Comparação de desempenho sobre o algoritmo Merge sort em vetor e em lista encadeada

Breno Oliveira, Gabriel Rocha, Rafael Greca, Victor Pereira

[1]Instituto de Matemática e Computação – Universidade Federal de Itajubá (UNIFEI)
Caixa Postal 50 – 37.500-903 – Itajubá – MG – Brazil

[2]Department of Mathematics and Computer – Federal University of Itajubá
Itajubá, M.G.

[3]Departamento de Matemática e Computação
Universidade Federal de Itajubá (UNIFEI) – Itajubá, MG – Brazil

brenooliveirareno@yahoo.com.br, gabrielrocha828@gmail.com

rafaelgreca97@hotmail.com, tanabebr@gmail.com

Abstract. *This report aims to compare performance between the merge sort algorithm and the merge sort algorithm using threaded lists. Code and algorithm performance benchmarks have been developed and tested in the NetBeans IDE. Throughout the report, some snippets of the merge sort implementation code will be presented using threaded lists. As part of the work, the code will be tested using specific performance metrics and will be compared to the normal merge performance in order to get the best performance in certain situations.*

Resumo. *Esse relatório tem como objetivo a comparação do desempenho entre o algoritmo de ordenação merge e o algoritmo de ordenação merge utilizando listas encadeadas. O código e as comparações do desempenho dos algoritmos foram desenvolvidos e testados na IDE NetBeans. No decorrer do relatório, serão apresentados alguns trechos do código de implementação do merge sort utilizando listas encadeadas. Como parte do trabalho, o código será testado usando métricas específicas de desempenho e será comparado com o desempenho do merge normal, a fim de obter qual terá o melhor desempenho em determinadas situações.*

1. Introdução

O merge sort é um algoritmo de ordenação logaritimico, implementado de forma recursiva e que utiliza a estratégia de dividir para conquistar. Ele pega um problema grande e o divide em sub-problemas através da recursividade, para então resolver esses sub-problemas. Após ele ter alcançado a solução de todos os sub-problemas, ele juntará as suas resoluções em um conjunto ordenado. Basicamente, ele dividirá um vetor ao meio, ordenará a sua primeira parte recursivamente, depois ordenará a segunda parte recursivamente e, por fim, ele irá intercalar as partes.

Como o algoritmo Merge usa a recursividade e a alocação dinâmica de um vetor auxiliar para ajudar na ordenação, existe um alto custo de memória e acaba se tornando mais custoso para a máquina. O que torna essa ordenação ineficiente em alguns casos, como a ordenação de n números, sendo n um valor muito pequeno.

Por outro lado seu tempo de execução é muito estável em relação a outros metodos de ordenação, considerando melhor, médio e pior caso das entradas de dados, seu resultado final (Em relação ao tempo) vai ser sempre semelhante pois em seu método ele seguirá os mesmo passos independentemente de o vetor estar ordenado ou não.

O algoritmo Merge Sort, utilizando listas encadeadas, usa o mesmo processo que o merge normal: dividir para conquistar, porém a única diferença é a prórpia utilização de listas encadeadas ao envés de vetores. Mesmo assim a sua complexidade assintótica permanece sendo O(n logn) e o algoritmo continua sendo estável.

2. Objetivo

O objetivo do trabalho é avaliar, através de testes práticos, o desempenho do Mergesort trabalhando em um vetor, seu desempenho trabalhando em uma lista encadeada, comparar os resultados de ambos e apresentá-los de maneira gráfica e expositiva.

