INSTITUTO DE INFORMÁTICA
Universidade Federal de Goiás

Trabalho final de AED1

André De Oliveira Machado Filho
João Victor Borges Tavares

INSTITUTO DE INFORMÁTICA
UFG

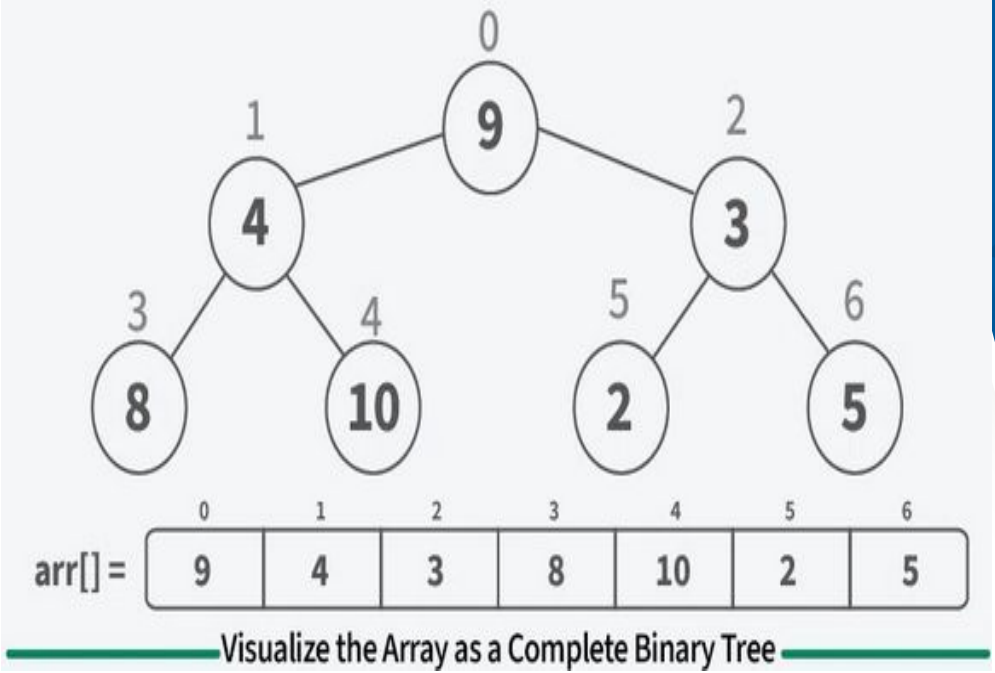
UFG
UNIVERSIDADE FEDERAL DE GOIÁS

- Heap Sort
- Selection sort

Layout: Simple right

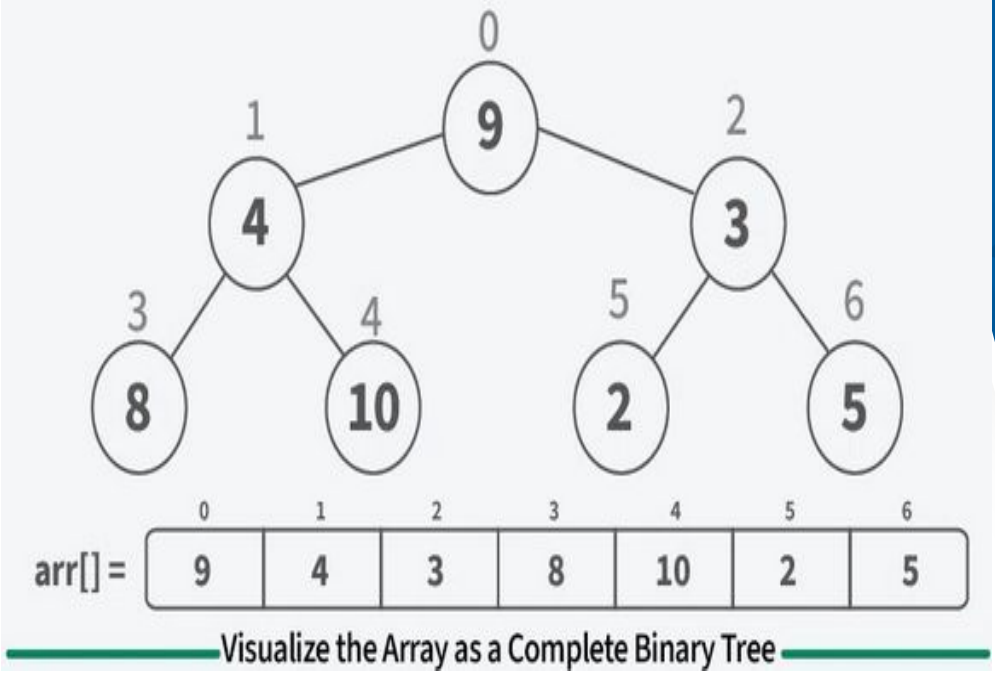
Heap sort

- Heap sort é uma técnica de ordenação que está no grupo de algoritmos baseado em comparação
 - Baseada na Estrutura de Dados do Heap Binário
- A Estrutura de Dados do Heap Binário é um tipo especial de árvore binária,
uma árvore binária quase completa mais especificamente



Heap sort

- Onde que os elementos das folhas são lidos da esquerda para a direita e por ser binária cada nó tem no máximo 2 elementos filhos
 - considerada quase completa pois todos os níveis são preenchidos, exceto talvez o último nível que pode ter um elemento faltando na última folha

