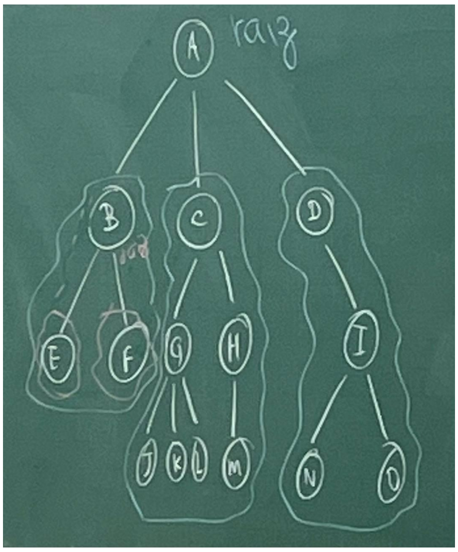
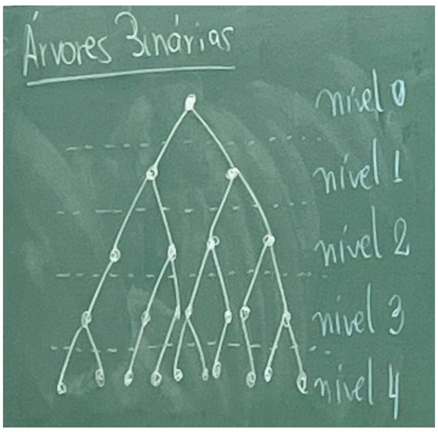
Árvores

- Uma árvore é um conjunto de elementos interligados entre si de tal forma que: um elemento é a **raiz** e os demais se dividem em $n \geq 0$ subconjuntos distintos(disjuntos) chamados **subárvores**.
    - Intersecção: Um nó ligando a outro nó
- Quando deixa de ser disjunto → **Gráfos**
- **Disjunto**: Os elementos não dependem, ou se relacionam com os outros
- **Conceitos**:
    - Os elementos são chamados de **nós**, e as linhas que os ligam, **arestas**.
    - O nó que não possui "*ascendente/pai*" é chamado de **raiz**.
    - O nó que não possui "*descendente/filho*" é denominado de **folha**.
    - O **grau** de um nó é a quantidade de subárvores que se originam dele.



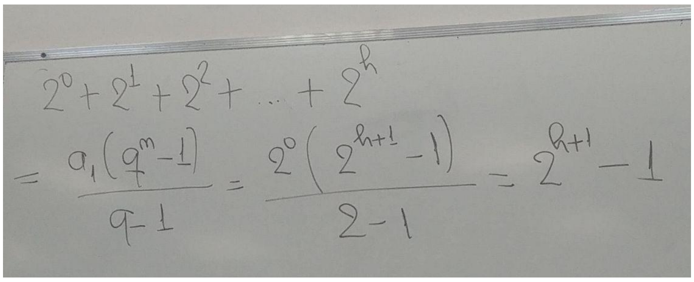
Grau(A) = 3, Grau(C) = 2, Grau(H) = 1, Grau(O) = vazio

    - Se uma árvore for tal que grau(v) <= 2, para qualquer v nó da árvore, então dizemos que essa é uma **árvore binária**.
- **Árvore Binária**:

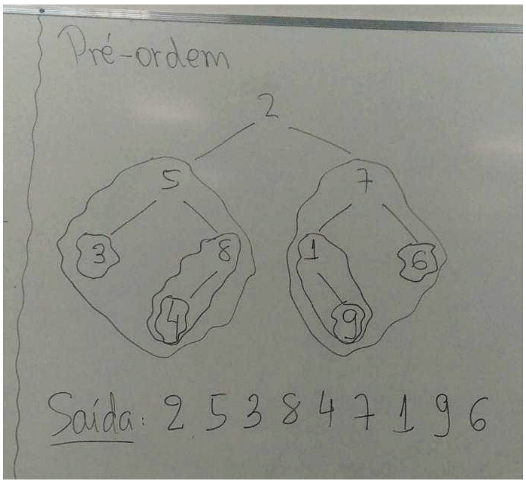


    - **Níveis** são "*gerações*" da árvore.
        - A altura de uma árvore é o seu maior nível. Ex: A altura é 4.