3. Algoritmo

A funcionalidade do Merge Sort é exemplificada na figura abaixo:

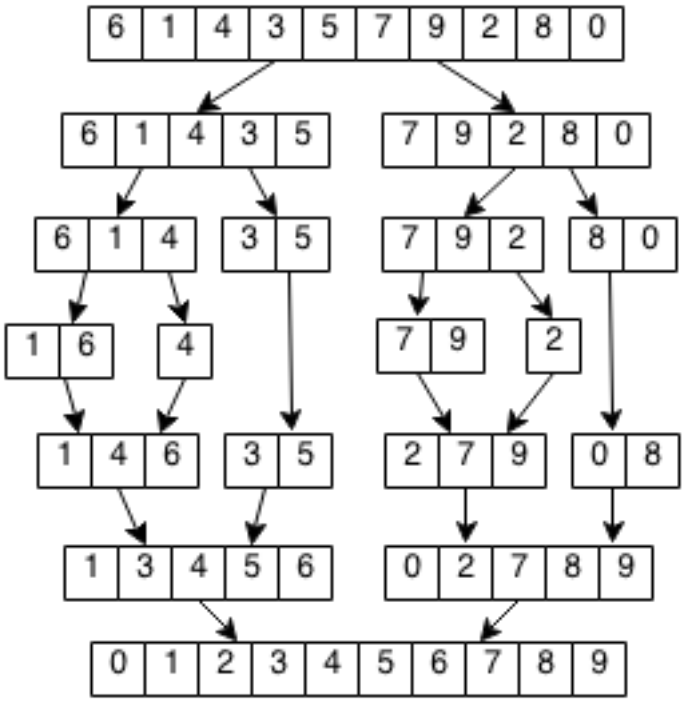

Figura 1. MergeSort

A aplicação do algoritmo em vetor e em lista encadeada segue o mesmo padrão, como o mostrado acima, porém as diferenças estão em sua implementação.

3.1. Merge

A função Merge irá receber um a ”cabeça”da lista, verificar se está vazia e se o ponteiro próximo da cabeça aponta para NULL. Se sim, então quer dizer que não existem mais nós para serem verificados e chama as funções Particiona para particionar a lista. Depois irá chamar a função SortedMerge para ordenar os elementos.

```
void MergeSort(struct Node** headRef){
    struct Node* head = *headRef;
    struct Node* a;
    struct Node* b;

    if ((head == NULL) || (head->prox == NULL)){
        return;}

    particiona(head, &a, &b);

    MergeSort(&a);
    MergeSort(&b);

    *headRef = integra(a, b);
}
```

Figura 2. Merge

3.2. Integra

A função Integra é responsável por intercalar e reconstruir a lista. Essa função possui dois casos bases: confere se o elemento a é individual ou b é um elemento individual. Ela é responsável por intercalar e reconstruir a lista. Toda vez que a função for executada, irá comparar os elementos recursivamente e ordená-los. No final, irá retornar a junção das duas listas.

```
struct Node* integra(struct Node* a, struct Node* b){
    struct Node* result = NULL;

    if (a == NULL)
        return(b);

    else if (b==NULL)
        return(a);

    if (a->valor <= b->valor){
        result = a;
        result->prox = integra(a->prox, b);}

    else{
        result = b;
        result->prox = integra(a, b->prox);}

    return(result);
}
```

Figura 3. Integra - SortedMerge

3.3. Particiona

Essa função é responsável por particionar a lista. A função Particiona irá criar dois nós: o tarta, que apontará para o primeiro nó, e o lebre, que apontará para o próximo a partir do primeiro nó. Enquanto o lebre for diferente de nulo, ele será incrementado por 1 e se o próximo nó não for NULL, ambos os nós (lebre e tarta) andarão um nó. Ao saírem dessa condição, lebre será NULL e o nó tarta estará apontando para o nó do meio da lista. A partir do próximo elemento do meio, será feita a partição da lista. O trecho do código responsável pela partição das listas é onde o ponteiro prox do tarta recebe NULL. Assim, particionando a lista em duas sub-divisões: esquerda e direita. Frente irá apontar para o

