Bab 6. Binary Tree

OBJEKTIF:

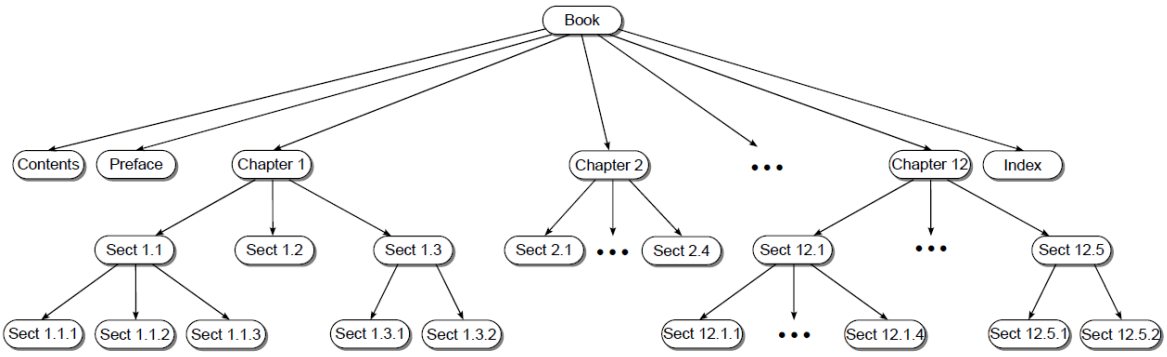
1. Mahasiswa mampu memahami Struktur Tree dan Binary Tree pada Python.
2. Mahasiswa mampu mengimplementasikan Binary Tree dan ADT Binary Search Tree pada Python.

Pada topik-topik sebelumnya kita telah membahas struktur sekuensial seperti array, linked list, stack, dan queue. Struktur-strutkur ini mengorganisasi data dalam bentuk *linear* dimana setiap elemen data mempunyai relasi "sebelum" dan "sesudah". Struktur-struktur *linear* ini dapat digunakan untuk menyelesaikan banyak jenis persoalan, tetapi beberapa persoalan memerlukan data diorganisasi secara *nonlinear*. Pada topik ini kita akan membahas struktur data tree yang dapat digunakan untuk menyusun data dalam urutan hirarki.

6.1 Struktur Tree

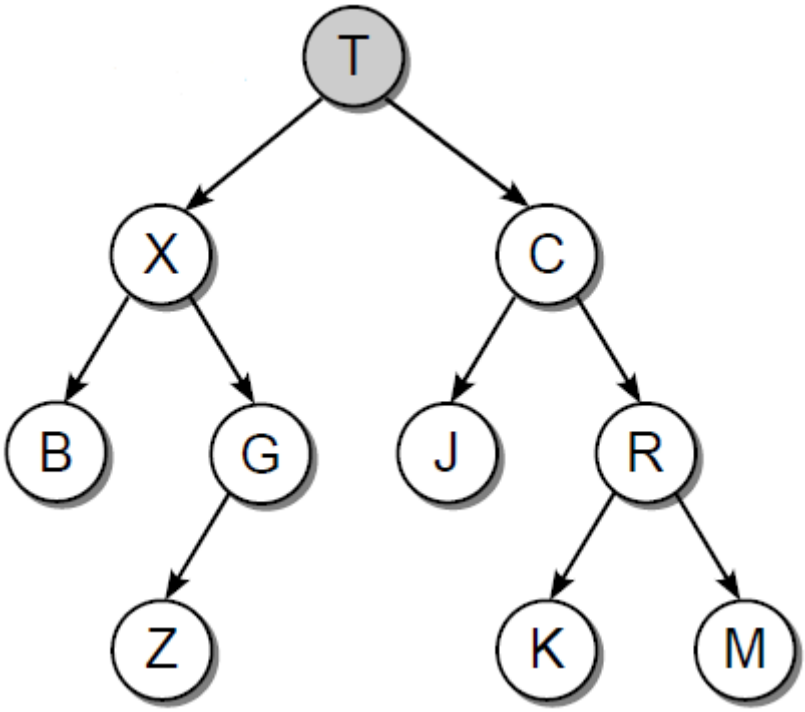
Sebuah struktur tree terdiri dari **node** dan **edge** yang mengorganisasikan data dalam bentuk hirarki. Hubungan antara elemen data dalam sebuah tree mirip dengan pohon keturunan, seperti "anak", "orang tua", "nenek moyang", dan sebagainya. Elemen data disimpan dalam node dan sepasang node dihubungkan oleh edge. Edge menggambarkan hubungan antara node yang terhubung oleh tanda panah atau garis langsung dalam struktur hirarki yang membentuk sebuah pohon terbalik, lengkap dengan ranting, dedaunan dan bahkan akar.

Struktur tree dapat digambarkan dengan pembagian sebuah buku menjadi bagian-bagian bab, subbab, dan subsubbab, seperti diilustrasikan pada gambar berikut:

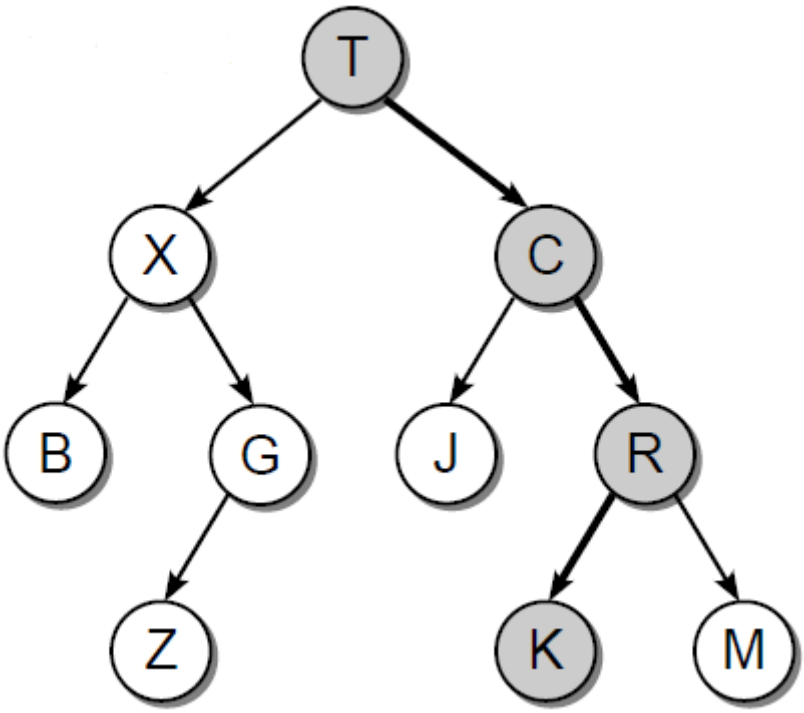


Terdapat beberapa istilah berbeda untuk tiap-tiap karakteristik dan komponen dari tree. Terminologi yang paling sering digunakan adalah hubungan keluarga atau deskripsi dari pohon yang sesungguhnya. Berikut adalah istilah-istilah dari komponen-komponen struktur tree:

- **Root**
  Node paling atas dari tree disebut dengan node **root** (node akar). Node ini menyediakan titik akses ke struktur tree. Node root adalah satu-satunya node dalam struktur tree yang tidak memiliki edge yang menunjuk langsung ke dirinya. Setiap struktur tree yang tidak kosong, pasti memiliki node root. Pada gambar tree berikut, node dengan nilai **T** adalah **root** dari tree:

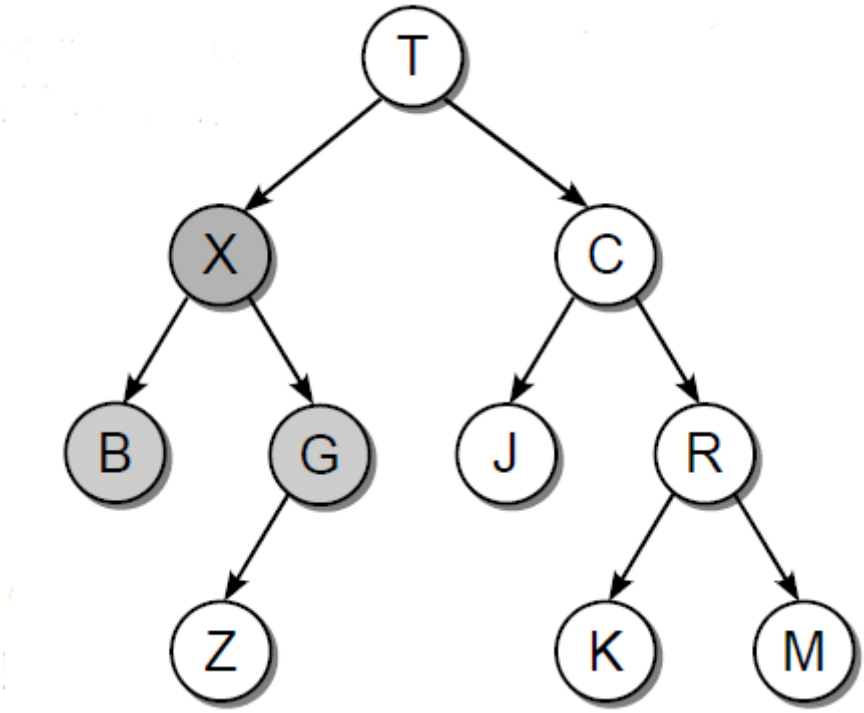


- **Path**
  Node-node lain dalam tree diakses dengan mengikuti edge yang ada, dimulai dari root node dan seterusnya hingga node tujuan tercapai. Node-node bertemu ketika mengikuti edge dari node awal hingga ke node tujuan membentuk sebuah ***path*** (jalur). Gambar berikut mengilustrasikan path dari node root T ke node K:



