Estrutura de Dados

Created	@March 28, 2023 12:29 PM
Tags	

▼ Ponteiro sem tipo e ponteiro para função

PONTEIRO SEM TIPO

> 📌 É um ponteiro que não possui um tipo de dados específicos associado a ele. É um ponteiro genérico.

Um ponteiro sem tipo é declarado usando o tipo de dados `void*`, o qual representa a ausência de um tipo e o asterisco indica que o ponteiro aponta para um endereço de memória, mas sem informar qual tipo de dado esse endereço contém.

`Void`: Ausência de um tipo de dado

`*`: aponta para um endereço de memória

É útil em situações que precisamos trabalhar com diferentes tipos de dados, mas não sabe de antemão qual tipo de dado será usado.

Para usar um ponteiro sem tipo, devemos converter o ponteiro para o tipo apropriado antes de usar. Isso é feito usando um operador de conversão de dados.

```
int valor = 42;
void *ptr = &valor; // armazena o endereço de memória da variável 'valor'
int *ptr_int = (int *) ptr; // converte o ponteiro sem tipo para um ponteiro de in
printf("%d", *ptr_int); // imprime o valor inteiro armazenado no endereço apor
```

A conversão do ponteiro é feita usando a sintaxe `tipo*`. Criando um novo ponteiro do tipo desejado que aponta para o mesmo endereço de memória.

📌 No momento que o ponteiro for referenciado (quando usar o endereço para acessar o valor na memória), é necessário informar ao compilador o tipo de dado (usando cast) para que o valor seja acessado corretamente

📌 Por definição, um ponteiro ocupa sempre 4 bytes de memória, independentemente do tipo de dado a ser apontado por ele.

Este é o ponto-chave que possibilita a declaração de ponteiros sem tipo.

Contudo, é importante declarar o tipo de dado depois para que o compilador saiba quantos bytes de memória deverão ser considerados.

> O operador ponto “.” é usado para acessar membros de uma estrutura ou união quando o objeto é uma variável não ponteiro.

> Já o operador seta “->” é usado para acessar membros de uma estrutura ou união quando o objeto é um ponteiro para a estrutura ou união.

PONTEIRO PARA FUNÇÃO

É um tipo de variável que armazena o endereço de memória de uma função. É como um ponteiro comum, que armazena o endereço de memória de uma variável, mas em vez disso armazena o endereço de memória de uma função. Úteis quando é necessário passar função como argumento para outra função..

📌 Podem “apontar” para funções, as quais são armazenadas na memória e associadas a um endereço

sintaxe: declaração de um ponteiro

```
tipoDeRetorno (*nomePonteiroParaFuncao)(listaArgumentos);
```

onde:

tipo de retorno: tio de dado que a função retorna.

lista de argumentos: lista de argumentos que a função recebe.

Exemplo:

O BubbleSort só é capaz de ordenar vários tipos de dados pois:

1. A declaração de vetor de ponteiros sem tipo
2. Uso de ponteiro para função comparar os valores durante a ordenação

ALLOC E MALLOC

São funções usadas para alocar espaço de memória dinâmica durante a execução do programa.

malloc: aloca um bloco de memória de um tamanho específico em bytes.

▼ Recursividade

Recursividade é uma técnica utilizada em programação na qual uma função é definida em termos de si mesma.

> 📌 Uma função recursiva é uma função que chama a si mesma repetidamente até que uma condição de parada seja alcançada.

É importante ter cuidado ao usar recursão, pois se a recursão não for escrita corretamente, pode causar um loop infinito e causar um estouro de pilha. É necessário ter certeza de que a condição de parada é alcançada em algum momento.

```
int fatorial(int n) {
   if (n == 0) { //condição de parada CASO BASE
      return 1;
   } else {
      return n * fatorial(n - 1); //recursividade, a função chama a si mesma
   }
}
```

▼ Encadeamento de Dados

27/ 03/ 2023

struct + ponteiro

- O objetivo é criar estruturas abstratas, genéricas e dinâmicas

- quando se tem um vetor, temos uma estrutura de memória de organização do vetor onde cada posição é contínua a outra 1, 2, 3… até o tamanho do dado que ali é armazenado

Graças a isso pode ser utilizado aritmética de ponteiros:

```
int *p //ponteiro para inteiros

int i;

p= &i;

p++; //incrementando 1 em um ponteiro
```

- O ponteiro vai somar mais um? não, vai fazer o apontamento de um valor, vai apontar para o próximo inteiro.

Com o encadeamento de dados usa-se ponteiros que pegam informações, colocam dentro de uma struct e vão encadeando uma na outra para gerar uma estrutura de dados.

|7|2|5|4|6 | —> dados que estarão separados na memória

|7| | -> |2| | -> … |6|NULL | -> não apontará para ninguém

- mesma estrutura usada, alocada muitas vezes.

- a struct vai ter dois campos: o info e o prox

info: informação

prox: vai ser o ponteiro que aponta para o próximo nó

ESTRUTURA DO NÓ

```
struct noh{

<tipo_dado (int, float...)>     info;        //|7| → info | | → prox

struct noh *prox;  /*o ponteiro aponta para uma struct nó, onde ele pode,

apontar para ele mesmo*/

}
```

sendo possível também criar uma estrutura circular, onde o último apontaria para o primeiro.

- o prox aponta para uma memória que é do tipo struct
- se eu coloco na info um ponteiro para void, apontará para um endereço e será possível encadear dados de diferentes tipos (int, float, char…). Contudo, se eu declaro no campo info um tipo determinado (int), apenas posso usar valores do tipo int.

COMO COLOCAR MAIS UM NÚMERO NESSES DADOS ENCADEADOS?

1 - Criar o espaço para guardar o número 8, com isso preciso alocar essa memória

```
struct noh *p= malloc(sizeof(struct noh));
```

O malloc devolverá um ponteiro p que aponta para a estrutura que foi alocada, onde o p terá que ser do tipo struct nó.

2 - Dar o valor desejado

p.info = 8; Contudo, como estamos falando de um ponteiro, precisa usar a seta para acessar os campos daquela struct

p->info=8 O operador -> será tipo o ponto, o qual é ?

- O operador ponto “.” é usado para acessar membros de uma estrutura ou união quando o objeto é uma variável não ponteiro.
- Já o operador seta “->” é usado para acessar membros de uma estrutura ou união quando o objeto é um ponteiro para a estrutura ou união.

3 - Quem é o próximo do 8?

p->prox = NULL (para ser o último)

—----------

A organização de uma lista encadeada é composta por uma cabeça (1) e uma cauda (último)

Cauda é um ponteiro para tipo struct noh e estaria apontando para o 6 antes da inclusão do 8. Após colocar o 8, a cauda aponta para o 8.