primeiro elemento da sub-divisão esquerda e Pos irá apontar para o primeiro elemento da sub-divisão direita. Esse processo de divisão irá se repetir até que sobre apenas dois elementos, fazendo a partição do lado esquerdo primeiro e depois do lado direito. Logo em seguida irá chamar a função Integra para intercalar e reconstruir a lista.

```
void particiona(struct Node* no,struct Node** frente, struct Node** pos){
        struct Node* lebre;
        struct Node* tarta;
        tarta = no;
        lebre = no->prox;

        while (lebre != NULL){
        lebre = lebre->prox;
        if (lebre != NULL){
                tarta = tarta->prox;
                lebre = lebre->prox;}}

        *frente = no;
        *pos = tarta->prox;
        tarta->prox = NULL;
}
```

Figura 4. Particiona - FrontBackSplit

Os códigos descritos, foram baseados no modelo apresentado por Quinston Pimenta [3].

4. Prática

A metodologia prática consistiu em 5 seeds de testes, onde cada uma delas recebeu diversas execuções para cada entrada de dados.

Durante a execução foram considerados entradas de dados de tamanhos 10000, 15000, 20000, 25000 e 30000 elementos aleatórios, além de os mesmos valores para uma entrada totalmente ordenada e uma ordenada de maneira decrescente para cada uma das seeds. Os testes foram executados seguindo as seguintes regras:

A memória da IDE foi limpa a cada execução de uma entrada de dados, a máquina utilizada não estava executando nenhum aplicativo durante os testes, apenas da IDE, afim de evitar que os resultados fossem afetados por fatores externos.

4.1. Equipamento Utilizado

IDE utilizada:
NetBeans 8.2
Máquina Utilizada:
Processador: Intel(R) core(TM) i5-5200U @ 2.20GHz 2.20 GHz
RAM: 6,00GB

5. Resultados

Apresenta-se abaixo os resultados dos testes práticos sobre o desempenho do MergeSort em tempo de execução, numero de comparações e numero de movimentações.

5.1. Tabelas e Gráficos

As tabelas apresentadas abaixo mostram, para cada seed de testes, o tempo de execução dos algoritmos, dado em segundos, o numero de comparações e o numero de movimentações (valores em média).

Os gráficos apresentam apenas a relação entre o desempenho do Mergesort em vetor em comparação ao Mergesorte em lista em tempo de execução, onde houve uma real diferença em valores, baseando-se no fato de que os valores de comparações e movimentações são muito parecidos em ambas as aplicações.

Merge Vetor	Tempo (Segundos)	Comparações	Movimentações
10000	0,0108	120489	254105
15000	0,0160	189333	397949
20000	0,0224	260902	548134
25000	0,0277	334036	701268
30000	0,0352	408769	856001

Merge Lista	Tempo (Segundos)	Comparações	Movimentações
10000	0,0064	120401	244018
15000	0,0089	189347	382964
20000	0,0115	260989	528222
25000	0,0179	334164	676397
30000	0,0204	408718	825951

