

Curso: Engenharia de Computação		Disciplina: Estruturas e recuperação de dados B	
Período: 04	Turma: 01	Turno: Integral	Data: 15/10/2020
Nome			RA
1 – Guilherme Marques Brait Garros			19264266
2 - Lucas Grass Beraldo			19246925
3 – Ramon Gazoni Lacerda			20062162
4 – Renato Barba dos Santos			19246529

1. INTRODUÇÃO

Nesse relatório abordaremos 5 algoritmos de ordenação: o Selection Sort, o Insertion Sort, o Merge Sort, o Quick Sort e o Heap Sort. Antes de abordá-los, iremos discutir um pouco do objetivo desses algoritmos e a busca pelos programadores do algoritmo ideal.

Métodos para a organização dos nossos dados ou objetos existem a séculos e os fazemos por um objetivo bem simples: a otimização de processos. Além disso, com a criação dos computadores e a possibilidade da organização ser feita por uma máquina, métodos de ordenação começaram a ser pensados documentados. Em meados de 1951 Betty Holberton já estava trabalhando com algoritmos de ordenação primitivos e em 1956 o Bubble Sort chega a ser implementado. Com o aumento da tecnologia, estudos e quantidade de dados, os algoritmos foram se aperfeiçoando, buscando serem cada vez mais rápidos. Apesar dos programadores e acadêmicos nem sempre conseguirem fazer algoritmos rápidos em todos os quesitos, eles melhoravam alguns aspectos com novas abordagens, como a busca da informação ou a organização para poucos números, ou a organização estável de um grupo de dados. Portanto, os algoritmos de ordenação não devem ser vistos como um melhor que o outro, mas deve-se analisar o problema proposto e identificar qual serve melhor para o problema em questão.

Mesmo assim os programadores continuam num estudo contínuo para desenvolver soluções mais rápidas que as atuais. Essa busca incessante ocorre paralelamente com o constante aumento do volume de dados que tratamos atualmente, fazendo com que busquemos maneiras mais rápidas para fazer com que esse aumento não prejudique os processos. Uma simples otimização de processo pode trazer ganhos consideráveis, seja no campo acadêmico ou empresarial. Nesse contexto tratamos de algumas dessas importantes metodologias de ordenação que foram criadas desde as primeiras implementações para organização de dados em computadores até códigos considerados os mais importantes do século XXI.

2. RESULTADOS E DISCUSSÃO

2.1. SELECTION SORT

O Selection Sort é um algoritmo com tempo linear, que garante a cada execução a ocupação da posição $P = \text{Número da Execução} - 1$, pelo menor elemento da parte não ordenada do Vetor. A cada execução o algoritmo busca pelo menor elemento da parte não ordenada do vetor, e coloca ele na primeira posição da parte não ordenada, ou seja, primeiro ele busca o menor elemento e depois troca esse elemento com a posição $P = \text{Número da Execução} - 1$.

218040 – ESTRUTURAS E RECUPERAÇÃO DE DADOS
B

```
lucas@lucas-VirtualBox: ~/Área de Trabalho/Estrutur
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$ gcc selection10.c -o selection10
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$ ./selection10

Para 10 elementos

Vetor gerado aleatoriamente nao ordenado
[1][916][785][85][554][287][469][927][831][139]
Vetor ordenado
[1][85][139][287][469][554][785][831][916][927]

Tempo = Instantaneo
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$
```

Figura 1. Selection Sort com 10 elementos.

```
lucas@lucas-VirtualBox: ~/Área de Trabalho/Estrutur
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$ gcc SelectionSort.c -o SelectionSort
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$ ./SelectionSort

Para 1.000.000 de elementos

Tempo = 909673 ms
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$
```

Figura 2. Selection Sort com 1 milhão de elementos.

2.2. INSERTION SORT

O Insertion Sort é um algoritmo com tempo linear, que garante a cada execução a ocupação da posição $P = \text{Número da Execução} - 1$, pelo menor elemento da parte não ordenada do vetor. Sendo assim, a cada execução ele compara o elemento da posição $i+1$ com o elemento da posição i , caso $i+1$ for menor do que i , troca os elementos do vetor de posição e subtrai 1 em cada, caso não for menor, interrompe o processo de comparação e vai para a próxima posição.

