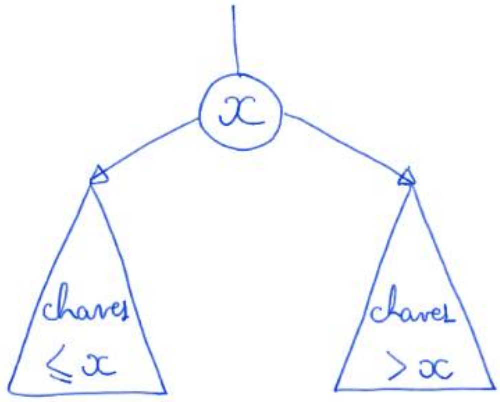
Algoritmos e Estruturas de Dados 1 (AED1)
Árvores binárias de busca (operações avançadas), tabelas de símbolos

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

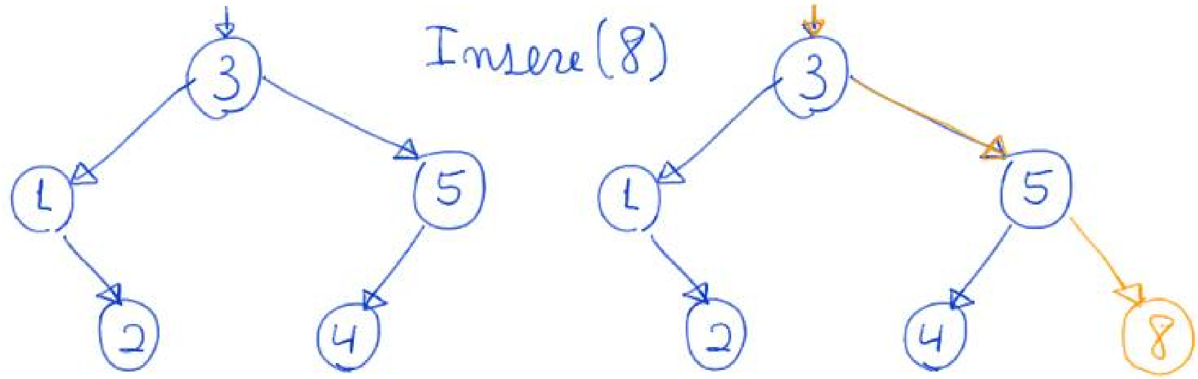
- é na implementação de tabelas de símbolos dinâmicas.

```
typedef int Chave;
typedef int Cont;

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

typedef Noh TS;
```

Vamos seguir discutindo como implementar as operações da tabela de símbolos

- usando uma árvore binária de busca,
    - além de analisar a eficiência destas em função da altura (h) da árvore.

Inserção(k):



- 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.

