FaculdadeBarretos
www.faculdadebarretos.com.br

Algoritmos e Estruturas de Dados

(*Árvores Binária de Busca - ABB*

Binary Search Tree (BST))

Prof. Me. Diogo Tavares da Silva
contato: *diogotavares@unibarretos.com.br*

Contextualização

- Estruturas de dados encadeadas lineares
  - encadeamento sequencial:
    - baseada em encadeamento apenas para um predecessor e sucessor
      - **Listas, pilhas e filas**
- Estruturas de dados encadeadas não-lineares
  - encadeamento hierárquico:
  - **Árvores, Grafos, Heaps, Dicionário, Tabela hash, ...**

Árvores Binárias

- ESTRUTURA DE DADOS NÃO-LINEAR
  - **Estruturas hierárquicas**
  - Cada nó pode ter, no máximo, dois filhos:
    - o filho esquerdo e o filho direito.
  - O nó superior (início) da árvore é chamado de **raiz**
    - os demais são ramificações
      - sub-árvores esquerda e direita

Árvores Binárias de Busca (ABBs)

- ***Binary Search Trees (BST)***
  - Árvores binárias construídas com o intuito de implementar o algoritmo de busca binária em uma estrutura encadeada
    - Complexidade da busca é O(log(n)) no caso médio.
  - Em estruturas linearmente encadeadas (lista, pilha, fila, etc.), o percurso e consequentemente a busca, é sempre sequencial (complexidade O(n))

Árvores Binárias de Busca (ABBs)

- **Propriedade principal (ABB)**
  - **Estrutura ordenada**
    - dado um **nó** da ABB:
      - todos os elementos da subárvore esquerda são menores que o valor armazenado no nó
      - Todos os elementos da subárvore direita são maiores que o valor armazenado no nó
  - Todas as demais propriedades de árvores binárias são aplicadas…

Árvores Binárias de Busca (ABBs)

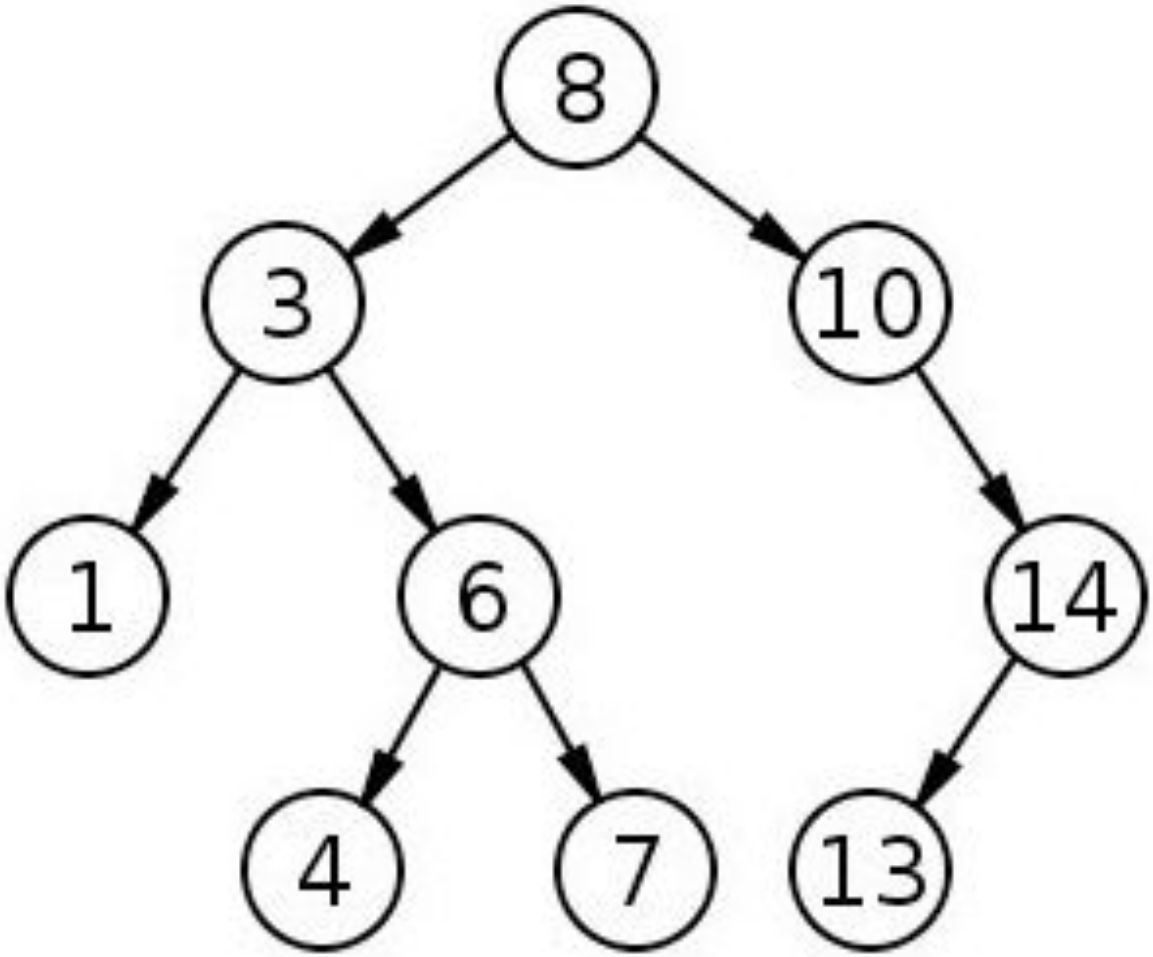
- ..ou seja,
  - Cada árvore possui apenas um Nó raiz
  - Os nós que não tem filhos são chamados de NÓS FOLHAS
  - Todos os outros nós, que não são nós folhas ou nó raiz são chamados de NÓS INTERMEDIÁRIOS.
  - Cada nó pode ter grau 0, 1 ou 2:
    - cada nó pode ter APENAS 0, 1, OU 2 FILHOS, não sendo permitido ter 3 ou mais filhos.

Árvores Binárias de Busca (ABBs)

- Resumindo:
  - em uma **ABB** (ou *BST*), para todo nó *n*:
    - Todos os elementos da esquerda devem ser menores que *n*
    - Todos os elementos da direita devem ser maiores que *n*
  - Com exceção da raiz, todo o nó está ligado por uma aresta a 1 nó predecessor (o pai)
  - Sempre existe **APENAS UM ÚNICO CAMINHO** da raiz a cada nó
    - (senão, não seria uma **árvore**)

