UNIVERSIDADE FEDERAL DE SANTA CATARINA
Estrutura de Dados

Gabriel Velho Ghellere
Luan Lorenzo dos Santos Borges
Pedro Dias Mendonça

EXERCÍCIO PROGRAMA: Análise e comparação de algoritmos de ordenação em listas encadeadas.

Araranguá, 2016

RESUMO

Esse relatório tem como objetivo descrever, e analisar os mais conhecidos algoritmos de ordenação de dados, são eles: Ordenação por inserção, seleção, mergesort, heapsort e quicksort. Para a comparação foi utilizada a estrutura de dados Lista Encadeada. Para realização do relatório foram realizados testes práticos com os algoritmos com listas de diferentes instancias para que além da maior confiabilidade dos resultados seja possível a escolha de um algoritmo para um caso especifico verificando qual se comporta melhor em cada situação.

Palavras-chave: Estrudura de Dados, Ordenação, *Sort*.

1 - INTRODUÇÃO

Algoritmos de ordenação como o próprio nome sugere tem o dever de ordenar dados (sejam números, letras e afins) em uma certa ordem definida para facilitar sua manipulação aumentando significadamente a eficiência de programas vinculados com os esses dados. O estudo da ordenação de dados é de extrema importância na área da computação pois é utilizado em simples aplicações até as mais complexas. Existem inúmeros algoritmos, alguns simples como de Inserção e Seleção e outros mais sofisticados como Mergesort, Heapsort, Quicksort, Shellsort que podem usar ferramentas como a recursividade para melhor desempenho.

2 - DESENVOLVIMENTO

O exercício consiste na aplicação de ordenação de dados crescente na estrutura de dados Listas Encadeadas, para tanto foram utilizados algoritmos já conhecidos.

- **Algoritmo de Inserção**, muito popular sua metodologia se assemelha a organização de cartas de um baralho. Seu consumo de tempo é proporcional ao numero de comparações que são realizadas "*v[i] > x*". Se considerarmos uma instância [A..n] tempo que seu tempo de execução no pior caso é proporcional a n^2 unidades de tempo*. (FEOFILOFF, 2009).*
- **Algoritmo de Seleção,** segue a ideia da escolha do menor elemento do vetor, segundo menor assim por diante rearranjando-os ordenadamente. Seu consumo de tempo se assemelha ao algoritmo de inserção, já que ambos precisam realizar o mesmo número de comparações. Em seu pior caso consome n^2 unidades de tempo. *(FEOFILOFF, 2009).*
- **Mergesort,** baseado na estratégia "dividir para conquistar" o MergeSort divide a estrutura a ser ordenada em duas assim é resolvido um problema auxiliar com instancia menor. Após ordenadas as duas metades são intercaladas. Sobre seu desempenho nota-se que a função encarregada de intercalar a estrutura consome um tempo proporcional ao tamanho da estrutura em seu pior caso, garantindo assim a eficiência do algoritmo consumindo um tempo proporcional a *(n log2 n)* apresentando uma maior eficiência se comparados aos algoritmos de inserção e seleção. *(FEOFILOFF, 2009).*
- **Heapsort,** esse algoritmo utiliza uma estrutura de dados chamada max-heap. Sendo um vetor [1..*m*] para ser considerado max-heap deve seguir a seguinte propriedade *v[[1/2f]] => v[f].* O algoritmo necessita de pelo menos duas funções auxiliares para sua execução uma para sua inserção e outra para transformá-lo em heap. O algoritmo é realizado em duas etapas, a primeira para transformar o vetor em max-heap e a segunda para sua ordenação *"maxheapficando"*. Seu tempo de execução depende da função de inserção e *maxheapficação* onde a inserção consome no maximo *log2(m+1)* e a função *maxheapfica* no máximo *log2m*, logo o algoritmo consome no máximo *n log2n. (FEOFILOFF 2009).*
- **Quicksort,** utilizando a estratégia de "dividir e conquistar" o algoritmo quicksort depende de uma função que divida a estrutura definindo um elemento pivô, esse pivô definirá os maiores e menores elementos da estrutura separando os menores a esquerda e maiores a direita. Por se tratar de um algoritmo recursivo a divisão segue até cair na base da recursão, assim ordenando a estrutura. Seu tempo de execução depende exclusivamente da função que divide, se a função dividir o a estrutura próximo a metade sua velocidade será *n log2n. (FEOFILOFF, 2009).*

2.1 OBJETIVOS GERAIS

Os objetivos da pesquisa além da implementação dos algoritmos vistos em sala na estrutura de dados listas encadeadas é a análise dos algoritmos em questão de tempo de execução, e espaço gasto por cada um.

