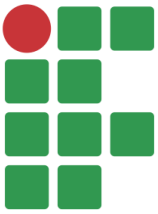

INSTITUTO FEDERAL DA PARAÍBA
CAMPUS CAMPINA GRANDE
BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO
DISCIPLINA DE LABORATÓRIO DE ESTRUTURAS DE DADOS
PROF. VICTOR ANDRÉ PINHO DE OLIVEIRA

Lab. Estruturas de Dados

Aula 5 - ED - Listas Duplamente Encadeadas

Introdução

Olá,

Na semana anterior nós implementamos uma Lista Simplesmente Encadeada, uma Lista onde cada nó tem um ponteiro para o próximo nó. Vimos como inserir os elementos de forma não Ordenada, pondo-os imediatamente no fim, e também como inseri-los de maneira Ordenada.

Hoje daremos continuidade às Listas Encadeadas, mas, desta vez, veremos **Listas Lineares Duplamente Encadeadas**, como implementá-las de forma não Ordenada e de forma Ordenada. Mais uma vez, reitero, é preciso estar bem afiado em ponteiros.

Sigam-me os bons...

Listas Duplamente Encadeadas

Uma **Lista** é uma estrutura que armazena elementos de mesmo tipo. Chamamos uma lista de **Linear** porque cada elemento tem apenas um antecessor e um sucessor[1]. Uma lista Linear pode ser Sequencial ou Encadeada. Uma Lista Linear é **Sequencial** quando os elementos estão dispostos sequencialmente na memória. Uma Lista Linear é **Encadeada** quando os elementos não estão em sequência na memória, mas cada elemento possui um ponteiro para o próximo elemento da Lista.

As Listas Encadeadas podem ser Simplesmente ou Duplamente Encadeadas. É **Simplesmente Encadeada** quando cada nó possui um ponteiro para o próximo nó. É **Duplamente encadeada** quando cada nó possui um ponteiro para o nó anterior e outro para o próximo nó.

Na aula de hoje, voltaremos nossa atenção para **Listas Duplamente Encadeadas**. Deixaremos o termo Linear subentendido.

[1] À exceção, claro, do primeiro e do último elemento da lista.

Listas Duplamente Encadeadas não Ordenadas

Como você bem sabe, as Listas podem ser Ordenadas e não Ordenadas. Chamamos uma Lista **Ordenada** quando os elementos são inseridos de maneira ordenada. Por outro lado, chamamos uma Lista de **não Ordenada** quando não há preocupação quanto a inserção ordenada dos elementos.

Começaremos vendo as Listas não Ordenadas. Atente que existem várias formas de implementar uma Lista Linear Encadeada. Aqui, apresentarei uma variante. Mas exploraremos outras possibilidades nos exercícios.

Segue abaixo o código.

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

struct sNODE{
  int dado;
  struct sNODE *ant;
  struct sNODE *prox;
};

struct sNODE *ini = NULL, *fim = NULL;

void inserir(int dado);
void remover(int dado);
struct sNODE *buscar(int dado);

int obter(struct sNODE *node);
int tamanho();
void imprimir();
void apagar();

int main(){
  return 0;
}

void inserir(int dado){
  struct sNODE *novo = (struct sNODE*) malloc(sizeof(struct
sNODE));
  novo->dado = dado;
  novo->ant = NULL;
  novo->prox = NULL;
```

```
  if (!ini)
     ini = fim = novo;
  else{
     fim->prox = novo;
     novo->ant = fim;
     fim = novo;
  }
}

struct sNODE *buscar(int dado){
  struct sNODE *aux = ini;

  while (aux){
     if (dado == aux->dado)
          return aux;
     aux = aux->prox;
  }

  return NULL;
}

void remover(int dado){
  struct sNODE *aux = buscar(dado);

  if (!aux)
     return;

  if (!aux->ant && !aux->prox)
     ini = fim = NULL;
  else if (aux == ini) {
     ini = ini->prox;
     ini->ant = NULL;
  } else if (aux == fim) {
     fim = fim->ant;
     fim->prox = NULL;
  } else {
     aux->ant->prox = aux->prox;
     aux->prox->ant = aux->ant;
  }
  free(aux);
```

```
}

void apagar(){
  struct sNODE *aux = ini, *ant;

  while (aux){
      ant = aux;
      aux = aux->prox;
      free(ant);
  }
  ini = fim = NULL;
}

int obter(struct sNODE *node){
  if (!node){
      printf("Erro ao obter dado. Ponteiro invalido.");
      exit(0);
  }

  return node->dado;
}

int tamanho(){
  struct sNODE *aux = ini;
  int tam = 0;

  while (aux){
      tam++;
      aux = aux->prox;
  }

  return tam;
}

void imprimir(){
  struct sNODE *aux = ini;

  printf("[ ");
  while (aux){
      printf("%d ", aux->dado);
      aux = aux->prox;
```

