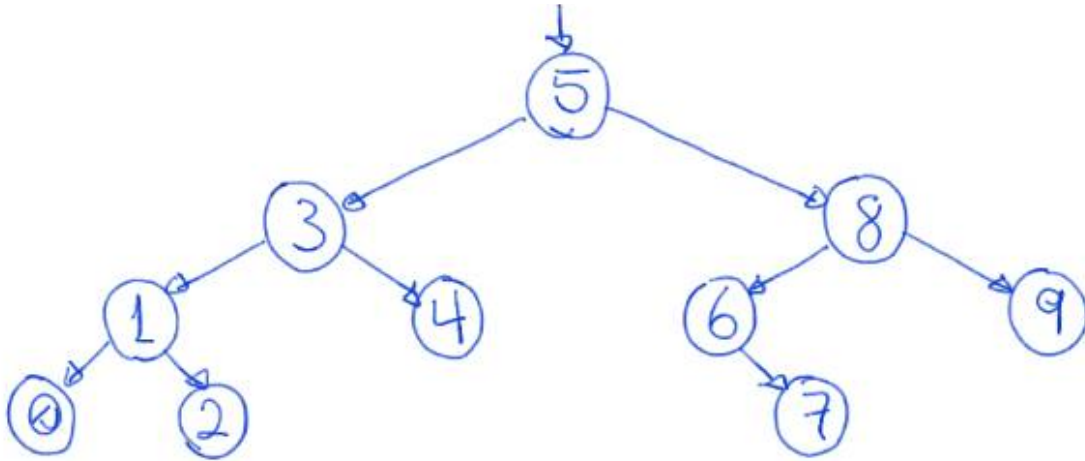


# Algoritmos e Estruturas de Dados 1 (AED1)

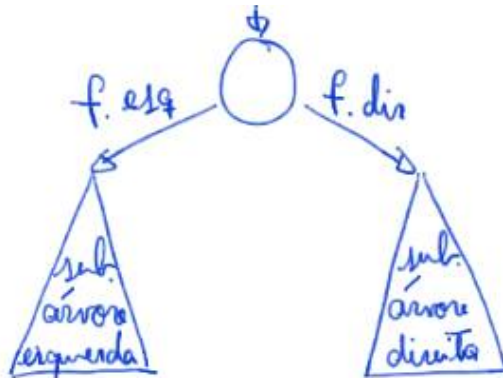
## Árvores binárias

Exemplo:

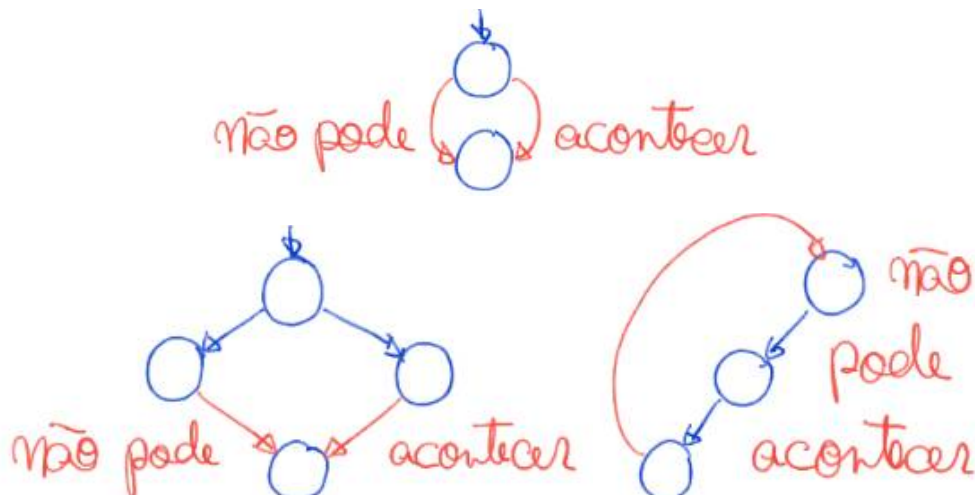


Definição de uma árvore binária:

- Temos a propriedade recursiva, segundo a qual toda árvore binária
  - é um elemento com uma subárvore esquerda e uma subárvore direita



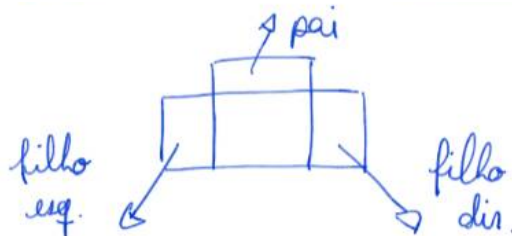
- ou é uma árvore vazia.
- Adicionamos à propriedade recursiva que
  1. cada elemento de uma árvore tem no máximo um pai,
    - a. sendo que o único elemento sem pai é a raiz.
  2. os filhos esquerdo e direito de cada elemento são distintos.
- É interessante verificar quais anomalias essas propriedades evitam
  - como união de caminhos e ciclos.



