

Notas de Aula - AED2 – Árvores: árvores binárias
Prof. Jefferson T. Oliva

Em AED1, vocês viram estruturas de dados lineares, como listas, filas e pilhas, as quais não são adequadas para representar dados que devem ser organizados de forma hierárquica.

A partir de hoje, veremos o último tema para a primeira prova: árvores, que são estruturas cujas informações são organizadas de forma hierárquica.

Árvores são estruturas bastante eficientes para o armazenamento de informações e possuem diversas aplicações, como: sistema de arquivos, processamento de língua natural, expressões algébricas, páginas web, compactação de dados, aprendizado de máquina, *clustering*, etc.

Formalmente, podemos dizer que uma árvore é um conjunto finito de nós, sendo que cada nó contém ao menos um registro (identificador), podendo ter ramificações para os nós descendentes e/ou ancestrais.

Assim, para cada árvore contém um nó raiz r , a partir de onde outras informações podem ser acessadas.

Os nós internos são filhos de r . A partir de cada nó interno é composta uma subárvore, a qual também possui um nó raiz, que é descendente de r .

Uma árvore pode ter apenas um único elemento (raiz). A partir do nó raiz, cada nó descendente é uma subárvore. Também, uma árvore pode conter nós folhas, os quais não possuem descendentes (filhos).

Em uma árvore pode ser calculada diversas propriedades, das quais as mais comuns são:

- Altura: distância entre o nó raiz e o nó mais profundo (comprimento do caminho mais longo).
- Profundidade: distância entre um determinado nó e a raiz, que tem profundidade 0.
- Balanceamento: se refere ao tamanho das subárvores. Diz-se que uma árvore balanceada possui todas as subárvores com diferença máxima de altura igual a 1.

Veremos quatro tipos de árvores: árvore binária, AVL, vermelha-preta e árvore B.

Hoje abordaremos árvores binárias.

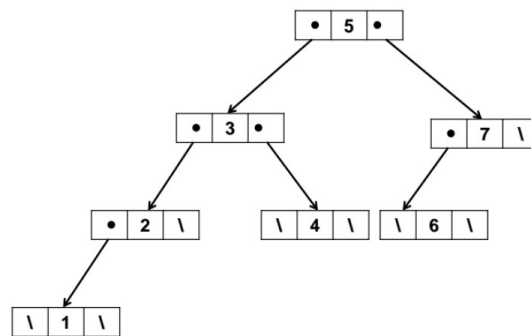
Árvores Binárias (AB)

Árvore em que cada nó contém no máximo dois filhos. As subárvores também contém até dois filhos. Apenas os nós folhas não tem descendentes.

As AB têm algumas propriedades interessantes:

- Uma AB com n elementos tem $n - 1$ ramos
- Uma AB de altura h tem no mínimo h elementos (quando o crescimento é apenas em uma direção, lembrando uma lista) e no máximo $2^h - 1$, quando uma árvore está completamente “cheia” (exceto os nós folhas). Assim, uma árvore cheia de altura h tem $2^{h+1} - 1$.
- Desse modo, a altura de uma AB é no máximo $n - 1$ (crescimento apenas em uma direção) e no mínimo $\log(n) - 1$.

Uma AB pode ser representada através de uma struct com as seguintes informações: informação, subárvore esquerda e subárvore direita.



Estrutura para representação de AB

```
typedef struct Node Node;
```

```

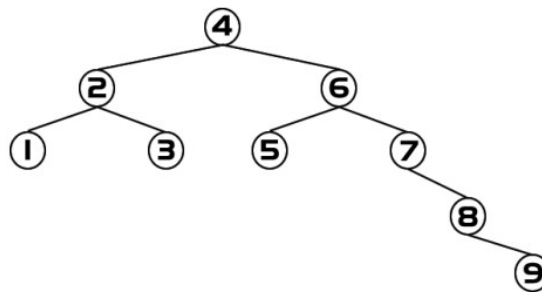
struct Node{
    int item;
    Node* right;
    Node* left;
};

int create(Node* tree){
    if (tree == NULL){
        ree = (Node*) malloc(sizeof(Node));
        tree->left = NULL;
        tree->right = NULL;
        return 1;
    }
    return 0;
}
  
```

```
int release(Node* tree){  
    if (tree != NULL){  
        ree(tree);  
  
        eturn 1;  
    }  
  
    return 0;  
}
```

Árvores Binárias de Busca (ABB)

Também conhecidas como árvores binárias de pesquisa, ABB são AB que exibem propriedades de ordenação.



