FUNDAÇÃO UNIVERSIDADE FEDERAL DE RONDÔNIA
UNIR

NÚCLEO DE TECNOLOGIA
DEPARTAMENTO ACADÊMICO DE COMPUTAÇÃO

PROGRAMAÇÃO II

linguagem C

Profa. Dra. Liliane Jacon

Porto Velho, 2022

Disciplina: PROGRAMAÇÃO I I Carga Horária 100 horas/aula
Professor: Dra.Liliane da Silva Coelho Jacon
Curso de Bacharelado em Computação (59) INF31023
Curso de Licenciatura em Computação (60) INF31091
Período: 2º semestre/2021 (ministrado entre 25/abr/22 e 9/ago/22)
Pré-requisito: Programação I

Objetivos
Apresentação de conceitos avançados que levem o aluno a uma maturidade em programação estruturada, com conhecimento de uma linguagem de programação com recursos avançados. Aprendizado de técnicas para construção de algoritmos e para análise da complexidade de algoritmos. Aprendizado de algoritmos clássicos de ordenação e busca em memória interna. Prática de Programação.

Ementa
Algoritmos de ordenação interna simples e avançados: conceitos básicos, métodos de ordenação bubblesort, quicksort, inserção, shellsort, seleção, heapsort, mergesort. Análise de algoritmos: conceitos básicos e análise dos casos (pior, médio e melhor). Algoritmos de busca interna: conceitos básicos, métodos de busca sequencial, sequencial indexada, binária, binária recursiva. Hashing: conceitos básicos, funções hash, tratamento de colisões, Método do endereçamento aberto (tentativa linear), Utilização lista encadeada com área de overflow e Método da divisão. Armazenamento em memória secundária: arquivos textos e binários. Índices: aplicações utilizando arquivos de dados e tabelas de índices.

Critério de Avaliação:
Pb1 = 1ª.Prova - Métodos de ordenação e buscas. Arquivos Textos
Pb2 = 2ª.Prova – Hashing e Arquivos Binários
Pb3 = 3ª.Prova – Revisão de tudo (dos dois bimestres)
Trab = trabalhos – atividades práticas
Média parcial = [(Pb1+Trab) + (Pb2 + Trab)] /2
Média Final = 2 maiores notas entre [(Pb1+trab) (Pb2+trab) e Pb3] dividido por 2

Bibliografia:

ASCENCIO, Ana Fernanda Gomes; ARAUJO, Graziela Santos de. **Estrutura de dados**: Algoritmos, Análise da Complexidade e Implementações em Java e C C++. Pearson Universidades Ed. São Paulo, 2010.

SCHILDT, H. **Turbo C** - guia do usuário. Editora McGraw-Hill, 1988.***Schildt***, H. C – completo e total. Editora McGraw-Hill, 1990.

CORMEN,Thomas H et al. **Algoritmos**: teoria e prática. trad. 2ª.ed. Rio de Janeiro: Campus/Elsevier, 2002.

ZIVIANI, N. **Projeto de Algoritmos com implementações em Java e C++,** Thomson 2007.

Cronograma das aulas de Programação II – 1º.sem/2021 100 horas/aula
Calendário: de 8/fev/2021 a 21/maio/2021 ERE
Aulas às terças e quintas – das 13:50 às 18 horas
INF31091 Licenciatura e INF31023 Bacharelado

DATA	Numero de Aulas	CONTEUDO
29/abr	5a	Apresentação conteúdo, avaliação e bibliografia **Arquivo Texto** – Conceitos, utilização e exemplos Exemplo e exercício prático (via Google meet)
6/maio	5	**Métodos de busca**: exaustiva, seqüencial e binária. Cronometragem do tempo gasto (milisegundos) utilizando as buscas estudas. Ordenação do índice pelo método da **Bolha**
13/maio	5	Construção de um programa para Geração de números aleatórios armazenados num vetor.
20/maio	5	**Método da Seleção e da Inserção Direta.** Construção de um programa completo para realizar buscas e ordenação de elementos armazenados num vetor (método da Bolha e da Seleção)
27/maio	5	Métodos de ordenação – **Inserção Binária** Alteração na quantidade de elementos do vetor p/ comparar o tempo gasto Comparação do tempo gasto entre os diferentes métodos estudados
3/junho	5	**Recursividade** Fatorial, Busca Binaria recursiva Métodos da **QuickSort, HeapSort** Comparações do Tempo gasto de execução entre os métodos já estudados
4/junho **Assíncrona**	5	**Método MergeSort** Exercício no laboratório – análise empírica do tempo gasto com os métodos estudados - Ordenação&Busca
10/junho	5	Exercício utilizando Arquivo texto Entrega dos trabalhos: Análise Empírica do Tempo Gasto com os métodos de ordenação estudados
17/junho	**5**	**Complexidade de algoritmos** – Regras de cálculo Complexidade dos métodos de ordenação - Taxa de crescimento
24/junho	5	**1ª.PROVA** Métodos de Ordenação e de Buscas Exercício utilizando Arquivo Texto
1/julho	5	Início de **Hashing** – conceitos – tratamento de colisão Função Hash Método do endereçamento aberto (tentativa linear)
8/julho	5	Hashing - Implementação Método de Endereçamento aberto – tentativa linear Hashing – Método de Encadeamento (Lista)
9/julho **Assíncrona**	**5**	Hashing – trabalho prático Implementação de exercício prático utilizando lista e Endereçamento aberto com tentativa linear
15/julho	5	Hashing: Método da Lista encadeada (com utilização de **ponteiros dinâmicos**)
22/julho	5	Início de **Arquivo Binário** – conceitos, comandos e exemplo Exemplos
23/julho **assíncrona**		Projeto utilizando Arquivo binário - atividade assincrona

29/julho		**2ª.PROVA – HASHING – endereçamento aberto e com Encadeamento (Lista Encadeada dinâmica)**
5/agosto		**EXAME/ substitutiva**
TOTAL DE AULAS MINISTRADAS		15 encontros de 5 aulas + 3 encontros assíncronos (sábados)

INDICE

1. ARMAZENAMENTO EM MEMÓRIA SECUNDÁRIA

O armazenamento de dados em variáveis, arrays, listas e árvores são temporários. Quando o programa termina sua execução, os dados armazenados nestas estruturas (em variáveis, arrays, listas ou árvores) são perdidos. Computadores utilizam arquivos para armazenamento de longo prazo de grandes volumes de dados.
Arquivos criados com base nos **fluxos de bytes são chamados arquivos binários** e arquivos criados com base nos **fluxos de caracteres são chamados arquivos de textos**. Arquivos de textos podem ser lidos por editores de texto (exemplo: bloco de notas), enquanto arquivos binários são lidos exclusivamente por um programa que converte dados em um formato legível por humanos.

1.1 ARQUIVOS TEXTOS

Comandos para abertura de arquivos

a) Comando necessário que permite que as informações sejam armazenadas temporariamente para realizar a transferência entre a memória do computador e o arquivo de dados.

```
FILE *arq;
```

b) Comando de abertura do arquivo (leitura e/ou gravação de dados)

```
arq = fopen ("nomeArquivo.txt", "tipo");
```

Modo	Arquivo	Função
"r"	Texto	Leitura. Arquivo deve existir.
"w"	Texto	Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"a"	Texto	Escrita. Os dados serão adicionados no fim do arquivo ("append").
"rb"	Binário	Leitura. Arquivo deve existir.
"wb"	Binário	Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"ab"	Binário	Escrita. Os dados serão adicionados no fim do arquivo ("append").
"r+"	Texto	Leitura/Escrita. O arquivo deve existir e pode ser modificado.
"w+"	Texto	Leitura/Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"a+"	Texto	Leitura/Escrita. Os dados serão adicionados no fim do arquivo ("append").
"r+b"	Binário	Leitura/Escrita. O arquivo deve existir e pode ser modificado.
"w+b"	Binário	Leitura/Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"a+b"	Binário	Leitura/Escrita. Os dados serão adicionados no fim do arquivo ("append").

Exemplo 1:

```
FILE *fp; /* declaração de um arquivo
fp= fopen("listatelefones.txt","r"); // o arquivo se chama listatelefones.txt para leitura.
                                     // O arquivo deve existir antes de ser aberto
if (!fp) {
    printf("Erro na abertura do arquivo");
    exit(0);
},
```

As funções ***fprintf*** e ***fscanf*** são utilizadas para escrever e para ler no arquivo.
A função ***fclose()*** é utilizada para fechar o arquivo.
Função *feof()* (função EOF End Of File) indica o fim do arquivo. As vezes é necessário verificar se um arquivo chegou ao fim. Para isso, podemos usar a função **feof(arq)**. Ela retorna não-zero se arquivo chegou ao EOF, caso contrário retorna zero.

Exemplo 2: Imprimir os números de 1 até 10 no arquivo teste.txt

```
#include <stdio.h>
main() {
  FILE *f = fopen("teste.txt", "w"); // abre arquivo texto para gravação
  int i;
  for (i=1; i<=10; i++)
        fprintf(f, "%d\n", i);
  fclose(f);
}
```

Exemplo 3: Imprimir os números que estão no arquivo teste.txt

```
#include <stdio.h>
main( ) {
   FILE *f = fopen("teste.txt", "r");
    int i;
    while (fscanf(f, "%d", &i) == 1)
            printf("%d\n", i);
    fclose(f);
}
```

Exemplo 4: Em um arquivo chamado "notas.txt" estão os dados nomes e notas de alunos. Em cada linha há o nome do aluno, seguido de três notas:

Maria 8 8 10
Jose 6 6.3 8
Camila 7 9.5 7.5
Pedro 10 10 10

Crie um programa que exiba o nome de cada aluno e sua média.
Observe que em toda linha, há um padrão:

string, espaço, número, espaço, número, espaço, número, enter
Ou seja: "%s %f %f %f\n"

```
#include <stdio.h>
main() {
 char nome[20];
float  nota1, nota2, nota3;
FILE *arq = fopen("notas.txt", "r");
if(arq == NULL)
printf("Erro, nao foi possivel abrir o arquivo\n");
else  {
while( !feof(arq) ){
    fscanf(arq,"%s %f %f %f\n", nome, &nota1, &nota2, &nota3);
                        float media = (nota1 + nota2 + nota3) /3;
    printf("%s obteve media %.2f\n", nome, media);
              }
   fclose(arq);
     }
     getchar();
}
```

//----------

EXERCÍCIOS

1) Suponha a existência das informações abaixo armazenadas num arquivo texto chamado "empresa.txt".

nome	Salario	Idade
Carmem	3500.00	23
Lucila	2890.45	42
Vera	6800.00	51
Luís	9300.50	55
Benjamin	12000.00	38

O seu programa deve ler as informações de um arquivo texto e, em seguida, deve armazená-las num vetor (cada linha do vetor corresponde a um funcionario – utilize STRUCT).
Fazer uma função para ler o nome de um funcionario X. Informar se ele está entre os funcionários da empresa, e também o salário e a sua idade.
Acrescente opções para inserir um novo funcionario e para remover um funcionario existente. Regrave as informações do vetor no arquivo texto (após vetor atualizado).
Menu:

1- Leitura de arquivo texto para carregar o vetor
2- Exibe o vetor
3- Consulta um funcionario (busca)
4- Inclusão de um novo funcionario no vetor
5- Remoção de um determinado funcionário dos vetor
6- Regravar e exibir o arquivo texto atualizado
7- Sair

```
struct funcionario{
char nome[10];
float salario;
int idade;
};
main(){
int opcao=0, tl=0, posicao;
struct funcionario vetor[10];
char nome[10];
while (opcao!=7){
printf("\n1 - Leitura do arquivo texto para carregar o vetor");
printf("\n2 - Exibe o vetor");
printf("\n3 - Consulta um funcionario (busca)");
printf("\n4 - Inclusão de um novo funcionario");
printf("\n5 - Remoção de um funcionario no vetor");
printf("\n6 - Regravar e exibir o arquivo texto atualizado ");
printf("\n7 - Sair");
printf("\n Opcao:"); scanf("%d",&opcao);
switch (opcao) {
case 1: carrega(vetor,tl);
        printf("\n tl %d", tl);
        break;
case 2: exibe(vetor,tl);  break;
```