- A propriedade recursiva vai nos ajudar a pensar nas operações.

Cada elemento de uma árvore binária é armazenado em um nó,

- implementado como um registro que possui os campos:
  - conteúdo,
  - apontador para o filho esquerdo,
  - apontador para o filho direito,
  - apontador para o pai
    - campo opcional, corresponde ao campo anterior
      - usado em listas duplamente ligadas,
    - e só precisamos dele para algumas operações.



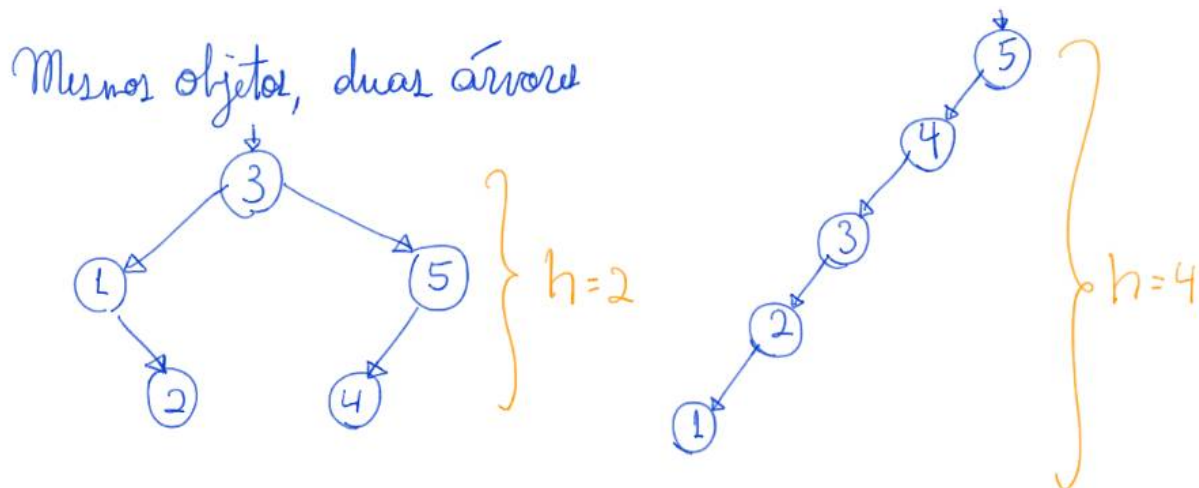
```
typedef int Cont;  
  
typedef struct noh {  
    Cont conteudo;  
    struct noh *pai; // opcional  
    struct noh *esq;  
    struct noh *dir;  
} Noh;
```

Notação e convenções:

- Usamos o termo **árvore** para nos referir
    - tanto ao conjunto de elementos que compõe uma árvore
    - quanto ao endereço da raiz de uma árvore.
      - Por isso, o uso da definição do tipo Arvore
- ```
typedef Noh *Arvore;
```
- Definimos a **subárvore** de um nó x, como sendo
    - x e seu conjunto de nós descendentes,
      - i.e., todos os nós para os quais existe caminho a partir de x.
    - Também podemos dizer que trata-se da árvore enraizada em x.
  - Chamamos de **folhas** os nós da árvore que não tem filhos,
    - i.e., cujos filhos são subárvores vazias.
  - Definimos a **altura** de um nó x como sendo
    - o comprimento do maior caminho de x
      - até uma folha de sua subárvore,
    - i.e., o número de saltos entre nós em tal caminho.
  - A altura (h) de uma árvore é a altura do nó raiz da mesma.

Vale notar que, árvores binárias diferentes,

- podem armazenar o mesmo conjunto de objetos. Por exemplo:



Observando as árvores anteriores, da esquerda para a direita, temos que:

- O nó com elemento 3 é **raiz** da primeira
  - e o nó 5 é **raiz** da segunda.
- Os nós 2 e 4 são **folhas** da primeira,
  - e o nó 1 é **folha** da segunda.
- A **altura** da primeira é 2
  - e da segunda é 4.

