TAD - Tipo Abstrato de Dados

Intro, Nomenclaturas e Conceitos Gerais

- **Tipos de dados**:
  - relacionado à linguagem de programação
  - define o conjunto de valores que pode assumir e as operações disponíveis
  - Ex.: variáveis do tipo *int* podem assumir valores inteiros e suportar operações aritméticas
- **Tipo Abstrato de Dados (TAD)**:
  - tipos de dados desvinculados da implementação
  - definido pelo part (valor,operação)
- **Estrutura de Dados (ED)**:
  - modo particular de armazenar e organizar dados
  - arranjo, lista ligada, árvore ...
- **Chave - tipo chave**:
  - em um banco de dados, **chave** é um valor que permite identificar registros em um repositório de dados
  - geralmente, é um dos campos do próprio registro
- **Registro**
  - está entre as estruturas de dados mais simples
  - é um valor que contém outros valores
  - exemplo: uma data pode ser armazenada como um registro contendo os campos ano, mês e dia

TADs em C

- podem ser implementas utilizando módulos
- cada TAD é implementado em um arquivo.c
- um arquivo.h deve ser feito com os protótipos das funções públicas e com as definições dos tipos de dados
- utiliza **structs**:

```
typedef struct {
    <type> <name>
    .
    .
    .
} <struct_name>;
```

- para acessar cada elemento dentro da estrutura:
  - Nome.Elemento1 ou Nome -> Elemento1 (caso for ponteiro)
  - caso ponteiro, utiliza-se a função **malloc**

Malloc(sizeof())

- retorna um tipo *void
- para transformá-lo, utiliza-se o “casting” (ex.: int * x = **(int *)** malloc(sizeof(int));)

Listas

- podem ser ordanadas ou não
- operações de busca e removação são comumente feitas em relação à chave do elemento
- a inserção varia: |ordenada|não ordenada| ||| |operação de busca|insere no início ou fim|

TAD Listas

- **estática**: utiliza vetores ou arranjos
  - **vantagens**: tempo constante de acesso aos dados e inserção (não-ordenada)
  - **desvantagens**: alto custo para remover (e inserir em uma posição dada); tamanho máximo é pré-definido
  - **quando utilizar**: listas pequenas, tamanho máximo conhecido, poucas operações de inserção/remoção
- **dinâmica**: utiliza listas ligadas (ponteiros)
  - forma comum: ponteiro "primeiro"
  - cada elemento aponta para o próximo
  - lista vazia: aponta para nulo
  - uma posição é definida por um ponteiro que aponta para cada nó/elemento da lista

Lista Linear Sequencial

- ordem lógica dos elementos (vista pelo usuário) == ordem física (em memória)
- arranjo de tamanho fixo
- **modelagem:**

```
#define MAX 50

typedef int TIPOCHAVE;

typedef struct {
    TIPOCHAVE chave;
    // etc ...
} REGISTRO;

typedef struct {
    REGISTRO A[MAX];
    int numeroDeElementos;
} LISTA;
```

- **funções comuns:**
  - inicializar a estrutura
  - quantida de elementos
  - exibir elemento
  - buscar elementos
  - inserir
  - excluir
  - reiniciar

I) Inicialização

```
void inicializaLista(LISTA * l) {
    l->numeroDeElementos = 0;
}
```

II) Exibir Lista

```
void exibirLista(LISTA * l) {
    for(int i = 0; i < l->numeroDeElementos; i++)
        printf("%i", l->A[i].chave);
    printf("\n");
}
```

III) Buscar elemento

A. Sem sentinela!

```
int buscaSequencial(LISTA * l, TIPOCHAVE ch) {
```

```
    for(int i = 0; i < l->numeroDeElementos; i++) {
        if(ch == l->A[i].chave)
            return i;
            break;

    }
    return -1;
}
```

B. Com sentinela!

- elemento extra (um registro)
- inserido no final da lista
- contém a chave do elemento buscado
- problema: se não houver espaço?
  - cria-se a lista com uma posição extra, que nunca terá um registro válido
  - basta alterar na modelagem: "... RGISTRO A[MAX+1]; ..."

```
 int buscaSentinela(LISTA *l, TIPOCHAVE ch) {
   int i = 0;
   l->A[l->numeroDeElementos].chave = ch;
   while(l->A[i].chave != ch)
       i++;
   if(i == l->numeroDeElementos)
       return -1;
   else return i;
 }
```