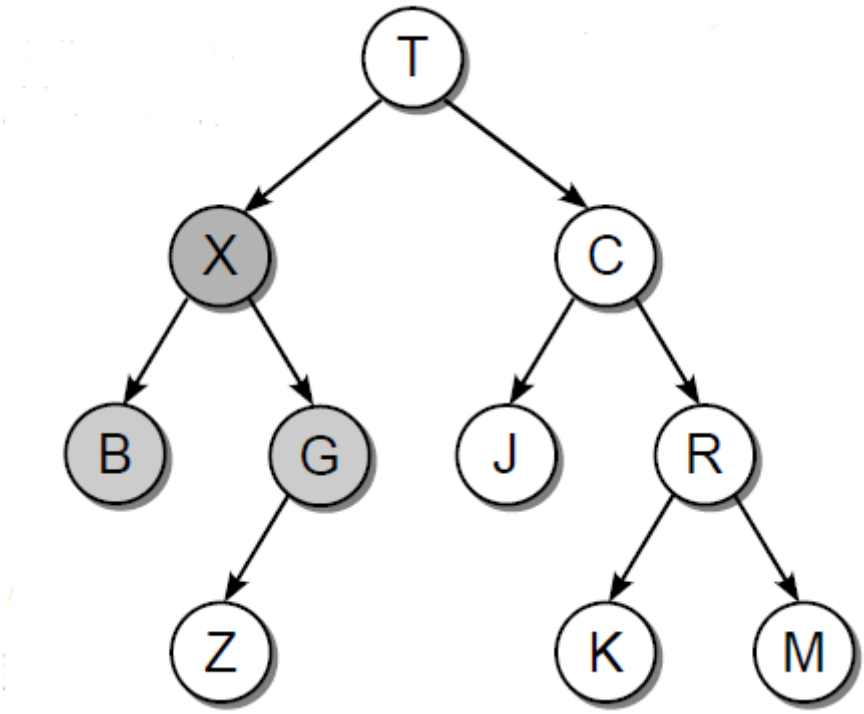
  Pada gambar di atas, node **T**, **C**, **R** dan **K** membentuk sebuah **path dari T ke K**.

- **Parent**
  Organisasi dari node-node membentuk hubungan diantara elemen data. Setiap node, kecuali root node, memiliki node ***parent*** (orangtua), yang dimana dapat dilihat dari node yang edge-nya menunjuk node lain. Sebuah node hanya dapat memiliki satu node parent sehingga menghasilkan path unik dari akar ke node manapun di dalam tree. Sebagai contoh, perhatikan gambar tree berikut:



Pada gambar di atas terdapat beberapa node parent, salah satunya adalah **X** yang merupakan **node *parent*** dari **B** dan **G**.

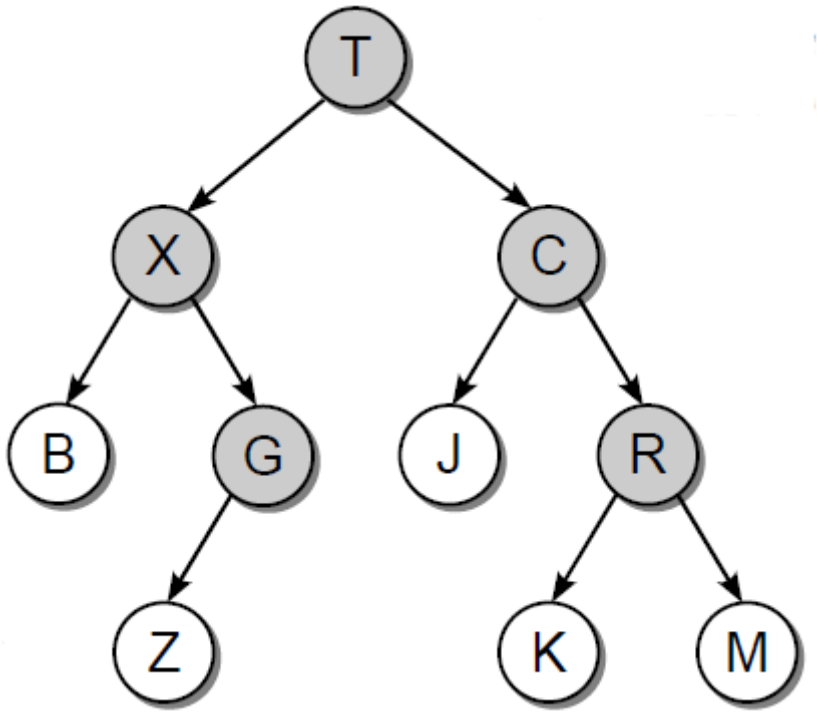
- **Child**
  Setiap node dapat memiliki satu atau lebih node ***child*** (anak) sehingga menghasilkan hirarki orangtua-anak. Node-node child dari sebuah node dapat dilihat dari node yang ditunjuk oleh node parent. Perhatikan gambar berikut:



Pada gambar di atas, **B** dan **G** adalah node-node ***child*** dari **X**. Semua node yang memiliki node parent yang sama disebut sebagai ***sibling*** (saudara). Perlu diperhatikan bahwa tidak

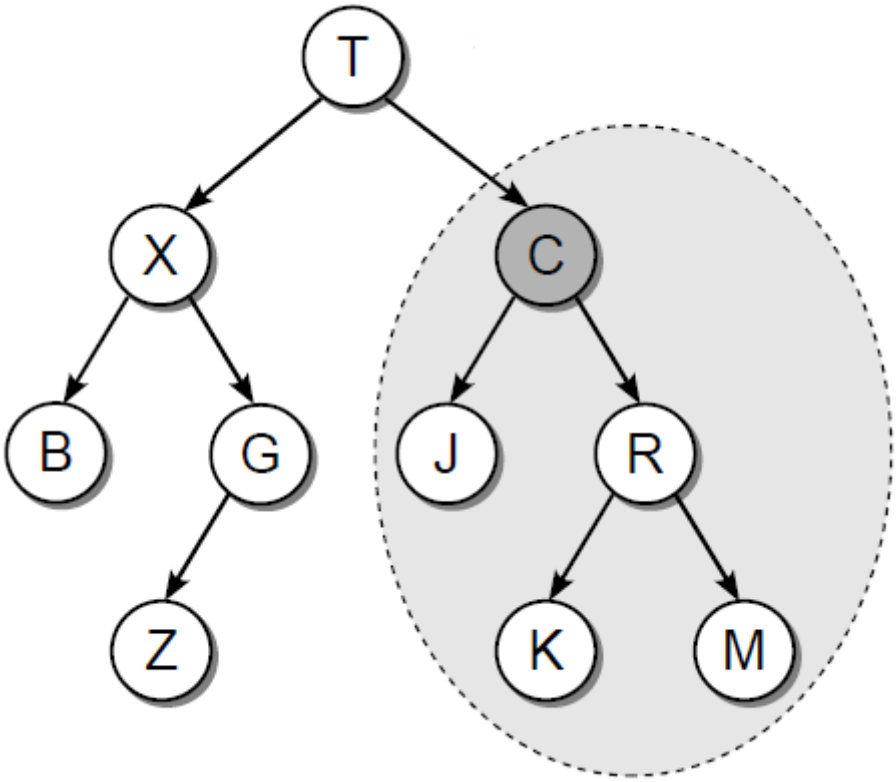
terdapat akses langsung antara sibling. Sebagai contoh, kita tidak bisa mengakses node C dari node X secara langsung.

- **Node**
  Node yang memiliki setidaknya satu node child disebut sebagai node ***interior*** sedangkan node yang tidak memiliki node child disebut sebagai node ***leaf*** (daun). Perhatikan gambar binary tree berikut:



  Pada gambar di atas, node dengan warna abu-abu adalah contoh dari node ***interior*** sedangkan node dengan warna putih adalah contoh dari node ***leaf***.

- **Subtree**
  Struktur tree adalah struktur rekursif. Setiap node dapat menjadi node root dari tiap-tiap **subtree**-nya, yang terdiri atas sekumpulan node dan edge dari tree yang lebih besar. Gambar berikut menunjukkan subtree dengan node C sebagai root-nya:



Perhatikan gambar di atas, kumpulan node dan edge di dalam lingkaran tersebut merupakan contoh dari sebuah subtree. Dalam subtree tersebut, C adalah node root-nya.