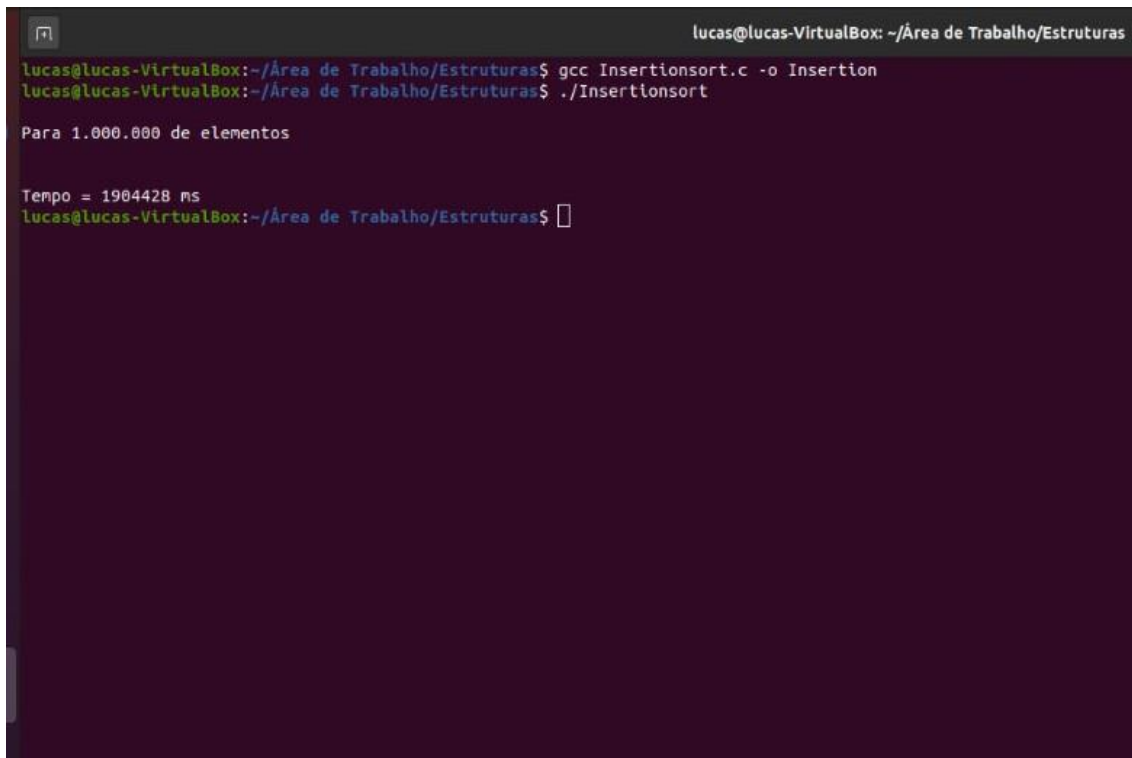
```
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$ gcc insertion10.c -o insertion
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$ ./insertion10

Para 10 elementos

Vetor gerado aleatoriamente nao ordenado
[647][24][578][432][876][749][48][115][125][171]
Vetor ordenado
[24][48][115][125][171][432][578][647][749][876]

Tempo = Instantaneo
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$
```

Figura 3. Insertion Sort com 10 elementos.



```
lucas@lucas-VirtualBox: ~/Área de Trabalho/Estruturas
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$ gcc Insertionsort.c -o Insertion
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$ ./Insertionsort

Para 1.000.000 de elementos

Tempo = 1904428 ms
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$
```

Figura 4. Insertion Sort com 1 milhão de elementos.