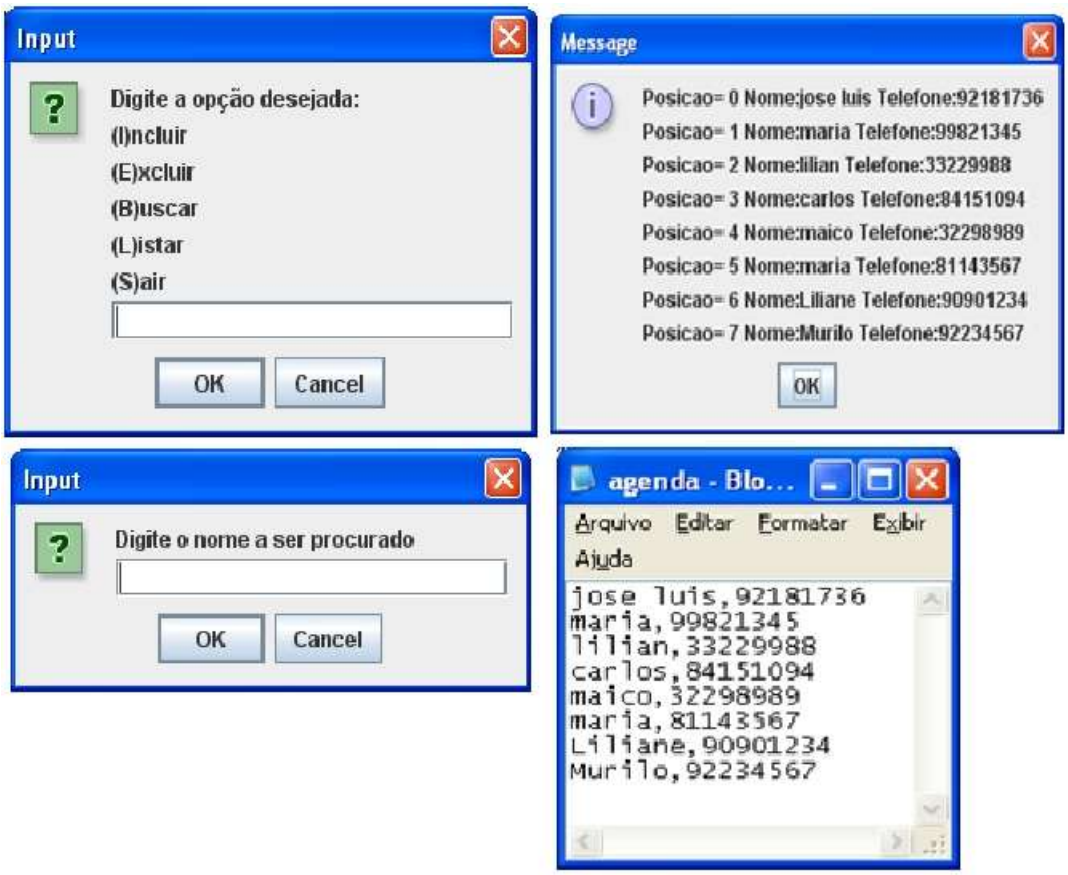
```
case 3: printf("\n Entre com o nome a ser procurado: ");
          scanf("%s",&nome);
posicao = busca(vetor, tl, nome);
if (posicao==-1)
  printf("\n Nao encontrou");
     else printf("\n Encontrou na posicao %d %s",posicao,nome);
break;
case 6: regravar(vetor,tl);
       break;
}
}
}
//---------------------------------------------------------------------------
CASE 1: LEITURA do arquivo texto empresa.txt para carrega o vetor e EXIBE
void carrega(struct funcionario vetor[10],int &tl){
tl=0;
char nome[10];
int idade;
float salario;
FILE *arq = fopen("empresa.txt", "r");
if(arq == NULL)
printf("Erro, nao foi possivel abrir o arquivo\n");
else  {
while( !feof(arq) ){
    fscanf(arq,"%s %f %d\n",nome,&salario,&idade);
    strcpy(vetor[tl].nome,nome);
    vetor[tl].salario = salario;
    vetor[tl].idade=idade;
    tl++;
                }
   fclose(arq);
}
}
void exibe(struct funcionario vetor[10],int tl){
int i;
for (i=0;i<tl;i++)
    printf("\n %d Nome=%s  Salario=$%5.2f  Idade=%d",
                                    i,vetor[i].nome, vetor[i].salario, vetor[i].idade);
}
//---------------------------------------------------------------------------
CASE 6: Regravar o arquivo Texto após alterações no vetor
int busca(struct funcionario vetor[10], int tl, char nome[10]){
int posicao=-1, i;
for (i=0;i<tl;i++){
if (strcmp(vetor[i].nome,nome)==0)
    posicao=i;
}
return posicao;
}
```

```
void regravar(struct funcionario vetor[10], int tl){
  FILE *f = fopen("empresa.txt", "w"); // abre arquivo texto para gravação
  int i;
  for (i=0; i<tl; i++)
        fprintf(f,"%s %f %d\n",vetor[i].nome,vetor[i].salario,vetor[i].idade);
  fclose(f);
  printf("\n Regravado com sucesso!");
}
```

//----------

EXERCÍCIO 2

Suponha um arquivo texto "agenda.txt" contendo NOME e CELULAR.
Faça um programa completo para fazer a leitura do arquivo texto e carregar as informações num vetor em memória principal.
Em seguida, implemente opções para incluir um novo contato, remover algum nome já existente, Busca um contato e exibir o seu celular, e também uma opção para listar todos.
Ao sair do programa, REGRAVE as atualizações no arquivo texto "agenda.txt".



//----------

2. MÉTODOS DE PESQUISA E CLASSIFICAÇÃO (memória principal)

2.1 ALGORITMOS DE PESQUISA

Pesquisar dados envolve determinar se um valor (denominado chave de pesquisa) está presente nos dados e, se estiver encontrar a localização do valor.

2.1.1 Busca Exaustiva (num array desordenado)

O algoritmo testa cada elemento e, quando alcança o fim do array, informa ao usuário que a chave não está presente. O array está desordenado.

```
int  buscaExaustiva (int vetor[], int tl, int elem)
{
  int i=0;
  while ((vetor[i]<>elem)&&(i<tl))
    i=i+1;
  if ((i < tl) && (vetor[i] == elem))
          return i;          // achou na posição i
          else return  -1; // não achou
}
```

2.1.2 Busca Seqüencial (num array ordenado)

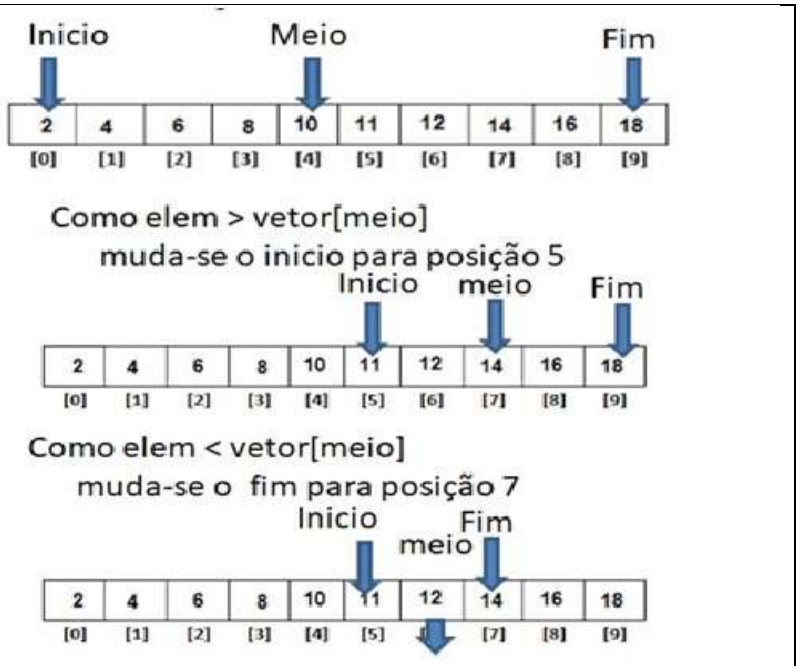
O array está ordenado. A busca deve testar até o primeiro elemento maior que a chave procurada. Como o vetor está ordenado, ao encontrar um elemento maior que a chave procurada, não adianta continuar verificando, pois dali para frente, os elementos serão todos maiores.

```
int   buscaSequencial(int vetor[], int tl, int elem)
  {
    int i=0;
    while ((elem > vetor[i])&&(i<tl))
          i=i+1;
    if ((i<tl) && (elem == vetor[i]))
        return i; // achou na posição i
     else return -1; // não achou
  }
```

2.1.3 Busca Binária (num array ordenado)

A primeira iteração testa o elemento do meio do array. Se isso corresponder a chave de pesquisa, o algoritmo termina. Se a chave de pesquisa for menor que o elemento do meio, a chave de pesquisa não poderá localizar nenhum elemento na segunda metade do array e o algoritmo continua apenas na primeira metade. Se a chave de pesquisa for maior que o elemento do meio, a chave de pesquisa não poderá localizar nenhum elemento na primeira metade do array e o algoritmo continua apenas com a segunda metade do array. Repare que a cada iteração, metade do array é descartado, ficando apenas a metade que interessa. E assim sucessivamente até localizar o elemento ou reduzindo o subarray ao tamanho zero.

```
  int buscaBinaria(int vetor[], int tl, int
elem)   {
     int inicio, meio, fim;
     inicio=0;
     fim=tl-1;
     meio=(inicio+fim)/2;

while((elem!=vetor[meio])&&(inicio<fim))
{
       if(elem>vetor[meio])
         inicio=meio+1;
         else
         fim=meio;
       meio=(inicio+fim)/2;
    }
     if (elem>vetor[meio])
       return(meio+1);
       else
       return(meio);
  }
```



BUSCA BINÁRIA RECURSIVA

```
int binariaRecursiva(int vetor[], int elem, int inicio, int fim) {
int meio;
   if (inicio > fim)
          return -1; //nao achou
      else {
          meio := (inicio+fim) div 2;
          if (elem == vetor[meio])
                  return meio;
              else if (elem < vetor[meio]
                    binariaRecursiva(vetor,elem,inicio ,meio-1)
                    else binariaRecursiva (vetor,elem,meio+1,fim);
           }
}
```

Chamada na função no principal (main)

```
        //Supor um vetor[50] com tl elementos.

        printf("\n Entre com o valor a ser procurado:");
        scanf("%d",&numProcurado);
        int posicao = binariaRecursiva(vetor, numProcurado, 0, tl-1);
        if (posição ==  -1)
                    printf("não achou");
                    else printf("\n achou na posição %d",posição);
```

REFAÇA a busca binária recursiva para retornar com a posição ideal de inserção

2.2 ALGORITMOS DE CLASSIFICAÇÃO

Classificar dados, isto é, colocar os dados em ordem crescente ou decrescente, é uma das aplicações mais importantes da computação. Todos os algoritmos aqui estudados terão o mesmo resultado final: o array classificado. A escolha do algoritmo só afeta o tempo de execução e o uso de memória do programa. Os algoritmos mais simples como o da bolha, da seleção direta e da inserção (direta e binária) são simples de programar, porém são pouco eficientes. O métodos mais eficientes de classificação são o HeapSort (método da seleção em árvore) e o Quicksort (método por troca).

- **Ordenação por troca**
  – *BubbleSort* (método da bolha)
  – *QuickSort* (método da troca e partição)
- **Ordenação por inserção**
  – *InsertionSort* (método da inserção direta)
  – *BinaryInsertionSort* (método da inserção direta binária)
- **Ordenação por seleção**
  – *SelectionSort* (método da seleção direta)
  – *HeapSort* (método da seleção em árvore)
- **Outros métodos**
  – *MergeSort* (método da intercalação)
  – *BucketSort* (método da distribuição de chave)

2.2.1 Algoritmo da Bolha

A técnica chamada de ordenação por submersão (*sinking sort*) funciona da seguinte forma: os valores maiores submergem para o final do array. Outra técnica é fazer os valores menores subirem gradualmente, até o topo do array (isto é, em direção ao topo do array). A ordenação pela submersão faz várias passagens pelo array. Cada passagem compara sucessivos pares de elementos. Se um par está em ordem crescente, os valores permanecem na mesma ordem. Se um par está em ordem decrescente, é realizada a troca entre os elementos do par.

