OPERAÇÕES EM ÁRVORES BINÁRIAS

Árvores binárias de busca e sua operacionalização

Principais operações em árvores binárias

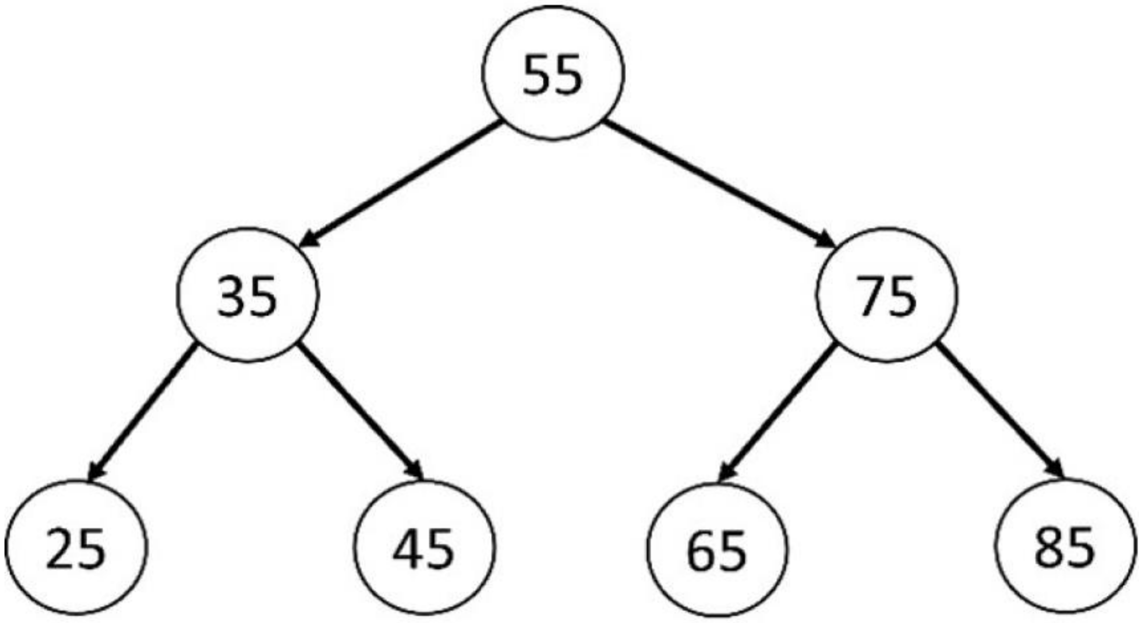
O problema da busca, inserção e remoção é um dos principais objetivos do estudo de estruturas de dados. Utilizar a estrutura de árvores para aplicar regras ao acessar seus dados pode facilitar, computacionalmente, a busca, a inserção e a remoção de dados.

As árvores binárias de busca (do inglês binary search trees – BST's) são árvores de nós organizados de acordo com certas propriedades. A partir desse ponto, considere que as árvores não permitem elementos repetidos. Observe a seguinte definição:

Definição 1: Seja **X** um nó em uma árvore binária de busca. Se **Y** é um nó na subárvore esquerda de **X**, então **Y**. chave ≤ **X**. chave. Se **Y** é um nó na subárvore direita de **X**, então, y. chave ≥ **X**. chave.

Ou seja, árvores binárias de busca são árvores que obedecem às seguintes propriedades:

- Dado um nó qualquer da árvore binária, todos os nós à sua esquerda são menores ou iguais a ele.
- Dado um nó qualquer da árvore binária, todos os nós à direita dele são maiores ou iguais a ele.



