

Árvores: árvores binárias

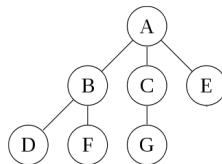
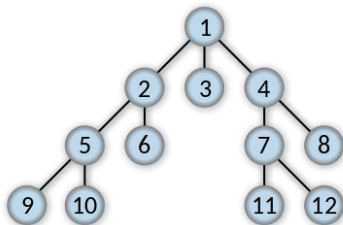
Prof. Jefferson T. Oliva

Algoritmos e Estrutura de Dados II (AE23CP)
Engenharia de Computação
Departamento Acadêmico de Informática (Dainf)
Universidade Tecnológica Federal do Paraná (UTFPR)
Campus Pato Branco

- Árvores Binárias
 - Representação
- Árvores Binárias de Busca
 - Busca
 - Inserção
 - Remoção
 - Percurso

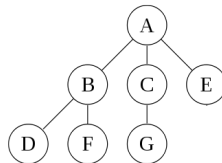
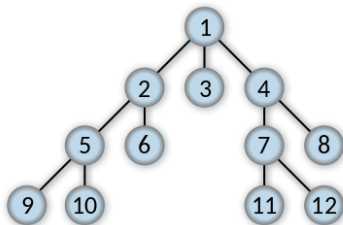
- Listas lineares, filas e pilhas não são adequadas para representar dados que devem ser organizados de forma hierárquica
- Árvore é uma estrutura de dados muito eficiente para armazenamento de informação
- Árvores são estruturas de dados não lineares
- Aplicações de árvores
 - Sistemas de arquivos
 - Compiladores
 - Expressões algébricas
 - Páginas Web

- Árvore
 - Conjunto finito de nós
 - Existe um nó raiz r com zero ou mais sub-árvores
 - Os nós internos são filhos de r
 - Cada sub-árvore também possui um nó raiz, que é descendente (nó filho) de r
 - Nós folhas são os nós que não possuem filhos



- Árvore

- Altura: distância entre o nó raiz e o nó folha mais profundo
 - Se a árvore tem apenas um nó, a sua altura é 0
- Profundidade: distância entre um nó e a raiz
- Balanceamento: uma árvore é balanceada se cada uma das subárvores de cada nó possuem, aproximadamente, a mesma altura

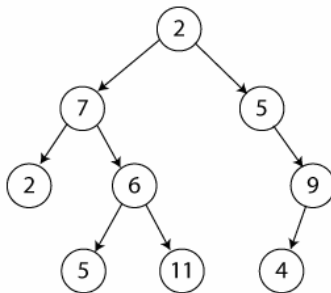


- Exemplos de tipos de árvores
 - Árvore binária
 - Árvore binária de busca
 - AVL
 - Árvore vermelha-preta (rubro-negra)
 - Árvore B

Árvores Binárias

Árvores Binárias

- Árvore em que cada nó contém um ou dois filhos
- As sub-árvores também contêm entre um e dois nós, exceto se são nós-folhas



- Propriedades de árvores binárias
 - Uma árvore binária com n elementos tem $n - 1$ ramos
 - Uma árvore binária de altura h tem no mínimo h elementos e no máximo $\sum_{i=0}^h 2^i = 2^{h+1} - 1$
 - Uma árvore binária de altura h com $2^{h+1} - 1$ elementos é denominada árvore cheia
 - A altura de uma árvore binária com n elementos ($n > 0$) é no máximo $n - 1$ e no mínimo $\log n - 1$
 - Em uma árvore binária cheia (cada nó, exceto folha, possui dois descendentes), a quantidade total de folhas é 2^h

Árvores Binárias

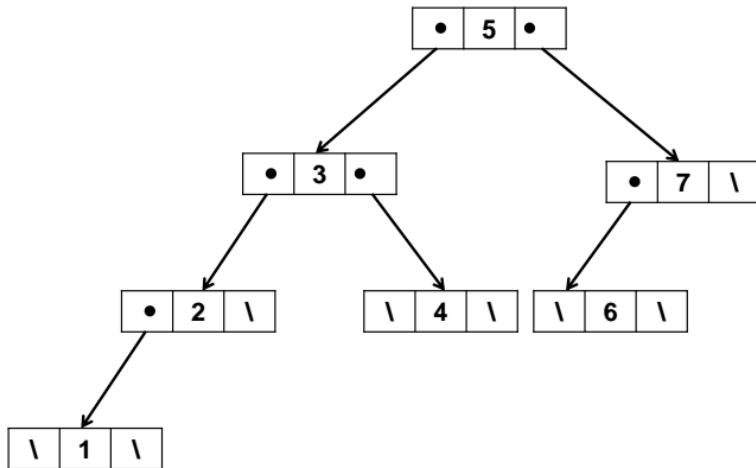
Representação

- Podemos representar uma árvore por meio de um registro (*struct*) de três campos:
 - Informação
 - Sub-árvore esquerda
 - Sub-árvore direita

