MODUL PERKULIAHAN

TFC251 – Praktikum Struktur Data

Tree Dalam Python

Penyusun Modul	:	Suamanda Ika Novichasari, M.Kom
Minggu/Pertemuan	:	9
Sub-CPMK/Tujuan Pembelajaran	:	1. Mahasiswa mampu menerapkan konsep tree pada bahasa pemrograman python
Pokok Bahasan	:	1. Struktur Data Tree 2. Binary Tree 3. Binary Search Tree

Program Studi Teknologi Informasi (S1)
Fakultas Teknik
Universitas Tidar
Tahun 2024

Materi 7
TREE DALAM PYTHON

Petunjuk Praktikum :

- Cobalah semua contoh penyelesaian kasus dengan kode program yang terdapat pada semua sub bab dalam modul ini menggunakan google colab.
- Dokumentasikan kegiatan dalam bentuk laporan studi kasus sesuai template laporan praktikum yang telah ditentukan.

Struktur Data Pohon merupakan tipe struktur data hierarkis di mana sekelompok elemen yang disebut sebagai node saling terhubung melalui sisi sehingga setiap dua node memiliki tepat satu lintasan yang menghubungkan di antara mereka.

7.1 Struktur Data Pohon (*Tree*)

Struktur data pohon adalah sebuah struktur hierarkis yang digunakan untuk mewakili dan mengatur data dengan cara yang mudah untuk dinavigasi dan dicari. Ini merupakan kumpulan dari node yang saling terhubung melalui tepi dan memiliki hubungan hierarkis di antara node-node tersebut.

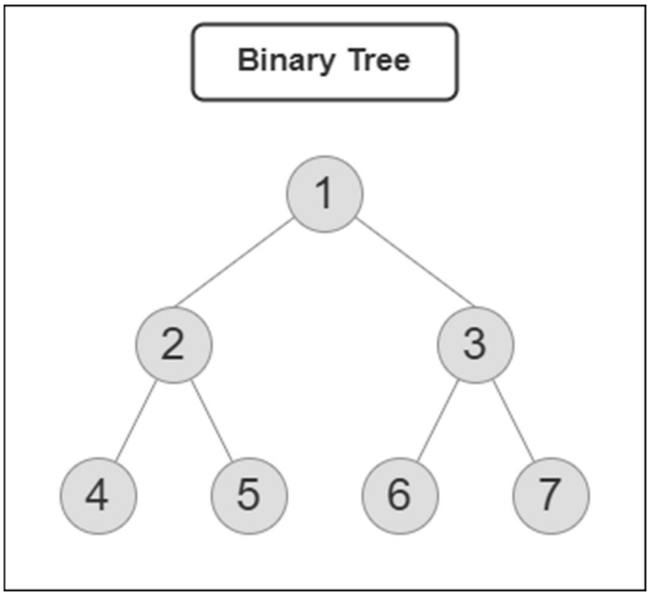
Terminologi dalam Struktur Data Pohon:

- Node Induk: Node yang menjadi pendahulu suatu node disebut node induk dari node tersebut. {B} adalah node induk dari {D, E}.
- Node Anak: Node yang merupakan pengganti langsung dari suatu node disebut node anak dari node tersebut. Contoh: {D, E} adalah node anak dari {B}.
- Node Akar: Node paling atas dari sebuah pohon atau node yang tidak memiliki node induk disebut node akar. {A} adalah node akar dari pohon tersebut. Sebuah pohon yang tidak kosong harus mengandung tepat satu node akar dan tepat satu jalur dari akar ke semua node lain dalam pohon.

- Node Daun atau Node Eksternal: Node yang tidak memiliki node anak disebut node daun. {K, L, M, N, O, P, G} adalah node daun dari pohon tersebut.
- Silsilah dari Sebuah Node: Node-node pendahulu di jalur dari akar ke node tersebut disebut silsilah dari node tersebut. {A, B} adalah node silsilah dari node {E}.
- Keturunan: Sebuah node x adalah keturunan dari node lain y jika dan hanya jika y adalah leluhur dari x.
- Saudara: Anak-anak dari node induk yang sama disebut saudara. {D, E} disebut saudara.
- Tingkat sebuah node: Jumlah tepi pada jalur dari node akar ke node tersebut. Node akar memiliki tingkat 0.
- Node Internal: Sebuah node dengan setidaknya satu node anak disebut Node Internal.
- Tetangga dari Sebuah Node: Node induk atau anak dari node tersebut disebut tetangga dari node tersebut.
- Subpohon: Setiap node dari pohon bersama dengan keturunannya.