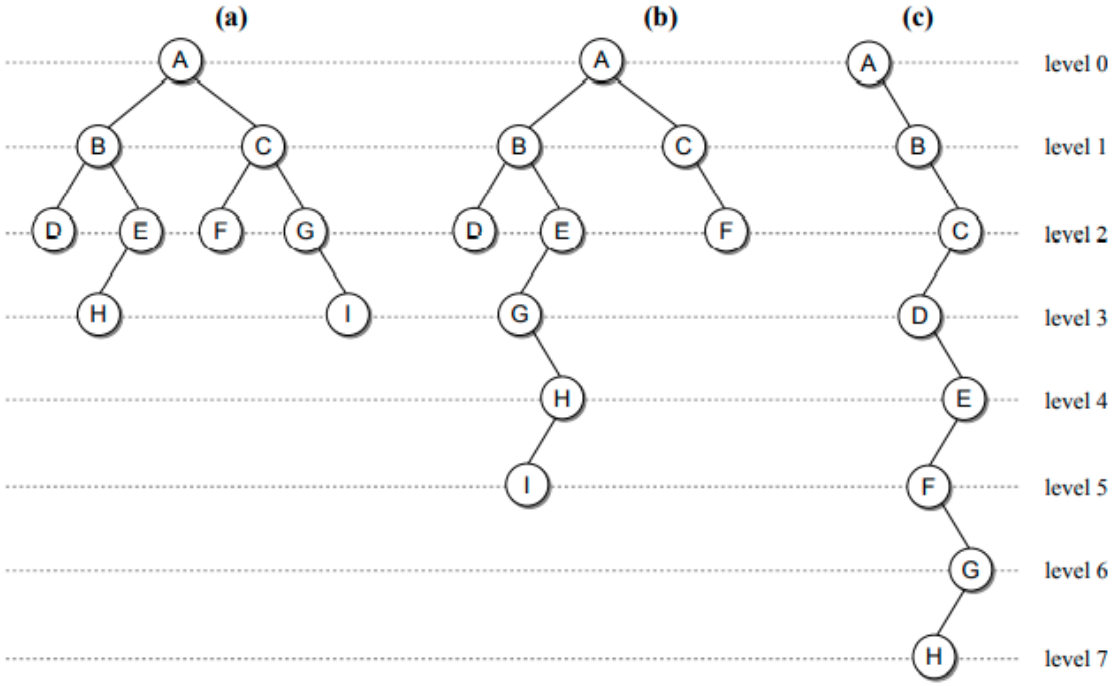
- **Relatives**
  Semua node dalam subtree adalah ***descendant*** (keturunan) dari root subtree. Misalnya, node J, R, K, dan M adalah turunan dari node C. ***Ancestor*** (nenek moyang) dari sebuah node adalah node parent hingga node root. Ancestor dari sebuah node dapat dilihat dari node-node sepanjang path dari node root sampai node tertentu. Node root adalah ancestor dari setiap node di dalam tree dan setiap node lain dalam tree adalah descendant dari node root.

6.2 Binary Tree

Struktur tree dapat memiliki berbagai bentuk dan bergantung pada jumlah dari node child yang diperbolehkan di setiap nodenya. Salah satu struktur tree yang sering digunakan dalam dunia komputer adalah binary tree. ***Binary tree*** adalah sebuah tree yang setiap node-nya memiliki paling banyak dua node child. Satu node child disebut sebagai node ***child* kiri** dan yang lainnya disebut sebagai node **child kanan**.

Karakteristik Binary Tree

Binary tree dapat mempunyai berbagai bentuk dan ukuran. Bentuk-bentuk binary tree bergantung pada banyaknya node dan bagaimana node-node tersebut terhubung. Gambar berikut mengilustrasikan berbagai bentuk tree yang terdiri dari 9 node:



Terdapat sejumlah karakteristik yang terasosiasikan dengan binary tree, kesemuanya bergantung pada pengorganisasian node-node dalam tree.

Node-node pada tree diorganisasikan ke dalam ***level*** dengan ketentuan:

- Node root pada level 0,
- Node children root pada level 1,
- Node children dari node children level 1 ada pada level 2, dan seterusnya

Jika diandaikan sebagai pohon keluarga, maka setiap level node sama seperti tingkat generasi. Sebagai contoh, pada gambar struktur tree (a) di atas, kita bisa lihat bahwa di level 1 terdapat dua node (B dan C), empat node di level 2 (D, E, F, G), dan dua node di level 3 (H dan I). Node root (A) selalu berada pada level 0.

Depth (kedalaman) dari node binary tree ditentukan dari jaraknya terhadap node root. Kedalaman node sesuai dengan level yang ditempatinya. Misalnya, pada gambar di atas, node G pada tree (a) memiliki kedalaman 2 level, pada tree (b) memiliki kedalaman 3 level, dan pada tree (c) memiliki kedalaman 6 level.

Height (ketinggian) dari binary tree adalah banyaknya level dalam struktur tree. Misalnya, pada gambar di atas, ketinggian dari tree (a) adalah 4, tree (b) adalah 6, dan tree (c) adalah 8.

Width (lebar) dari binary tree adalah jumlah node terbanyak yang ada pada suatu level di struktur tree. Misalnya, pada gambar di atas, lebar dari tree (a) adalah 4, tree (b) adalah 3, dan tree (c) adalah 1.

Size (ukuran) dari binary tree ditentukan dari banyaknya node dalam tree. Ukuran dari ketiga tree di atas, (a), (b), dan (c) adalah 9.

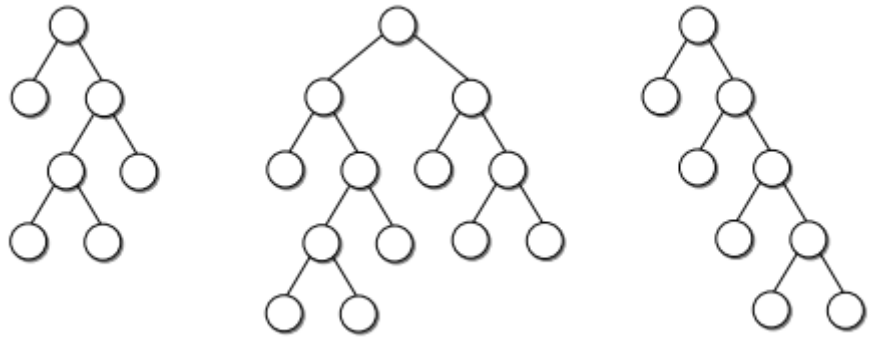
Perlu diperhatikan bahwa sebuah tree kosong memiliki ketinggian 0, lebar 0, dan ukuran 0.

Tipe-tipe Binary Tree

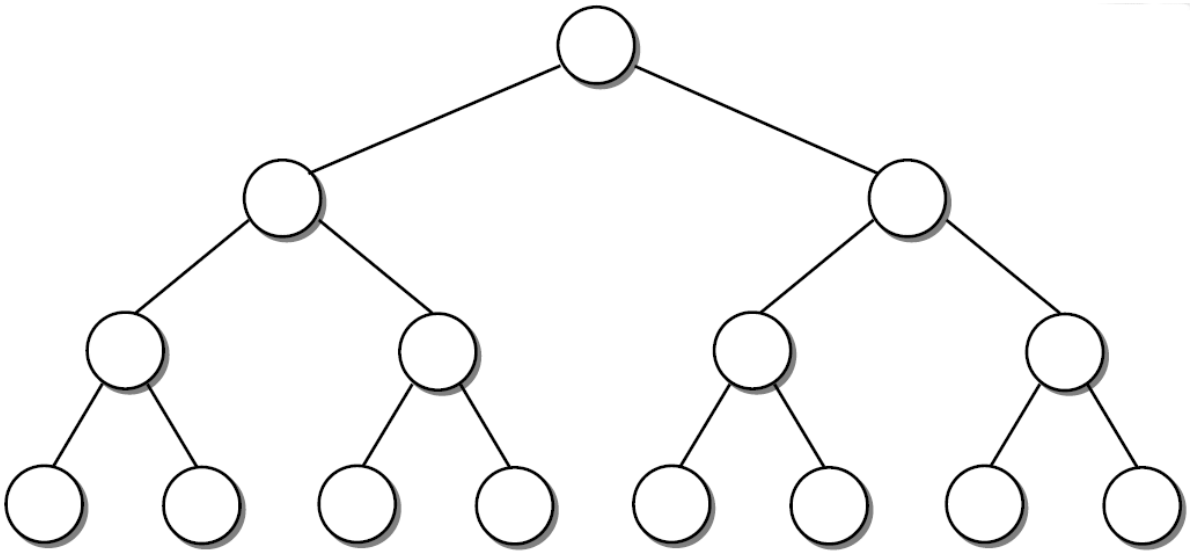
Terdapat sejumlah tipe-tipe binary tree, tiga di antaranya adalah:

- Binary Tree Penuh
- Binary Tree Sempurna
- Binary Tree Komplit

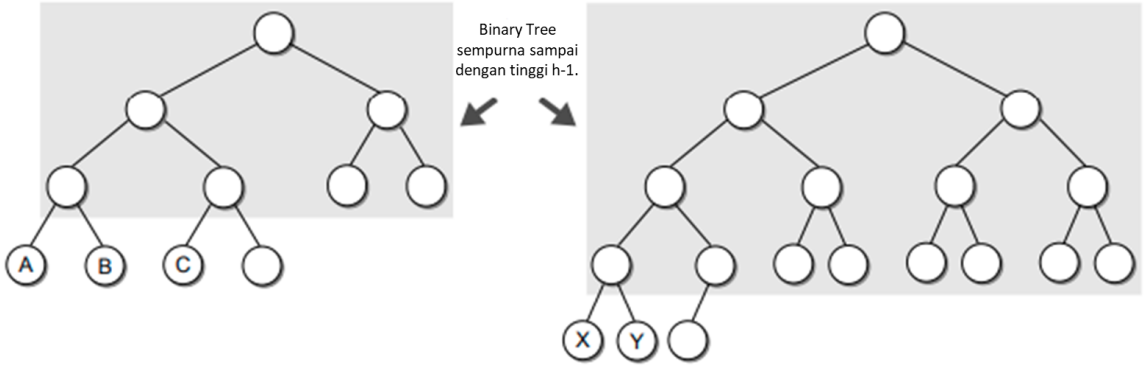
Binary tree penuh adalah binary tree yang setiap node interiornya memiliki dua node children. Binary tree penuh dapat mempunyai berbagai bentuk, seperti diilustrasikan pada gambar berikut:



Binary tree sempurna adalah binary tree penuh yang semua node leaf-nya ada pada level yang sama. Berikut ini adalah contoh dari binary tree penuh sempurna.