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

Noh *novoNoh(Chave chave, Cont conteudo) {
    Noh *novo;
    novo = (Noh *)malloc(sizeof(Noh));
    novo->chave = chave;
    novo->conteudo = conteudo;
    novo->tam = 1; // opcional
    novo->esq = NULL;
    novo->dir = NULL;
    //    novo->pai = ?? // opcional
    return novo;
}
```

- Versão recursiva

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

- Versão iterativa

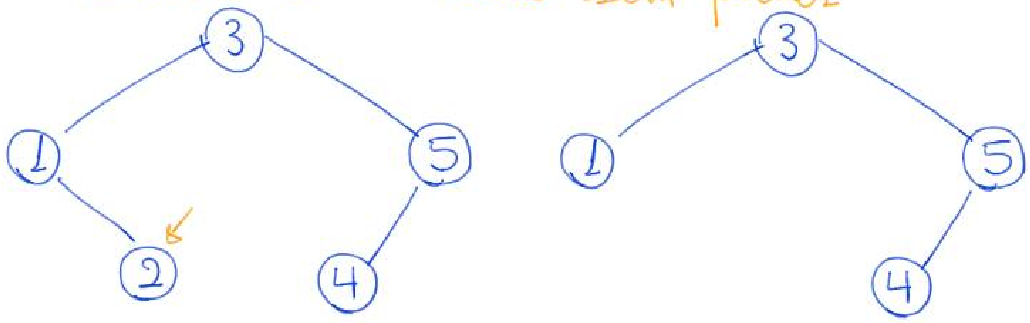
```
Arvore insereIsimples(Arvore r, Noh *novo) {
    Noh *corr, *ant = NULL;
    if (r == NULL)
        return novo;
    corr = r;
    // desce na árvore até encontrar apontador NULL
    while (corr != NULL) {
        ant = corr;
        if (novo->chave <= corr->chave)
            corr = corr->esq;
        else // novo->chave > corr->chave
            corr = corr->dir;
    }
    // insere novo noh como filho do último noh do caminho
    if (novo->chave <= ant->chave)
        ant->esq = novo;
    else // novo->chave > ant->chave
        ant->dir = novo;
    return r;
}
```

- Eficiência: tanto implementação iterativa quanto recursiva
    - leva tempo O(altura) no pior caso,
        - já que em cada chamada recursiva (ou iteração)
            - desce um nível na árvore.
- Quiz1: Modificar inserção para que ela atualize correta e eficientemente
    - o número de objetos (tam) de cada subárvore e o pai de cada nó.

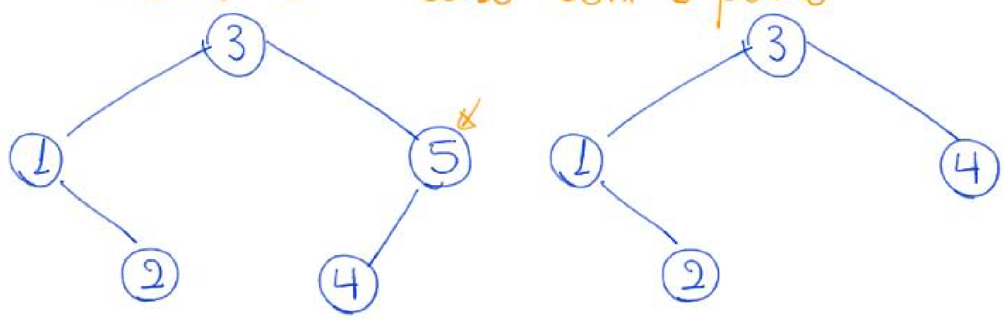
Remoção(k):

- 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:

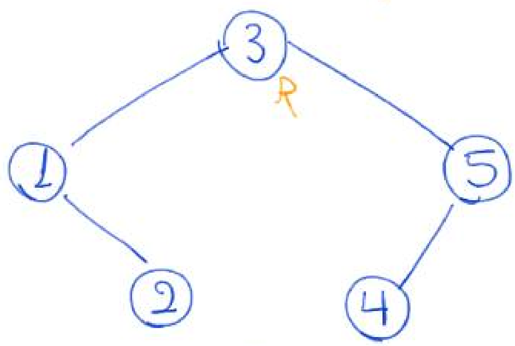
remover 2 - caso sem filhos



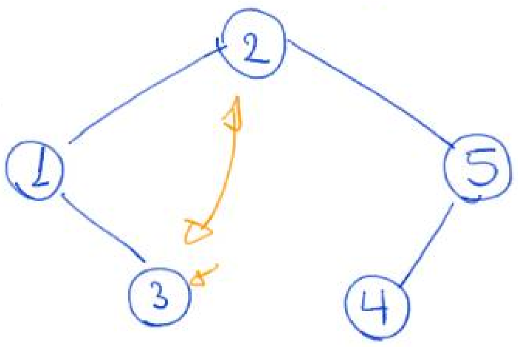
remover 5 - caso com 1 filho



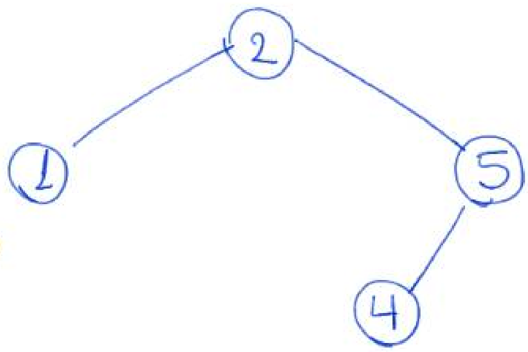
remover 3 - caso com 2 filhos



depois de trocar
o 3 com seu
antecessor (2)



remove o 3
de sua
nova posição



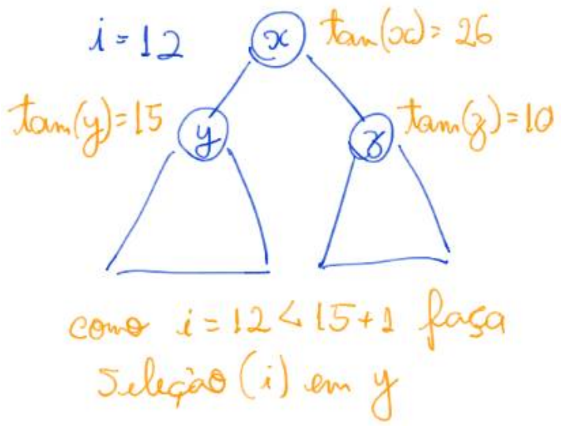
```c
TS *TSremoverSimples(TS *tab, Chave chave) {
    Noh *alvo, *pai, *aux;
    alvo = tab;
    pai = NULL;
    while (alvo != NULL && alvo->chave != chave) {
        pai = alvo;
        if (chave < alvo->chave)
            alvo = alvo->esq;
        else
            alvo = alvo->dir;
    }
    if (alvo == NULL) // nada a remover
        return tab;
    aux = removeRaizSimples(alvo);
    if (pai == NULL) // aux é a nova raiz
        return aux;
    // atualizando o campo do pai corretamente
    if (pai->esq == alvo)
        pai->esq = aux;
    if (pai->dir == alvo)
        pai->dir = aux;
    return tab;
}
```

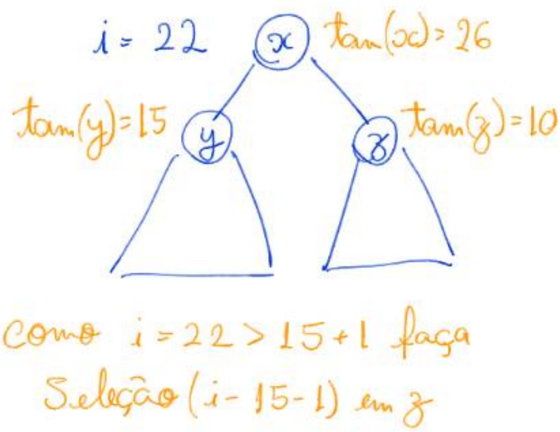
```
Arvore removeRaizSimples(Arvore alvo) {
    Noh *aux, *pai;
    if (alvo->esq == NULL && alvo->dir == NULL)
    { // Caso 1: é folha
        free(alvo);
        return NULL;
    }
    if (alvo->esq == NULL || alvo->dir == NULL)
    { // Caso 2: só tem 1 filho
        if (alvo->esq == NULL)
            aux = alvo->dir;
        else // alvo->dir == NULL
            aux = alvo->esq;
        free(alvo);
        return aux;
    }
    // Caso 3: tem 2 filhos
    aux = alvo->esq;
    pai = alvo;
    while (aux->dir != NULL) {
        pai = aux;
        aux = aux->dir;
    }
    // substitui o nó alvo pelo predecessor dele
    alvo->chave = aux->chave;
    alvo->conteudo = aux->conteudo;
```