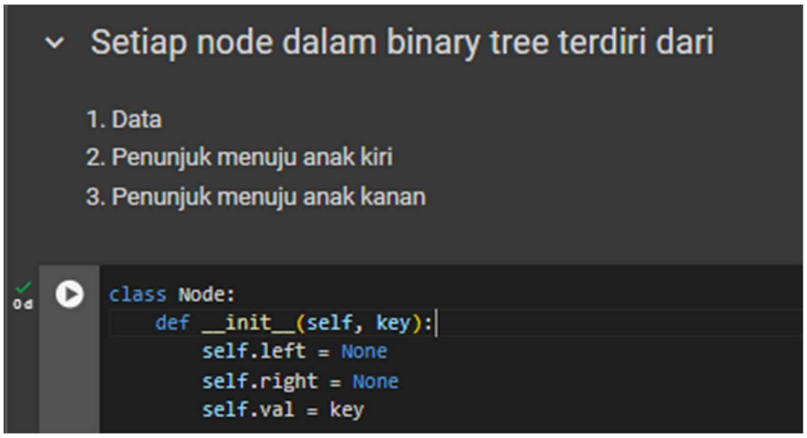
7.2 Binary Tree

Dalam sebuah pohon biner, setiap node dapat memiliki maksimum dua anak yang terhubung kepadanya. Beberapa jenis umum dari pohon biner meliputi pohon biner penuh, pohon biner lengkap, pohon biner seimbang, dan pohon biner degeneratif atau patologis.



Node tertinggi dalam sebuah pohon biner disebut sebagai akar, sementara node-node terendah disebut sebagai daun. Secara konseptual, pohon biner membentuk susunan hierarkis, menempatkan node akar di puncak dan node daun di dasar. Setiap node individu dalam pohon memiliki atribut berikut:

1. Data
2. Penunjuk menuju anak kiri
3. Penunjuk menuju anak kanan



Pohon biner banyak digunakan dalam berbagai bidang ilmu komputer, seperti penyimpanan dan pengambilan data, evaluasi ekspresi, routing jaringan, dan kecerdasan buatan permainan. Selain itu, mereka berfungsi sebagai dasar untuk berbagai implementasi algoritma yang mencakup rutinitas pencarian, teknik pengurutan, dan analisis graf.

Implementasi sederhana dari binary tree

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key


# Driver code
if __name__ == '__main__':
    # Create root
    root = Node(1)
    ''' following is the tree after above statement
    1
    / \
    None None'''
    root.left = Node(2)
    root.right = Node(3)

    ''' 2 and 3 become left and right children of 1
       1
      /  \
     2    3
    / \  / \
   None None None None'''