O que alterar? O prox de 6 que eestá valendo NULL, para isso eu preciso pegar a cauda e apontar para esse nó

cauda->prox= p (aponta para p, pois o ponteiro p está apontando para o 8 que será a nova cauda)

cauda= p; (a cauda é o nó apontado pelo p)

—----------

TIPO ABSTRATO DE DADO (TAD)

Definição que te diz o que você tem disponível para usar, mas não tem a informação de como aquilo foi feito.

typedef -> definição de tipo

```
typedef struct noh Noh; //→ existe uma definição de tipo

typedef Noh*     pNoh; //→ ponteiro para nó
```

```
//-------------------------------------------------------------------------
//descritor de lista

typedef struct dLista DLista;

typedef DLista        pDLista; //→ ponteiro para descritor de lista
```

Para criar uma lista, deverei fazer a seguinte chamada:

criarLista();

Cria o descritor da lista, o qual a representa

| | | |

Como colocar uma informação?

- Uma operação incluirInfo, podendo fazer uma chamada para incluir cada valor à lista
- Ao chamar essa operação, ele criará um nó

incluirInfo (..., 7) —> | 7 | |, será o único, a cabeça e a cauda apontaram para ele

incluirInfo (...,2) —> aloca a memória e inclui o 2, o qual será a nova cauda (colocando sempre no final)

- Para adicionar ao inicio, usaria a operação incluirInfoInicio e esse valor seria alocado e tomaria a posição de cabeça

(o void e o int vai mostrar um valor ou posição nas operações)

- O primeiro parâmetro sempre será o ponteiro descritor de lista, se eu quero incluir uma info, preciso dizer em qual lista eu desejo incluir.

▼ Lista Lineares - Teoria

INTRODUÇÃO

Lista linear: estrutura de dados composta por uma sequência de elementos com alguma relação entre eles. Para alcançar x termo, devemos acessar os 4 termos anteriores.

- cada elemento da lista é armazenado em um vetor da lista
- Ex: O primeiro elemento da lista linear a é armazenado na primeira posição do vetor e assim sucessivamente.

Contudo, definir uma lista usando um vetor tem algumas desvantagens devido à natureza estática dos vetores, já que seu tamanho deve ser obrigatoriamente definido previamente. Impondo limitações ao dinamismo das listas lineares quando implementadas usando vetores.

UTILIZAREMOS ENCADEAMENTO DE DADOS COMO BASE PARA A CRIAÇÃO DE UMA LISTA LINEAR

ESTRUTURA DE DADOS

- O encadeamento é uma estrutura denominada nó a qual contêm a informação a ser armazenada e a referência ao próximo nó da lista
- a informação do nó vai depender do contexto da lista, já que o próximo nó que será apontado é sempre do mesmo tipo

A estrutura de um nó é feita a partir da definição de um tipo de dado heterogêneo(`struct`), contendo os campos info e prox → |info|prox|

- O campo **info** precisa ser declarado com um tipo de dado (substituído na parte tipo_dado).

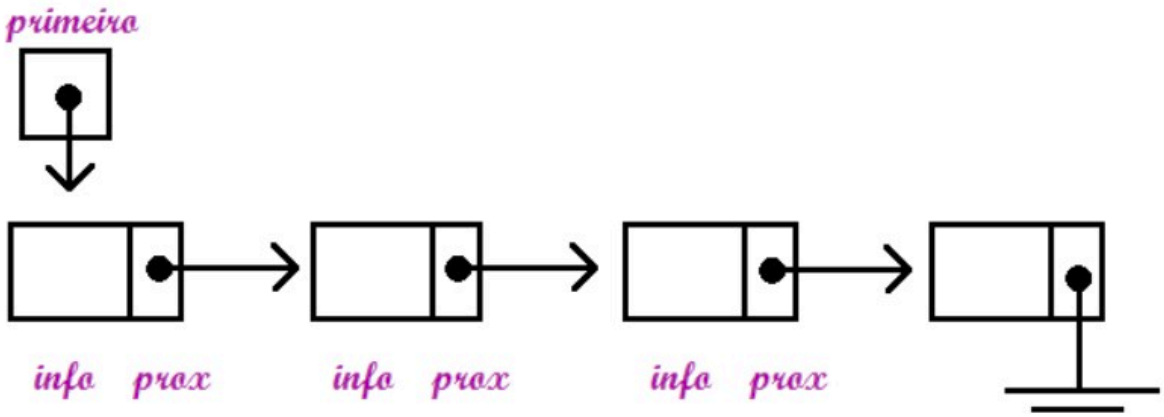
```
struct noh{
<tipo_dado> info;
```

```
struct noh *prox;
};
```

- O campo **prox** é um ponteiro para o tipo de dado que se refere a própria estrutura do nó.
- Aponta para a memória de outro nó da lista.

→ Pode parecer que se trata de um declaração recursiva, mas não é. A declaração permite que o campo prox aponte para o endereço de memória do próprio nó onde está contido.

- A junção de vários nós é um alista encadeada.
- O inicio é indicado por uma informação chamada de **cabeça.**
- Seu final é indicado por um informação chamada de **cauda.**



DEFINIÇÃO DE TAD PARA A LISTA LINEAR

> TAD em C é uma forma de definir um tipo de dado abstrato que expõe apenas as operações disponíveis nesses dados, sem expor os detalhes de implementação. Isso permite que o código que utiliza os dados se concentre apenas nas operações disponíveis, sem precisar se preocupar com os detalhes de implementação.

- A especificação da lista linear é formalizada seguindo o princípio da abstração de dados com a definição de um tipo abstrato de dado para representar suas funcionalidades
- Com a estruturação dos dados na forma linear, pode ser feitas diversas operações sobre ela, como incluir ou retirar um elemento.
- Há dois tipos de declaração, tipos de dados e operações
- A primeira parte do TAD tem as definições dos tipos de dados requeridos para a estruturação da lista linear
- Contém as declarações dos tipos de dados (Noh e pNoh) relacionados à definição do nó
- contém a declaração da estrutura que representa o descritor de lista que armazena as informações gerais sobre a lista linear, como a cabeça, calda e quantidade de nós
- **A lista linear sempre será representada, acessada e manipulada por meio do descritor**
- A relação entre a lista e o descritor vai ser sempre 1:1, cada lista sempre será representada por um único descritor
- Tem definições de ponteiros para função
- Também são listadas quais operações poderão ser realizadas sobre os elementos da lista linear

FUNÇÕES

CRIAR LISTA: Responsável por instanciar o descritor que vai representar a lista, o qual será retornado como resultado da função

→ O descritor terá seus campos inicializados com valores padrões para indicar uma lista vazia (cabeça e calda NULL e quantidade 0)

→ retorna um pDLista que será um ponteiro para o descritor da lista, por médio dele serão realizadas as atualizações no descritor da lista

INCLUIR INFO: Representa a operação de inclusão de uma nova info na lista.

→ a assinatura possui dois argumentos, o pDLista e void* (ponteiro para o descritor da lista a ser alterada e a informação que será incluída no final da lista)

EXCLUIR INFO: Representa a operação de remover uma informação da lista linear.