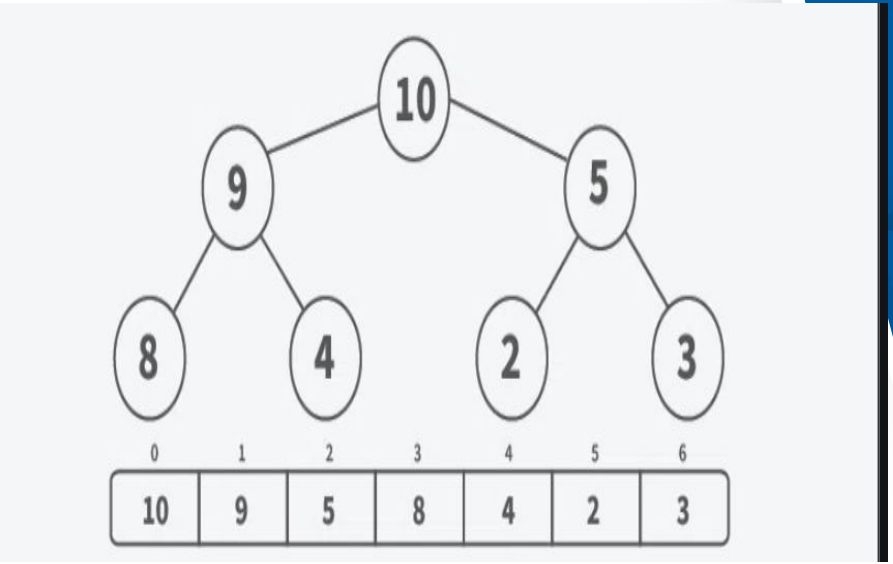


Etapas do Algoritmo

1 - Construção do heap (Max-Heap ou Min-Heap):

- Se for usado Max-Heap, é necessário organizar a estrutura de forma que cada nó pai seja maior ou igual aos seus filhos , isso é feito quando queremos ordenar em ordem crescente .

- Se for usado Min-Heap, a estrutura deve garantir que cada nó pai seja menor ou igual aos seus filhos , isso é feito quando queremos ordenar em ordem decrescente



Etapas do Algoritmo

2 - Em um Heap Sort, após a construção do Max-Heap (ou Min-Heap), o algoritmo segue os seguintes passos:

- **Troca o elemento da raiz (que é o maior no Max-Heap ou o menor no Min-Heap) com o último elemento do array (ou da árvore binária).**
- **Remove (ou desconsidera) esse último elemento da estrutura ativa, pois ele já está na posição correta da ordenação.**
- **Reorganiza (heapifica) a árvore novamente para restaurar a propriedade do Max-Heap ou Min-Heap.**
- **Repete esse processo: troca a nova raiz com o último elemento restante, desconsidera esse elemento, e heapifica novamente.**
- **Esse processo se repete até restar apenas um elemento na estrutura.**

Índices: A Chave da Estrutura no Heap Sort

- Esse algoritmo é muito eficiente quando implementado usando arrays (vetores), pois a estrutura de heap pode ser simulada diretamente pelos índices dos elementos, sem precisar de ponteiros ou nós como em uma árvore real
- Para todos os indices `i do vetor vamos ter essas relações:`
- `P  = indice do elemento pai`

