6. Listas Encadeadas

Segundo Celes e Rangel (2002), para representar um grupo de dados, é possível usar um vetor em C. O vetor é a forma mais primitiva de representar diversos elementos agrupados. Para simplificar a discussão dos conceitos que serão apresentados agora, vamos supor que temos que desenvolver uma aplicação que deve representar um grupo de valores inteiros. Para tanto, podemos declarar um vetor escolhendo um número máximo de elementos.

```
#define MAX 1000
int vet[MAX];
```

Ao declararmos um vetor, reservamos um espaço contíguo de memória para armazenar seus elementos, conforme ilustra a figura abaixo.

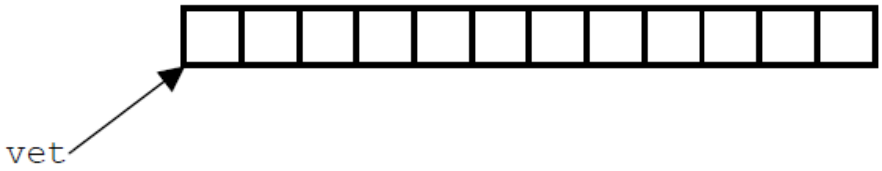


Figura 6.1: Um vetor ocupa um espaço contíguo de memória, permitindo que qualquer elemento seja acessado indexando-se o ponteiro para o primeiro elemento.

O fato de o vetor ocupar um espaço contíguo na memória nos permite acessar qualquer um de seus elementos a partir do ponteiro para o primeiro elemento. De fato, o símbolo *vet*, após a declaração acima, como já vimos, representa um ponteiro para o primeiro elemento do vetor, isto é, o valor de vet é o endereço da memória onde o primeiro elemento do vetor está armazenado. De posse do ponteiro para o primeiro elemento, podemos acessar qualquer elemento do vetor através do operador de indexação vet[i]. Dizemos que o vetor é uma estrutura que possibilita acesso randômico aos elementos, pois podemos acessar qualquer elemento aleatoriamente.

No entanto, o vetor não é uma estrutura de dados muito flexível, pois precisamos dimensioná-lo com um número máximo de elementos. Se o número de elementos que precisarmos armazenar exceder a dimensão do vetor, teremos um problema, pois não existe uma maneira simples e barata (computacionalmente) para alterarmos a dimensão do vetor em tempo de execução. Por outro lado, se o número de elementos que precisarmos armazenar no vetor for muito inferior à sua dimensão, estaremos subutilizando o espaço de memória reservado.

A solução para esses problemas é utilizar estruturas de dados que cresçam à medida que precisarmos armazenar novos elementos (e diminuam à medida que precisarmos retirar elementos armazenados anteriormente). Tais estruturas são chamadas *dinâmicas* e armazenam cada um dos seus elementos usando alocação dinâmica.

Nas seções a seguir, discutiremos a estrutura de dados conhecida como *lista encadeada*. As listas encadeadas são amplamente usadas para implementar diversas

outras estruturas de dados com semânticas próprias, que serão tratadas nas seções seguintes.