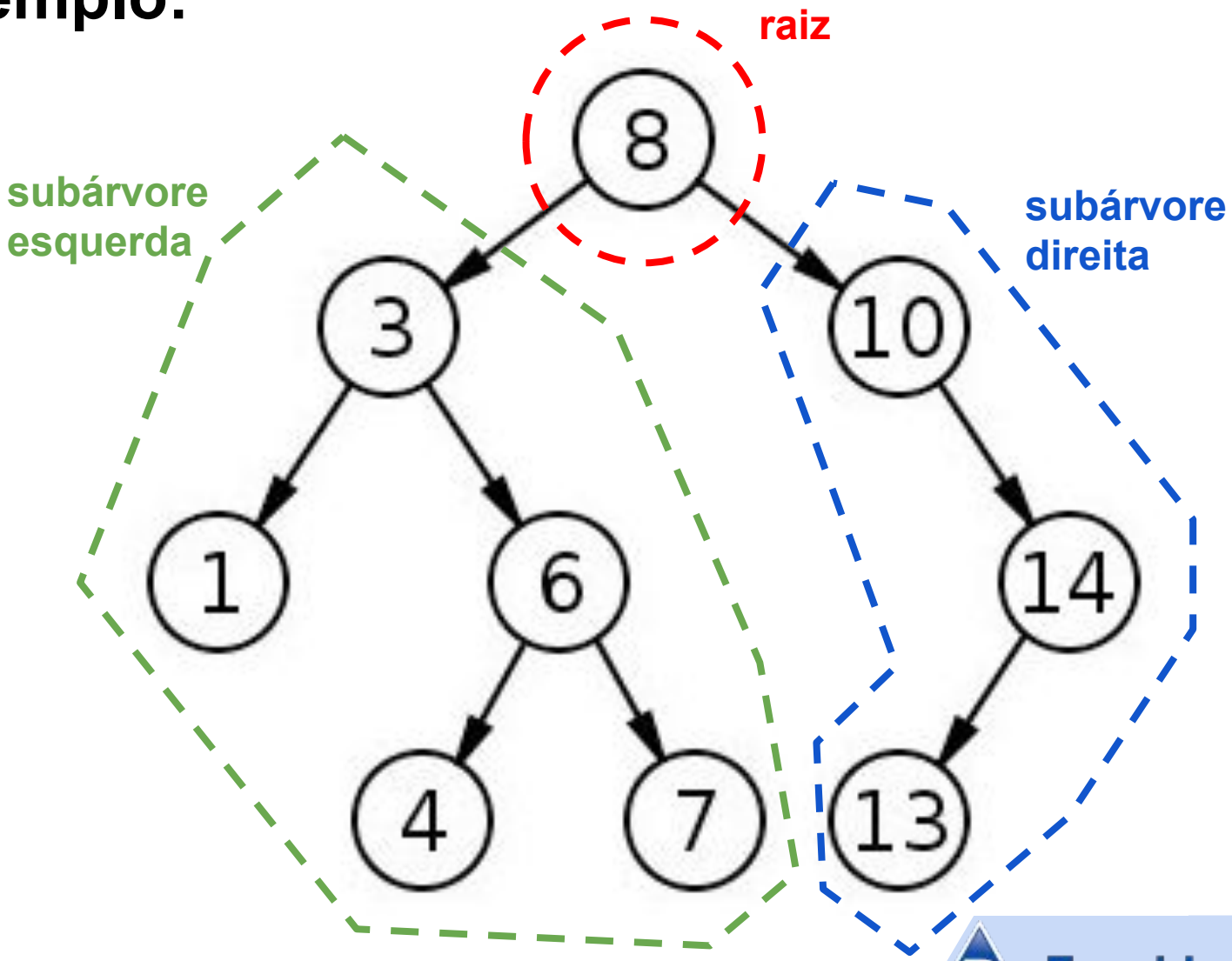
Árvores Binárias de Busca (ABBs)

- **exemplo:**



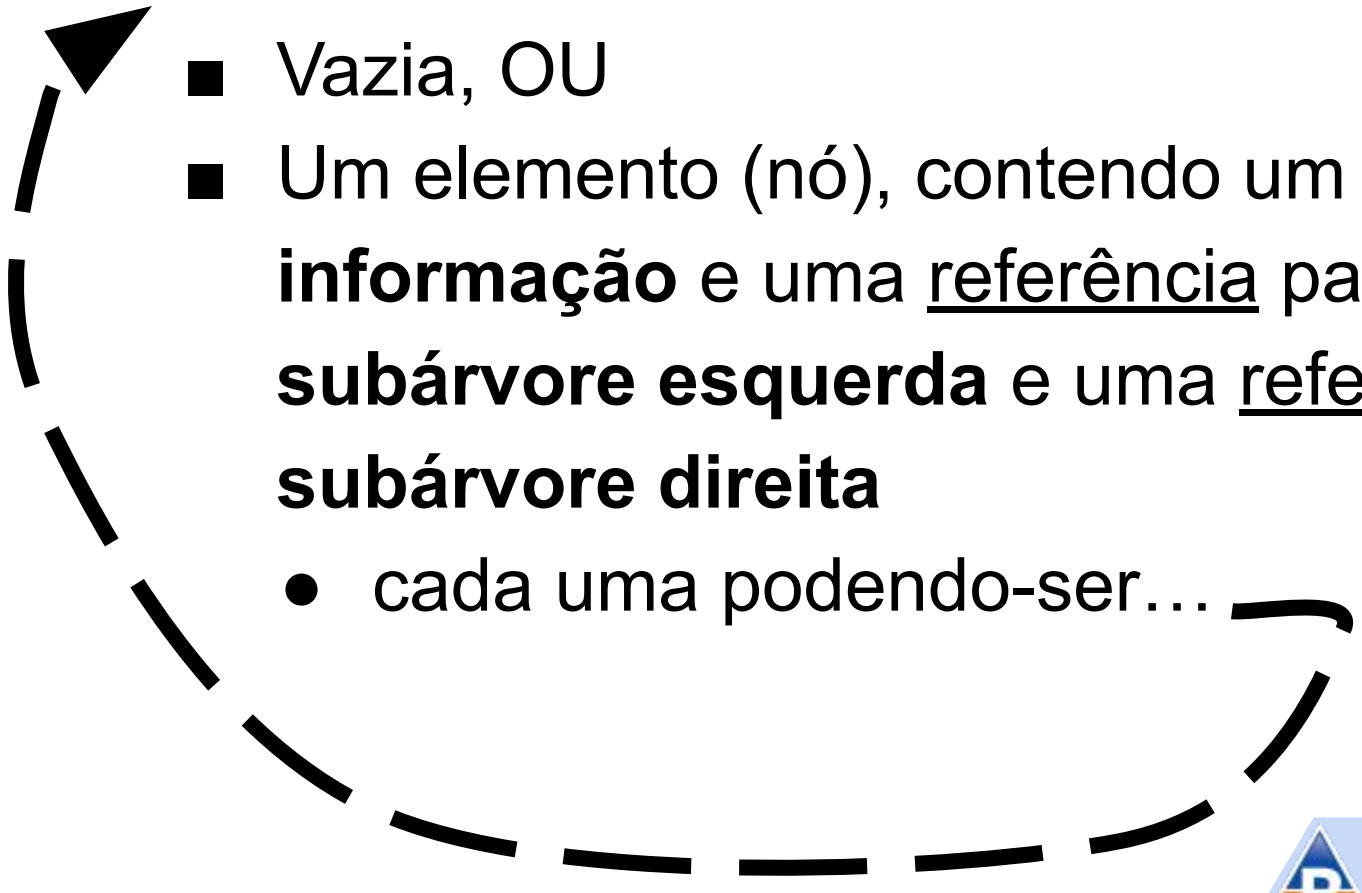
Árvores Binárias de Busca (ABBs)