Uma ABB deve atender as seguintes propriedades:

- Todo nó tem uma chave.
- As chaves (se houver) na subárvore esquerda são menores do que a chave na raiz.
- As chaves (se houver) na subárvore direita são maiores do que a chave na raiz.
- Sobre essas duas últimas propriedades, podem ser inversas (esquerda maior e direita menor), mas são menos usadas.
- As sub-árvores esquerda e direita são também ABB

Principais operações em uma árvore binária de busca:

- Pesquisa
- Inserção
- Remoção
- Percurso (impressão): infix, prefix e postfix

Pesquisa

O objetivo é encontrar o nó com o mesmo valor procurado.

Parecida com a busca binária, mas não há necessidade de cálculos adicionais.

Em uma ABB, a busca é iniciada pela raiz e atende uma das seguintes condições:

- Se o valor for encontrado, o nó é retornado.
- Se o valor é menor que o registro do nó analisado, uma chamada recursiva da função é realizada para a subárvore esquerda.
- Se o valor é maior que o registro do nó analisado, uma chamada recursiva da função é realizada para a subárvore direita.
- A função pode retornar NULL, caso a busca chega ao limite da árvore (nó-folha) e o item não é encontrado.

Implementação da função de busca:

```
Node* search(Node* tree, int value){  
    if (tree != NULL)  
        if (tree->item == value)  
            return tree;  
        else if (tree->item < value)  
            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? Explorando as subárvores esquerda e direita, respectivamente.

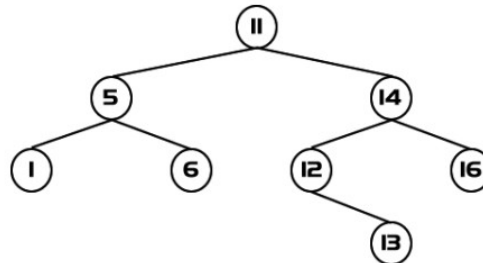
Complexidade para busca, mínimo e máximo:

- Melhor caso: $O(1)$, quando o item procurado está na raiz.
- Caso médio: $O(\log n)$, caso a árvore esteja balanceada e o item procurado esteja em um dos nós folhas ou não é encontrado.
- Pior caso: $O(n)$, quando a árvore tem a forma de uma lista (todos os elementos estão apenas em uma direção) e o item está no nó-folha ou não é encontrado.

Inserção

Percorre a estrutura até chegar a um determinado ponteiro cujo nó filho (direito ou esquerdo) seja NULL

Exemplo: geração de uma árvore binária de busca a partir da sequência 11, 5, 14, 12, 13, 1, 6 e 16



Código para inserção de um nó:

```

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

    return tree;
}
  
```

A função acima permite a inserção de itens repetidos. O você faria para que a árvore não contenha itens repetidos? Resposta: no último *else* basta adicionar *if (value > tree->item)*

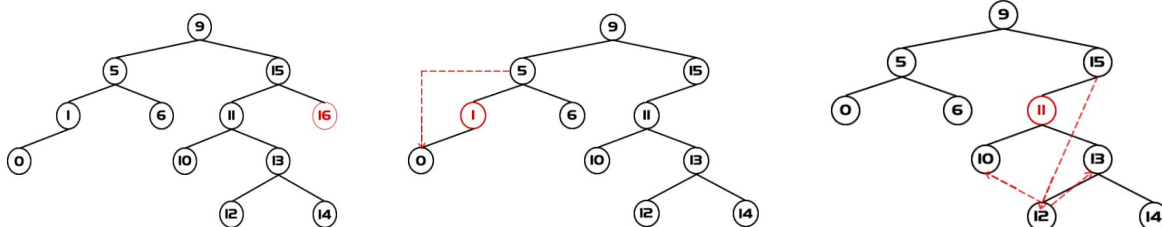
Complexidade para inserção:

- Melhor caso: $O(1)$, quando o item procurado está na raiz.
- Caso médio: $O(\log n)$, caso a árvore esteja balanceada e o item procurado esteja em um dos nós folhas ou não é encontrado.
- Pior caso: $O(n)$, quando a árvore tem a forma de uma lista (todos os elementos estão apenas em uma direção) e o item está no nó-folha ou não é encontrado.

Remoção

Três casos básicos devem ser considerados para a remoção de um nó *z* em uma árvore binária de busca:

- Caso *z* não tenha filhos, remova-o de modo que o nó pai substitua *z* por NULO
- Caso o nó tenha apenas um filho, o mesmo deve substituir o nó *z*
- Caso *z* possua dois filhos, utilizar o nó *y* que contenha o menor valor da subárvore direita para substituir *z*



Código para a remoção de um nó