```
    if (pai == alvo) pai->esq = removeRaizSimples(aux);
    else pai->dir = removeRaizSimples(aux);
    return alvo;
}
```

- Eficiência: leva tempo O(altura) no pior caso,
    - já que em cada iteração desce um nível na árvore.
- Quiz2: Modificar remoção para que ela atualize correta e eficientemente
    - o número de objetos (tam) de cada subárvore e o pai de cada nó.

Seleção(i): 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.
- 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 da árvore.
    - 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.

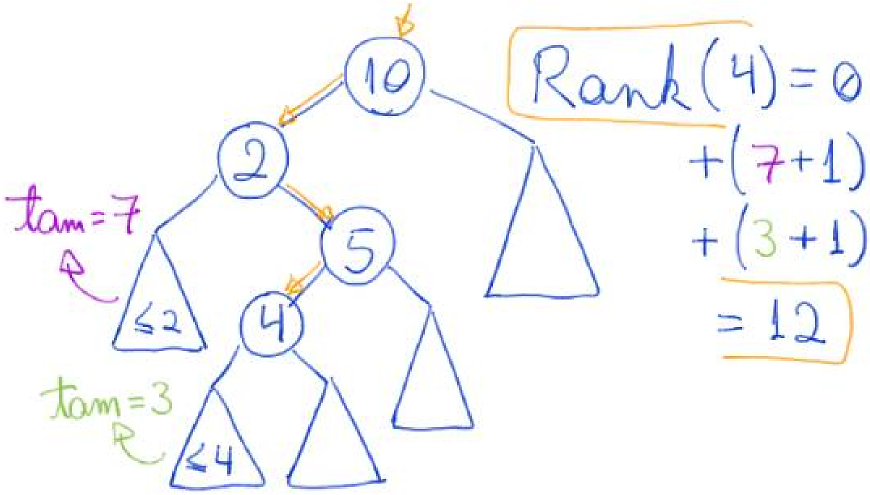