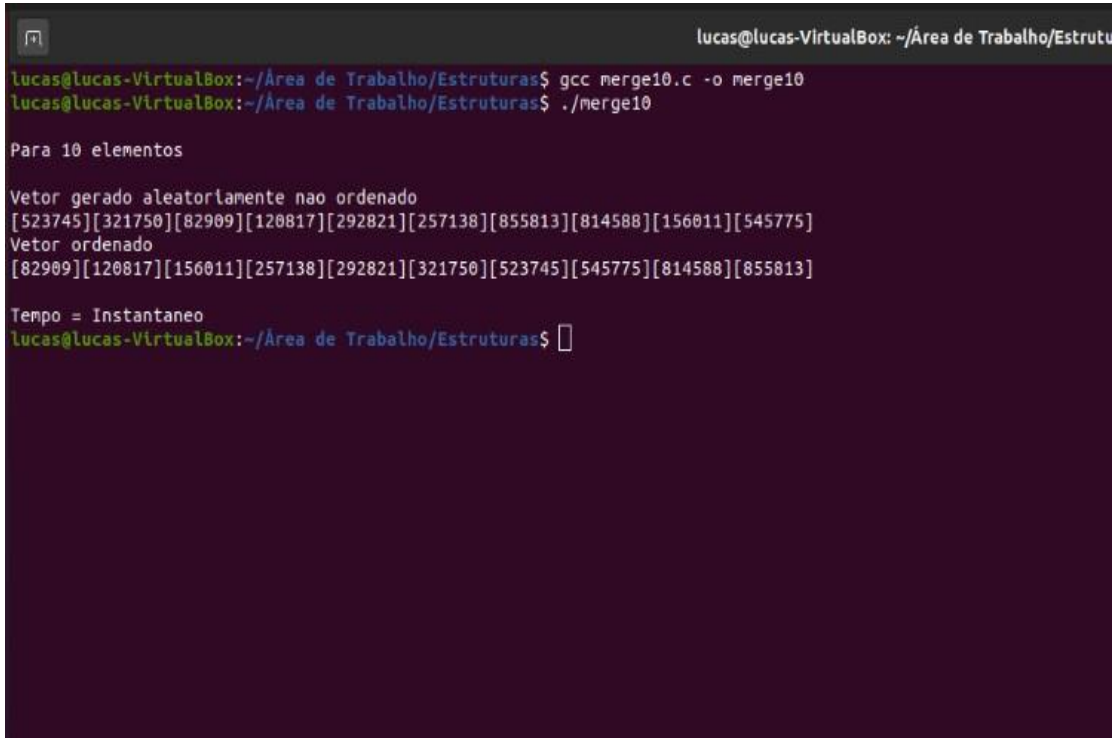
2.3. MERGE SORT

O Merge Sort é um algoritmo que se baseia em uma estratégia militar de dividir para conquistar [2], com isso, o Vetor é dividido na metade e cada subgrupo é dividido no meio até que todos tenham apenas um elemento. Essa parte do algoritmo é feito a partir de recursão chamando o próprio Merge Sort do início até a metade e da metade + 1 até o fim. Ao fim das chamadas recursivas, cada chamada vai para função merge que faz voltar os subgrupos e ordena eles, ou seja, quando juntam dois subgrupos de um elemento eles são ordenados e formam um subgrupo de dois elementos sucessivamente até chegar em apenas dois subgrupos, isso irá se repetir até fechar a última recursão.

Com as duas metades ordenadas o Vetor passa novamente pela função merge para juntar as últimas metades e ordená-las também. Normalmente, a função merge é feita com dois sub vetores, um com a metade da esquerda e outro com a metade da direita, mas a equipe escolheu utilizar apenas um sub vetor e duas variáveis auxiliares,

**218040 – ESTRUTURAS E RECUPERAÇÃO DE DADOS
B**

para simular os dois vetores, e andar com essas duas variáveis de acordo com as condições para o algoritmo decorrer menos tempo e ocupar menos espaço na memória.



