Estruturas de Dados Homogêneos
Vetores e Matrizes

1. Objetivos

- Apresentar os conceitos de Estruturas de Dados Homogêneos Unidimensionais;
- Apresentar os conceitos de Estruturas de Dados Homogêneos Multidimensionais.
- Exemplificar o uso das Estruturas de Dados Homogêneos

2. Vetores (Matrizes Unidimensionais)

Vetores são agrupamentos de determinado objeto em ordem sequencial na memória. Ou seja, é um tipo de dado usado para representar uma certa quantidade de variáveis de mesmo tipo que são referenciadas pelo mesmo nome. Consiste em locações contíguas de memória. O endereço mais baixo corresponde ao primeiro elemento. Em linguagem C, os vetores tem 0 como índice do primeiro elemento, portanto sendo declarado um vetor de 10 elementos, o índice varia de 0 a 9.

Sintaxe:

Tipo_da_variavel nome_da_variavel[tamanho];

Ao se deparar com uma declaração semelhante a essa, a linguagem C reserva na memória um espaço para armazenar a quantidade de valores determinada em tamanho.

Declaramos variáveis normais assim:

int contador1, contador2, contador3, contador4, ..., contadorN;

Agora, podemos declarar essa mesma variável *contador* como um vetor e evitamos "repetições".

int contador[n];

Acima, está um exemplo de declaração de vetores. Os vetores seguem o mesmo princípio da declaração de uma variável normal. O valor entre colchetes, depois do nome, informa quantos elementos a variável terá.

Podemos ler o contador como uma variável normal:

scanf("%d",&contador[0]);

ou

scanf("%d",&contador); / se não for passado o elemento pra ser acessado após o nome da variável o C, entende que é para acessar o primeiro elemento do vetor */*

Um programa que demonstra isso pode ser:

#include <stdio.h>

```
#include <stdlib.h>

int main() {
   int x, conta[3]; /*onde, conta é um vetor para armazenar 3 valores do tipo inteiro*/

   for (x=0;x<=2;x++){ /*Utiliza-se a estrutura de repetição "For" para percorrer um vetor*/
      scanf(" %d", &conta[x]);
      printf("O indice %d contem %d\n", x, conta[x]);
   }
system("Pause");
return(0);
}
```

1.1. Inicializando Elementos em Uma Matriz Unidimensional:

Para inicializar elementos de um vetor, você pode colocar os valores do elemento dentro de abre e fecha colchetes após a declaração do vetor.

Exemplo:

```
int vetor[3] = {1,2,3};
```

Um programa que demonstra isso pode ser:

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

int main(){
   int x, conta[3]={10,11,12};

   for (x=0;x<=2;x++){
      printf("O indice %d contem %d\n", x, conta[x]);
   }
system("Pause");
return(0);
}
```

1.2. Exercício Exemplo:

Escreva um algoritmo que leia um vetor com 50 posições de números inteiros e mostre somente os positivos.

Solução:

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

int main(){
   int vetor[50];
   int i; //Variável para controle do laço for.
```

```
    printf("Digite 50 valores inteiros: ");
    //Lendo os valores que preencherão o vetor.
    for (i=0; i<=49; i++){
        scanf("%d", &vetor[i]);
    }

    //Imprimindo a saída.
    printf("\nForam digitados os seguintes valores positivos: ");
    for (i=0; i<=49; i++){
        if (vetor[i]>0){
            printf("%d ", vetor[i]);
        }
    }

    system("Pause);
    return (0);
}
```

3. Strings

Strings são vetores que armazenam elementos do tipo caractere (char). As strings têm o seu último elemento como um '\0', ou seja, ao percorrer um vetor do tipo strings, sabe-se que a palavra, ou frase, armazenada chegou ao fim quando encontra-se '\0', neste caso, o sistema retorna um valor nulo. A declaração geral para uma string é:

```
char nome_da_string [tamanho];
```

Não é possível igualar duas strings, é necessário que seus elementos sejam igualados um a um, ou seja:

```
string1==string2; /* NÃO faca isto */
```