→ a função possui três argumentos, pDLista, void* e FuncaoComparacao (ponteiro para o descritor da lista que será alterada, a info a ser removida e a função que vai compara as informações contidas na lista para remover a informação)

→ a função retorna um valor inteiro indicando se a exclusão foi realizada ou não (0 se não e 1 se sim)

CONTÉM INFO: Representa a operação que verifica a existência de uma determinada informação na lista.

→ a função possui 3 argumentos, os mesmos da excluir info, mas essa função não irá fazer nenhuma alteração, apenas vai verificar a existência da informação na lista.

IMPRIMIR LISTA: Representa a operação para exibição das informações contidas na lista.

→ a função possui 2 argumentos, pDLista e FuncaoImpressao (ponteiro para descritor da lista que será impressa e a função responsável pela impressão da informação contida em cada nó)

DESTRUIR LISTA: Representa a operação de destruição da lista, a liberação da memória alocada para armazenar todas as informações da lista

→ a função possui apenas um argumento pDLista, que vai representar o ponteiro para descritor da lista que será destruída.

→ O descritor da lista não será destruído, apenas vai ser liberada a memória alocada para cada nó da lista.

→ Os valores dos campos do descritor serão ajustados para se referir a uma lista vazia

DUPLICAR LISTA: Representa a operação de geração de uma cópia da lista linear.

→ a função possui um único argumento pDLista, o qual representa a lista a ser duplicada

→ essa função retorna um ponteiro para outro descritor de lista que será a cópia da lista original

DIVIDIR LISTA: Representa a operação de dividir uma lista em duas.

→ a função possui dois argumentos, pDLista e int (ponteiro descritor da lista a ser dividia e a posição onde a lista será dividida)

→ as informações até a posição dada, incluindo ela, ficarão na lista original e as seguintes serão alocadas na nova lista

IMPLEMENTAÇÃO DAS OPERAÇÕES DA LISTA LINEAR

- É usado dois tipos de dados heterogêneos para estruturar a lista encadeada

```
#ifndef STRUCTS_H
#define STRUCTS_H

struct noh{
void *info;
struct noh *prox;
};

struct dLista{
int quantidade;
struct noh *primeiro;
struct noh *ultimo;
}
```

- O struct noh é usado para estruturar as informações da lista encadeada
- O campo prox foi declarado como um ponteiro para a prórpia estrutura noh, isso vai permitir o encadeamento das informações contidas na lista linear

printa a lista ao contrário e depois printa na ordem

```
if (pd == NULL)
   {
      printf("Lista nao existe ainda, nao eh possivel incluir!");
      return;
```

```
    }
    /* aloca memoria para guardar a nova info */
    pNoh pNovo = malloc(sizeof(Noh));

    /* guarda a informa� �o no campo "info" do novo noh alocado */
    pNovo->info = info;
    /* como a nova info ficar na �ltima posicao da lista,
      n�o ter� pr�ximo */
    pNovo->prox = pd->primeiro;

    /* salva a referencia ao ultimo noh da lista antes de atualizar o descritor */
    pNoh auxUltimo = pd->ultimo;
    /* o novo noh ser� identificado como �ltimo da lista pelo descritor */
    pd->primeiro = pNovo;
    /* a lista tem mais um noh */
    pd->quantidade++;

    /* Se o novo noh � o �nico da lista, o descritor
      precisa atualizar tamb�m o ponteiro para o primeiro noh da lista */
    if (pd->primeiro == NULL)
       pd->primeiro = pNovo;
    else
    {
       /* se n�o for o �nico da lista, coloc�-o no final da lista */
       pd->primeiro = pNovo;
    }
```

▼ free

a função apaga o nó ou a informação que está nele?

ChatGPT

A função `excluirInfo` remove o nó da lista e, em seguida, libera a memória alocada para esse nó usando a função `free`. A informação armazenada no nó é perdida quando o nó é removido da lista e sua memória é liberada.

Observe que a função `excluirInfo` recebe um ponteiro `void` para a informação a ser excluída (`info`) como um de seus argumentos, mas esse ponteiro não é usado para excluir a informação em si. Em vez disso, ele é usado para comparar a informação armazenada em cada nó da lista com a informação que precisa ser excluída.

Portanto, se você deseja manter a informação armazenada no nó que será excluído, você precisa fazer uma cópia dessa informação antes de chamar a função `excluirInfo`. Além disso, é importante lembrar que, se a informação armazenada no nó contém outras alocações de memória dinâmica, você precisa liberar essas alocações de memória separadamente antes de chamar a função `free` no nó em si.

então a função free apaga o nó e perde a info?

ChatGPT

Sim, isso mesmo. A função `free` libera a memória alocada para o nó em questão, mas não faz nada em relação à informação armazenada dentro do nó. Quando a memória do nó é liberada usando `free`, a informação armazenada dentro do nó é perdida.

Portanto, é importante lembrar que, antes de chamar a função `free`, você deve verificar se ainda precisa da informação armazenada no nó. Se precisar manter essa informação, você precisa fazer uma cópia da informação antes de chamar `excluirInfo` ou outra função que remova o nó da lista.

Além disso, se a informação armazenada no nó contiver outras alocações de memória dinâmica (por exemplo, se for uma

estrutura que contém ponteiros para outras estruturas ou matrizes alocadas dinamicamente), você também precisará liberar a memória alocada para essas estruturas ou matrizes separadamente antes de chamar `free` no nó em si.

▼ Listas Lineares - notas de aula

Encadeamento de dados

```
struct noh {

void * //info

struct noh*  //prox
}
```

- Para acessar o ultimo elemento da lista precisa passar por todos os anteriores

```
struct DLista {
 struct noh* cabeca
 struct noh* cauda
 int qtde
}
```

- TAD tipo abstrato de dados
- Noh - associado a uma struct noh
- pNoh - ponteiro para struct noh

- é usado um ponteiro para função para imprimir os valores de uma lista pois não sabe qual string de comparação usar para fazer o printf

`typedef void (*FuncaoImpressao) (Void*)`

DLista = descritor da lista

pDLista = ponteiro para descritor de lista

As listas em C são implementadas usando o conceito de encadeamento de dados. Cada elemento da lista é um nó que contém uma informação e um ponteiro para o próximo nó na lista. A lista é representada por um descritor de lista, que contém um ponteiro para a cabeça da lista, um ponteiro para a cauda da lista e o número de elementos na lista.

Para criar uma lista, é necessário criar um descritor de lista e inicializá-lo com a cabeça e a cauda igual a `NULL` e o número de elementos igual a zero. Em seguida, é possível adicionar elementos à lista usando a operação `incluirInfo`, que cria um novo nó com a informação passada como parâmetro e o adiciona ao final da lista.