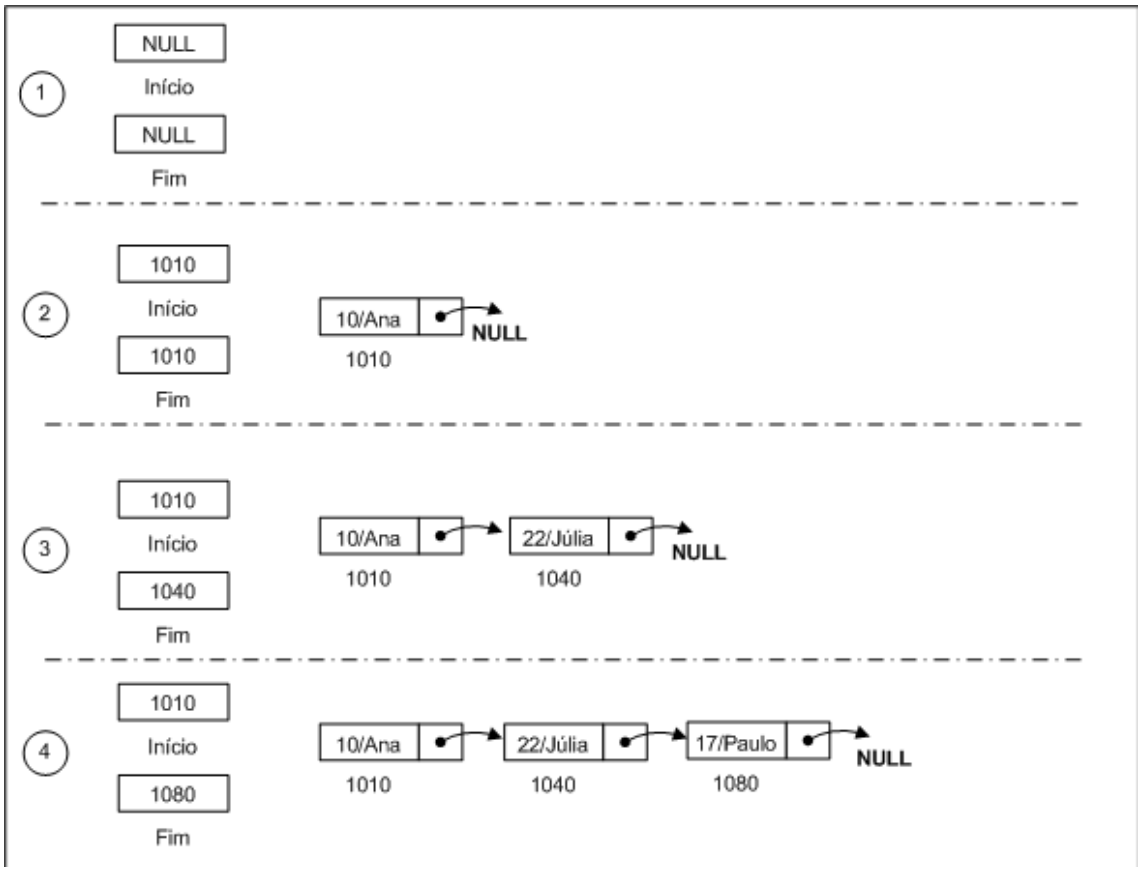
6.1. Listas Encadeadas Desordenadas

Esta lista é implementada usando ponteiros. A memória para armazenar os dados é alocada em tempo de execução. Na próxima seção será descrito o funcionamento de cada uma das operações e apresentado o código de um programa completo que controla o cadastro de um número inteiro uma lista usando ponteiros. Esta lista possui dois ponteiros, um que guarda o endereço do primeiro elemento da lista (início) e outro que guarda o endereço do último elemento da lista (fim).

6.1.1 Operações Básicas

Inserir Elemento

A figura abaixo ilustra a inserção de três elementos em uma lista dinâmica desordenada. Inicialmente a lista está vazia, portanto o valor do ponteiro início e fim é NULL (passo 1). Quando um elemento vai ser inserido, a memória é alocada e os dados são armazenados (passo 2). Todo nó criado em uma lista dinâmica desordenada, não tem vizinho seguinte, por isso ele aponta para NULL. Na inserção do primeiro elemento, os ponteiros início e fim apontam para este elemento (no exemplo, endereço 1010).

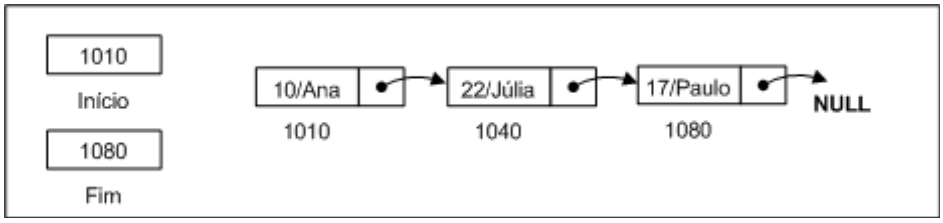


No passo 3 temos a chegada de um outro elemento. A memória é alocada e o novo nó tem os dados preenchidos e tem que ser anexado aos outros nós da lista. Todo nó que chega aponta para NULL e o ponteiro fim passa a ter o endereço do nó que acabou de chegar. O nó que anteriormente era o último da lista(1010) passará a ter como vizinho o nó que acabou de chegar (ou seja, aponta para 1040).

