RESUMO - LISTAS - AED

Listas Lineares Sequenciais

Ordem (lógica) dos elementos da lista coincide com sua posição física (em memória).
Elementos adjacentes da lista ocupam posições contíguas de memória.

Inicialização: nroElem = 0 (pra sabermos o fim da lista e não termos que percorrer o arranjo inteiro, caso ele não esteja completo);

Vantagens

- Acesso direto O(1) aos elementos da lista através dos índices. Mas nem sempre temos o índice.
- Se estiver ordenada então a busca acaba sendo feita em O(log(n)).

Desvantagens

- Implementação estática. Necessita determinar o tamanho de elementos a serem recebidos previamente. É possível aumentar mas é custoso O(n).
- Inserção e exclusão no pior caso são O(n), pois terá que deslocar todos os elementos.

Listas Lineares Ligadas

Ordem (lógica) dos elementos (nós) não necessariamente coincide com sua posição física na memória. Ela pode ser implementada de forma estática (usando-se um array) ou, em linguagens de programação que oferecem suporte à alocação dinâmica, com uso de ponteiros.

Vantagens

- Inserção não precisa deslocar n elementos, é só alterar o ponteiro.
- Apenas a inserção é O(1), porém ainda há a busca que no pior caso será O(n)

Listas Ligadas Estáticas

É formada pelo array de **registros**, um indicador de início e outro de nós disponíveis(dispo).
inicio e dispo são as entradas de duas listas que compartilham o mesmo array, sendo uma para os elementos efetivos da lista, e a outra para armazenar as posições livres.

Inicialização: inicio = -1 (lista vazia)
dispo = 0 (primeiro elemento disponível fica na posição 0)
Cada posição (i) do array indica que o prox é o i+1
Próximo do último será = -1 (fim da lista)

Desvantagens da Lista Estática

- Tamanho ainda deve ser definido no início do programa.

Listas Ligadas Dinâmicas

São listas as quais não precisam do gerenciamento de memória por parte do programador, pois a linguagem já faz o gerenciamento das posições de memória. Esse recurso é chamado alocação dinâmica de memória.
Neste caso é utilizada uma **estrutura diferente**. Ela é um ponteiro do tipo **NO**, que armazena ao menos uma chave e o ponteiro prox, que se refere ao próximo elemento da lista, que também é do tipo Nó.
Para inserir um elemento usamos o método malloc:

NO novo = (NO*) malloc(sizeof(NO));*

NO novo:* indica que estamos separando um espaço na memória que receberá um ponteiro de NO e o chamaremos de novo.
(NO):* é necessário para que seja feito o casting de tipo.
malloc(sizeof(NO)): cria um espaço de memória do tamanho de um NO. Terá espaço para o struct NO com todos os seus "atributos": chave, prox, nUsp, nome, turma etc
e para excluir um elemento/NO usamos:
free(p); //sendo p o ponteiro para o Nó que desejamos excluir

No Cabeça

É criado um No(vazio) que indica o início da Lista. Não muda complexidade do algoritmo, mas **facilita MUITO sua implementação,** já que evita tenhamos que pensar em casos como Lista vazia ou inserção no primeiro elemento.
Na struct LISTA teremos apenas o NO* cab;

Inicialização: No cabeça receberá um malloc
Prox do cabeça será NULL, caso não seja uma lista circular.

No Sentinela

No a priori vazio no fim da Lista. Serve para diminuir um pouco a complexidade do algoritmo de busca. Colocamos a chave buscada neste NO. Ao fazermos isso garantimos que a chave sempre será encontrada. Então podemos simplificar o algoritmo com apenas uma pergunta.

```
while(p->chaveAtual != chaveBuscada) {
        //vai para a próxima chave
}
//Obs: Essa busca serve para listas ordenadas e desordenadas.
```

E por fim, fazemos a verificação: Se a chave encontrada se encontra no NO sentinela, então retorne NULL (chave não existe):

```
if(p == l->sentinela) return NULL;
if(p->chaveAtual == chaveBuscada) return p;
return NULL;
```

Inicialização: No sentinela receberá um malloc
Prox do sentinela será NULL;
Inicio receberá sentinela.