Binary tree komplit adalah binary tree sempurna sampai dengan satu level di atas level terbawahnya. Misalkan pada binary tree komplit dengan tinggi (h), level 0 sampai dengan level (h-1) adalah binary tree sempurna. Gambar berikut menunjukkan dua contoh binary tree komplit:



6.3 Implementasi Binary Tree

Binary tree umumnya diimplementasi sebagai struktur dinamis mirip seperti linked list. Binary tree merupakan struktur data yang dapat digunakan untuk mengimplementasikan berbagai ADT berbeda. Karena operasi-operasi yang dimiliki oleh binary tree bergantung pada aplikasinya, pada bagian ini kita tidak mendefinisikan ADT Binary Tree generik dan mengimplementasikannya ke

sebuah class yang biasanya kita lakukan pada struktur-struktur data sebelumnya. Namun, kita akan membuat dan bekerja langsung dengan binary tree.

Mendefinisikan Class `_BinTreeNode`

Untuk membuat struktur binary tree, kita harus mempunyai sebuah class untuk merepresentasikan node dalam tree. Node dalam binary tree selain mempunyai dua child: child kiri dan child kanan juga menyimpan data. Class bernama `_BinTreeNode` berikut adalah class yang merepresentasikan node dalam binary tree:

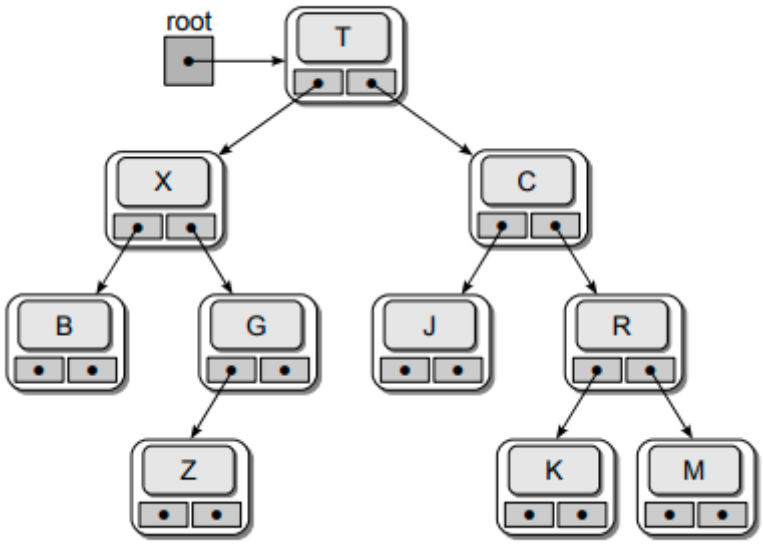
```
class _BinTreeNode :
    def __init__( self, data ):
        self.data = data
        self.left = None
        self.right = None
```

Constructor `_BinTreeNode` menerima satu argumen yang menyatakan data yang akan disimpan dalam node. *Field* `left` digunakan untuk mereferensikan node child kiri dan *field* `right` digunakan untuk mereferensikan child kanan. Nilai inisialisasi masing-masing node child tersebut adalah `None`.

Kode berikut mencontohkan pembentukan binary tree menggunakan class `_BinTreeNode`:

```
root = _BinTreeNode('T')
root.left = _BinTreeNode('X')
root.left.left = _BinTreeNode('B')
root.left.right = _BinTreeNode('G')
root.left.right.left = _BinTreeNode('Z')
root.right = _BinTreeNode('C')
root.right.left = _BinTreeNode('J')
root.right.right = _BinTreeNode('R')
root.right.right.left = _BinTreeNode('K')
root.right.right.right = _BinTreeNode('M')
```

Kode di atas akan menghasilkan struktur binary tree seperti terlihat pada gambar berikut:



Tree Traversal

Traversal binary tree adalah cara sistematis untuk mengakses atau "mengunjugi" setiap node dalam binary tree. Proses traversal ini umumnya digunakan untuk mengimplementasikan operasi-operasi pada binary tree. Operasi-operasi yang dilakukan saat mengunjungi sebuah node bergantung pada aplikasi. Operasi-operasi ini dapat berupa operasi yang sesederhana mencetak data yang disimpan node maupun operasi kalkulasi yang rumit yang melibatkan posisi node yang dikunjungi.

Proses traversal pada struktur *linear* seperti linked list mudah dilakukan. Kita dapat memulai dari node pertama dan mengiterasinya node per node secara berurut. Namun bagaimana kita mengunjungi node-node dalam binary tree? Tidak ada path tunggal dari root ke setiap node dalam tree. Keterhubungan antar node-node akan membawa ke bagian bawah tree. Jika kita mengikuti keterhubungan tersebut kita akan mencapai sebuah node leaf dan kita tidak akan mempunyai akses ke node-node lainnya dalam tree.

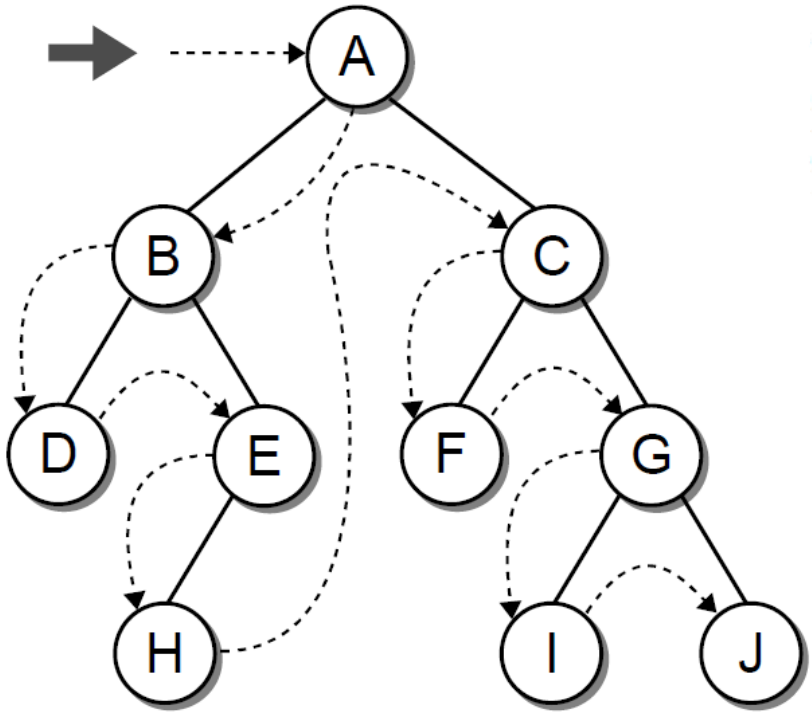
Pada bagian ini kita akan membahas skema-skema traversal yang umum dilakukan pada binary tree. Kita akan membahas empat skema traversal yaitu:

- Preorder Traversal
- Inorder Traversal
- Postorder Traversal
- Breadth-First Traversal

Preorder Traversal

Proses traversal pada tree harus dimulai dari node root, karena node root adalah satu-satunya akses masuk ke strutkur tree. Setelah mengunjungi node root, kita dapat melakukan proses traversal ke node-node pada subtree kirinya lalu ke node-node pada subtree kanannya. Karena setiap node adalah root dari subtreenya sendiri, maka kita dapat mengulangi proses yang sama pada setiap node. Sehingga proses traversal ini dapat kita lakukan secara rekursif.

Perhatikan gambar berikut:



Garis putus-putus menunjukkan alur node-node yang dikunjugi saat proses traversal: A, B, D, E, H, C, F, G, I, J. Proses traversal seperti gambar di atas disebut sebagai ***preorder traversal*** karena pertama-tama kita mengunjungi node terlebih dahulu lalu melakukan proses traversal kembali ke subtree kiri dan subtree kanannya.