- **exemplo:**



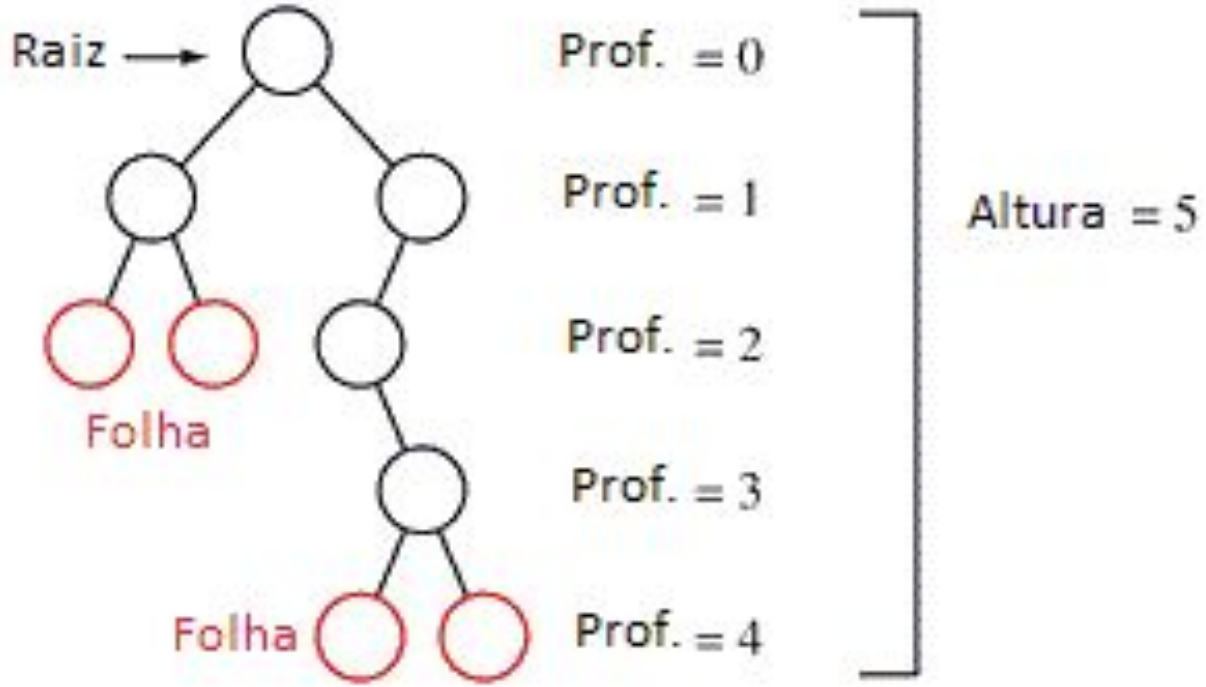
Árvores Binárias de Busca (ABBs)

- **IMPORTANTE OBSERVAR!**
  - Definição de ABB é totalmente **RECURSIVA**
  - Um árvore:
    - Vazia, OU
    - Um elemento (nó), contendo um volume de **informação** e uma referência para a **subárvore esquerda** e uma referência para **subárvore direita**
      - cada uma podendo-ser…



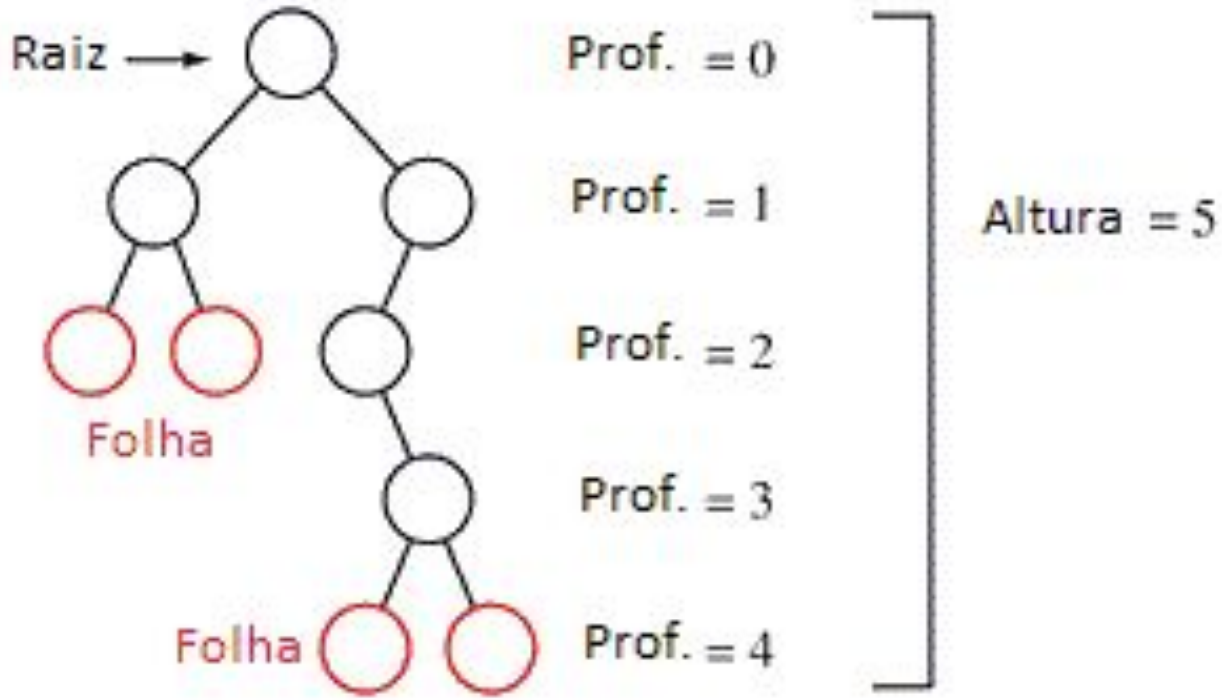
O conceito de Profundidade e Altura

- **Profundidade**
  - Os nós de uma árvore possuem profundidade
    - A profundidade do nó raiz é 0
    - A profundidade dos demais nós é igual ao número de arestas percorridas até a raiz



O conceito de Profundidade e Altura

- **Altura**
  - Todos os nós da árvore possuem uma altura:
    - Os nós folha (nós terminais) possuem altura 1
    - altura definida por quantos “níveis” a árvore tem do nó até a raiz



Ferramentas educacionais e apoio visual

- Uma boa forma de visualizar o que estamos aprendendo, é a ferramenta **Visualgo:**
  - https://visualgo.net/en/bst

Projetando uma biblioteca para ABBs

- Operações sobre a estrutura de ABB:
  - **Criação**
  - **Percurso**
  - **Inserção**
  - **Busca**
  - **Remoção**

Usaremos destas propriedades recursivas para implementar tais métodos recursivamente…

https://github.com/diogots/Curso-AlgortimoseEstruturasdeDados/tree/61f8d4503525b6e77913fd88c0242b0d2c354d1f/Bibliotecas%20completas%20(Lista%2C%20Pilha%2C%20Fila%2C%20ABB)

Projetando uma biblioteca para ABBs

- Primeiro ponto de projeto
  - Implementar o nó da árvore:

```
struct noArv{
    int info;
    struct noArv *esq;
    struct noArv *dir;
};
typedef struct noArv ArvBB;
```

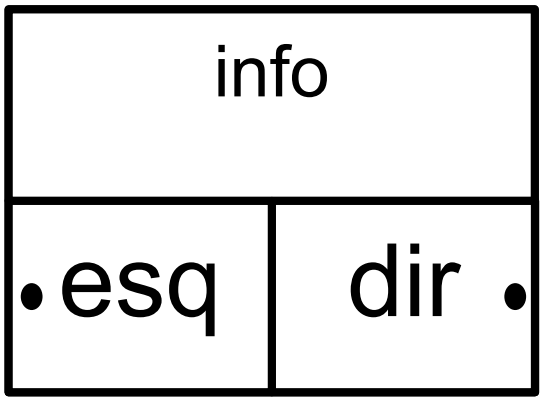
Projetando uma biblioteca para ABBs

- **Estrutura `noArv`:**
  - **`info`:** Um campo inteiro que armazena o valor associado ao nó.
    - representa a chave de busca de um nó maior de informações.
  - **`esq`:** Um ponteiro para o filho esquerdo do nó.
    - aponta para o próximo nó na subárvore esquerda do nó atual.
  - **`dir`:** Um ponteiro para o filho direito do nó.
    - aponta para o próximo nó na subárvore direita do nó atual.