Pai: `(i -1) / 2`

último nó pai: `(n/2)-1`

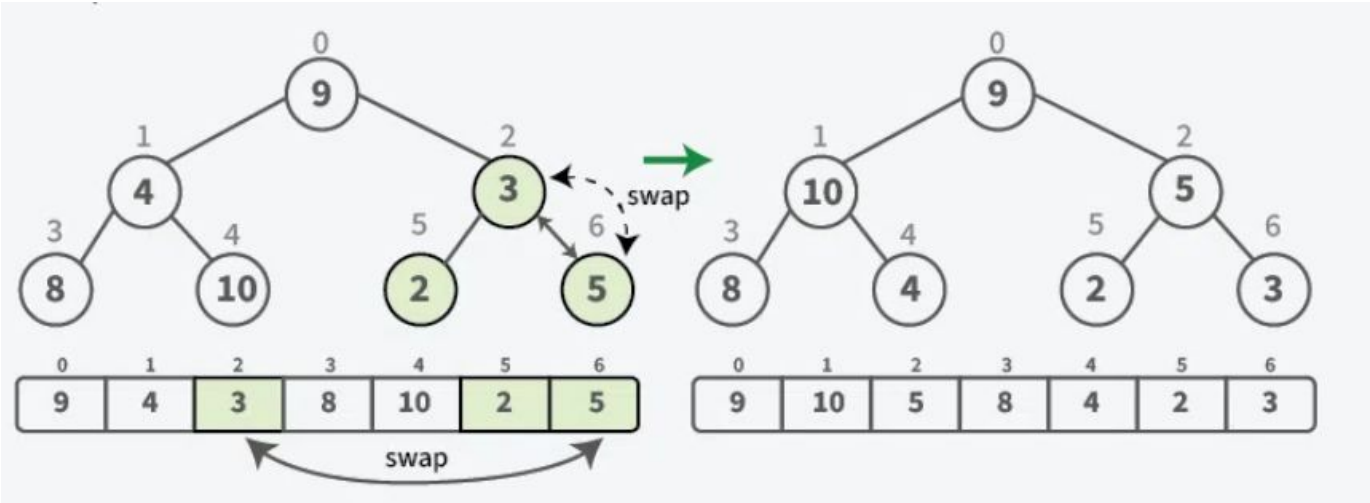
Filho da esquerda: `2 * p + 1`

Filho da direita : `2 * p + 2`

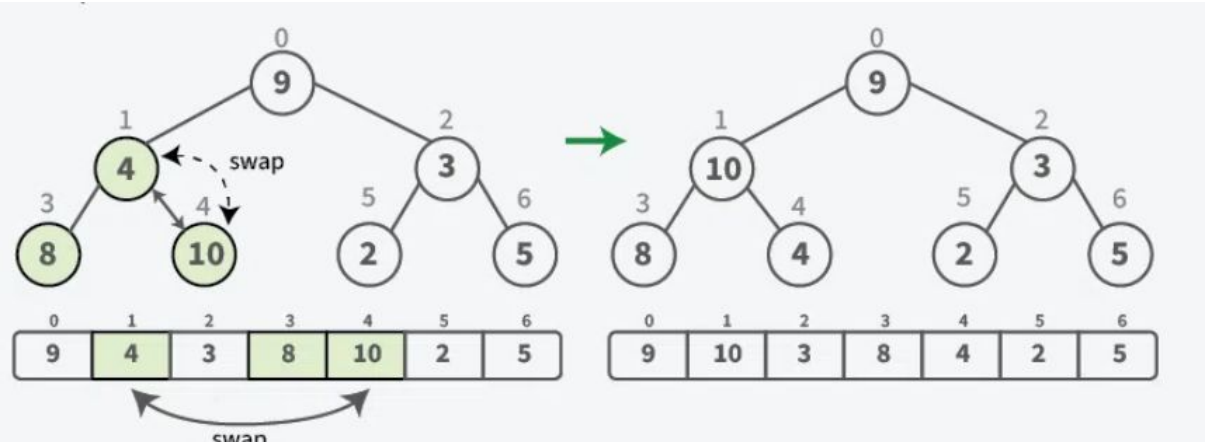
Construindo o max heap

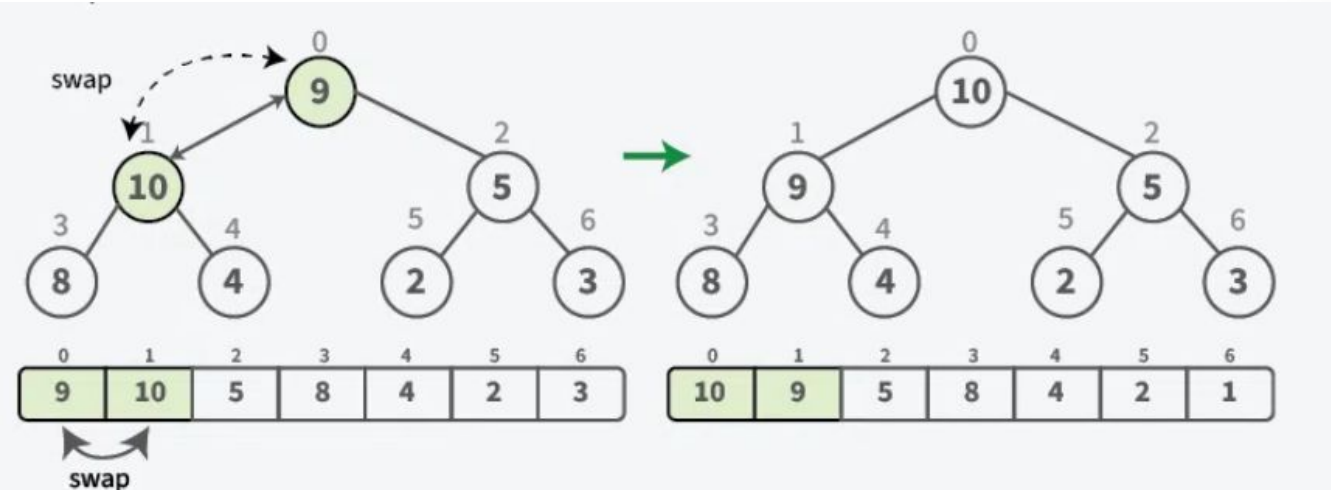
Como exemplo para mostrar como funciona o algoritmo vamos construir o max heap

índice último nó pai:
(n/2)-1



Se diminuirmos 1 no valor do indice do antigo pai, acessamos o pai ao lado





Observe que agora, do lado direito, temos um Max Heap completamente montado, em que o valor de cada nó pai é maior que os valores de seus nós filhos