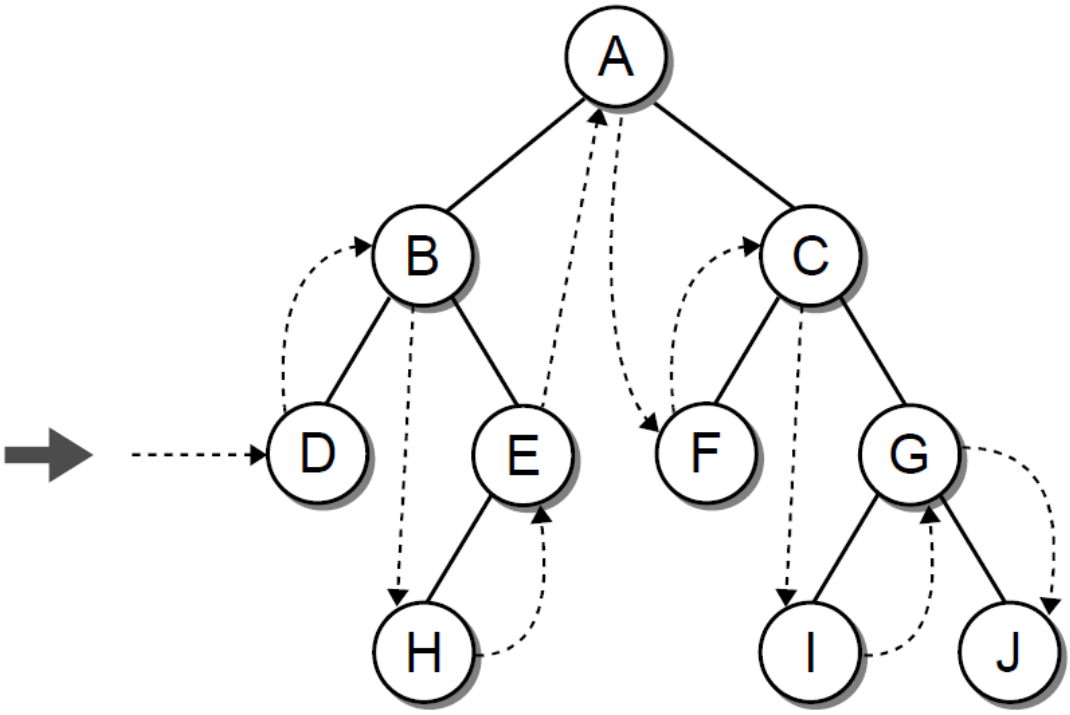
Kita dapat menuliskan fungsi rekursif yang melakukan proses preorder traversal seperti berikut:

```
def preorderTrav(subtree):
    if subtree is not None:
        print(subtree.data, end = ' ') # Kunjungi node
        preorderTrav(subtree.left)     # Panggil rekursif subtree kiri
        preorderTrav(subtree.right)    # Panggil rekursif subtree kanan
```

Fungsi di atas menerima sebuah argument `subtree` yang merupakan referensi ke node root dari subtree. Pertama, kita menguji apakah argument `subtree` bernilai `None` atau tidak. Jika argument `subtree` tidak bernilai `None` maka node root ini dikunjungi (pada fungsi ini kita mencetak data pada node yang dikunjungi) lalu kita memanggil fungsi ini secara rekursif untuk melakukan traversal pada subtree kiri dan subtree kanan dari node root ini. Argument `subtree` pada pemanggilan rekursif akan mereferensikan *null* (bernilai `None`) jika node root tidak mempunyai child kiri atau child kanan atau keduanya.

Inorder Traversal

Pada *preorder traversal*, pertama-tama kita mengunjungi node root lalu melakukan proses traversal kembali ke subtree kirinya dan ke subtree kanannya. Pada *inorder traversal*, kita melakukan traverse ke subtree kiri terlebih dahulu lalu mengunjungi node root dan kemudian melakukan traverse ke subtree kanannya. Gambar berikut menunjukkan alur node-node yang dikunjungi pada proses *inorder traversal*:



Pada gambar di atas, urutan node-node yang dikunjungi pada proses *inorder traversal* adalah: D, B, H, E, A, F, C, I, G, J.

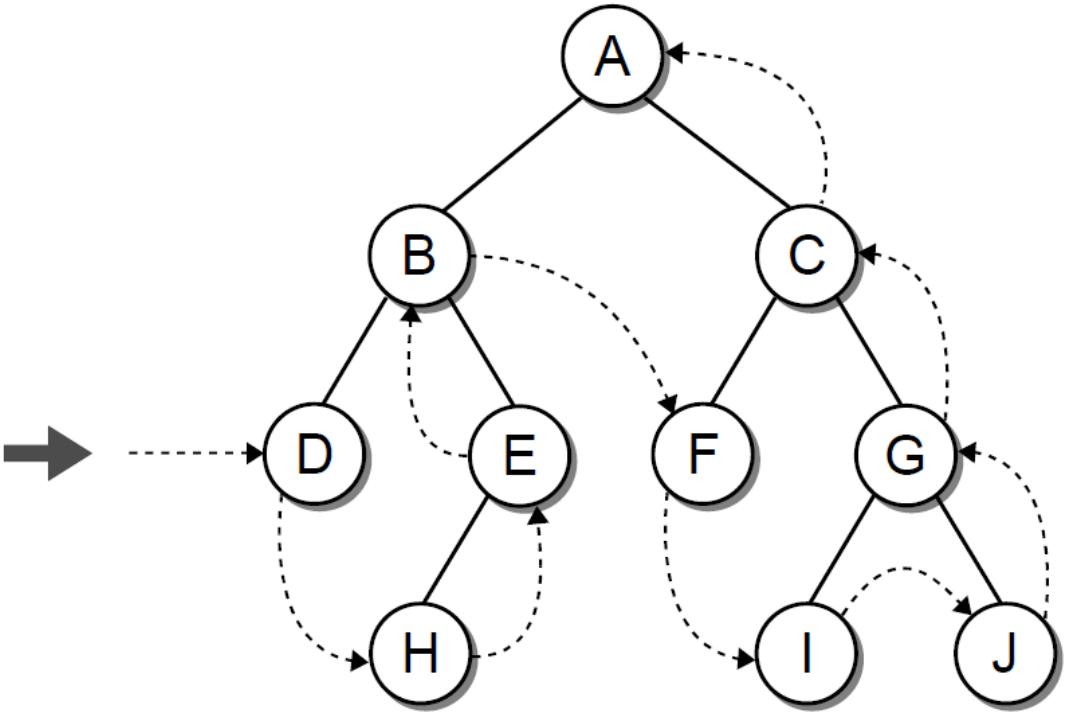
Berikut adalah fungsi proses *inorder traversal*:

```
def inorderTrav(subtree):
    if subtree is not None:
        inorderTrav(subtree.left)     # Panggil rekursif subtree kiri
        print(subtree.data, end = ' ') # Kunjungi node
        inorderTrav(subtree.right)    # Panggil rekursif subtree kanan
```

Fungsi di atas mirip dengan fungsi yang melakukan proses *preorder traversal*. Perbedaannya hanya pada urutan *statement*-nya. Pada fungsi di atas operasi mengunjungi node dilakukan setelah pemanggilan rekursif untuk subtree kiri dari node root. Lalu, pemanggilan rekursif untuk subtree kanan dilakukan setelah mengunjugi node.

Postorder Traversal

Proses ***postorder traversal***, dapat dipandang sebagai kebalikan dari proses *preorder traversal*. Pada *postorder traversal*, subtree kiri dan subtree kanan dari setiap node di-*traverse* terlebih dahulu sebelum node tersebut dikunjungi. Gambar berikut menunjukkan alur node-node yang dikunjungi pada proses *postorder traversal*:



Pada gambar di atas, urutan node-node yang dikunjungi pada proses *postorder traversal* adalah: D, H, E, B, F, I, J, G, C, A.

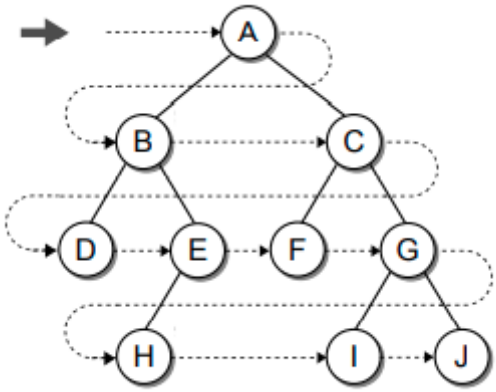
Berikut adalah fungsi proses *postorder traversal*:

```
def postorderTrav(subtree):
    if subtree is not None:
        postorderTrav(subtree.left)    # Panggil rekursif subtree kiri
        postorderTrav(subtree.right)   # Panggil rekursif subtree kanan
        print(subtree.data, end = ' ') # Kunjungi node
```

Fungsi *postorder traversal* di atas mirip dengan fungsi *preorder traversal* maupun fungsi *inorder traversal*. Perbedaannya pada urutan *statement-statementnya*. Pada fungsi *postorder traversal*, pemanggilan rekursif untuk subtree kiri dilakukan terlebih dahulu. Lalu dilanjutkan dengan pemanggilan rekursif untuk subtree kanan. Kemudian setelah kedua pemanggilan rekursif tersebut, node dikunjungi.

