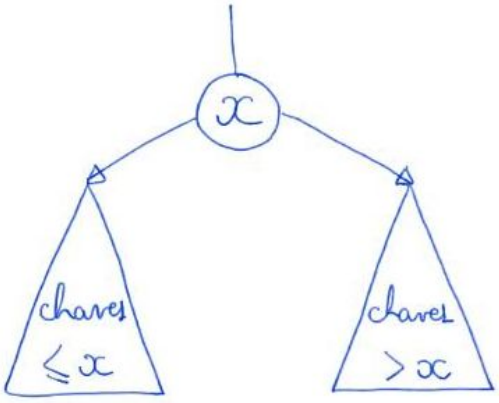
AED1 - Aula 22
Árvores binárias de busca (operações avançadas), tabelas de símbolos

Árvores binárias de busca

São árvores binárias que respeitam 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;
   int tam;
   struct noh *pai;
   struct noh *esq;
   struct noh *dir;
} Noh;
```

Vamos seguir discutindo como implementar as operações

- numa árvore binária de busca, além de analisar a eficiência das mesmas
  - em função da altura (h) da árvore.

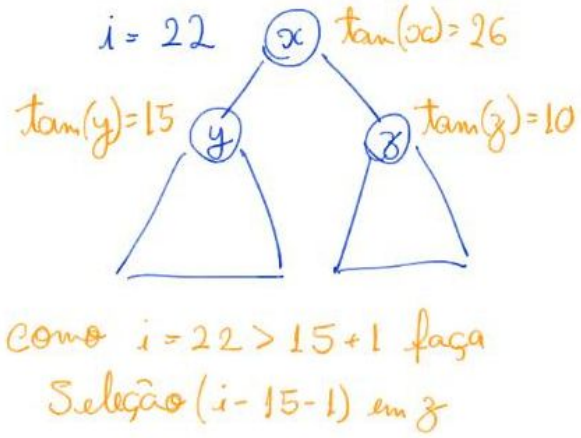
seleção(i) - com eficiência O(altura)

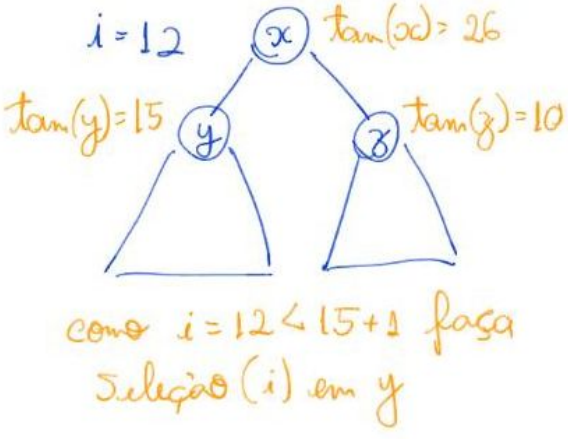
- para ficar eficiente é necessário armazenar, em cada nó,
  - o número de objetos (tam) na árvore enraizada neste objeto.

- 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;
   int tam;
   struct noh *pai; // opcional
   struct noh *esq;
   struct noh *dir;
} Noh;
```

- Note que, dada uma árvore com raíz x, filho esquerdo y e filho direito z, temos a relação:
    - tam(x) = tam(y) + tam(z) + 1
- Procedimento:
    - comece na raiz
        - seja tam_fe = tam(filho esquerdo)
        - se i = tam_fe + 1 devolva um apontador para a raiz
        - se i < tam_fe + 1 chame “selecao(i)” recursivamente na subárvore esquerda
        - se i > tam_fe + 1 chame “selecao(i - tam_fe - 1) recursivamente na subárvore direita






```
Noh *TSselec(Arvore r, int i)
{
   int t_esq;
   if (r == NULL)
       return NULL;
   if (r->esq != NULL)
       t_esq = r->esq->tam;
   else
       t_esq = 0;
   if (i == t_esq + 1)
       return r;
   if (i < t_esq + 1)
```

```
        return TSselec(r->esq, i);
    // i > t_esq + 1
    return TSselec(r->dir, i - t_esq - 1);
}
```

rank(k) - com eficiência O(altura)