Projetando uma biblioteca para ABBs

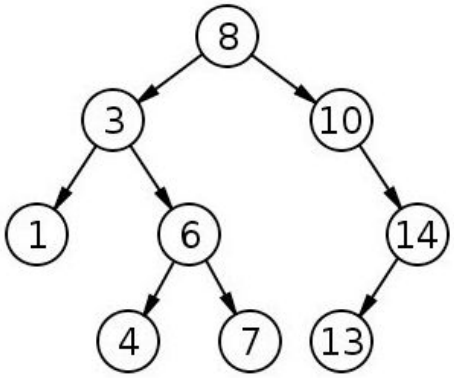
- Primeiro ponto de projeto
  - Implementar o nó da árvore:

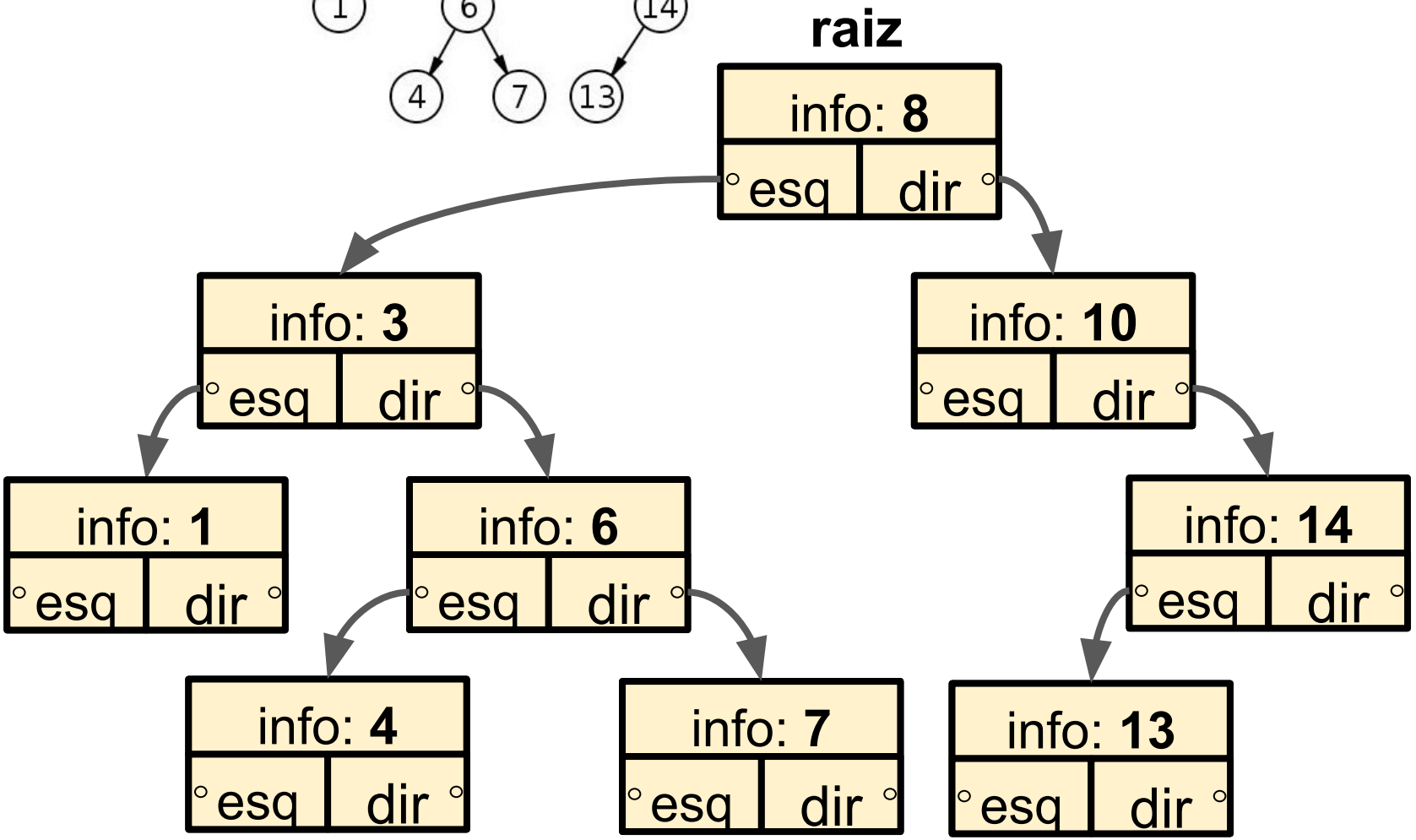
```
struct noArv{
    int info;
    struct noArv *esq;
    struct noArv *dir;
};
typedef struct noArv ArvBB;
```



Árvores Binárias de Busca (ABBs)

- **exemplo:**





Projetando uma biblioteca para ABBs

- Operações sobre a estrutura de ABB:
  - **Criação**
  - **Percurso**
  - **Inserção**
  - **Busca**
  - **Remoção**

Usaremos destas propriedades recursivas para implementar tais métodos recursivamente…

Criação de uma ABB

- Duas abordagens:
  - Criar uma estrutura nó e uma estrutura árvore que armazena um nó pra raiz
  - Assim como para listas, usar da notação recursiva para uma Árvore ser apenas um ponteiro para raiz
    - USAREMOS ESSA ABORDAGEM

Criação de uma ABB

- Função **criaABB:**
  - retorna um ponteiro NULL para um tipo ArvABB
    - ou seja, cria uma árvore vazia!

***** não esqueça de adicionar tudo no arquivo .h ***

Criação de uma ABB

```c
// Função para criar uma Árvore Binária de Busca (ABB)
ArvBB* criaABB(){
    // Retorna NULL, que representa uma árvore vazia
    return NULL;
}
```

Criação de uma ABB

- Função **vaziaABB:**
  - Testar se a árvore está vazia
  - Retorna Verdadeiro se estiver vazia ou Falso, em caso contrário

Criação de uma ABB

- Função **vaziaABB:**

```
// Função para verificar se a ABB está vazia
int vaziaABB(ArvBB* raiz){
    // Se a raiz for NULL (ou seja, a árvore está vazia), retorna 1
    // Caso contrário, retorna 0
    if(!raiz)
        return 1;
    else return 0;
}
```

Projetando uma biblioteca para ABBs

- Operações sobre a estrutura de ABB:
  - **Criação**
  - **Percurso**
  - **Inserção**
  - **Busca**
  - **Remoção**

Usaremos destas propriedades recursivas para implementar tais métodos recursivamente…

Percurso (Travessia) em árvores binárias

- ABBs não estruturas hierárquicas
  - Existem várias formas de fazer o **percurso**
    - percorrer cada um dos nós executando alguma ação
  - Comumente existem 3 percursos clássicos:
    - **Pré-ordem (RED)**
      - percurso em profundidade
    - **Em-ordem (ERD)**
      - percurso ordenado
    - **Pós-ordem (EDR)**

Conceituação preliminar

- O intuito dos percursos em EDs é visitar a estrutura e efetuar algum tipo de operação
  - é necessário criar uma função **visit()**
    - que receba um nó de informação e um
      - **ponteiro para um função**
        - tal função executará a ação desejada