Breadth-First Traversal

Proses-proses *traversal preorder*, *inorder*, dan *postorder* adalah contoh-contoh dari traversal *depth-first*. Pada traversal *depth-first*, node-node pada tree di-*traverse* lebih dalam sebelum mengembalikan node-node level yang lebih tinggi. Jenis lain dari traversal yang dapat dilakukan pada binary tree adalah ***breadth-first traversal***. Pada *breadth-first traversal*, node-node dikunjungi level per level dari kiri ke kanan. Gambar berikut menunjukkan alur node-node yang dikunjungi pada proses *breadth-first traversal*:



Rekursi tidak dapat digunakan pada implementasi *breadth-first traversal* karena pemanggilan rekursif akan mengarah ke node-node yang lebih dalam. Kita harus menggunakan pendekatan lain. Langkah yang mungkin terpikir pertama kali adalah dengan mengunjungi sebuah node yang diikuti dengan dua *child*-nya. Sehingga pada contoh binary tree di atas, kita mengunjugi node A lalu dilanjutkan dengan mengunjugi node B dan C. Namun, apa yang terjadi ketika kita mengunjugi node B? Kita tidak dapat mengunjugi dua *child*-nya, D dan E, sampai kita selesai mengunjugi node C. Kita harus memikirkan suatu cara untuk mengingat kedua *child* dari B sampai dengan setelah node C dikunjungi. Begitu juga saat kita mengunjungi node C, kita harus menyimpan dua *child* dari C sampai dengan setelah kedua *child* dari B dikunjungi. Setelah mengunjungi node C, kita harus menyimpan empat node - D, E, F, dan G - yang merupakan empat node berikutnya yang akan dikunjungi dalam urutan keempat node tersebut disimpan. Cara terbaik untuk menyimpan *child-child* dari node adalah dengan menggunakan sebuah queue. Lalu, kita dapat menggunakan sebuah *loop* yang bergerak melintasi node-node dalam urutan kiri ke kanan pada setiap levelnya.

Berikut adalah fungsi proses *breadth-first traversal*:

```
def breadthFirstTrav(root):
    # Buat queue dan tambahkan node root ke dalamnya.
    q = Queue()
    q.enqueue(root)

    # Kunjungi setiap node dalam tree.
    while not q.isEmpty():
        # Hapus node berikutnya dalam queue dan kunjungi.
        node = q.dequeue()

        print(node.data, end = ' ') # Kunjungi node

        # Tambahkan child left dan child right ke queue
        if node.left is not None:
            q.enqueue(node.left)
        if node.right is not None:
            q.enqueue(node.right)
```

Kode Lengkap Implementasi Binary Tree

Berikut adalah kode lengkap dari class `_BinTreeNode` beserta fungsi-fungsi traversalnya:

Module *`binarytree.py`*

```
from linkedlistqueue import Queue     # Diperlukan untuk fungsi breadthFirstTrav

# Class _BinTreeNode merepresentasikan node dari binary tree.
# Class ini mempunyai field data untuk menyimpan data,
# field left untuk menghubungkan node ke child kirinya,
# dan field right untuk menghubungkan node ke child kanannya.
class _BinTreeNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Fungsi ini melakukan proses preorder traversal dari sebuah
# binary tree.
def preorderTrav(subtree):
    if subtree is not None:
        print(subtree.data, end = ' ') # Kunjungi node
        preorderTrav(subtree.left)     # Panggil rekursif subtree kiri
        preorderTrav(subtree.right)    # Panggil rekursif subtree kanan

# Fungsi ini melakukan proses indorder traversal dari sebuah
# binary tree.
def inorderTrav(subtree):
    if subtree is not None:
        inorderTrav(subtree.left)     # Panggil rekursif subtree kiri
        print(subtree.data, end = ' ') # Kunjungi node
        inorderTrav(subtree.right)    # Panggil rekursif subtree kanan

# Fungsi ini melakukan proses postorder traversal dari sebuah
# binary tree.
def postorderTrav(subtree):
    if subtree is not None:
        postorderTrav(subtree.left)    # Panggil rekursif subtree kiri
        postorderTrav(subtree.right)   # Panggil rekursif subtree kanan
        print(subtree.data, end = ' ') # Kunjungi node

# Fungsi ini melakukan proses breadth-first traversal dari sebuah
# binary tree.
def breadthFirstTrav(root):
    # Buat queue dan tambahkan node root ke dalamnya.
    q = Queue()
    q.enqueue(root)

    # Kunjungi setiap node dalam tree.
    while not q.isEmpty():
        # Hapus node berikutnya dalam queue dan kunjungi.
        node = q.dequeue()

        print(node.data, end = ' ') # Kunjungi node

        # Tambahkan child left dan child right ke queue
```

```
        if node.left is not None:
            q.enqueue(node.left)
        if node.right is not None:
            q.enqueue(node.right)
```

Menguji Implementasi Binary Tree

Kode berikut membangun binary tree yang kita gunakan sebagai contoh pada implementasi fungsi-fungsi traversal lalu memanggil empat fungsi traversal.

```
# Bangun sebuah binary tree sesuai gambar modul tulis
root = _BinTreeNode('A')

root.left = _BinTreeNode('B')
root.left.left = _BinTreeNode('D')
root.left.right = _BinTreeNode('E')
root.left.right.left = _BinTreeNode('H')
root.right = _BinTreeNode('C')
root.right.left = _BinTreeNode('F')
root.right.right = _BinTreeNode('G')
root.right.right.left = _BinTreeNode('I')
root.right.right.right = _BinTreeNode('J')

# Preorder traversal
print('Preorder traversal:')
preorderTrav(root)
print()

# Inorder traversal
print('Inorder traversal:')
inorderTrav(root)
print()

# Postorder traversal
print('Postorder traversal')
postorderTrav(root)
print()

# Breadth-First traversal
print('Breadth-First traversal:')
breadthFirstTrav(root)
```

Output dari kode di atas:

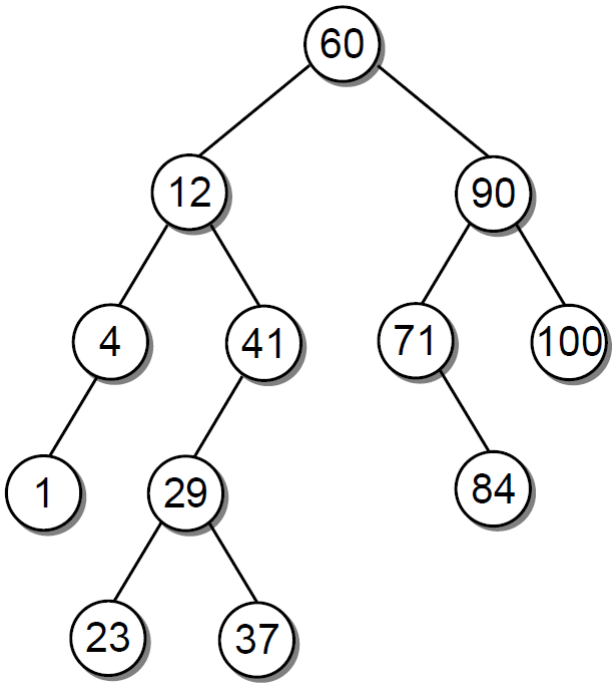
```
Preorder traversal:
A B D E H C F G I J
Inorder traversal:
D B H E A F C I G J
Postorder traversal
D H E B F I J G C A
Breadth-First traversal:
A B C D E F G H I J
```

6.4 Binary Search Tree

Binary search tree adalah binary tree yang setiap nodenya memiliki sebuah nilai dan node-node tersebut disusun berdasarkan nilainya dengan ketentuan-ketentuan berikut:

- Node-node yang bernilai lebih kecil dari suatu node dalam tree berada di subtree kiri.
- Node-node yang bernilai lebih besar dari suatu node dalam tree berada di subtree kanan.
- Tidak ada nilai duplikat pada node-node.

Gambar berikut adalah sebuah contoh struktur *binary search tree*:



Perhatikan pada gambar di atas, node root bernilai 60. Semua node-node pada subtree kirinya memiliki nilai lebih kecil daripada 60 dan semua node-node pada subtree kanannya memiliki nilai lebih besar daripada 60. Relasi yang sama juga berlaku ke setiap node di subtree-subtree.

Operasi-operasi yang terdapat dalam ADT *Binary Search Tree*:

- Operasi `length()`: mengembalikan ukuran dari binary search tree. Diakses menggunakan fungsi `len()`.
- Operasi `contains(data)`: mencari apakah `data` terdapat di dalam binary search tree. Diakses menggunakan operator `in`.
- Operasi `min()`: mengembalikan nilai minimum dalam binary search tree.
- Operasi `max()`: mengembalikan nilai maksimum dalam binary search tree.
- Operasi `add(data)`: memasukkan `data` ke dalam binary search tree.
- Operasi `remove(data)`: menghapus node yang berisi `data` dalam binary search tree.