    root.left.left = Node(4)
    '''4 becomes left child of 2
        1
       / \
      2   3
     / \ / \
    4 None None None
   / \
   None None'''
```

Penerapan Pohon Biner:

- Algoritma pencarian: Algoritma pencarian biner menggunakan struktur pohon biner untuk mencari elemen tertentu secara efisien. Pencarian dapat dilakukan dalam kompleksitas waktu O(log n), di mana n adalah jumlah node dalam pohon.
- Algoritma pengurutan: Pohon biner dapat digunakan untuk mengimplementasikan algoritma pengurutan yang efisien, seperti pengurutan pohon pencarian biner dan pengurutan tumpukan.
- Sistem basis data: Pohon biner dapat digunakan untuk menyimpan data dalam sistem basis data, dengan setiap node mewakili sebuah catatan. Ini memungkinkan operasi pencarian yang efisien dan memungkinkan sistem basis data untuk menangani jumlah data yang besar.

- Sistem file: Pohon biner dapat digunakan untuk mengimplementasikan sistem file, di mana setiap node mewakili sebuah direktori atau file. Ini memungkinkan navigasi dan pencarian yang efisien dari sistem file.
- Algoritma kompresi: Pohon biner dapat digunakan untuk mengimplementasikan pengkodean Huffman, sebuah algoritma kompresi yang memberikan kode berpanjang variabel kepada karakter berdasarkan frekuensi kemunculannya dalam data masukan.
- Pohon keputusan: Pohon biner dapat digunakan untuk mengimplementasikan pohon keputusan, sebuah jenis algoritma pembelajaran mesin yang digunakan untuk klasifikasi dan analisis regresi.
- Kecerdasan buatan permainan: Pohon biner dapat digunakan untuk mengimplementasikan kecerdasan buatan permainan, di mana setiap node mewakili gerakan yang mungkin dalam permainan. Algoritma kecerdasan buatan dapat mencari pohon untuk menemukan gerakan terbaik yang mungkin.

Penerapan real time dari Pohon Biner:

- DOM dalam HTML.
- Penjelajah file.
- Digunakan sebagai struktur data dasar dalam Microsoft Excel dan lembar kerja.
- Alat editor: Microsoft Excel dan lembar kerja.
- Evaluasi sebuah ekspresi
- Algoritma Routing

Keuntungan Pohon Biner:

- Pencarian yang efisien: Pohon biner terutama efisien saat mencari elemen tertentu, karena setiap node memiliki maksimal dua node anak, memungkinkan penggunaan algoritma pencarian biner. Ini berarti operasi pencarian dapat dilakukan dalam kompleksitas waktu O(log n).
- Traversing yang terurut: Struktur pohon biner memungkinkan mereka untuk diperlakukan dalam urutan tertentu, seperti urutan dalam, urutan pra, dan urutan pasca. Ini memungkinkan operasi dilakukan pada node-node dalam urutan tertentu, seperti mencetak node-node dalam urutan terurut.
- Efisien dalam penggunaan memori: Dibandingkan dengan struktur pohon lainnya, pohon biner relatif efisien dalam penggunaan memori karena hanya membutuhkan dua pointer anak per node. Ini berarti mereka dapat digunakan

untuk menyimpan jumlah data yang besar dalam memori sambil tetap menjaga operasi pencarian yang efisien.

- Penyisipan dan penghapusan yang cepat: Pohon biner dapat digunakan untuk melakukan penyisipan dan penghapusan dalam kompleksitas waktu O(log n). Ini membuat mereka menjadi pilihan yang baik untuk aplikasi yang membutuhkan struktur data dinamis, seperti sistem basis data.