Para imprimir os valores da lista, é possível usar um ponteiro para função que recebe um ponteiro para o valor a ser impresso. Isso é necessário porque não se sabe qual string de formatação usar no `printf` para cada tipo de dado que pode ser armazenado na lista.

```
#define CRIAR_LISTA_H

#include "O structs.h"

pDLista criarLista(){
//aloca memória para o descritor
   pDLista pd = malloc (sizeof(DLista))
```

```
//sera os compos com os valores default
pd → quantidade = 0
pd → primeiro = NULL
pd → ultimo = NULL
return pd;
}
```

```
#define INCLUIR_INFO_H

void incluirInfor (pDLista pd, void* info){
   if (pd == NULL){
      printf (" lista nao existe ainda, nao eh incluir!);
}
//aloca memoria para guardar a nova infor
   pNoh pNovo = malloc(sizeof(Noh));

//guarda a informacao no campo "info" do novo noh alocado
   pNovo → info = info;

//como a nova info fica na ultima posicao da lista nao terá prox
   pNovo → prox = NULL;

   //a lista tem mais um noh  pd = ponteiro para o descritor
   pd → quantidade++;


//salva a referencia ao ultimo noh da lista antes de atualizar o descritor
   pNoh auxUltimo = pd → ultimo; // recebe o que tem no ponto ultimo do des

//o novo noh sera identificado como ultimo da lista pelo descritor
pd → ultimo     = pNovo //oq tem dentro do ponteiro novo vai ser colocado no

//se o novo noh eh o unico da lista, o descritor precisa atualizar tambem o po
if (pd → primeiro == NULL )
```

```
        pd → primeiro = pNovo;
    else{
    // se nao for o unico da lista coloca-o no final da lista
```

10/04/2023

TAD em C é uma forma de definir um tipo de dado abstrato que expõe apenas as operações disponíveis nesses dados, sem expor os detalhes de implementação. Isso permite que o código que utiliza os dados se concentre apenas nas operações disponíveis, sem precisar se preocupar com os detalhes de implementação.

- TAD_ListaLinear.h
- Primeira coisa que ela faz é implementar tudo oq está dito no tipo abstrado de dados

Include para cada implementação escritos no tipo abstrato de dado (TAD)

No exemplo,o professor fez cada função em um arquivo separado

O que fazer no trabalho que esta no final do capítulo?

Se tiver uma questão na prova, assim como na lista

```
//código usando quando se quer percorrer uma lista
PNoh pAux = pd→primeiro;
While(pAux != NULL)
```

```
{
//função

pAux = pAux → prox
}
```

20/04/23

- praticamente em toda a função que vai percorrer a lista o `aux = pd → primeiro`

Dicas exercícios da lista:

- usar o material como fonte
- O primeiro não precisa fazer, é só dizer o seguinte “eu preciso de um ponteiro para o descritor de lista, é para tentar esclarecer isso”
- pq eu preciso do pd? pq dependendo da função ela vai ter que alterar o descritor, para isso vai ser usado um ponteiro (para q a sua alteração seja refletida no programa)
- 2, 3 e 4 ja estão feitas
- 5 vai dividir a lista baseado no valor
- 6 unir listas

▼ Fila e Pilha - Teoria

FILA

- forma de organização de elementos
- caracterizada pela organização de seus elementos de forma que o primeiro elemento inserido deve ser o primeiro a ser retirado da fila

- a disciplina de acesso da fila é conhecida como FIFO (firs in first out)
- percebe-se que os elementos circulam dentro da fila, a medida que retiramos o primeiro e o recolocamos na fila (agora será o ultimo)
- a fila é uma lista linear com uma disciplina de acesso aos elementos para garantir a regra FIFO
- a implementação da regra é obtida pela escolha de uma das extremidades da lista linear como início da fila e a extremidade oposta como o fim da fila
- uma vez que garantimos a inserção de novos elementos sempre na extremidade final da fila e, em contrapartida, a retirada de elementos seja sempre realizada na extremidade indicada como início, teremos a implementação da lógica da estrutura de dados fila

Definição do TAD para Fila

- a especificação da fila é formalizada também seguindo o princípio da abstração de dados com a definição de um tipo abstrato de dados (TAD) para representar suas funcionalidades
- nas primeiras linhas temos a declaração dos tipos de doados (DFila e pDFila), os quais representam as definição de tipos para o descritor da fila e o ponteiro para o descritor
- também há as definições de operações associadas à fila: criar, enfileirar, desenfileirar e filaVazia

criar fila: seu propósito é alocar memória para armazenar o descritor da fila e inicializar seus campos para representar a configuração de uma fila que inicialmente será vazia

enfieleirarInfo: inclui uma informação na fila, respeitando a disciplina de acesso FIFO. A info sempre seŕa incluída sempre na mesma extremidade da lista que é usada como final da fila.

desenfileirarInfo: remove uma informação do início da fila. A remoção vai ser sempre o início da fila. Ademais, a informação retirada do início da fila é retornada pela função.

filaVazia: sinaliza se a fila tem alguma informação. Retorna:

0 = fila vazia

≠0 = fila não vazia

Implementação do TAD de fila

- vamos usar como base a implementação de lista lista linear, tendo em vista que uma fila pode ser tratada como uma lista com a disciplina de acesso
- é feito um `#include` da biblioteca de lista.h no fila.h
- como a implementação de fila está baseada na lista.h, precisamos apenas definir a estrutura do descritor da fila (dFila)
- A estrutura contem apenas o copo pdLista, o que é o ponteiro para o descritor da lista linear usada para armazenar as infos da lista que vai representar a fila.

```
struct dFila{
   pDLista pdLista;
};
```

- o descritor da fila é representado por um descritor de uma lista linear
- pd vai apontar para um pdLista (que será o struct DFila), o qual apontará para um descritor de lista (o struct DLista)

- em enfileirar Info, o argumento pdFila→pdLista deixa claro a dependência da implementação da fila em relação à biblioteca da lista linar

▼ Fila e Pilha - notas de aula

São estruturas de dados com disciplina de acesso aos elementos.

FIFO → First In First Out (lista)

LIFO → Last In First Out (pilha)

Tem características particulares, o acesso aos seus elementos não é feito de forma aleatória:

- Na lista, se quisermos pegar o 5 elemento vc percorre a lista e acessa. Os demais elementos ficaram como antes.
- Na fila, o primeiro elemento que entra na fila é o primeiro a sair da fila.
- Na pilha, o ultimo elemento a entrar na pilha é o primeiro a sair. O primeiro elemento que foi colocado será o ultimo a sair.

Em uma fila, se quero acessar o segundo item, devo antes pegar o primeiro item da lista. Mudando a fila. O mesmo ocorre na pilha e não ocorre na lista.

A biblioteca de lista está contida na biblioteca de fila, minha fila é uma lista

- Enfileirar pode ser considerado o incluirInfo que está presente na lista.h

A biblioteca lista.h também está contida na biblioteca da pilha.h.