```
  }
  printf("]\n");
}
```

O código acima mostra **uma forma** de implementar uma Lista de int.

Bem, para implementarmos uma Lista Duplamente Encadeada vamos precisar de:

- um registro (struct). Esse registro representa um nó da Lista. Ele contém três campos:
  - dado: para armazenar o elemento
  - ant: um ponteiro para o nó anterior da Lista
  - prox: um ponteiro para o próximo nó da Lista
- um ponteiro para o início da Lista (ini) e outro para o fim (fim)
- de algumas funções que controlam a manipulação da estrutura:
  - inserir: permite inserir um elemento no fim da Lista
  - remover: permite remover um determinado elemento da Lista
  - buscar: retorna um ponteiro para o nó de um elemento buscado, caso encontrado
  - apagar: apaga toda a Lista
- de mais algumas funções auxiliares:
  - obter: retorna o elemento do nó apontado pelo ponteiro passado
  - tamanho: retorna a quantidade de elementos
  - imprimir: imprime na tela a Lista

Como estamos implementando uma Lista de int, tanto o campo dado quanto as funções foram implementados esperando receber tal tipo.

Adicionalmente, note que, conforme podemos ver no código acima, você pode declarar todas as suas funções (assinaturas) acima da main e deixar para implementá-las em qualquer outro lugar abaixo no arquivo. Isso te dá mais liberdade em algumas situações. Por hora, estou apenas deixando como uma boa dica.

Por estarmos implementando uma Lista, que é um tipo de Estrutura de Dados, não podemos manipular os ponteiros diretamente, mas somente por meio das funções. As funções ditam as regras de acesso à Estrutura. Assim, basta entendermos como cada função funciona.

Função inserir

A função inserir tem o objetivo de inserir um elemento no final da Lista.

```
void inserir(int dado){
  struct sNODE *novo = (struct sNODE*) malloc(sizeof(struct
sNODE));
  novo->dado = dado;
```

```
  novo->ant = NULL;
  novo->prox = NULL;

  if (!ini)
      ini = fim = novo;
  else{
      fim->prox = novo;
      novo->ant = fim;
      fim = novo;
  }
}
```

A função inserir não retorna nada e recebe o elemento a ser introduzido na Lista. A primeira coisa a fazer é alocar o espaço para o novo nó da Lista. Preenchemos o campo dado desse nó com o dado passado à função e os campos ant e prox com NULL, deixando para ajustar o ponteiro ant tão logo coloquemos o elemento no fim da Lista. A princípio, o novo nó não aponta para nenhum outro nó. Atribuímos NULL ao ponteiro prox para indicar o fim da Lista, bem como para evitar apontar para uma região que não seja de fato um nó. O ponteiro ant permanecerá NULL unicamente quando o nó inserido for o único da Lista, situação em que o elemento é tanto o primeiro quanto o último.

Depois de criado o nó, vamos inseri-lo à Lista. Como queremos inserir no fim, podemos nos deparar com uma das duas situações: Lista vazia ou Lista não vazia. Uma Lista está vazia quando os ponteiros ini e fim estão NULL. Se esse for o caso, os ponteiros ini e fim deverão apontar para o novo e único elemento da Lista. Caso contrário, pegamos o último nó da Lista e o fazemos apontar para o novo nó (fim->prox = novo). Também precisamos ajustar o ponteiro ant, fazendo-o apontar para o atual fim da Lista (novo->ant = fim). Somente depois é que atualizamos o ponteiro fim para o novo elemento.

Função buscar

A função buscar tem o objetivo de buscar um elemento na Lista e retornar sua posição - endereço do nó -, caso ele exista na Lista. É exatamente a mesma implementação da buscar da Lista Simplesmente Encadeada.

```
struct sNODE *buscar(int dado){
  struct sNODE *aux = ini;

  while (aux){
      if (dado == aux->dado)
            return aux;
      aux = aux->prox;
```

```
  }

  return NULL;
}
```

A função retorna o ponteiro para o nó (caso o dado exista na Lista) e recebe o elemento a ser buscado. Inicializamos um ponteiro auxiliar (aux) com o início da Lista (ini). Em seguida entramos num loop while passando aux como expressão lógica. Isso quer dizer "enquanto não chegar ao fim da Lista", pois, em C, NULL é FALSO e diferente de NULL é VERDADEIRO. Verificamos se o campo dado do nó em questão (aux->dado) é igual ao dado que estamos buscando. Em caso verdadeiro retornamos aux. Note a forma como avançamos para o próximo nó da Lista: atribuímos à aux o endereço do próximo nó (aux = aux->prox).

Caso o dado não se encontre na Lista, retornamos o ponteiro NULL.

Função remover

A função remover tem por objetivo remover um determinado elemento da Lista.