Quiz1: Como podemos observar,

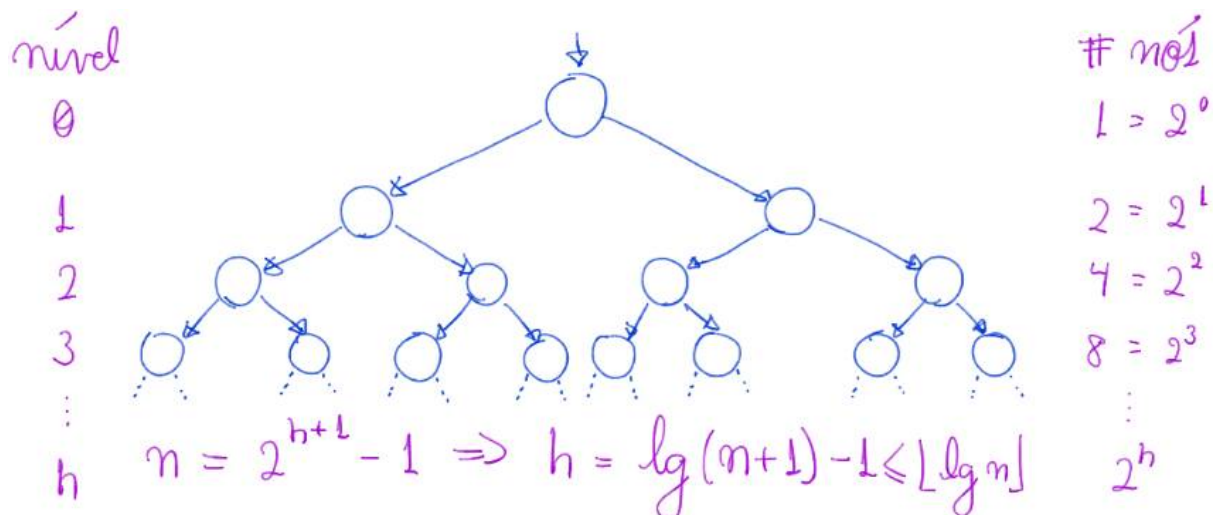
- a altura de uma árvore binária com  $n$  nós pode variar muito.
- Qual a estrutura/formato da árvore binária com maior altura?
  - E da árvore com menor altura?

No pior caso,

- a árvore terá altura  $n - 1$ , caso cada nó tenha apenas um filho.
  - Note que esta árvore corresponde a uma lista ligada.

No melhor caso,

- a altura será  $\approx \lg n$ , caso seja completa ou quase completa,
  - caso em que todos os níveis estão cheios, exceto talvez o último.



Sendo  $n$  o número total de nós da árvore, temos:

Calcular a soma dos termos de uma PG de razão 2

$$n = 1 + 2 + 4 + 8 + \dots + 2^h$$

$$(a) \quad n = 2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^h$$

$$(b) \quad 2 \cdot n = 2^1 + 2^2 + 2^3 + \dots + 2^h + 2^{h+1}$$

Subtraindo (a) de (b) temos:

$$(2-1)n = (2^1 + \dots + 2^h + 2^{h+1}) - (2^0 + 2^1 + \dots + 2^h)$$

$$n = 2^{h+1} - 1$$

Para calcular a altura de uma árvore, podemos explorar sua estrutura recursiva.

Altura: se árvore corrente não for vazia

- obtenha recursivamente a altura da subárvore esquerda,
- obtenha recursivamente a altura da subárvore direita,
- devolva 1 mais a altura da maior subárvore.

```
int altura(Arvore r) {
    int hesq, hdir;
    if (r == NULL) // atenção para o valor devolvido no caso base
        return -1;
    hesq = altura(r->esq);
    hdir = altura(r->dir);
    if (hesq > hdir)
        return hesq + 1;
    return hdir + 1;
}
```

- Eficiência de tempo  $O(n)$ , pois cada nó é visitado uma vez.
- Eficiência de espaço  $O(\text{altura})$ ,
  - por conta da altura máxima da pilha de execução/recursão,
    - que pode crescer tanto quanto a altura da árvore.