Conceituação preliminar

- **Projetando a função visit()**
  - como exemplo vamos implementar uma função que imprima um nó da ABB

```
// Função para imprimir a informação no nó atual
void printNodeABB(ArvBB *nodeArv){
    // Imprime o valor do campo 'info' do nó
    cout << " " << nodeArv->info << " ";
}
```

Conceituação preliminar

- **Projetando a função visit()**
  - em seguida, a função *visitABB():*

```
// Função para visitar um nó e executar uma função nele
void visitABB(ArvBB *nodeArv, void (*func)(ArvBB*)){
    // Executa a função 'func' no nó
    func(nodeArv);
}
```

Conceituação preliminar

- **Projetando a função visit()**
  - em seguida, a função *visitABB():*

```
// Função para visitar um nó e executar uma função nele
void visitABB(ArvBB *nodeArv, void (*func)(ArvBB*)){
    // Executa a função 'func' no nó
    func(nodeArv);
}
```

Recebe uma função que possui com parâmetro de entrada um ponteiro para uma árvoreBB

Percurso (Travessia) Pré-Ordem

- **Percurso RED ou em Profundidade**
  - Visita a raiz de cada subárvore (**R**)
    - em seguida, percorre recursivamente os seus filhos da esquerda para a direita (subárvore esquerda) (**E**)
    - em seguida, percorre recursivamente os seus filhos da direita para a esquerda (subárvore direita). (**D**)

Percurso (Travessia) Em-Ordem

- **Percurso ERD ou percurso ordenado**
  - Percorre recursivamente os filhos da esquerda para a direita (subárvore esquerda) (**E**)
    - em seguida, visita a raiz de cada subárvore (**R**)
    - em seguida, percorre recursivamente os seus filhos da direita para a esquerda (subárvore direita). (**D**)

Percurso (Travessia) Pós-Ordem

- **Percurso EDR**
  - Percorre recursivamente os filhos da esquerda para a direita (subárvore esquerda) (**E**)
    - em seguida, percorre recursivamente os seus filhos da direita para a esquerda (subárvore direita). (**D**)
    - em seguida, visita a raiz de cada subárvore (**R**)

Implementação dos Percursos

- **Percurso Pré-ordem (RED)**

```
// Função para realizar uma travessia em pré-ordem da árvore (RED)
//executa a função recebida como parâmetro em cada nó
void preOrdemABB(ArvBB *raiz,void (*func)(ArvBB*)){
    // Verifica se a árvore não está vazia
    if (!vaziaABB(raiz)){
        // Visita a raiz e imprime seu valor
        visitABB(raiz,func);
        //cout << " << ";
        // Realiza a travessia em pré-ordem da subárvore esquerda
        preOrdemABB(raiz->esq,func);
        //cout << " >> ";
        // Realiza a travessia em pré-ordem da subárvore direita
        preOrdemABB(raiz->dir,func);
    }
}
```

Implementação dos Percursos

- **Percurso Em-ordem (ERD)**

```
// Função para realizar uma travessia em ordem da árvore (ERD)
void emOrdemABB(ArvBB *raiz,void (*func)(ArvBB*)){
    // Verifica se a árvore não está vazia
    if (!vaziaABB(raiz)){
        // Realiza a travessia em ordem da subárvore esquerda
        //cout << " << ";
        emOrdemABB(raiz->esq,func);
        // Visita a raiz e imprime seu valor
        visitABB(raiz,func);
        //cout << " >> ";
        // Realiza a travessia em ordem da subárvore direita
        emOrdemABB(raiz->dir,func);
    }
}
```

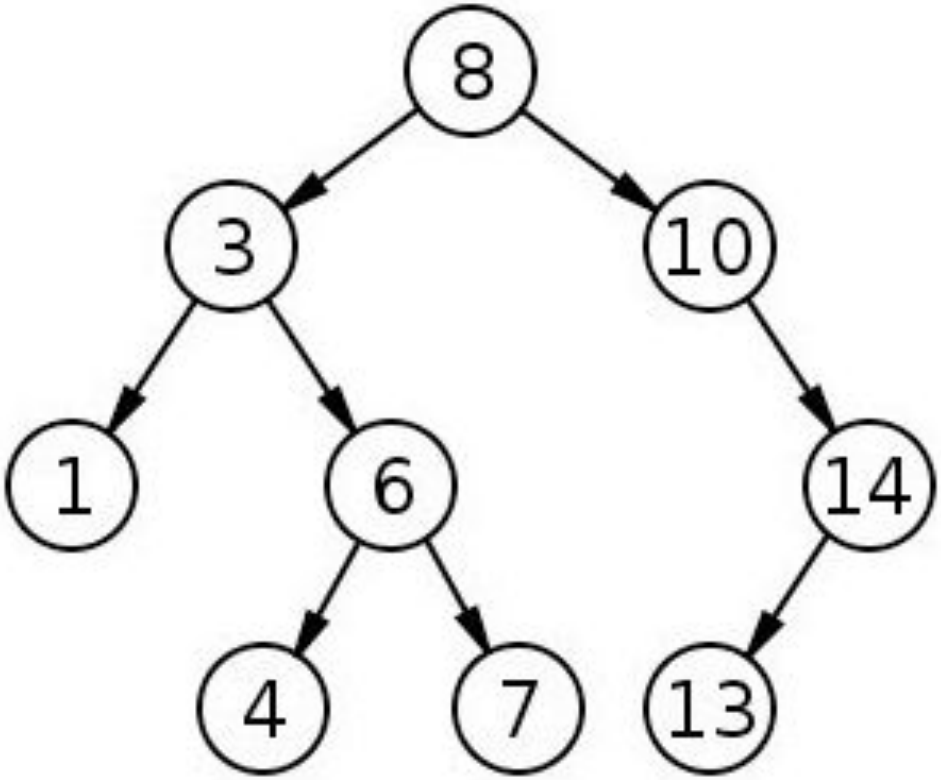
Implementação dos Percursos

- **Percurso Pós-Ordem (EDR)**

```
// Função para realizar uma travessia em pós-ordem da árvore (EDR)
void posOrdemABB(ArvBB *raiz,void (*func)(ArvBB*)){
    // Verifica se a árvore não está vazia
    if (!vaziaABB(raiz)){
        // Realiza a travessia em pós-ordem da subárvore esquerda
        //cout << " << ";
        posOrdemABB(raiz->esq,func);
        //cout << " >> ";
        // Realiza a travessia em pós-ordem da subárvore direita
        posOrdemABB(raiz->dir,func);
        // Visita a raiz e imprime seu valor
        visitABB(raiz,func);
    }
}
```