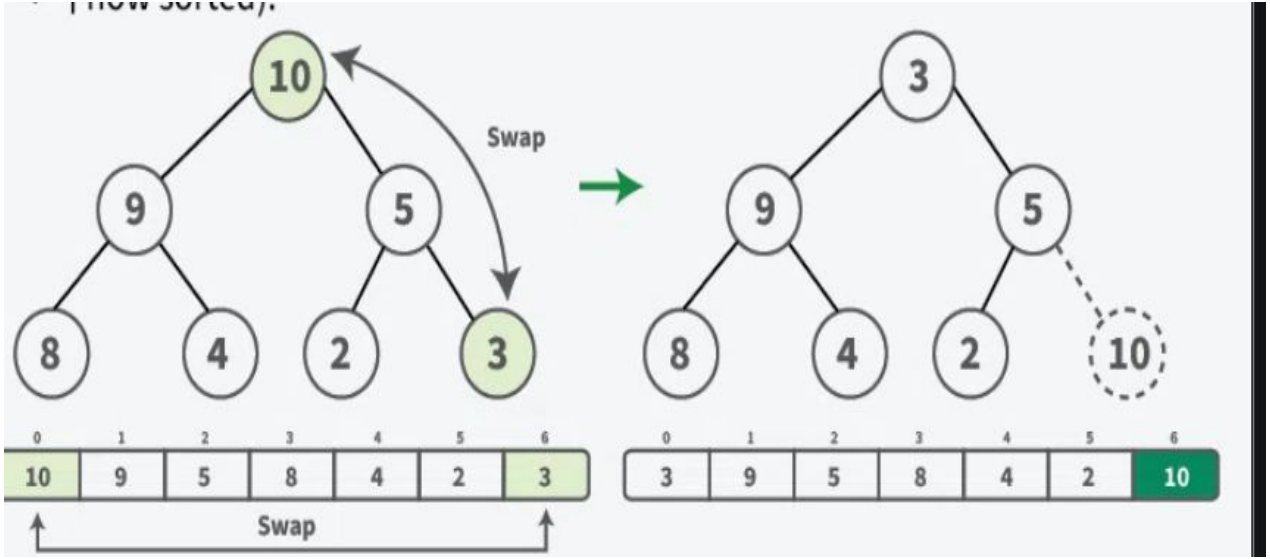
Como podemos observar, ao montar o Max Heap, é necessário percorrer todos os elementos para comparar e garantir que o maior valor fique no nó pai. Por isso, a complexidade dessa operação é linear em todos os casos, ou seja, no pior, médio e melhor caso, teremos uma complexidade de O(n)

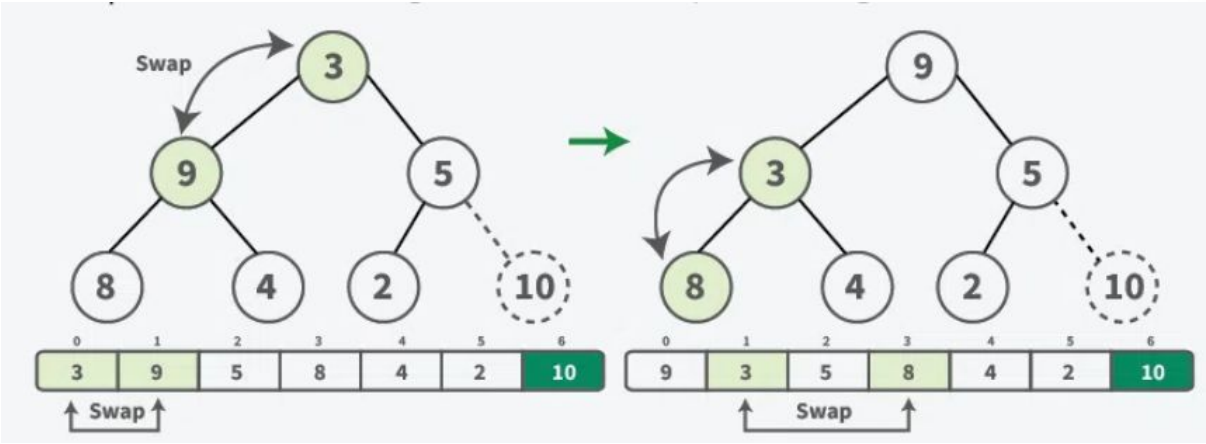
processo de codificação

```c
void construir_max_heap(int arr[], int n , unsigned long long int *contador_trocas) {

    for (int i = n / 2 - 1; i >= 0; i--) {
        max_heapify(arr, n, i, contador_trocas);
    }
}       You, last week • fazendo base para implementar heap sort
```

```c
void max_heapify(int arr[], int n, int i , unsigned long long int *contador_trocas) {
    int maior = i;
    int esq = 2 * i + 1;
    int dir = 2 * i + 2;

    if (esq < n && arr[esq] > arr[maior]) {
        maior = esq;
    }


    if (dir < n && arr[dir] > arr[maior]) {
        maior = dir;
    }


    if (maior != i) {
        swap(&arr[i], &arr[maior], contador_trocas);


        max_heapify(arr, n, maior, contador_trocas);
    }
}       You, last week • fazendo base para implementar heap sort
```

Segunda parte do algoritimo





É aqui que está o segredo do porquê, no pior caso, o algoritmo possui complexidade O(n log n) e não O(n²)

Ao trocar a raiz com o último elemento, é necessário reorganizar o Max Heap para manter sua estrutura. Isso exige percorrer parte da árvore, e no pior caso, pode ser preciso descer até o último nível para restabelecer a propriedade de heap

Relação entre a altura e o número de elementos

Em uma árvore binária completa, existe uma relação direta entre a altura da árvore (h) e o número total de nós (n).

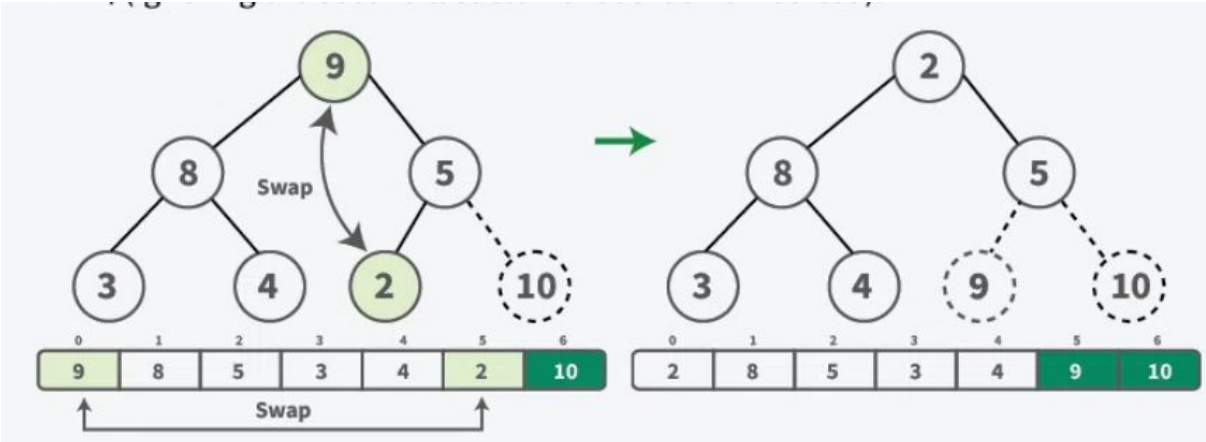
O número máximo de nós em uma árvore binária completa de altura h é dado por:

$$n \leq 2^h$$

Aplicando log dos dois lados

Vamos que a altura h da árvore é proporcional a $\log_2(n) \leq h$
Isso mostra que a profundidade máxima de uma árvore binária completa cresce de forma logarítmica em relação ao número de elementos.

Por conta disso no pior caso onde vamos ter que descer até o final da árvore vamos precisar fazer no máximo $\log_2(n)$ operações


9
8
5
3
4
2
10
Swap
0 1 2 3 4 5 6
9 8 5 3 4 2 10
Swap
2
8
5
3
4
9
10
0 1 2 3 4 5 6
2 8 5 3 4 9 10

processo de codificação

```
196  void heap_sort_crescente(int arr[], int n, unsigned long long int *contador_trocas) {
197
198      construir_max_heap(arr, n , contador_trocas);
199
200
201      for (int i = n - 1; i > 0; i--) {
202          swap(&arr[0], &arr[i], contador_trocas);
203
204
205          max_heapify(arr, i, 0, contador_trocas);
206      }
207  }       You, last week • fazendo base para implementar heap sort
```

Complexidade matematica

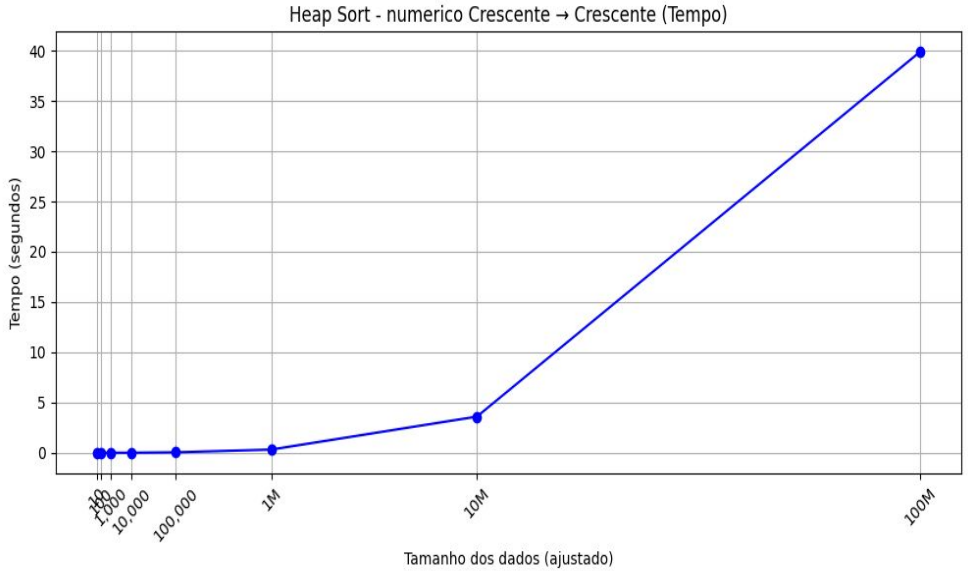
A construção do Max Heap (ou Min Heap) tem complexidade O(n).
Depois, durante a ordenação, cada remoção da raiz e reorganização do heap tem custo $O(\log_2(n))$, e isso acontece n vezes.

Por isso, a complexidade total do algoritmo é:
$O(n \log_2(n))$

Análise Empírica de Métricas de Ordenação Numérica

1. Ordenação de Dados Já Ordenados em Ordem Crescente → Crescente

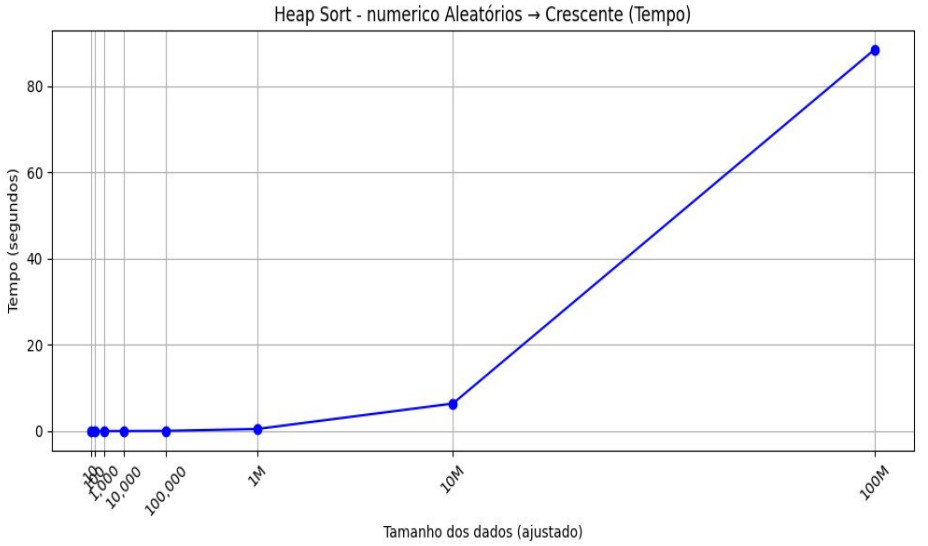
Tamanho	Tempo (s)	Trocas	Primeiros Elementos (Ordenados)
10	0.0000060	30	1 2 3 4 5 6 7 8 9 10
100	0.0000240	640	1 2 3 ... 20
1,000	0.0002510	9,708	1 2 3 ... 20
10,000	0.0033960	131,956	1 2 3 ... 20
100,000	0.0430140	1,650,854	1 2 3 ... 20
1M	0.3171820	19,787,792	1 2 3 ... 20
10M	3.5986860	231,881,708	1 2 3 ... 20
100M	39.8949720	2,652,454,802	1 2 3 ... 20