C. Busca binária = O(log n) - só serve quando os **elementos estão ordenados**

IV) Iserir elemento

- lista não estiver cheia e índice passado pelo usuário for válido: desloca todos os elemetnso posteriores uma posição a direita; insere o elemento; soma um no numero de elementos; retorna true; caso contrário, false

```
bool inserirElemento(LIST * l, REGISTRO r, int i) {
    if((l->numeroDeElementos == MAX)
    || (i < 0)
    || (i > l->numeroDeElementos))
        return false;
    for(int j = l->numeroDeElementos; j > i; j--)
        l->A[j] = l->A[j-1];
    l->A[i] = r;
    l->numeroDeElementos++;
    return true;
}
```

V) Exclusão - verifica se o elemento existe na chave passada pelo usuário - se houver: excluir o elemento e desloca todos os outros posteriores para uma posição para a esquerda e diminui em um o numero de elementos; retorna true - se não: retorna false

```
bool excluir(TIPOCHAVE ch, LISTA * l) {
    int pos = buscaSequencial(l, ch);
    if(pos == -1) return false;
    for(int i = pos; j < l->numeroDeElementos - 1; j++)
        l->A[j] = l->A[j+1];
    l->numeroDeElementos--;
    return true;
}
```

VI) Reinicializarção da lista

```
void reinicializarLista(LISTA * l) {
    l->numeroDeElementos = 0;
```

```
}
```

Listas Sequenciais Ordenadas

Inserção

```
bool inserirElementoListaOrdenada(LISTA * l, REGISTRO r) {
    if(l->numeroDeElementos >= MAX) return false;
    int pos = l->numeroDeElementos;
    while(pos > 0 && l->A[pos-1].chave > r.chave) {
        l->A[pos] = l->A[pos-1];
        pos--;
    }
    l->A[pos] = r;
    l->numeroElemento++;
}
```

Busca Binária - O(log n)

```
int buscaBinaria(LISTA * l, TIPICHAVE ch) {
    int esq, dir, meio;
    esq = 0; // primeira posicao olhada
    dir = l->numeroDeElementos-1; // ultima posicao olhada
    while(esq <= dir) {
        meio = ((esq+dir)/2);
        if(l->A[meio].chave == ch)
            return meio; // se o elemento do meio for o elemento buscado, retorna
        else {
            if(l->A[meio].chave < ch)
                esq = meio + 1;
            else dir = meio + 1;
        }
    }
    return -1;
}
```

Obs.: embora a busca na exclusão fique mais eficiente com a busca binária, ainda num vetor ordenado, os elementos devem ser deslocado a fim de ocupar o espaço deixado pelo elemento excluído, não reduzindo a complexidade total do algoritmo

Lista Ligada (linked list) (implementação dinâmica)

Ideia geral

- é uma “array” dinâmica
- as operações de inserção e remoção são menos custosas
- Um ponteiro para o primeiro elemento
- cada elemento tem um ponteiro para indicar seu sucessor
- **Complexidade**
    - Busca: O(n)
    - Inserção: O(1)
    - Remoção: O(1)
- **desvantagens:**
    - não suporta busca binária
    - espaço extra na memória para o ponteiro
    - “no cache friendly”

Tipos

- lista ligada simples (simple linked list)
- lista duplamente ligada (double linked list)
- lista ligada circular (circular linked list)

Lista Simplesmente Ligada

Modelagem

- **representação:**
  - um ponteiro para o primeiro nó da lista
  - o primeiro nó é chamado de cabeça (head)
  - se a lista está vazia, então o valor do ponteiro da cabeça é nulo (null)
  - cada nó da lista consiste em ao menos duas partes: data (int, string, etc...) e ponteiro (para o próximo nó ou um endereço para outro ponteiro)
  - em C, um nó pode ser representado com structs

```
struct Node {
  int data; // informação
  struct Node* next; // ponteiro para o proximo elemento da lista
}
```

-> um jeito mais complexo de fazer:

```
#include<stdio.h>
#include<stdlib.h> // malloc()

typedef int TIPOCHAVE;

typedef struct {
    TIPOCHAVE chave;
    // outros campos ...
} REGISTRO; // ou tipo do "elemento"

typedef struct aux {
    REGISTRO r;
    struct aux* prox;
} ELEMENTO; // ou nó

typedef ELEMENTO * PONT;

typedef struct {
    PONT inicio; // ponteiro para o primeiro elemento
} LISTA;
```

Criar/Inicializar Lista Dinâmica

```
void criaLista(LISTA * l) {
    l->prox = NULL;
}
```

Imprimir Elementos

```
void printList(Struct Node* n) {
    while (n !=NULL) {
        printf("%d ", n->data);
        n = n->next;
    }
}
```

Retornar o número de elemento

- percorrer o número de elementos

```
int tamanho(LISTA * l) {
    PONT posicao = l->inicio;
    int tam = 0;
    while(posicao != NULL) {
        tam++;
        posicao = posicao->prox;
    }
    return tam;
}
```

