Estrutura de Dados

Hamilton José Brumatto

Bacharelado em Ciências da Computação - UESC

5 de fevereiro de 2024

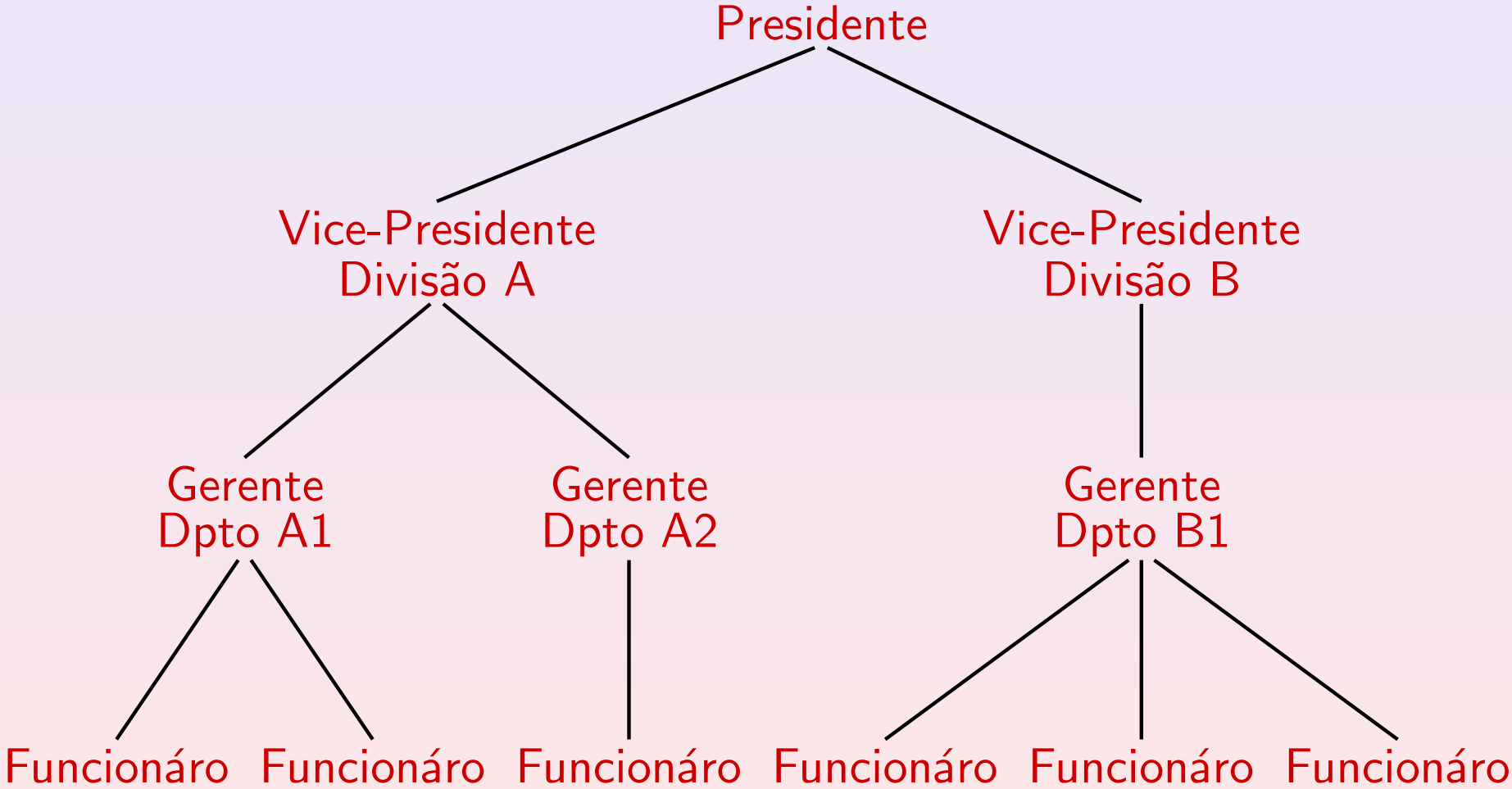
Árvores enraizadas: Árvores Binárias

Conceitos gerais

- O conceito de árvore é mais amplo, ele vem da definição de grafos.
- Uma árvore é um grafo acíclico.
- Vamos, no entanto, nos restringir a um tipo específico de árvores: árvores enraizadas.
- Neste conceito, um dos nós é especial, é chamado de raiz, e os demais nós são descendentes deste.
- Apesar de árvores enraizadas serem contempladas no contexto de grafos orientados, não trabalharemos com a visão de grafos neste momento.

Árvores enraizadas - Conceitos básicos

- Estrutura complexa que representa uma hierarquia.



Representação Matemática

- Uma árvore T é um conjunto finito, não vazio de nós.
- $T = \{r\} \cup T_1 \cup T_2 \cup \ldots \cup T_n$, com as propriedades:
  - Um nó especial da árvore, r, é chamado de raiz da árvore; e
  - O restante dos nós é particionado em $n \geqslant 0$ subconjuntos, T_1, T_2, ..., T_n, cada um dos quais sendo uma árvore.