Figura 5. Tabela - Seed 1

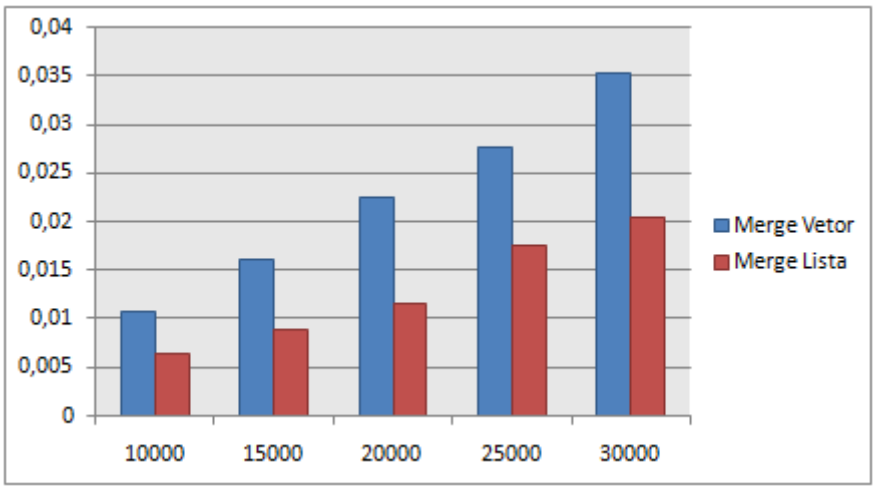

Figura 6. Gráfico de Tempo - Seed 1

Merge Vetor	Tempo (Segundos)	Comparações	Movimentações
10000	0,0110	120406	254022
15000	0,0162	189165	397781
20000	0,0231	260964	548196
25000	0,0275	333904	701136
30000	0,0357	408571	855803

Merge Lista	Tempo (Segundos)	Comparações	Movimentações
10000	0,0052	120424	244041
15000	0,0090	189181	382798
20000	0,0131	260877	528110
25000	0,0171	334184	676417
30000	0,0223	408621	825854

Figura 7. Tabela - Seed 2

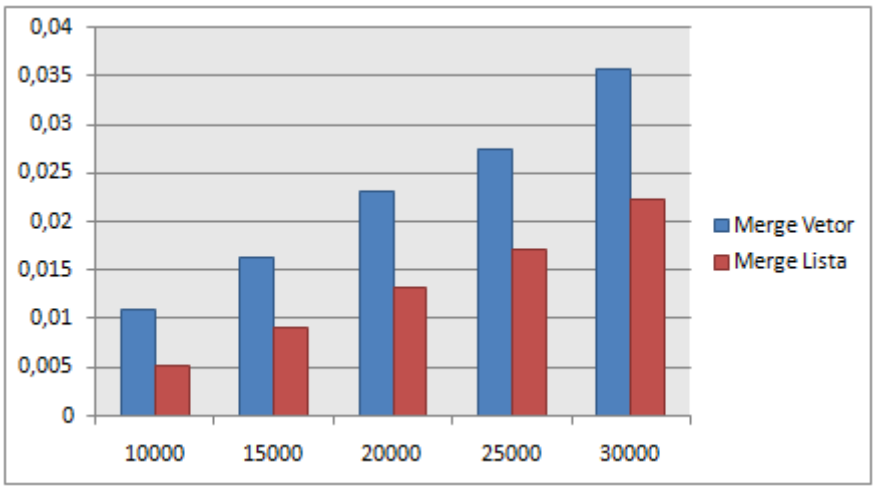

Figura 8. Gráfico de Tempo - Seed 2

Merge Vetor	Tempo (Segundos)	Comparações	Movimentações
10000	0,0097	120393	254009
15000	0,0132	189099	397715
20000	0,0194	260929	548161
25000	0,0253	334175	701407
30000	0,0335	408530	855762

Merge Lista	Tempo (Segundos)	Comparações	Movimentações
10000	0,0065	120422	244039
15000	0,0096	189253	382870
20000	0,0131	260712	527945
25000	0,0174	334111	676344
30000	0,0235	408679	825912

