UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

LISTA DE CLIENTES

Pedro Henrique Woiciechovski

25/10/2017

1.Introdução:

As listas acabam por ser uma das formas mais básicas e importantes de se conhecer para conceitos mais básicos de programação, nesse trabalho deveríamos ser capazes de implementar as listas com base em arquivos de texto(.txt) disponibilizados a nós pelo professor.

A parte inicial e primordial foi compreender a diferença entre as listas *sequenciais* e as listas *encadeadas*. As sequenciais podem ser facilmente comparadas a um vetor simples onde se é simples de fazer alterações como adicionar um elemento uma vez que tudo que se precisa é achar a posição desejada e lá adicionar o elemento e o mesmo se refere a remoção, achar a posição numérica dele então remove-lo para então arrumar a lista de forma a não existir buracos.

Já nas encadeadas o funcionamento é diferente uma vez que os elementos podem estar em qualquer local da memória e eles acabam por ter a referência para o próximo elemento da lista e aquele que tem referência para o próximo elemento nulo sendo considerado o último ao mesmo tempo que o primeiro deve ser indicado de maneira externa. Isso tem impĺicações em um cuidado extra que deve ocorrer para arrumar constantemente as referências dos elementos pois uma indicação de próximo elemento errado pode acabar por fazer com que toda lista fique embaralhada.

Nesse projeto também nos foi solicitados que o programa gerado fosse capaz de ler os arquivos com *RG* e *Nome* dos supostos clientes e adicioná-los a um dos tipos de lista que seria escolhido no momento em que o programa se inicia. O programa também deveria ser capaz de adicionar elementos, seja na primeira, última ou qualquer outra posição que fosse de escolha do usuário, assim como de remover registros em todas as posições seja ela a primeira, a última ou alguma do meio. Por fim todas as funções deveria ser capazes de apresentar *número de comparações*, *número de cópias/movimentações* que realizou e seu *tempo de execução*.

2.Funções do Sistema:

Em seguida será apresentado as escolhas tomadas para a lógica das funções que variam em certos pontos e se igualam em outros por escolhas realizadas.

2.1 PREENCHIMENTO

O primeiro desafio foi definir uma forma de preencher as listas, afinal isso decide como será trabalhar com aquela lista e inicialmente se usou o vetor simples com tamanho pré definido para a lista sequencial e arquivos com referências para o próximo elemento assim como para anterior na encadeada, como mostra a *figura 1* e a *figura 2*.

```
inicio

    estrutura lista[MAX]
    inteiro numeroElemento, parte2
    carácter parte1[30]

    enquanto(linha diferente de nulo)
        linha<-ler(Arquivo)

        parte1 <- separa(linha, onde tem ‘,’)
        parte2 <- separa(linha, onde tem ‘,’)

        lista[numeroElemento].Nome = parte1
        lista[numeroElemento].RG = parte2

        numeroElemento = numeroElemento+1

    fimEnquanto
fim
```

Figura 1: Pseudocódigo da adição da lista sequencial

```
inicio

    cabeçalho lista
    inteiro numeroElemento, parte2
    carácter parte1[30]
    estrutura *registro

    enquanto(linha diferente de nulo)
        linha<-ler(Arquivo)

        registro<-alocar memória(tamanho de estrutura)

       se(numeroElemento = 0)
            parte1 <- separa(linha, onde tem ‘,’)
            parte2 <- separa(linha, onde tem ‘,’)

            registro.Nome <- parte1
            registro.RG <- parte2
```

```
                registro.próximo <- Nulo
                registro.anterior <- Nulo

                lista.primeiro <- registro
                lista.ultimo <- registro

                numeroElemento = numeroElemento+1

            se não
                parte1 <- separa(linha, onde tem ‘,’)
                parte2 <- separa(linha, onde tem ‘,’)

                registro.Nome <- parte1
                registro.RG <- parte2

                registro.próximo <- Nulo
                registro.anterior <-lista.ultimo

                lista.ultimo <- registro

                numeroElemento = numeroElemento+1

        fimEnquanto
fim
```