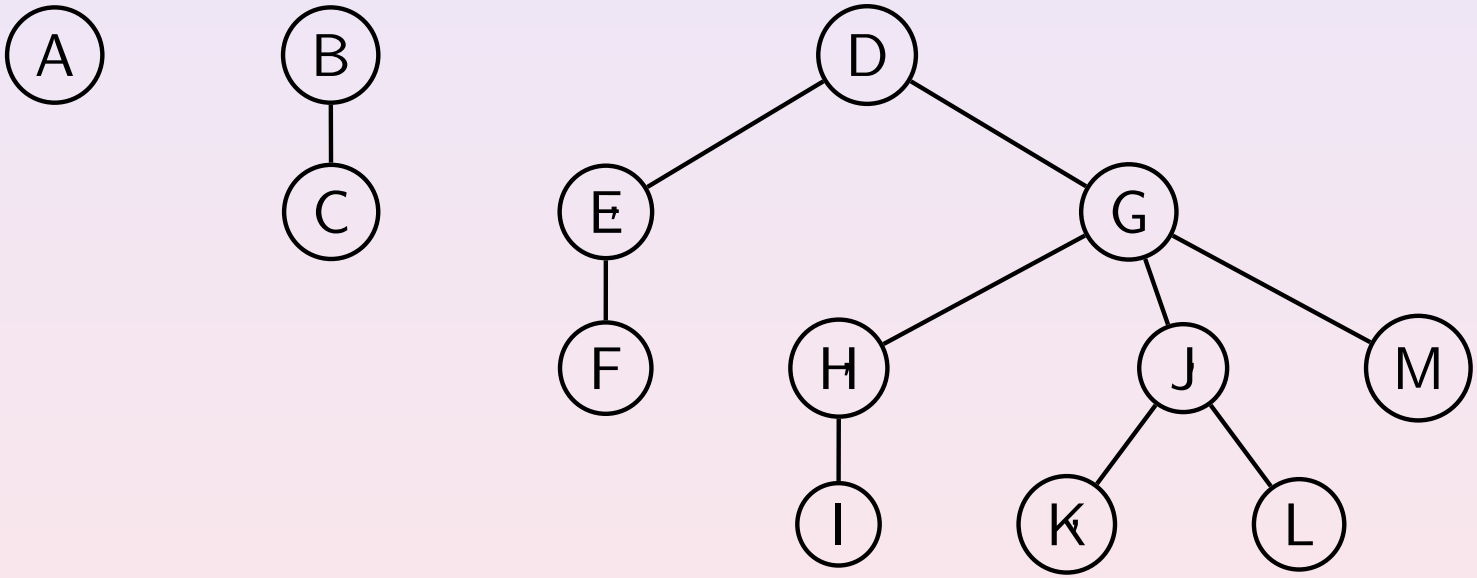
Resumindo, representa-se a árvore como:

$$T = \{r, T_1, T_2, \ldots, T_n\}$$

Representação Matemática

- Exemplos:
  - $T = \{A\}$
  - $T = \{B, \{C\}\}$
  - $T = \{D, \{E, \{F\}\}, \{G, \{H, \{I\}\}, \{J, \{K\}, \{L\}\}, \{M\}\}\}$

Representação Gráfica



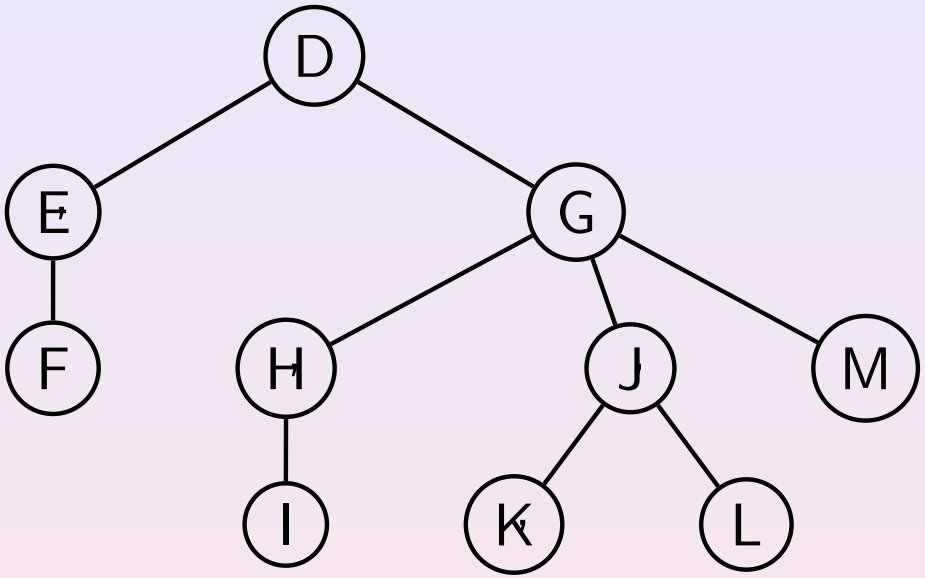
Terminologia

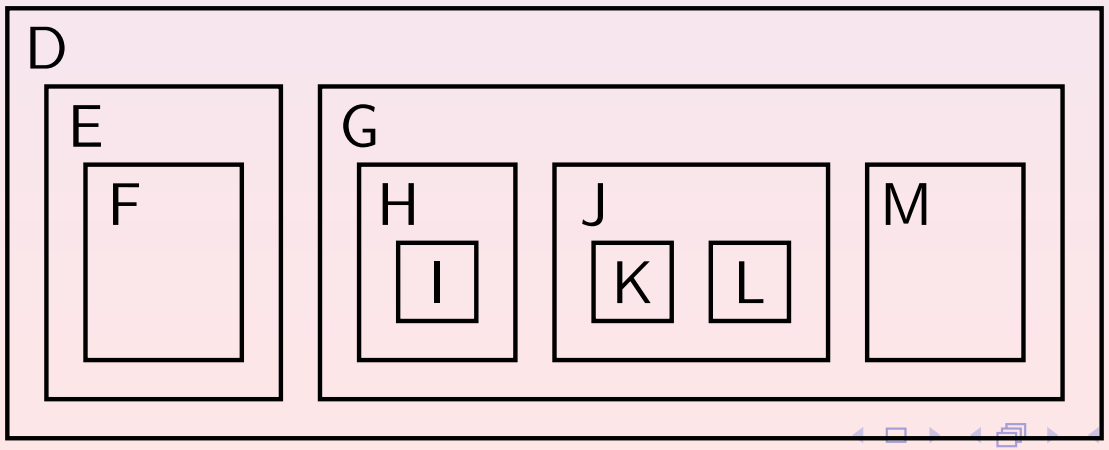
- Nó - um elemento da árvore, pode ser a raiz (no topo), uma folha (terminador) ou um ramo (intermediário)
- Grau - (de um nó) é o número de subárvores relacionadas com aquele nó (esta terminologia é diferente se a árvore não for enraizada)
- Folha - Um nó de grau 0.
- Filho/Neto - Nó raiz de uma subárvore, com relação à árvore que pertence.
- Raiz - Não é filho, origem das árvores.
- Irmãs - Raízes distintas de subárvores de uma mesma árvore
- Caminho - Sequência não vazia de nós.
- Comprimento - Quantos nós foram passados (com excessão do primeiro) em uma sequência.