- Mudah diimplementasikan: Pohon biner relatif mudah diimplementasikan dan dipahami, menjadikannya pilihan yang populer untuk berbagai aplikasi.
- Berguna untuk pengurutan: Pohon biner dapat digunakan untuk mengimplementasikan algoritma pengurutan yang efisien, seperti pengurutan tumpukan dan pengurutan pohon pencarian biner.

Kekurangan Pohon Biner:

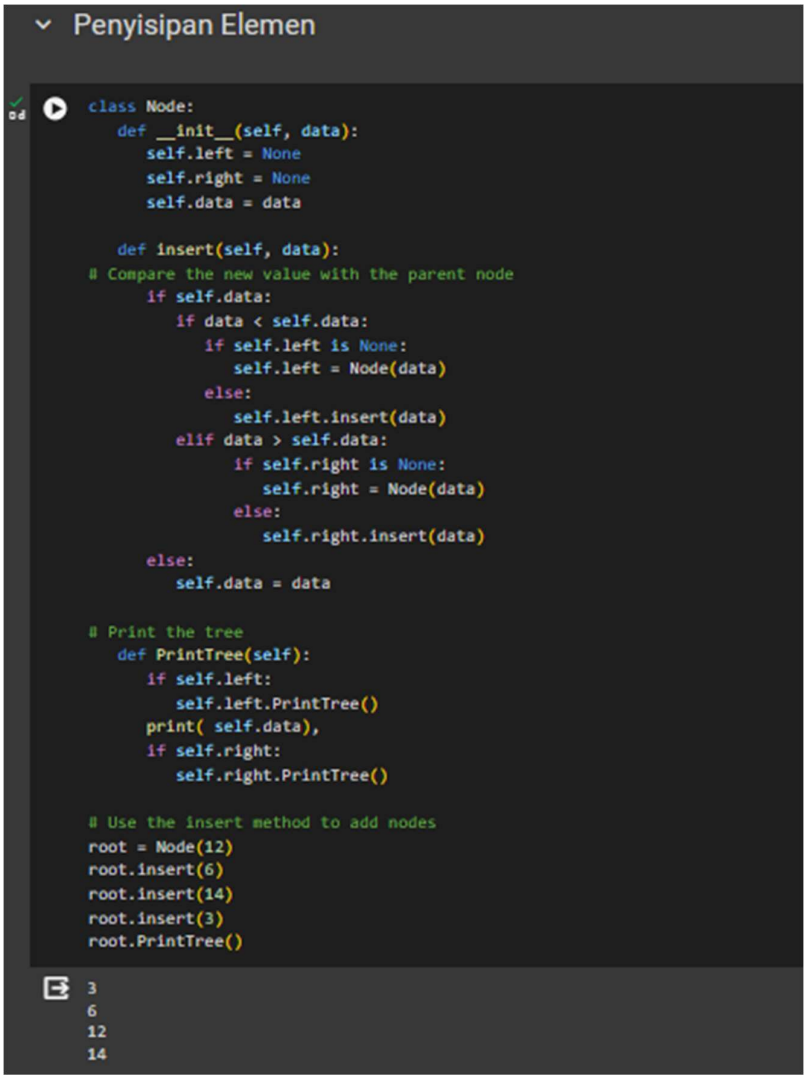
- Struktur terbatas: Pohon biner terbatas pada dua node anak per node, yang dapat membatasi kegunaannya dalam aplikasi tertentu. Misalnya, jika suatu pohon memerlukan lebih dari dua node anak per node, struktur pohon yang berbeda mungkin lebih cocok.
- Pohon yang tidak seimbang: Pohon biner yang tidak seimbang, di mana satu subpohon jauh lebih besar dari yang lain, dapat mengakibatkan operasi pencarian yang tidak efisien. Ini dapat terjadi jika pohon tidak seimbang dengan baik atau jika data dimasukkan dalam urutan yang tidak acak.
- Ineffisiensi ruang: Pohon biner dapat tidak efisien dalam penggunaan ruang dibandingkan dengan struktur data lainnya. Hal ini karena setiap node membutuhkan dua pointer anak, yang dapat menjadi jumlah memori yang signifikan untuk pohon yang besar.
- Kinerja lambat dalam skenario terburuk: Dalam skenario terburuk, sebuah pohon biner dapat menjadi degeneratif, yang berarti setiap node memiliki hanya satu node anak. Dalam kasus ini, operasi pencarian dapat terdegradasi menjadi kompleksitas waktu O(n), di mana n adalah jumlah node dalam pohon.
- Algoritma penyeimbangan yang kompleks: Untuk memastikan bahwa pohon biner tetap seimbang, berbagai algoritma penyeimbangan dapat digunakan, seperti pohon AVL dan pohon merah-hitam. Algoritma-algoritma ini dapat rumit untuk diimplementasikan dan memerlukan overhead tambahan, menjadikannya kurang cocok untuk beberapa aplikasi.

Operasi Dasar pada Pohon Biner:

- Menyisipkan sebuah elemen.
- Menghapus sebuah elemen.
- Mencari sebuah elemen.
- Menghapus sebuah elemen.
- Melintasi sebuah elemen. Terdapat empat (utamanya tiga) jenis lintasan dalam sebuah pohon biner yang akan dibahas selanjutnya.

7.2.1 Penyisipan Elemen

Untuk menyisipkan ke dalam sebuah pohon, kita menggunakan kelas node yang sama yang telah dibuat sebelumnya dan menambahkan kelas sisip ke dalamnya. Kelas sisip membandingkan nilai node dengan node induk dan memutuskan untuk menambahkannya sebagai node kiri atau node kanan. Akhirnya, kelas PrintTree digunakan untuk mencetak pohon tersebut.



7.2.2 Lintasan Elemen

In-order Traversal

Dalam metode lintasan ini, subpohon kiri dikunjungi terlebih dahulu, kemudian akar, dan kemudian subpohon kanan. Kita harus selalu ingat bahwa setiap node dapat mewakili sebuah subpohon itu sendiri.

Pada program Python di bawah ini, kami menggunakan kelas Node untuk membuat placeholder untuk node akar serta node kiri dan kanan. Kemudian, kami membuat fungsi sisip untuk menambahkan data ke pohon. Akhirnya, logika lintasan urutan dalam diimplementasikan dengan membuat daftar kosong dan menambahkan node kiri terlebih dahulu diikuti oleh node akar atau induk.

Pada akhirnya, node kanan ditambahkan untuk menyelesaikan lintasan urutan dalam. Harap dicatat bahwa proses ini diulang untuk setiap subpohon sampai semua node dilintasi.

In-order Traversal