left	item	right
esquerda	informação	direita

Árvores Binárias

Representação



- Na figura acima, \ representa NULL

- Estrutura de dados para a representação de uma árvore binária:

```
typedef struct Node Node;  
  
struct Node{  
    int item;  
    Node *right;  
    Node *left;  
};
```

- Primeiras operações com árvore binária

```
// criar nó
Node* create(int key){
    Node* no = (Node*) malloc(sizeof(Node));
    no->key = key;
    no->left = NULL;
    no->right = NULL;

    return no;
}
```

- Primeiras operações com árvore binária

```
// liberar nó
int release(Node* no) {
    if (no != NULL) {
        free(no);

        return 1;
    }

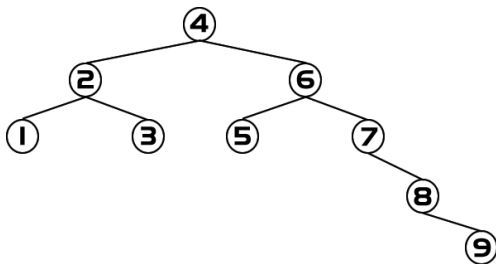
    return 0;
}
```

- Exercício: implemente um algoritmo para liberar (nó por nó) uma árvore binária.

Árvores Binárias de Busca

Árvores Binárias de Busca

- Árvore binária de pesquisa
- Árvores binárias que exibem propriedades de ordenação



Árvores Binárias de Busca

- Uma árvore binária de busca satisfaz as seguintes propriedades:
 - 1 Todo elemento tem uma chave
 - 2 As chaves (se houver) na sub-árvore esquerda são menores do que a chave na raiz
 - 3 As chaves (se houver) na sub-árvore direita são maiores do que a chave na raiz
 - 4 As sub-árvores esquerda e direita são também árvores binárias de pesquisa

Árvores Binárias de Busca

- Principais operações em uma árvore binária de busca:
 - Pesquisa
 - Inserção
 - Remoção

Árvores Binárias de Busca

Pesquisa

- O objetivo é encontrar o nó que contém o item com o mesmo valor procurado
- Lembram da busca binária?
- A busca é iniciada pela raiz da árvore
 - Caso o valor seja encontrado, o nó é retornado
 - Caso contrário:
 - É realizada uma chamada recursiva para a sub-árvore esquerda se o valor procurado é menor que o item verificado
 - É realizada uma chamada recursiva para a sub-árvore direita se o valor procurado é maior

Árvores Binárias de Busca

Pesquisa

```
Node* search(Node* tree, int value){
    if (tree != NULL)
        if (tree->item == value)
            return tree;
        else if (value < tree->item)
            return search(tree->left, value);
        else
            return search(tree->right, value);
    else
        return NULL;
}
```

- Como encontrar a menor ou a maior chave em uma árvore binária de busca?

- Como encontrar a menor ou a maior chave nem uma árvore binária de busca?
Resposta: explorar as sub-árvores esquerda (menor) ou direita (maior)

- Exercício: implemente a versão iterativa do algoritmo *search*.

- Eficiência
 - Busca por um determinado valor (também o mínimo ou o máximo)
 - Melhor caso: $O(1)$
 - Caso médio: $O(\log n)$
 - Pior caso: $O(n)$

Árvore Binária de Busca

Inserção

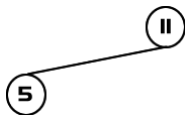
- Percorre a estrutura até chegar a um determinado ponteiro cujo nó filho (direito ou esquerdo) seja *NULO*
- Exemplo: geração de uma árvore binária de busca a partir da sequência 11, 5, 14, 12, 13, 1, 6 e 16
 - Criação da árvore com o item 11