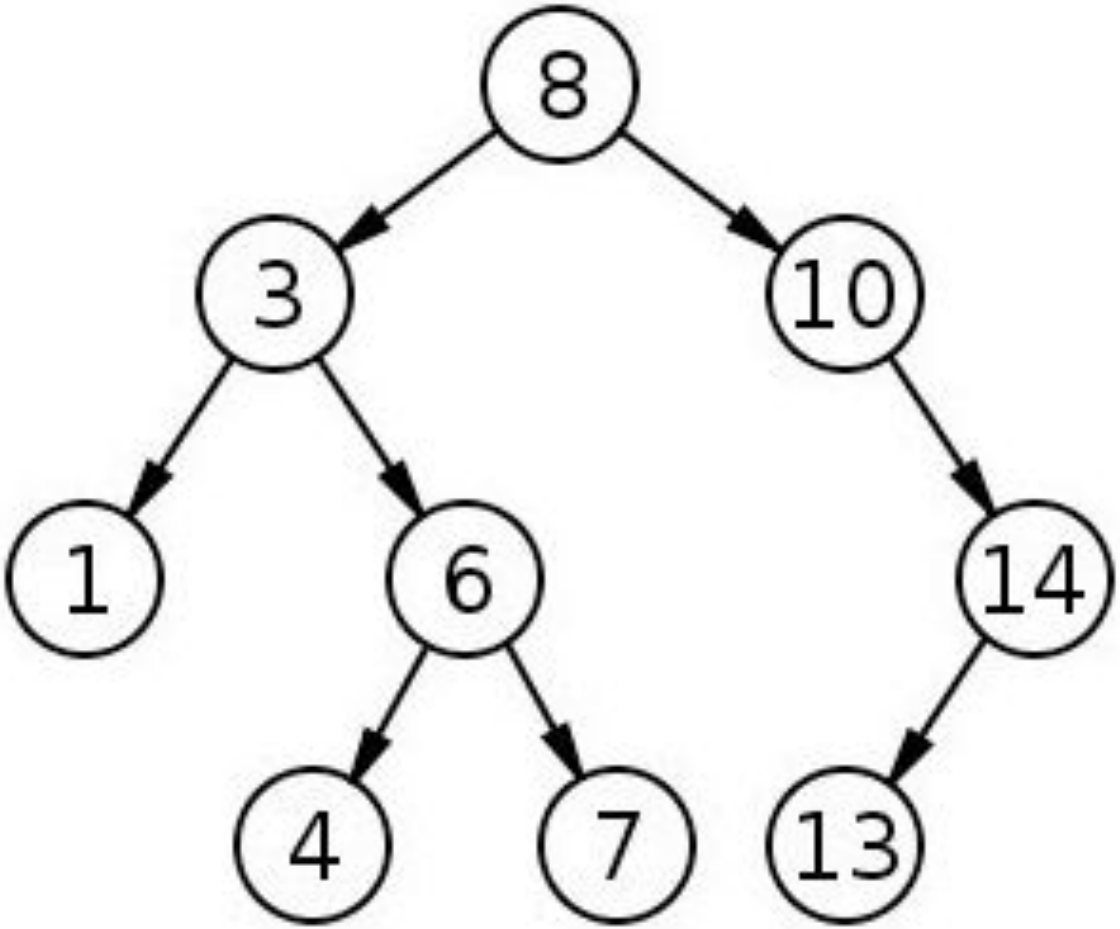
Implementação dos Percursos

- **Exemplo, dado que eu queira percorrer e imprimir a seguinte árvore:**




```
ArvBB *arv;
arv = criaABB();
arv = insereABB(arv, 8);
arv = insereABB(arv, 3);
arv = insereABB(arv, 10);
arv = insereABB(arv, 1);
arv = insereABB(arv, 6);
arv = insereABB(arv, 4);
arv = insereABB(arv, 7);
arv = insereABB(arv, 14);
arv = insereABB(arv, 13);
preOrdemABB(arv,printNodeABB);
emOrdemABB(arv,printNodeABB);
posOrdemABB(arv,printNodeABB);
```

Implementação dos Percursos




```
Percurso RED:  8  3  1  6  4  7  10  14  13
Percurso ERD:  1  3  4  6  7  8  10  13  14
Percurso EDR:  1  4  7  6  3  13  14  10  8
```

Implementação dos Percursos

- **Usando a estratégia Pré-ordem para calcular a altura da árvore:**

```c
// Função para retornar o máximo entre dois inteiros
int max(int a, int b){
    // Se 'a' for maior que 'b', retorna 'a'
    // Caso contrário, retorna 'b'
    if(a>b)
        return a;
    else return b;
}

// Função para calcular a altura da Árvore Binária de Busca (ABB)
int alturaABB(ArvBB *raiz){
    // Se a árvore estiver vazia, retorna -1
    if(vaziaABB(raiz))
        return -1;
    // Caso contrário, retorna 1 mais o máximo entre as alturas das subárvores esquerda e direita
    else return 1 + max(alturaABB(raiz->esq),alturaABB(raiz->dir));
}
```

Implementação dos Percursos

- **TAREFA!**
  - **Analizar na implementação de ABB disponibilizada, os seguintes métodos:**
    - copie o trecho de código e coloque em um chat de IA. Peça para analisar e explicar
      - **preOrdemHierarquico**
        - usa percurso pré-ordem para imprimir a árvore em hierarquia
      - **getNumElementosABB**
        - usa percurso pré-ordem para percorrer e contar o número de elementos da ABB usando o método visit()

Projetando uma biblioteca para ABBs

- Operações sobre a estrutura de ABB:
  - **Criação**
  - **Percurso**
  - **Inserção**
  - **Busca**
  - **Remoção**

Usaremos destas propriedades recursivas para implementar tais métodos recursivamente…

Inserção em ABBs

- A inserção em ABBs deve ocorrer sempre em NÓS FOLHAS!
- Deve-se percorrer da raiz até as extremidades para inserir o novo elemento em um nó-folha de modo a respeitar a propriedade de ordenação.

Inserção em ABBs

- Algoritmo **Inserção**:
  - **Se** a árvore estiver vazia
    - adicionar novo nó
  - **Senão**
    - **Se** o novo elemento é menor que a raiz:
      - percorra a subárvore esquerda fazendo comparações até encontrar a posição correta.
    - **Senão Se** o novo elemento é maior que a raiz:
      - percorra a subárvore direita fazendo comparações até encontrar a posição correta.
    - **Senão**
      - Tratar repetições (ou não).

Inserção em ABBs

- Implementação:
  - Função que criar um novo nó de árvore:

```
// Função para criar um novo nó na Árvore Binária de Busca (ABB)
ArvBB* novoNoABB(int elem){
    // Cria um novo nó
    ArvBB* novo = new ArvBB;
    // Atribui o valor do elemento ao campo 'info' do novo nó
    novo->info = elem;
    // Inicializa os ponteiros 'esq' e 'dir' do novo nó como NULL
    novo->esq = NULL;
    novo->dir = NULL;
    // Retorna o novo nó
    return novo;
}
```