```
void remover(int dado){
  struct sNODE *aux = buscar(dado);

  if (!aux)
     return;

  if (!aux->ant && !aux->prox)
     ini = fim = NULL;
  else if (aux == ini) {
     ini = ini->prox;
     ini->ant = NULL;
  } else if (aux == fim) {
     fim = fim->ant;
     fim->prox = NULL;
  } else {
     aux->ant->prox = aux->prox;
     aux->prox->ant = aux->ant;
  }
  free(aux);
}
```

A função não retorna nada e recebe o elemento a ser removido. Note que a função é um pouco mais simples se comparada à função remover das Listas Simplesmente Encadeadas. A razão para isso é a presença do ponteiro ant em cada nó. Contudo, ao

removermos um elemento, ainda temos que considerar algumas situações. Falaremos sobre elas abaixo.

A função começa inicializando o ponteiro aux com o retorno da função buscar. Em seguida, verificamos se aux é NULL, caso em que buscar não achou o elemento a ser removido e, portanto, retorna porque não há mais o que fazer.

Continuando, podemos cair em uma das quatro situações:

1. a Lista só tem um elemento, situação em que os ponteiros ant e prox do nó são NULL. Nesse caso, precisamos atualizar o ponteiro ini e fim para NULL.
2. o dado buscado se encontra no início da Lista. Nesse caso, precisamos avançar o ponteiro ini (ini = ini->prox) e depois ajustar o ponteiro ant do nó ini para NULL (agora, o atual nó ini é o primeiro da Lista; antes era o segundo).
3. o dado buscado se encontra no fim da Lista. Nesse caso, o penúltimo nó da Lista passará a ser o último. Assim, recuamos o ponteiro fim (fim = fim->ant) e atualizamos o ponteiro prox do nó fim (agora, o atual nó fim é o último da Lista; antes era o penúltimo).
4. o dado buscado se encontra no meio da Lista. Nesse caso, fazemos com que o ponteiro prox do nó anterior a aux (aux->ant->prox) aponte para o nó depois de aux (aux->prox). Assim, aux->ant->prox deverá apontar para aux->prox. Semelhantemente, fazemos com que o ponteiro ant do nó posterior (aux->prox->ant) aponte para o nó anterior a aux (aux->ant). Assim, aux->prox->ant passará a apontar para aux->ant.

Em todos os casos, como aux aponta para o nó a ser removido, ele será usado para desalocar o ponteiro com free. Adicionalmente, a função remover, conforme implementada acima, só remove a primeira ocorrência.

Função apagar

A função apagar não recebe nem retorna nada. Tem o objetivo de apagar todos os nós da Lista, deixando-a vazia.

```
void apagar(){
  struct sNODE *aux = ini, *ant;

  while (aux){
      ant = aux;
      aux = aux->prox;
      free(ant);
  }
  ini = fim = NULL;
}
```

A função está implementada exatamente da mesma forma que apagar da Lista Simplesmente Encadeada. Declaramos e inicializamos dois ponteiro: aux e ant. O

ponteiro aux corre por toda a Lista, passando de nó a nó. Note a ordem que construímos o loop. Primeiro atribuímos aux para ant, e depois avançamos aux para o próximo nó. Somente após é que chamamos free(ant). O loop segue até encontrar o fim da Lista, momento em que aux se torna NULL.

Já que estamos apagando toda a Lista, estamos setando os ponteiros ini e fim para NULL.

Funções auxiliares

E agora, veremos as funções auxiliares. Todas exatamente iguais às das Lista Simplesmente Encadeadas.

Função obter

A função obter recebe um ponteiro para o nó e retorna o dado, caso ele esteja presente na Lista.

```
int obter(struct sNODE *node){
  if (!node){
      printf("Erro ao obter dado. Ponteiro invalido.");
      exit(0);
  }

  return node->dado;
}
```

Talvez você esteja se perguntando sobre a necessidade desta função, questionando se não seria melhor acessar o ponteiro direto e pronto. NÃO. Não seria melhor acessar o ponteiro diretamente porque precisamos manter o controle sobre o acesso à Lista. Note que a função obter verifica, antes de retornar o elemento, se o ponteiro passado como argumento é um ponteiro válido, isto é, diferente de NULL. Mais uma vez, é importante lembrar que, por estarmos implementando uma Estrutura de Dados, o acesso às variáveis da Estrutura deve se dar sempre por meio das funções, pois são elas que controlam o acesso e a manipulação das variáveis, mantendo a integridade da Estrutura.

Observe que caso o ponteiro NULL seja passado como argumento, a função encerra o programa, pois não há dado a ser retornado.

Outro detalhe que vale comentar é que a função não verifica se o ponteiro passado é, de fato, um nó da Lista. Assumimos que o programador chamou a função buscar para obter o ponteiro para o nó.

Função tamanho

Retorna o tamanho da Lista.

