**LISTA DINÂMICA ENCADEADA**

- Em uma lista encadeada, há alocação dinâmica de memória, isto é, alocação de memória em tempo de execução, o que permite que a cada novo elemento, um espaço a mais de memória seja alocado para armazená-lo. Isso dispensa a necessidade da definição prévia da quantidade de elementos da lista.

- Os elementos não são armazenados em espaços contíguos de memória, razão esta pela qual não há acesso direto (indexado) aos elementos da lista.

- Cada elemento guarda, além de suas informações, um ponteiro para o próximo elemento, para que seja possível percorrer todos os elementos da lista. O último elemento aponta para *NULL*.

- Vantagens:

* Melhor utilização dos recursos de memória;
* Facilidade de inserção e remoção, já que não é necessário deslocamento de elementos.

- Desvantagens:

* Acesso indireto aos elementos;
* Necessidade de percorrer a lista para acessar um elemento.

- A lista dinâmica encadeada é vantajosa para aplicações onde inserção e remoção em lista ordenada são as operações mais frequentes.

ESTRUTURA DO NÓ

- Definição do elemento:

// define Elem como todo ponteiro para a struct elemento

typedef struct elemento Elem;

struct elemento{

struct aluno dados;

struct elemento \*prox;

};

ALOCAÇÃO DINÂMICA DE MEMÓRIA

- Indicamos para a função *malloc* o tipo do dado e o número de bytes a serem alocados. A função retorna um ponteiro do tipo *void\** para a memória alocada. sizeof é utilizado para fornecer o tamanho.

// tipo No e tamanho igual à necessária para armazenar uma struct no

No novoNo = (No) malloc (sizeof(struct no));

// retorna ponteiro novoNo

OPERAÇÕES

Modelagem

- *Lista* será um ponteiro de ponteiro, sinalizando o primeiro elemento da lista. Este é o chamado nó cabeça.

struct aluno{

int matricula;

char nome[30];

float n1,n2,n3;

};

struct elemento{

struct aluno dados;

struct elemento \*prox;

};

typedef struct elemento Elem;

// ponteiro para ponteiro

// receberá o endereço para a lista a ser criada

typedef struct elemento\* Lista;

void main(){

// ponteiro que receberá o ponteiro para a lista que será criada

Lista \*li;

}

- Para percorrer a lista, usamos um ponteiro: o ponteiro *\*no* que atribuímos a ele o ponteiro para o início da lista *\*li*, isto é, para o primeiro elemento.

- Quando procedemos uma operação que envolve rearranjo da ordem, utilizamos um ponteiro *\*anterior*, e um ponteiro *\*no* ou *\*atual* para iterar sobre a lista de forma a acessar cada elemento sem perder acesso ao seu antecessor.

Inicialização

- Cria a lista: há a criação do primeiro elemento da lista e o ponteiro recebe o endereço deste.

// aloca memória para o primeiro elemento e

// passa seu endereço para o ponteiro li

Lista\* cria\_lista(){

Lista\* li = (Lista\*) malloc(sizeof(Lista));

// se deu certo, preenche os campos do primeiro elemento com NULL

if(li != NULL)

\*li = NULL;

return li;

}

Desfaz lista

void libera\_lista(Lista\* li){

if(li != NULL){

Elem\* no;

while((\*li) != NULL){ // enquanto houver lista

no = \*li; // no recebe li, que é o começo da lista

//(cabeça)

\*li = (\*li)->prox; // li passa a apontar para o próximo elemento

free(no); // apaga o elemento

}

free(li); // finaliza por liberar o último nó

}

}

Tamanho

int tamanho\_lista(Lista\* li){

if(li == NULL)

return 0;

int cont = 0;

// no recebe o início da lista, e depois, a cada iteração, recebe um por um dos nós

// e quantidade é contabilizada em cont

Elem\* no = \*li;

while(no != NULL){

cont++;

no = no->prox;

}

return cont;

}

Inserção na lista ordenada

int insere\_lista\_ordenada(Lista\* li, struct aluno al){

if(li == NULL) // se a lista não existe

return 0;

Elem \*no = (Elem\*) malloc(sizeof(Elem));

if(no == NULL) // se não foi possível criar o novo nó

return 0;

no->dados = al;

if((\*li) == NULL){ // se lista vazia, insere no início

no->prox = NULL;

\*li = no;

return 1;

}

else{

Elem \*ant, \*atual = \*li;

// atual começa no início da lista; guarda o anterior

// quando faz atual receber o próximo

while(atual != NULL && atual->dados.matricula < al.matricula){

ant = atual;

atual = atual->prox;

}

if(atual == \*li){ // inserção no início

no->prox = (\*li);

\*li = no;

}else{ // inserção no meio ou fim; se fim,

// prox recebe NULL

no->prox = atual; // próximo do atual vira o próximo do nó

ant->prox = no; // próximo do anterior recebe o nó

}

return 1;

}

}

Remoção

int remove\_lista(Lista\* li, int mat){

if(li == NULL)

return 0;

if((\*li) == NULL)

return 0;

Elem \*ant, \*no = \*li;

// percorre a lista procurando o elemento de interesse

while(no != NULL && no->dados.matricula != mat){

ant = no;

no = no->prox;

}

if(no == NULL) // se não for encontrado

return 0;

if(no == \*li) // remoção no início

\*li = no->prox;

else // remoção no meio ou não fim

ant->prox = no->prox;

free(no);

return 1;

}

Busca por posição

int consulta\_lista\_posicao(Lista\* li, int pos, struct aluno \*al){

if(li == NULL || pos <= 0)

return 0;

Elem \*no = \*li;

int i = 1;

while(no != NULL && i < pos){

no = no->prox;

i++;

}

if(no == NULL) // se a posição não se enontra

return 0;

else{

\*al = no->dados; // copia conteúdo do nó na posiçã

return 1;

}

}

Busca por elemento

int consulta\_lista\_elemento(Lista\* li, int mat, struct aluno \*al){

if(li == NULL)

return 0;

Elem \*no = \*li;

while(no != NULL && no->dados.matricula != mat){

no = no->prox;

}

if(no == NULL) // se o elemento não se encontra

return 0;

else{

\*al = no->dados;

return 1;

}

}