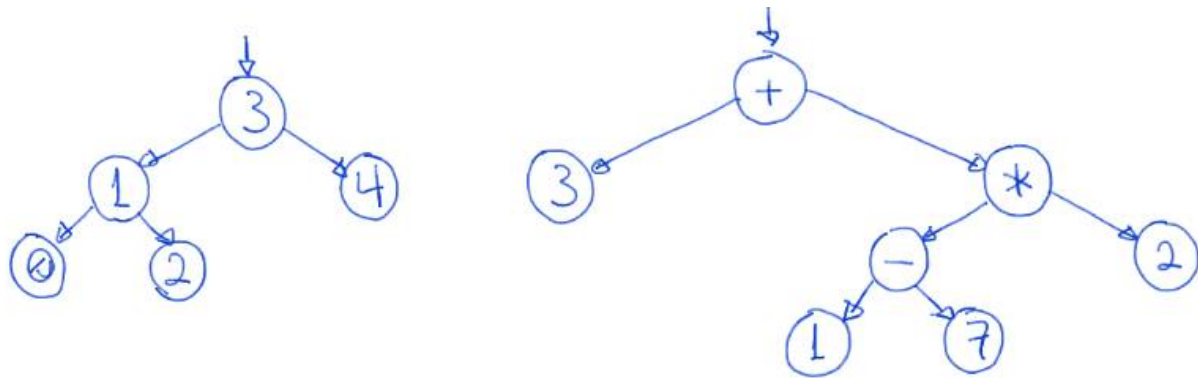
Árvores balanceadas: uma árvore binária é balanceada

- se as subárvores esquerda e direita de cada nó
  - tiverem aproximadamente a mesma altura.
- Neste caso, a altura da árvore é da ordem de  $\lg n$ , i.e.,  $O(\log n)$ .
- Isto é importante pois, como veremos em seguida,
  - diversas operações na árvore levam tempo proporcional à altura.
- Infelizmente, é fácil que uma árvore binária fique desbalanceada,
  - conforme ocorrem inserções e remoções.

Agora vamos discutir como implementar algumas operações numa árvore binária

- e vamos avaliar a eficiência das mesmas.
  - Atentem que diversas operações não utilizam o campo pai.

Exemplos para percursos:



Percurso in-ordem: se árvore corrente não for vazia

- chame recursivamente “percurso in-ordem” para subárvore do filho esquerdo
- devolva objeto da raiz
- chame recursivamente “percurso in-ordem” para subárvore do filho direito

```
void inOrdem(Arvore r) {
    if (r != NULL) {
        inOrdem(r->esq);
        printf("(%d) ", r->conteudo);
        inOrdem(r->dir);
    }
}
```

- Esta forma de varredura de árvore é chamada de in-ordem
  - pois a raiz de cada subárvore é visitada entre os filhos.
- Também é conhecida por e-r-d,
  - referência à esquerda-raiz-direita.
- Implementar este percurso sem usar recursão, demanda uma pilha.

```
void inOrdemI(Arvore r) {
    Noh *p, *pilha[100];
    int topo = 0; // inicializa a pilha
    p = r;       // começa pela raiz
    // enquanto nó corrente não for nulo ou houverem nós na pilha
    while (p != NULL || topo > 0) {
        if (p != NULL) {
            pilha[topo++] = p; // empilha p
            p = p->esq;        // visita filho esquerdo de p
        } else {
            p = pilha[--topo]; // desempilha
            printf("(%d) ", p->conteudo);
            p = p->dir; // visita o filho direito de p
        }
    }
}
```

- Quiz2: Tente implementar o percurso in-ordem sem pilha ou recursão,
  - para perceber a importância das mesmas.

- Eficiência de tempo  $O(n)$ ,
  - pois cada nó é visitado uma vez.
- Eficiência de espaço  $O(\text{altura})$ ,
  - por conta da altura máxima da pilha (seja de recursão ou explícita),
    - que pode crescer tanto quanto a altura da árvore.