O passo 4 mostra uma outra inserção de nó. Veja que a cada novo elemento, apenas o endereço do ponteiro fim é modificado. O ponteiro início permanece com o mesmo valor.

Consultar Elemento

Para fazer uma consulta em uma lista dinâmica é necessário saber qual elemento deseja consultar. Neste caso faremos uma consulta por matrícula. Um ponteiro auxiliar será usado para percorrer a lista, visitando cada nó a procura do elemento.



Na figura acima temos uma lista com três elementos. Caso quiséssemos consultar o elemento de matrícula 25, iríamos percorrer a lista até chegar no último nó, cujo endereço do vizinho é NULL (nó de endereço 1080) e ficaríamos sabendo que este elemento não se encontra na lista. Quando um elemento é encontrado, seus dados são apresentados e quando ele não está na lista, uma mensagem de erro deve ser dada ao usuário.

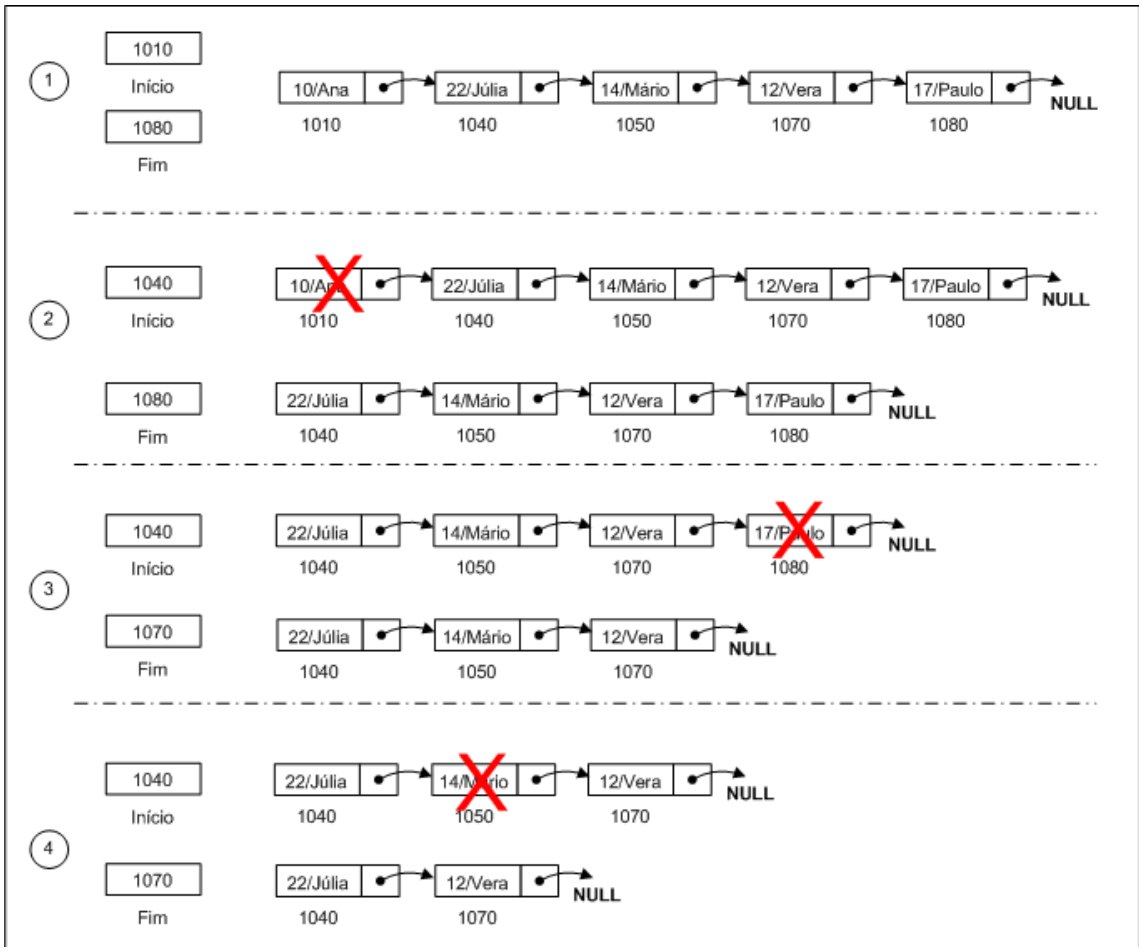
Remover Elemento

Para a remover um elemento é necessário saber qual elemento deseja remover. Para isso, a matrícula do aluno a ser removido deve ser lida. É feita uma varredura em todos os nós da lista. Assim que o elemento é encontrado, seus dados devem ser apresentados ao usuário (neste caso a matrícula e o nome).

Dessa forma ele pode verificar se realmente é aquele o elemento a ser removido. Quando o elemento não é encontrado, uma mensagem de erro deve ser dada ao usuário. Na figura a seguir são ilustrados os três casos de remoção: remover primeiro da lista, último da lista, elemento no meio da lista. No passo 1 temos a lista com cinco elementos. Primeiramente será feita a remoção do aluno com matrícula 10. Este é o primeiro nó da lista. Esta remoção é feita modificando o valor do nó início para o endereço do vizinho do nó que está sendo removido, neste caso, endereço 1040 (Passo 2).

Em seguida será removido a aluno de matrícula 17. Este aluno é o ultimo nó da lista, a sua remoção irá ter que atualizar o ponteiro fim. Além disso, o elemento que tinha como vizinho seguinte o nó de matrícula 17, terá agora NULL como vizinho. Portanto, dois ponteiros são atualizados (Passo 3).

No passo 4 será feita a última remoção, aluno com matrícula 14 (endereço 1050), que está armazenado em um nó no meio da lista. Nesta remoção, os ponteiros início e fim não são alterados. No entanto, o elemento que vem antes do removido (1040), terá agora como vizinho o elemento que era vizinho do que será removido (1070).



Listagem de Todos os Elementos

A operação de listagem possibilita a visualização dos dados de todos os elementos cadastrados. É feita uma varredura na lista partindo do endereço início até o último nó, todos os dados de todos os elementos são apresentados ao usuário.