Terminologia

- Altura - É o comprimento do caminho mais longo do nó até uma folha descendente do nó.
- Profundidade - (de um nó) Comprimento do raiz da árvore ao nó.
- Altura da árvore - É a altura do raiz.
- Ancestral - (de um nó) Um nó de menor profundidade, em relação a este, desde que esteja no caminho do comprimento da profundidade.
- Descendente - Um nó de uma profundidade maior, em relação a este. Sendo este parte do caminho para o comprimento da profundidade.

Representações



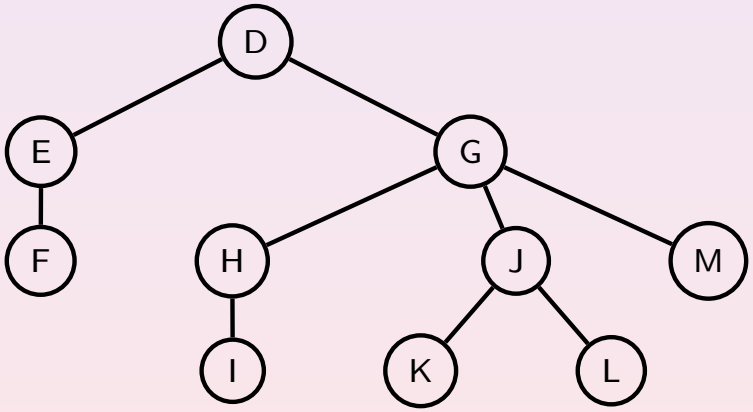


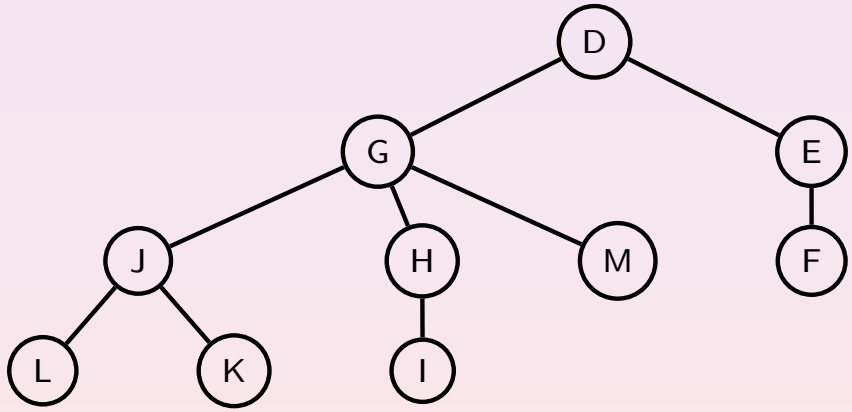
Representações

```
class D {
        class E {
                class F {
                }
        }
        class G {
                class H {
                        class I {
                        }
                }
                class J {
                        class K {
                        }
                        class L {
                        }
                class M {
                }
        }
}
```

Representações

- Não existe ordem entre os filhos. Algumas árvores possuem, como a árvore $n-$ária, por conta das árvores vazias.
- As duas árvores a seguir são isomorfas:





Implementação de uma árvore

- São várias formas possíveis, estática, dinâmica, ...
- Normalmente se implementa de forma dinâmica para permitir inserir ou remover ramos.
- Assim cada ramo passa a ser um objeto/posição de memória alocado dinamicamente
- Os nós apontam para filhos (e para o pai) usando ponteiros da linguagem.
- Como todos os nós são árvores, então as operações realizadas nas árvores ocorrem de forma recursiva:
  - Um valor de um nó pode depender do resultado de seus descententes
  - É feita uma chamada recursiva (até que não exista descendentes / folha)
  - O nó processa o resultado da chamada a seus filhos e retorna seu valor.