Existem outras formas de percurso, como:

- pós-ordem ou e-d-r,
  - em que a raiz é visitada depois dos filhos

```
void posOrdem(Arvore r) {
    if (r != NULL) {
        posOrdem(r->esq);
        posOrdem(r->dir);
        printf("%d\n", r->conteudo);
    }
}
```

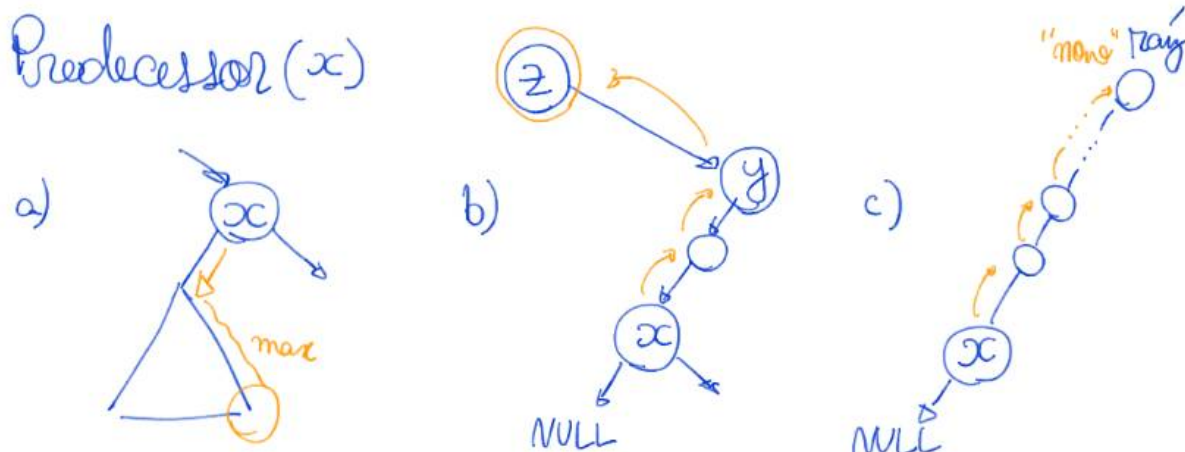
- Quiz3: Notam uma relação com notação pós-fixa?
  - Dica: apliquem o percurso na árvore de operações aritméticas.
- Pré-ordem ou r-e-d, em que a raiz é visitada antes dos filhos.
  - Quiz4: Faça a versão recursiva do percurso em pré-ordem.
    - Faça também a versão iterativa com pilha.

- Quiz5: Percorrer uma árvore por níveis, imprimindo todos os nós de um nível,
- antes de passar para os nós do próximo nível,
  - sendo que começamos pela raiz,
    - e o nível de um nó corresponde à distância da raiz até ele.
  - Este tipo de percurso não usa recursão ou pilha, mas sim uma fila
    - e lembra nosso algoritmo para cálculo de distâncias.

Predecessor (sucessor): dado um apontador para um nó queremos o apontador

- para o nó que o precede (sucede) no percurso in-ordem.

- Note que o primeiro (último) nó do percurso
  - não tem predecessor (sucessor).
- Para implementar essas operações, vamos usar em cada nó,
  - um apontador para o pai do mesmo.
  - O pai da raiz é NULL.



- Procedimento, sendo x o nó corrente:
  - a) se o filho esquerdo (direito) de x é não vazio,
    - devolva o nó mais à direita da subárvore enraizada neste filho.
  - b) caso contrário, siga repetidamente apontadores
    - para o antecessor de x até visitar nós y e z
      - tal que y é filho direito (esquerdo) de z.
      - Devolva z.
    - Observe que, x é o nó mais à esquerda (direita)
      - da subárvore direita (esquerda) de z.
      - Portanto, x é sucessor (predecessor) de z.
  - c) se não encontrar, devolva “none”.

```
Noh *predecessor(Noh *x) {
    Noh *p;
    if (x->esq != NULL) {
        p = x->esq;
        while (p->dir != NULL)
            p = p->dir;
    } else {
        p = x->pai;
        while (p != NULL && p->esq == x) {
            x = p;
            p = p->pai;
        }
    }
    return p;
}
```

- Eficiência de tempo  $O(\text{altura})$ ,
  - pois no primeiro laço o apontador desce em uma subárvore,
  - no segundo laço ele sobe em direção à raiz
  - e apenas um dos laços ocorre.
- Eficiência de espaço  $O(1)$ ,
  - pois número e tamanho de variáveis auxiliares
    - não depende do tamanho da árvore.

Quiz6: Implemente a operação sucessor,

- que é complementar à predecessor,
- i.e., queremos o nó que sucede o atual no percurso in-ordem.

Quiz7: Suponha que os apontadores pai estão vazios em uma árvore.

- Como projetar uma função para preenchê-los?