Exemplo

```
// Lista encadeada desordenada.cpp : main project file.
#include "stdafx.h"
#include "stdio.h"
#include "conio.h"
#include "stdlib.h"

typedef struct Temp
{
        int info; //variável que receberá a informação
        struct Temp *proximo; //ponteiro para o próximo elemento da lista
}TNODO;
```

```
// cria o inicio da lista
TNODO *inicio; //ponteiro para o início da lista
TNODO *fim; //ponteiro para o fim da lista

//-------------------------------------------------------
// Funcao que define a lista como vazia.
void CriaLista()
{
  inicio = NULL;
  fim = NULL;
}
//-------------------------------------------------------
// Funcao que insere um elemento do inicio da lista.

TNODO *CriaElemento()
{
   TNODO *p;
   p =  (TNODO *) malloc(sizeof(TNODO)); //aloca memória e atribui ao ponteiro p
   if (p==NULL)
   {
     printf("Erro de alocacao\n");
   }
   printf("Informe Numero: ");
   scanf("%d", &p->info);
   p->proximo = NULL;
   return p;
}

void inserir()
{
  TNODO *novoElemento;
  novoElemento = CriaElemento();

  if (inicio==NULL) // insere no início, caso a lista esteja vazia
  {
    inicio = novoElemento; //ponteiro início aponta para novo elemento
    fim = novoElemento; //ponteiro fim também aponta para novo elemento
  }
  else //insere no fim, caso a lista já possua elementos
  {
      fim->proximo = novoElemento; //ponteiro próximo, do último elemento,
                                   aponta para o novo
      fim = fim->proximo; //desloca ponteiro fim para o elemento seguinte
  }
}
```

```
//---------------------------------------------------------
// Funcao que imprime toda a lista.
void listar()
{
  TNODO *p;
  if (inicio == NULL)
  {
    printf("--- fim da lista ---\n\n");
  }
  // Caso a lista nao esteja vazia
  p = inicio;
  while (p !=NULL)
  {
    printf("Info = %d\n",p->info);
    p = p->proximo;
  }
  printf("--- fim da lista ---\n\n");
}

//---------------------------------------------------------
// Funcao que busca um elemento na lista.
// Retorna:
//        - NULL caso nao encontre
//        - ponteiro para o NODO onde esta' o dado, se conseguir encontrar

TNODO *BuscaDado(int dado)
{
  TNODO *p;
  if (inicio == NULL)
  {
   return NULL;  // Lista Vazia
  }
  // Caso a lista nao esteja vazia
  p = inicio;
  while (p !=NULL) {
    if (p->info == dado)  // achou !!
          return p;       // retorna um ponteiro para
                    //o inicio da lista
    else
                  p = p->proximo;
  }
  return NULL;
}

//---------------------------------------------------------
//Funcao que consulta um elemento da lista
void consultar()
```

```
{
   int i;
   TNODO *p;
   printf("Informe Numero a ser consultado: ");
   scanf("%d", &i);
   p = BuscaDado(i);
   if (p != NULL)
      printf("+++Achou o %d\n", p->info);
   else
      printf("---Nao achou o %d\n", i);
}

//-------------------------------------------------------
// Funcao que remove um elemento especificado por 'dado'
void remover()
{
  int i;
  TNODO *atual, *anterior;

  printf("Informe Numero a ser removido: ");
  scanf("%d", &i);