Implementasi ADT Binary Search Tree

Implementasi parsial dari ADT Binary Search Tree dapat dilihat pada kode berikut:

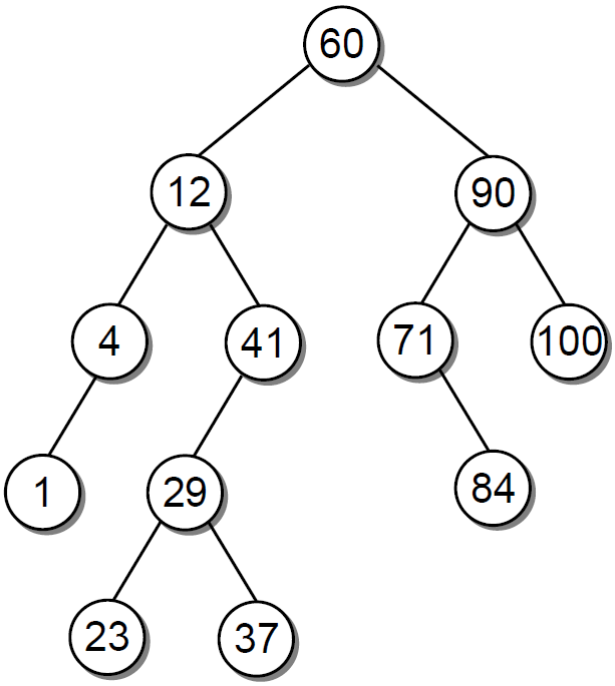
```
class BST:
    # Constructor untuk membuat binary search tree baru
    # Field _root: menyimpan referensi ke node root dari tree.
    # Field _size: menyimpan ukuran tree (banyaknya node).
    def __init__(self):
        self._root = None
        self._size = 0

    # Method __len__ mengembalikan ukuran (nilai field _size)
    # dari binary search tree.
    # Method ini diakses menggunakan fungsi len().
    def __len__(self):
        return self._size

# Class _BSTNode merepresentasikan node dalam binary search tree.
class _BSTNode:
    # Constructor untuk membuat node binary search tree baru.
    # Field data: Menyimpan nilai pada node.
    # Field left: Referensi ke node child kiri.
    # Field right: Referensi ke node child kanan.
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

Kita akan menambahkan operasi-operasi lain pada bagian-bagian setelah ini. Constructor pada class `BST` di atas mendefinisikan *field* `_root` untuk mereferensikan node root dan *field* `_size` untuk menyimpan ukuran (banyaknya elemen) dari binary search tree. *Field* `_size` dibutuhkan oleh *method* `__len__` yang mengembalikan ukuran binary search tree. Class `_BSTNode` yang didefinisikan pada baris 20 sampai 28 adalah class *private* yang digunakan untuk merepresentasikan node-node binary search tree.

Method `__contains__` (Pencarian Node)

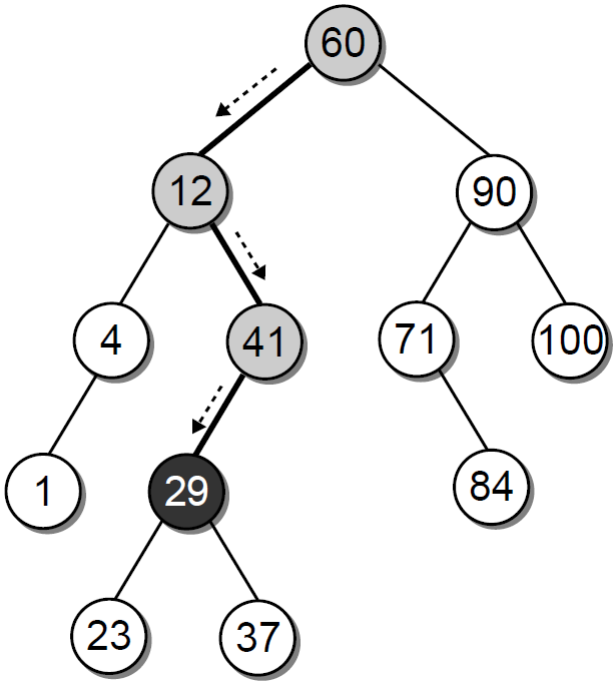
Pencarian suatu nilai pada binary search tree dimulai dari node root. Kita membandingkan nilai target yang dicari dengan nilai pada node root. Jika nilai target yang dicari sama dengan nilai pada root node, pencarian selesai. Jika nilai target tidak sama maka kita harus memutuskan jalur mana yang kita tempuh: ke child kiri atau ke child kanan. Dari definisi binary search tree, kita mengetahui bahwa nilai dalam node root lebih besar daripada nilai node-node pada subtree kirinya dan lebih kecil daripada nilai node-node pada subtree kananya. Sehingga, jika nilai target lebih kecil kita bergerak ke subtree kiri dan jika nilai target lebih besar kita bergerak ke kanan. Kita mengulangi perbandingan pada node root dari subtree dan mengambil jalur yang sesuai. Proses ini berulang sampai nilai target ditemukan atau sampai kita mendapatkan *child null* (yang berarti nilai target tidak ditemukan).

Sebagai contoh, misalkan kita mempunyai binary search tree seperti berikut:



Kita ingin melakukan pencarian nilai 29 dalam binary search tree di atas. Kita memulai pencarian dengan membandingkan nilai target (29) dengan nilai node root (60). Karena nilai target lebih kecil daripada nilai node root (29 < 60), kita bergerak ke kiri, ke node dengan nilai 12. Nilai target kita bandingkan dengan 12. Karena nilai target lebih besar dari 12 (29 > 12), kali ini kita bergerak ke kanan. Kita membandingkan nilai target dengan 41. Karena nilai target lebih kecil dari 41 (29 < 41), kita bergerak ke kiri. Akhirnya ketika kita melihat nilai pada *child* kiri dari node yang menyimpan nilai 41, kita menemukan nilai target dan menyelesaikan pencarian dengan sukses.

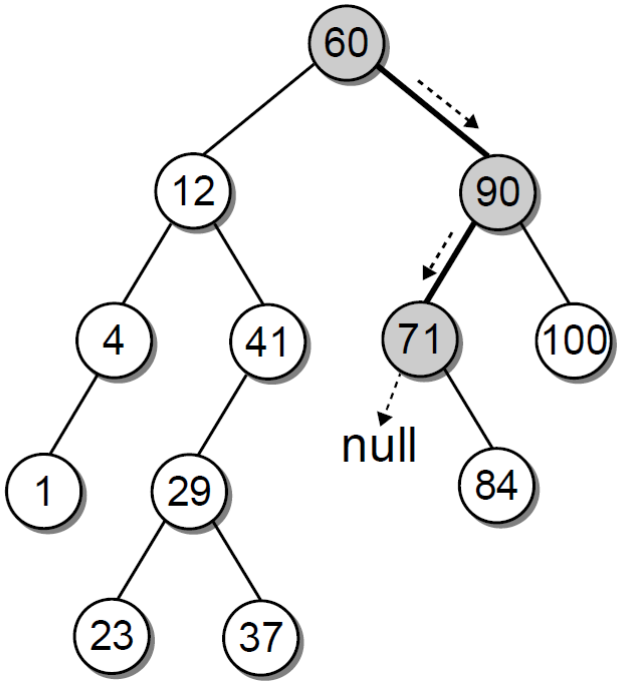
Gambar berikut mengilustrasikan jalur yang ditempuh saat melakukan pencarian nilai 29 pada contoh binary search tree:



Sekarang misalkan kita mencari nilai yang tidak ada di dalam tree. Misalkan kita ingin mencari nilai 67 pada contoh binary search tree di atas. Proses pencarian ini sama seperti saat kita melakukan pencarian nilai 29. Perbedaannya adalah ketika kita sampai node dengan nilai 71 dan membandingkannya dengan nilai target, seharusnya perbandingan dilanjutkan ke *child* kiri dari node tersebut. Namun, karena node ini tidak mempunyai *child* kiri, maka kita akan mencapai

node *null*. Sehingga, ketika pencarian berakhir ke node *null*, maka ini menandakan pencarian tersebut tidak berhasil atau nilai target tidak ditemukan dalam tree.

Gambar berikut mengilustrasikan proses pencarian nilai 67 pada contoh binary search tree:



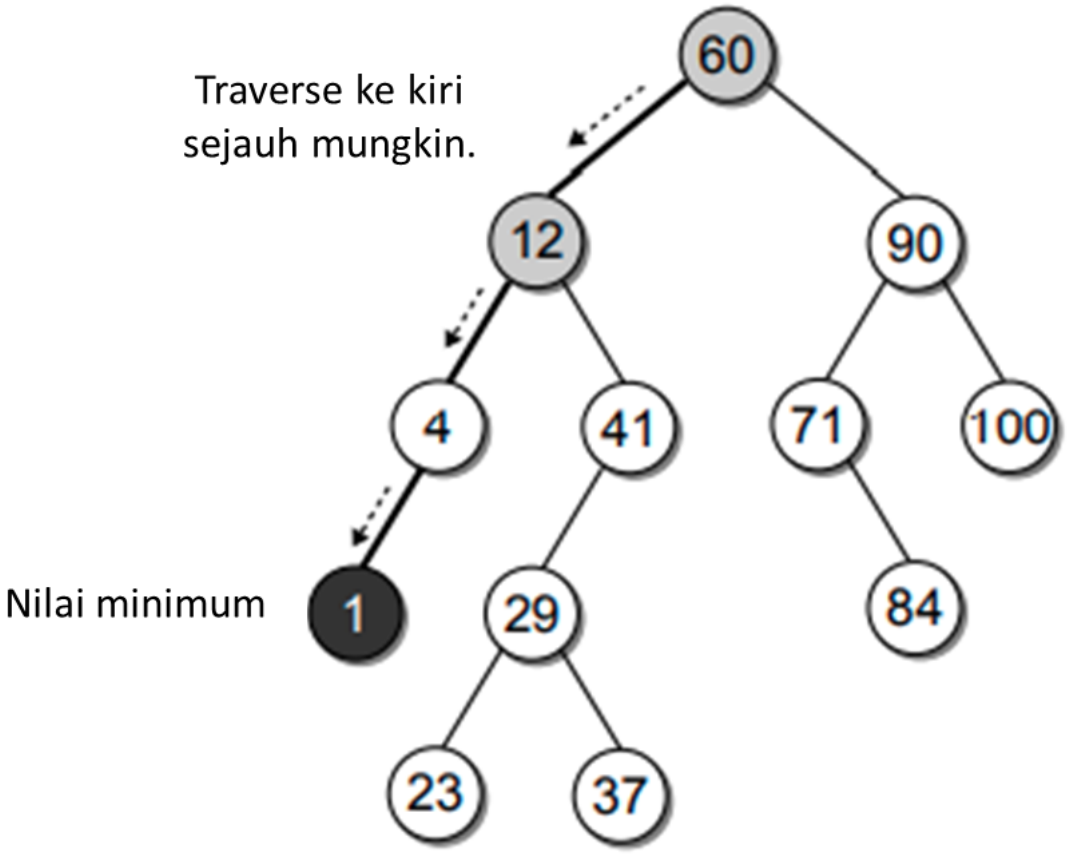
Implementasi Pencarian pada ADT Binary Search Tree

```
class BST:
    # ... Hanya method-method yang terkait operasi pencarian yang ditampilkan.