Buscar elemento

- não suporta busca binária - busca sequencial - complexidade O(n)
- recebe uma chave
- retorna o endereço do elemento (se existir)
- retorna null caso não encontre

a. busca sequencial

```
PONT buscaSequencial(LISTA * l, TIPOCHAVE ch) {
    PONT posicao = l->inicio;
    if(posicao == NULL) return NULL;
    while(posicao != NULL) {
        if(posicao->r.chave == ch) return posicao;
        posicao = posicao->prox;
    }
    return posicao;
}
```

b. busca em lista ordenada pelos valores das chaves dos registros

```
PONT buscaSeqOrd(LISTA * l, TIPOCHAVE ch) {
    PONT posicao = l->inicio;
    while(posicao != NULL && posicao->r.chave < ch) posicao = posicao->prox;
    if(posicao != NULL && posicao->r.chave == ch) return posicao;
    return NULL;
}
```

Inserção de um nó

- 3 maneiras:
  - na frente da lista
  - depois de um nó específico
  - no final da lista
- **Na frente**

```
// é preciso dar um ponteiro para um ponteiro (head_ref) para a cabeça da lista e um
inteiro
void push(struct Node** head_ref, int new_data)  {
    // alocação dinâmica do novo nó
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    // inserir dado
    new_node->data = new_data;

    // fazer o ponteiro de proximo do novo nó ser a cabeça
    new_node->next = (*head_ref);
    (*head_ref)  = new_node;
}
```

- **Depois de um dado nó**

```
void insertAfter(struct Node* prev_node, int new_data) {
    // 1 - verificar se o nó dado aponta para NULL
    if(prev_node == NULL)
        printf("o nó anterior não pode ser nulo\n");
```

```
        return;

    // 2 - alocar o novo nó
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    // 3 - inserir dado
    new_node->data = new_data;

    // fazer o "proximo" do novo nó ser o proximo do nó dado
    new_node->next = prev_node->next;

    // apontar o "proximo" do nó dado para o novo nó
    prev_node->next = new_node;
}
```

- **No final da lista**

```
void append(struct Node** head_ref, int new_data) {

    // 1 alocar o novo nó
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    // 2 criar um apontador para o último
    struct Node * last = * head_ref;

    // 3 atribuir valor do dado
    new_node->data = new_data;

    // 4 proximo do novo nó apontar para nulo
    new_node->next = NULL;

    // 5 se a lista estiver vazia, o novo_nó é também a cabeça (head)
    if(*head_ref == NULL) {
        *head_ref = new_node;
        return;
    }

    // 6 se não estiver vazia, percorrer até o último nó
    while(last->next != NULL)
        last = last->next;

    // 7 apontar o proximo do ultimo nó para o novo nó
    last->next = new_node;
    return;
}
```

-> **Complexidade** - Tempo: O(n) - pode ser otimizado para ser O(1), mantendo-se um ponteiro para indicar a raba da lista (tail)

Remover elemento por posição

- passar a chave do elemento que se quer excluir

```
int remover(tipoLista * l, tipoChave ch) {

    tipoApontador p = pesquisar(l, ch); // precisa de uma funcao de pesquisa de elemento -
ver acima: BUSCA SEQUENCIAL

    int elemento_remover = removePosicao(l, p); // precisa de uma funcao auxiliar para
remover a posicao

    if(elemento_remover == 0) return; // posicao inválida, não é possível remover


}

static int removePosicao(tipoLista * l, tipoApontador p) {
```

```
    tipoApontador p = l->primeiro;

    if(p == NULL) return 0; // posicao invalida

    // para um unico elemento
    if(p == l->primeiro && p == l->ultimo) {
        criar(l);
        free(p);
        return 1;
    }

    // remove do inicio
    if(p == l->primeiro) {
        l->primeiro = l->primeiro->proximo;
        free(p);
        return 1;
    }

    // remove do meio
    tipoApontador aux = l->primeiro; // necessário o auxiliar para não perder a posição
e/ou lista

    while(aux->proximo != NULL && aux->proximo != p) {
        aux = aux->proximo;
    }

    aux->proximo = p->proximo;

    if(aux->primox == NULL) l->ultimo = aux

    free(p);
    return 1;
}
```

Zerar a lista

- criar uma variável auxiliar antes de dar “free” no endereço
- se fizer sem auxiliar, perde o conteúdo do endereço, incluíndo quais os proximos elementos da lista

```
void zerarLista(LISTA * l) {
    PONT endereço = l->inicio;
    while(endereco != NULL) {
        PONT apagar = endereco;
        endereco = endereco->prox;
        free(apagar);
    }
    l->inicio = NULL;
}
```