Árvore Binária de Busca

Inserção

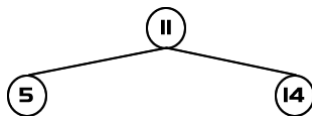
- Inserção do item 5



Árvore Binária de Busca

Inserção

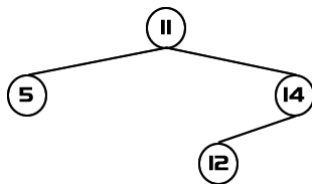
- Inserção do item 14



Árvore Binária de Busca

Inserção

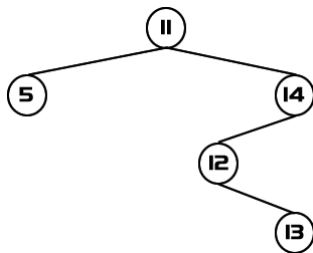
- Inserção do item 12



Árvore Binária de Busca

Inserção

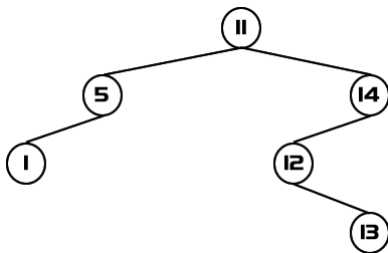
- Inserção do item 13



Árvore Binária de Busca

Inserção

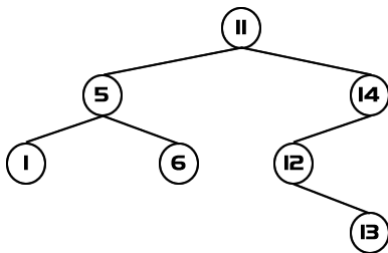
- Inserção do item 1



Árvore Binária de Busca

Inserção

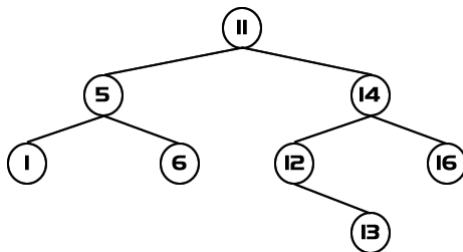
- Inserção do item 6



Árvore Binária de Busca

Inserção

- Inserção do item 16



Árvore Binária de Busca

Inserção

```
Node* insert(Node* tree, int value){
    if (tree == NULL)
        tree = create(value);
    else if (value < tree->item)
        tree->left = insert(tree->left, value);
    else
        tree->right = insert(tree->right, value);

    return tree;
}
```

Árvore Binária de Busca

Inserção

- Eficiência
 - Melhor caso: $O(1)$
 - Caso médio: $O(\log n)$
 - Pior caso: $O(n)$

Árvore Binária de Busca

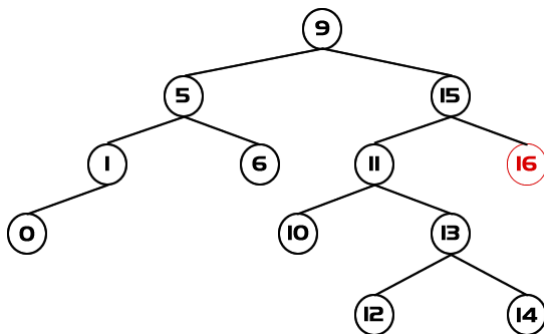
Remoção

- Três casos básicos devem ser considerados para a remoção de um nó z em uma árvore binária de busca:
 - ① Caso z não tenha filhos, remova-o de modo que o nó pai substitua z por *NULO*
 - ② Caso o nó tenha apenas um filho, o mesmo deve substituir o nó z
 - ③ Caso z possua dois filhos, utilizar o nó y que contenha o menor valor da sub-árvore direita para substituir z
 - Ou, o nó y que contenha o maior valor da sub-árvore esquerda para substituir z