Bubble Sort Example

9, 6, 2, 12, 11, 9, 3, 7
6, 9, 2, 12, 11, 9, 3, 7
6, 2, 9, 12, 11, 9, 3, 7
6, 2, 9, 12, 11, 9, 3, 7
6, 2, 9, 11, 12, 9, 3, 7
6, 2, 9, 11, 9, 12, 3, 7
6, 2, 9, 11, 9, 3, 12, 7
6, 2, 9, 11, 9, 3, 7, 12

```
public void bolha()   {
   int i, tam;
   tam=tl-1;
   while (tam>1) {
      for (i=0;ivetor[i+1]) {
           int aux=vetor[i];
           vetor[i]=vetor[i+1];
           vetor[i+1]=aux; }
      } //for do i
    tam=tam-1;
} // while do tam
```

2.2.2 Algoritmo da Inserção (Direta e Binária)

A primeira iteração seleciona o segundo elemento do array e, se for menor que o primeiro elemento, o permuta com o primeiro elemento (ou seja, **reorganiza apenas os dois primeiros elementos do array de forma crescente através de um IF**).

A próxima iteração (etapa 2 na figura abaixo) examina o terceiro elemento e o insere na posição correta com relação aos dois primeiros elementos de modo que todos os três elementos fiquem em ordem crescente (resultado ilustrado na etapa 3). E assim sucessivamente para os demais elementos.

Diferença entre inserção direta e binária: ao procurar o local ideal para inserir o novo elemento entre os elementos já ordenados, pode-se optar por uma busca seqüencial ou pela busca binária.

Inserção Direta - Exemplo

Vetor Inicial	(27 12 20 37 19 17 15)
Etapa1:	(27 | 12 20 37 19 17 15)
Etapa2:	(12 27 | 20 37 19 17 15)
Etapa3:	(12 20 27 | 37 19 17 15)
Etapa4:	(12 20 27 37 | 19 17 15)
Etapa5:	(12 19 20 27 37 | 17 15)
Etapa6:	(12 17 19 20 27 37 | 15)
Etapa7:	(12 15 17 19 20 27 37)

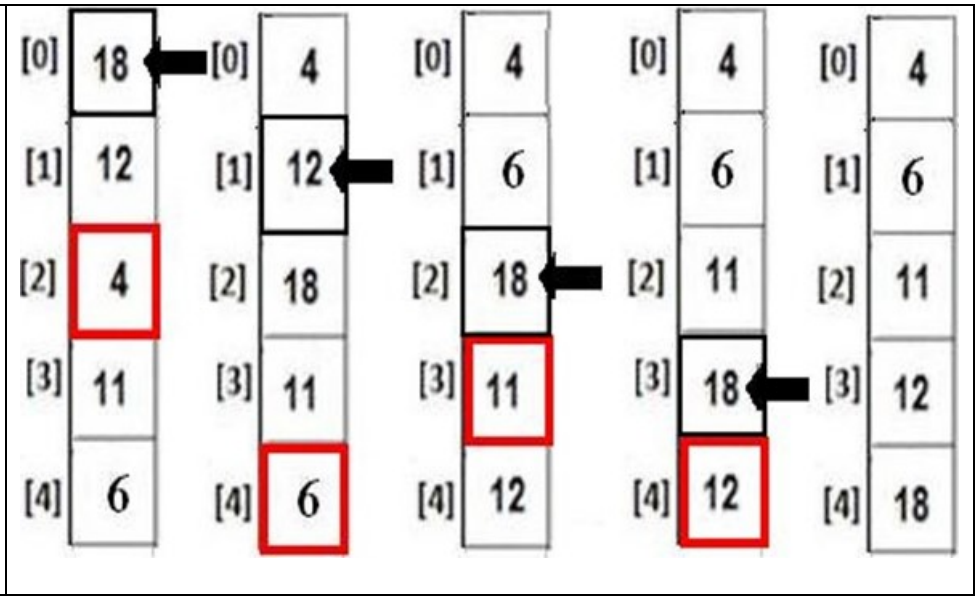
```
public void insercaoDireta ()   {
  int aux, i, j, n, posi;
  if (vetor[0]>vetor[1])  {
    aux=vetor[0];
    vetor[0]=vetor[1];
    vetor[1]=aux;
  }
  for (n=2;n<tl;n++)  {
    aux=vetor[n];
   //aux é o elemento a ser reinserido de forma ordenada
   // n-1 é o tamanho do vetor que já está ordenado
    posi=buscaSequencial(vetor[n],n-1);
    for(j=n ;j>posi ;j--)
          vetor[j]=vetor[j-1];
    vetor[posi]=aux;
  }  }
```

2.2.3 Algoritmo da Seleção

A primeira iteração do algoritmo seleciona o menor elemento e o permuta com o primeiro elemento (na figura abaixo o menor elemento é o 4 e ele deve ficar na posição 0 do vetor). A segunda iteração seleciona o segundo menor item (que é o menor item dos elementos restantes) e o permuta com o segundo elemento (o segundo menor elemento é o 6 e ele deve ficar na posição 1 do vetor). E assim sucessivamente até o penúltimo elemento.

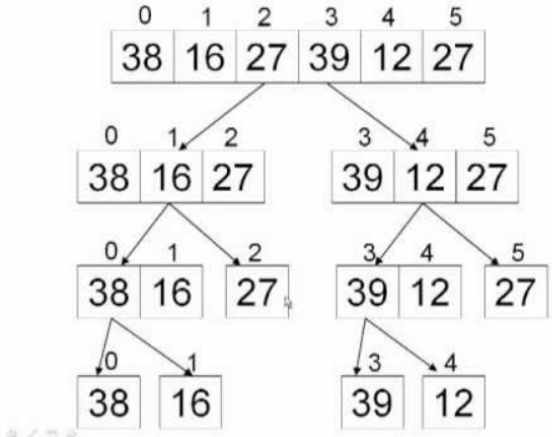
Seleção Direta: Consiste em encontrar a menor chave por pesquisa seqüencial. Encontrando a menor chave, essa é permutada com a que ocupa a posição inicial do vetor, que fica então reduzido a um elemento. O processo é repetido para o restanto do vetor, sucessivamente, até que todas as chaves tenham sido selecionadas e colocadas em suas posições definitivas.

```
void selecao(int vetor[],int tl)
{
    int i, j, menor, posmenor;
    for (i=0;i<tl-1;i++)      {
      menor=vetor[i];
      posmenor=i;
      for(j=i+1;j<tl;j++)
        if(vetor[j]<menor)
        {
          menor=vetor[j];
          posmenor=j;
        }
      vetor[posmenor]=vetor[i];
      vetor[i]=menor;
    }
  }
```



2.2.4 MergeSort

A idéia principal é combinar /juntar 2 listas já ordenadas. Primeiramente, o algoritmo quebra um array original em dois outros de tamanhos menores (recursivamente) até obter arrays de tamanho 1, depois retorna da recursão combinando os resultados. Um dos principais problemas com o MergeSort é que ele faz uso de array auxiliar.



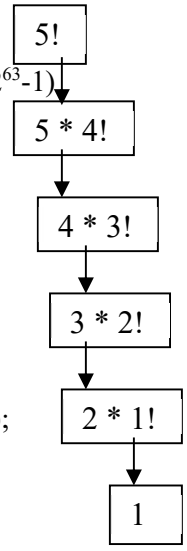
EXTRA: RECURSIVIDADE

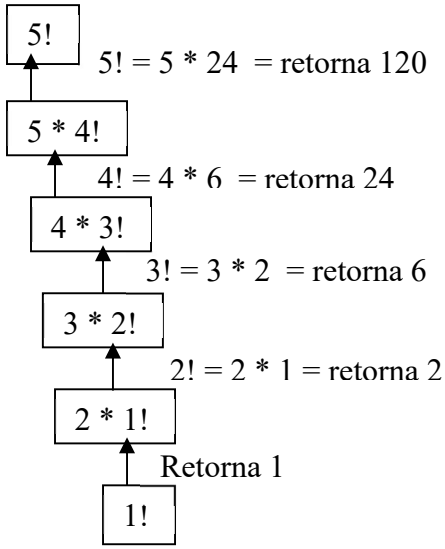
Uma função é recursiva quando, dentro do seu código, chama a si própria.
Considere o fatorial de um inteiro positivo n, escrito n! Por exemplo:

5! = 5 * 4 * 3 * 2 * 1 = 120
O cálculo de 5! pode ser escrito da seguinte forma: 5 * 4!
O cálculo de 4! pode ser escrito da seguinte forma: 4 * 3!
E assim sucessivamente até 1! que é igual a 1.

Ou seja, podemos utilizar a recursividade para resolver o cálculo do fatorial de n, conforme código abaixo:

```
   long fatorial (long numero)
 // long= -9.223.372.036.854.775.808 (-2^63 ate 2^63-1)
   {
     if (numero<=1)
        return 1;
        else
        return numero * fatorial(numero-1);
   }
   void exibeFatorial()    {
     printf("\nCalcula fatorial de 0 a 9");
     for (int i=0;i<10;i++)
          printf("\n %d!   =    %d",  i,  fatorial(i) );
   }
   main()  {
         exibeFatorial();
}
```





A execução do programa fornecerá o seguinte resultado:

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
```

Considerações sobre utilizar iteração ou recursão: QQ problema que pode ser resolvido recursivamente também o pode ser iterativamente (não-recursivo). Uma abordagem recursiva é preferida quando espelha mais naturalmente o problema e resulta em um programa mais fácil de entender. Uma abordagem recursiva pode ser frequentemente implementada com menos linhas de código. Mas não se esqueça: a recursividade invoca sucessivas chamadas do mesmo método/função e sucessivos empilhamentos na execução do programa. A conseqüência (recursão) pode ser cara tanto em termos de tempo do processador como de espaço de memória, mas em compensação, o resultado (código) é muito mais elegante ☺!

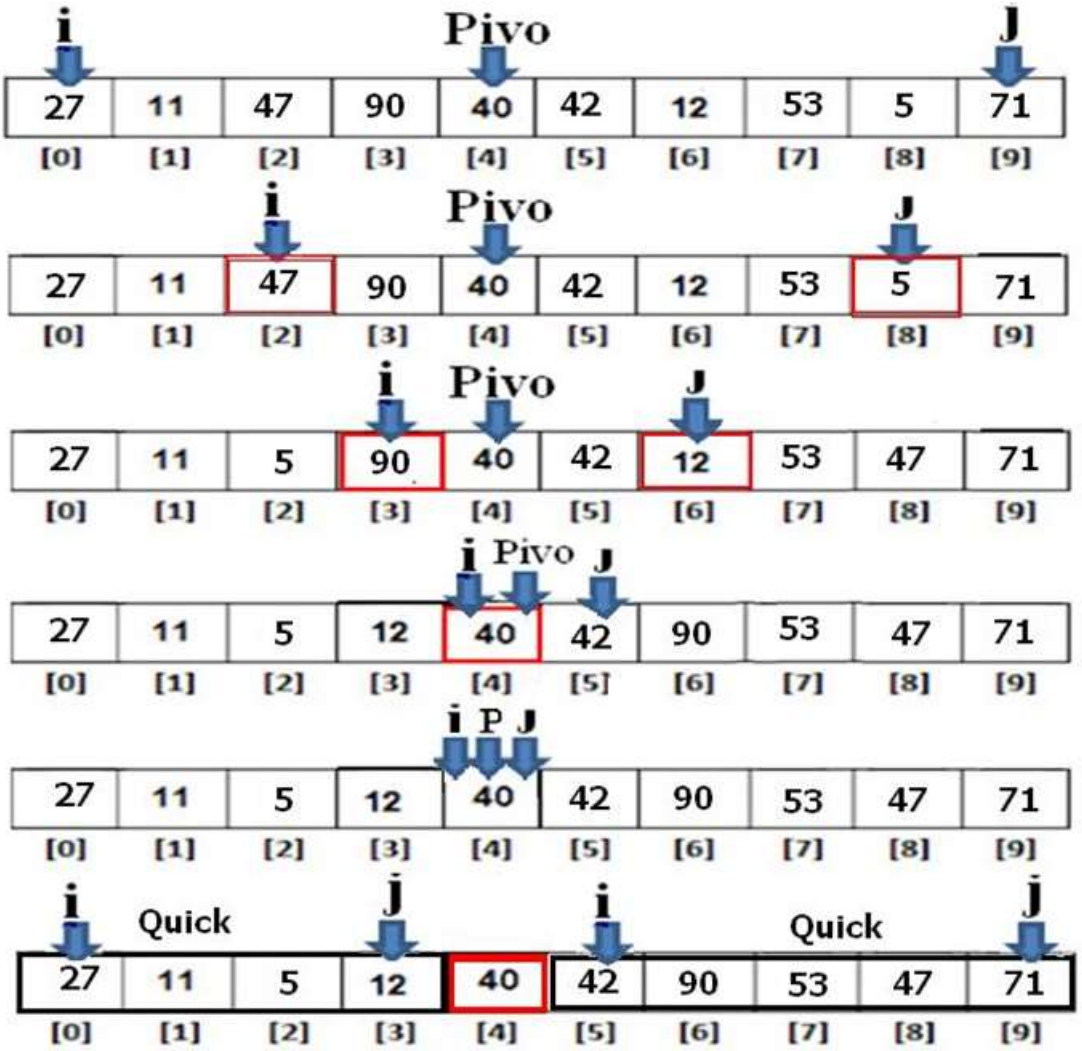
REFAÇA a busca binária recursiva para retornar com a posição ideal de inserção

2.2.5 QuickSort

Este método se baseia no paradigma de divisão e conquista e foi inventado por Hoare. O seu desempenho é o melhor, na maioria das vezes.

a) escolha um pivô (por exemplo o elemento do meio do array)
b) "arrume" o array de tal forma:
   - ANTES do pivô devem ficar os elementos menores que o pivô;
   - APÓS o pivô, deverão ficar apenas os elementos maiores que o pivô;
   - esta "arrumação" implica, algumas vezes, na mudança de posição do pivô dentro do array (nem sempre). Ao final desta etapa, o pivô estará na sua posição ideal dentro do array.

c) aplique a mesma lógica aos dois subarrays (da esquerda e da direita), o que implica num processo recursivo.

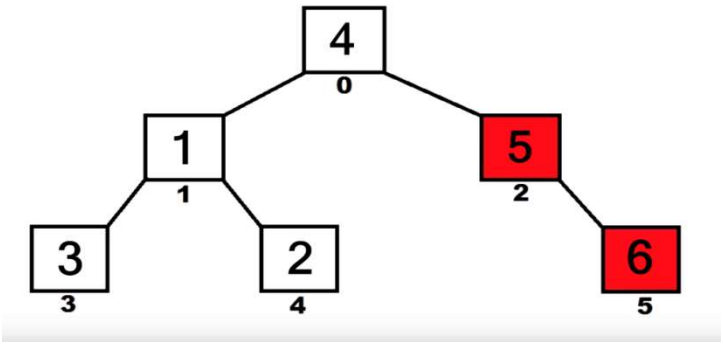