Inverter Lista - complexidade: - Tempo: O(n) - Espaço: O(1)

```
void reverter(tipoLista * l, tipoChave ch) {

    tipoApontador p = l->primeiro;

    if (p == NULL) {
        printf("Lista vazia.\n");
        return;
    }

    if(p->proximo == NULL) {
        printf("Lista unitária.\n");
        return;
    }
```

```
    tipoApontador anterior = NULL;

    while(p != NULL) {
        p = l->primeiro->proximo;
        l->primeiro->proximo = anterior;
        anterior = l->primeiro;
        l->primeiro = proximo;
    }
    l->primeiro = anterior;
}
```

Lista duplamente ligada (implementação dinâmica)

Ideia Geral

- similar à lista padrão
- ponteiro para o nó anterior é adicionado na struct
- operações de busca, remoção e inserção semelhantes à lista padrão, atentando-se ao ponteiro para o anterior
- ponteiro para o fim pode ser conveniente
- **desvantagens**:
    - não suporta busca binária
    - implementação mais difícil
- **vantagem**:
    - busca pelo elemento que se quer remover

Lista circular

Ideia geral

- pode ser estática ou dinâmica
- estática: vetor; dinâmica: similar à lista ligada padrão

Estática - ponteiros para início e fim - iterar elemento A e B (“fazer a volta”): **utiliza-se o MOD**(%) - busca binária: é preciso considerar que o início não é 0 e que é possível estourar o vetor -> **PIVO** = ((inicio+fim)/2)%(MAX_TAM); - dado inserido no fim: *fim = (fim+1)%(TAM)* - quando inicio == fim -> lista cheia

Dinâmica - l->ultimo->proximo = l->primeiro ao invés de NULL - l->primeiro->anterior = l->ultimo (na duplamente ligada)

Pilas e Filhas, digo, Pilhas e Filas

Pilhas

Geral

- são especializações de lista nas quais as inserções e remoções são realizadas na mesma extremidade - **TOPO**
- tipo “LIFO” - Last In First Out (ultimo elemento inserido é o primeiro a ser removido)
- o TOPO representa o ultimo elemento inserido
- se TOPO == 0, a pilha está vazia

Operações e noemclaturas

- inserir = empilhar (PUSH)
- remover = desempilhar (POP)
- criar, topo, vazia, inverte, conta, imprime, etc.

Empilhar

```
// estática
s.topo = s.topo + 1;
s[s.topo] = valor;

// dinâmica
void push(tipoPilha *p, tipoChave ch) {

    tipoApontador novo = (tipoApontador)malloc(sizeof(tipoNo));
    if(novo == NULL) // memória cheia
        return;
    novo->elemento.chave = ch;
    novo->proximo = p->topo;
    p->topo = novo;
}
```

Desempilhar

```
// estática
if SACK-EMPTY(S)
    error "underflow"
else S.topo = S.topo - 1;
    return S[S.topo + 1];

// dinâmica
void pop(tipoPilha *p) {
    if(vazia(p))
        return;
    tipoApontador aux = p->topo;
    p->topo = p->topo->proximo;
    free(aux);
}
```

Aplicações

- undo/redo (desfazer, refazer) etc.
- notação pós-fixa/infixa

Filas

Geral

- lista cujas inserções e remoções são feitas em extremidades opostas (FRENTE/TRÁS ou HEAD/TAIL ou CABEÇA/RABA)
- FIFO: First In First Out (primeiro a entrar é o primeiro a sair)
- o elemento removido é sempre o primeiro elemento (início)
- final - inicio + 1 => indica quantos elementos tem na fila

Operações e nomenclaturas comuns

- inserir = ENFILEIRAR (QUEUE)
- remover = DESENFILEIRAR (DEQUEUE)
- etc.

Enfileirar

```
void enfileirar(tipoFila *f, tipoChave ch) {
    tipoApontador novo = (tipoApontador)malloc(sizeof(tipoNo));
    if(novo == NULL) return; // memória cheia
    novo->elemento.chave = ch;
    novo->proximo = NULL;
    if(f->primeiro == NULL)
        f->primeiro = novo;
    else f->ultimo->proximo = novo;
    f->ultimo = novo;
    contador++; // opcional
}
```

Desenfileirar

```
void desenfileirar(tipoFila *f) {
    if(f->primeiro == NULL)
        return; // fila vazia
    if(f->primeiro == f->ultimo) {
        f->ultimo = NULL;
        return;
    }
    tipoApontador aux = f->primeiro;
    f->primeiro = f->primeiro->proximo.
    free(aux);
}
```

Referências

UNIVESP

IME

diegofurts

Geeks for Geeks

laura