```c
Noh *TSselec(Arvore r, int pos) {
    int tam_esq;
    if (r == NULL)
        return NULL;
    if (r->esq != NULL)
        tam_esq = r->esq->tam;
    else
        tam_esq = 0;
    if (pos == tam_esq + 1)
        return r;
    if (pos < tam_esq + 1)
        return TSselec(r->esq, pos);
    // pos > tam_esq + 1
    return TSselec(r->dir, pos - tam_esq - 1);
}
```

- Eficiência: leva tempo O(altura) no pior caso,
    - já que em cada chamada recursiva desce um nível na árvore.
- Quiz3: Fazer versão iterativa da função anterior,
    - já que ela apresenta recursão caudal.

Rank(k): Assim como na seleção, para ficar eficiente é necessário armazenar
  - em cada nó o número de objetos (tam) na árvore enraizada nele.
- 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 rank = 0, tam_esq;
    while (r != NULL && r->chave != chave) {
        if (chave < r->chave) r = r->esq;
        else { // chave > r->chave
            if (r->esq != NULL) tam_esq = r->esq->tam;
            else tam_esq = 0;
            rank += tam_esq + 1;
            r = r->dir;
        }
    }
    if (r != NULL) {
        if (r->esq != NULL) rank += r->esq->tam;
        rank++;
    }
    return rank;
}
```

- Eficiência: leva tempo O(altura) no pior caso,
  - já que em cada iteração desce um nível na árvore.

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).

Como a altura (h) de uma árvore binária de busca pode variar

- desde lg n até n,
    - por exemplo se inserirmos os elementos em ordem,
- para que nossa implementação de tabela de símbolos seja eficiente
    - precisamos garantir que a altura da árvore não crescerá demais.
- Estudaremos isso em AED2 no tópico
    - árvores binárias de busca balanceadas.

Resposta Quiz1: Modificar inserção para que ela atualize correta e eficientemente

- o número de objetos (tam) de cada subárvore e o pai de cada nó?

- Versão iterativa

```
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;
        corr->tam++;
        if (novo->chave <= corr->chave)
            corr = corr->esq;
        else // novo->chave > corr->chave
            corr = corr->dir;
    }
    novo->pai = ant;
    if (novo->chave <= ant->chave) ant->esq = novo;
    else ant->dir = novo;
    return r;
}
```

- Versão recursiva

```
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 { // novo->chave > r->chave
        r->dir = insereR(r->dir, novo);
        r->dir->pai = r;
    }
    r->tam++;
    return r;
}
```

Resposta Quiz2: Modificar remoção para que ela atualize correta e eficientemente

- o número de objetos (tam) de cada subárvore?

```
TS *TSremover(TS *tab, Chave chave) {
    Noh *alvo, *pai, *aux;
    alvo = TSbusca(tab, chave);
    if (alvo == NULL) // nada a remover
        return tab;
    pai = alvo->pai;
    aux = removeRaiz(alvo);
    if (pai == NULL) // aux é a nova raiz
        return aux;
    // atualizando o campo pai corretamente
    if (pai->esq == alvo)
        pai->esq = aux;
    if (pai->dir == alvo)
        pai->dir = aux;
    return tab;
}

Arvore removeRaiz(Arvore alvo) {
    Noh *aux, *pai, *p;
    if (alvo->esq == NULL && alvo->dir == NULL)
    { // Caso 1: é folha
        p = alvo->pai;
        free(alvo);

        while (p != NULL) {
            p->tam--;
            p = p->pai;
        }

        return NULL;
    }
```

```c
    if (alvo->esq == NULL || alvo->dir == NULL)
    { // Caso 2: só tem 1 filho
        if (alvo->esq == NULL)
            aux = alvo->dir;
        else // alvo->dir == NULL
            aux = alvo->esq;
        aux->pai = alvo->pai;
        p = alvo->pai;
        free(alvo);

        while (p != NULL) {
            p->tam--;
            p = p->pai;
        }

        return aux;
    }
    // Caso 3: tem 2 filhos
    aux = TSmax(alvo->esq);
    // substitui o nó alvo pelo predecessor dele
    alvo->chave = aux->chave;
    alvo->conteudo = aux->conteudo;
    pai = aux->pai;
    if (pai == alvo)
        pai->esq = removeRaiz(aux);
    else // aux->pai != alvo
        pai->dir = removeRaiz(aux);

    p = aux->pai;
    while (p != NULL) {
        p->tam--;
        p = p->pai;
    }

    return alvo;
}
```