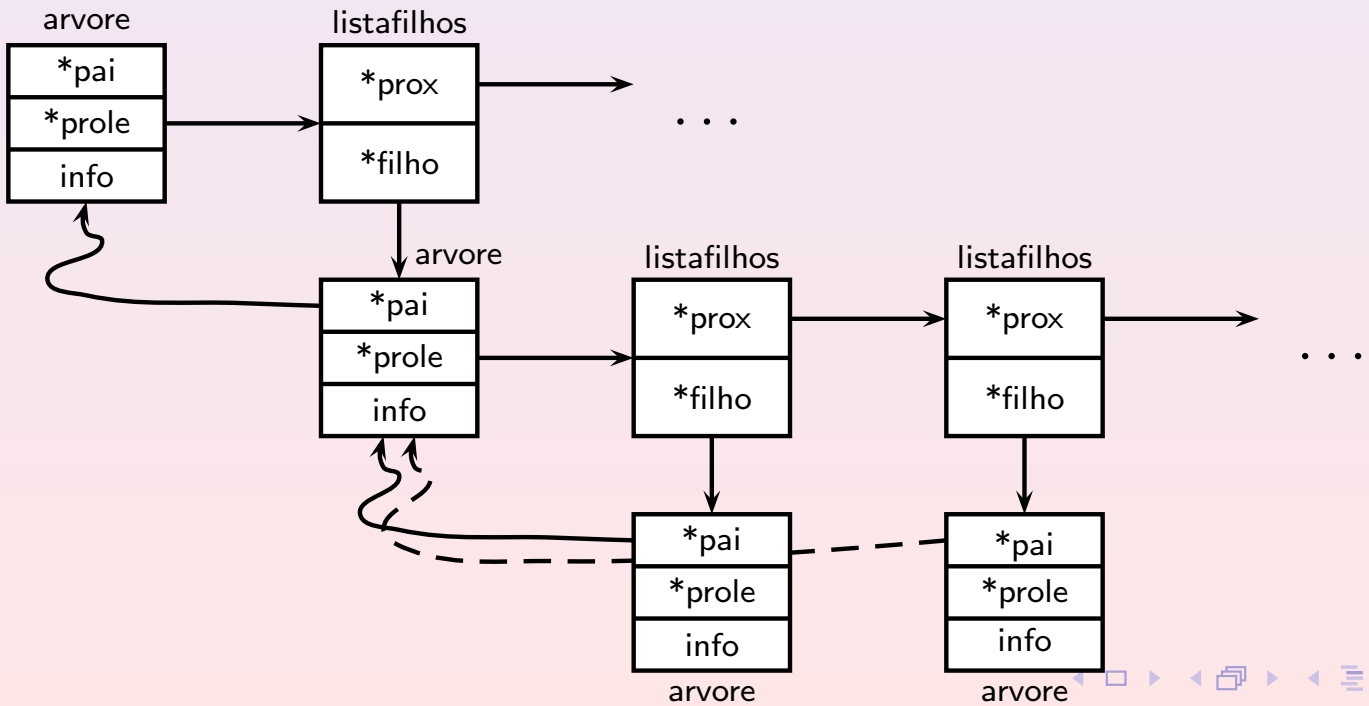
Implementação de uma árvore

- Na implementação algumas funções que são importantes:
  - void iniciar(arvore *a) → inicia a estrutura interna da árvore, atribuindo info.
  - obj_t info(arvore *a) → retorna o conteúdo de informação.
  - int altura(arvore *a) → retorna a altura da árvore.
  - int profundidade(arvore *a) → retorna a profundidade.
  - arvore *pai(arvore *a) → retorna quem é o pai da árvore.
  - int filhos(arvore *a) → retorna quantos filhos tem a árvore.
  - arvore *filho(arvore *a, int i) → retorna um filho, por índice.
  - int insereFilho(arvore *a, arvore *f) → insere o filho e retorna o índice.
  - void removeFilho(arvore *a, int i) → remove o filho na posição i.
- A implementação está no código que faz parte desta aula.

Implementação de uma árvore - Estrutura Dinâmica

```
typedef struct arvore{
  obj_t info;
  listafilhos *prole;
  struct arvore *pai;
} arvore;
```

```
typedef struct listafilhos{
  struct arvore *filho;
  struct listafilhos *prox;
} listafilhos;
```



Implementação de uma árvore - Códigos

```
void iniciar(arvore *a, obj_t info) {
  a->info = info;
  a->pai = NULL;
  a->prole = NULL;
}

obj_t info(arvore *a) {
  obj_t ret = '\0';
  if(a! =NULL)
    ret = a->info;
  return ret;
}
```

Implementação de uma árvore - Códigos

```
int altura(arvore *a) {
  int nfilhos;
  int alt=0, h, i;

  nfilhos = filhos(a);
  for(i=1; i <= nfilhos; i++) {
    h = altura(filho(a,i));
    if(alt <= h) alt = h+1;
  }
  return alt;
}

int profundidade(arvore *a) {
  int p = 0;

  if(a->pai != NULL) p = profundidade(a->pai) + 1;
  return p;
}
```

Implementação de uma árvore - Códigos

```
int filhos(arvore *a) {
  int k = 0;
  listafilhos *f = a->prole;
  while(f! =NULL) {
    f=f->prox;
    k++;
  }
  return k;
}

arvore *filho(arvore *a, int i) {
  int k = 1;
  arvore *filho = NULL;
  listafilhos *f = a->prole;
  while(k < i && f ! = NULL) {
    f = f->prox;
    k++;
  }
  if(k==i && f ! = NULL) filho = f->filho;
  return filho;
```

Implementação de uma árvore - Códigos

```
int insereFilho(arvore *a, arvore *f) {
  int k = 1;
  listafilhos *fi = a->prole;

  if(fi == NULL) {
    a->prole = (listafilhos *) malloc(sizeof(listafilhos));
    fi = a->prole;
  } else{
    while(fi->prox != NULL) {
      fi = fi->prox;
      k++;
    }
    fi->prox = (listafilhos *) malloc(sizeof(listafilhos));
    fi = fi->prox;
    k++;
  }
  f->pai = a;
  fi->filho = f;
  return k;
}
```