- **programa para excluir uma fila**
- jogar da fila para a pilha inverte a ordem da fila

Desempilhar retira o ultimo que entrou na pilha

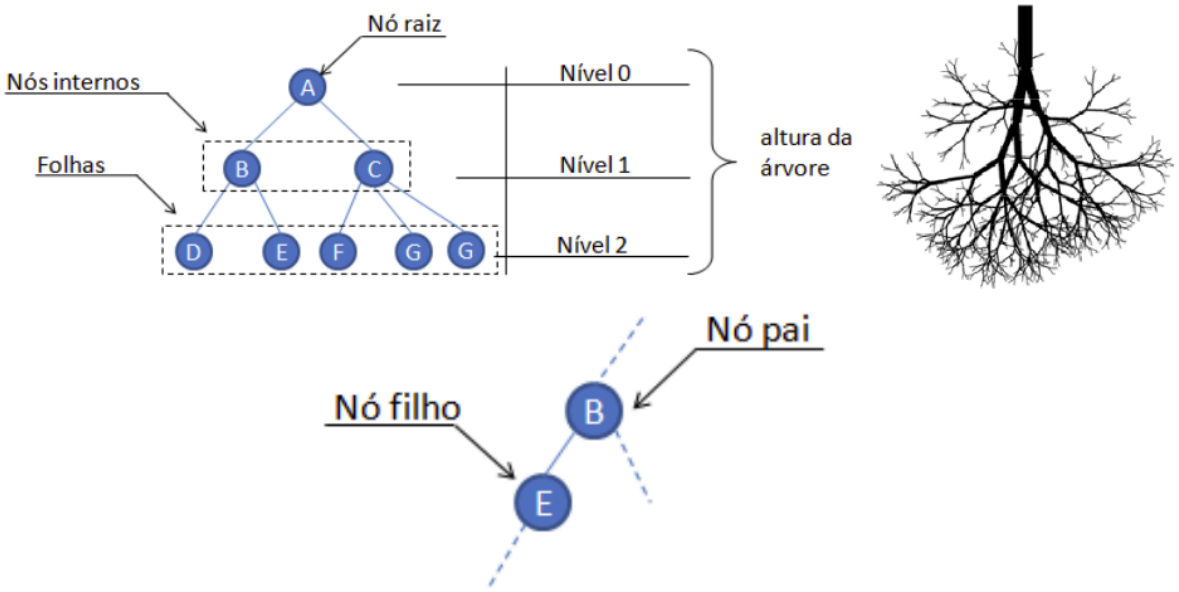
- **programa para passar um decimal para binário**

O #idndef, #define e #endif tem uma função importante:

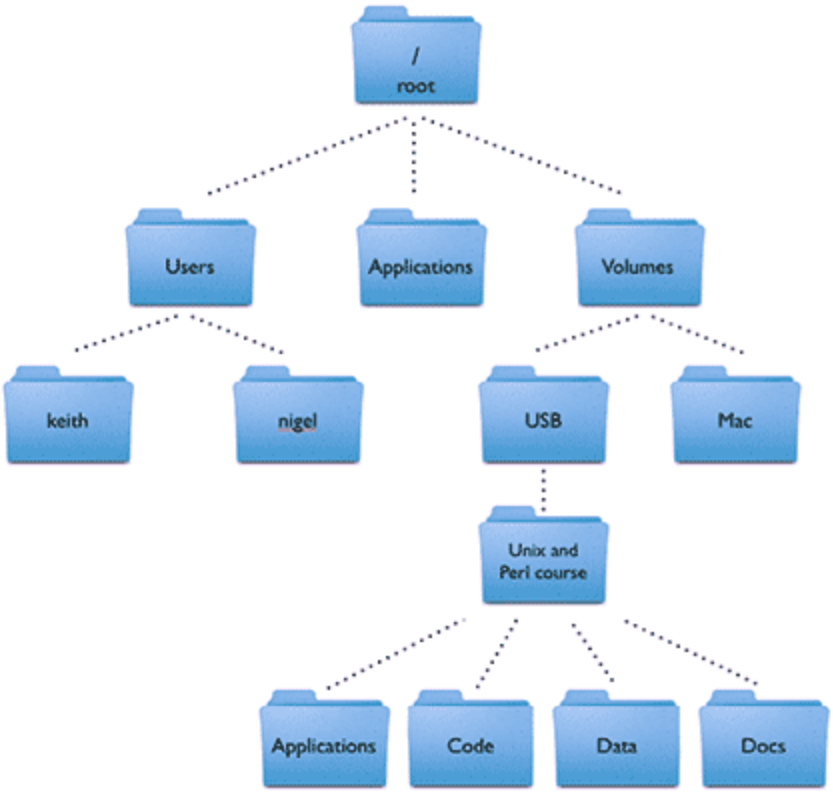
- são diretivas de compilador, instruções dadas ao compilador na hora de compilar o código
- diz que, se não foi definido o, por exemplo, INCLUIR_INFO_H, defina-o
- se ele tentar compilar esse mesmo código para outra inclusão, não funcionará

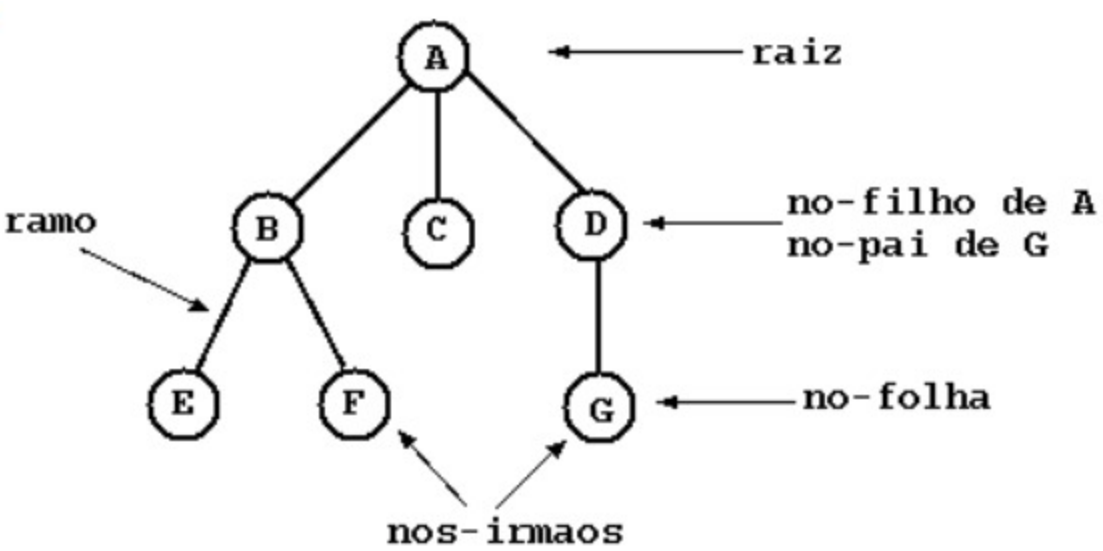
▼ Árvores

- estruturas não lineares
- seus elementos, designados por nós, podem ter mais de um predecessor ou mais de um sucessor