Figura 2: Pseudocódigo da adição da lista encadeada

Porém, como será mostrado mais a frente, por questão dos métodos de organização ambos foram alterados para funcionar com os métodos de organização e as maiores listas. A ideia foi gerar um vetor de ponteiros com as referências de para cada respectivo elemento como mostra a figura 3.

```
inicio

        cabeçalho lista
        inteiro numeroElemento, parte2
        carácter parte1[30]
        estrutura *registro, **vetorPonteiros

        vetorPonteiros<-aloca memória(tamanho de estrutura * tamanhoLista)

        enquanto(linha diferente de nulo)

        linha<-ler(Arquivo)

                registro<-alocar memória(tamanho de estrutura)
```

```
            se(numeroElemento = 0)

                    parte1 <- separa(linha, onde tem ‘,’)
                    parte2 <- separa(linha, onde tem ‘,’)

                    registro.Nome <- parte1
                    registro.RG <- parte2

                    registro.próximo <- Nulo
                    registro.anterior <- Nulo

                    lista.primeiro <- registro
                    lista.ultimo <- registro

                    vetorPonteiros[0] = registro

                    numeroElemento = numeroElemento+1

            se não
                    parte1 <- separa(linha, onde tem ‘,’)
                    parte2 <- separa(linha, onde tem ‘,’)

                    registro.Nome <- parte1
                    registro.RG <- parte2

                    registro.próximo <- Nulo
                    registro.anterior <-lista.ultimo

                    lista.ultimo <- registro

                    vetorPonteiros[numeroElemento] = registro

                    numeroElemento = numeroElemento+1
        fimEnquanto
fim
```

Figura 3: Pseudocódigo da adição da listas na sua forma final

Embora a forma de ambas fique igual ainda existe uma forma diferente de trabalhar com elas uma vez que suas estrutura são diferentes.

2.2 ADIÇÃO NO FINAL

Por questões da forma como foi escolhida trabalhar com a geração das listas(o vetor de ponteiros) todas as adições assim como a remoção ficaram iguais para ambos apenas mudando a estrutura. A forma usada foi gerar um novo vetor de ponteiros com uma casa de diferença do original, apagar o original e assim passar a referência do novo como retorno para o vetor que chamou o original como mostra a *figura 4*.

```
inicio
        Estrutura *dado, **vetorNovo
        inteiro contador

        vetorNovo<-aloca memória(tamanho de estrutura * (tamanhoLista+1))

        dado = aloca memória(tamanho de estrutura)

        vetorNovo[tamanhoLista]<-registro

        para(contador<-0 ; contador<tamnhoLista ; contador<-contador+1)

                vetorNovo[contador]<-vetorOriginal[contador]

        fim para

        desaloca(vetorOriginal)

        numeroElemento = numeroElemento +1

        retorne vetorNovo

fim
```

Figura 4: Pseudocódigo da adição na posição final

Embora isso traga algumas vantagens existe uma questão muito importante na adição e remoção de registros na lista encadeada que também ocorre na organização da lista que será debatida mais tarde.

2.3 ADIÇÃO EM POSIÇÃO ESCOLHIDA

Nessa adição existe uma diferença na lógica da anterior. Ela também cria o vetor auxiliar e retorna ele, mas a forma de marcar a posição correta é diferente, uma vez que ela pode ser qualquer uma da lista inteira. Por isso foi adicionado uma comparação para ver se a posição é a correta e caso seja adicionar o elemento nela, não copiar o elemento do vetor original. Isso resulta em comparações que tem como objetivo ver qual a posição atual e qual elemento do vetor original deve ser adicionado naquela posição como foi apresentado na *figura 5*.

```
inicio
        Estrutura *dado, **vetorNovo
        inteiro contador

        vetorNovo<-aloca memória(tamanho de estrutura * (tamanhoLista+1))
```

```
        dado = aloca memória(tamanho de estrutura)

        para(contador<-0 ; contador<tamnhoLista ; contador<-contador+1)

                se(contador < posiçãoNovo)
                        vetorNovo[contador]<-vetorOriginal[contador]

                se não se(contador = posiçãoNovo)
                        vetorNovo[contador]<-registro

                se não
                        vetorNovo[contador]<-vetorOriginal[contador-1]
        fim para

        desaloca(vetorOriginal)

        numeroElemento = numeroElemento +1

        retorne vetorNovo

fim
```

Figura 5: Pseudocódigo da adição em posição qualquer

Veja que na figura os elementos do vetor original são adicionados de forma a verificar se já se passou da posição desejada para saber se deve ser adicionado o registro gerado ou copiar o elemento do vetor original.