```
class Node:
    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data
# Insert Node
    def insert(self, data):
        if self.data:
            if data < self.data:
                if self.left is None:
                    self.left = Node(data)
                else:
                    self.left.insert(data)
            elif data > self.data:
                if self.right is None:
                    self.right = Node(data)
                else:
                    self.right.insert(data)
        else:
            self.data = data
# Print the Tree
    def PrintTree(self):
        if self.left:
            self.left.PrintTree()
        print( self.data),
        if self.right:
            self.right.PrintTree()
# Inorder traversal
# Left -> Root -> Right
    def inorderTraversal(self, root):
        res = []
        if root:
            res = self.inorderTraversal(root.left)
            res.append(root.data)
            res = res + self.inorderTraversal(root.right)
        return res
```

```
root = Node(27)
root.insert(14)
root.insert(35)
root.insert(10)
root.insert(19)
root.insert(31)
root.insert(42)
print(root.inorderTraversal(root))
```

```
[10, 14, 19, 27, 31, 35, 42]
```

Pre-order Traversal

Dalam metode lintasan ini, node akar dikunjungi terlebih dahulu, kemudian subpohon kiri, dan akhirnya subpohon kanan.

Pada program Python di bawah ini, kami menggunakan kelas Node untuk membuat placeholder untuk node akar serta node kiri dan kanan. Kemudian, kami

membuat fungsi sisip untuk menambahkan data ke pohon. Akhirnya, logika lintasan pra-pesanan diimplementasikan dengan membuat daftar kosong dan menambahkan node akar terlebih dahulu diikuti oleh node kiri.

Pada akhirnya, node kanan ditambahkan untuk menyelesaikan lintasan pra-pesanan. Harap dicatat bahwa, proses ini diulang untuk setiap subpohon sampai semua node dilintasi.

Pre-order Traversal

```
class Node:
   def __init__(self, data):
      self.left = None
      self.right = None
      self.data = data
# Insert Node
   def insert(self, data):
      if self.data:
         if data < self.data:
            if self.left is None:
               self.left = Node(data)
            else:
               self.left.insert(data)
         elif data > self.data:
            if self.right is None:
               self.right = Node(data)
            else:
               self.right.insert(data)
         else:
            self.data = data
# Print the Tree
   def PrintTree(self):
      if self.left:
         self.left.PrintTree()
      print( self.data),
      if self.right:
         self.right.PrintTree()
# Preorder traversal
# Root -> Left ->Right
   def PreorderTraversal(self, root):
      res = []
      if root:
         res.append(root.data)
         res = res + self.PreorderTraversal(root.left)
         res = res + self.PreorderTraversal(root.right)
      return res
```

```
root = Node(27)
root.insert(14)
root.insert(35)
root.insert(10)
root.insert(19)
root.insert(31)
root.insert(42)
print(root.PreorderTraversal(root))
```

```
[27, 14, 10, 19, 35, 31, 42]
```

Post-order Traversal

Dalam metode lintasan ini, node akar dikunjungi terakhir, oleh karena itu namanya. Pertama, kita melintasi subpohon kiri, kemudian subpohon kanan, dan akhirnya node akar.

Pada program Python di bawah ini, kami menggunakan kelas Node untuk membuat placeholder untuk node akar serta node kiri dan kanan. Kemudian, kami membuat fungsi sisip untuk menambahkan data ke pohon. Akhirnya, logika lintasan pasca-dipesan diimplementasikan dengan membuat daftar kosong dan menambahkan node kiri terlebih dahulu diikuti oleh node kanan.

Pada akhirnya, node akar atau induk ditambahkan untuk menyelesaikan lintasan pasca-dipesan. Harap dicatat bahwa, proses ini diulang untuk setiap subpohon sampai semua node dilintasi.

Post-order Traversal

```
class Node:
    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data
# Insert Node
    def insert(self, data):
        if self.data:
            if data < self.data:
                if self.left is None:
                    self.left = Node(data)
                else:
                    self.left.insert(data)
            elif data > self.data:
                if self.right is None:
                    self.right = Node(data)
                else:

                    self.right.insert(data)
        else:
            self.data = data
# Print the Tree
    def PrintTree(self):
        if self.left:
            self.left.PrintTree()
        print( self.data),
        if self.right:
          self.right.PrintTree()
        # Postorder traversal
        # Left ->Right -> Root
    def PostorderTraversal(self, root):
        res = []
        if root:
          res = self.PostorderTraversal(root.left)
          res = res + self.PostorderTraversal(root.right)
          res.append(root.data)
        return res
```

```
root = Node(27)
root.insert(14)
root.insert(35)
root.insert(10)
root.insert(19)
root.insert(31)
root.insert(42)
print(root.PostorderTraversal(root))
```