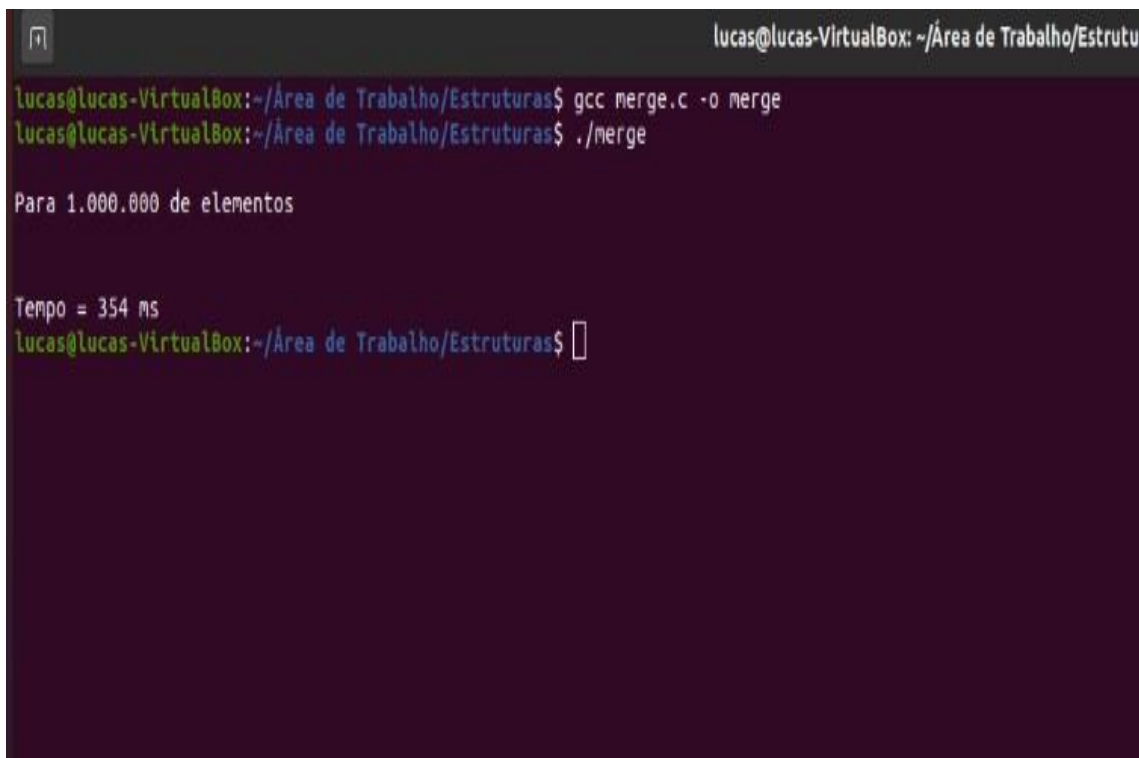
```
lucas@lucas-VirtualBox: ~/Área de Trabalho/Estruturas
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$ gcc merge10.c -o merge10
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$ ./merge10

Para 10 elementos

Vetor gerado aleatoriamente nao ordenado
[523745][321750][82909][120817][292821][257138][855813][814588][156011][545775]
Vetor ordenado
[82909][120817][156011][257138][292821][321750][523745][545775][814588][855813]

Tempo = Instantaneo
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$
```

Figura 5. Merge Sort com 10 elementos.



```
lucas@lucas-VirtualBox: ~/Área de Trabalho/Estruturas$ gcc merge.c -o merge
lucas@lucas-VirtualBox: ~/Área de Trabalho/Estruturas$ ./merge

Para 1.000.000 de elementos

Tempo = 354 ms
lucas@lucas-VirtualBox: ~/Área de Trabalho/Estruturas$
```

Figura 6. Merge Sort com 1 milhão de elementos.