Implementação de uma árvore - Códigos

```
void removeFilho(arvore *a, int i) {
  int k, nfilhos;    arvore *f = NULL;    listafilhos *ant, *fi = a- >prole;
  if(i == 1) {
    if(fi ! = NULL) {
      a- >prole = fi- >prox;
      f = fi- >filho;
      free(fi);
    }
  } else {
    k = 1;
    while(fi ! = NULL && k < (i-1)) fi = fi- >prox;
    if(fi ! = NULL) {
      ant = fi;
      fi = fi- >prox;
      if(fi ! = NULL) {
        ant- >prox = fi- >prox;
        f = fi- >filho;
        free(fi);
      }
      else ant- >prox = NULL;
    }
  }
  if(f ! = NULL) {
    nfilhos = filhos(f);
    if(nfilhos > 0) for(k = 1; k <= nfilhos; k++) removeFilho(f,1);
    free(f);
  }
  return;
}
```

Percurso em árvores

- A árvore é uma estrutura de dados para guardar informações de maneira hierárquica.
- Mas uma vez guardada as informações, podemos querer tratar a árvore de diversas formas:
  - Listar tudo que está guardado.
  - Identificar se uma determinada informação está guardada.
  - Traçar uma ordem parcial com as informações guardadas.
- Para isto, é necessário "visitar" as informações guardadas nos nós da árvore.
- A "visita" aos nós se dá através de uma "busca em percurso" na árvore.
- São dois tipos básicos (herdados dos grafos): percurso em largura ou em profundidade.

Percurso em árvores

- Percurso em largura:
  - Começa pelo raiz e visita cada um dos filhos do nó.
  - Após isto, visita cada filho de cada filho...
- Percurso em profundidade:
  - Começa pelo raiz e recursivamente visita o filho.
  - Não havendo mais filhos a visitar de um nó, começa o percurso no nó irmão deste...
- Ao realizar a visita temos duas opções: Tratar a info do nó antes de visitar os filhos, ou após visitar os filhos: pré-percurso e pós-percurso.

Percurso em Profundidade

- Este é o mais simples.
  - Em um pré-percurso, trata primeiro a informação do nó.
- Para cada um dos filhos, chama o percurso em profundidade recursivamente.
  - Em um pós-percurso, trata por último a informação do nó.
- Fim do percurso.
- Normalmente este percurso é chamado de DFS (Depth First Search), busca primeiro em profundidade.

Percurso em Profundidade - Código

```
void dfs(arvore *a, void function(obj_t), boolean pre) {
  if(pre) function(a->info);
  listafilhos *fi = a->prole;
  while(fi != NULL) {
    dfs(fi->filho,function,pre);
    fi = fi->prox;
  }
  if(!pre) function(a->info);
  return;
}
```

Percurso em Largura

- Este traz uma complicação maior, pois vamos visitar os irmãos de um nó antes de visitar seus filhos.
- É necessário guardarmos estes filhos para uma visita futura.
- Existe uma estrutura que nos ajuda nisto: Fila.
- Os nós que precisamos visitar são colocados numa fila.
- Cada vez que visitamos um nó, colocamos seus filhos na fila.
- Assim, garantimos que os filhos do nó estão na fila de visita antes de visitarmos os irmãos que já estão na fila, também.
- Como não há recursão neste tipo de percurso, então não faz sentido em falar em pré percurso e pós percurso, a visita é feita na ordem de largura e pronto.
- Normalmente este percurso é chamado de BFS (Breadth First Search), busca primeiro em largura.

Percurso em Largura - Código

```
void bfs(arvore *a, void function(obj_t)) {
  fila f; iniciarFila(&f);
  enfileirar(&f,a);
  while(!ehvazia(&f)) {
    arvore *v = frente(&f); desenfileirar(&f);
    listafilhos *fi = v->prole;
    while(fi != NULL) {
      enfileirar(&f,fi->filho);
      fi = fi->prox;
    }
    function(v->info);
  }
  return;
}
```

Árvores N-árias

- Ainda é possível restringir o escopo de árvores.
- Podemos restringir quanto à quantidade de filhos que cada nó pode ter.
- Por exemplo, vamos especificar que a árvore possui exatamente N filhos.
- Este caso parece ser recursivo ao infinito, portanto precisamos de um limite.
- Definimos uma árvore vazia, como uma árvore válida.
- A árvore vazia não possui valor no raiz, nem filhos, é apenas uma árvore que irá auxiliar na quantidade de filhos possíveis.

Árvores N-árias

- Uma árvore N-ária T é definida como:
  - O conjunto (árvore) é vazio, $T = \emptyset$ ou
  - O conjunto consiste de uma raíz R, e exatamente N árvores N-árias distintas:

$$T = \{R, T_1, T_2, \ldots, T_N\}$$

- Exemplo de árvores 3-árias (ternárias):