Análise Empírica de Métricas de Ordenação Numérica

2. Ordenação de Dados Aleatórios → Crescente

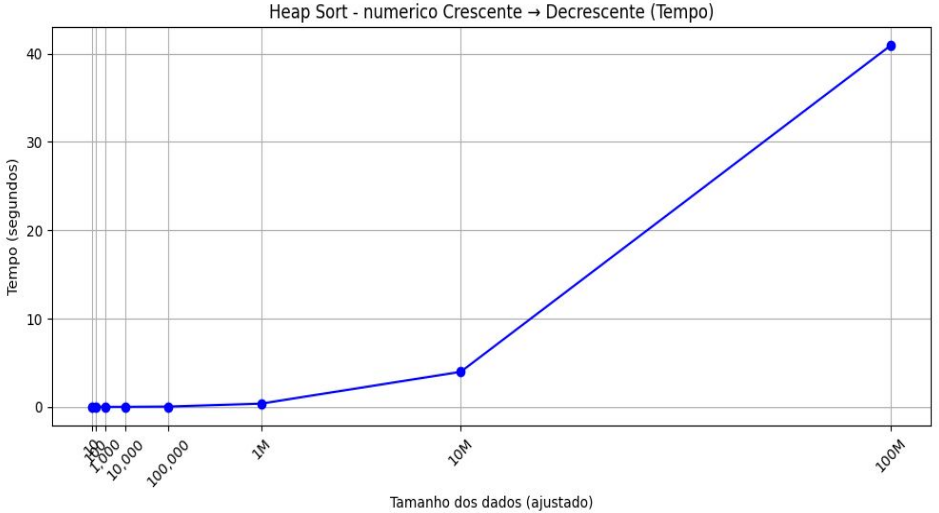
Tamanho	Tempo (s)	Trocas	Primeiros Elementos (Ordenados)
10	0.0000030	27	5136076 73478952 ... 2091243229
100	0.0000150	584	5136076 23149027 ... 444835065
1,000	0.0001920	9,084	5136076 5607661 ... 47472618
10,000	0.0030320	124,194	596055 597860 ... 4502919
100,000	0.0361390	1,575,030	9933 57368 ... 617784
1M	0.4662420	19,048,368	272 1165 ... 40543
10M	6.3665680	223,836,447	216 272 ... 3508
100M	88.4744070	2,571,583,550	58 72 ... 430



Análise Empírica de Métricas de Ordenação Numérica

3. Ordenação de Dados Já Ordenados em Ordem Crescente → Decrescente

Tamanho	Tempo (s)	Trocas	Primeiros Elementos (Ordenados)
10	0.0000030	21	10 9 8 ... 1
100	0.0000140	516	100 99 ... 81
1,000	0.0001750	8,316	1000 999 ... 981
10,000	0.0026070	116,696	10000 9999 ... 9981
100,000	0.0319030	1,497,434	100000 99999 ... 99981
1M	0.3681550	18,333,408	1000000 999999 ... 999981
10M	3.9698390	216,912,428	10000000 9999999 ... 9999981
100M	40.9332120	2,500,251,420	100000000 99999999 ... 99999981

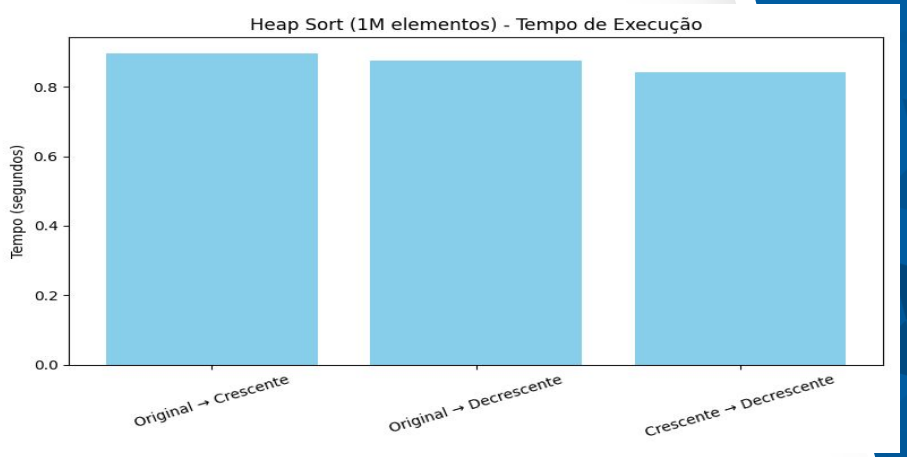


Análise Empírica de Métricas de Ordenação Textual

Resultados dos Testes de Ordenação (1.000.000 elementos)

heap Sort - Dados Textuais

Caso de Teste	Direção da Ordenação	Tempo de Execução (s)	Número de Trocas
Original → Crescente	CRESCENTE	0.8976200	19,048,277
Original → Decrescente	DECRESCENTE	0.8764140	19,048,448
Crescente → Decrescente	DECRESCENTE	0.8428970	18,333,401