2.4 ADIÇÃO NO INÍCIO

Na adição no início a lógica usada foi extremamente próxima à usada na adição no final, porém no lugar de copiar o vetor original nas mesmas posições ele adiciona todos uma posição a mais do que a sua original dando espaço para que o registro seja o elemento no ponto inicial. Essa lógica foi representada na *figura 6*.

```
inicio
        Estrutura *dado, **vetorNovo
        inteiro contador

        vetorNovo<-aloca memória(tamanho de estrutura * (tamanhoLista+1))

        dado = aloca memória(tamanho de estrutura)

        vetorNovo[0]<-registro

        para(contador<-1 ; contador<tamnhoLista-1 ; contador<-contador+1)
```

```
            vetorNovo[contador]<-vetorOriginal[contador-1]

        fim para

        desaloca(vetorOriginal)

        numeroElemento = numeroElemento +1

        retorne vetorNovo

fim
```

Figura 6: Pseudocódigo da adição no início

2.4 REMOVER ELEMENTO

Para ser capaz de fazer a remoção foi utilizado uma lógica próxima da usada na adição em posição qualquer, assim seria capaz de remover qualquer elemento sem se preocupar com posição dele e poder fazer a remoção. Outra coisa diferente é que o vetor auxiliar gerado possui uma posição a menos e não a mais, isso implica em um controle de quais elementos devem ir para o novo vetor e qual vai ficar.Isso é apresentado na *figura 7*.

```
inicio

        para(contador<-0 ; contador<tamnhoLista ; contador<-contador+1)

                se(contador = posiçãoRemover-1)
                        continuar

                se não se(contador < posiçãoRemover-1)
                        vetorNovo[contador]<-vetorOriginal[contador]

                se não
                        vetorNovo[contador-1]<-vetorOriginal[contador]
        fim para

        desaloca(vetorOriginal)

        numeroElemento = numeroElemento -1

        retorne vetorNovo

fim
```

Figura 7: Pseudocódigo da remoção de elemento

2.5 PESQUISA POR RG

Em pesquisa pelo RG existe uma diferença existente em códigos como aquele para imprimir a lista e salvar ele em algum arquivo dos que adicionam elemento. Essa diferença vem do fato que esse código se utilizam das propriedades de cada tipo de lista para fazer a busca. Ou seja na sequencial é um simples loop que percorre o vetor, já na encadeada é necessário saber qual o próximo elemento para poder continuar avaliando a lista em busca do rg selecionado. A *figura 8* apresenta a ideia de um laço simples apenas percorrendo o vetor e adicionando um controlador para verificar se achou ou não e caso não seja capaz de imprimir na tela a mensagem de não encontrado.

```
inicio

        para(contador<-0 ; contador<tamnhoLista ; contador<-contador+1)

                se(vetor[contador].rg = rgProcurado)
                        imprimir(vetor[contador].nome,vetor[contador].rg)
                        achou = 1
        fim para

        se(achou = 0)
                imprimir(RG não encontrado)

fim
```

Figura 8: Pseudocódigo da procura do elemento por rg na lista sequencial, uso da ideia de loop simples.

Já na lista encadeada foi usado uma forma um pouco diferente, onde o elemento avaliado naquele momento é atualizado por referência de quem é o próximo dele, ou seja ao invés de uma atualização simples se é utilizado as referências de ponteiros para poder avaliar um novo elemento no próximo laço. A *figura 9* apresenta esse conceito utilizado.

```
inicio

        para(contador<-0 ; contador<tamnhoLista ; contador<-contador+1)


                se(elemento.rg = rgProcurado)
                        imprimir(elemento.nome,elemento.rg)
                        quebra

                 se não se(listaEncadeada.ultimo.rg = elemento.rg e elemento.rg !=
rgProcurado)
                        imprimir(RG não encontrado)
                        quebra
```

```
            elemento<-elemento.proximo
        fim para

fim
```

Figura 9: Pseudocódigo da procura do elemento por rg na lista encadeada.

Note que na lista encadeada foi feito uma verificação se o elemento avaliado é o mesmo que o último elemento da lista encadeada e se ele for diferente do rg procurado será impresso em tela a mensagem de que não foi encontrado.

