

# Trabalho Final

## Grupo G

**Algoritmos de Ordenação:** Insertion Sort e Merge Sort  
**Estrutura de Dados:** Lista Linear Simplesmente Encadeada

**Marcos Paulo Caetano** **Lucas Almeida Oliveira Isaac<sup>2</sup>** **Breno Machado Barros<sup>3</sup>** **Lucca Magnino<sup>4</sup>**  
**Mendes Queiroz<sup>1</sup>**

1. marcospaulo2@discente.ufg.br

2. lucas\_isaac@discente.ufg.br

3. breno\_machado@discente.ufg.br

4. luccamagnino@discente.ufg.br

**Prof. Me. Raphael Guedes**

**2024**

**INF**

INSTITUTO DE  
INFORMÁTICA



# Sumário

## ► 1. Introdução

- O que são algoritmos de ordenação?
- Estruturas de Dados: Lista Linear Simplesmente Encadeada

## ► 2. Processo de Implementação

## ► 3. Resultados

- Gráfico de Desempenho
- Complexidade
- Estatísticas das Soluções
- Teste ao vivo

## ► 4. Lições aprendidas e Conclusão



# Introdução

# O que são algoritmos de ordenação?

- Algoritmos que organizam elementos de uma sequência, como números ou palavras em uma ordem específica (crescente, decrescente, etc.).
- Importante para aplicações em busca, otimização, e processamento de dados.

## Algoritmos Escolhidos:

- **Insertion Sort(Básico)**: É simples, eficiente para conjuntos pequenos ou quase ordenados.
- **Merge Sort(Avançado)**: Usa uma abordagem de "Dividir e Conquistar", eficiente para grandes conjuntos e essencial em bibliotecas modernas.

# Estruturas de Dados: Lista Linear Simplesmente Encadeada

- Uma lista onde cada elemento (nó) armazena um valor e a referência para o próximo.
- Permite inserções e remoções dinâmicas, ideal para operações que exigem flexibilidade estrutural.

## Objetivo do trabalho:

Implementar e comparar os algoritmos **Insertion Sort** e **Merge Sort** usando **vetores** e **listas linearmente encadeadas** e avaliar desempenho teórico e prático em diferentes cenários.



# Processo de Implementação

# Estrutura da Lista Linear Simplesmente Encadeada

A estrutura de dados que nós escolhemos foi a lista linear simplesmente encadeada, onde cada nó contém um valor e um ponteiro para o próximo.

```
struct elemento
{
    char nome[100];
    struct elemento *proximo;
};
typedef struct elemento Elemento;

Lista *cria_lista()
{
    Lista *lista = (Lista *)malloc(sizeof(Lista));
    if (lista != NULL)
    {
        *lista = NULL;
    }
    else
    {
        printf("Erro na alocao da lista!\n");
        exit(1);
    }

    return lista;
}
```

# Função de Inserção de Elementos na Lista

A função `insertion` insere um novo elemento no final da lista encadeada.

```
int insertion(Lista *lista, int valor)
{
    if (lista == NULL)
    {
        printf("Erro na alocação da lista!\n");
        exit(1);
    }
    Elemento *no = (Elemento *)malloc(sizeof(Elemento));
    if (no == NULL)
    {
        printf("Erro na alocação do no!\n");
        exit(1);
    }
    no->valor = valor;
    no->proximo = NULL;

    if (*lista == NULL)
    {
        *lista = no;
    }
    else
    {
        Elemento *aux = *lista;
        while (aux->proximo != NULL)
        {
            aux = aux->proximo;
        }
        aux->proximo = no;
    }
    return 1;
}
```



# Leitura e Processamento de Arquivo

A função `arquivo` abre um arquivo, lê seus dados e os insere na lista encadeada.

```
main_li.c

void arquivo(char *arquivo, Lista *lista)
{
    FILE *file = fopen(arquivo, "r");
    if (file == NULL)
    {
        printf("Erro ao abrir o arquivo %s\n", arquivo);
        exit(1);
    }

    char nome[100];
    while (fgets(nome, sizeof(nome), file))
    {
        nome[strcspn(nome, "\n")] = '\0';
        insertion(lista, nome);
    }

    fclose(file);
}
```

# Função para Imprimir a Lista

A função `imprime_lista` percorre a lista e imprime os nomes armazenados.