2.2 METODOLOGIA

Como o desempenho de cada algoritmo depende também do processamento, para resultados mais confiáveis os testes foram realizados em no mesmo computador, com a linguagem de programação C usando as bibliotecas stdio.h, stdlib.h e time.h e o compilador GCC. O tempo foi marcado com 7 casas decimais com uma função da biblioteca time.h e o espaço consumido pelas variáveis com a função sizeof.

O computador onde foram realizados os testes possui a seguinte configuração:

- Processador: *Intel(R) Core(TM) i5-4200U CPU @ 1.60GHz;*
- Memória: 3979 MiB; com frequência de 1333Mhz
- Ubuntu 16.04 LTS

Os testes foram realizados sobre as seguintes condições:

- Listas encadeadas geradas aleatoriamente.
- Media de tempo é a média de 50 iterações.
- A quantidade de elementos da lista é 100, 200, 400, 800, 1600, 32000, 64000, 128000, 256000, 512000, 1024000, 2048000, 4096000.

2.2.1 DESCRIÇÃO DOS ALGORITMOS IMPLEMENTADOS.

- **Ordenação por Inserção:** A lista utilizada para ordenação por inserção foi a Lista duplamente encadeada com *cabeça* e *rabo* a função InserirLista se encarrega de inserir os valores aleatórios na lista, a função **OrdenacaoInsercao** ordena a lista.
  - Função **OrdenacaoInsercao:** A função recebe o endereço da cabeça da lista, uma célula auxiliar é inicializada como *p→seg→seg* e enquanto *p→seg* for diferente de nulo (fim da lista) um novo ponteiro auxiliar *q* é declarado como *p→ant, o* conteudo de *p* é guardado em uma variável auxiliar *x* para comparação posterior. Ainda dentro do while se *q→conteudo* for maior que x um novo *while* é iniciado declarando *q→seg→conteudo* como *q→conteudo* e *q* retorna como *q→ant.* Fora do segundo while *q→seg→conteudo* recebe *q→conteudo* e *p* incrementa como *p→seg*. Resumindo, a função compara elemento por elemento da lista se o elemento comparado for menor que seu anterior ele segue trocando sua posição com o anterior até encontrar um elemento igual ou menor, quando o ponteiro *p* chegar ao fim da lista ela estará ordenada.
- **Ordenação por Seleção** A lista utilizada para ordenação por inserção foi a Lista duplamente encadeada com *cabeça* e *rabo* a função InserirLista se encarrega de inserir os valores aleatórios na lista, a função **OrdenacaoSelecao** ordena a lista.
  - Função **OrdenacaoSelecao** Esta função recebe a cabeça da lista e cria uma célula auxiliar com a célula seguinte *p = lst→seg.* Um primeiro laço

de repetição *while* percorre a lista encontrando o valor mínimo guardando o elemento seguinte na variável auxiliar *q*, quando esse valor é encontrado ele é trocado de lugar com o conteúdo de da célula auxiliar *p* repetindo o processo até que a lista esteja ordenada.

- **Mergesort:** Para o algoritmo mergesort a lista encadeada escolhida foi Lista duplamente encadeada, os elementos aleatórios da lista são inseridos na função InserirLista que recebe o endereço da lista e o elemento a ser inserido. Após a inserção ocorre a chamada da função **Mergesort**.
  - Função **Mergesort,** uma função recursiva que recebe o endereço da lista, sua base é a lista com um único elemento. A lista é dividida pela função divide, assim listas são ordenadas e intercaladas.
  - Função **divide** recebe o endereço da lista e devolve a uma parte da lista após o meio. A lógica da função consiste na utilização de dois ponteiros, enquanto um anda duas células a cada iteração outro anda somente uma, assim quando o ponteiro chegar ao fim da lista o outro estará na metade.
  - Função **Intercala** essa função recebe duas listas encadeadas ordenadas e retorna uma ordenada, é uma função recursiva com duas bases pois o primeiro elemento de cada lista irá determinar qual das duas listas será retornada após a intercalação (foi usado esse recurso para que não seja necessário a utilização de um vetor ou lista auxiliar (exercício 9.1.7 Algoritmos em C, Feofiloff Paulo, pg 69 aplicado em listas encadeadas)) .
- **QuickSort:** A lista escolhida para o quicksort foi a lista duplamente encadeada com rabo e cabeça, sua inserção é feita na função inserirlista, A função **Quicksort** é uma função de ordenação recursiva e depende da função **separa.**
  - Função **Separa** essa função é aonde a magica do quicksort realmente ocorre, ela encontra o valor médio da lista e divide com os valores menores a esquerda e maiores a direita.
  - Função **Quicksort** se trata de uma função recursiva que divide a lista consecutivamente com a função separa até que ela esteja ordenada.