Figura 9. Tabela - Seed 3

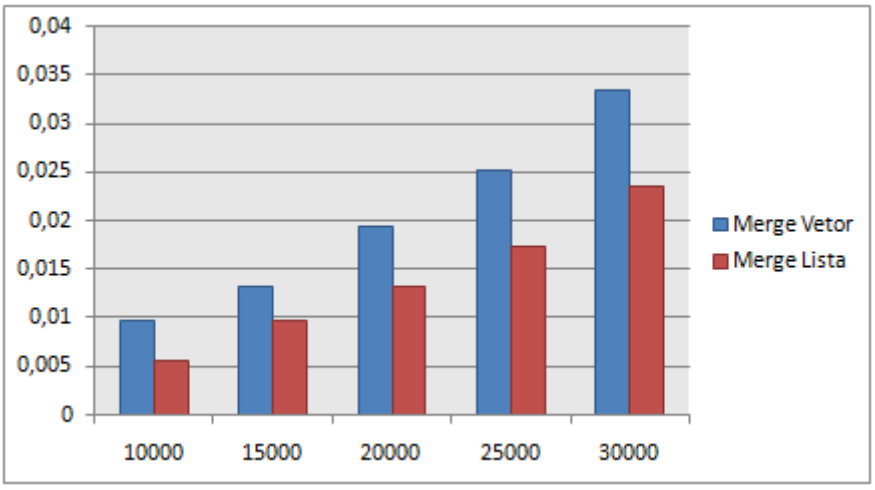

Figura 10. Gráfico de Tempo - Seed 3

Merge Vetor	Tempo (Segundos)	Comparações	Movimentações
10000	0,01	120447	254063
15000	0,0148	189346	997962
20000	0,0221	260856	548088
25000	0,0287	334142	701374
30000	0,0344	408575	855807

Merge Lista	Tempo (Segundos)	Comparações	Movimentações
10000	0,0053	120605	244222
15000	0,0087	189299	382916
20000	0,0121	260932	528165
25000	0,0165	334053	676286
30000	0,0206	408687	825920

Figura 11. Tabela - Seed 4

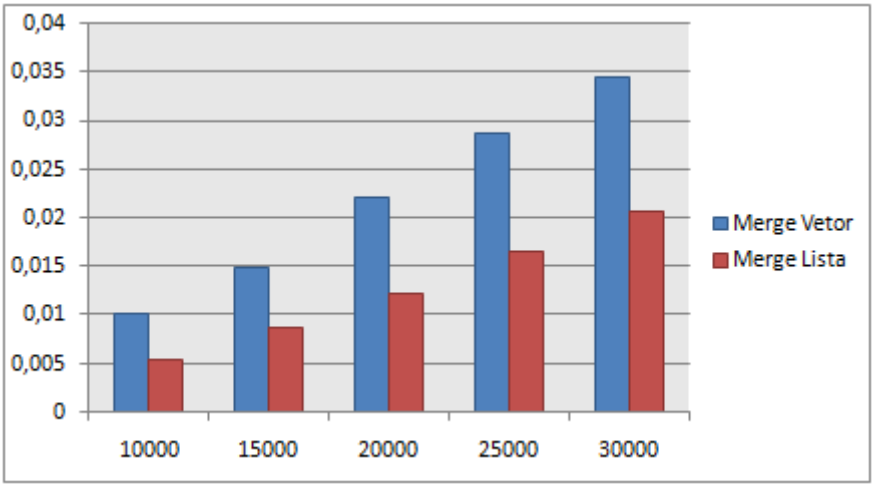

Figura 12. Gráfico de Tempo - Seed 4

Merge Vetor	Tempo (Segundos)	Comparações	Movimentações
10000	0,0099	120409	254025
15000	0,0150	189363	397979
20000	0,0222	260991	548233
25000	0,0282	333972	701255
30000	0,0328	408604	855836

Merge Lista	Tempo (Segundos)	Comparações	Movimentações
10000	0,0051	120494	244111
15000	0,0088	189387	383004
20000	0,0111	260941	528174
25000	0,0165	334074	676307
30000	0,0207	408457	825690