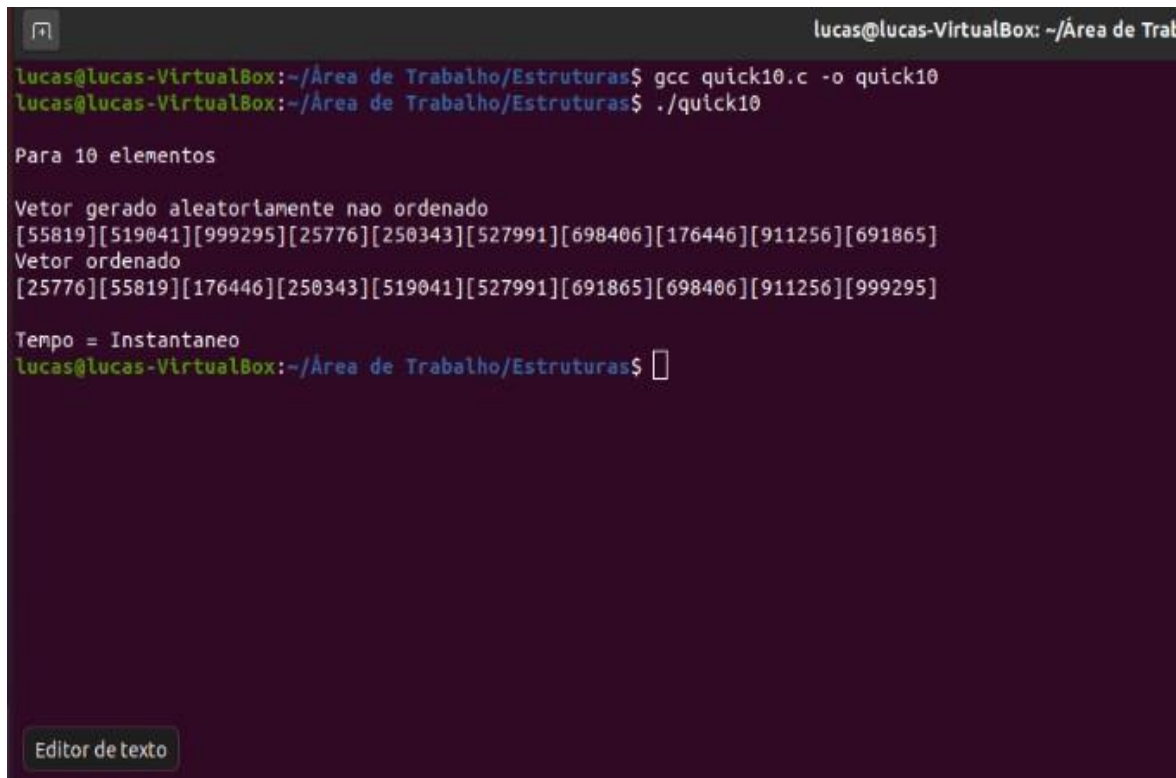
2.4. QUICK SORT

O Quick Sort foi desenvolvido por Charles Antony Richard Hoare em 1960 em uma tentativa de traduzir russo para inglês. O primeiro desenvolvimento do algoritmo foi apenas teórico pois na época não foi possível implementá-lo, sendo implementado pela primeira vez somente em 1980 e rendendo ao seu desenvolvedor um Turing Award, prêmio concedido todos os anos para uma pessoa por contribuição à computação. O algoritmo foi amplamente difundido após o estudo e refinamento feito pelo Robert Sedgwick, Professor da universidade de Princeton.

O primeiro passo para garantir uma boa execução do Quick Sort é realizar um Shuffle [1], processo que embaralha de forma aleatória os elementos do vetor, pois diferente do Merge Sort, ele não se beneficia de uma pré ordenação dos dados. Após esse processo, ele utiliza uma estratégia semelhante ao do Merge Sort de conquistar e dividir: ele seleciona um pivô e o utiliza para dividir o vetor em duas partes. O algoritmo separa na esquerda (parte da posição 0 até o pivô) os elementos menores que o pivô e na direita (parte do pivô até a posição final), entrando em uma recursão que continua

**218040 – ESTRUTURAS E RECUPERAÇÃO DE DADOS
B**

dividindo o vetor até chegar em um unitário (um vetor de apenas um elemento), voltando e dividindo novamente os vetores restantes sem considerar a parte já ordenada, garantido a ordenação completa dos dados. É possível fazer muitas melhorias no código de acordo com cada problema acelerando ainda mais sua execução.