```
void quickSort(int[]a, int inicio, int fim) {
    int aux, i, j, pivo;
    i=inicio;    j=fim;
    pivo=a[(i+j)/2];
    do {
      while(a[i]<pivo) i++;
      while(a[j]>pivo) j--;
      if(i<=j)  {
        aux=vetor[i];
        vetor[i]=vetor[j];
        vetor[j]=aux;
        i++;
        j--;
      }
    } while(i<=j);
    if (j>inicio) quickSort(a,inicio,j);
    if (i<fim) quickSort (a,i,fim);
}
```

```
public void startaQuick()
{
  quickSort(vetor,0,tl-1);
}
```

2.2.6 HeapSort - árvore binária quase completa

É um método avançado de algoritmos de ordenação por seleção. Foi desenvolvido em 1964 por Robert W. Floyd e J.W.J Williams. É um ótimo ordenador de dados tendo consumo de memória bastante reduzido.
Utiliza uma técnica de estruturação de dados chamada HEAP (árvore binária quase completa)
Técnica MAX-HEAP: o nó pai é sempre maior do que qualquer um dos seus filhos
Exemplo: 4 1 5 3 2 6




```
void heapsort(int a[], int n) {
   int i = n / 2, pai, filho, t;
   while(true) {
      if (i > 0) {
          i--;
          t = a[i];
      } else {
          n--;
          if (n <= 0) return;
          t = a[n];
          a[n] = a[0];
      }
      pai = i;
      filho = i * 2 + 1;
```

```
        while (filho < n) {
            if ((filho + 1 < n)  &&  (a[filho + 1] > a[filho]))
                filho++;
            if (a[filho] > t) {
               a[pai] = a[filho];
               pai = filho;
               filho = pai * 2 + 1;
            } else {
               break;
            }
        }
        a[pai] = t;
    }
}
```

RESUMO

- ✓ Os algoritmos de ordenação são de extrema importância para a eficiência de muitos aplicativos, os quais necessitam da rapidez no acesso às informações.
- ✓ O quicksort é considerado como o mais veloz, em média, dos algoritmos conhecidos, mas mais complexo de se implementar, se comparado com os outros apresentados.

3. COMPLEXIDADE DE ALGORITMOS

O objetivo da análise de complexidade é verificar o desempenho do algoritmo em função de grandes quantidades de dados (n grande).
Esta análise pode ser realizada de 2 (duas) formas: análise teórica e análise assintótica:

3.1 ANÁLISE TEÓRICA

- Uma boa idéia é estimar a eficiência de um algoritmo em função do tamanho do problema
- Em geral, assume-se que "n" é o tamanho do problema, ou número de elementos que serão processados
- E calcula-se o número de operações que serão realizadas sobre os n elementos

Na linguagem C, é preciso calcular o tempo gasto por cada método, através da diferença entre o tempo inicial e o tempo final, na execução do algoritmo.

```
#include<time.h>
time_t tempoi, tempof;
double diferenca = difftime(tempof,tempoi);
//diferença em segundos
```

Observação: para comparar dois algoritmos, deve ser utilizado o mesmo ambiente de hardware e software

3.2 ANÁLISE ASSINTÓTICA

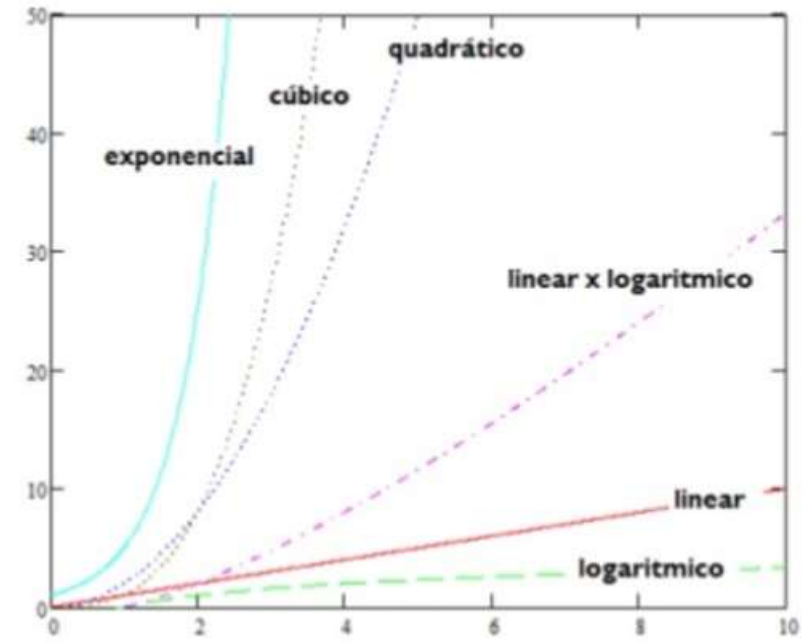
Existe uma simbologia apropriada

- **Ω Ômega (limite inferior)** tempo mínimo gasto
- **O *big Oh* (limite superior) pior caso** (adota-se este!)
- **θ teta (limite restrito**) quando superior=inferior terão o mesmo grau

Geralmente, algoritmos são medidos pela complexidade do pior caso O *big Oh*

3.3 CLASSE-PROBLEMA E SUAS COMPLEXIDADES

CLASSE problema	
O(1)	**Complexidade constante.** As instruções são executadas um número fixo de vezes (raro)
Log n	**Complexidade Logarítmica**. Algoritmos que resolvem um problema transformando-o em problemas menores. (muito rápido – ótimo)
N	**Complexidade Linear**. Cada vez que N dobra de tamanho o tempo de execução dobra (muito rápido – ótimo)
n Log n	**Logarítmico x Linear**. Um problema é resolvido quebrando-o em problemas menores, resolvendo cada um deles independentemente e depois ajuntando as soluções. (algoritmos divisão e conquista)
$\mathbf{n^2}$	**Complexidade quadrática**. Os Itens são processados aos pares., muitas vezes um anel dentro do outro. São úteis para resolver problemas pequenos.
$\mathbf{n^3}$	**Complexidade cúbica**. São úteis para resolver pequenos problemas. Se n é 100, o numero de operações é 1 milhão. Sempre que n dobra, o tempo de execução fica multiplicado por 8
$\mathbf{n^k}$	**Complexidade polinomial.** Freqüentemente de baixa ordem ($k \leq 10$), eficiente. (exemplo 2^n é mais rápido que n^5)
$\mathbf{a^n}$	**Complexidade exponencial**. Algoritmos de Força Bruta. Geralmente não são úteis sob o ponto de vista prático. 2^n, n!, n^n Intratáveis. Ele é tão difícil que não existe um algoritmo polinomial para resolvê-lo



Comparação de curvas no mesmo gráfico. Eixo Y limitado a 50
Eixo X = tamanho (n) e Eixo Y = tempo

Exponencial = 2^n (força bruta)
Cúbico = n^3 (apenas pequenos problemas)
Quadrático = n^2 (um for dentro do outro) parábola
Linear x Logarítmico = n $\log_2$ n (resolve quebrando em problemas menores e depois junta as soluções)
Linear = n (pequeno trabalho sobre cada item de entrada) um for sobre um vetor de n
Logarítmico = $\log_2$ n (resolve um problema transformando-o em problemas menores)

3.6 REGRAS PARA CÁLCULO DA COMPLEXIDADE DE UM ALGORITMO

Supondo que operações simples demoram uma unidade de tempo para executar, considere o programa abaixo para calcular $\sum i^3$ (Somatória i=1 até n)

```
main(){
      int soma=0; // 1 unidade de tempo
      for (int i=1;  i<=n; i++) // n+1 unidades de tempo para testar i<=n
              soma = soma + (i*i*i); // 4 unidades de tempo
                                     // (1 soma, 2 multiplicações e 1 atribuição)
      printf("%d",soma); // 1 unidade de tempo
}
```

Custo = 4 * (n+1) + 2
= 4n + 4 + 2
Custo Total = 4n + 6 = O(n) Linear

- Observa-se que o importante é a repetição do comando for (i=1; i<=n;i++)
- Desconsidera-se as constantes e valores menores para o cálculo
- Em geral, não consideramos os termos de ordem inferior da complexidade do algoritmo, apenas o termo predominante.

2º. Exemplo: Suponha Custo = $3n^2+100n$
100n tem um peso grande, mas a partir de n=11, é o termo $\mathbf{n^2}$ que afeta no crescimento da função: uma parábola.
A constante 3 também tem uma influência irrelevante. Por isso dizemos que este algoritmo é da ordem de $\mathbf{n^2}$ ou que tem complexidade $\mathbf{O(n^2)}$

Repetições
O tempo de execução de uma repetição é o tempo dos comandos dentro da repetição (incluindo testes) vezes o número de vezes que é executada

Repetições aninhadas: A análise é feita de dentro para fora.
O tempo total de comandos dentro de um grupo de repetições aninhadas é o tempo de execução dos comandos multiplicado pelo produto do tamanho de todas as repetições
O exemplo abaixo é **O(n^2)**

```
for (i = 0; i< n; i++)
    for (j = 0; j < n; j++)
        k = k+1;
```

Blocos de comandos consecutivos

- É a soma dos tempos de cada um bloco, o que pode significar o máximo entre eles
- O exemplo abaixo é O(n^2), apesar da primeira repetição ser O(n)

```
for (i = 0; i < n; i++)  // primeiro for o custo é n
        k = 0;
for (i = 0; i< n; i++) // segundo bloco de for aninhados o custo é n^2
    for (j = 0; j < n; j++)
        k = k+1;
```

Comando de Seleção
Para uma cláusula condicional, considere **o maior tempo executado referente aos comandos do então e aos comandos relativos ao senão (else)**
O exemplo abaixo **é O(n)**

```
se i < j
        i = i+1;
        else {
            for (k = 1; k<=n; k++)  // maior custo, ou seja, n
                i = i*k;
           }
```

EXERCÍCIO: Estime quantas unidades de tempo são necessárias para rodar o algoritmo abaixo

