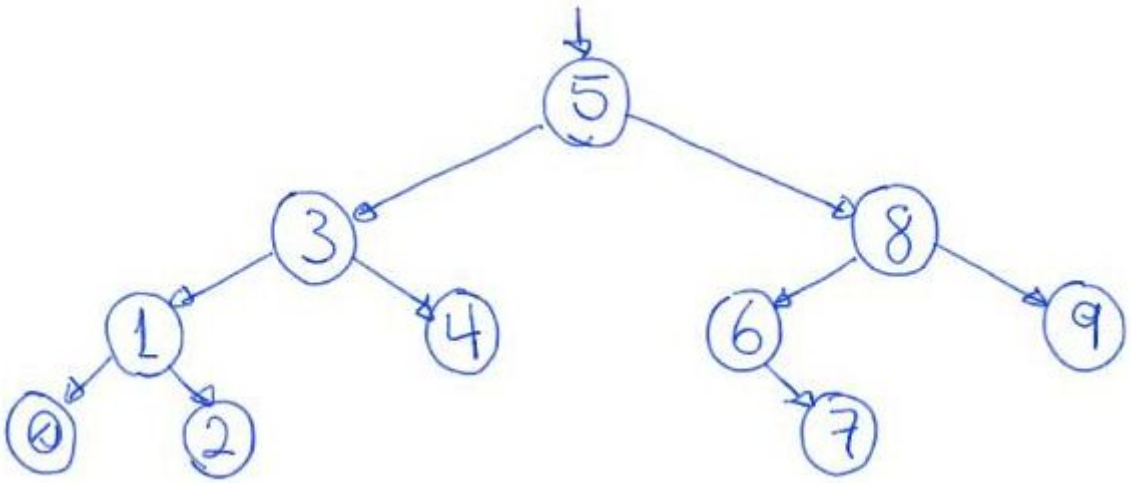
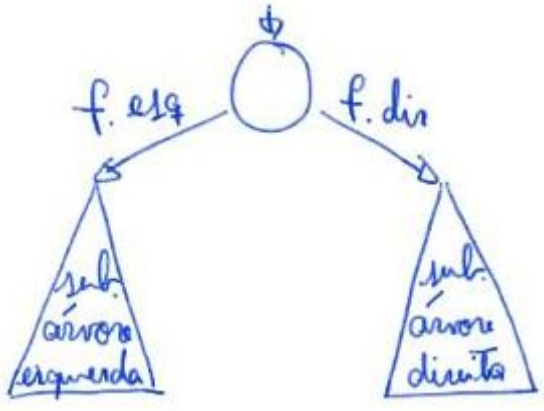
AED1 - Aula 21
Árvores binárias, tabelas de símbolos, árvores binárias de busca (operações básicas)

Árvores binárias

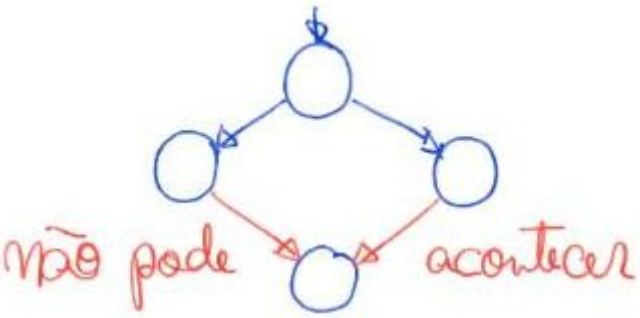


Definição de uma árvore binária:

- 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.