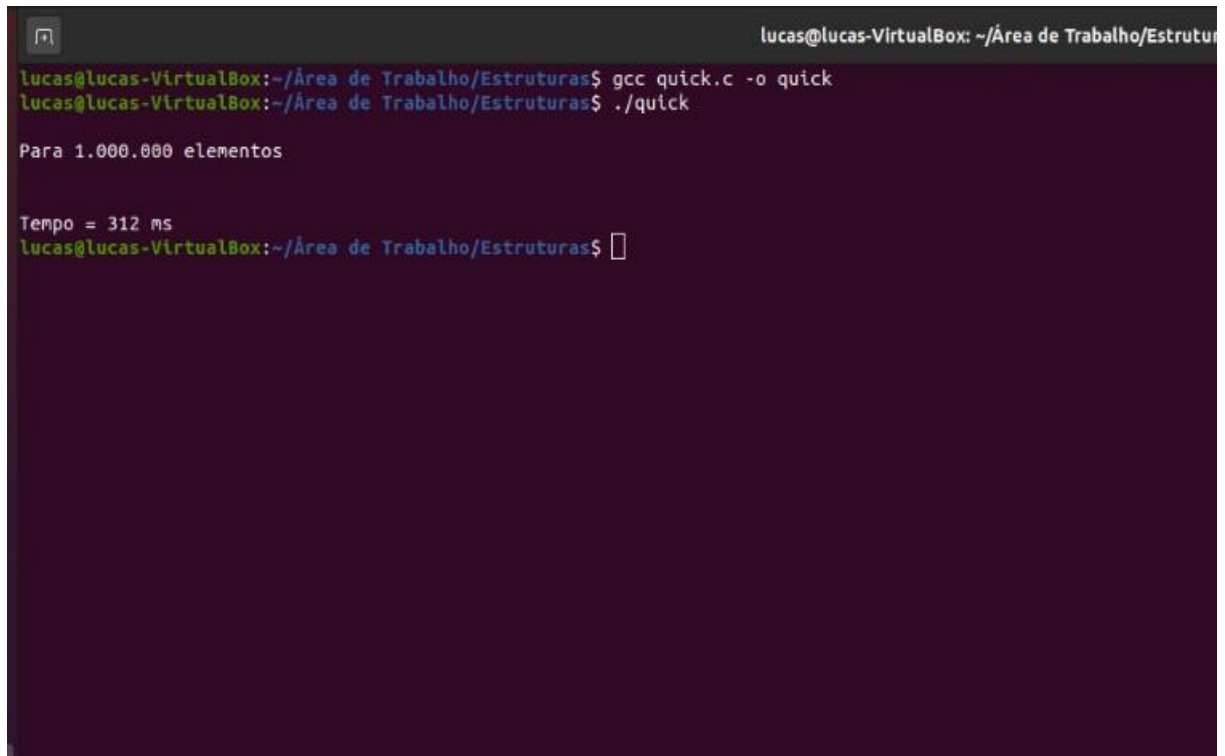
```
lucas@lucas-VirtualBox: ~/Área de Trabalho/Estruturas$ gcc quick10.c -o quick10
lucas@lucas-VirtualBox: ~/Área de Trabalho/Estruturas$ ./quick10

Para 10 elementos

Vetor gerado aleatoriamente não ordenado
[55819][519041][999295][25776][250343][527991][698406][176446][911256][691865]
Vetor ordenado
[25776][55819][176446][250343][519041][527991][691865][698406][911256][999295]

Tempo = Instantaneo
lucas@lucas-VirtualBox: ~/Área de Trabalho/Estruturas$
```

Figura 7. Quick Sort com 10 elementos.



```
lucas@lucas-VirtualBox: ~/Área de Trabalho/Estruturas
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$ gcc quick.c -o quick
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$ ./quick

Para 1.000.000 elementos

Tempo = 312 ms
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$
```

Figura 8. Quick Sort com 1 milhão de elementos.

2.5. HEAP SORT

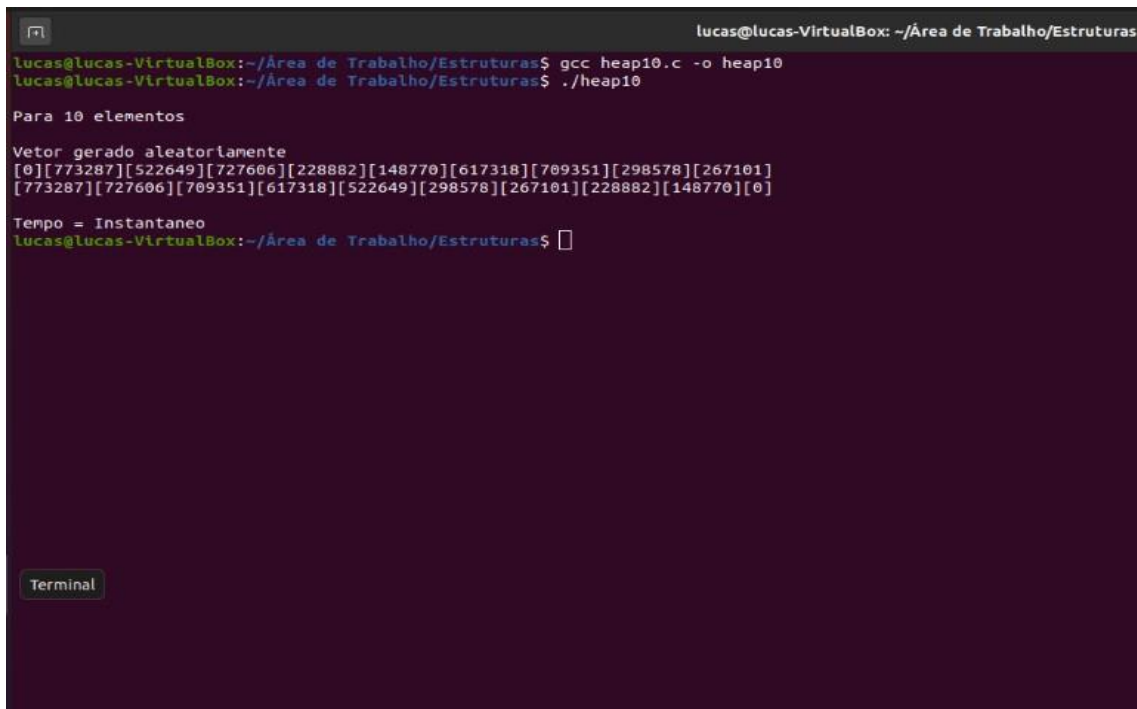
O Heap Sort é um método de ordenação bastante eficiente no qual se utiliza de uma estrutura de dados chamada heap. Essa estrutura de dados simula uma árvore em um vetor, isso traz um grande diferencial para a ordenação, podendo acessar e navegar pelo vetor como se fosse a estrutura de árvore, e a partir disso, podemos buscar elementos e varrer a árvore num tempo de aproximadamente $\log N$. Ao incorporar esse comportamento, o Heap Sort torna-se capaz de fazer sua ordenação com um número de $N \log N$ comparações e trocas, mesmo no pior caso, tornando-se um dos seus diferenciais. Além disso, o algoritmo em questão utiliza somente a memória alocada pelo vetor, mas não é um método de ordenação estável, é iterativo e de média dificuldade de implementação de comparado com os demais algoritmos de ordenação.