- as árvores são um caso especial de grafos em que cada nó tem zero ou mais sucessores, mas tem apenas UM predecessor, exceto o primeiro nó, a raiz da árvore
- são estruturas naturalmente adequadas para representar informação organizadas em hierarquias (ex: estrutura de pastas)





- o predecessor de um nó se chama nó-pai
- seus sucessores sãos os nós-filhos
- o grau de um nó é o numero de nós-filhos que descendem do nó-pai

- um nó-folha não tem filhos, tem grau 0
- um nó-raiz não tem pai
- os arcos que ligam os nós são chamados de ramos
- chama-se caminho ua sequência de ramos entre dois nós
- o comprimento de um caminho é o número de ramos nele contido
- a profundidade de um nó n é o comprimento do caminho n até à raiz; a profundidade da raiz é 0
- a altura de um nó é o comprimento do caminho desde esse nó até o seu folha mais profundo
- **se navega em uma árvore sempre de cima para baixo**

Raiz: primeiro elemento, que dá origem aos demais

c) Nó: qualquer elemento da árvore

d) altura de uma árvore: quantidade de níveis a partir do nó-raiz até o nó mais distante (a raiz está no

nível zero)

e) grau de uma árvore: número máximo de ramificações da árvore

f) grau de um nó: número máximo de ramificações a partir desse nó

g) filho: sucessor de um determinado nó

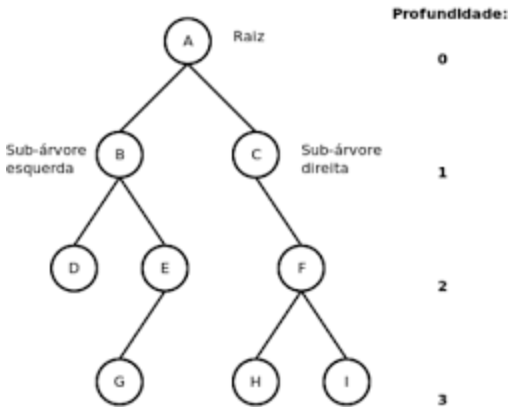
h) pai: único antecessor de um dado elemento

i) folha: elemento final (sem descendentes, sem filhos)

j) nível de um nó: é distância que um nó tem em relação ao nó raiz (a raiz está no nível 0)

ÁRVORE N-ÁRIA

Árvore binária de pesquisa/busca



Uma árvore constituída por um conjunto finito de nós. Se o conjunto for vazio, a árvore diz-se vazia, caso contrário obedece às seguintes regras:

→ possui um nó especial, a raiz da árvore

→ cada nó possui no máximo dois filhos, filho-esquerdo e filho-direito

→ cada nó, exceto a raíz, possui exatamente um nó-pai

- uma árvore binária totalmente preenchida é uma árvore binária em que todos os nós, exceto nós-folha, têm 2 filhos
- uma árvore binária perfeita é uma árvore binária totalmente preenchida em que todos os nós-folha estão à mesma profundidade
- //uma árvore binária completa é uma árvore binária em que todos os níveis, exceto possivelmente o último, estão completamente preenchidos e todos os nós estão mais à esquerda possível.

- a profundidade de uma árvore binária é determinada pelo maior nível de qualquer nó folha

→ Em uma sequência de valores dada, o primeiro valor fica na raiz da árvore, os seguintes são à esquerda ou direita da raiz, obedecendo à relação de ordem, como folhas e em níveis cada vez mais baixos

ordem: 14, 6, 17

```
     14      →6 é maior que 14? não, é menor, então vai para a esquerda
   /    \     →17 é maior que 14? sim, é maior, então vai para a direita
  6      17
```

percorrer a árvore

```
visita(raiz)
   if (raiz != NULL){
      visita (raiz→esq)
      imprime (raiz→info)
      visita (raiz→dir)
}

main

   visita(14); //primeiro visita a esquerda do 14
```

APLICAÇÃO

```
//struct descritor da arvore
struct dArvore {
pNohArvore raiz;
int quantidadeNohs;
int grau;
}
```

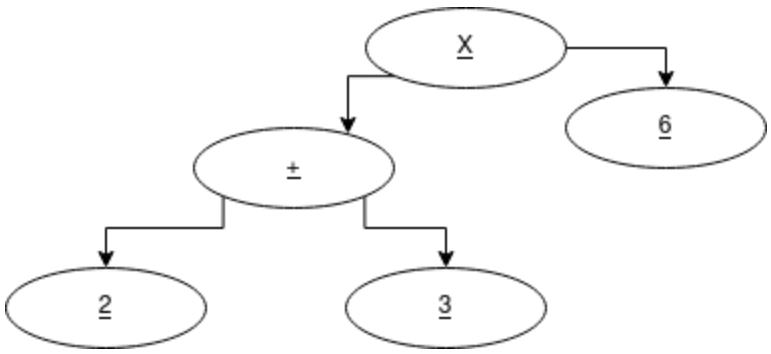
```
//struct descritor do nohArvore
struct nohArvore{
void *info;
pNohArvore esquerda;
pNohArvore direita;
}
//no caso de uma árvore nAria
struct nohArvores{
void *info;
pDLista filhos; //aponta para um descritor lista de filhos
}

//// tipos de dados
//descritor da arvore
typedef struct dArvore DArvore;
typedef NohArvore* pNohArvore;
```

CAMINHAMENTOS PADRÕES EM ÁRVORES BINÁRIAS

→ notação para expressões aritméticas

(2+3)* 6 → notação infixa (operador fixado entre os operandos)



As folhas são os valores

EM ORDEM (in order)

- tem a ver com árvore binária, não necessariamente de pesquisa
- esquerda, raiz(imprimir), direita
- fazendo o caminhamento em ordem em uma árvore de expressão aritmética, terá a operação em ordem 2 + 3 * 6

PRE ORDEM

- raiz(imprime), esquerda, direita
- * + 2 3 6 → notação pré fixada

POS ORDEM

- esquerda, direita, raiz
- 6 3 2 + *

FUNÇÃO QUANTIDADE FOLHAS

```
int qtdeFolhas (pNohArvore raiz){
   if ( raiz → esquerda == NULL && raiz → direita == NULL
      return 1;

   if (raiz == NULL)
      return 0; //nenhuma folha a direita e esquerda

   return qtdeFolhas (raiz→esquerda) + qtdeFolhas (raiz→esquerda);
}
```

→ a quantidade de folhas de uma raiz é a soma dos números de folhas a esquerda e a direita