Figura 13. Tabela - Seed 5

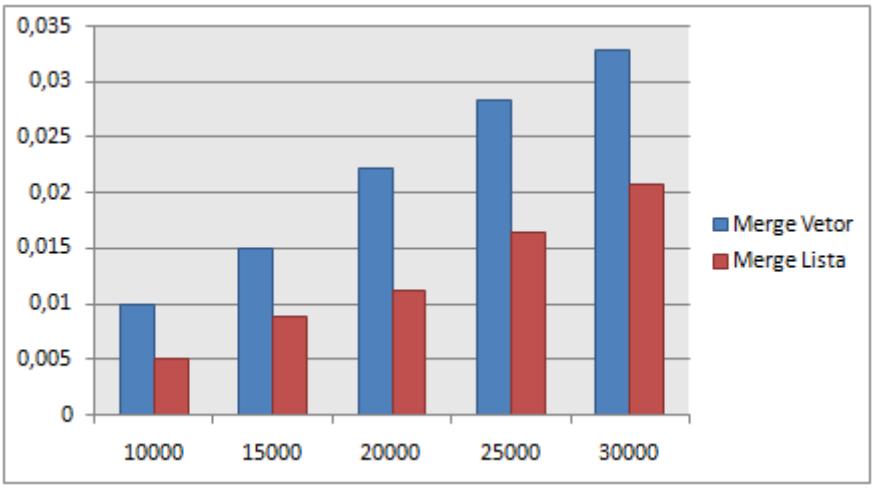

Figura 14. Gráfico de Tempo - Seed 5

Merge Vetor	Tempo (Segundos)	Comparações	Movimentações
10000	0,0098	69008	202624
15000	0,0145	106364	314980
20000	0,0199	148016	435248
25000	0,0235	188476	555708
30000	0,0285	227728	674960

Merge Lista	Tempo (Segundos)	Comparações	Movimentações
10000	0,0025	69008	192625
15000	0,0046	106364	299981
20000	0,0074	148016	415249
25000	0,0086	188476	530709
30000	0,0108	227728	644961

Figura 15. Tabela - Entrada Crescente

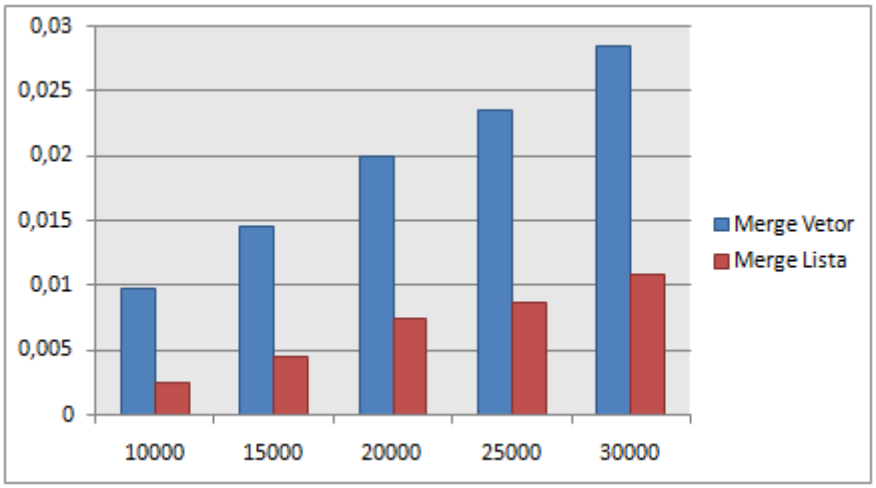

Figura 16. Gráfico de Tempo - Entrada Crescente

Merge Vetor	Tempo (Segundos)	Comparações	Movimentações
10000	0,0096	69008	202624
15000	0,0143	106364	314980
20000	0,0188	148016	435248
25000	0,0236	188476	555708
30000	0,0295	227728	674960

Merge Lista	Tempo (Segundos)	Comparações	Movimentações
10000	0,0028	64608	188225
15000	0,0055	102252	295869
20000	0,0076	139216	406449
25000	0,0082	178756	520989
30000	0,0104	219504	636737