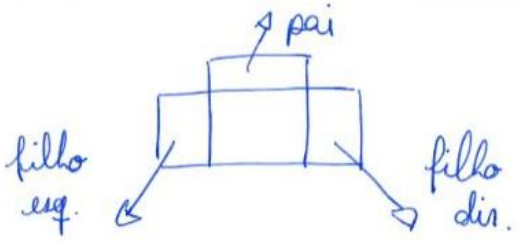
- Adicionamos à propriedade recursiva que
  - cada elemento de uma árvore tem no máximo um pai,
    - sendo que o único elemento sem pai é a raíz.
  - os filhos esquerdo e direito de cada elemento são distintos.
- É interessante verificar quais anomalias essas propriedades evitam
  - como união de caminhos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



- Destacamos que,
    - 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 encadeadas,
        - e só precisamos dele para algumas operações.




```c
typedef int Cont;

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

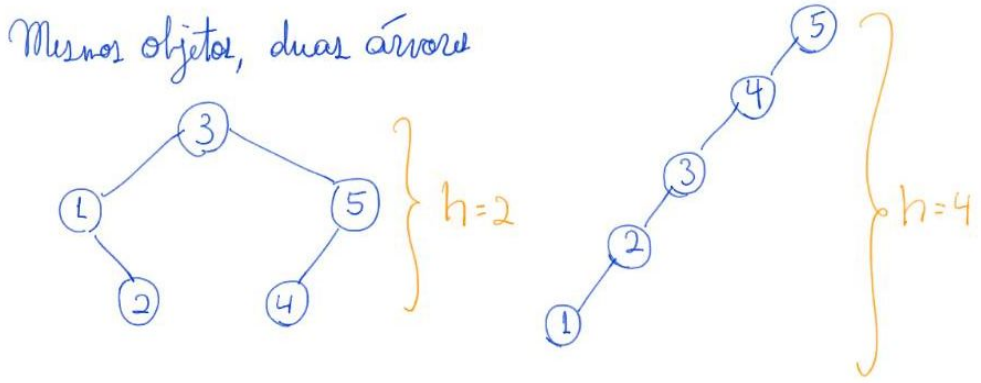
```
} Noh;
```

Notação e convenções:

- Usamos o termo árvore para nos referir
    - tanto ao conjunto de elementos que compõe um árvore
    - quanto ao endereço da raiz de uma árvore.
        - Por isso, o uso da definição de 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:



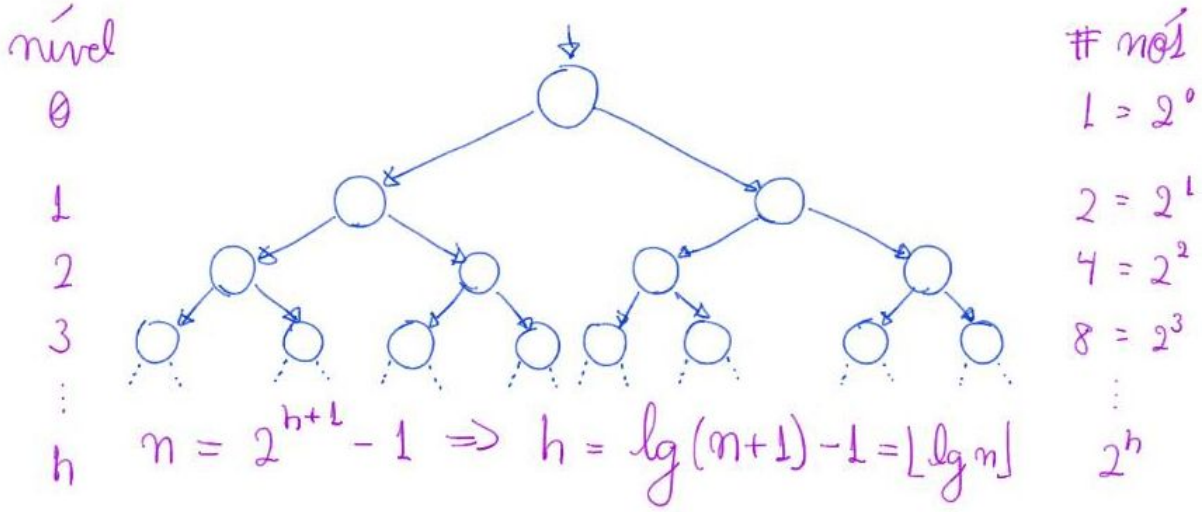
Observando as árvores anteriores, da esquerda para a direita, temos que:

- 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.

Como pudemos observar,

- a altura de uma árvore binária com n nós pode variar muito:
    - desde n - 1, caso seja uma lista encadeada,

    - até ~= lg n, caso seja completa ou quase completa, caso em que
        - todos os níveis estão cheios, exceto talvez o último.



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

- altura - com eficiência O(n)
    - 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)
       return -1;
   hesq = altura(r->esq);
   hdir = altura(r->dir);
   if (hesq > hdir)
       return hesq + 1;
   return hdir + 1;
}
```