No algoritmo 2, é possível verificar a implementação da árvore binária da imagem 6. Observe:

```
class NoArvore:
  def __init__(self, chave = None, esquerda = None, direita = None):
      self.chave = chave
      self.esquerda = esquerda
      self.direita = direita

if __name__ == '__main__':
   raiz = NoArvore(55)
   raiz.esquerda = NoArvore (35)
   raiz.direita  = NoArvore (75)

   raiz.direita.esquerda  = NoArvore(65)
   raiz.direita.direita   = NoArvore(85)
   raiz.esquerda.esquerda = NoArvore(25)
```

Para imprimir a árvore implementada, utilizaremos uma estratégia recursiva para percorrer os nós da árvore e imprimir cada nó

```
cont = [10]


def ImprimeArvoreRecurs(raiz, nivel):
    if (raiz == None):
       return nivel += cont[0]
```

```
# Imprime Filhos à Direita
ImprimeArvoreRecurs(raiz.direita, nivel)
print()
for i in range(cont[0], nivel):
    print(end=" ")
    print(f"{raiz.chave}<")


# Imprime Filhos à Esquerda
ImprimeArvoreRecurs(raiz.esquerda, nivel)


def ImprimeArvore(raiz):
   ImprimeArvoreRecurs(raiz, 0)
```

Para imprimir a árvore implementada, utilizaremos uma estratégia recursiva para percorrer os nós. Se a raiz não for nula (caso base, linha 4), então o algoritmo incrementa um pequeno controlador de níveis da árvore (linha 7), e fazemos duas chamadas recursivas, uma para a subárvore da esquerda e outra para a subárvore da direita. As linhas entre 12 e 15 servem para configurar os nós percorridos em formato de árvore binária.

Busca de nós em BST

O algoritmo de busca decorre diretamente da definição. Considere **X** a chave que desejamos localizar. Compara-se **X** com a raiz; se **X** está nessa raiz, o algoritmo de busca para. Caso contrário, se **X** é menor que a raiz, executa-se o algoritmo recursivamente na subárvore esquerda, caso contrário, na subárvore direita.

```
def BuscaBST(raiz, chave):
    if raiz is None or raiz.chave == chave:
        return raiz


    if raiz.chave < chave:
        return = BuscaBST(raiz.direita, chave)
    else:
        return = BuscaBST(raiz.esquerda, chave)
```

Na busca, existem algumas situações especiais. A primeira ocorre quando a chave buscada está na raiz. Nesse caso, o nó que contém a chave não tem pai, por isso, o algoritmo de busca deverá retornar NULO na referência para o pai. Outra situação especial ocorre quando é realizada uma busca em uma árvore vazia. Nesse cenário, o algoritmo deverá retornar:

NULO

Para o nó que contém a chave.

NULO

Para referência do nó que é pai do nó que contém a chave buscada.

FALSE

Para referência ao booleano.

O algoritmo de busca é utilizado nas operações de inserção e de remoção de nós nas árvores binárias de busca.

Sabemos que a análise de complexidade é baseada na identificação do pior caso e na análise do número de operações elementares que o resolva. O pior caso é, sem dúvida, não encontrar a chave buscada. Nesse cenário, o algoritmo realizará uma comparação por nível até encontrar um nó que não possua o filho no qual a chave buscada poderia estar.

Pela definição de árvore binária de busca, não há restrição para a altura da árvore, sendo uma árvore com topologia zigue-zague, que são árvores nas quais cada nó só possui um filho, podendo ser uma árvore binária de busca. Entretanto, árvores desse teor possuem altura proporcional à **n**. Assim, a complexidade da busca em uma árvore binária de busca é **O**(**n**).

Inserção e remoção em BST

Inserção e remoção em árvores binárias de busca

Inserção de nós em BST

A inserção de uma nova chave em uma árvore binária de busca ocorre sempre em um novo nó, posicionado como uma nova folha da árvore. Isso decorre do fato de que a posição do nó que contém a nova chave é determinada pela sua busca na árvore.

Já vimos que estruturas de dados, nas quais realizamos busca, inserção e remoção, não permitem chave duplicada. Assim, a busca pela chave que desejamos inserir na árvore deverá, obrigatoriamente, falhar e retornar como resultado o nó que será pai do novo nó inserido na árvore.

```
def InserirBST(raiz, chave):
  if raiz is None:
    return NoArvore(chave)
  else:
    if raiz.chave == chave:
      return raiz
    elif raiz.chave < chave:
      raiz.direita = InserirBST(raiz.direita, chave)
    else:
      raiz.esquerda = InserirBST(raiz.esquerda, chave)
  return raiz

if __name__ == '__main__':
    raiz = NoArvore(55)
    InserirBST(raiz, 35)
    InserirBST(raiz, 75)
    InserirBST(raiz, 25)
    InserirBST(raiz, 45)
    InserirBST(raiz, 65)
    InserirBST(raiz, 85)
    ImprimeArvore(raiz)
```

No algoritmo apresentado, temos a implementação em Python da inserção de nós em uma árvore binária de busca.