FUNÇÃO QUANTIDADE NÓS

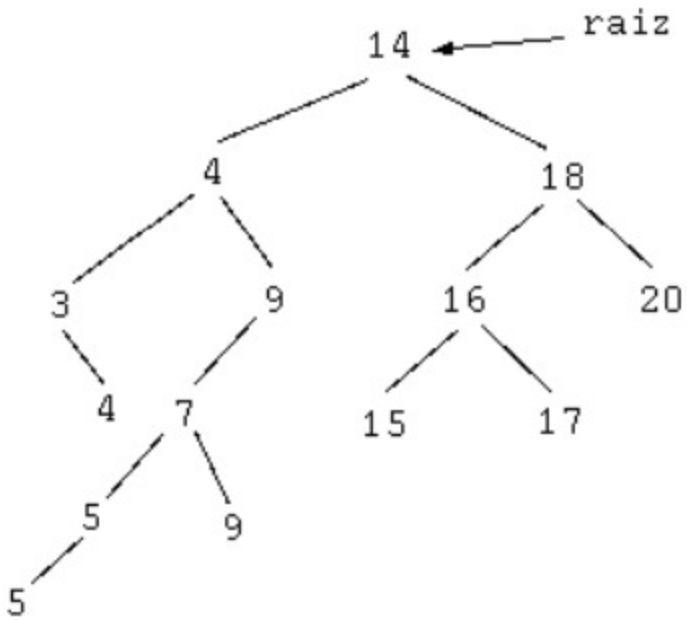
```
int qntdeNohs (pNohArvore raiz)
{
   if (raiz == NULL)
      return 0; //nenhuma folha a direita e esquerda

   return 1 + qtdeFolhas (raiz→esquerda) + qtdeFolhas (raiz→esquerda);
}
```

EXCLUIR NÓ DA ÁRVORE

paiFolhaMaisAEsquerda

recebe uma raiz e me devolve outra raiz, mas o nó que ela devolve é o pai da folha mais esquerda



Se a **raiz→esquerda ≠ NULL** entra em **raiz→esquerda→esquerda** que vai chegar no 18 e ver se ele é o pai da folha mais a esquerda, contudo ele será o pai da folha mais a esquerda apenas se a esquerda da sua raiz a esquerda é null, se a raiz→esquerda→esquerda tiver alguém ele será o avô.

No caso base:

- antes de liberar a memória do nó, salvamos a sua esquerda e direita
- Então eu exclui o 14, mas antes salvei seus filhos nos auxiliares

No caso 1:

- Se a esquerda e a direita forem nulos, não terá nada pra fazer.

No caso 2:

- Se a esquerda tiver algo (≠ NULL) retorna a esquerda, se ela for NULL retorna a direita

No caso 3:

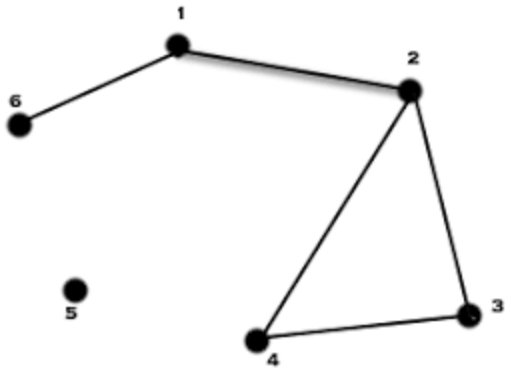
- Declara o pai da folha mais esquerda e o localiza (16), chama o paiFolhaMaisAEsquerda passando o aux direita pois quer o número maior mais próximo
- Se o paiFolhaEsquerda ≠ NULL, o auxFolhaEsquerda vai receber o valor a esquerda do pai da folha mais esquerda, nesse caso o pai da folha mais esquerda é o 16 então o auxFolhaEsquerda vai ser o número 15
- o paiFolhaEsquerda (16) vai receber em sua esquerda a direita do auxFolhaEsquerda (15)

- a esquerda do auxFolhaEsquerda (15) vai receber em sua esquerda o auxEsquerda (4)
- a direita do auxFolhaEsquerda (15) vai receber o auxDireita (18)
- retorna o auxFolhaEsquerda (15)
- (se ele for nulo o próprio auxDireita vai entrar como pai do auxEsquerda **auxDireita→esquerda = auxEsquerda)**

▼ Grafos

Grafo é uma estrutura formada por dois tipos de objetos: vértices e arestas

- cada aresta é um par não ordenado de vértices, ou seja, um conjunto com exatamente dois vértices
- uma aresta como (v, w), será denotada simplesmente por vw ou wv
- dizemos que essa aresta incide em v e em w e que eles são as pontas da aresta e vizinhos (ou adjacentes)