$$T_a = \{A, \emptyset, \emptyset, \emptyset\},$$

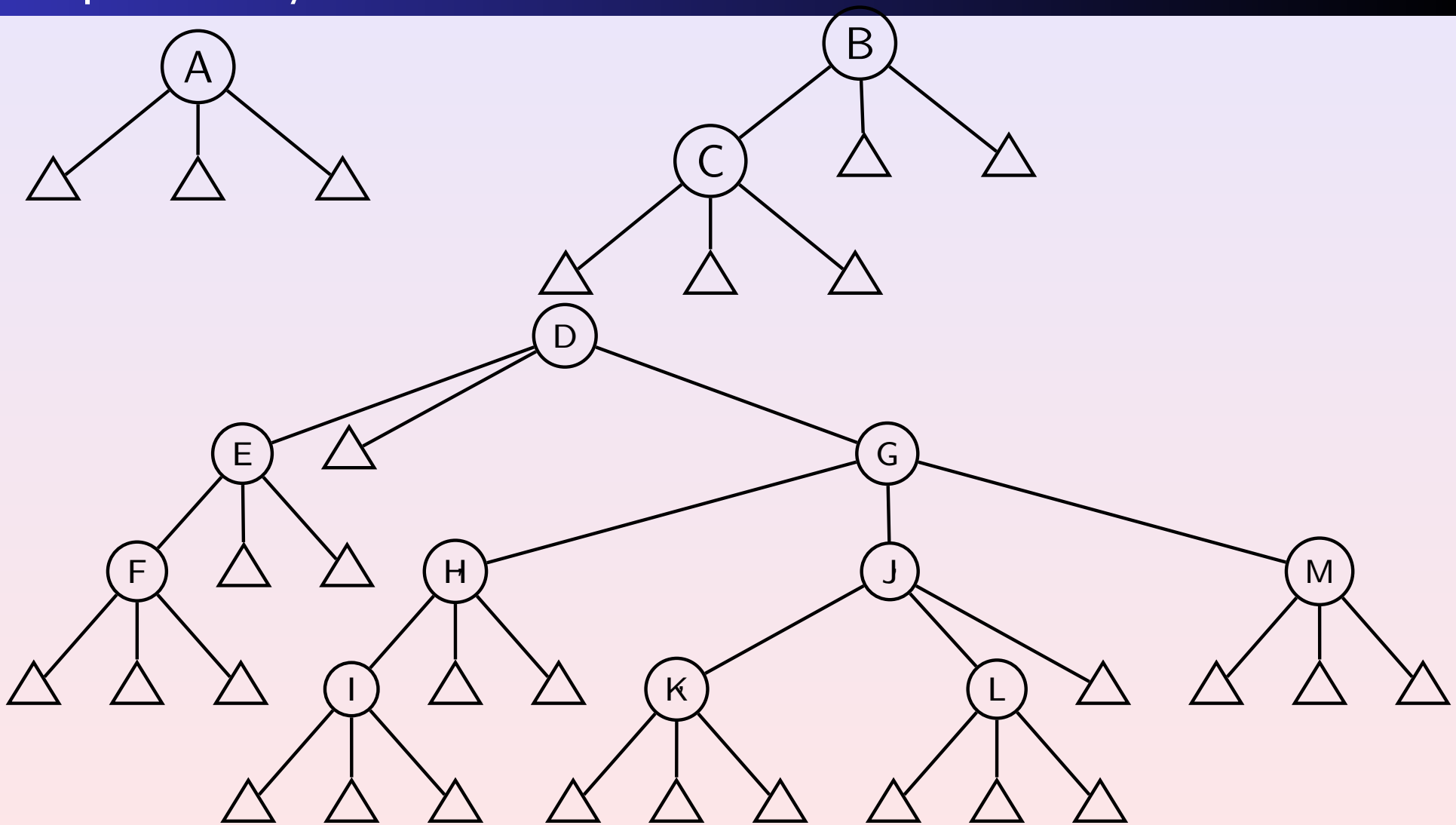
$$T_b = \{B, \{C, \emptyset, \emptyset, \emptyset\}, \emptyset, \emptyset\}$$

$$T_d = \{D, \{E, \{F, \emptyset, \emptyset, \emptyset\}, \emptyset, \emptyset\}, \{G, \{H, \{I, \emptyset, \emptyset, \emptyset\}, \emptyset, \emptyset\}, \{J, \{K, \emptyset, \emptyset, \emptyset\}, \{L, \emptyset, \emptyset, \emptyset\}, \emptyset\}, \{M, \emptyset, \emptyset, \emptyset\}\}, \emptyset\}$$

Árvores N-árias

- As árvores vazias são chamadas de nós externos
- As árvores não vazias são chamadas de nós internos.
- Folhas são nós internos que somente possuem subárvores que são nós externos.
- Uma árvore N-ária com $n \geqslant 0$ nós internos possui $(N-1)n+1$ nós externos.
- A altura de um nó externo é -1.
- A altura de uma folha é 0.

Representação Gráfica



Implementações de árvore *N*-ária

- Utiliza as mesmas rotinas implementadas na árvore geral.
- A implementação é mais simples, usa somente uma estrutura.
- A lista de filhos é então um vetor de ponteiros para novas árvores, dentro da própria estrutura da árvore.
- As árvores vazias são representadas como NULL na lista de filhos.
- A ordem importa, filhos $\{T_1, T_2, \emptyset\} \neq \{T_1, \emptyset, T_2\}$.
- Cada árvore filha tem sua posição fixada por um índice do vetor.
- As implementações estão em códigos que acompanham esta aula.

Implementações de árvore *N*-ária

- A grande diferença da implementação em relação à árvore geral é o tratamento dos filhos.
- Ao invés de navegar via ponteiros, temos um vetor onde percorremos iterativamente.
- Um cuidado especial é ao inserir um novo filho, é preciso agora dizer qual a posição.
- O que fazer se for inserir um filho em um lugar que já existe uma árvore:
  - Retornar "falso" obrigando explicitamente a remover o que lá existia pois não pode inserir algo em um lugar já ocupado, ou
  - Considerar que é mandatório a operação e remove o filho que lá existia e insere um novo.
  - Muitas vezes a solução depende do problema proposto
- Vamos mostrar a implementação de três funções apenas: *removeFilho*, e os dois percursos *bfs* e *dfs*, o resto está no material de aula.