Para que se entenda um pouco mais sobre o Heap Sort o descreveremos um pouco. Iniciamos com vetor desordenado e nosso primeiro passo é implementar uma regra, todas árvores – sendo ela a principal ou uma sub árvore dentro do vetor - devem ter o

**218040 – ESTRUTURAS E RECUPERAÇÃO DE DADOS
B**

número do pai maior que dos seus filhos. Para isso percorremos as sub árvores fazendo essa análise e se a regra não estiver sendo cumprida afundamos o pai, ou seja, o trocamos com o maior filho entre os dois, e se esse novo filho tiver outros 2 filhos também verificamos se é cumprida a regra e fazemos isso até chegarmos na folha. Após percorrermos essas sub árvores e chegarmos no pai (ou raiz) e fizermos a análise para ele, teremos todo vetor com essa regra verificada. Como os filhos são menores que o pai temos certeza que a raiz (ou pai central) de toda árvore é o maior número daquele vetor, logo retiramos ele e o imprimimos. Para que o primeiro espaço não fique vazio pegamos o último filho, colocamos ele na primeira posição e o afundamos. Ao afundá-lo novamente podemos garantir que o maior valor estará no pai e por aí vai até terminar o vetor.

218040 – ESTRUTURAS E RECUPERAÇÃO DE DADOS B



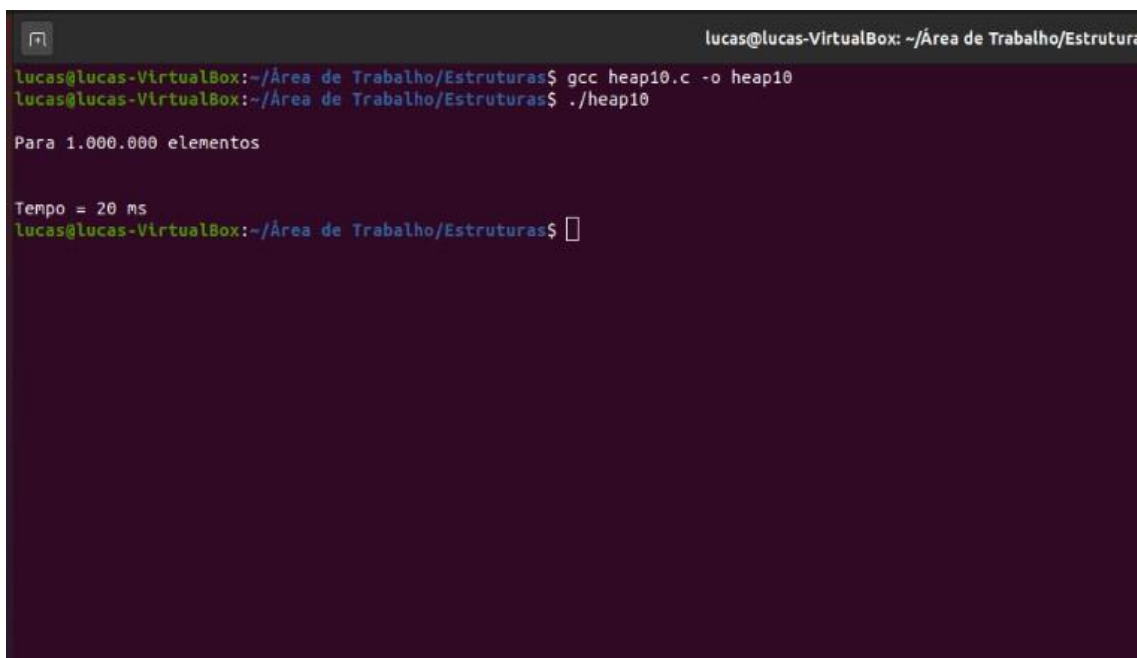
```
lucas@lucas-VirtualBox: ~/Área de Trabalho/Estruturas
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$ gcc heap10.c -o heap10
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$ ./heap10

Para 10 elementos

Vetor gerado aleatoriamente
[0][773287][522649][727606][228882][148770][617318][709351][298578][267101]
[773287][727606][709351][617318][522649][298578][267101][228882][148770][0]

Tempo = Instantaneo
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$
```