2.3 RESULTADOS E CONCLUSÃO

Após a realização dos experimentos foram geradas as tabelas de análise dos algoritmos e gráfico de instância x tempo. De onde podemos tirar as algumas conclusões.

Sobre a velocidade dos algoritmos podemos chegar a conclusão que o algoritmo de **ordenação por inserção** se destaca na ordenação das menores listas (até 100 elementos) mais seu tempo de execução aumenta significativamente com o aumento de tamanho da instância. O algoritmo de **ordenação por seleção** segue o mesmo padrão do algoritmo de inserção porém apresentou pior desempenho em todas as instâncias.

O **mergesort e o quicksort** apresentaram um ótimo desempenho nos testes principalmente se comparados a inserção e seleção. Observa-se que o mergesort apresenta melhor velocidade com o tamanho da instância pois apresenta maior constante de proporcionalidade que os demais, ou seja o mergesort se torna mais indicado para maiores listas.

	Análise dos algoritmos.			
Instancia	**Inserção**		**Seleção**	
	Tempo(s)	**Espaço (bytes)**	**Tempo(s)**	**Espaço (bytes)**
100	0.0000008	2404	0.0000104	2400
200	0.0000300	4804	0.0000510	4800
400	0.0001120	9604	0.0001681	9600
800	0.0004430	19204	0.0006640	19200
1600	0.0018130	38404	0.0030680	38400
3200	0.0074980	76804	0.0133930	76800
6400	0.0298590	153604	0.0523630	153600
256000	87.6355373	6144004	148.7023970	6144000
512000	543.4001350	12288004	643.7355790	12288000
1024000	2180.5810290	24576004	2852.7082540	24576000
2048000	8940.3821230*	49152004	12266.6544922*	49152000
4096000	36655.5564063*	98304004	49066.5819688*	98304000

*Tabela 1: Análise dos algoritmos de inserção e seleção (valores com * são estimados por proporcionalidade).*

	Análise dos algoritmos.			
Instancia	**MergeSort**		**QuickSort**	
	Tempo(s)	**Espaço (bytes)**	**Tempo(s)**	**Espaço (bytes)**
100	0.0000130	2400	0.0000070	2944
200	0.0000270	4800	0.0000150	5864
400	0.0000570	9600	0.0000310	11784
800	0.0001270	19200	0.0000710	23560
1600	0.0002730	38400	0.0001570	46872
3200	0.0006430	76800	0.0003460	93960
6400	0.0012830	153600	0.0007640	187992
256000	0.1252940	6144000	0.0392120	8112008
512000	0.3131650	12288000	0.1010350	16304008
1024000	0.6889630*	24576000	0.2914020	32688008
2048000	1,5157186*	49152000	0.9115490	65456008
4096000	3,3345809*	98304000	3.2655140	130992008

*Tabela 2: Análise dos algoritmos MergeSort e QuickSort (valores com * são estimados por proporcionalidade).*

Sobre o espaço gasto os algoritmos mergesort, ordenação por inserção e seleção apresentaram valores próximos, equivalentes ao tamanho da instância. O espaço a mais alocado pelo **quicksort** é compensado em velocidade, pois é o algoritmo que apresenta melhor desempenho nas instâncias testadas.

Seguem os gráficos Instância x tempo;

Gráfico 1 Ordenação por inserção.