- Um nível k pode ter, no máximo, 2^k nós.
- O crescimento dos nós conforme aumenta-se o nível, exponencial
- O máximo de nós numa árvore de altura h é:

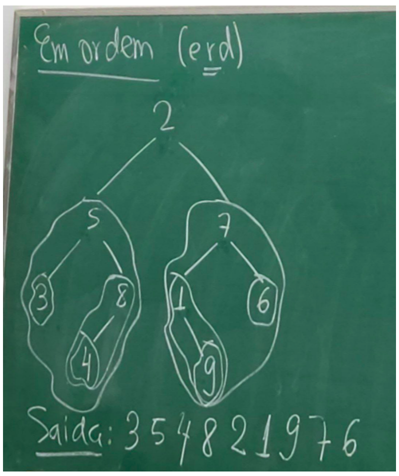


- Árvores são estruturas de dados **não-lineares** pois há várias formas distintas de se percorrer os elementos. As 3 formas padrão são:
  1. **Pré ordem**: Visita primeiro a raiz. Recursivamente visita a subárvore da esquerda e depois de finalizar visita a subárvore da direita



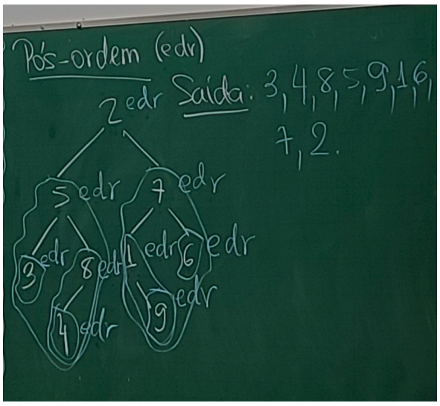
→ red

  2. **Em ordem**: Visita a subárvore da esquerda recursivamente, depois a raiz e depois a subárvore da direita.



→ eRd

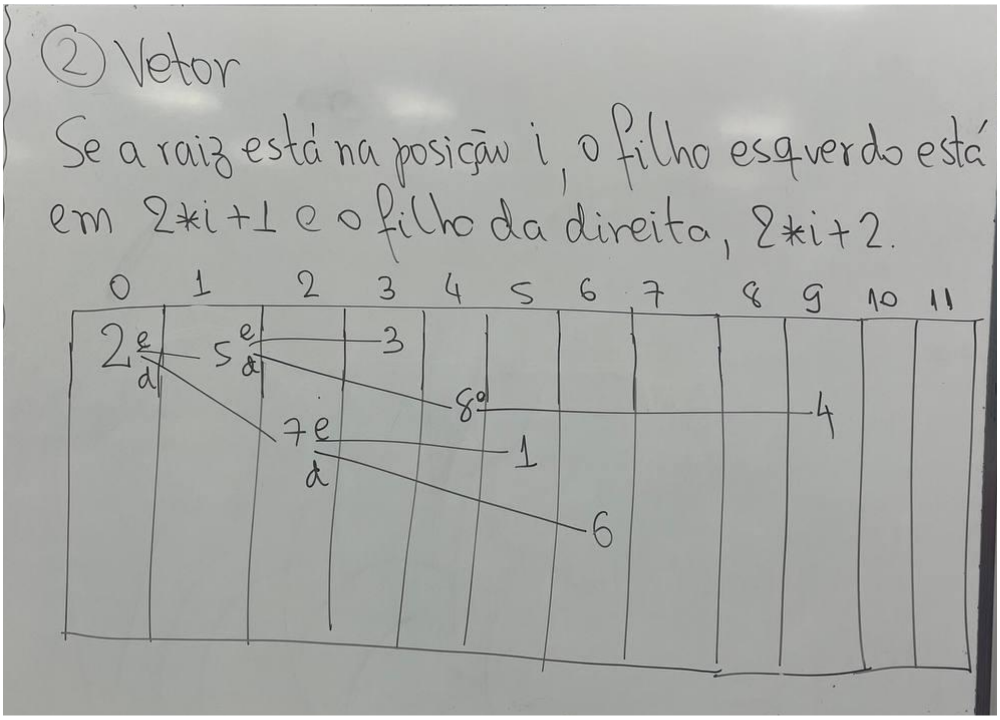
3. **Pós-ordem**: Primeiro a subárvore esquerda, depois a subárvore da direita e por final a raiz