Figura 9. Heap Sort com 10 elementos.



```
lucas@lucas-VirtualBox: ~/Área de Trabalho/Estruturas
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$ gcc heap10.c -o heap10
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$ ./heap10

Para 1.000.000 elementos

Tempo = 20 ms
lucas@lucas-VirtualBox:~/Área de Trabalho/Estruturas$
```

Figura 10. Heap Sort com 1 milhão de elementos.

3. CONCLUSÃO

Devido a pandemia não foi possível realizar o projeto presencialmente na universidade. Portanto, além de reforçar e aprofundar o conhecimento aprendido na prática sobre análise de algoritmos e algoritmos de ordenação, o projeto ajudou no desenvolvimento dos alunos acerca de projetos a distância e no aprendizado da individualidade do desenvolvedor, que deve ter uma equipe para desenvolver de forma adequada, mas deve saber se adaptar e buscar conhecimento por si próprio.

Após o desenvolvimento dos algoritmos realizado pela equipe foram feitos testes afim de comprovar a eficácia e qualidade dos mesmos, que mostraram o resultado esperado, sendo assim conseguimos atingir o objetivo esperado e entregar o projeto funcional e sem erros.

	SelectionSort (ms)	InsertionSort (ms)	MergeSort (ms)	QuickSort (ms)	HeapSort (ms)
10	Instantaneo	Instantaneo	Instantaneo	Instantaneo	Instantaneo
100	Instantaneo	Instantaneo	Instantaneo	Instantaneo	Instantaneo
1000	Instantaneo	1	Instantaneo	Instantaneo	Instantaneo
10000	94	173	1	1	Instantaneo
100000	9707	19506	16	14	1
1000000	960331	1829223	191	171	10

Figura 11. Tabela completa com o Tempo para todo algoritmo e número de elementos (N).

Como evidenciado na tabela a cima, os algoritmos não são iguais e tem seus pontos positivos e negativos, O Selection Sort é linear e usado para vetores pequenos e com muitas trocas, Insertion Sort também é linear e rápido para vetores pequenos ou pré-ordenado, além disso ele é estável, para vetores grandes esses algoritmos são muito demorados, portanto para lidar com vetores maiores do que 100.000 elementos foram desenvolvidos o Merge e Quick Sort, sendo o Merge um algoritmo Rápido para vetores grande e que garante Estabilidade. Já o Quick Sort, apesar de não ser estável,

ele ocupa menos espaço em memória e pode ser muito otimizado, garantindo probabilisticamente o resultado mais rápido. O Heap Sort é um algoritmo que insere os dados em uma árvore de forma já ordenada, aproveitando das regras de inserção para evitar comparações e reduzir o tempo de execução.

4. BIBLIOGRAFIA

- [1] <https://www.coursera.org/lecture/algorithms-part1/shuffling-12vcF>
- [2] <http://aorta.coop/wp-content/uploads/2017/05/Divide-and-Conquer-1.pdf>
- [3] <https://www.geeksforgeeks.org/merge-sort/>