Árvore Binária de Busca

Remoção

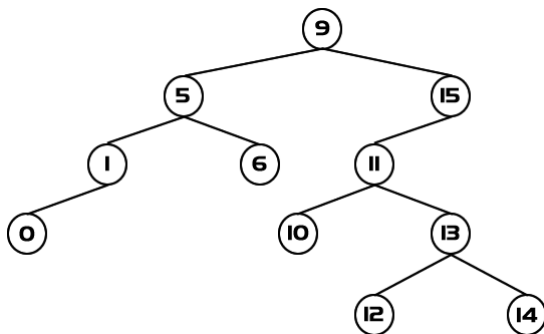
- Exemplo: remoção do item 16 da árvore



Árvore Binária de Busca

Remoção

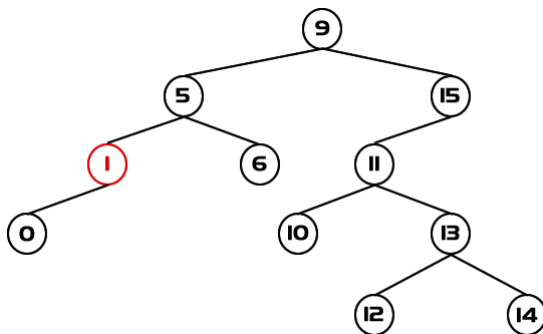
- Exemplo: remoção do item 16 da árvore



Árvore Binária de Busca

Remoção

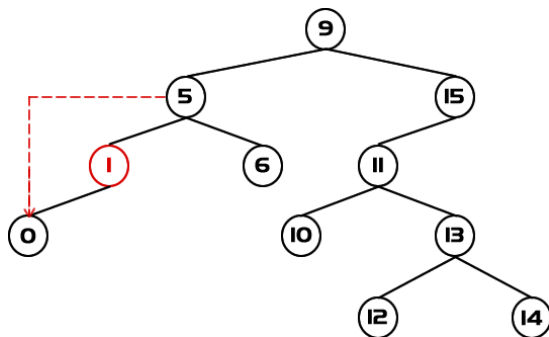
- Exemplo: remoção do item 1 da árvore



Árvore Binária de Busca

Remoção

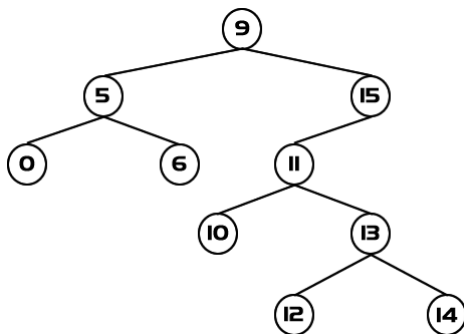
- Exemplo: remoção do item 1 da árvore



Árvore Binária de Busca

Remoção

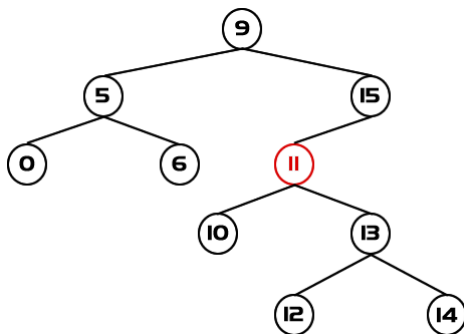
- Exemplo: remoção do item 1 da árvore



Árvore Binária de Busca

Remoção

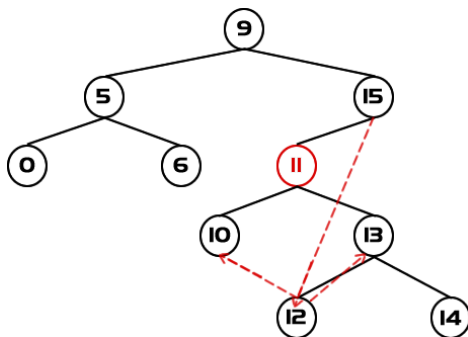
- Exemplo: remoção do item 11 da árvore



Árvore Binária de Busca

Remoção

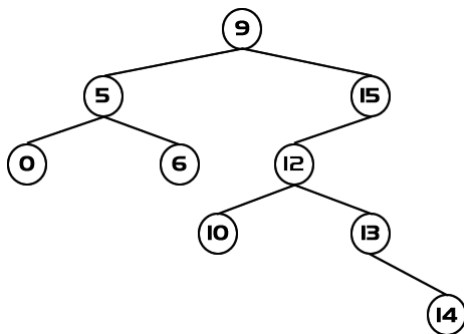
- Exemplo: remoção do item 11 da árvore



Árvore Binária de Busca

Remoção

- Exemplo: remoção do item 11 da árvore



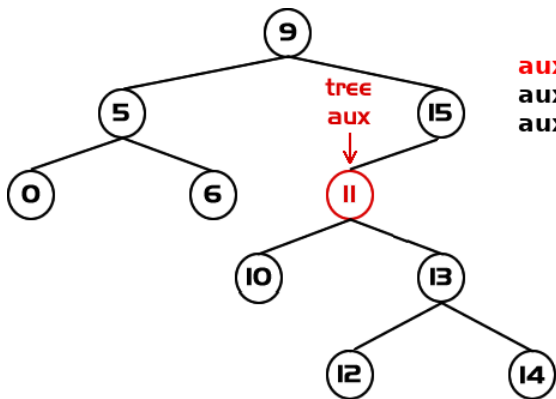
Árvore Binária de Busca

Remoção

```
int delete(Node* tree, int value){
    Node *aux, *auxP, auxF;
    if (tree != NULL){
        if (value < tree->item)
            delete(tree->left, value);
        else if (value > tree->item)
            delete(tree->right, value);
        else{
            aux = tree;
            if (aux->left == NULL) // Caso aux tenha apenas um ou nenhum descendente
                tree = tree->right;
            else if (aux->right == NULL) // aux tem apenas um descendente (esquerdo)
                tree = tree->left;
            else{ // o nó a ser excluído possui dois descendentes
                auxP = aux->right; // raiz da sub-árvore esquerda de aux
                auxF = auxP; // auxiliar para procurar o menor descendente de auxP
                /*Procurar o menor descendente da sub-árvore direita de aux*/
                while (auxF->left != NULL){
                    auxP = auxF; // A partir daqui, auxP é o nó pai de auxF
                    auxF = auxF->left;
                }
                auxP->left = auxF->right; // o nó procurado pode ter um descendente
                auxF->left = aux->left; // primeira parte da substituição de aux
                auxF->right = aux->right; // segunda parte da substituição de aux
                tree = auxF; // atualização do nó raiz
            }
            free(aux); // liberação segura do nó procurado para a exclusão
            return 1; // exclusão bem sucedida
        }
    }
    return 0; // o nó com o valor procurado não foi encontrado
}
```