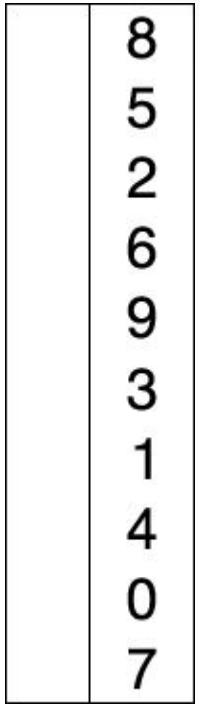


Selection Sort

Como funciona

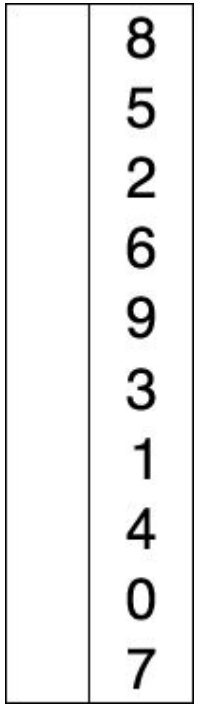
- O Selection Sort é um método de ordenação bem simples, ele seleciona o menor elemento da sequência e coloca na primeira posição do array. Este processo ocorre N vezes, ou seja, ele irá ocorrer até que o array esteja ordenado.

- Passo a passo
  - Percorre o vetor do início ao fim.
  - Em cada posição, encontra o menor valor no restante do vetor.
  - Troca esse menor valor com o valor da posição atual.
  - Repete o processo para próxima posição, até o penúltimo elemento.



Análise de complexidade

- Complexidade de tempo
  - Pior caso: O(n²)
  - Caso médio: O(n²)
  - Melhor caso: O(n²)
- Isso ocorre pois, o número de comparação é sempre o mesmo, pois os dois laços aninhados sempre percorrem toda a parte não ordenada do vetor/lista, independente do estado inicial dos dados.
- Complexidade do espaço
  - O(1)
- Número de trocas
  - Pior caso: n-1 trocas (O(n))
  - Melhor caso: 0(se já estiver ordenado)
  -



Implementação: Vetor vs Lista

```c
void selection_sort_vetor_crescente(int arr[], int n, unsigned long long int *contador_trocas) {
    int i, j, min_idx;
    for (i = 0; i < n - 1; i++) {
        min_idx = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        if (min_idx != i) {
            swap_int(&arr[min_idx], &arr[i], contador_trocas);
        }
    }
}
```

⇨ O acesso direto aos elementos arr[j] e arr[min_idx] é muito eficiente.

Implementação: Vetor vs Lista

```c
void selection_sort_lista_crescente(Lista* lista, unsigned long long int *contador_trocas) {
    if (!lista || lista->tamanho < 2) return;

    No *i, *j;
    for (i = lista->inicio; i->proximo != NULL; i = i->proximo) {
        No *min_no = i;
        for (j = i->proximo; j != NULL; j = j->proximo) {
            if (j->valor < min_no->valor) {
                min_no = j;
            }
        }
        if (min_no != i) {
            swap_nos(i, min_no, contador_trocas);
        }
    }
}
```

⇨ Necessidade de percorrer a lista com ponteiros.

Implementação: Vetor vs Lista

Conclusão

- Embora a complexidade teórica seja a mesma $O(n^2)$, a implementação em lista é significativamente mais lenta na prática devido ao custo do acesso sequencial aos dados.

Desempenho Empírico-Dados Números (vetor)

1. Ordenação de Dados Já Ordenados em Ordem Crescente → Crescente

Tamanho	Tempo (s)	Trocas	Primeiros Elementos (Ordenados)
10	0.0000020	0	1 2 3 4 5 6 7 8 9 10
100	0.0000130	0	1 2 3 ... 20
1.000	0.0007240	0	1 2 3 ... 20
10.000	0.0765580	0	1 2 3 ... 20
100.000	4.2839210	0	1 2 3 ... 20
1.000.000	432.6217190	0	1 2 3 ... 20
10.000.000	~12 horas	0	(não executado)
100.000.000	~50 dias	0	(não executado)

Desempenho Empírico-Dados Númericos (vetor)

2. Ordenação de Dados Aleatórios → Crescente

Tamanho	Tempo (s)	Trocas	Primeiros Elementos (Ordenados)
10	0.0000040	5	5136076 73478952 ... 2091243229
100	0.0000110	94	5136076 23149027 ... 444835065
1.000	0.0005590	994	5136076 5607661 ... 47472618
10.000	0.0742650	9.988	596055 597860 ... 4502919
100.000	4.2380680	99.987	9933 57368 ... 617784
1.000.000	427.1488810	999.980	272 1165 ... 40543
10.000.000	~12 horas	9.999.999	(estimado)
100.000.000	~50 dias	99.999.999	(estimado)

Desempenho Empírico-Dados Númericos (vetor)