O código abaixo serve para igualar duas strings (isto é, copia os caracteres de uma string para o vetor da outra):

```
#include <stdio.h>
int main (){
    int count;
    char str1[100],str2[100];
    for (count=0;str1[count];count++)
        str2[count]=str1[count];
    str2[count]='\0';
}
```

A condição no loop for acima é baseada no fato de que a string que está sendo copiada termina em '\0'. Quando o elemento encontrado em str1[count] é o '\0', o valor retornado para o teste condicional é falso (nulo). Desta forma a expressão que vinha sendo verdadeira (não zero) continuamente, torna-se falsa.

3.1. Funções Básicas para Manipulação de Strings.

3.1.1. gets

Função para fazer a leitura de uma strings a partir do teclado, ou seja, utilizada no lugar do "scanf". A declaração geral para a função gets é:

gets (nome_da_string);

O programa abaixo demonstra o funcionamento da função gets():

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

int main (){
   char string[100];
   printf ("Digite o seu nome: ");
   gets (string);
   printf ("\n\n Ola %s",string);

   system("Pause");
   return(0);
}
```

Repare que é válido passar para a função printf() o nome da strings. Como o primeiro argumento da função printf() é uma string também é válido fazer:

*printf (string); //*isto imprimirá somente a string.

3.1.2. strcpy

Função para copiar uma strings-origem para uma strings-destino. Seu funcionamento é semelhante ao exemplo citado inicialmente para cópia entre strings, porém, de forma automática. Sua forma geral é:

strcpy (string_destino, string_origem);

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h> //Cabeçalho que contém a função strcpy

int main (){
   char str1[100],str2[100],str3[100];
   printf ("Entre com uma string: ");
   gets (str1);
   strcpy (str2,str1); /* Copia str1 em str2 */
   strcpy (str3,"Voce digitou a string "); /* Copia "Voce digitou a string" em str3
*/
   printf ("\n\n%s%s",str3,str2);
  system("Pause);
  return(0);
}
```

3.1.3. strcat

Função utilizada para acoplar duas strings, onde a strings-origem permanecerá inalterada e será anexada ao final da strings-destino. Sua forma geral é:

strcat (string_destino,string_origem);

Exemplo:

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

int main (){
   char str1[100],str2[100];
   printf ("Entre com uma string: ");
   gets (str1);
   strcpy (str2,"Voce digitou a string: ");
   strcat (str2,str1);/* str2 armazenará "Voce digitou a strings: " + o conteúdo de str1 */
   printf ("\n\n%s",str2);

   system("Pause");
   return(0);
}
```

3.1.4. strlen

Função utilizada para retornar o comprimento da string fornecida. O terminador nulo não é contado. Isto quer dizer que, de fato, o comprimento do vetor da strings deve ser um a mais que o inteiro retornado por strlen(). Sua forma geral é:

strlen (string);

Exemplo:

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

int main (){
   int size;
   char str[100];
   printf ("Entre com uma string: ");
   gets (str);
   size=strlen (str);
   printf ("\n\nA string que voce digitou tem tamanho %d", size);

  system("Pause");
  return(0);
}
```

3.1.5. strcmp

Função utilizada para comparar a strings 1 com a string 2. Se as duas forem idênticas a função retorna zero. Se elas forem diferentes a função retorna nãozero. Sua forma geral é:

strcmp (string1,string2);

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main (){
   char str1[100],str2[100];
   printf ("Entre com uma string: ");
   gets (str1);
   printf ("\n\nEntre com outra string: ");
   gets (str2);
   if (strcmp(str1,str2))
      printf ("\n\nAs duas strings são diferentes.");
   else
      printf ("\n\nAs duas strings são iguais.");

   system("Pause");
  return(0);
}
```

3.2. Exercício Exemplo:

Faça um programa que leia quatro palavras pelo teclado, e armazene cada palavra em uma string. Depois, concatene todas as strings lidas numa única string. Por fim apresente esta como resultado ao final do programa.

Solução:

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

int main(){
   char str[20],strConcat[80];
   int i; //Variável para controle do laço for.

   strcpy (strConcat," ");//Inicializando a string com vazio
   for (i=0; i<4; i++){
      printf("Digite a palavra: ");
      gets(str);
      strcat (strConcat,str);
   }

   printf("As quatro palavras digitadas foram: %s", strConcat);

   System("Pause");
   return (0);
```

```
}
```