Figura 17. Tabela - Entrada Decrescente

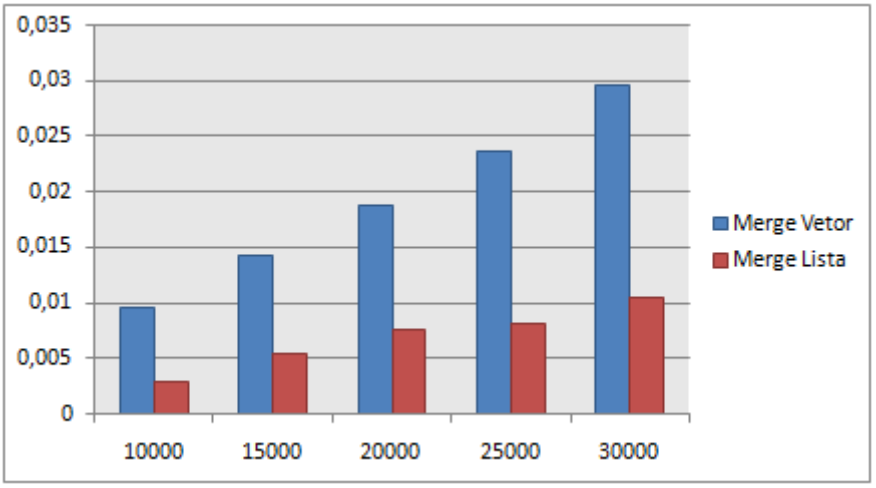

Figura 18. Gráfico de Tempo - Entrada Decrescente

Os quesitos de comparação e movimentação, são muito parecidos em ambos os dois algoritmos, pois eles se baseam no mesmo método de ordenação, apenas mudando a aplicação devido a natureza das entradas de dados.

6. Conclusão

Baseado nos dados obtidos nos testes, vê-se que ambos os algoritmos possuem o desempenho muito parecido em comparações e movimentações, porém o Merge sort em lista encadeada possui uma considerável vantagem no que condiz ao tempo de execução, onde as comparações e movimentações aumentam à medida que o valor de N (quantidade de números de entradas) cresce.

Essa diferença se dá em grande parte pelo fato do Merge sort em vetor ser sempre obrigado a alocar memória dinamicamente para um vetor auxiliar em todas as ordenações, enquanto o Mergesorte em lista, apesar de movimentar uma estrutura de dados dinâmica, não precisa alocar novos espaços de memória para fazer sua ordenação.

É importante avaliar também que no caso Merge sort em lista é preferível usar um algoritmo que insira os valores em uma lista de maneira já ordenada, pois o uso de um método de ordenação pode ser muito custoso de se implementar, e pode também significar uma perda de desempenho de uma aplicação.

Conclui-se então que o uso destes métodos de ordenação é situacional e necessita do bom-senso do programador ou do grupo que trabalha na aplicação para definir qual é o melhor algoritimo para resolução do seu problema.

Referências

[1] GEEKIES FOR GEEKIES. Merge Sort for Linked Lists. Disponível em: https://www.geeksforgeeks.org/merge-sort-for-linked-list/. Acesso em: 16/04/2018.

[2] Ghellere, G.V; Borges, L.L.S; Mendonça, P.S. Análise e comparação de algoritmos de ordenação em listas encadeadas. Disponível em: https://github.com/GabsGear/Algoritmos-de-ordenacao-de-listas-encadeadasem-c. Acesso em: 28/04/2018.

[3] Quinston Pimenta, MergeSortForLikedList. Disponível em: https://www.dropbox.com/s/1q79706gbg9ou48/MergeSortForLinkedList.txt?dl=0. Acesso em: 25/04/2018.

[4] Sedgewick, R; Wayne, K. Algorithms Fourth Edition; Addison-Wesley, 2011.

[5] Shene, Ching-Kuang. A Comparative Study of Linked List Sorting Algorithms. Disponível em: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.31.9981rep=rep1type=pdf. Acesso em: 23/04/2018.

[6] TECHIE DELIGHT. Merge Sort Algorithm for Singly Linked List. Disponível em: http://www.techiedelight.com/merge-sort-singly-linked-list/. Acesso em: 25/04/2018.