2.6 SALVAR E IMPRIMIR EM TELA

Para ambos os caso a lógica usada ficou igual a aquela apresentada acima na pesquisa, um laço simples para a sequencial percorrendo ela e imprimindo em tela o elemento atual e na encadeada alteração de elemento por referência ao próximo elemento da lista ser impresso em tela. Em si essa é são as principais diferenças que podem ser visualizada na *figura 10* que compara os dois tipos de código códigos.

```
inicio

  para(contador<-0;contador<tamnhoLista;
contador<-contador+1)

imprimir(vetor[contador].nome,vetor[conta
dor].rg)

fim para

fim
```

```
inicio

   para(contador<-0; contador<tamnhoLista
; contador<-contador+1)

imprimir(elemento.nome,elemento.rg)
   elemento<-elemento.próximo

   fim para

fim
```

Figura 10: Pseudocódigo de impressão de elementos, a esquerda de lista sequencial, a direita de lista encadeada.

Para salvar em um arquivo o método usado foi o mesmo, com a diferença da utilização de uma função que escreve em um arquivo existente em C.

3.Comparações entre os tipos de listas:

No final a lista sequencial se provou mais simples de fazer as alterações desejada na forma inicial de vetor simples e também se provou mais rápido na hora de alocar memória e de fazer buscas dentre outras coisas. Porém um vetor simples tem um sério problema que seria seu tamanho, sem usar o vetor de ponteiros, ao tentar passar usar as maiores listas, como de um milhão ou de dez milhões de registros, não era gerado a lista inteira, pois não existia espaço sequencial suficiente na memória para isso. Sendo assim a lista sequencial tinha uma séria limitação para grandes listas.

A lista encadeada era um pouco mais complexa na questão de adicionar, remover e pesquisar por um elemento, más ela não enfrentou o problema de espaço na memória da lista sequencial uma vez que cada elemento era alocado separadamente com referências para o elemento anterior e para o próximo. Ao passar para um vetor de ponteiros uma outra questão foi levantada, as referências aos outros registros, quando se adiciona um elemento, quando se remove e principalmente quando se organiza um elemento, por isso foi desenvolvido um algoritmo que percorre o vetor de ponteiros da lista encadeada e que arruma as referências, esse algoritmo é apresentado na *figura 11*.

```
inicio

    para(contador<-0 ; contador<tamnhoLista ; contador<-contador+1)

        se(contador = 0)
            listaEncadeada.primeiro<-vetorPonteiro[contador]
            vetorPonteiro[contador].proximo<-vetorPonteiro[contador+1]
            vetorPonteiro[contador].anterior<-nulo

        se não se(contador = numeroElemento)
            listaEncadeada.ultimo<-vetorPonteiro[contador]
            vetorPonteiro[contador].anterior<-vetorPonteiro[contador-1]
            vetorPonteiro[contador].proximo<-nulo

        se não
            vetorPonteiro[contador].anterior<-vetorPonteiro[contador-1]
            vetorPonteiro[contador].proximo<- vetorPonteiro[contador+1]

    fim para

fim
```

Figura 11: Pseudocódigo do algoritmo para arrumar referências de elementos na lista encadeada.

4.Métodos de Organização:

Com o vetor de ponteiros já implementado os métodos de organização foram facilitados graças ao fator da troca de posição dos elementos, principalmente da lista encadeada. Também já leve em consideração o algoritmo de organização de referências da lista encadeada já implementado pois assim, não é necessário se preocupar com as referências estarem corretas no final da organização pois terá uma parte do código se certificando disso.

4.1 SELECTION SORT

Um dos mais simples de ser implementado, ele segue a primeira lógica que aprendemos para organizar um vetor, o elemento atual é separado e comparado com o vetor inteiro em busca de um elemento menor e caso ache, troca eles de posição.