A principal operação para realização da inserção de um novo nó é a busca. Ela determina a posição e se é possível ou não realizar a inserção. Após a busca, as operações seguintes são todas realizadas em O(1). Sendo assim, a complexidade da inserção é a complexidade da busca que é O(n).

Remoção nós em BST

O processo de remover o nó de uma árvore binária deve obedecer a algumas regras.

1. O nó a ser deletado é uma folha: a remoção é simples. Basta buscar pelo nó e removê-lo.
2. O nó a ser excluído tem apenas um filho: copie o filho para o nó e o exclua.
3. O nó a ser excluído tem dois filhos: encontre o sucessor em ordem do nó. Copie o conteúdo do sucessor em ordem para o nó e o exclua.

```
def DeleteBST(raiz, chave):
  if raiz is None:
      return raiz
  if chave < raiz.chave:
      raiz.esquerda = DeleteBST(raiz.esquerda, chave)
  elif(chave > raiz.chave):
      raiz.direita = DeleteBST(raiz.direita, chave)
  else:
      if raiz.esquerda is None:
            temp = raiz.direita
         raiz = None
         return temp
      elif raiz.direita is None:
         temp = raiz.esquerda
         raiz = None
         return temp
      temp = ValorNo(raiz.direita)
      raiz.chave = temp.chave
      raiz.direita = DeleteBST(raiz.direita, temp.chave)
  return raiz
```

Linhas 2 e 3

Neste passo, se a raiz for None, retorne à raiz (caso base).

Linhas 4 e 5

Neste passo, se a chave for menor que o valor da raiz, defina a instrução raiz.esquerda = DeleteBST(raiz.esquerda, chave).

Linhas 6 e 7

Neste passo, se a chave for maior que o valor da raiz, defina a instrução raiz.direita = DeleteBST (raiz.direita, chave).

Caso contrário

Neste passo:

- Verifique se a raiz for um nó folha, então retorne nulo.
- (Linhas 9 e 12): caso contrário, se tiver apenas o filho esquerdo, retorne-o;
- (Linhas 13 e 16): caso contrário, se tiver apenas o filho direito, retorne-o;
- (Linha 17 e 19): caso contrário, defina o valor de raiz como seu sucessor em ordem e repita para excluir o nó com o valor do sucessor em ordem. Para esse processo, temos a função **ValorNo** para auxiliar.

Ao final

Neste passo, retorne à raiz.

```
def ValorNo(no):
    atual = no
    while(atual.esquerda is not None):
        atual = atual.esquerda
    return atual
```

No estudo da complexidade da remoção, vimos que a remoção é dependente da busca e que ela tem complexidade da **O**(**n**). No caso de remoção de uma folha, ela depende somente da busca. Em seguida, realizamos operações elementares para remover o nó. Portanto, a complexidade do caso 1 é **O**(**n**).

No pior caso, vamos remover um nó interno do penúltimo nível. Portanto, o custo computacional da busca é **O**(**n**), uma vez que vamos percorrer **n**−1 nós para encontrar o nó que será removido. As operações de reapontamento são elementares. Portanto, a complexidade também é **O**(**n**).

No caso de remover um nó com dois descendentes, novamente, a estrutura é próxima a uma árvore zigue-zague. Haverá somente um nó com dois descendentes e ele está no nível **k**. A primeira busca para encontrar o nó que desejamos remover executa **k** comparações, uma vez que esse nó está no nível **k**. Em seguida, procuraremos o substituto do nó no ramo zigue-zague da estrutura, percorrendo-a até o nível **k**−1. Portanto, o custo da busca e dos reapontamentos é de **k**−1 comparações. Logo, **O**(**n**).