Árvore Binária de Busca

Remoção



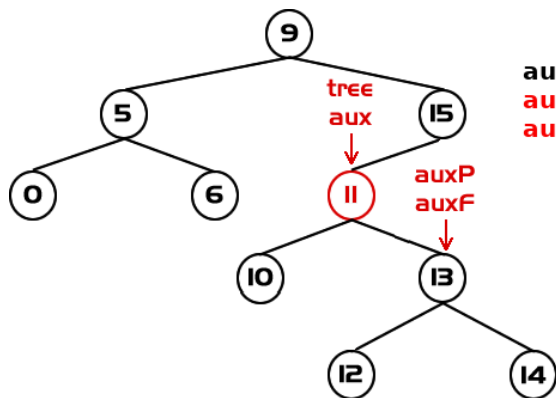
aux = tree

auxP = aux->right

auxF = auxP

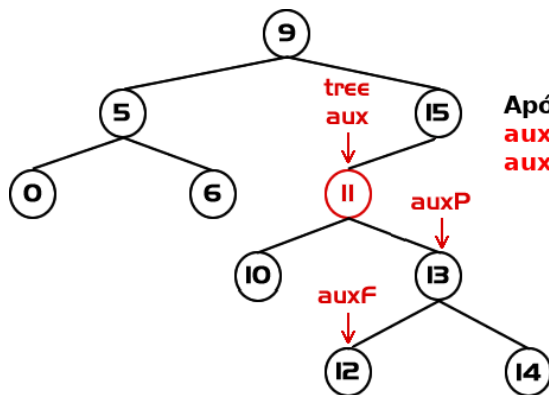
Árvore Binária de Busca

Remoção



Árvore Binária de Busca

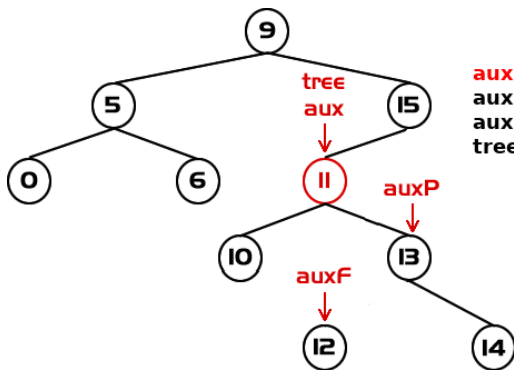
Remoção



...
Após o laço while
auxP = auxF
auxF = auxF->left

Árvore Binária de Busca

Remoção

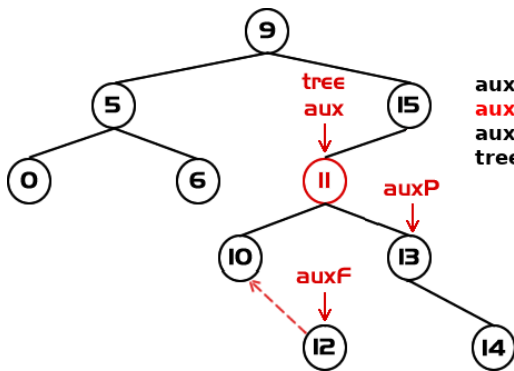


...

auxP->left = auxF->right
auxF->left = aux->left
auxF->right = aux->right
tree = auxF

Árvore Binária de Busca

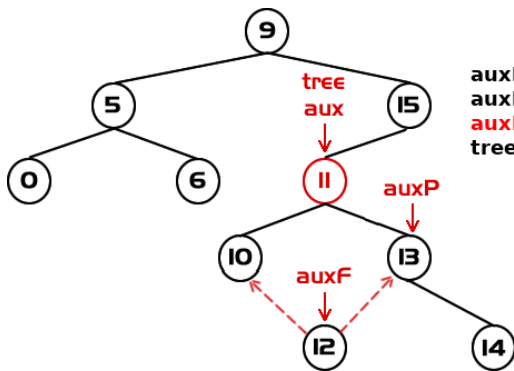
Remoção



...
auxP->left = auxF->right
auxF->left = aux->left
auxF->right = aux->right
tree = auxF

Árvore Binária de Busca

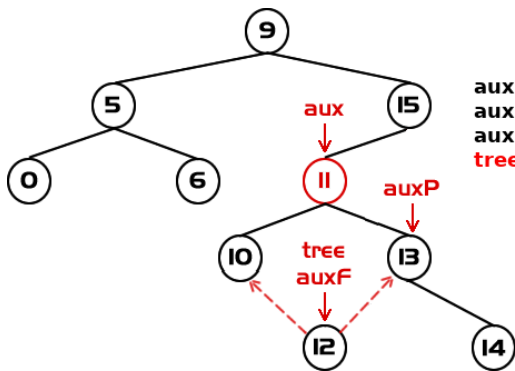
Remoção



...
auxP->left = auxF->right
auxF->left = aux->left
auxF->right = aux->right