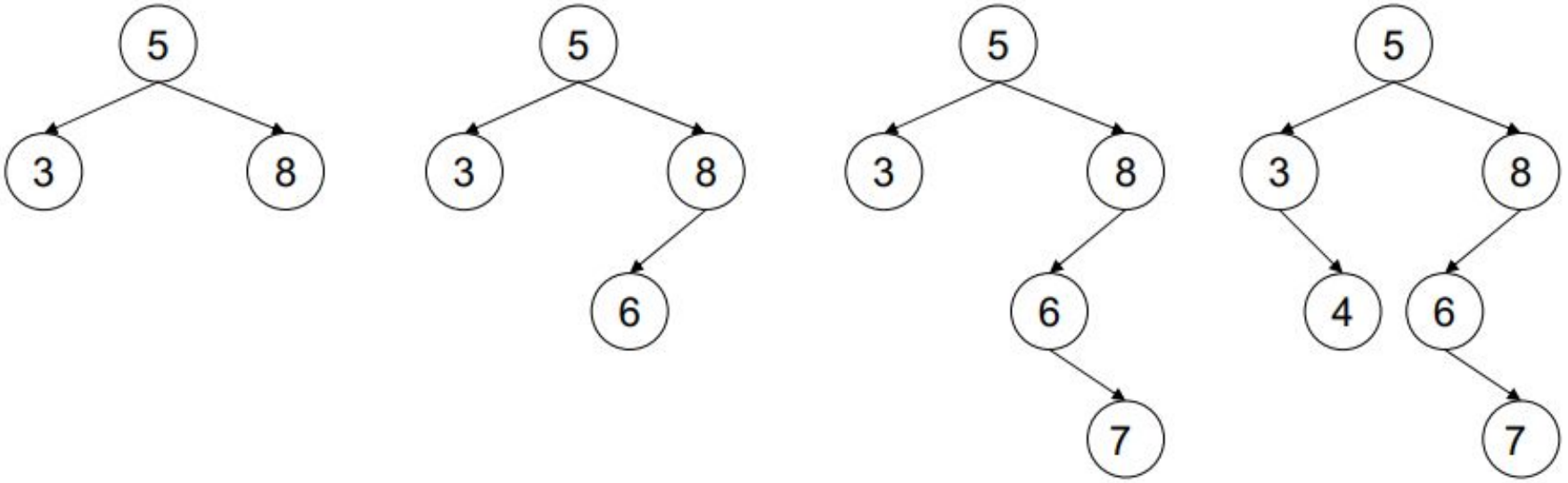
Inserção em ABBs

- Implementação:
  - Função de inserção recursiva:

```c
// Função para inserir um elemento na ABB
ArvBB* insereABB(ArvBB *raiz, int elem){
    // Verifica se a ABB está vazia
    if(vaziaABB(raiz)){
        // Se a ABB estiver vazia, cria um novo nó com o elemento
        ArvBB* novo = novoNoABB(elem);
        // A raiz se torna o novo nó
        raiz = novo;
    }else{
        // Se o elemento for menor que o valor da raiz
        if(elem < raiz->info)
            // Insere o elemento na subárvore esquerda
            raiz->esq = insereABB(raiz->esq, elem);
        else
            // Senão, insere o elemento na subárvore direita
            raiz->dir = insereABB(raiz->dir, elem);
    }
    // Retorna a raiz da ABB
    return raiz;
}
```

Inserção em ABBs

- Algoritmo **Inserção**:
  - exemplo:



Projetando uma biblioteca para ABBs

- Operações sobre a estrutura de ABB:
  - **Criação**
  - **Percurso**
  - **Inserção**
  - **Busca**
  - **Remoção**

Usaremos destas propriedades recursivas para implementar tais métodos recursivamente…

Busca em ABBs

- Implementação de busca binária
- Basta comparar o valor buscado a partir do nó raiz e seguir recursivamente, respeitando as propriedades de ABB.

Busca em ABBs

- **Algoritmo Busca:**
  - **Se** árvore não vazia:
    - **Se** conteúdo da raiz igual ao elemento:
      - Achou
    - **Senão Se** o valor for menor que o elemento da raiz
      - seguir pelo ramo da esquerda, fazendo as comparações
    - **Senão Se** o valor for maior que o elemento da raiz
      - seguir pelo ramo da direita, fazendo as comparações
  - **Senão**
    - elemento não está presente na árvore

Busca em ABBs

- **Implementação do algoritmo de busca:**

```
// Função para buscar um elemento na Árvore Binária de Busca (ABB)
ArvBB* buscaElemABB(ArvBB *raiz, int elem){
    // Verifica se a árvore não está vazia
    if(!vaziaABB(raiz)){
        // Se o valor do nó atual é igual ao elemento procurado
        if (raiz->info == elem){
            cout << "Elemento " << elem << " encontrado!" << endl;
            // Retorna o nó que contém o elemento
            return raiz;
        }
        // Se o elemento é menor que o valor do nó atual
        else if(elem < raiz->info)
            // Busca o elemento na subárvore esquerda
            raiz = buscaElemABB(raiz->esq,elem);
        else
            // Se o elemento é maior que o valor do nó atual, busca o elemento na subárvore direita
            buscaElemABB(raiz->dir,elem);
    }else{ // Se a árvore está vazia, imprime uma mensagem indicando que o elemento não foi encontrado
        cout << "Elemento " << elem << " não encontrado!" << endl;
        return raiz;
    }
}
```

Projetando uma biblioteca para ABBs

- Operações sobre a estrutura de ABB:
  - **Criação**
  - **Percurso**
  - **Inserção**
  - **Busca**
  - **Remoção**

Usaremos destas propriedades recursivas para implementar tais métodos recursivamente…

Remoção de elemento em ABBs

- Remover um elemento da árvore consiste em analisar se a remoção desestrutura as propriedades de ABB
  - se necessário fazer os ajustes para manter a árvore correta

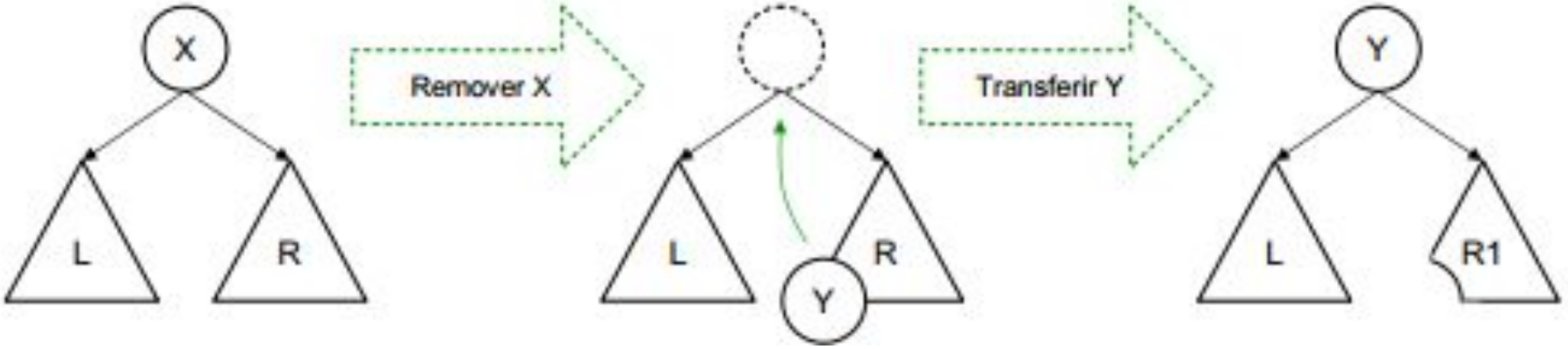
Remoção de elemento em ABBs

- Para remover um dado elemento em uma ABB existem as seguintes situações possíveis:
  - Não existe nenhum nó com chave igual ao elemento
  - O nó é folha
  - O nó possui apenas uma subárvore
  - O nó possui as duas subárvores

Remoção de elemento em ABBs

- Quando o elemento a ser removido possui as duas sub-árvore não vazias
  - Geralmente:
  - Encontrar o elemento de **menor** valor na sub-árvore **direita** do elemento a ser removido
    - Transferi-lo para o nó ocupado pelo elemento removido

Remoção de elemento em ABBs



Remoção de elemento em ABBs

- Implementação da remoção:

```
// Função para remover um elemento de uma Árvore Binária de Busca (ABB)
ArvBB* removeElemABB(ArvBB * raiz, int elem){
    // Verifica se a árvore está vazia
    if(!vaziaABB(raiz)){
        // Se o elemento é menor que o nó atual, procura na subárvore esquerda
        if(elem < raiz->info){
            raiz->esq = removeElemABB(raiz->esq,elem);
        }
        // Se o elemento é maior que o nó atual, procura na subárvore direita
        else if (elem > raiz->info){
            raiz->dir = removeElemABB(raiz->dir,elem);
        }
        // Se o elemento é igual ao nó atual, remove o nó
        else{
```

....