Gráfico 2: Ordenação por Seleção

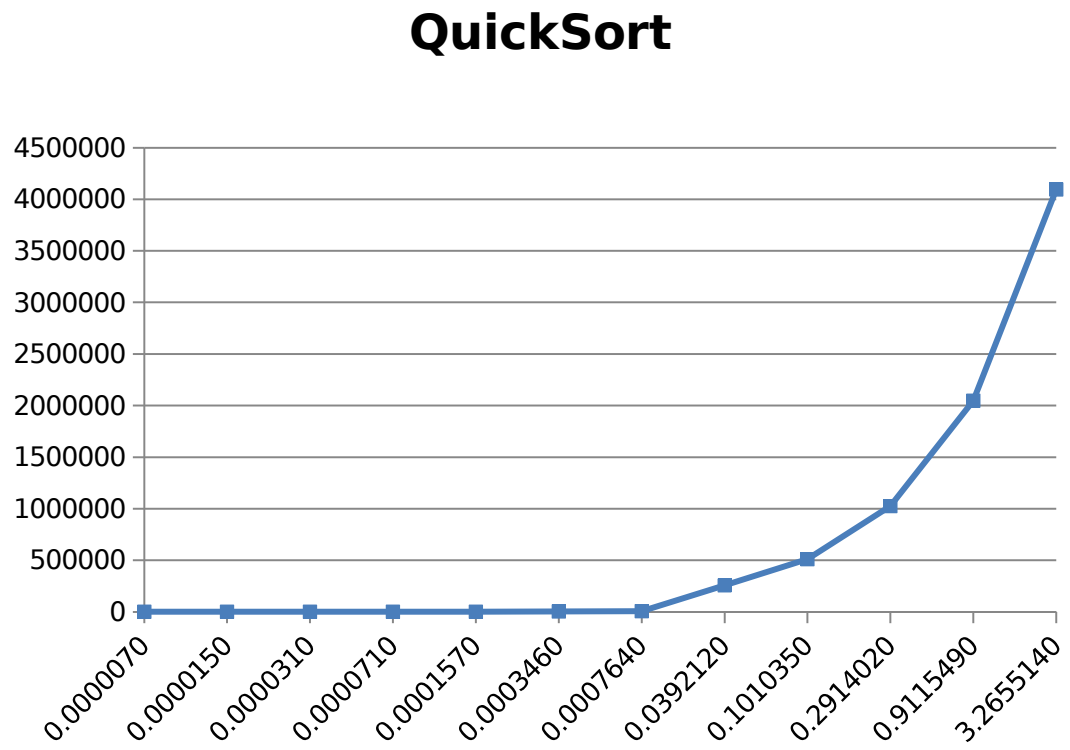

Gráfico 3: QuickSort

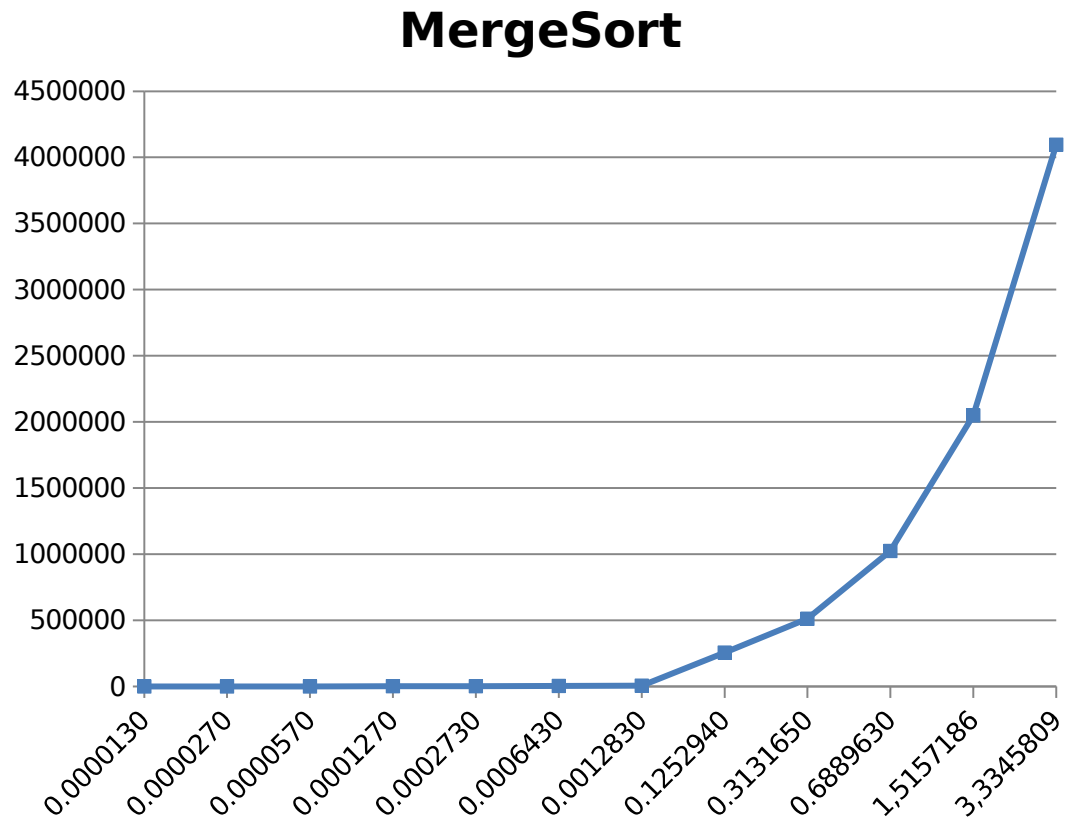

Gráfico 4 MergeSort

REFERÊNCIAS BIBLIOGRÁFICAS

FEOFILOFF, Paulo 1946- Algoritmos / Paulo feofiloff – Rio de Janeiro : Elsevier 2009 il.