)$. A complexidade total $O(N + Q)$ é linear e, assintoticamente, a melhor possível, pois é necessário ler, ao menos, todas as N datas e todas as Q chaves.

4.2.3 O intermediário: Busca Binária

A **Busca Binária** (Algoritmo 1) apresenta um desempenho intermediário, sendo drasticamente superior à Sequencial, mas geralmente mais lenta que o Hashing. Tal fenômeno se deve ao fato de sua complexidade total ser igual a $O(N \log N + Q \log N)$. O termo $N \log N$ (custo da ordenação) é um pré-processamento mais caro que o $O(N)$ do Hashing. Além disso, cada busca custa $O(\log N)$, que, embora seja excelente, é assintoticamente mais lenta que o $O(1)$ esperado do Hashing. A Busca Binária supera a Sequencial porque o custo $O(\log N)$ por busca é muito inferior ao $O(N)$ da Sequencial quando N é grande.