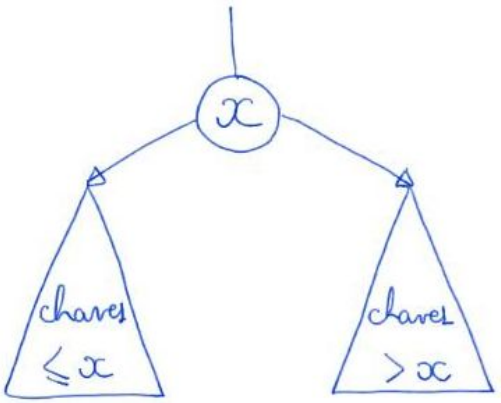
Árvores balanceadas

- Uma árvore binária é balanceada se as subárvores esquerda e direita
    - tiverem aproximadamente a mesma altura.
- Neste caso, a altura da árvore é da ordem de lg 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.

Árvores binárias de busca

O que diferencia uma árvore binária qualquer de uma árvore binária de busca

- é a propriedade de busca, i.e., dado um nó com chave x:
    - os elementos na subárvore esquerda tem chave <= x
    - e os objetos na subárvore direita tem chave > x.



- Observe que esta propriedade mantém os elementos ordenados na árvore.

Uma importante aplicação de árvores binárias de busca

- é na implementação de tabelas de símbolos dinâmicas.

```
typedef struct noh
{
   Chave chave;
   Cont conteudo;
   struct noh *esq;
   struct noh *dir;
} Noh;
```

Agora vamos discutir como implementar as operações numa árvore binária de busca

- e vamos avaliar a eficiência das mesmas em função da altura (h) da árvore:
    - atentem que diversas operações não utilizam o campo pai.

busca(k) - com eficiência O(altura)

- comece na raiz
- repita o seguinte processo até chegar num apontador vazio
    - se a chave do nó atual = k devolva apontador para ele
    - se k < chave do nó atual desça para o filho esquerdo
    - se k > chave do nó atual desça para o filho direito
- devolva “none”

Versão recursiva

```
Noh *TSbuscaR(Arvore r, Chave chave)
{
   if (r == NULL)
       return r;
   if (r->chave == chave)
```

```
        return r;
    if (chave < r->chave)
        return buscaR(r->esq, chave);
    // r->chave > chave
    return buscaR(r->dir, chave);
}
```

Versão iterativa

```
Noh *TSbuscaI(Arvore r, Chave chave)
{
    while (r != NULL && r->chave != chave)
    {
        if (chave < r->chave)
            r = r->esq;
        else
            r = r->dir;
    }
    return r;
}
```

min (max) - com eficiência O(altura)

- comece na raiz
- desça pelo filho esquerdo (direito) até encontrar um apontador vazio
- devolva um apontador para o último objeto visitado

```
Noh *TSmin(Arvore r)
{
    while (r->esq != NULL)
        r = r->esq;
    return r;
}
```

percurso ordenado - com eficiência O(n)

- se árvore corrente não for vazia
    - chame recursivamente “percurso ordenado” para subárvore enraizada no filho esquerdo
    - devolva objeto da raíz
    - chame recursivamente “percurso ordenado” para subárvore enraizada no filho direito

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

```c
void TSperc(TS *tab)
{
   inOrdemR(tab);
   printf("\n");
}
```

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

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

Existem outras formas de percurso, como:

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

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

- Notam uma relação com notação pós-fixa?