```
main(){
    int i = 0, j;
    int a[n];
    while (i<=n) {
        a[i]=0;
        i++;
    }
    for (i = 1; i<=n; i++)
        for (j=1; j <=n; j++)
            a[i] = a[i]+i+j;
}
```

3.4 ANÁLISE ASSINTÓTICA DOS MÉTODOS DE PESQUISA E ORDENAÇÃO

BUSCAS		
Busca Exaustiva (vetor desordenado)	**O (n)**	Pesquisa elemento por elemento, de modo que a função do tempo em relação ao número de elementos é linear, ou seja, cresce proporcionalmente
Busca Sequencial	**O (n)**	Vetor ordenado
Busca Binária (divisão e conquista)	**O (log n)**	Vetor tem que estar ordenado

Métodos de Ordenação

SIMPLES	Bolha	O($\mathbf{n^2}$) quadrático	percorrer o vetor diversas vezes, e a cada passagem faz flutuar para o final o maior elemento da sequência
	Seleção simples	O($\mathbf{n^2}$) quadrático	Não é estável Interessante até 1000 itens
	Inserção Direta	O($\mathbf{n^2}$) quadrático	Usar quando está quase ordenado – estável Ideal para n pequeno
Eficientes	**ShellSort** (inserção)	**O(n (n log n)^2)**	eficiente para até 10.000 itens Usa pivô, mas não é estável
	HeapSort (seleção)	**O (n log n)**	Usa Fila de prioridades Melhor que o ShellSort Não é estável
	QuickSort (divisão e conquista)	**O (n log n)**	Não é estável se o pivô for desequilibrado (O (n^2)) Preferido maioria aplicações 2 vezes mais rápido que o HEAP
	MergeSort (dividir para conquistar)	**O (n log n)**	Estável Usa recursividade Criado por Von Newmann

1º. TRABALHO

1) (ordenação) O método da INSERÇÃO tinha como característica: 1º) remanejar/reposicionar os dois primeiros elementos de forma ordenada no arranjo; 2º) buscar a posição ideal para inserir o 3º. elemento entre os dois primeiros elementos já ordenados e executar esta inserção de tal forma que os três primeiros fiquem ordenados; 3º.) e assim sucessivamente entre os demais elementos. Suponha que existam os seguintes elementos no arranjo:
{30,56,20,15,16,24,40};
Assinale a alternativa que mostra, passo a passo, a troca de elementos no arranjo durante a ordenação pelo método da INSERÇÃO.

(a) {30,56,20,15,16,24,40};
{15,56,20,30,16,24,40};
{15,16,20,30,56,24,40};
{15,16,20,24,56,30,40};
{15,16,20,24,30,56,40};
{15,16,20,24,30,40,56}

(b) {30,56,20,15,16,24,40};
{20,30,56,15,16,24,40};
{15,20,30,56,16,24,40};
{15,16,20,30,56,24,40};
{15,16,20,24,30,56,40};
{15,16,20,24,30,40,56}

(c) Nenhuma das anteriores

2) (ordenação) O método da SELEÇÂO tinha como característica: 1º) buscar o menor elemento do arranjo e trocá-lo com o primeiro elemento (posição zero); 2º.) buscar o segundo menor elemento do arranjo e trocá-lo com o segundo elemento (posição um) 3º.) e assim sucessivamente para os demais elementos. Suponha que existam os seguintes elementos no arranjo:
{30,56,20,15,16,24,40};
Assinale a alternativa que mostra, passo a passo, a troca de elementos no arranjo durante a ordenação pelo método da SELEÇÃO.

a) {30,56,20,15,16,24,40};
{15,56,20,30,16,24,40};
{15,16,20,30,56,24,40};
{15,16,20,24,56,30,40};
{15,16,20,24,30,56,40};
{15,16,20,24,30,40,56}

b) {30,56,20,15,16,24,40};
{20,30,56,15,16,24,40};
{15,20,30,56,16,24,40};
{15,16,20,30,56,24,40};
{15,16,20,24,30,56,40};
{15,16,20,24,30,40,56}

c) Nenhuma das anteriores

2º.TRABALHO

Métodos a serem analisados:

- Bolha
- Seleção
- Inserção
- MergeSort
- QuickSort
- HeapSort

Observar alguns itens solicitados, tais como:

- Número de chaves: 50.000 ,
- 200.000 e
- 500.000

Preencha a tabela com o tempo gasto para cada método (em minutos, segundos e milesegundos)

	50.000 chaves			**200.000 chaves**			**500.000 chaves**		
MÉTODOS	Melhor	Médio	Pior	Melhor	Médio	Pior	Melhor	Médio	Pior
Bolha									
Seleção									
Inserção Direta									
Inserção Binária									
MergeSort									
HeapSort									
QuickSort									

4. ARQUIVOS BINÁRIOS: CONTINUAÇÃO DE ARMAZENAMENTO EXTERNO

Estudamos algumas estruturas de dados cujas informações ficam temporariamente armazenadas em memória principal (vetores, matrizes, listas, listas encadeadas, árvores). A desvantagem na utilização da memória principal é que, ao encerrarmos o programa, todas as informações são automaticamente perdidas.
Agora, estudaremos o armazenamento de dados em memória secundária utilizando arquivo de dados binários (já estudamos arquivos textos – capítulo 1). Abordaremos arquivos binários cujos dados são organizados em blocos contendo bytes contínuos de informação (arquivos de dados não formatados). Estes dados só podem ser acessados via programação.

Modo	Arquivo	Função
"r"	Texto	Leitura. Arquivo deve existir.
"w"	Texto	Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"a"	Texto	Escrita. Os dados serão adicionados no fim do arquivo ("append").
"rb"	Binário	Leitura. Arquivo deve existir.
"wb"	Binário	Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"ab"	Binário	Escrita. Os dados serão adicionados no fim do arquivo ("append").
"r+"	Texto	Leitura/Escrita. O arquivo deve existir e pode ser modificado.
"w+"	Texto	Leitura/Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"a+"	Texto	Leitura/Escrita. Os dados serão adicionados no fim do arquivo ("append").
"r+b"	Binário	Leitura/Escrita. O arquivo deve existir e pode ser modificado.
"w+b"	Binário	Leitura/Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"a+b"	Binário	Leitura/Escrita. Os dados serão adicionados no fim do arquivo ("append").

Exercícios resolvidos

1. **Criando um Arquivo de dados não formatado contendo Registros de Conta de Clientes**. Neste exercício vamos fazer um programa que lê dados do teclado e, em seguida, grava-os no arquivo de dados. Os arquivos de dados gerados só podem ser lidos e gravados via programação.
2. **Lendo um Arquivo de dados não formatado contendo Registros de Conta de Clientes**. O programa lè um arquivo de dados não formatado, e exibe os registros na tela.

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
#include<iostream>
struct conta{
    int num_conta;
    char tipo_conta;
    char nome[30];
    float limite;
    float saldo;
};
//=============================================================
conta leCliente() {   // leitura de um único cliente
    conta Cliente;
    printf("\nEntre com numero da conta:");
    scanf("%d",&Cliente.num_conta);
    printf("\nEntre com tipo da conta (E/C):");
    Cliente.tipo_conta=toupper(getche());
    printf("\nEntre com nome:");
```

```
     gets(Cliente.nome);gets(Cliente.nome);
     printf("\nEntre com limite:");
     scanf("%f",&Cliente.limite);
     printf("\nEntre com saldo:");
     scanf("%f",&Cliente.saldo);
     return (Cliente);
}
void exibe(conta Cliente)  // exibe um único cliente{
    printf("\n----------------------\n");
    printf("Conta: %d\n",Cliente.num_conta);
    if (Cliente.tipo_conta=='E')
        printf("Tipo: Especial \n");
        else printf("Tipo: Comum \n");
    printf("Nome: %s \n",Cliente.nome);
    printf("Limite: %.2f\n",Cliente.limite);
    printf("Saldo: %.2f\n",Cliente.saldo);
    printf("----------------------\n");
}
// faz leitura de cliente  (C comum ou E Especial) e  grava num novo arquivo Clientes.bin
// faz a leitura dos clientes cadastrados e exibe na tela
main(){
   conta Cliente;
   char opcao='S';//deseja continuar
   FILE *arqNovo; // ponteiro para a estrutura predefinida FILE
   arqNovo = fopen("Clientes.bin","wb");//arquivo novo p/gravação apenas
   while (opcao=='S')
   {
       Cliente = leCliente();
       fwrite(&Cliente,sizeof(conta),1,arqNovo);
       strset(Cliente.nome,' ');

       printf("deseja continuar (s/n)?");
       opcao=toupper(getche());
   }
   fclose(arqNovo);
   arqNovo = fopen("Clientes.bin","rb"); // leitura dos clientes cadastrados
   fread(&Cliente, sizeof(conta), 1, arqNovo);
   // feof retorna  diferente de zero se fim
   // feof retorna zero se não for detectado fim
   while (!feof(arqNovo))
   {
       //exibe o registro lido
       exibe(Cliente);
       fread(&Cliente, sizeof(conta), 1, arqNovo);
   }
    fclose (arqNovo);
    system("PAUSE");
}
```

---

3. **Consultando um determ.registro no arquivo de dados criado anteriormente**
   Também exibe os registros gravados no Clientes.bin

```
main()  {
   conta Cliente;
   FILE *arqNovo; // ponteiro para a estrutura predefinida FILE
   arqNovo = fopen("Clientes.bin","r+b");//abre ja existente leitura/gravaçao
   if (arqNovo!= NULL) { // verifica se o arquivo existe e foi localizado
      // exibe todos os registros do arquivo
      fread(&Cliente, sizeof(conta), 1, arqNovo);
      while (!feof(arqNovo))  {
```

```
        //exibe o registro lido
        exibe(Cliente);
        fread(&Cliente, sizeof(conta), 1, arqNovo);
      }
      //.....consulta uma conta
      int num;
      printf("\nEntre com o numero da conta a ser procurada: ");
      scanf("%d",&num);

      fseek(arqNovo,0,SEEK_SET); //posiciona o cursor no inicio do arquivo
      fread(&Cliente, sizeof(conta),1,arqNovo);
      int posicao = 0;

      while ((!feof(arqNovo))&&(num != Cliente.num_conta))   {
        fread(&Cliente, sizeof(conta), 1, arqNovo);
        posicao++;
      }
      if (num == Cliente.num_conta)     {
            printf("encontrou na posicao: %d",posicao);
            fseek(arqNovo, posicao*sizeof(conta),SEEK_SET);
            fread(&Cliente, sizeof(conta),1,arqNovo);
            exibe(Cliente);
      }     else printf("\n Conta inexistente");
      fclose(arqNovo);
   }
}
```

4. **Atualizando um Arquivo de dados não formatado contendo Registros de Conta de Clientes**. Neste exercício deve ser realizado depósitos e retiradas de valores no saldo das contas dos clientes. Para isto, você deve ler o numero da conta a ser atualizada (depósito ou retirada) e fazer uma busca no arquivo. Posicione-se novamente para regravar o registro na mesma posição em que ele foi lido. O posicionamento dos registros, por meio do comando **fseek(posição);** vai de 0 até n-1 (o primeiro registro estará na posição 0 e o último na posição n-1) .
   FAÇA UM MENU NO PROGRAMA PRINCIPAL CONTENDO:
   1 – Ler um cliente e gravá-lo no arquivo
   2 – Exibir os clientes cadastrados
   3 – Buscar de um cliente ao fornecer o número da conta
   4 – Exibir os clientes com conta comum
   5 – Exibir os clientes com conta especial
   6 – Fazer retirada para um determinado cliente ao fornecer o número da conta
   7 – Fazer depósito para um determinado cliente ao fornecer o número da conta
   8 – Totalizar todos os saldos das contas exibindo para cada cliente o nro.conta, o nome cliente e o saldo

4.1 ÍNDICES: USO DE ARQUIVO DE DADOS BINÁRIO E TABELA DE ÍNDICE

Neste exercício, observe que existem 2 (duas) estruturas: um arquivo de dados binário (desordenado) e uma tabela de índice (ordenada).

- um novo funcionário deve ser inserido após o último registro no Arquivo de Dados Binário. Conforme a ilustração abaixo pode-se observar que o arquivo de dados binário encontra-se DESORDENADO e as novas inserções devem ser realizadas após o último registro.