→ edr

- **Visitar**: Qualquer ação que se faça: Escrever, ler, editar...

```
preOrdem(raiz)
      printf(raiz)
      preOrdem(raiz.esq)
      preOrdem(raiz.dir)
emOrdem(raiz)
      emOrdem(raiz.esq)
      printf(raiz)
      emOrdem(raiz.dir)
posOrdem(raiz)
      posOrdem(raiz.esq)
      posOrdem(raiz.dir)
      printf(raiz)
```



- **Representações de uma árvore binária**:

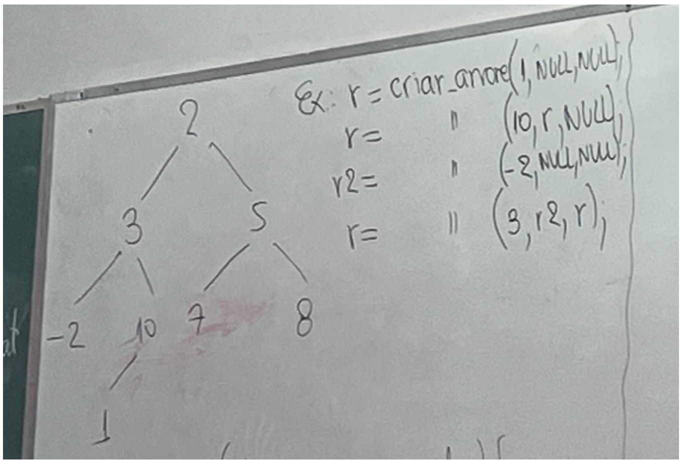
```
typedef struct no{
      Item dado;
      Struct no *esq, *dir;
}no;
```

  - **OBS**: Item é um tipo qualquer. Há duas formas de defini-lo:
    1. Typedef int Item;
    2. #define Item int
       - **Macro** → Tudo que começa com #

1. **Criação da árvore**: Vai criando de baixo pra cima

```
no *criar_arvore(Item x, no *esq, no *dir){
      no *raiz = malloc(sizeof(no));
      raiz → esq = esq;
      raiz → dir = dir, raiz → dado = x;
      return raiz;
}
```

ex:



2. **Buscar um elemento**: Complexidade : Máximo de elementos que se pode ter na árvore. Visando n elementos → **O(n)**. Em quantidade de níveis → Exponencial. **Busca sequencial**

```
no *busca(Item  x, no *raiz){
      if(raiz != NULL)
            if(raiz → dado == *) return raiz;
            else{
                  no *esq = busca(x, raiz → esq);
                  if(esq != NULL) return esq;
                  else return busca(x, raiz → dir);
            }
      }
      else return NULL;
}
```

Ex: Busca por 7 (Árvore da imagem da página 4)

busca(7, **2**)

↳ busca(7, **3**)

      ↳ busca(7, **-2**)

            ↳ busca(7, **NULL**)

            ↳ busca(7, **NULL**)

      ↳ busca(7, **10**)

            ↳ busca(7, **1**)

                  ↳ busca(7, **NULL**)

                  ↳ busca(7, **NULL**)

↳ busca(7, **5**)

      ↳ busca(7, **7**)

3. **Quantidade de nós**

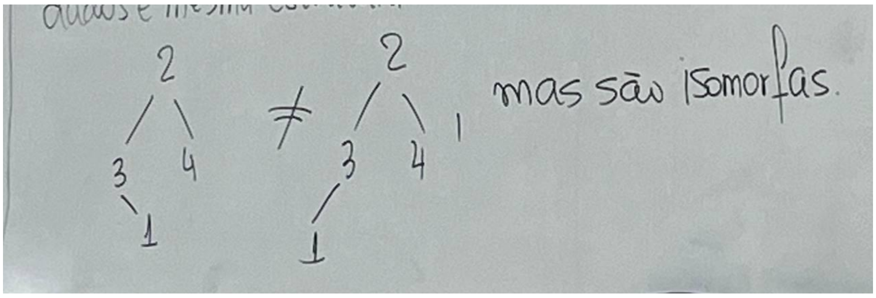
```
int qntd_nos( no *raiz){
      if(raiz == NULL) return 0;
      else{
            return 1 + qntd_nos(raiz → esq) +  qntd_nos(raiz → dir);
      }
}
```