- assim como no caso da seleção, para ficar eficiente é necessário armazenar, em cada nó, o número de objetos (tam) na árvore enraizada neste objeto.
    - lembrar de atualizar o valor de tam nas operações que modificam a árvore.
- note que, o rank de uma chave k corresponde ao número de objetos com chave menor ou igual a k.
    - por isso a ideia é fazer uma busca em que vamos somando o número de nós que ficou à esquerda do caminho percorrido.
- Procedimento:
    - comece na raiz, com uma variável rank = 0.
    - repita o seguinte processo até chegar num apontador vazio
        - se k < chave do nó atual desça para o filho esquerdo
        - caso contrário
            - rank += tam(filho esquerdo) + 1
            - se a chave do nó atual = k então devolva rank
            - se k > chave do nó atual então desça para o filho direito
    - devolva rank

```
int TSrank(Arvore r, Chave chave)
{
    int tam = 0, t_esq;
    while (r != NULL && r->chave != chave)
    {
        if (chave < r->chave)
            r = r->esq;
        else
        {
            if (r->esq != NULL)
                t_esq = r->esq->tam;
            else
                t_esq = 0;
            tam += t_esq + 1;
            r = r->dir;
        }
    }
    if (r != NULL)
    {
        if (r->esq != NULL)
            tam += r->esq->tam;
        tam++;
```

```
    }
    return tam;
}
```

inserção - com eficiência O(altura)

- comece na raiz
- repita o seguinte processo até chegar num apontador vazio
    - se k <= chave do nó atual desça para o filho esquerdo
    - se k > chave do nó atual desça para o filho direito
- substitua o apontador vazio pelo novo objeto, atribua seu apontador pai para o objeto que o precedeu no caminho da busca e atribua NULL aos apontadores dos filhos.

```
Noh *novoNoh(Chave chave, Cont conteudo)
{
    Noh *novo;
    novo = (Noh *)malloc(sizeof(Noh));
    novo->chave = chave;
    novo->conteudo = conteudo;
    novo->esq = NULL;
    novo->dir = NULL;
    //    novo->pai = ??
    return novo;
}

Arvore insereI(Arvore r, Noh *novo)
{
    Noh *corr, *ant = NULL;
    if (r == NULL)
    {
        novo->pai = NULL;
        return novo;
    }
    corr = r;
    while (corr != NULL)
    {
        ant = corr;
        if (novo->chave <= corr->chave)
            corr = corr->esq;
        else
            corr = corr->dir;
    }
    novo->pai = ant;
    if (novo->chave <= ant->chave)
        ant->esq = novo;
    else
        ant->dir = novo;
    return r;
```

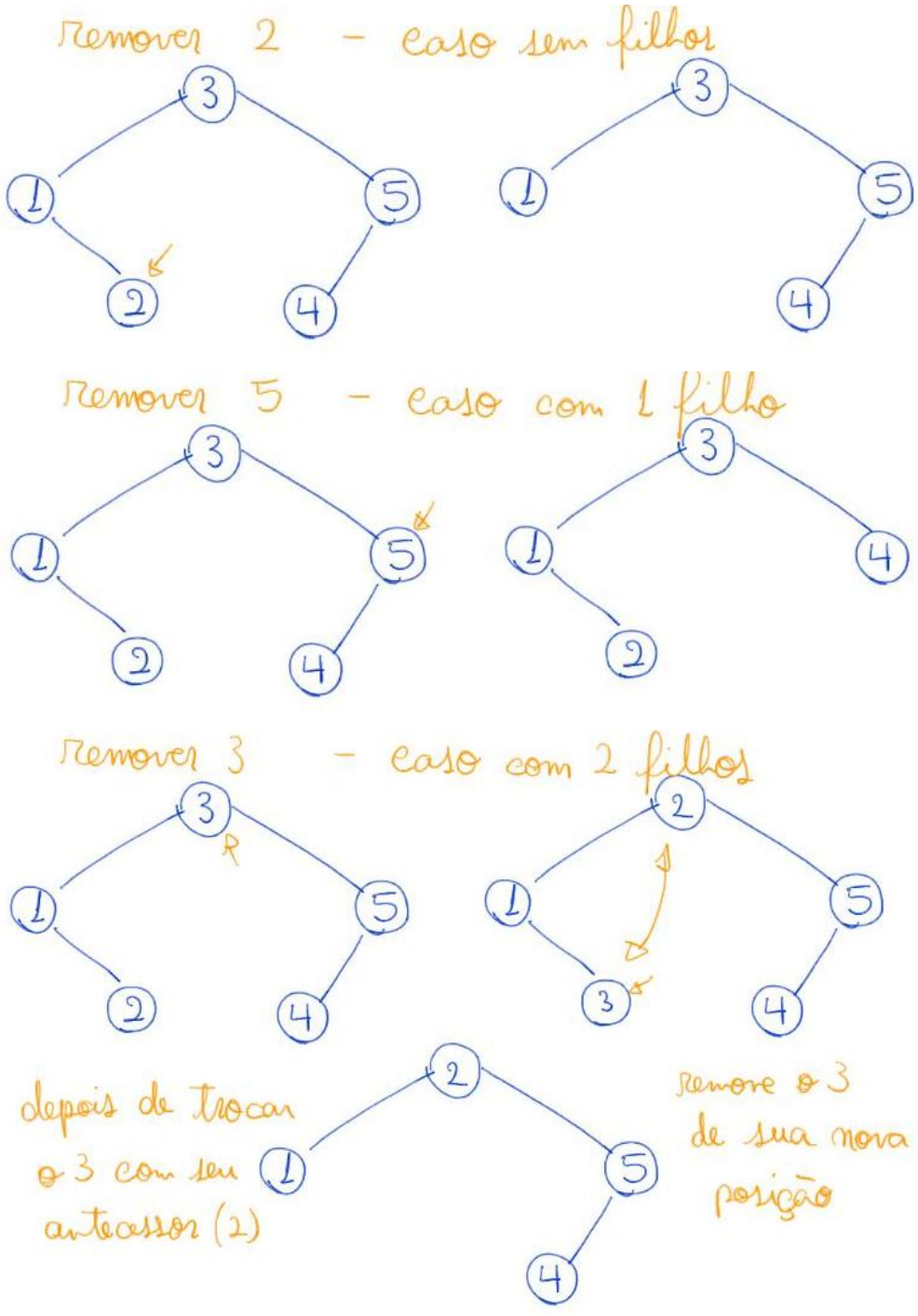
```
}

Arvore insereR(Arvore r, Noh *novo)
{
   if (r == NULL)
   {
       novo->pai = NULL;
       return novo;
   }
   if (novo->chave <= r->chave)
   {
       r->esq = insereR(r->esq, novo);
       r->esq->pai = r;
   }
   else
   {
       r->dir = insereR(r->dir, novo);
       r->dir->pai = r;
   }
   return r;
}

TS *TSinserir(TS *tab, Chave chave, Cont conteudo)
{
   Noh *novo = novoNoh(chave, conteudo);
   // return insereR(tab, novo);
   return insereI(tab, novo);
}
```