Embora simples ele traz alguns problemas referente a tempo de execução em listas maiores, demora demais para executar deixando testes até mesmos inviáveis para listas muito grandes como de dez milhões de registros, embora seja eficiente para listas pequenas. a *figura 12* apresenta a lógica usada na construção do algoritmo.

```
inicio

    para(contador1<-0 ; contador1<tamnhoLista ; contador1<-contador1+1)
        para(contador2<-contador1+1 ; contador2<tamnhoLista ;
    contador2<-contador2+1)

            se(vetorPonteiros[contador2].rg < vetorPonteiros[contador1].rg)
                ponteiroAuxiliar<-vetorPonteiros[contador2]
                vetorPonteiros[contador2]<-vetorPonteiros[contador1]
                vetorPonteiros[contador1]<-ponteiroAuxiliar

        fim para
    fim para

fim
```

Figura 12: Pseudocódigo do selection sort

Note que o segundo contador começa a partir da próxima posição em relação ao primeiro contador, ignorando assim os elementos já organizados.

4.2 BUBBLE SORT

O bubble sort é outro que é simples de compreender e de implementar uma vez que ele só analisa o elemento atual e o que está logo em seguida dele. A parte difícil foi gerar o controle já que pode ser necessário passar o bubble mais de uma vez para organizar a lista em questão, por isso foi gerado um laço maior que enquanto um controlador não for zero ele não para de rodar o laço, estando zerado somente quando não tiver feito nenhuma movimentação. A *figura 13* apresenta o pseudocódigo do bubble sort.

```
inicio

    enquanto(trocaRealizadas != 0)
        trocasRealizadas<-0
        para(contador<-0 ; contador<tamnhoLista ; contador<-contador+1)

            se(vetorPonteiros[contador+1].rg < vetorPonteiros[contador].rg)
                ponteiroAuxiliar<-vetorPonteiros[contador+1]
                vetorPonteiros[contador+1]<-vetorPonteiros[contador]
                vetorPonteiros[contador]<-ponteiroAuxiliar
                trocasRealizadas<-trocasRealizadas+1
```

```
            fim para
        fim enquanto

fim
```

Figura 13: Pseudocódigo do bubble sort

Embora ele pareça ser menos eficiente que os demais a um primeiro momento, se é visto que para listas já organizadas onde se é necessário arrumar apenas um ou dois elementos ele acaba sendo um dos melhores. Mesmo assim não foi realizados testes nas maiores listas estando desorganizadas pela questão do tempo de demora para finalizar a execução.

4.3 INSERTION SORT

O insertion é o último dos métodos mais simples, sua organização se baseia em ler e organizar um pedaço do vetor e no próximo laço aumentar um pouco esse tamanho e ir assim até ele ler e organizar o vetor inteiro. Nas listas menores ele tem vantagens sobre os outros em caso de estarem desorganizadas, porém ele também demorou demais nas listas maiores. Também é válido comentar o fato de ele não ser tão eficiente para listas já organizadas, pois sua forma de leitura em partes de tamanho crescente acaba atrasando na questão de listas já organizadas. A *figura 14* apresenta o insertion usado.

```
inicio

        para(contador1<-2 ; contador1<tamnhoLista ; contador1<-contador1+1)
                para(contador 2<-0 ; contador2<contador1 ; contador2<-contador2+1)

                        se(vetorPonteiros[contador2].rg < vetorPonteiros[contador1].rg)
                                ponteiroAuxiliar<-vetorPonteiros[contador2]
                                vetorPonteiros[contador2]<-vetorPonteiros[contador1]
                                vetorPonteiros[contador1]<-ponteiroAuxiliar

                fim para
        fim para

fim
```

Figura 14: Pseudocódigo do insertion sort

Foi escolhido se iniciar com dois por ser o menor número para ocorrer comparações é com dois elementos

4.4 QUICK SORT

O funcionamento do quicksort muda bastante dos anteriores por apresentar uma maior fragmentação da lista que os outros, ele faz isso com base em um pivô, um número de base que serve como base para ele organizar a lista, a seleção desse pivô acaba por ser de grande importância pois se mal escolhido pode aumentar tempo de execução, número de movimentações e etc. Seguindo o modelo passado o pivô escolhido foi o elemento do meio como mostra a *figura 15* a seguir.

```
inicio
        inteiro contador1<-inicioLista, contador2<-fimLista, pivo, meio

        meio <-(contador1+contador2)/2
        pivo<- lista[meio].rg

        enquanto(contador1 < contador2)
                enquanto(lista[contador1].rg < pivo)
                        contador1<-contador1+1

                enquanto(lista[contador2].rg > pivo)
                        contador2<-contador2-1

                        se(contador1 < contador2)
                                ponteiroAuxiliar<-vetorPonteiros[contador2]
                                vetorPonteiros[contador2]<-vetorPonteiros[contador1]
                                vetorPonteiros[contador1]<-ponteiroAuxiliar

                                contador1<-contador1+1
                                contador2<-contador2-1
                        fim se

                        se(contador2 > inicioLista)
                                chama quickSort(lista<-lista, inicioLista<-inicioLista,
fimLista<-contador2)
                        fim se

                        se(contador1 < inicioLista)
                                chama quickSort(lista<-lista, inicioLista<-contador1,
fimLista<- fimLista)
                        fim se

                fim para
        fim para

fim
```