```
int tamanho(){
```

```
  struct sNODE *aux = ini;
  int tam = 0;

  while (aux){
      tam++;
      aux = aux->prox;
  }

  return tam;
}
```

Basicamente, a função declara um contador (tam) e incrementa-o enquanto aux corre pela Lista.

Função imprimir

A função imprimir não recebe nem retorna nada. Apenas imprime a Lista na tela.

```
void imprimir(){
  struct sNODE *aux = ini;

  printf("[ ");
  while (aux){
      printf("%d ", aux->dado);
      aux = aux->prox;
  }
  printf("]\n");
}
```

Usamos aux para correr pela Lista. Em cada loop, imprimimos o dado do nó.

Listas Duplamente Encadeadas Ordenadas

Agora chegou a hora de implementar uma Lista Duplamente Encadeada Ordenada. Ora, se já sabemos que em uma Lista Ordenada os elementos já são inseridos de forma ordenada, basta modificarmos a função inserir.

Função inserir_ord

A função inserir_ord insere os elementos de forma ordenada na Lista. Não retorna nada e recebe o elemento a ser inserido.

```
void inserir_ord(int dado){
  struct sNODE *aux = ini;
  struct sNODE *novo = (struct sNODE*) malloc(sizeof(struct
sNODE ));
```

```
  novo->dado = dado;
  novo->ant = NULL;
  novo->prox = NULL;

  while(aux && dado > aux->dado)
      aux = aux->prox;

  if (!ini)
      ini = fim = novo;
  else if (aux == ini){
      novo->prox = ini;
      ini->ant = novo;
      ini = novo;
  } else if (!aux){
      fim->prox = novo;
      novo->ant = fim;
      fim = novo;
  } else {
      aux->ant->prox = novo;
      novo->ant = aux->ant;
      novo->prox = aux;
      aux->ant = novo;
  }
}
```

Declaramos a variável aux e a inicializamos com ini. Adicionalmente, declaramos um registro novo e alocamos o espaço para ele. Depois preenchemos os campos do registro.

Em seguida, usamos aux para correr pela Lista até o ponto em que o elemento deve ser inserido, ou seja, avançamos o ponteiro aux enquanto o dado a ser inserido é maior que o dado do nó. O ponteiro aux vai na expressão do while para garantir de que não vamos tentar acessar o campo dado de um ponteiro vazio. Se aux for NULL a segunda parte da expressão já não é avaliada. Quando o loop se encerra é porque encontramos a posição para o novo nó.

Ao inserir o novo nó, podemos cair em uma das 4 situações:

1. a Lista está vazia. Nesse caso, o novo nó será o primeiro e o último da Lista.
2. aux aponta para o primeiro nó da Lista. Nesse caso, fazemos o novo nó apontar para ini (novo->prox = ini), o antigo nó ini apontar para o novo nó (ini->ant = novo), para, somente depois, atualizamos o ponteiro ini (ini = novo)
3. aux é NULL, ou seja, o elemento a ser inserido é maior que todos os outros e, portanto, deverá ser inserido no fim da Lista. Nesse caso, fazemos o último nó apontar para o novo (fim->prox = novo), o novo nó apontar para o antigo fim

(novo->ant = fim), para, somente depois, atualizamos o ponteiro fim (fim = novo)

4. o novo nó será inserido em alguma posição pelo meio da Lista. Nesse caso, fazemos com que:
    a. o nó anterior ao nó aux aponte para o novo (aux->ant->prox = novo[2]) e o novo nó aponte para o nó anterior de aux (novo->ant = aux->ant);
    b. o novo nó aponte para aux (novo->prox = aux) e aux aponte para novo (aux->ant = novo).

Na aula de hoje implementamos nossa terceira Estrutura: Lista Duplamente Encadeada. Por um lado foi mais fácil, porque aproveitamos parte do raciocínio das Listas Simplesmente Encadeadas. Por outro, em algumas funções, aumentou o grau de atenção por termos que controlar mais um ponteiro da Estrutura. Mais uma vez, peço que revise todo o material com atenção, não deixando nenhuma dúvida pairando na cabeça.

Na próxima aula iremos tratar sobre a Estrutura de Dados chamada de Pilha. Veremos como implementar uma Pilha com arrays e com Listas Encadeadas.

Bons estudos e até a próxima!

[2] Como o campo ant também é um registro sNode, podemos, sem problemas, usar o operador seta.