3. Ordenação de Dados Já Ordenados em Ordem Crescente → Decrescente

Tamanho	Tempo (s)	Trocas	Primeiros Elementos (Ordenados)
10	0.0000040	5	10 9 8 ... 1
100	0.0000160	50	100 99 ... 81
1.000	0.0011190	500	1000 999 ... 981
10.000	0.0822190	5.000	10000 9999 ... 9981
100.000	4.4191140	50.000	100000 99999 ... 99981
1.000.000	441.7858740	500.000	1000000 999999 ... 999981
10.000.000	~12 horas	5.000.000	(estimado)
100.000.000	~50 dias	50.000.000	(estimado)

Desempenho Empírico-Dados Númericos (lista)

1. Ordenação de Lista Crescente → Crescente

Tamanho	Tempo (s)	Tempo Humano	Trocas	Primeiros Elementos (Ordenados)
10	0.0000050	Instantâneo	0	1 2 3 ... 10
100	0.0000090	Instantâneo	0	1 2 3 ... 20
1.000	0.0007150	0.001 segundos	0	1 2 3 ... 20
10.000	0.0927730	0.09 segundos	0	1 2 3 ... 20
100.000	6.8440800	~7 segundos	0	1 2 3 ... 20
1.000.000	1071.7271220	**~18 minutos**	0	1 2 3 ... 20
10M	(estimado) ~29,8 horas	(não executado)	0	(não executado)
100M	(estimado) ~124 dias	(não executado)	0	(não executado)

Desempenho Empírico-Dados Númericos (lista)

2. Ordenação de Lista Aleatória → Crescente

Tamanho	Tempo (s)	Tempo Humano	Trocas	Primeiros Elementos (Ordenados)
10	0.0000020	Instantâneo	5	5136076 73478952 ... 2091243229
100	0.0000150	Instantâneo	94	5136076 23149027 ... 444835065
1.000	0.0009440	0.001 segundos	994	5136076 5607661 ... 47472618
10.000	0.0771650	0.08 segundos	9.988	596055 597860 ... 4502919
100.000	6.8243710	~7 segundos	99.987	9933 57368 ... 617784
1.000.000	1072.0005770	**~18 minutos**	999.980	[padrão similar aos anteriores]
10M	(estimado) ~29,8 horas	(não executado)	9.999.999	[omissos por volume]
100M	(estimado) ~124 dias	(não executado)	99.999.999	[omissos por volume]

Desempenho Empírico-Dados Númericos (lista)

3. Ordenação Inversa de Lista Crescente → Decrescente

Tamanho	Tempo (s)	Tempo Humano	Trocas	Primeiros Elementos (Ordenados)
10	0.0000030	Instantâneo	5	10 9 8 ... 1
100	0.0000170	Instantâneo	50	100 99 ... 81
1.000	0.0007400	0.001 segundos	500	1000 999 ... 981
10.000	0.0828550	0.08 segundos	5.000	10000 9999 ... 9981
100.000	6.9606880	~7 segundos	50.000	100000 99999 ... 99981
1.000.000	1050.4134280	**~17.5 minutos**	500.000	1000000 999999 ... 999981
10M	(estimado) ~29,2 horas	(não executado)	5.000.000	10000000 9999999 ... 9999981
100M	(estimado) ~123 dias	(não executado)	50.000.000	100000000 99999999 ... 99999981

Desempenho Empírico-Dados Textuais (lista)

1. Implementação com Vetor (Array)

A implementação em vetor se beneficia do acesso contíguo à memória, o que melhora a localidade de cache. No entanto, isso não é suficiente para superar a complexidade quadrática.

Caso de Teste	Direção da Ordenação	Tempo de Execução (Estimado)	Nro. de Trocas (Máximo)	Observações
Original → Crescente	CRESCENTE	**~7 horas**	999.999	Ordenação padrão de dados aleatórios. O tempo é dominado pelas comparações.
Original → Decrescente	DECRESCENTE	**~7 horas**	999.999	Performance similar ao caso crescente, pois o número de comparações não muda.
Crescente → Decrescente	DECRESCENTE	**~7 horas**	999.999	Pior caso para trocas. Cada elemento precisa ser movido.

Desempenho Empírico-Dados Textuais (vetor)

2. Implementação com Lista Duplamente Encadeada

A versão com lista duplamente encadeada sofre uma penalidade de performance devido ao acesso sequencial via ponteiros (ponteiro->proximo), que impede a otimização de cache pelo processador.

Caso de Teste	Direção da Ordenação	Tempo de Execução (Estimado)	Nro. de Trocas (Máximo)	Observações
Original → Crescente	CRESCENTE	**~8.5 horas**	999.999	Mais lento que o vetor devido ao custo do acesso sequencial por ponteiros.
Original → Decrescente	DECRESCENTE	**~8.5 horas**	999.999	O número de comparações e a lógica de acesso permanecem os mesmos.
Crescente → Decrescente	DECRESCENTE	**~8.5 horas**	999.999	A complexidade quadrática do acesso aos nós domina qualquer outro fator.

Conclusão

Vantagens

- Simples de implementar e entender;
- Requer pouca memória;
- Minimo de trocas;

Desvantagens

- Extremamente lento para grandes entradas;
- Não se beneficia de dados pré-ordenados;
- Muita ineficiência quando implementado com listas encadeadas;

Obrigado

Dúvidas ou sugestões?

Github do projeto
:https://github.com/AndreFilho0/trabalho_final_aed1

INSTITUTO DE INFORMÁTICA UFG

UFG UNIVERSIDADE FEDERAL DE GOIÁS