li.c

```
void imprime_lista(Lista *lista)
{
    if (lista == NULL)
    {
        printf("Erro na alocação da lista!\n");
        exit(1);
    }

    Elemento *no = *lista;
    while (no != NULL)
    {
        printf("%s\n", no->nome);
        no = no->proximo;
    }
}
```

# Função de Ordenação Crescente (Insertion Sort)

A função `ordenar_crescente` implementa a ordenação crescente utilizando o algoritmo **Insertion Sort** adaptado para lista encadeada

```
int ordenar_crescente(Lista *lista)
{
    if (lista == NULL || *lista == NULL)
    {
        printf("Erro na alocação da lista!\n");
        exit(1);
    }

    Lista ordenado = NULL;
    Elemento *atual = *lista;

    while (atual != NULL)
    {
        Elemento *proximo = atual->proximo;

        if (ordenado == NULL || strcmp(atual->nome, ordenado->nome) < 0)
        {
            atual->proximo = ordenado;
            ordenado = atual;
        }
        else
        {
            Elemento *aux = ordenado;
            while (aux->proximo != NULL && strcmp(aux->proximo->nome, atual->nome) < 0)
            {
                aux = aux->proximo;
            }
            atual->proximo = aux->proximo;
            aux->proximo = atual;
        }
        atual = proximo;
    }

    *lista = ordenado;
    return 1;
}
```

# Função de Ordenação Decrescente

A função `ordenar_decrescente` inverte a lista já ordenada de forma crescente para obter a ordem decrescente.

```
int ordenar_decrescente(Lista *lista)
{
    if (lista == NULL || *lista == NULL)
    {
        printf("Erro na alocação da lista!\n");
        exit(1);
    }

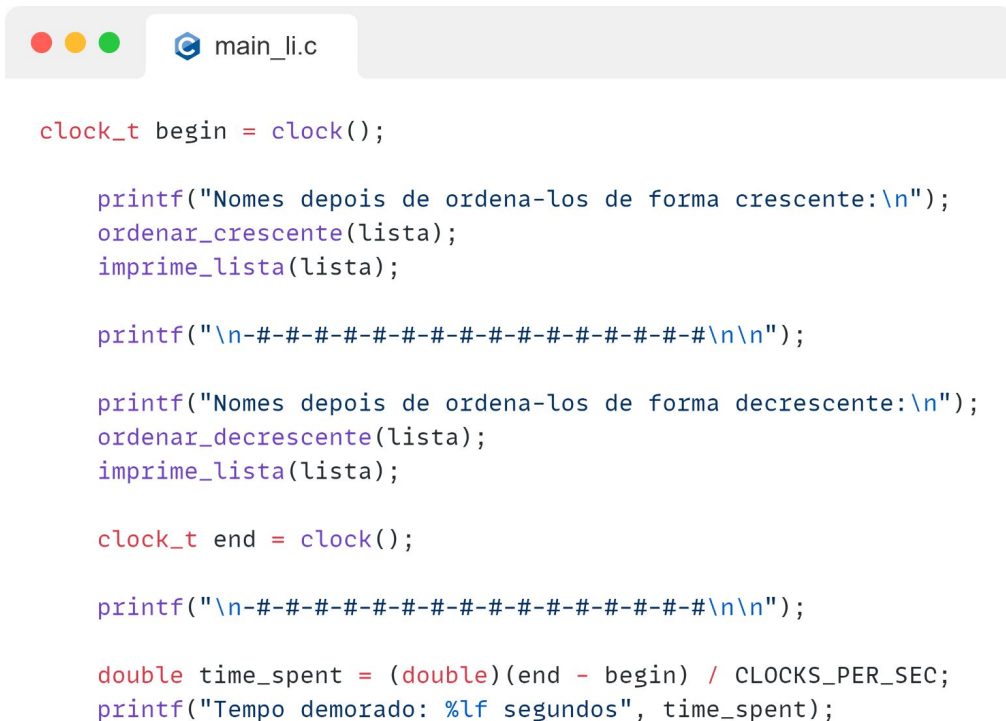
    if (ordenar_crescente(lista))
    {
        Elemento *anterior = NULL;
        Elemento *atual = *lista;
        Elemento *proximo;

        while (atual != NULL)
        {
            proximo = atual->proximo;
            atual->proximo = anterior;
            anterior = atual;
            atual = proximo;
        }

        *lista = anterior;
        return 1;
    }
    return 0;
}
```