4. **Altura da árvore**

```
int altura(no *raiz){
      if(raiz == NULL) return 0;
      else{
            int h_esq = altura(raiz → esq);
            int h_dir = altura(raiz → dir);
```

```
            return 1 + max(h_esq, h_dir);  // (h_esq > h_dir ? h_esq :
h_dir) → Operador ternário
      }
}
```
↳ Onde:
```
int max(int a, int b){
      if(a < b) return b;
      return a;
}
```

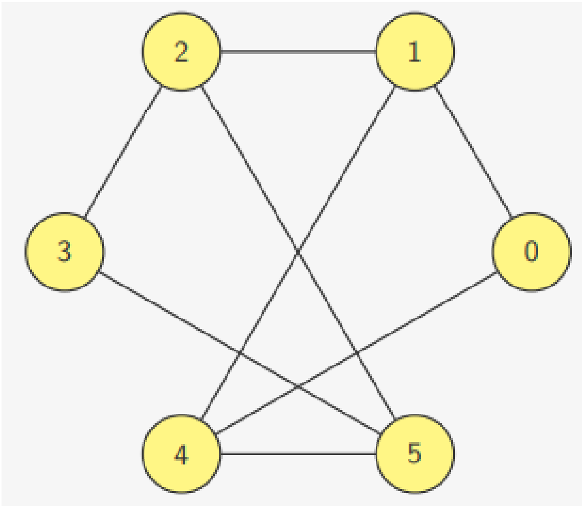
OBS: Duas árvores são iguais se possuíres mesmos dados e mesma estrutura
↳ Apenas um dado igual **porém local diferente, já a torna diferente**



- **Árvore binária de busca (ABB)**:
  - É uma árvore tal que nó r com subárvores esquerda Te direita Td satisfaz:
    - e < r, ∀e E Te //E → Pertence
    - d > r, ∀d E Td
  - Todo nó **a esquerda tem que ser menor que o valor da raiz** e todo nó à **direita da raiz tem que ser maior que a raiz**
  - Pior caso de busca binária → ela está ordenada de forma **crescente,** pois acabaríamos **jogando ou tudo pra direita ou tudo para esquerda custando O(n)** **(BOA QUESTÃO DE PROVA)** Nesse caso **seria melhor fazer a busca binária pelo vetor que custaria O(1)**
  - No caso como ela é ordenada de forma crescente ela joga tudo para a **direita**, se estivesse ordenada de forma decrescente jogaria tudo para a **esquerda**
  - **Mínimo** → Elemento mais à esquerda
  - **Máximo** → Elemento mais à direita
  - **Inserção na heap custa O(logn) em todos os casos**

Grafos

- Estrutura matemática em computação
- Um grafo é um conjunto de **objetos que têm interligação entre si**
- Chamamos esses objetos de **vértices**

    - Ex: Pessoas em uma rede social →ligamos duas pessoas que se conhecem
    - Pontos ou bolinhas
- Chamamos conexão entre os objetos de **arestas**
    - Ex: Relação de amizade na rede social
    - Linhas ou curvas
- Representamos um grafo visualmente
    - Com os **vértices** representados por pontos
    - Com as **arestas** representadas por curvas ligando dois **vértices**
- **Adjacência**
    - 

    - O vértice 0 é vizinho do vértice 4
        - Dizemos que vizinhos → adjacentes
        - Os vértices 0, 1 e 5 formam uma vizinhança do vértice 4
            - vizinhança → conjunto de adjacentes
- **TAD Grafo**