tree = auxF

Árvore Binária de Busca

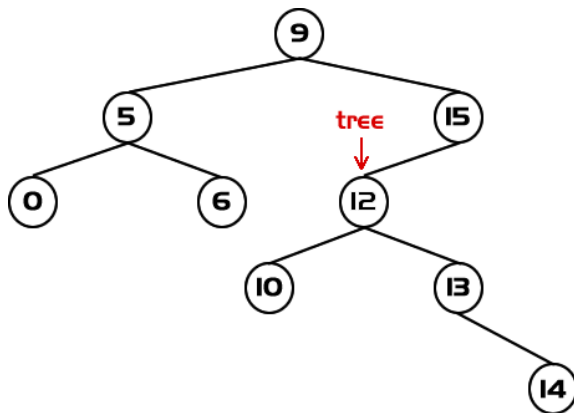
Remoção



...
auxP->left = auxF->right
auxF->left = aux->left
auxF->right = aux->right
tree = auxF

Árvore Binária de Busca

Remoção



desalocar(aux)

- Exercício: com base no exemplo anterior, remova, em sequência, os seguintes itens da árvore do último slide.
 - 10
 - 12

Árvore Binária de Busca

Remoção

- Eficiência
 - Melhor caso: $O(1)$
 - Caso médio: $O(\log n)$
 - Pior caso: $O(n)$

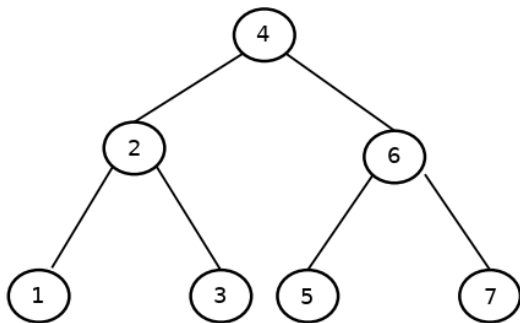
- Pré-ordem
 - A raiz é visitada primeiramente
 - Após, as sub-árvores da direita à esquerda são processadas em pré-ordem

```
void prefix(Node* tree){  
    if (tree != NULL){  
        printf("%d", tree->item);  
        prefix(tree->left);  
        prefix(tree->right);  
    }  
}
```