  if (inicio==NULL)
  {
     printf("Lista vazia");  // Lista vazia !!!
  }
  else
  {   // Caso a lista nao esteja vazia
     atual = inicio;
     anterior = inicio;
     while (atual != NULL)
     {
                if (atual->info == i) // achou !!
                {
                    if (atual == inicio) // se esta removendo o primeiro da lista
                    {
                            inicio = inicio->proximo;
                            free(atual);
                            break;
                     }
                    else  // esta removendo do meio da lista
                    {
                           if (atual == fim)
                           fim = anterior;
                           anterior->proximo = atual->proximo;
                           free(atual);
                           break; // Refaz o encadeamento
```

```
                }
            }
            else  // continua procurando no prox. nodo
            {
              anterior = atual;
              atual = atual->proximo;
            }
        }
    }
}

void menu()
{
   int op;
   do
   {
       printf("\nOpcoes: \n\n");
       printf(" 1 - Inserir novo numero\n");
       printf(" 2 - Consultar numero\n");
       printf(" 3 - Remover numero\n");
       printf(" 4 - Listar todos os numeros\n");
       printf(" 0 - para sair \n\n");
       printf("Entre com a sua opcao: ");
       scanf("%d", &op); /* Le a opcao do usuario */
       switch (op)
       {
            case 1: inserir(); break;
            case 2: consultar(); break;
            case 3: remover(); break;
            case 4: listar(); break;
            case 0: break;
            default: printf("\n\n Opcao invalida"); getche(); break;
       }
   } while (op != 0);
}

//-----------------------------------------------
// Programa principal
int main()
{
    CriaLista();
    menu();
}
```

6.2. Lista Encadeada Ordenada

Esta lista é implementada usando ponteiros, no entanto, quando for feito o caminhamento pelos nós da lista, ela deve estar ordenada por algum campo. Esta lista possui apenas o ponteiro início, que guarda o endereço do primeiro elemento da lista.

6.2.1 Operações Básicas

Inserir Elemento
Na figura 6.1 é ilustrada a inserção de quatro elementos em uma lista dinâmica ordenada. No passo 1 a lista está vazia, com o ponteiro início apontando para NULL.

No passo 2 temos a inserção do elemento de matrícula 17. Como a lista está vazia, o ponteiro início vai apontar para este elemento (endereço 1080). No passo 3 temos a chegada de um outro elemento, matrícula 10. É verificado que ele tem a matrícula menor do que o primeiro elemento da lista, então este novo elemento terá que ser inserido no início da lista.

Assim, o elemento novo vai apontar para o primeiro da lista e o ponteiro início irá apontar para o novo nó. Em seguida, teremos a inserção de um aluno com matrícula 14, que será inserido no meio da lista. Para isso, teremos que descobrir entre quais elementos ele irá ser inserido, para manter a lista ordenada e fazer as ligações dos ponteiros corretamente.