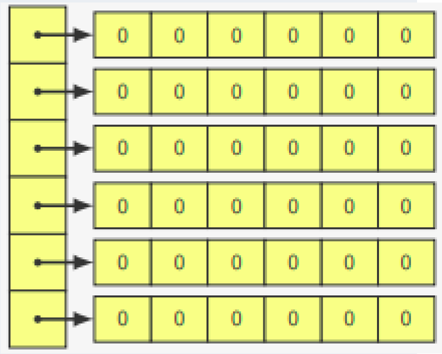
```c
typedef struct {
 int **adj;  int n;
} Grafo;
typedef Grafo * p_grafo;

p_grafo criar_grafo(int n);
void destroi_grafo(p_grafo g);
void insere_aresta(p_grafo g, int u, int v);
void remove_aresta(p_grafo g, int u, int v);
int tem_aresta(p_grafo g, int u, int v);
void imprime_arestas(p_grafo g);
. . .
```

    - Por que ponteiro de ponteiro?
        - Porque cada vetor é um vetor
        - Vetor simples → Ponteiro simples
    - As arestas tendem a modificar mais
    - Matriz para o pc é só uma forma de abstrair o vetor

- ### Inicialização e destruição
    - Para cada um tem que alocar o vetor correspondente

```
p_grafo criar_grafo(int n) { //Inicialização
      int i, j;
      p_grafo g = malloc(sizeof(Grafo));
      g->n = n;
      g->adj = malloc(n * sizeof(int *)); //Aloca int * pois é um ponteiro longo → 8 bytes
//Ponteiro duplo → Aponta para outros ponteiros, aqui alocamos o ponteiro para n ponteiros → 1ª dimensão
      for (i = 0; i < n; i++)
         g->adj[i] = malloc(n * sizeof(int));//Para cada ponteiro aloca um vetor de tamanho n → 2ª dimensão
      for (i = 0; i < n; i++)
         for (j = 0; j < n; j++)
            g->adj[i][j] = 0;
      return g;
}
```



```
void destroi_grafo(p_grafo g) { // Destruição
   int i;
   for (i = 0; i < g->n; i++)
      free(g->adj[i]);
   free(g->adj);//Se fosse antes, os n vetores ficariam perdidos, “órfão”
   free(g); //Por que o free + null? Free → Desmarca o endereço mas não zera o que está dentro
Free + null → Desmarca o endereço e zera o que estava alocado
//Cada malloc deve ter seu free
 }
```

- ### Manipulando Arestas

```
void insere_aresta(p_grafo g, int u, int v) {
  g->adj[u][v] = 1; //Ligando os bits da matriz
  g->adj[v][u] = 1;
}
```

```
void remove_aresta(p_grafo g, int u, int v) {
  g->adj[u][v] = 0; //Apenas muda para 0
  g->adj[v][u] = 0;
}
```

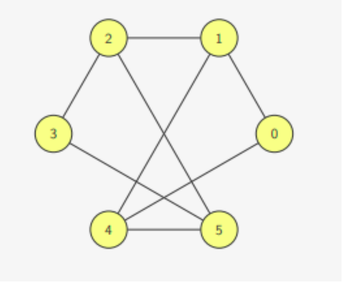
```c
int tem_aresta(p_grafo g, int u, int v) {
  return g->adj[u][v];
}
```

- ### Lendo e Imprimindo um Grafo

```c
p_grafo le_grafo() {
   int n, m, i, u, v;
   p_grafo g;
   scanf("%d %d", &n, &m); //Quantidade de vértices e quantidade de arestas
   g = criar_grafo(n);
   for (i = 0; i < m; i++) {
      scanf("%d %d", &u, &v); //Ler o par de vértices interligados por uma aresta
      insere_aresta(g, u, v);
   }
 return g;
}
```

```c
void imprime_arestas(p_grafo g) {
   int u, v;
   for (u = 0; u < g->n; u++)
      for (v = u + 1; v < g->n; v++)
         if (g->adj[u][v])
            printf("{%d,%d}\n", u, v); //Só imprime o que for 1
}
```

- ### Quem é o mais popular?
  - O **grau** de um vértice é o seu número de vizinhos
  - Mesmo conceito em árvores
  - **Diferença entre grafos e árvore** = Filhos diferentes podem ter o mesmo pai → **BOA QUESTÃO DE PROVA**