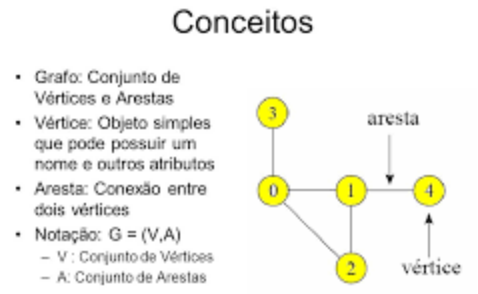


Grafo: conjunto de vértices e arestas

Vértices: objeto simples que pode ter nome e outros atributos

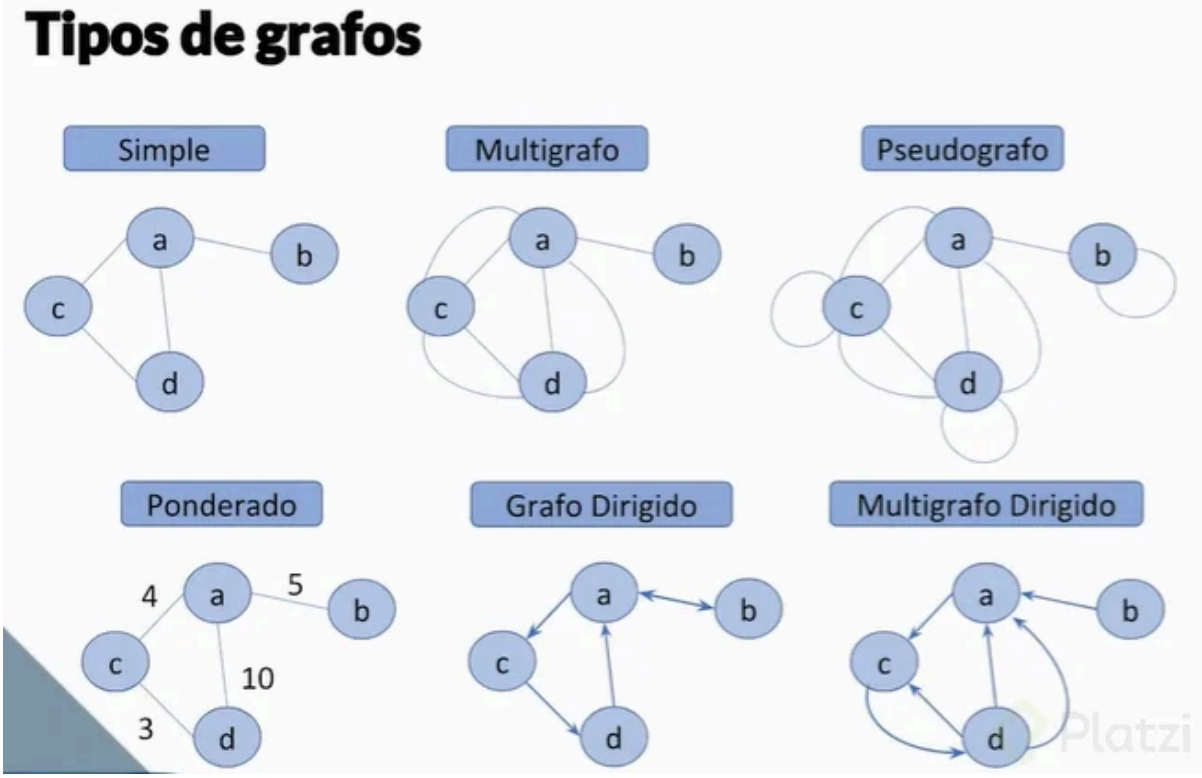
Aresta: conexão entre dois vértices

G = (V, A) → formado por um conjunto de vértices e um conjunto de arestas



A = { (3,0), (1,0), (1,4), (2,1), (0,2) }

V = { 3,1,0,2,4 }



Exemplo de implementação de grafo

REPRESENTAÇÃO COMPUTACIONAL DE GRAFO

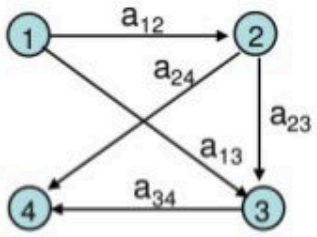
MATRIZ DE ADJACÊNCIA

Representação de grafos

Formas de representação e matrizes associadas a um grafo:

- **Matriz de adjacência**:
  Uma linha para cada vértice
  Uma coluna para cada vértice

$$a_{ij} = 1 \rightarrow (i, j) \in A$$
$$a_{ij} = 0 \rightarrow (i, j) \notin A$$

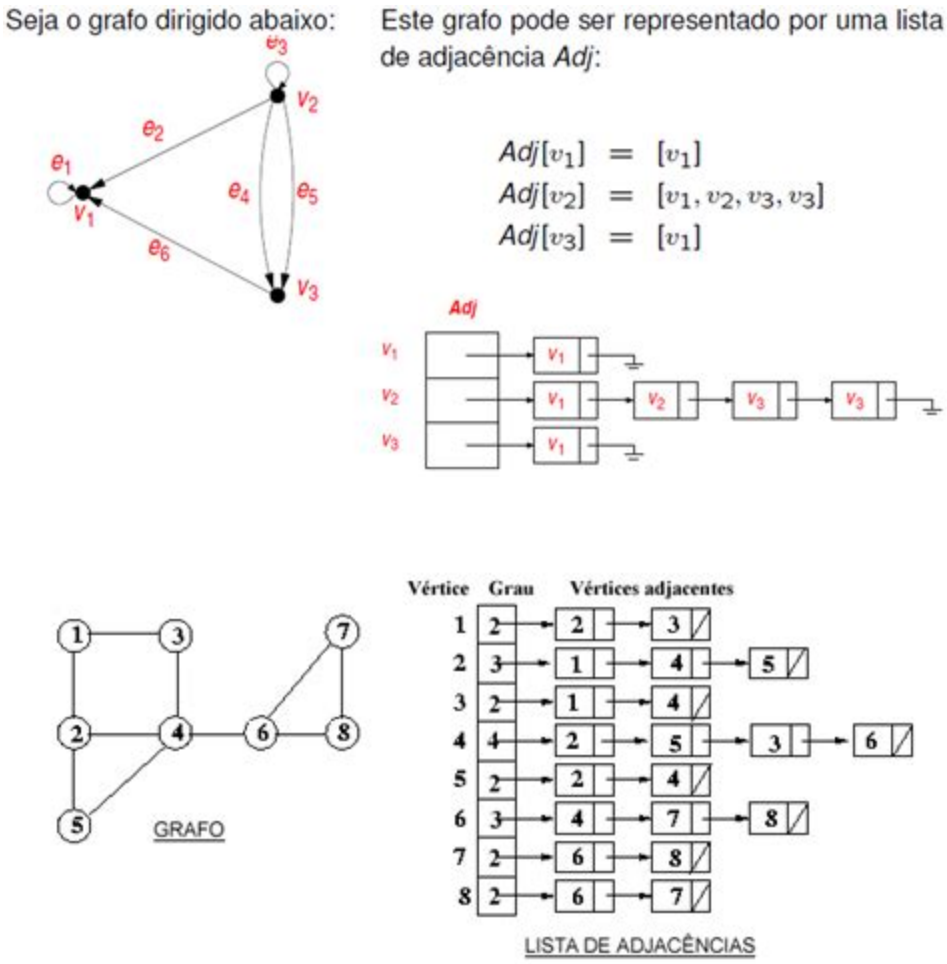


$$A_{n \times n} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Laboratório de Programação II – Grafos

- do 1 eu consigo chegar no 2? se sim: 1, se não: 0
- a12 = 1
- Para adicionar mais um nó, é necessário ampliar a matriz, adicionando mais um linha e coluna
- O acesso é mais rápido, contudo ela não é tão dinâmica e manipulável para alteração de nós

LISTA DE ADJACÊNCIA



- teremos uma lista principal com todos os vértices (coluna vértice)
- se tenho um grafo com 10 vértices, tenho uma lista de 10 nós
- tenho o grau do 1 vértice (2, pois faz 2 ligações)
- após isso tenho os vértices adjacentes, como ele é de grau 2, terá 2 vértices adjacentes (o 2 e depois o 3, ou vice versa)

NO CÓDIGO

```
struct dGrafo {
   pDLista listaVertices;  //grafo é um conjunto de vértices
   FuncaoComparacao fc;
   FuncaoImpressao fi;
   FuncaoAlocacao fa;
}

struct vertice {
   void *id;
   int grau; //o grau pode ser obtido do descritor
   pDLista listaAdjacenciais;
}
```

- possui um struct vértice (tipo o nó), e um pDGrafo, seu descritor
- tem função comparação pois vértices são conjuntos e não podem ter infos repetidas

---

vértices visitados: 4, 5, 2, 1, 3, 7, 6, 8,

→ no 3 voltou no 1 e viu para onde poderia ir

vértices pendentes (pilha): 8, 6, 3, 2

→ aí desempilha e checa se visitou todos que estavam pendentes

---

- grafo é uma lista de vértices
- cada vértice tem uma info, um grau e uma lista de adjacências (vértices adjacentes aquele vértice)
- ordem dos vértices não importa

12/06 grafos

15/06 grafos

19/06 revisão e exercícios

22/06 2 prova (árvores e grafos)

26/06 revisão

29/06 recuperação (prova)

03 a 06/07 fechamento da disciplina

1 - Excluir vértice

2 - Excluir aresta

// apaga o ponteiro nas listas de adjacências e muda os graus

3 - Existe caminho Hamiltoniano

4 - Existe caminho Euleriano

5 - Contêm subgrafo