Implementações de árvore *N*-ária: removeFilho

```
void removeFilho(arvore *a, uint i) {
  int k;
  arvore *f = NULL;
  assert(i < NARIA);

  if(a!=NULL) {
    f = a->filhos[i];
    a->filhos[i] = NULL;
    if(f != NULL)       for(k = 0; k < NARIA; k++)
        removeFilho(f,k);
    free(f);
  }
  return;
}
```

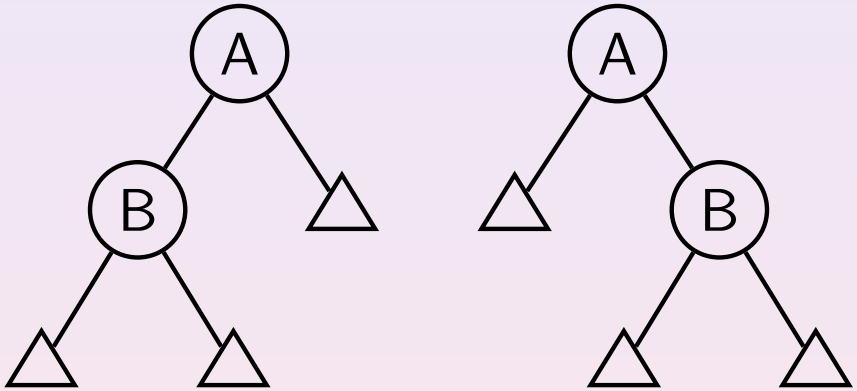
Implementações de árvore *N*-ária: buscas

```
void bfs(arvore *a, void function(obj_t)) {
  fila f; iniciarFila(&f);

  enfileirar(&f,a);
  while(!ehvazia(&f)) {
    arvore *v = frente(&f); desenfileirar(&f);
    for(int i = 0; i < NARIA; i++)
      if(v->filhos[i] != NULL) enfileirar(&f,v->filhos[i]);
    function(v->info);
  }
  return;
}

void dfs(arvore *a, void function(obj_t), boolean pre) {
  if(pre) function(a->info);
  for(int i = 0; i < NARIA; i++)
    if(a->filhos[i] != NULL) dfs(a->filhos[i],function,pre);
  if(!pre) function(a->info);
  return;
}
```

Árvores Binárias

- O conjunto é vazio, $T = \emptyset$; ou
- O conjunto consiste em uma raiz, R, e em exatamente duas árvores binárias distintas T_e e T_d, $T = \{R, T_e, T_d\}$
- A árvore T_e é dita subárvore esquerda de T, e a árvore T_d é dita a subárvore direita de T.

Exemplo de Árvores Binárias

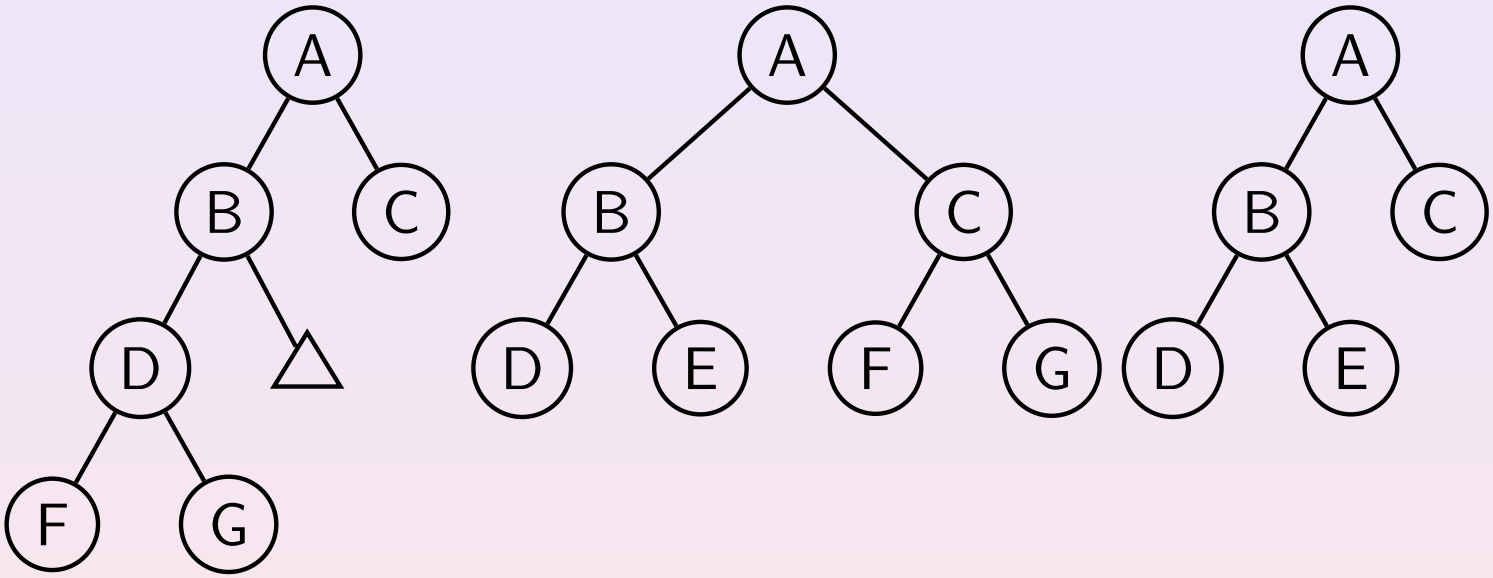


Árvores binárias distintas, a primeira possui a subárvore esquerda, como nó interno, e a segunda a subárvore direita: $\{A, \{B, \emptyset, \emptyset\}, \emptyset\}$ e $\{A, \emptyset, \{B, \emptyset, \emptyset\}\}$

Definições sobre árvores binárias

- Uma árvore binária completa de profundidade d é a árvore estritamente binária onde todas as folhas estejam no nível d.
- Uma árvore binária completa de profundidade d possui 2^d folhas e $2^d - 1$ nós não folhas.
- Uma árvore binária quase completa de profundidade d ocorre quando:
  - Cada folha na árvore está no nível d ou $d - 1$.
  - Para cada nó n na árvore que contenha um descendente direito folha no nível d, então todos descendentes esquerdos que forem folha também estão no nível d.
- Uma árvore (completa ou quase completa) estritamente binária com n folhas possui no total $2n - 1$ nós.
- Uma árvore quase completa que não seja estritamente binária com n folhas possui no total $2n$ nós
- Existe uma única árvore binária quase completa com n nós, esta árvore será estritamente binária se n for ímpar