- r-e-d ou pré-ordem,
    - em que a raiz é visitada antes dos filhos.

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

Podemos ainda percorrer uma árvore por níveis,

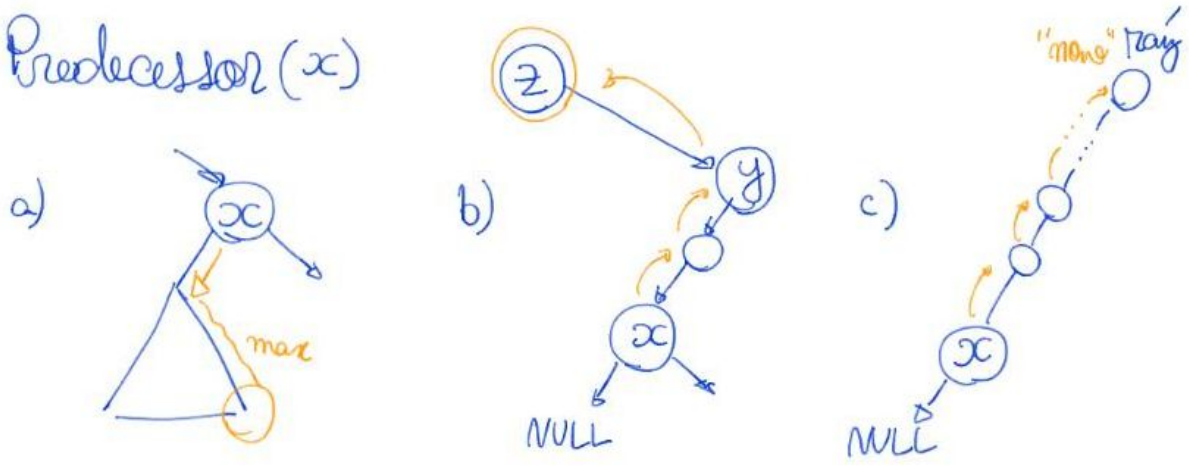
- imprimindo todos os nós de um nível,
    - antes de passar para os nós do próximo,
- sendo que começamos pela raiz,
    - e o nível de um nó corresponde à distância da raiz até ele.
- Este tipo de percurso não usa recursão ou pilha, mas sim uma fila
    - e lembra nosso algoritmo para cálculo de distâncias.

predecessor (sucessor) - com eficiência O(altura)

- para implementar essas operações, vamos usar em cada nó,
    - um apontador para o pai do mesmo.
    - O pai da raiz é NULL.
- Isso nos obriga a atualizar esses valores nas operações que alteram a árvore, i.e., inserção e remoção.

```
typedef struct noh
{
   Chave chave;
   Cont conteudo;
   struct noh *pai;
   struct noh *esq;
   struct noh *dir;
} Noh;
```

- Procedimento:
    - encontre o objeto alvo usando busca
    - a) se o filho esquerdo (direito) é não vazio, devolva max 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 min (max) da subárvore direita de z. Portanto, x é sucessor (predecessor) de z.
    - c) se não encontrar, devolva “none”.




```
Noh *TSpred(TS *tab, Chave x)
{
   Noh *q, *p;
   q = buscaI(tab, x);
   if (q == NULL)
       return NULL;
   if (q->esq != NULL)
       return max(q->esq);
   p = q->pai;
   while (p != NULL && p->esq == q)
   {
       q = p;
       p = p->pai;
   }
   return p;
}
```

- Suponha que estes apontadores estão vazios em uma árvore,
    - como projetar uma função para preenchê-los?
- Também é possível modificar a busca, para que ela guarde
    - o antepassado mais recente do nó com chave menor que ele.
    - Assim, podemos implementar estas operações sem usar o campo pai.