# Medição de Tempo de Execução

O tempo de execução da ordenação é medido usando a função `clock()` para registrar o tempo antes e depois da execução da função de ordenação.



```
main_li.c

clock_t begin = clock();

printf("Nomes depois de ordena-los de forma crescente:\n");
ordenar_crescente(lista);
imprime_lista(lista);

printf("\n-#-#-#-#-#-#-#-#-#-#-#-#-#\n\n");

printf("Nomes depois de ordena-los de forma decrescente:\n");
ordenar_decrescente(lista);
imprime_lista(lista);

clock_t end = clock();

printf("\n-#-#-#-#-#-#-#-#-#-#-#-#-#\n\n");

double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
printf("Tempo demorado: %lf segundos", time_spent);
```

# Estrutura do Merge Sort para Listas Encadeadas

Merge Sort foi adaptado para listas encadeadas, dividindo a lista recursivamente e intercalando as sublistas ordenadas.



ListaDinEncadInt.c

```
Elem* merge_sort_rec_int(Elem* head, int ordem) {  
    if (head == NULL || head->prox == NULL)  
        return head;  
  
    Elem* meio = divide_lista_int(head);  
    Elem* esquerda = merge_sort_rec_int(head, ordem);  
    Elem* direita = merge_sort_rec_int(meio, ordem);  
  
    return intercala_listas_int(esquerda, direita, ordem);  
}
```

# Função Intercalação do Merge Sort

A função `intercala_listas` combina duas sublistas ordenadas em uma lista única ordenada.

```
ListaDinEncadInt.c

Elem* intercala_listas_int(Elem* lista1, Elem* lista2, int ordem) {
    if (lista1 == NULL) {
        return lista2;
    }
    if (lista2 == NULL) {
        return lista1;
    }

    Elem* result = NULL;

    if ((ordem == 1 && lista1->valor <= lista2->valor) ||
        (ordem == -1 && lista1->valor > lista2->valor))
    {
        result = lista1;
        troca_count++; // Incrementa o contador de trocas
        result->prox = intercala_listas_int(lista1->prox, lista2, ordem);
    } else {
        result = lista2;
        troca_count++; // Incrementa o contador de trocas
        result->prox = intercala_listas_int(lista1, lista2->prox, ordem);
    }

    return result;
}
```

# Função Merge Sort para Ordenação

A função `merge_sort` coordena a execução do Merge Sort sobre a lista encadeada.



ListaDinEncadInt.c

```
void merge_sort_int(Lista* li, int ordem) {  
    if (li == NULL || *li == NULL) return;  
    *li = merge_sort_rec_int(*li, ordem);  
}
```



# Resultados e Encerramento da Execução

Após a ordenação, o programa exibe a lista final e finaliza a execução de maneira limpa, garantindo que o fluxo do código seja concluído adequadamente.

```
main_li.c

int main()
{
    Lista *lista = cria_lista();

    printf("Abrindo o arquivo!\n\n");
    arquivo("nomes_aleatorios.txt", lista);

    printf("Nomes antes de ordena-los:\n");
    imprime_lista(lista);

    printf("\n-#-#-#-#-#-#-#-#-#-#\n\n");

    clock_t begin = clock();

    printf("Nomes depois de ordena-los de forma crescente:\n");
    ordenar_crescente(lista);
    imprime_lista(lista);

    printf("\n-#-#-#-#-#-#-#-#-#-#\n\n");

    printf("Nomes depois de ordena-los de forma decrescente:\n");
    ordenar_decrescente(lista);
    imprime_lista(lista);

    clock_t end = clock();

    printf("\n-#-#-#-#-#-#-#-#-#-#\n\n");

    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
    printf("Tempo demorado: %lf segundos", time_spent);

    libera_lista(lista);
    return 0;
}
```

# Liberação de Memória

Após o processamento, a memória alocada para a lista é liberada usando a função `libera_lista`.