```
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{// Exclusão de item
            aux = tree;
            if (aux->left == NULL) // funciona para dos casos: quando o nó não possui descendentes ou apenas um do lado direito
                tree = tree->right;
            else if (aux->right == NULL) // nó possui descendente ao lado esquerdo
                tree = tree->left;
            else{ // nó possui descendentes em ambos lados
                auxP = aux->right; // Nó pai de auxF
                auxF = auxP; // menor nó da subárvore direita (irá substituir o nó removido)
                while (auxF->left != NULL){
                    auxP = auxF;
                    auxF = auxF->left;
                }
                auxP->left = auxF->right; // O lado esquerdo de auxP recebe o lado direito de auxF
                auxF->left = aux->left; // o lado esquerdo de auxF aponta para ao lado esquerdo do nó a ser removido
                auxF->right = aux->right; // o lado direito de auxF aponta para ao lado direito do nó a ser removido
                tree = auxF; // atualização do nó analisado
            }
            free(aux); // exclusão segura do nó
            return 1;
        }
    }

    return 0;
}
```

Ver slides de 44 a 51.

Exercício no slide 52.

Complexidade para remoção:

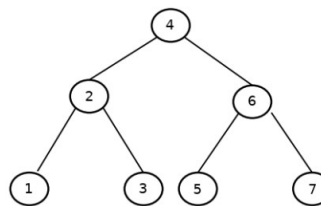
- Melhor caso: $O(1)$, quando o item procurado está na raiz.
- Caso médio: $O(\log n)$, caso a árvore esteja balanceada e o item procurado esteja em um dos nós folhas ou não é encontrado.
- Pior caso: $O(n)$, quando a árvore tem a forma de uma lista (todos os elementos estão apenas em uma direção) e o item está no nó-folha ou não é encontrado.

Percurso

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\n", tree->item);
        prefix(tree->left);
        prefix(tree->right);
    }
}
```

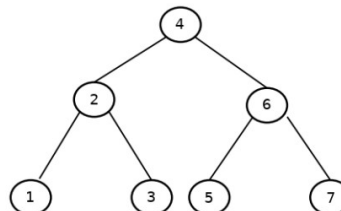


Percurso: 4, 2, 1, 3, 6, 5, 7

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
- Em árvores binárias, a raiz é visitada entre as duas subárvores
- Em árvores n-árias ($n > 2$), 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

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

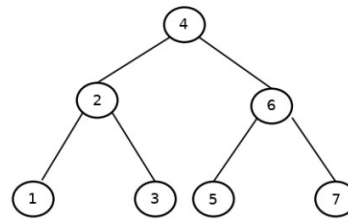


Percurso: 1, 2, 3, 4, 5, 6, 7

Pós-ordem

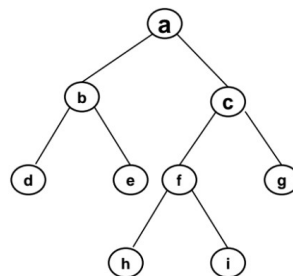
- A raiz é a última a ser visitada
- Todas as subárvores da direita até a esquerda são percorridas em pós-ordem

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



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

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



Referências

- Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Clifford, S. Algoritmos: teoria e prática. Elsevier, 2012.
- Horowitz, E., Sahni, S. Rajasekaran, S. Computer Algorithms. Computer Science Press, 1998.
- Pereira, S. L. Estrutura de Dados e em C: uma abordagem didática. Saraiva, 2016.
- Szwarcfiter, J.; Markenzon, L. Estruturas de Dados e Seus Algoritmos. LTC, 2010.
- Ziviani, M. Projetos de Algoritmos: com implementações em Pascal e C. Thomson, 2004.