4. Matrizes Multidimensionais

Uma matriz é uma variável que pode armazenar múltiplos valores do mesmo tipo. Em todos os exemplos apresentados até o momento, as matrizes consistiram de uma fileira de dados, ou seja, matrizes unidimensionais, também conhecidas como "Vetores". No entanto, a Linguagem C permite matrizes bi, tri e multidimensionais. O melhor modo de visualizar uma matriz bidimensional é com uma tabela com linhas e colunas. Se uma matriz contém três dimensões, visualize-a como várias páginas, cada uma das quais contendo uma tabela bidimensional.

Por exemplo, ao declarar-se uma matriz bidimensional, o primeiro valor que for especificado informará o número de linhas, e o segundo valor, o número de colunas:

```
int tabela [2] [3];
```

3.1. Inicializando Elementos em Uma Matriz Bidimensional:

Para inicializar elementos de matriz, você pode colocar os valores do elemento dentro de *abre e fecha colchetes* após a declaração da matriz. O comando a seguir usa a mesma técnica para inicializar uma matriz bidimensional. No entanto, neste caso, o comando especifica os valores para cada linha da matriz dentro de chaves:

```
int tabela [2] [3] = {{1,2,3}, {4,5,6}};
```

O compilador inicializará os elementos da matriz como mostrado a seguir:

1 2 3

4 5 6

3.2. Percorrendo em Um Laço Uma Matriz Bidimensional:

Quando os programas trabalham com matrizes bidimensionais, normalmente usa-se duas variáveis para acessar elementos da matriz. O programa a seguir usa as variáveis linha e coluna para exibir os valores contidos dentro da matriz tabela:

```
#include <stdio.h>
void main(void) {
   int linha, coluna;
   float tabela[3][5] = {{1.0, 2.0, 3.0, 4.0, 5.0}, {6.0, 7.0, 8.0, 9.0, 10.0},
{11.0, 12.0, 13.0, 14.0, 15.0}};

   for (linha = 0; linha < 3; linha++) {
      for (coluna = 0; coluna < 5; coluna++) {
         printf("tabela[%d][%d] = %f\n", linha, coluna, tabela[linha][coluna]);
      }
   }
}
```

Colocando laços for um dentro do outro, como mostrado, o programa exibirá os elementos contidos na primeira linha da matriz (1.0 até 5.0). Em seguida, o programa

irá se mover para a próxima linha, e, depois, para a terceira linha, exibindo os elementos dentro de cada linha.

3.3. Passando uma Matriz Bidimensional Para uma Função:

Ao passar matrizes para uma função, não será necessário especificar o número de elementos na matriz. Em matrizes bidimensionais, não será necessário especificar o número de linha na matriz, mas, sim, especificar o número de colunas. O programa a seguir usa a função *exibe_2d_matriz* para exibir o conteúdo de variáveis matrizes bidimensionais:

```
#include <stdio.h>
void exibe_2d_matriz(int matriz[][10], int linhas)
{
   int i, j;
   for (i = 0; i < linhas; i++){
      for (j = 0; j < 10; j++)}
            printf("matriz[%d][%d] = %d\n", i, j, matriz[i][j]);
      }
   }
}

void main(void)
{
   int a[1][10] = {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}};
   int b[2][10] = {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, {11, 12, 13, 14, 15, 16, 17,
18, 19,20}};
   int c[3][10] = {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, {11, 12, 13, 14, 15, 16, 17,
18, 19, 20}, {21, 22, 23, 24, 25, 26, 27, 28, 29, 30}};

   exibe_2d_matriz(a, 1);
   exibe_2d_matriz(b, 2);
   exibe_2d_matriz(c, 3);
}
```