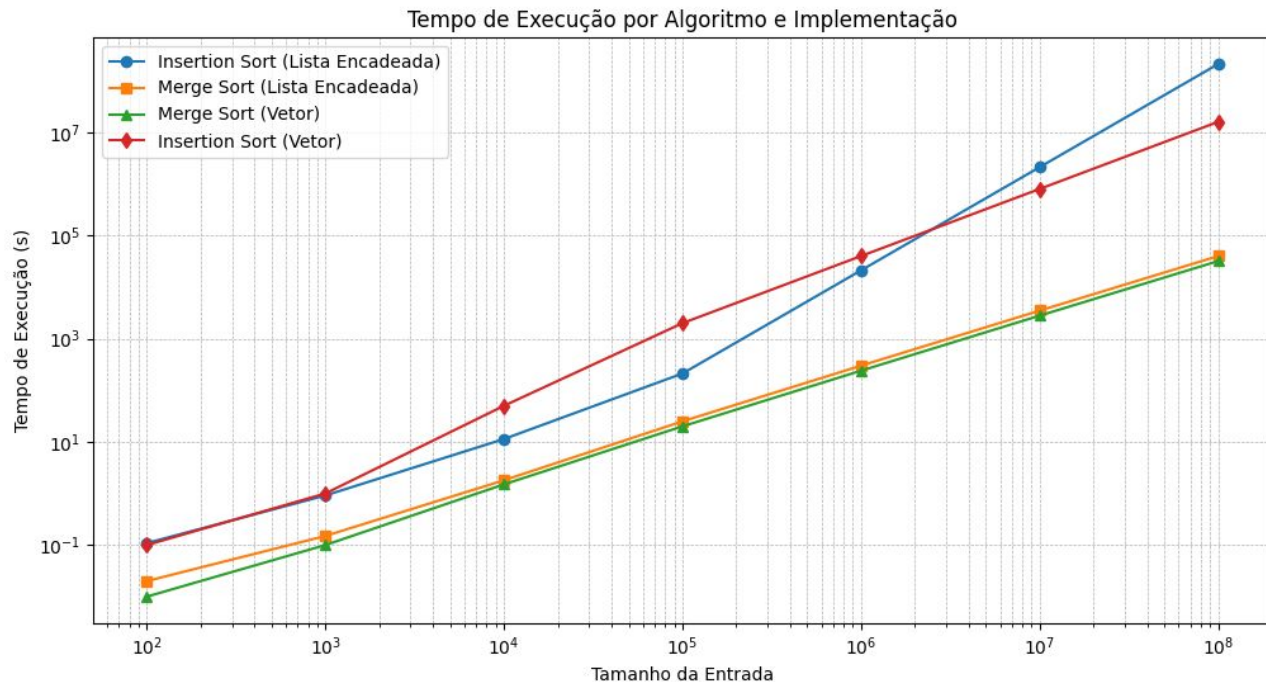
```
void libera_lista(Lista *lista)
{
    if (lista != NULL)
    {
        Elemento *no;
        while ((*lista) != NULL)
        {
            no = *lista;
            *lista = (*lista)->proximo;
            free(no);
        }
        free(lista);
    }
    else
    {
        printf("Erro na alocação da lista!\n");
        exit(1);
    }
}
```