- Tabela de Indice (ordenada) denominada “vetIndice”, cujo TF = 50. O vetor de índice (vetIndice) tem dois campos. São eles: chave e posição. Observe que a chave deve sempre permanecer ordenada crescentemente dentro do vetor. O par (chave e a posição) deve ser inserido de forma

ordenada no vetor de Indice vetIndice. OBSERVE QUE O INDICE DEVE PERMANECER ORDENADO (sempre!).

OBS: utilize o índice para exibir os funcionários de forma ordenada pela chave

Tabela vetIndice

CHAVE	Posição
110	6
210	3
310	8
410	4
510	0
610	5
710	1
810	7
910	2

Arquivo de Dados Binário "funcionario.dat"

	Matrí-cula	nome	depto	salário
0	510	Erivelto	pessoal	$ 500.00
1	710	Geisa	financeiro	$ 2800.00
2	910	José	Administra	$ 8000.00
3	210	Benício	Financeiro	$ 430.00
4	410	Daniela	Pessoal	$ 278.00
5	610	Fernando	Financeiro	$ 5340.00
6	110	Abel	Administra	$ 3000.00
7	810	Hítalo	administra	$ 5420.00
8	310	Cesar	Pessoal	$ 50.00

```
#include<stdio.h>
#include<string.h>
#include<stdbool.h>
#define TF 50
struct partab{
   int chave;
   int posicao;
};
struct funcionario{
    int matricula;
    char nome[10];
    float salario;
    char depto[10];
};

void leTecladoGravaArq(){
   funcionario f;
FILE *arquivo = fopen("Funcionarios.dat","ab");// adiciono novo funcionario no final
if (arquivo==NULL) // pergunto se o arquivo existe
  FILE *arquivo = fopen("Funcionarios.dat","wb"); // cria o arquivo se ele não existir e permite
gravacao fwrite
printf("\nEntre com a matricula:"); scanf("%d",&f.matricula);
printf("\nEntre com o nome.....:");fflush(stdin); gets(f.nome);
printf("\nEntre com o depto....:");fflush(stdin); gets(f.depto);
printf("\nEntre com o salario..:");scanf("%f",&f.salario);
fwrite(&f, sizeof(funcionario),1, arquivo);
printf("\n sucesso ao gravar o novo funcionario f %s",f.nome);
fclose(arquivo);
}
void exibeArquivoDesord(){
funcionario f;
int posicao=0;
```

```
FILE *arquivo = fopen("Funcionarios.dat","rb");
if (arquivo==NULL)
  printf("\n Arquivo nao existe");
  else {
    fread(&f,sizeof(funcionario),1,arquivo);
    while (!feof(arquivo)){
        printf("\nPosicao %d  Matricula%d  Nome%s  Depto%s  Salario %5.2f",
   posicao, f.matricula,f.nome, f.depto,f.salario);
 posicao++;
 fread(&f,sizeof(funcionario),1,arquivo);
 }
  fclose(arquivo);
  }
}
```

void criaIndice(struct partab vetIndice[], int &tl){// opcao 3 do menu para criar a tabela de indice

```
   funcionario f;
   partab par;
   int posicao=0;
   FILE *arquivo = fopen("Funcionarios.dat","rb");
   if (arquivo==NULL)
      printf("\n Arquivo Funcionarios.dat inexistente");
      else {
        fread(&f,sizeof(funcionario),1,arquivo);
  while (!feof(arquivo)){
      par.chave = f.matricula;
 par.posicao=posicao;
 vetIndice[posicao]= par;
      posicao++;
      fread(&f,sizeof(funcionario),1,arquivo);
  }
  fclose(arquivo);
  printf("\n Tabela de Indice vetIndice foi criada mas esta desordenada");
  // a tabela de indice foi criado, no entanto ela esta desordenada
  int i, tam;
        tam=posicao;
        while (tam>1)  {
          for (i=0;i<tam;i++)  {
            if(vetIndice[i].chave>vetIndice[i+1].chave)   {
               partab aux=vetIndice[i];
               vetIndice[i]=vetIndice[i+1];
               vetIndice[i+1]=aux;
            }
          }   //for do i
          tam=tam-1;
        } // while do tam
        // ordenou a tabela de indice pela Bolha.(vetIndice)
        printf("\n Tabela de Indice vetIndice foi ordenada");
        tl = posicao;
}
```

```
}
void exibeIndiceArquivo(struct partab vetIndice[], int tl){//opcao 4 do menu para exibir a
tabela e o arquivo de dados
      funcionario f;
      int posicao=0;
      FILE *arquivo = fopen("Funcionarios.dat","rb");
      if (arquivo==NULL)
        printf("\nArquivo inexistente");
        else {
          printf("\n vetIndice           Arquivo Binario Funcionarios.dat");
          fread(&f,sizeof(funcionario),1,arquivo);
   while (!feof(arquivo)){
      printf("\nCHAVE:%d Posicao:%d        Matricula:%d Nome:%s Depto:%s Salario:%5.2f",
         vetIndice[posicao].chave, vetIndice[posicao].posicao,
 f.matricula,f.nome,f.depto,f.salario);
   posicao++;
   fread(&f,sizeof(funcionario),1,arquivo);
   }
   fclose(arquivo);
}
}
void relatorioOrdenado(struct partab vetIndice[], int tl){
funcionario f;
int posicao,i;
FILE *arquivo = fopen("Funcionarios.dat","rb");
if (arquivo==NULL)
  printf("\nArquivo Inexistente");
  else {
  // relatorio ordenado
  printf("\nRelatorio Ordenado pela Matricula");
  for (i=0;i<tl;i++){
  posicao = vetIndice[i].posicao;
  fseek(arquivo,posicao*sizeof(funcionario),SEEK_SET);
  fread(&f,sizeof(funcionario),1,arquivo);
  printf("\nPosicao %d  Matricula%d  Nome%s  Depto%s  Salario %5.2f",
     posicao, f.matricula,f.nome, f.depto,f.salario);
}
  fclose(arquivo);
  }
}
int buscaBinaria(struct partab vetIndice[], int tl, int matricula){
int inicio, fim, meio;
inicio=0;
fim=tl-1;
meio=(inicio+fim)/2;
while ((matricula != vetIndice[meio].chave)&& (inicio<fim)){
     if (matricula > vetIndice[meio].chave)
          inicio=meio+1;
          else
          fim = meio;
```

```
    meio=(inicio+fim)/2;
 }
 if ((meio<tl)&&(matricula == vetIndice[meio].chave))
    return meio;
    else return -1;
}
void consultaUm(struct partab vetIndice[], int tl){
int matricula;
funcionario f;
int posicao, posdado;
printf("\nEntre com a matricula a ser consultada:");
scanf("%d",&matricula);
posicao = buscaBinaria(vetIndice, tl, matricula);
if (posicao==-1)
   printf("\n Matricula nao existe %d",matricula);
   else {
      FILE *arquivo = fopen("Funcionarios.dat","rb");
      if (arquivo==NULL)
         printf("\n arquivo inexistente");
         else {
            posdado = vetIndice[posicao].posicao;
            fseek(arquivo,posdado*sizeof(funcionario),SEEK_SET);
            fread(&f,sizeof(funcionario),1,arquivo);
            printf("\nPosicao no arq.dados:%d  Matricula%d  Nome%s  Depto%s  Salario %5.2f",
    posdado, f.matricula,f.nome, f.depto,f.salario);
  fclose(arquivo);
}
   }
}
main(){
int opcao=0;
int tl=0;
struct partab vetIndice[30];
while (opcao!=7){
printf("\n\n1 - le do teclado e grava no final do arquivo Binario funcionario.dat");
printf("\n2 - Exibe todos funcionarios do arquivo binario desordenado");
printf("\n3 - Criacao da tabela ordenada vetIndice a partir da leitura do arquivo funcionario.dat");
printf("\n4 - Exibe a tabela vetIndice e o arquivo de dados funcionario.dat");
printf("\n5 - Relatorio ordenado pela matriculas mas contendo todas as informacoes do
funcionario");
printf("\n6 - Consulta um funcionario pela Matricula");
printf("\n7 - Sair");
printf("\nOpcao?");
scanf("%d",&opcao);
switch (opcao){
case 1: leTecladoGravaArq();
        break;
case 2: exibeArquivoDesord();
        break;
case 3: criaIndice(vetIndice,tl);
```

```
        break;
case 4: exibeIndiceArquivo(vetIndice,tl);
        break;
    case 5: relatorioOrdenado(vetIndice, tl);
        break;
case 6: consultaUm(vetIndice,tl);
        break;
}
}
}
```

5.HASHING

Algumas aplicações precisam que as operações de inserção, remoção e consulta seja realizada de maneira bem rápida. Para isto vamos utilizar o HASHING (estrutura de dados em tabela hashing) através de uma função de espalhamento.
Uma função de espalhamento (função hash) **hash(k)** transforma uma chave **k** em um endereço **E**:

E = hash (**k)**

Este endereço é usado como base para o armazenamento e recuperação de registros.
Problema: no espalhamento (hash) duas chaves podem levar ao mesmo endereço (colisão) – portanto, colisões devem ser tratadas.

Exemplo:

Suponha que desejamos armazenar 75 registros, sendo que a chave para os registros é um campo sobrenome. Suponha que foi reservado espaço para manter 1000 registros. Os registros devem ser "espalhados", por exemplo, através do seguinte procedimento: pega-se os dois primeiros caracteres do sobrenome e obtêm suas representações ASCII (código numérico); multiplica estes números e usa-se os três dígitos menos significativos do resultado para servir de endereço.

Exemplo: nome = Maria da Silva
Sobrenome= Silva
Dois primeiros caracteres do sobrenome = Si
(S=83) * (i =105) = 8715
Endereço: 715

Observe que os registros são colocados em posições aparentemente aleatórias, sem considerar a sua ordem alfabética. Entretanto, 2 (dois) sobrenomes diferentes podem produzir o mesmo endereço. Neste caso, ocorreu uma colisão (as chaves são **sinônimas**).

5.1 COMO ESCOLHER UMA FUNÇÃO HASH

O objetivo é que a função "espalhe" as chaves da maneira mais uniforme possível entre os endereços disponíveis. Um dos métodos para criar funções hashing é o **método da divisão**. Uma **chave x** é mapeada em um dos **m endereços** da tabela hashing calculando o resto da divisão de chave x por m, ou seja, a função de hashing é dada por h(x) = x mod m.
Siga os seguintes passos:
a) representação da chave numericamente: Caso a chave não seja numérica, pode ser usado os códigos ASC II dos caracteres (todos) para formar um número.
Por exemplo: LILIANE = 76 73 76 73 65 78 69
b) soma: realize a soma dos números (soma(LILIANE) = 234)

c) divisão do resultado da soma pelo espaço de endereçamento para obter o endereçamento final. O objetivo é reduzir o tamanho do número gerado de forma que ele fique no intervalo de posições endereçáveis do arquivo. Exemplo:

ender = soma % N, onde:

ender = endereço final (resto da divisão – entre 0 e N-1)

soma = é o resultado da soma (passo anterior)

N = é o número de endereços disponíveis

No exemplo da chave "LILIANE", se há 50 posições disponíveis para os registros, então:

Ender => 234 % 50 ➔ 34

5.2 DISTRIBUIÇÃO DE REGISTROS ENTRE ENDEREÇOS

- Função de espalhamento perfeita (ou uniforme): dado um conjunto de chaves, a função HASH gera endereços que não produz colisões (ou seja, não haverá chaves sinônimas).
- Função de espalhamento com o pior caso: qualquer que seja a chave fornecida, a função HASH sempre gera o mesmo endereço (todas as chaves são sinônimas).
- Função aceitável: é aquela que gera poucos sinônimos.

Como resolver o problema das colisões?

Não podemos armazenar dois registros/objetos num mesmo espaço, então precisamos localizar uma posição alternativa para todos os valores depois do primeiro.

5.3 TABELA HASHING IMPLEMENTADA COM ENDEREÇAMENTO ABERTO (TENTATIVA LINEAR)

Todos os registros são armazenados na própria tabela sem a necessidade de espaços extras ou ponteiros. Este método é utilizado quando ó numero de chaves a serem armazenadas é reduzido e as posições vazias na tabela são utilizadas para tratamento de colisões.

No caso de colisão (endereço inicial calculado já está ocupado), deve-se encontrar uma posição livre após o endereço calculado na função Hash. Sucessivas posições são pesquisadas em ordem até que um espaço disponível seja localizado.