```
[10, 19, 14, 31, 42, 35, 27]
```

7.2.3 Hapus Elemen

Diberikan sebuah pohon biner, hapus sebuah node darinya dengan memastikan bahwa pohon menyusut dari bawah (yaitu, node yang dihapus digantikan oleh node paling bawah dan paling kanan). Berbeda dengan penghapusan BST, di sini kita tidak memiliki urutan di antara elemen-elemen, jadi kita menggantinya dengan elemen terakhir.

Hapus Elemen

```
[18] class Node:
        def __init__(self, data):
            self.data = data
            self.left = None
            self.right = None

    # Inorder traversal of a binary tree


    def inorder(temp):
        if(not temp):
            return
        inorder(temp.left)
        print(temp.data, end=" ")
        inorder(temp.right)

    # function to delete the given deepest node (d_node) in binary tree


    def deleteDeepest(root, d_node):
        q = []
        q.append(root)
        while(len(q)):
            temp = q.pop(0)
            if temp is d_node:
                temp = None
                return
            if temp.right:
                if temp.right is d_node:
                    temp.right = None
                    return
                else:
                    q.append(temp.right)
            if temp.left:
                if temp.left is d_node:
                    temp.left = None
                    return
                else:
                    q.append(temp.left)
```

```python
# function to delete element in binary tree
def deletion(root, key):
    if root == None:
        return None
    if root.left == None and root.right == None:
        if root.key == key:
            return None
        else:
            return root
    key_node = None
    q = []
    q.append(root)
    temp = None
    while(len(q)):
        temp = q.pop(0)
        if temp.data == key:
            key_node = temp
        if temp.left:
            q.append(temp.left)
        if temp.right:
            q.append(temp.right)
    if key_node:
        x = temp.data
        key_node.data = x
        deleteDeepest(root, temp)
    return root

# Driver code
if __name__ == '__main__':
    root = Node(10)
    root.left = Node(11)
    root.left.left = Node(7)
    root.left.right = Node(12)
    root.right = Node(9)
    root.right.left = Node(15)
    root.right.right = Node(8)
    print("The tree before the deletion: ", end = "")
    inorder(root)
    key = 11
    root = deletion(root, key)
    print();
    print("The tree after the deletion: ", end = "")
    inorder(root)
```

```
The tree before the deletion: 7 11 12 10 15 9 8
The tree after the deletion: 7 8 12 10 15 9
```

7.3 Pencarian Elemen pada Binary Search Tree (BST)

Misalkan kita ingin mencari nomor X. Kita mulai dari akar. Kemudian Kita membandingkan nilai yang ingin dicari dengan nilai akar. Jika sama, pencarian selesai. Jika lebih kecil, kita tahu bahwa kita perlu pergi ke subpohon kiri karena dalam pohon pencarian biner semua elemen dalam subpohon kiri lebih kecil dan semua elemen dalam subpohon kanan lebih besar. Ulangi langkah di atas sampai tidak ada lagi penelusuran yang mungkin. Jika pada iterasi apa pun, kunci ditemukan, kembalikan Benar. Jika tidak, kembalikan Salah.

Pencarian Pada Binary Search Tree (BST)

```python
# Python3 function to search a given key in a given BST
class Node:
    # Constructor to create a new node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

# A utility function to insert
# a new node with the given key in BST
def insert(node, key):
    # If the tree is empty, return a new node
    if node is None:
        return Node(key)

    # Otherwise, recur down the tree
    if key < node.key:
        node.left = insert(node.left, key)
    elif key > node.key:
        node.right = insert(node.right, key)

    # Return the (unchanged) node pointer
    return node

# Utility function to search a key in a BST
def search(root, key):
    # Base Cases: root is null or key is present at root
    if root is None or root.key == key:
        return root

    # Key is greater than root's key
    if root.key < key:
        return search(root.right, key)

    # Key is smaller than root's key
    return search(root.left, key)
```

```python
# Driver Code
if __name__ == '__main__':
    root = None
    root = insert(root, 50)
    insert(root, 30)
    insert(root, 20)
    insert(root, 40)
    insert(root, 70)
    insert(root, 60)
    insert(root, 80)

    # Key to be found
    key = 6

    # Searching in a BST
    if search(root, key) is None:
        print(key, "not found")
    else:
        print(key, "found")

    key = 60

    # Searching in a BST
    if search(root, key) is None:
        print(key, "not found")
    else:
        print(key, "found")
```

```
6 not found
60 found
```

Latihan

Buatlah implementasi penyisipan, traversal dan hapus elemen pada Binary Search Tree (BST) dalam bahasa pemrograman python !