# Resultados

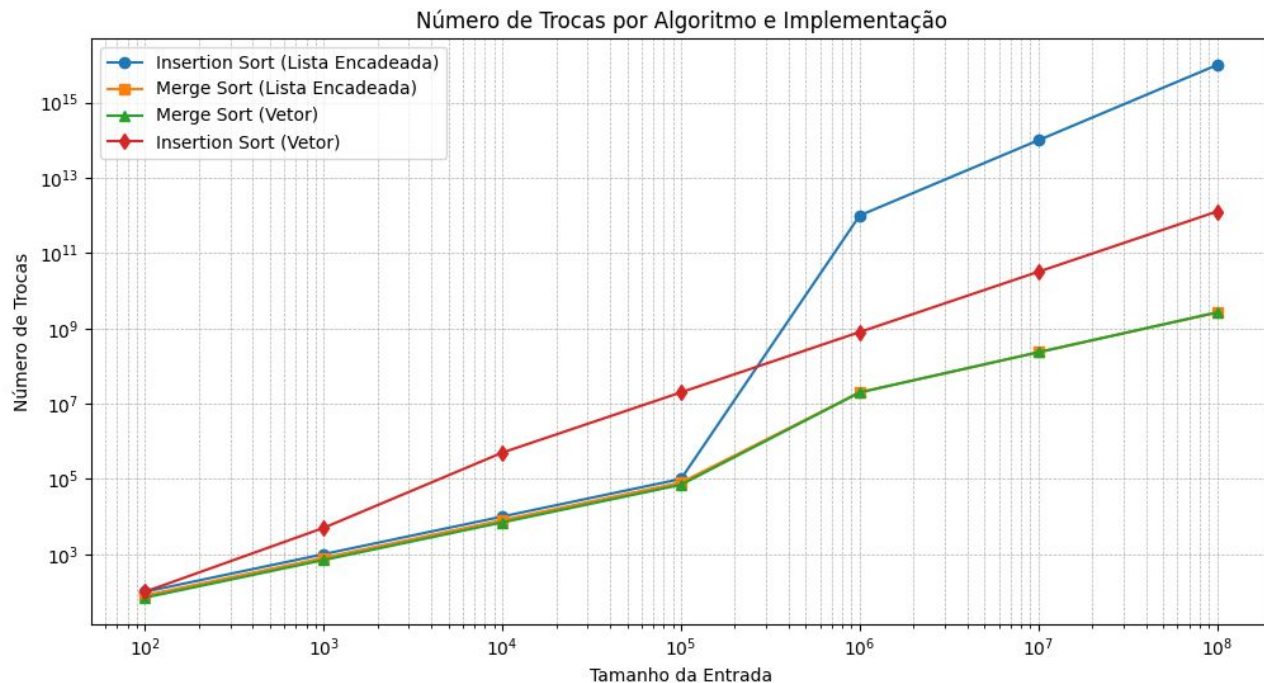
# Gráfico de Desempenho

## Desempenho na Prática:



# Gráfico de Desempenho

## Desempenho na Prática:



# Complexidade

## Análise Teórica:

### Insertion Sort:

- **Listas linearmente simplesmente encadeadas:**
  - Melhor caso:  $O(n)$  – ocorre quando os elementos já estão ordenados.
  - Pior caso:  $O(n^2)$  – ocorre quando os elementos estão em ordem inversa.
  - Cada inserção percorre a lista até encontrar a posição correta.

### Merge Sort:

- **Listas linearmente simplesmente encadeadas:**
  - Complexidade geral:  $O(n \log n)$  - devido à divisão recursiva e à intercalação das sublistas.
  - A lista é dividida em metades repetidamente  $(\log n)$  e cada intercalação percorre todos os elementos  $(n)$ .
- **Vetores:**
  - Complexidade geral:  $O(n \log n)$

# Estatísticas das Soluções

## Avaliação Empírica:

### Insertion Sort em listas encadeadas:

- **Tempo de execução:**
  - Crescente:  $T_c \approx O(n)$ .
  - Aleatório:  $T_a \approx O(n^2)$ .
  - Decrescente:  $T_d \approx O(n^2)$ .
- **Número de trocas:**
  - Crescente: Quase nenhuma troca.
  - Aleatório: Máximo de trocas esperado.
  - Decrescente: Todas as trocas necessárias.

### Merge Sort em listas encadeadas:

- **Tempo de execução:** Crescente, Aleatório e Decrescente:  $T \approx O(n \log n)$ .
- **Número de trocas:** Constante por nível de divisão e intercalação.

# Estatísticas das Soluções

## Comparação entre Insertion Sort e Merge Sort:

### Tempo de execução (estimado):

- **Insertion Sort:** Tempos significativamente maiores para entradas grandes, devido à complexidade  $O(n^2)$
- **Merge Sort:** Melhor desempenho para entradas grandes, devido à eficiência de  $O(n \log n)$

### Eficiência:

- **Merge Sort** é superior para listas maiores e entradas desordenadas.
- **Insertion Sort** pode ser mais eficiente em listas pequenas ou quase ordenadas.



# Teste ao vivo

Demonstração do código em tempo real:

6 5 3 1 8 7 2 4

6 5 3 1 8 7 2 4



# Lições aprendidas e Conclusão

# Lições Aprendidas e Conclusão

## Desafios Enfrentados:

- Adaptação dos algoritmos para a estrutura de Lista Linear Simplesmente Encadeada: Gerenciamento de referências ao navegar pelos nós da lista e lidar com a complexidade na hora realizar trocas.
- Aprendemos que a escolha da estrutura de dados impacta diretamente o desempenho e que cada algoritmo tem forças e limitações específicas

**Conclusão:** O nosso trabalho reforçou a importância de entender os fundamentos teóricos e práticos de algoritmos e estruturas de dados utilizando Insertion Sort e Merge Sort, que são complementares em suas aplicações, dependendo do contexto e da estrutura de dados utilizada.

# Obrigado!