5. Sistemas de Busca:

5.1. Busca Linear

Um problema trivial é determinar a posição de um elemento em um vetor, ou determinar que ele não está presente. Quando temos um vetor sem ordenação usamos o algoritmo de busca linear, que verifica todos os elementos do vetor sequencialmente, até encontrar o elemento procurado ou até ter percorrido todo o vetor. Exemplo de busca linear:

```
#include <stdio.h>

main() {
   int i,p,N,ra[200];
   int procurado = 45678.
```

```
/* preenche ra[] com a lista de matriculados em MC102E e coloca o número de
matriculados (que será menor que 200) em N */
    ConsultaMatriculados("MC102E",ra,&N);

    p = -1; /* ainda nao achei */
    for(i=0;i<N;i++) {
       if (ra[i] == procurado) {
              p = i; /* achei na posicao i */
              break; /* saio do for, pois j´a o encontrei*/
       }
    }

    if (p<0){
       printf("n~ao encontrei :-(\n");
    } else {
       printf("encontrei, era o %d-ésimo da lista.\n",p+1);
/* note p+1: em português contamos a partir do 1 */
    }
```

A busca linear leva, no pior caso, tempo proporcional ao tamanho N do vetor. No melhor caso, leva tempo constante (podemos dar sorte e o elemento procurado ser o primeiro).

5.2. Busca Binária

É possível buscar algo sem verificar todos os elementos? Se o vetor estiver ordenado, sim. Quando temos elementos de algum domínio ordenável, podemos realizar busca em tempo proporcional a $log_2 N$. Números inteiros e reais, e nomes alfabéticos são ordenáveis; vetores multidimensionais, matrizes, cores, números complexos, imagens não são. O algoritmo de busca binária é semelhante ao que usamos para buscar um nome na lista telefônica ou uma palavra no dicionário: em vez de procurar linearmente página por página a partir da primeira, fazemos uma estimativa (o bom e velho chute) de onde o elemento procurado deve estar (palavras iniciadas por X ficam no fim do dicionário...) e abrimos o livro naquela posição. Se as palavras da página aberta estiverem após o elemento procurado, descartamos as páginas localizadas após o nosso chute inicial e repetimos a busca no trecho entre a primeira página e o local da primeira estimativa.

Quando não temos informação sobre os dados do vetor, uma boa estimativa é pegar o elemento do meio do vetor, *V[N/2]*. Se o elemento procurado, x, estiver antes de *V[N/2]*, repetimos a busca somente no vetor *V[0] . . . V[N/2]*. O número de vezes que conseguimos dividir algo por 2 até chegar em vetores de 1 elemento ´e $log_2 N$, portanto a busca binária leva tempo proporcional a $log_2 N$. O programa abaixo implementa a busca binária:

```
#include <stdio.h>
#define N 400

main() {
   int V[N],e,d,m; /* e=esquerda, d=direita, m=meio */
   int procurado = 313, pos=-1;
```

```
    leia_os_valores_ordenados_de_algum_lugar(V);

    e=0; d=N;
    do {
       m = (e+d)/2; // elemento do meio

       if (V[m]==procurado) { // encontrei ?
              pos = m;
              break;
       }

       if (procurado < V[m]) {
              d=m;
       } else {
              e=m+1;
       }
    } while(e<d);

    if (pos>=0) {
       printf("Encontrei %d na posi¸c~ao %d.\n",procurado,pos);
    } else {
       printf("N~ao encontrei %d.\n",procurado);
    }
}
```

6. Ordenação:

O algoritmo mais trivial para ordenar um vetor é gerar todas as *N!* permutações e escolher uma das permutações que esteja ordenada. Naturalmente ninguém faz isso, por enquanto vamos nos contentar com algoritmos que ordenam um vetor fazendo em torno de *N2* operações de troca e comparação (o que já é bem melhor que *N!*).