Exemplos - omitindo-se quase todas árvores vazias



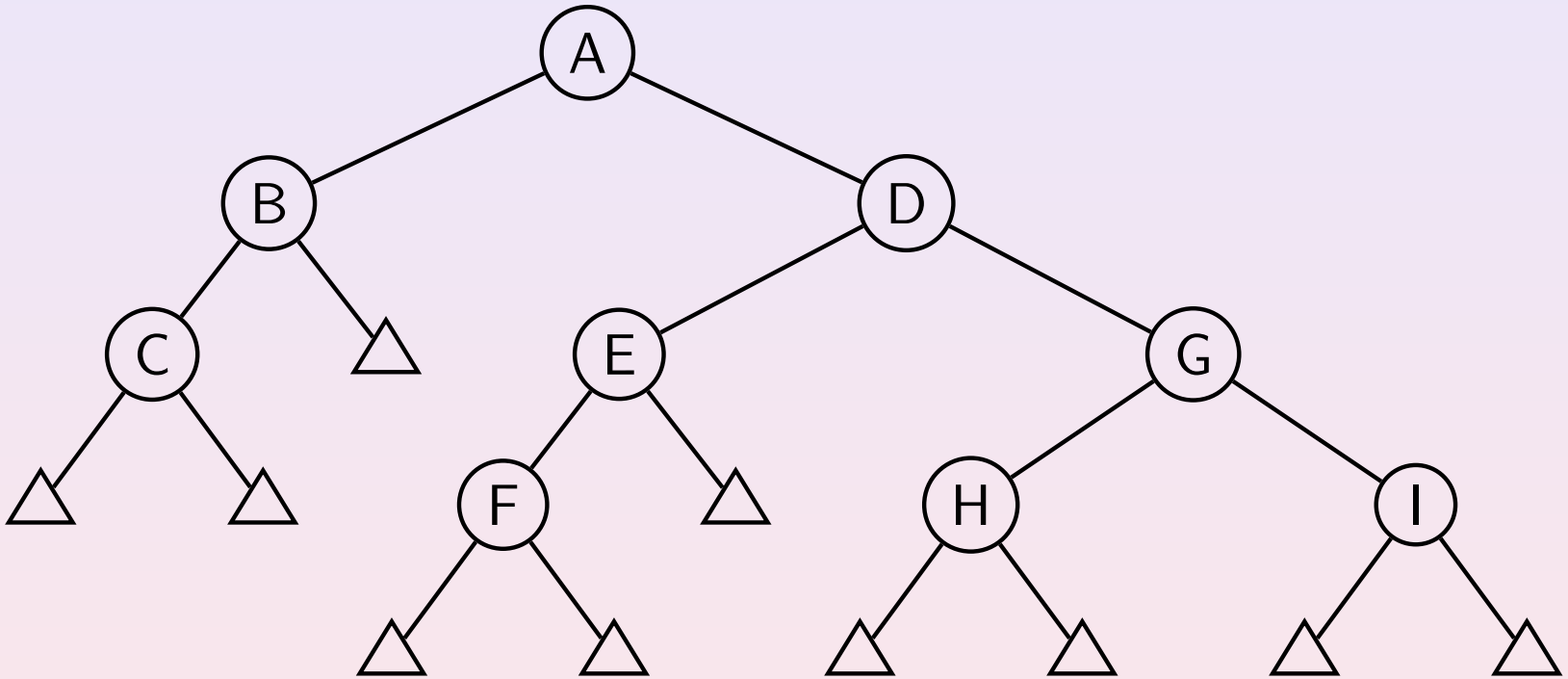
(a) (b) (c)

- (a) Árvore não estritamente binária
- (b) Árvore binária completa
- (c) Árvore estritamente binária quase completa

Percurso em Árvores Binárias

- O percurso, tal qual em árvores gerais pode ser em largura ou em profundidade.
- Nas árvores binárias os percursos em profundidade são de três tipos:
  - Percurso em pré-ordem: Faz-se uma pesquisa sobre o conteúdo do raiz, em seguida, recursivamente aplica-se o percurso no filho esquerdo e depois no direito.
  - Percurso em ordem simétrica (ou em ordem): Aplica-se recursivamente o percurso no filho esquerdo, faz-se a pesquisa no raiz, e aplica-se recursivamente o percurso no filho direito.
  - Percurso em pós-ordem: Aplica-se recursivamente primeiro o percurso no filho esquerdo, em seguida no filho direito e então faz-se a pesquisa no raiz.
- O percurso em ordem é o único restrito para árvores binárias, também aparece nas árvores m-múltiplas.

Exemplo de percurso em Árvores Binárias



Percurso em pré-ordem: ABCDEFGHI
Percurso em ordem: CBAFEDHGI
Percurso em pós-ordem: CBFEHIGDA

Implementações em Árvores Binárias

- As implementações em árvores binárias são semelhantes às implementações de *N*-árias.
- Substitui-se o vetor de filhos por dois filhos: FilhoEsquerdo e FilhoDireito.
- As operações para os filhos, substitui-se o índice por duas operações, uma para cada filho.
- Nesta implementação, inclui-se os percursos.
- Uma implementação especial da árvore binária não utiliza o ponteiro `pai`, já que basta, a partir do raiz, em algum percurso, recursivamente, chegar a todos os filhos.