Árvore Binária de Busca

Percurso

- Pré-ordem



- Percurso: 4, 2, 1, 3, 6, 5, 7

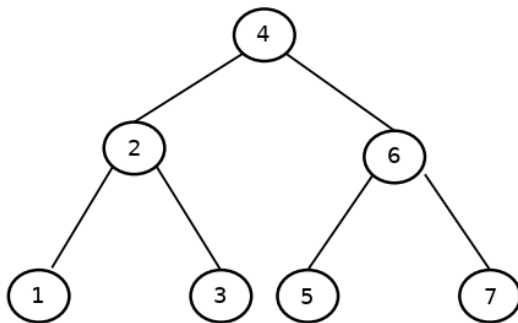
- Ordem simétrica
 - Primeiramente, a sub-árvore esquerda é percorrida em ordem simétrica
 - Após, a raiz é visitada
 - Por último, a sub-árvore direita é percorrida em ordem simétrica

```
void infix(Node* tree){  
    if (tree != NULL){  
        infix(tree->left);  
        printf("%d", tree->item);  
        infix(tree->right);  
    }  
}
```

Árvore Binária de Busca

Percurso

- Ordem simétrica



- Percurso: 1, 2, 3, 4, 5, 6, 7

- Ordem simétrica
 - Em árvores binárias, a raiz é visitada entre as duas subárvores
 - Em árvores n -nárias ($n > 2$), A subárvore da esquerda é percorrida em ordem simétrica, depois a raiz é visitada e depois as outras subárvores são visitadas da esquerda para a direita, sempre em ordem simétrica

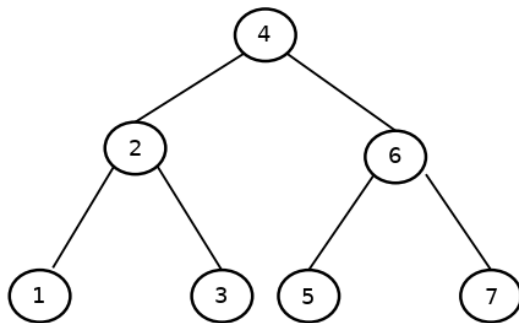
- Pós-ordem
 - A raiz é a última a ser visitada
 - Todas as subárvores da direita até a esquerda são percorridas em pós-ordem

```
void posfix(Node* tree){  
    if (tree != NULL){  
        posfix(tree->left);  
        posfix(tree->right);  
        printf("%d", tree->item);  
    }  
}
```

Árvore Binária de Busca

Percurso

- Pós-ordem

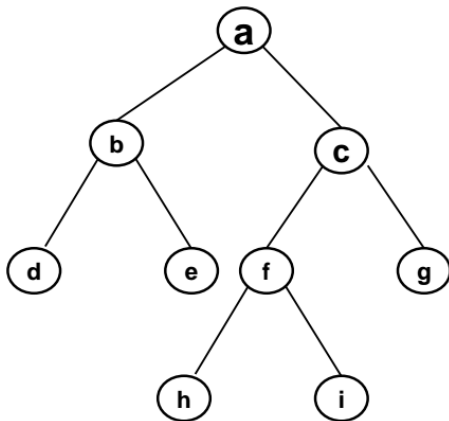






- Percurso: 1, 3, 2, 5, 7, 6, 4

Árvore Binária de Busca

Percurso

Exercício: para a árvore abaixo, faça os três tipos de percurso



-  Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Clifford, S.
Algoritmos: teoria e prática.
Elsevier, 2012.
-  Pereira, S. L.
Estrutura de Dados e em C: uma abordagem didática.
Saraiva, 2016.
-  Horowitz, E., Sahni, S. Rajasekaran, S.
Computer Algorithms.
Computer Science Press, 1998.
-  Szwarcfiter, J.; Markenzon, L.
Estruturas de Dados e Seus Algoritmos.
LTC, 2010.



Ziviani, N.

Projeto de Algoritmos - com implementações em Java e C++.

Thomson, 2007.