Inserção quando ocorrer colisão: O novo registro é gravado na primeira posição disponível, após o endereço calculado pela função Hash. A busca por uma posição livre para armazenar uma chave x será feita através de tentativa linear. Caso a tabela esteja totalmente preenchida, a chave x não pode ser armazenada.

Busca (Recuperação) a chave sofre hash uma vez para determinar a posição inicial e verificar se ela contêm os dados desejados. Se ela contiver, a pesquisa é concluída. Se não contiver, espaços sucessivos são pesquisados linearmente até que os dados desejados sejam localizados.

Remoção: esta operação é delicada, pois não se pode remover de fato uma chave do endereço, pois haveria perda de sequência de tentativas. Com isto, cada endereço da tabela é marcado como livre (L), ocupado (O), ou removido (R).

```
#include <stdio.h>
#include <conio.h>

#define TF 8

struct registro{
        char status; // L-livre ou O-ocupado ou R-removido
        int  chave;
};
```

```
void criar(struct registro p[TF]){
   int i;
   for (i=0; i<TF; i++){
      p[i].status= 'L';
      p[i].chave=0;
   }
   printf("\n tabela Hash enderecamento aberto - criada");
}

int Hash(int chave){
   int ender=0;
   ender = chave % TF;
   return ender;
}
void inserir (struct registro p[TF]){
int chave, ender;
printf("\nDigite a chave:");
scanf("%d",&chave);
ender = Hash(chave);
if (p[ender].status=='L'){
   p[ender].chave=chave;
   p[ender].status='O'; }
else {
       int i=0;
       while (i<TF && p[(ender+i)%TF].status != 'L')
      i = i+1; // i++;
      if (i<TF){
           p[(ender+i)%TF].chave = chave;
          p[(ender+i)%TF].status = 'O';
       } else printf("tabela cheia!");
     }//else
}//inserir
void mostrar_hash(struct registro p[TF]){
int i;
while (i < TF){
printf("\n Chave[%d]:%d  Status:%c",i,p[i].chave,p[i].status);
i++;
}
}
int buscar(int chaveprocurada, struct registro p[TF]){
int i=0;
int ender = Hash(chaveprocurada);
while (i<TF && p[(ender+i)%TF].status != 'L' &&  p[(ender+i)%TF].chave != chaveprocurada)
    i = i + 1;
if (p[(ender+i)%TF].chave == chaveprocurada)
                  return (ender+i)%TF;
      else   return TF; // não encontrou
}
```

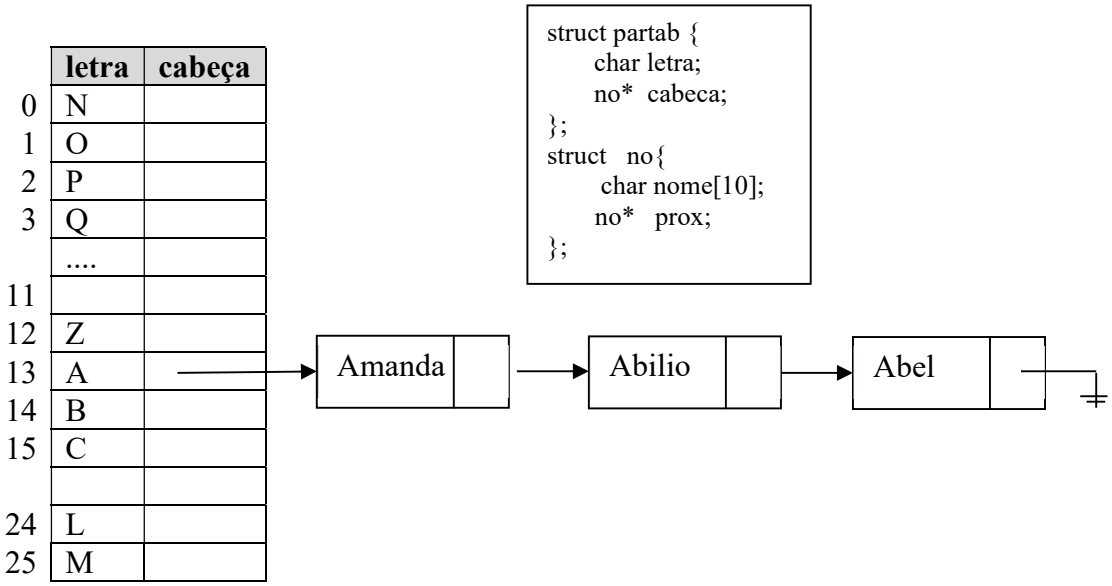
```
void remover(struct registro p[TF]){
int ender, chave;
printf ("Digite a chave a ser removida: ");
    scanf ("%d", &chave);
    ender = buscar(chave,p);
    if (ender < TF){
       p[ender].status='R';
       printf("removeu com sucesso %d",chave);
   } else
   printf("chave não encontrada na tabela");
}

main() {
registro p[TF]; // tabela p tem posicoes de 0 até 7
    criar(p);
    int opcao=0;
    while (opcao!=4){
        printf ("\n");
        printf("\n1 - Inserir Chave");
        printf ("\n2 - Mostrar tabela Hashing");
        printf ("\n3 - Remover chave");
        printf ("\n4 - Sair");
        printf ("\nOpcao?: ");
        scanf ("%d", &opcao);
        switch (opcao){
        case 1:
            inserir(p);
            break;
        case 2:
        mostrar_hash(p);
            break;
        case 3:
    remover(p);
            break;
        }
    }
}
```

5.4 TABELA HASHING IMPLEMENTADA COM LISTA (ENCADEAMENTO)

Neste tipo de implementação, a tabela Hashing é um vetor com m posições, sendo que cada posição possui um ponteiro para uma lista encadeada onde estão contidos todos os elementos que possuem o mesmo endereço mapeado.

EXEMPLO: Faça um programa que apresente o menu de opções abaixo. As opções devem ser implementadas em uma tabela hashing. Utilize como chave da tabela a primeira letra do nome. Suponha que a tabela possua tamanho 26 (observe que temos 26 letras no alfabeto) e que os dados de cada entrada sejam armazenados na forma do método da divisão. Utilize **lista encadeada dinâmica.**

MENU
1 – Inserir apenas o nome
2 – Consultar todas as pessoas
3 – Consultar uma pessoa
4 – Consultar todas as pessoas com uma inicial digitada
5 – Sair



Ex: inserir LILIANE **letra L (código 76) 76 div 26 = 2 resto = 24**
Ex: inserir AMANDA **letra A (código 65) 65 div 26 = 2 resto = 13**

```
#include<stdio.h> #include<string.h> #include<ctype.h>

struct no{
   char nome[15];
   no* prox;
};
struct partab{
   char chave;
   no* cabeca;
};
```

```
void criar(struct pessoa tab[]){
int i,j;
pessoa p;
j=65;
p.chave=(char)65;
p.cabeca=NULL;
i=13;
for (int cont=0;cont<26;cont++){
     tab[i] = p;
     printf("\n pos=%d chave %c  ",i,p.chave);
     j++;
     i++;
     p.chave=(char)j;
     if (i==26) i=0;
}
}//criar
int hash(char nome[]){
int ender = (nome[0] % 26);
return ender;
}
void exibirPessoas(struct pessoa tab[]){
int i;
no *atual;
for(i=0; i<26; i++){
printf("\n(%d) Chave: %c ",i, tab[i].chave);
atual = tab[i].cabeca;
while (atual != NULL){
  printf("\n      Nome: %s",atual->nome);
  atual = atual->prox;
}
}
printf("\n FIM do exibir todos");
}
void inserir(struct pessoa tab[], char nome[]){
int ender=0;
no *novo = new no;
strcpy(novo->nome,nome);
ender = hash(nome);
if (tab[ender].cabeca==NULL){
    novo->prox=NULL;
    tab[ender].cabeca=novo;
}
else {
    novo->prox = tab[ender].cabeca;
    tab[ender].cabeca=novo;
}
printf("\n inseriu no ender %d",ender);
}
```

```
void consultarPessoa(struct pessoa tab[], char nome[]){
no* atual;
int ender = hash(nome);
atual=tab[ender].cabeca;
while ((atual!=NULL)&& (strcmp(atual->nome, nome) !=0))
        atual = atual->prox;

if ((atual !=NULL) && (strcmp(atual->nome,nome)== 0))
printf("\n Nome encontrado!! %s",atual->nome);
else
printf("\n não encontrado");
}
void consultarInicial(struct pessoa tab[], char inicial){
int i;
no*atual;
   int ender = (int)inicial %26;
atual = tab[ender].cabeca;
while (atual != NULL){
  printf("\n      Nome: %s",atual->nome);
  atual = atual->prox;
}
printf("\n\n**********************************\n");
}

main() {    //// 2º. Exercicio    Hash nome[0] *********
int op = 0;
struct pessoa tabela[26];
char inicial;
char nome[15];
criar(tabela);
while(op!=5){
printf("\n1 - Inserir apenas o nome");
printf("\n2 - Consultar a tabela das pessoas");
printf("\n3 - Consultar uma pessoa");
printf("\n4 - Consultar todas as pessoas com uma inicial digitada");
printf("\n5 - Sair\n");
scanf("%d", &op);
switch(op){
case 1:
   strcpy(nome,"");
   printf("\nDigite o nome a inserir: ");getchar();
   scanf("%s", &nome);
   strcpy(nome, strupr(nome));
   inserir(tabela,nome);
break;

case 2: exibirPessoas(tabela);
break;
```

```
case 3:
   strcpy(nome,"");
   printf("\nDigite o nome que esta procurando: ");
   scanf("%s", &nome);
   strcpy(nome, strupr(nome));
   consultarPessoa(tabela,nome);
break;
case 4:
   printf("Qual inicial deseja procurar? ");getchar();
scanf("%c",&inicial);
inicial = toupper(inicial);
printf("\n*** Nomes com a inicial %c ***", inicial);
consultarInicial(tabela,inicial);
break;
}
}
}
```

5.5 EXERCÍCIOS DE HASHING

1) Desenhe um esboço da tabela do 2º. Exercício (tabela hashing iimplementada com lista encadeada). A cada inserção de nome, redesenhe o esboço da tabela. Inserir os nomes: Liliane, Abel, Abilio, Amanda, Zeila, Nair, Zuleika, Noemia.
2) Faça um programa que apresente o menu de opções abaixo. As operações devem ser implementadas em uma tabela hashing. Utilize como chave de tabela, o tipo de produto. Suponha que existam apenas os tipos informados abaixo (em número de 4, use a ordem alfabética para organizar a ordem da tabela hashing) e que os dados de cada entrada da tabela sejam armazenados na forma de uma **lista encadeada.**

MENU
1 – Inserir produto
2 – Consultar todos os produtos cadastrados de um tipo
3 – Contar quantos produtos estão cadastrados em cada tipo
4 – sair

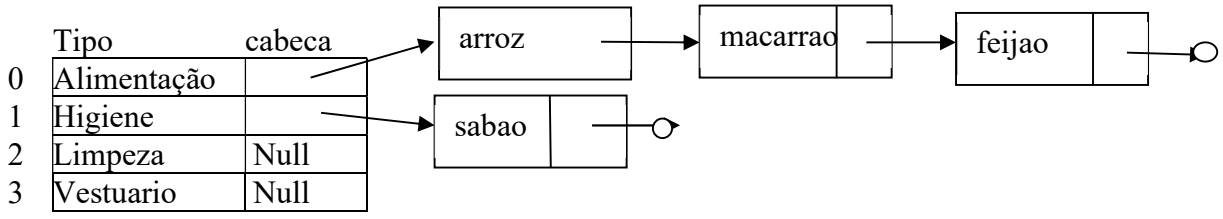
Observações:
- Na opção 1 deve ser inserido um produto de cada vez, sendo que para cada um deles o usuário deve fornecer a descrição e o tipo (A alimentação, H higiene, L limpeza, e V vestuário)
- Na opção 2 o usuário deve digitar a letra que corresponde ao tipo a ser consultado e todos os produtos do tipo fornecido devem ser listados (listar apenas a descrição) Caso não tenha nenhum produto do tipo fornecido, mostrar mensagem.
- Na opção 3 o programa de e mostrar quantos produtos estão cadastrados. Exemplo:

Alimentação – 5 produtos Higiene – 3 produtos

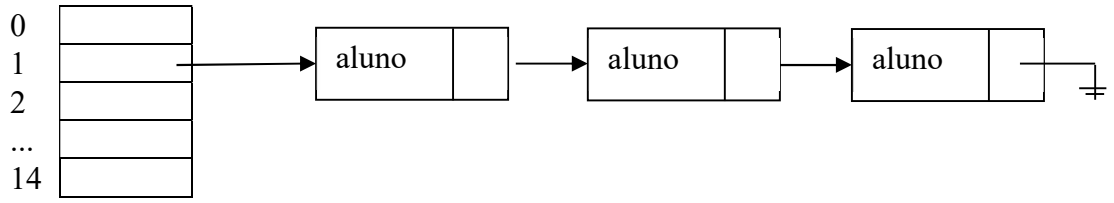
NO

descricao	prox

Limpeza – 0 produtos Vestuário – 2 produtos



3) Faça um programa que apresente o menu de opções abaixo. As operações devem ser implementadas em uma tabela Hashing. Use o **método da divisão**. Como chave da tabela, utilize o código do aluno. Suponha que a tabela possua tamanho 15 e que os dados de cada entrada sejam armazenados na forma de uma lista encadeada dinâmica.
   Ender = Hash (código % 15);
   1 – Cadastrar aluno (código, nome e nota final)
   2 – Consultar aprovados (nota final de no mínimo 7)
   3 – Consultar todos os alunos
   4 - Sair



---

4) Faça um programa que apresente o menu de opções abaixo. As operações devem ser implementadas em uma tabela Hashing. Utilize como chave da tabela o mês de aniversário. Suponha que a tabela possua tamanho 20 e que seja implementada com o método do **endereçamento aberto** (tentativa linear).
   1- Cadastrar um amigo (nome, dia e mês de aniversário)
   2- Consultar aniversariantes de um mês
   3- Contar as pessoas com idade superior a 18
   4- Excluir uma pessoa pelo nome
   5- Excluir as pessoas de um determinado mês
   6- Sair

5) Refaça o exercício 3, utilizando lista encadeada.

6. INVERSÃO

Chave Principal: NUMERO

Chave secundária:
Demais campos da estrutura

	NUMERO	NOME	IDADE	STATUS
0	**1000**	Enio	23	T
1	**1010**	Eber	19	T
2	**1030**	Cleusa	42	T
3	**2000**	Fátima	56	T
4	**2050**	Rui	35	T
5	**2080**	Flavia	19	T
6	**3010**	Diana	27	F
7	**3080**	Maria	23	T
8	**4000**	Antonio	20	T
9	**4080**	Claudia	19	T
10	**4150**	Afonso	35	T
11	**4300**	Marcos	27	T
12	**4700**	Ademar	35	F
13	**4900**	Fabio	42	F
14	**5000**	Vilma	20	T

Inversão por Idade

19	→ 9, 5, 1
20	→ 14, 8
23	→ 7, 0
27	→ 11, 6
35	→ 12, 10, 4
42	→ 13, 2
56	→ 3

A inversão ocorre quando se deseja que a estrutura seja reorganizada por um outro campo (chave secundária).

- Relatório dos funcionários com idade entre 22 e 27 anos (inclusive) Sendo que os nomes dos funcionários devem ser agrupados por idade.

A estrutura (arquivo) não foi projetada para responder, especificamente estas questões. A amplitude é o intervalo que a chave secundária abrange, ou seja, a variação de valor que o campo IDADE pode observar.

Nas inversões apresentadas, os registros são identificados por seus endereços físicos. Isto permite o acesso direto ao registro, após a sua localização na lista invertida.

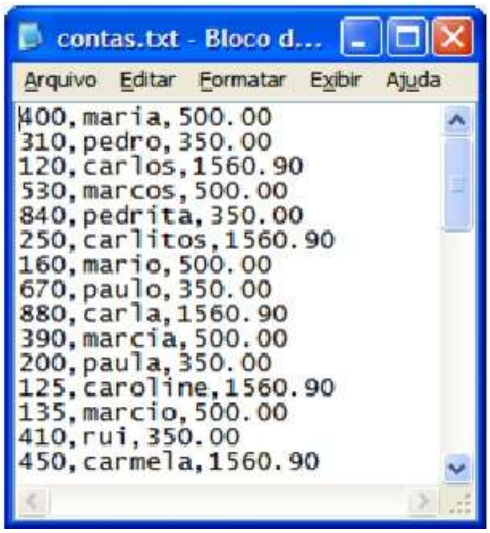
No entanto, acarreta o problema de que as listas são válidas apenas para aquela disposição física dos registros. Na eventualidade do arquivo sofrer uma "reorganização" que envolva mudança nos endereços dos registros, todas as iinversões deverão ser novamente geradas, a partir das nova disposição dos registros.

Implementar:
- Construir inversão por idade
- Relatórios: Todos os funcionários por Idade e Por faixa de Idade (inicial/ final)

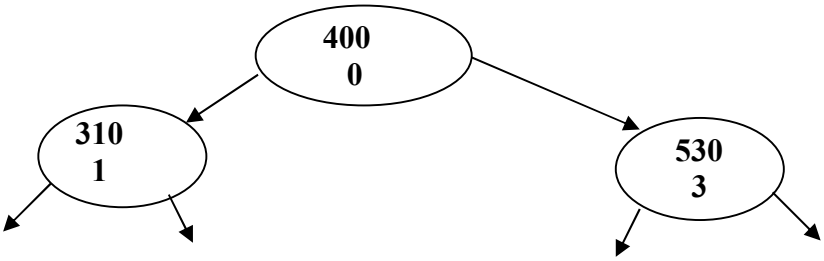
7. INDICES – Utilização de árvore binária como estrutura de apoio no acesso aos dados armazenados em arquivo Binário

Suponha o arquivo de **CLIENTES.dat** conforme descrição abaixo (em disco):

	Código do Cliente	Nome	Saldo
0	400	Maria	500,00
1	310	Pedro	350,00
2	120	Carlos	1560,90
3	530	Marcos	500,00
....	....		



O programa a seguir apresenta uma rotina que cria uma árvore binária como estrutura de apoio (em memória principal), indexada pelo código do cliente. Cada nó da árvore contêm o código do cliente e a posição do registro no arquivo.



O programa também contêm uma rotina de Busca. Nele, a busca de um determinado código do cliente é feito na árvore primeiramente. Ao localizar o código do cliente na árvore, deve ser utilizado a posição do registro no arquivo para a leitura em disco.

EXERCÍCIO: Deseja-se emitir um relatório (que você deve fazer) em ordem crescente pelo código do cliente. Ou seja, o relatório deverá ficar ordenado pelo código do cliente. Devem ser exibidas TODAS as informações de cada cliente. OBS: não pode ser aplicado nenhum método de ordenação e, sim, deve ser utilizada a árvore como estrutura de apoio para emitir o relatório em ordem pelo Código do Cliente.
Suponha que as seguintes classes abaixo já estejam prontas:
- classe NoARV (nó da árvore)
- classe ARVORE
- Classe Registro

```
public class carregaArvore
{
  String nomeArquivo;
  RandomAccessFile arq;
  arvore a;
```

```
  public static void main(String[]args){
      carregaArvore x = new carregaArvore();
while (true) {
 String opcao = JOptionPane.showInputDialog(null,
                                    "Digite a opção desejada:\n"+
 "(C)riação Arvore-leitura arq.Direto\n"+
 "(E)xibe Arvore Em ordem\n"+
 "(B)usca arvore e exibe registro do arquivo\n"+
 "(S)air\n");
opcao = opcao.toUpperCase();
char op = opcao.charAt(0);
switch (op) {
case 'C':
x.criacao();
break;
case 'E':
x.exibe();
break;
case 'B':
x.busca();
break;
case 'S':
System.exit(0);
break;
default:
JOptionPane.showMessageDialog(null, "Opção Inválida!");
}
}
}//main

  public carregaArvore()
  {
  nomeArquivo="clientes.dat"; //arquivo binario SEM serialização de objetos
  }

  public void criacao()   //cria arvore a partir do arquivo Direto com 50 registros
  {
    a = new arvore();//cria arvore binaria vazia
    char nome[] = new char[15];
    char temp;
    int i;
    Registro reg = new Registro();
    try
    {
      arq = new RandomAccessFile("clientes.dat","r");
      // todos os registros lidos que estiverem em branco serao ignorados
      System.out.printf("\n%-10s%-15s%10s\n","Conta","Nome","Saldo");
      try
      {
      int posii=-1;
      while (true)
      {
        do
        {
         reg.setConta(arq.readInt());
         for (i=0;i<nome.length;i++)
         {
            temp = arq.readChar();
            nome[i]=temp;
         }//leitura do nome
```

```
          reg.setNome(new String(nome).replace('\0',' '));
          double saldo = arq.readDouble();
          reg.setSaldo(saldo);
          //System.out.printf("\nsaldo:%10.2f\n",saldo);
         } while (reg.getConta()==0);
         //String cc = String.format("%s",reg.getConta());
         posii++;
         a.insere(reg.getConta(),posii);
         System.out.printf("%-10d%-12s%10.2f\n",
                    reg.getConta(),reg.getNome(),reg.getSaldo());
      }//while
      }
      catch(EOFException fim)
      {
        arq.close();
        JOptionPane.showMessageDialog(null,"fim da leitura");
        return;
      }//fim
      }//try da leitura
    catch (IOException erro)
    {
      System.out.println("erro na leitura do arquivo");
      System.exit(1);
    }//erro
    finally
    {
      try
      {          arq.close();        }
      catch (IOException erro)
      {
       System.out.println("erro ao fechar o arquivo");
       System.exit(1);
      }
    }//finally
}//criacao

public void exibe()
{
   String texto = a.chamaInOrdem();
   JOptionPane.showMessageDialog(null,texto);
}//exibe

public void busca()
{
  int i, conta;
  char nome[] = new char[15];
  double saldo;
  try
  {       arq = new RandomAccessFile(nomeArquivo,"rw");      }
  catch(IOException erro)
  {
   System.out.println("arquivo não existe");
   System.exit(1);
  }
  Scanner entrada  = new Scanner(System.in);
  Registro reg = new Registro();
  try
  {
    String opcao = JOptionPane.showInputDialog(null,"\nDigite uma conta: ");
    conta = Integer.parseInt(opcao);
```

```
      while (conta != 0)
      {
         //String cc = String.format("%s",conta);
         int posicao=a.procura(conta);
         String texto = "\nPosicao: "+posicao+"  conta: "+conta+"\n";
         if (posicao !=-1)
         {
           arq.seek( (posicao) * 42 );
           // inteiro=4 bytes; string=30 bytes; double=8 bytes => total=42 bytes
           reg.setConta(arq.readInt());
           for (i=0;i<15;i++)
           {
            char temp = arq.readChar();
            nome[i]=temp;
           }//leitura do nome
           reg.setNome(new String(nome).replace('\0',' '));
           saldo = arq.readDouble();
           reg.setSaldo(saldo);
           texto += "   Conta: "+reg.getConta()+"   Nome: "+reg.getNome();
           JOptionPane.showMessageDialog(null,texto);
         }
         opcao = JOptionPane.showInputDialog(null,"\nDigite uma conta: ");
         conta = Integer.parseInt(opcao);
      }//while
    }
    catch(IOException erro)
    {
      System.out.println("erro ao gravar no arquivo");
      System.exit(1);
    }
    finally
    {        try{
        arq.close();
      } catch(IOException erro)
      {
       System.out.println("erro fechando o arquivo");
       System.exit(1);
      }
    }
  }//gravacao

}//class ArquivoDireto
```

8. REFERÊNCIAS BIBLIOGRÁFICAS

ASCENCIO, Ana Fernanda Gomes. **Estrutura de dados**: Algoritmos, Análise da Complexidade e Implementações em Java e C C++. Pearson Universidades Ed. 2010.

CORMEN,Thomas H et al. **Algoritmos**: teoria e prática. trad. 2ª.ed. Rio de Janeiro: Campus/Elsevier, 2002.

SCHILDT, H. **Turbo C** - guia do usuário. Editora McGraw-Hill, 1988.**Schildt**, H. C – completo e total. Editora McGraw-Hill, 1990.

SZWARCFITER, J.L. e MARKEZON,L. **Estruturas de dados e seus algoritmos** LTC ed. Rio de Janeiro, 1994.

TANENBAUM, A. **Estruturas de Dados em C**. Ed. Makron, 1997.

ZIVIANI, N. **Projeto de Algoritmos com implementações em Java e C++,** Thomson 2007.