6.1. Bubble Sort (Método da Bolha)

A ordenação por bolha (*bubble sort*) simula o processo de bolhas gás em um líquido: as bolhas trocam de posição com o líquido lentamente, até que o equilíbrio (gás menos denso escapando para cima, líquido permanece abaixo) é atingido.

No *bubble sort* realizamos *N−1* "rodadas" de trocas entre elementos adjacentes. Cada rodada garante que um elemento a mais já está na sua posição correta, portanto cada rodada subsequente tem um elemento a menos de comprimento. O algoritmo do *bubble sort*, em C, é dado abaixo (nos exemplos de algoritmos de ordenação, para maior clareza, estamos assumindo que *N* está definido em algum lugar e que o vetor V foi devidamente preenchido).

```
/* bubble sort */
int i,j,V[N],t;

for(i=N-1;i>0;i--) {
   for(j=0;j<i;j++) {
```

```
        if (V[j] > V[j+1]) {
                t = V[j];
                V[j] = V[j+1];
                V[j+1] = t;
        }
    }
}
```

6.2. Selection Sort (Seleção)

O algoritmo de ordenação por seleção realiza *N−1* rodadas de troca, mas em cada rodada fazemos apenas uma troca, e a troca realizada garante que um elemento é colocado na sua posição final no vetor ordenado. Dado um vetor de *N* posições, selecionamos o menor elemento (através de busca linear, mantendo variáveis com o menor valor encontrado até o momento e seu índice) e trocamos o menor elemento com o elemento no índice 0. Repetimos o procedimento para o subvetor *1..N*, e assim sucessivamente. O algoritmo de *selection sort* em C é dado abaixo.

```
/* selection sort */
int i,j,V[N],min,minpos;

for(i=0;i<N-1;i++) {
    min = V[i];
    minpos = i;
    /* procura minimo do subvetor i...N-1 */
    for(j=i+1;j<N;j++)
        if (V[j] < min) {
                min = V[j];
                minpos = j;
        }
    /* troca V[i] com V[minpos] (=min) */
    V[minpos] = V[i];
    V[i] = min;
}
```

6.3. Insertion Sort (Inserção)

O algoritmo de ordenação por inserção é baseado na ideia de que é muito mais simples manter organizado algo que já está arrumado do que ordenar uma grande bagunça. No *insertion sort* você começa com um vetor ordenado de 1 elemento (trivial: ele sempre está ordenado), e insere os demais elementos colocando-os na posição correta. Este algoritmo é semelhante ao que fazemos intuitivamente para manter cartas de baralho ordenadas na mão quando recebemos uma carta por vez em jogos de buraco, canastra ou pôquer. Sempre temos dois grupos de elementos: os já ordenados no início do vetor e os não ordenados no final. Realizamos *N−1* rodadas. Em cada rodada, um elemento não ordenado é deslocado para a esquerda (através de trocas com seus vizinhos) até que chegue na sua posição correta dentro do vetor ordenado. O algoritmo do *insertion sort*, em C, é dado abaixo.

```
/* insertion sort */
int i,j,t,V[N];

for(i=1;i<N;i++) {
   j = i;
   /* arrasta V[i] para a esquerda até chegar na posição ordenada */
   while( (j>0) && (V[j]<V[j-1]) ) {
      t = V[j];
      V[j] = V[j-1];
      V[j-1] = t;
      j--;
   }
}
```

Observação: o loop interno poderia perfeitamente ser escrito como

```
   for(j=i;(j>0) && (V[j]<V[j-1]);j--) {
      t = V[j];
      V[j] = V[j-1];
      V[j-1] = t;
   }
```

Pode parecer estranho para quem está acostumado com o loop for mais simples de outras linguagens, mas é perfeitamente válido em C.

Estes três algoritmos de ordenação levam tempo proporcional a N^2. Os melhores algoritmos gerais para ordenação conseguem tempo proporcional a *NlogN*. Em algumas situações especiais (intervalos pequenos e limitados de valores, por exemplo) conseguimos realizar ordenação em tempo linear (proporcional a N).