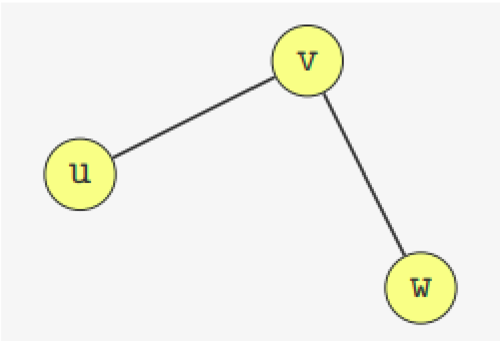


→ **2 = Grau 3**
→ **1 = Grau 3**
→ **3 = Grau 2**

```c
int grau(p_grafo g, int u) {
   int v, grau = 0;
   for (v = 0; v < g->n; v++)
      if (g->adj[u][v])
         grau++;
   return grau;
 }
```

```c
 int mais_popular(p_grafo g) { //Mais ligações → o(n²) → grau_max
//A complexidade vai aumentando por conta da recursividade
//Percorrer todos para descobrir o maior grau
    int u, max, grau_max , grau_atual;
    max = 0;
    grau_max = grau(g, 0);
    for (u = 1; u < g->n; u++) {
       grau_atual = grau(g, u);
       if (grau_atual > grau_max) {
          grau_max = grau_atual;
          max = u;
       }
    }
    return max;
 }
```

- **Indicando amigos**




```c
void imprime_recomendacoes(p_grafo g, int u) {
   int v, w;
   for (v = 0; v < g->n; v++) {
      if (g->adj[u][v]) {  //Visitando vértice v que é vizinho do
vértice u
         for (w = 0; w < g->n; w++) {
            if (g->adj[v][w] && w != u && !g->adj[u][w])//Pegando em não
é o u e que não são amigos de uqu
               printf("%d\n", w);
         }
      }
```

```
    }
  }
```

Arco é um par não ordenado → não importa a ordem, com arcos podemos ter direções.
todo arco é um dígrafo → basta considerar cada aresta como dois arcos
→ A matriz de um grafo é simétrica → **falsa** (**boa questão de prova**)
priorizaremos lista por adjacencias quando tivermos mt elementos, pois é mais caro armazenar na memória

- **Caminhos em grafos**
  - Uma sequência **sem repetições de vértices vizinhas**
  - Começando em **s** e terminando em **t**
- **Componentes Conexas**
  - Um grafo pode ter várias "partes"
  - Partes que estão conectados entre si
  - Definição
    - Um par de vértices está no mesmo componente se e somente se **existe caminho entre eles**
      - Não há caminhos entre vértices de componentes distintas
    - Um grafo conexo tem apenas uma componente conexa
  - **Busca em profundidade**
    - Vá o máximo possível em uma direção
    - Se não encontramos o vértice, volte o mínimo possível
    - E pegue um novo caminho por um vértice não visitado
    - Estrutura recursiva para ir nos vizinhos não visitados até então.
    - Escolhe o índice de menor número/valor
    - Caso não encontremos o vértice procurado visitando todos os vértices chegamos ao final dessa busca → **não há caminho para este.**
  - Encontrar um caminho s a t:
    1. Começo de v = s
    2. Para cada vizinho w de v
       2.1. Se visitado[w] == 0, visito w
       → w não foi visitado

Ciclos em Grafos

- Uma sequência de vértices vizinhos sem repetição **exceto pelo primeiro e o último vértice que são idênticos**
- **SEM REPETIÇÃO**

Árvores

- Uma árvore é um grafo conexo acíclico → se fecharmos a árvore formamos um grafo, por isso dizemos que grafo é um tipo especial de árvore
  - Uma floresta é um grafo acíclico

    - Suas componentes conexas são árvores
- Um subgrafo é um grafo obtido a partir da remoção de vértices e arestas
    - Podemos considerar também árvores → florestas que são subgrafos de um grafo dado

Caminho não repete vértice

```
typedef struct {
      int **adj;
      int n;
} Grafo;

typedef Grafo *p_grafo;
(matriz de adjancências)
typedef Struct no {
      int v;
      struct no*prox;
}no

typedef no *p_prox;
typedef Struct {
      p_no *adjacencia;
      int n;
}Grafo;
```

ELE VAI DAR UM GRAFO E TEMOS QUE EXECUTAR O ALGORITMO DE BUSCA EM PROFUNDIDADE PARA MOSTRAR O CAMINHO –:> Boa questão de prova