Finalmente, no passo 5 teremos a inserção do aluno com matrícula 22. Esse será inserido no final da lista.

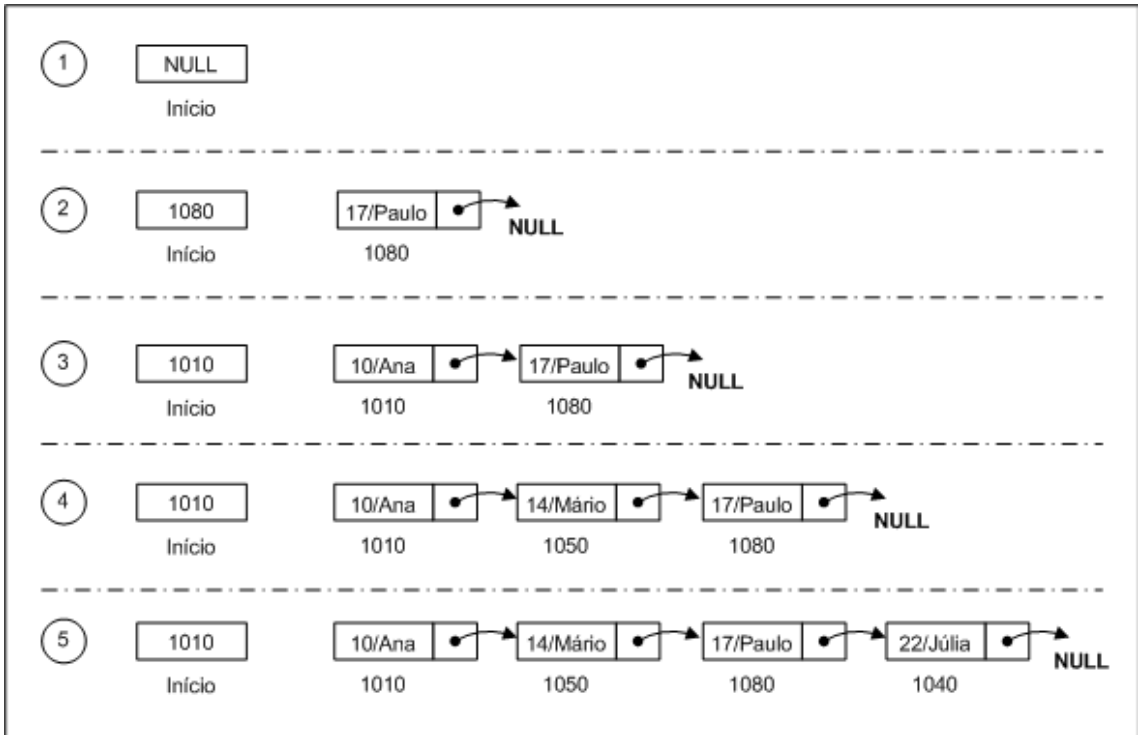


Figura 6.1: Inserção ordenada na lista encadeada

Para as operações de listagem e consulta seguem a mesma regra utilizada na lista encadeada desordenada. Para a remoção a diferença é a ausência do ponteiro fim, já que a inserção pode acontecer em qualquer posição da lista.

Exemplo: função para inserir na lista encadeada de forma ordenada

```
void inserir(int dado)
{
  TNODO *novono, *noanterior, *noatual;
  novono =  (TNODO *) malloc(sizeof(TNODO));

  novono->info = dado;
  novono->proximo = NULL;

  //Inserir novono na lista
  if (inicio == NULL) /* Ainda nao existe nenhum elemento na lista */
  {
    inicio = novono;
  }
  else
  {
     noatual = inicio;
     if (noatual->info > dado) // elemento inserido no inicio da lista
     {
       novono->proximo = inicio;
       inicio = novono;
     }
     else // elemento sera inserido no meio ou final da lista
     {
      while (noatual != NULL)
      {
           if (noatual->info < dado) // procura o local da insercao
           {
             noanterior = noatual;
             noatual = noatual->proximo;
           }
           else // encontrou o local onde sera inserido
              noatual = NULL;
      }
      novono->proximo = noanterior->proximo;
      noanterior->proximo = novono;
   }
  }
}
```