Lista Circular e com No cabeça

Lista em que o prox do fim aponta para o No cabeça.

Inicialização: No cabeça receberá um malloc

Prox do cabeça será cabeça;

Lista Circular, com No cabeça e Duplamente ligada

Lista em que os elementos possuem além do campo prox um campo ant, para que seja possível percorrer a lista nos dois sentidos.

Inicialização: No cabeça receberá um malloc
Prox do cabeça será cabeça;
Ant do cabeça será cabeça.

Na inserção ainda é necessário o ant da busca, caso desejemos inserir ordenadamente.

Observações:

- SEMPRE lembrar de verificar 3 condições: lista vazia, primeiro elemento e qualquer elemento após o primeiro.
- Em lista sequencial na inserção deve-se verificar se os índices são válidos e se a lista não está cheia.
- Em lista sequencial (estática) na inclusão e exclusão de elementos deve-se deslocar os elementos para que seja criado um espaço para o elemento a ser inserido e retirar o espaço após o elemento ser excluido.

```
typedef struct S{
	TIPOCHAVE chave;
	S* prox;
}NO;
typedef struct{
	NO* inicio;
	NO* sent;
}LISTA;
void inicializar(LISTA *l){
	l->sent = (NO*)malloc(sizeof(NO));
	l->inicio = l->sent;
}
/*
	Busca para lista não ordenada. Não serve para inserir elementos de forma ordenada.
*/
NO* busca(LISTA l, TIPOCHAVE ch, NO* *ant){
	*ant = NULL;
	NO* p = l.inicio;
	l.sent->chave = ch;

	while(p->chave != ch){
		*ant = p;
		p = p->prox;
	}
	if(p == l.sent) return NULL;
```

```
	if(p->chave == ch) return p;
		return NULL;
}

/*
	Busca para lista ordenada
*/

NO* buscaListOrd(LISTA l, TIPOCHAVE ch, NO* *ant){
	*ant = NULL;
	l.sent->chave = ch;
	NO* p = l.inicio; //não precisa verificar se é null pq tem ao menos o sentinela
	while(p->chave < ch){
	*ant = p;
	p = p->prox;
	}
	if(!p) return NULL;
	if(p == l.sent) return NULL;
	if(p->chave == ch) return p;
	return NULL;
}

bool inserir(LISTA *l, TIPOCHAVE ch){
	NO* ant;
	NO* posicao = buscaListOrd(*l, ch, &ant);

	NO* novo = (NO*)malloc(sizeof(NO));
	if(!novo) return false;
	novo->chave = ch;

	if(!ant){
	if(l->inicio == l->sent){
		novo->prox = l->sent;
		l->inicio = novo;
```

```
        }else{
                novo->prox = l->inicio;
                l->inicio = novo;
        }

        }else{
        novo->prox = ant->prox;
        ant->prox = novo;
        }
        return true;
}

bool excluir(LISTA *l, TIPOCHAVE ch){
        NO* ant;
        NO* p = busca(*l, ch, &ant);

        if(!p) return false; //caso o elemento procurado não exista
        ant->prox = p->prox;
        free(p);
        return true;

}
void exibir(LISTA l){
        NO* atual = l.inicio;

        while(atual != l.sent){
        printf("%i\n",atual->chave);

        atual = atual->prox;
        }
        printf("\nFim da lista\n\n");
}
int main(){
        printf("Lista dinamica com No cabeca\n");
```

```
    LISTA l;
    inicializar(&l);

    inserir(&l,2);
    inserir(&l,1);
    inserir(&l,4);
    inserir(&l,3);
    exibir(l);

}
```