- como modificar inserção para que ela atualize correta e eficientemente o número de objetos (tam) de cada subárvore?

remoção

- use a busca para localizar um objeto x a ser removido.
    - se tal objeto não existe não há o que fazer.
- se x não possui filhos basta removê-lo e fazer o apontador de seu pai para ele igual a NULL.
    - se x fosse a raiz, a nova árvore é vazia.
- se x possui um filho conecte diretamente o pai de x com o filho de x, atualizando seus apontadores.
    - se x fosse a raiz, seu filho se torna a nova raiz.
- se x possui dois filhos troque x pelo objeto y que antecede x, ou seja,
    - pelo maior elemento da subárvore esquerda de x.
    - note que temporariamente a propriedade de busca é violada por x em sua nova posição.
    - então remova x de sua nova posição

        - note que essa remoção cairá num dos casos mais simples, já que na nova posição x não tem filho direito
            - caso contrário y não seria o maior elemento da subárvore esquerda.
- a seguir exemplos dos vários casos da remoção:




```
Arvore removeRaiz(Arvore alvo)
{
   Noh *aux, *p;
   if (alvo->esq == NULL && alvo->dir == NULL)
   {
```

```
        free(alvo);
        return NULL;
    }
    if (alvo->esq == NULL || alvo->dir == NULL)
    {
        if (alvo->esq == NULL)
            aux = alvo->dir;
        if (alvo->dir == NULL)
            aux = alvo->esq;
        aux->pai = alvo->pai;
        free(alvo);
        return aux;
    }
    aux = max(alvo->esq);
    alvo->chave = aux->chave;
    alvo->conteudo = aux->conteudo;
    p = aux->pai;
    if (p == alvo)
        p->esq = removeRaiz(aux);
    else // aux->pai != alvo
        p->dir = removeRaiz(aux);
    return alvo;
}

TS *TSremover(TS *tab, Chave chave)
{
    Noh *alvo, *p, *aux;
    alvo = buscaI(tab, chave);
    if (alvo == NULL)
        return tab;
    p = alvo->pai;
    aux = removeRaiz(alvo);
    if (p == NULL)
        return aux;
    if (p->esq == alvo)
        p->esq = aux;
    if (p->dir == alvo)
        p->dir = aux;
    return tab;
}
```

- como modificar remoção para que ela atualize correta e eficientemente o número de objetos (tam) de cada subárvore?

Resumindo opções de implementação de tabela de símbolos

Eficiência das operações em vetor ordenado:

- busca - O(log n), deriva da busca binária.

- min (max) - O(1).
- predecessor (sucessor) - O(log n), deriva da busca binária.
- percurso ordenado - O(n), mínimo possível já que é o tamanho da saída.
- seleção - O(1).
- rank - O(log n), deriva da busca binária.
- inserção - O(n).
- remoção - O(n).

Eficiência das operações em árvores binárias de busca:

- busca - O(h).
- min (max) - O(h).
- predecessor (sucessor) - O(h).
- percurso ordenado - O(n).
- seleção - O(h).
- rank - O(h).
- inserção - O(h).
- remoção - O(h).