Remoção de elemento em ABBs

- Implementação da remoção:

```
else{
    ArvBB *aux = raiz;
    // Se o nó atual não tem filho à esquerda, substitui o nó pelo filho à direita
    if(raiz->esq == NULL){
        raiz = raiz->dir;
        delete aux;
    }
    // Se o nó atual não tem filho à direita, substitui o nó pelo filho à esquerda
    else if (raiz->dir == NULL){
        raiz = raiz->esq;
        delete aux;
    }
    // Se o nó atual tem ambos os filhos, substitui o nó pelo menor elemento da subárvore direita
    else{
        aux = raiz->dir;
        while(aux->esq)
            aux = aux->esq;
        raiz->info = aux->info;
        raiz->dir = removeElemABB(raiz->dir,aux->info);
    }
}
```

....

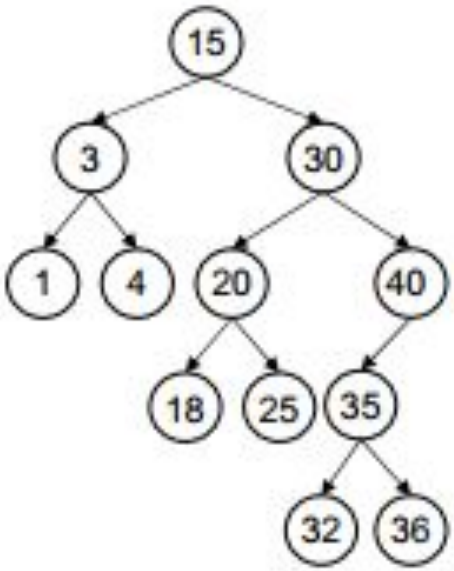
Remoção de elemento em ABBs

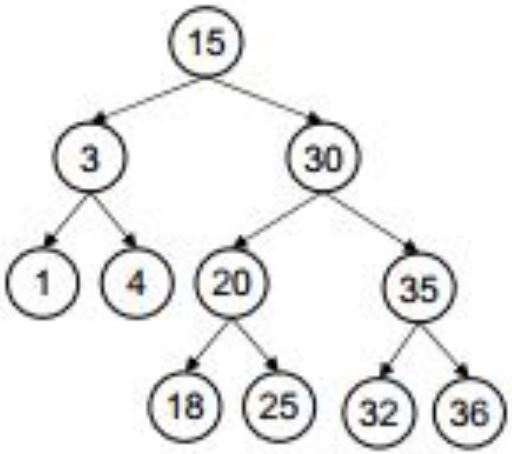
- Implementação da remoção:

```
        }
        // Retorna a raiz da árvore
        return raiz;
    }
    // Se a árvore está vazia, retorna NULL
    else return NULL;
}
```

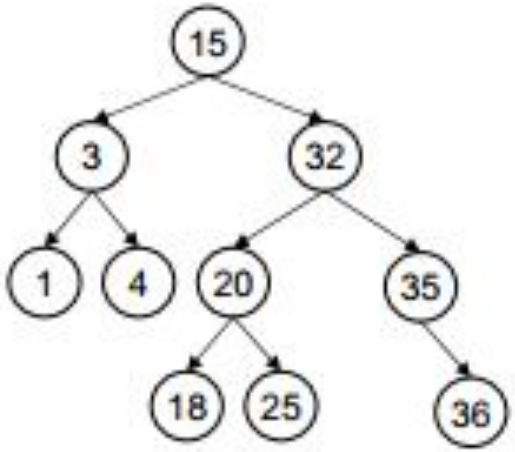
....

Remoção de elemento em ABBs



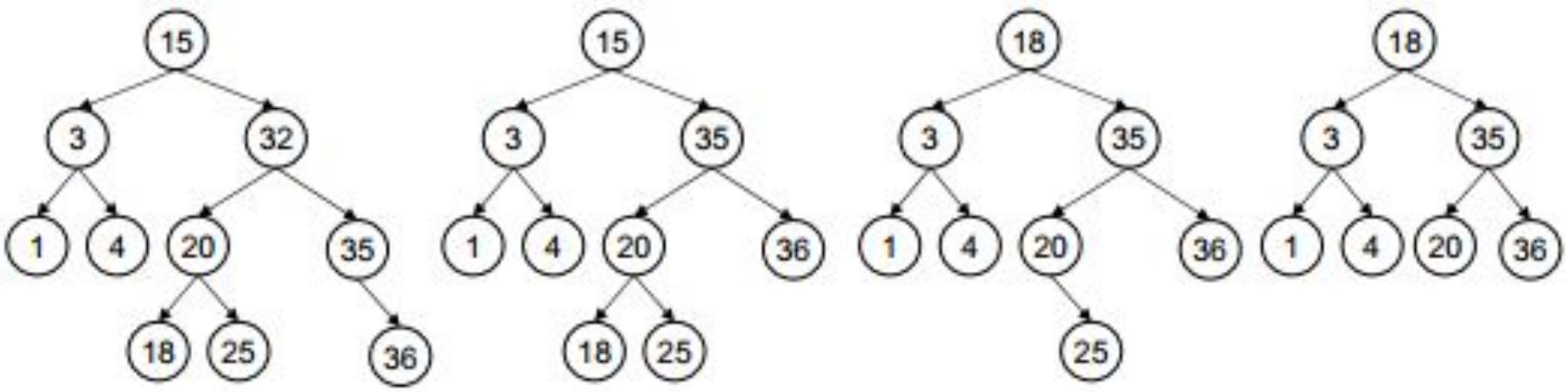


x = 40



x = 30

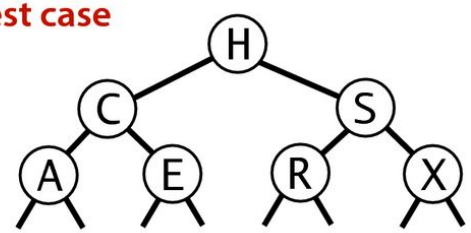
Remoção de elemento em ABBs



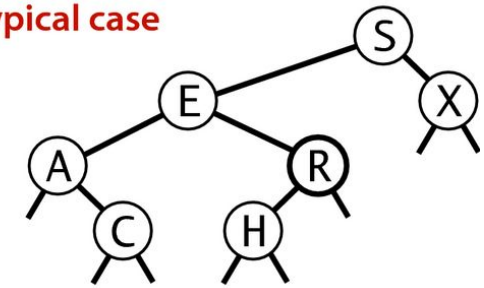
x = 32 x = 15 x = 25

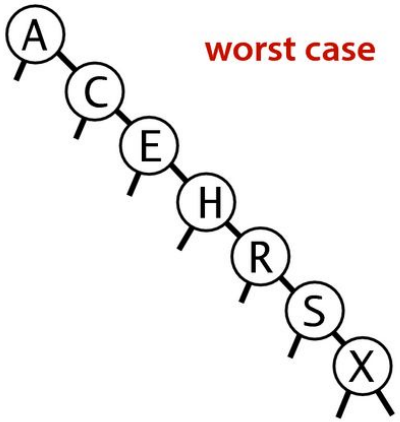
O problema do balanceamento

best case



typical case





worst case

BST possibilities

O problema do balanceamento

O problema do balanceamento

- https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/st-bst.html
- https://algs4.cs.princeton.edu/32bst/#preview1