Figura 15: Pseudocódigo do quick sort

A forma de pensar "dividir para conquistar" usada no quicksort é reutilizada nos próximos sorts.

Pelo uso da fragmentação o quicksort pode ser usado nas maiores listas assim como as próximas formas de organização apresentadas.

4.5 MERGE SORT

O merge sort consiste em transformar a lista em pequenos vetores, fragmentá-la para assim começar a organizar a lista e depois disso unir o vário vetores criados em um só(merge). Nele se cria um vetor auxiliar que recebe as posições correta, que faz a união dos demais vetores. Para essa forma de organização foram usados comparações que inicialmente verificar se o todo de um vetor já foi para então simplesmente adicionar o outro. Caso não verifica qual o menor e adiciona ele na posição atual do vetor auxiliar e depois copia o vetor auxiliar para o vetor original. Esse exemplo é dado na *figura 16*.

```
inicio
        inteiro contador1, contador2, contador3, meio

        se(inicioLista = fimLista)
                quebra
        fim se

        meio <-(inicioLista + fimLista)/2
        mergeSort(lista<-lista, inicioLista<-inicioLista, fimLista<-meio)
        mergeSort(lista<-lista, inicioLista<-meio+1, fimLista<-fimLista)

        contador1<-inicioLista
        contador2<-fimLista
        contador3<-0

        vetorListaAuxiliar<-aloca(tamanho de estruturaLista * (fimLista-InicioLista+1))

        enquanto(contador1<meio ou contador2<fimLista )
                se(contador1= meio+1)
                        vetorListaAuxiliar[contador3]<-lista[contador2]
                        contador2<-contador2+1
                        contador3<-contador3+1
                fim se
                se não
                        se(contador2= fimLista+1)
                                vetorListaAuxiliar[contador3]<-lista[contador1]
                                contador1<-contador1+1
                                contador3<-contador3+1
                        fim se
                        se não
                                se(lista[contador1].rg< lista[contador1].rg)
                                        vetorListaAuxiliar[contador3]<-lista[contador1]
                                        contador1<-contador1+1
                                        contador3<-contador3+1
```

```
                        fim se
                        se não
                                vetorListaAuxiliar[contador3]<-lista[contador2]
                                contador2<-contador2+1
                                contador3<-contador3+1
                        fim se
                fim se
        fim se
    fim enquanto
fim
```

Figura 16: Pseudocódigo do merge sort

4.6 SHELL SORT

O método de shell sort usa de uma divisão feita por elementos de igual distância de um salto 's', sendo 's' o número de elementos a serem pulados.Ou seja se eu vou começar com o elemento de posição '0' então o vetor será *0,s,2s,3s,...,Ns*. Depois de organizar o vetor o tamanho do salto diminui e recomeça o processo até o salto ser menor que 1 elemento por vez. Esse raciocínio está demonstrado na *figura 17*.

```
inicio

inteiro contador1, contador2, salto, tamanho
estruturaLista *ponteiro auxiliar

tamanho<-numeroElementos
salto<-1

enquanto(salto < tamanho)

salto<-(salto)*3+1

fim enquanto

enquanto(salto > 1)

        salto<-salto/3

        para(contador1<-salto; i<-i+1)
                ponteiroAuxilar<-vetor[contador1]
                contador2<-contador1-salto

                enquanto(contador2>= 0 e ponteiroAuxiliar.rg < lista[contador2].rg)

                lista[contador2+salto] <- lista[contador2]
                contador2<-contador2-salto
```

```
            fim enquanto

            lista[contador2+salto] <- ponteiroAuxiliar

        fim para
fim enquanto

fim
```

Figura 17: Pseudocódigo do shell sort

A fórmula usada de (salto)*3+1 foi retirada de um estudo, onde Donald Knuth provou que esta é a melhor forma de escolher o tamanho de um salto em comparação às outras fórmulas.

4.7 BUSCA BINÁRIA

A pesquisa binária funciona de forma análoga ao metodo da bisseccao, indo diretamente a posição do meio verificando se é ela, caso não verifica se é maior ou menor e com base nisso decide se vai suas a primeira ou a segunda metade e quando decide encontra a metade dele e vai para lá. Ele repete isso até encontrar o elemento buscado ou retorna que não foi possível encontrar o elemento. Embora seja mais eficiente que a busca simples ele exige que a lista esteja organizada para poder funcionar. O algoritmo que explica a lógica está presente na *figura 18*.

```
inicio
        enquanto(inicio <= fim)
                posiçãoAvaliada <-(fim+inicio)/2

                se(vetor[posiçãoAvaliada].rg = rgBuscado)
                        quebra
                fim se

                se(vetor[posiçãoAvaliada].rg > rgBuscado)
                        fim<-posiçãoAvaliada-1
                fim se
                se não
                        inicio<-posiçãoAvaliada+1
                fim se

        fim enquanto

        se(inicio>fim)
                imprimir(RG nao encontrado)
        fim se
```

```
        se não
                imprimir(vetor[posiçãoAvaliada].rg,vetor[posiçãoAvaliada].nome)
        fim se
fim
```

Figura 18: Pseudocódigo da busca binária

5.Comparações entre os métodos de organização:

Dentre os vários métodos de organização foram realizados testes com os arquivos de diferentes tamanhos e foi apresentado número de comparações, número de movimentações e o tempo de execução. Pela velocidade do computador usado para os teste o tempo acabou não sendo um bom parâmetro para verificar eficiência do método, dando assim mais destaque aos números de comparações e movimentações. Será apresentado duas tabelas, a primeira irá apresentar os métodos de organização que não foram testadas nas maiores listas, em seguida aquele que foram na segunda tabela. As tabelas terão: Tempo, número de comparações(C) e número de movimentações(M).

	Selection Sort			**Bubble Sort**			**Isertion Sort**		
	Tempo	*C*	*M*	*Tempo*	*C*	*M*	*Tempo*	*C*	*M*
10	0	163	119	0	121	113	0	51	113
50	0	3258	1924	0	2704	1912	0	704	1912
100	0	12767	7601	0	11461	7583	0	2661	7583
1k	0	1244030	725090	0	1238024	725072	0	243022	725066
10k	1s	125145127	75365381	1s	124145109	753653327	0	25135109	75365327

Tabela 1: Tabela de métodos que não rodaram nas maiores listas

	Quick Sort			Merge Sort			Shell Sort		
	Tempo	*C*	*M*	*Tempo*	*C*	*M*	*Tempo*	*C*	*M*
10	0	50	77	0	140	88	0	46	51
50	0	401	442	0	1170	672	0	436	578
100	0	924	977	0	2702	1544	0	973	1315
1k	0	13293	11966	0	40432	21952	0	15567	21024
10k	0	172291	142403	0	530140	287232	0	269526	344769
1m	1s	23886254	18934349	0	80250578	41902848	2s	69057423	80861688
10m	8s	329947820	251844939	8s	1111776840	577364656	50s	1418388788	1583853776

Tabela 1: Tabela de métodos que não rodaram nas maiores listas

É relevante apresentar que o bubble se torna útil com o aumento do tamanho das listas.

6.Conclusão:

Na visão geral os algoritmos de ordenação e busca se levanta a pergunta: Qual é mais eficiente? Não existe uma resposta correta para essa pergunta, varia de situação para situação. A busca binária usa menos comparações para encontrar uma resposta, porém a lista deve estar organizada para isso. A mesma coisa para os algoritmos de ordenação, o quicksort é mais eficiente para listas maiores, para outras listas o shellsort se mostrou mais eficiente. Para listas já organizadas métodos como bubble sort e insertion sort se apresentam mais eficientes. Em geral a situação ao qual você se encontra é que define qual a melhor solução para ela esse é o principal fator a ser levado em consideração.

7.Referencias:

https://web.inf.ufpr.br/menotti/ci056-2015-2-1/slides/aulaORDShellSort.pdf