    # Method __contains__ (diakses dengan operator in) menerima
    # sebuah nilai target yang dicari.
    # Method ini mengembalikan true jika nilai target ditemukan
    # dan mengembalikan false jika nilai target tidak ditemukan.
    # Method ini menggunakan method bantu _bstSearch.
    def __contains__(self, nilaiDicari):
        return self._bstSearch(self._root, nilaiDicari) is not None

    # Method _bstSearch adalah method bantu yang secara rekursif
    # mencari nilai target dalam tree.
    def _bstSearch(self, subtree, target):
        if subtree is None :
            return None
        elif target < subtree.data: # Target berada di subtree kiri.
            return self._bstSearch(subtree.left, target)
        elif target > subtree.data: # Target berada di subtree kanan
            return self._bstSearch(subtree.right, target)
        else:
            return subtree
```

Method `min()` dan method `max()` (Nilai Minimum dan Maksimum)

Nilai minimum dalam binary search tree adalah nilai dari node paling kiri dari binary search tree. Sedangkan nilai maksimum dalam binary search tree adalah nilai dari node paling kanan dari binary search tree.

Gambar berikut mengilustrasikan proses pencarian nilai minimum dalam binary search tree:



Untuk mencari nilai minimum dalam binary search tree, kita hanya perlu mencari node paling kiri. Kita melakukannya dengan meng-*traverse* ke arah kiri sejauh mungkin. Kode berikut adalah implementasi dari *method* `min()`:

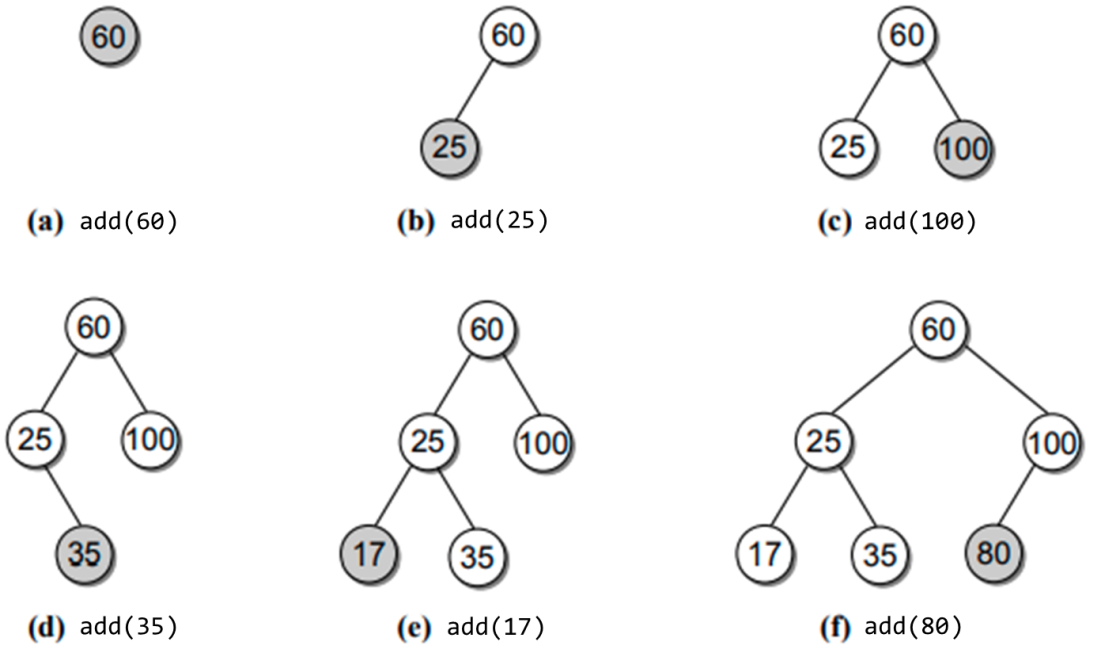
```
class BST:
    #... Hanya kode yang terkait method min saja yang ditampilkan.

    # Method min mengembalikan nilai minimum dalam binary search tree
    def min(self):
        nodeMinimum = self._bstMinimum(self._root)
        if nodeMinimum is None:
            return 0
        return nodeMinimum.data

    # Method _bstMinimum melakukan pencarian node paling kiri dengan
    # melakukan traverse arah kiri secara rekursif.
    def _bstMinimum(self, subtree):
        if subtree is None:
            return None
        elif subtree.left is None:
            return subtree
        else:
            return self._bstMinimum(subtree.left)
```

Method `_bstMinimum` adalah *method* bantu untuk mencari node paling kiri dalam binary search tree. *Method* ini membutuhkan root dari tree atau root dari subree sebagai argument. Method ini mengembalikan sebuah referensi ke node paling kiri (node dengan nilai minimum) atau mengembalikan `none` jika tree kosong.

Cara yang sama juga digunakan untuk mencari nilai maksimum dalam binary search tree. Perbedaannya adalah karena nilai maksimum berada di node paling kanan maka proses *traverse* dilakukan ke arah kanan. Implementasi dari *method* `max()` dijadikan latihan.

Method `add(data)` (Penambahan Data)

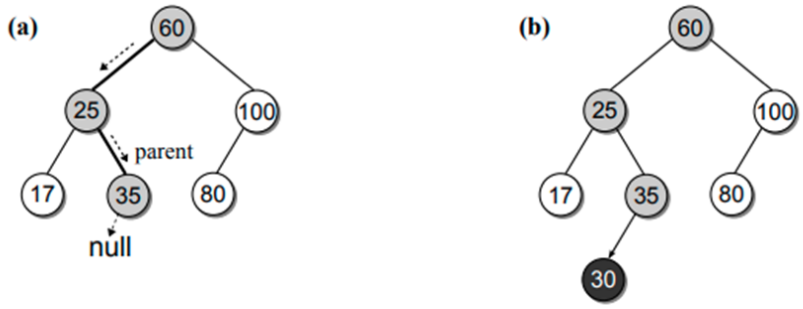
Binary search tree dibangun dengan memasukkan data satu per satu. Saat data-data dimasukkan, sebuah node baru dibuat untuk setiap data dan dihubungkan ke posisi yang sesuai pada tree. Misalkan kita membangun sebuah binary search tree dengan daftar data berikut: `[60, 25, 100, 35, 17, 80]`. Node-node yang ditambahkan sesuai dengan urutannya pada daftar. Gambar berikut mengilustrasikan langkah-langkah penambahan data-data ini:



Penjelasan gambar di atas adalah sebagai berikut:

- **(a)** Kita memulai dengan memasukkan nilai 60. Sebuah node dibuat dan *field* datanya ditetapkan dengan nilai tersebut. Karena kita memulai dari tree kosong, maka node pertama menjadi root dari tree.
- **(b)** Kemudian, kita memasukkan nilai 25. Karena 25 lebih kecil dari maka node yang berisi nilai ini ditempatkan pada posisi node *child* kiri dari node root.
- **(c)** Nilai 100 ditambahkan ke *child* kanan dari root karena nilai ini lebih besar daripada 60 (nilai root).
- **(d)** Ketika kita memasukkan nilai 35 ke tree, kita tidak mengubah posisi dari node-node yang sudah ada di tree. Nilai 35 dimasukkan ke node yang merupakan *child* kanan dari node yang menyimpan nilai 25.
- **(e)** Sama seperti sebelumnya, nilai 17 dimasukkan ke tree pada posisi *child* kiri dari node yang menyimpan nilai 25.
- **(f)** Terakhir, nilai 80 dimasukkan ke tree pada posisi *child* kiri dari node yang menyimpan nilai 100.

Sekarang misalkan kita ingin menambahkan nilai 30 ke binary search tree yang telah dibangun di atas. Bagaimana caranya kita menuliskan kode program untuk penambahan data ini? Kita harus terlebih dahulu mencari node yang akan dihubungkan dengan node baru bernilai 30. Misalkan kita melakukan pencarian dengan *method* `_bstSearch` untuk nilai 30, maka proses pencarian ini akan menuju ke node dengan nilai 35 sebelum akhirnya bergerak ke *child* kiri dari node tersebut dan menghasilkan *null*. Perhatikan bahwa node dengan nilai 35 ini adalah lokasi dimana node baru dimasukkan. Gambar berikut mengilustrasikan kemiripan proses pencarian nilai dengan penambahan data:

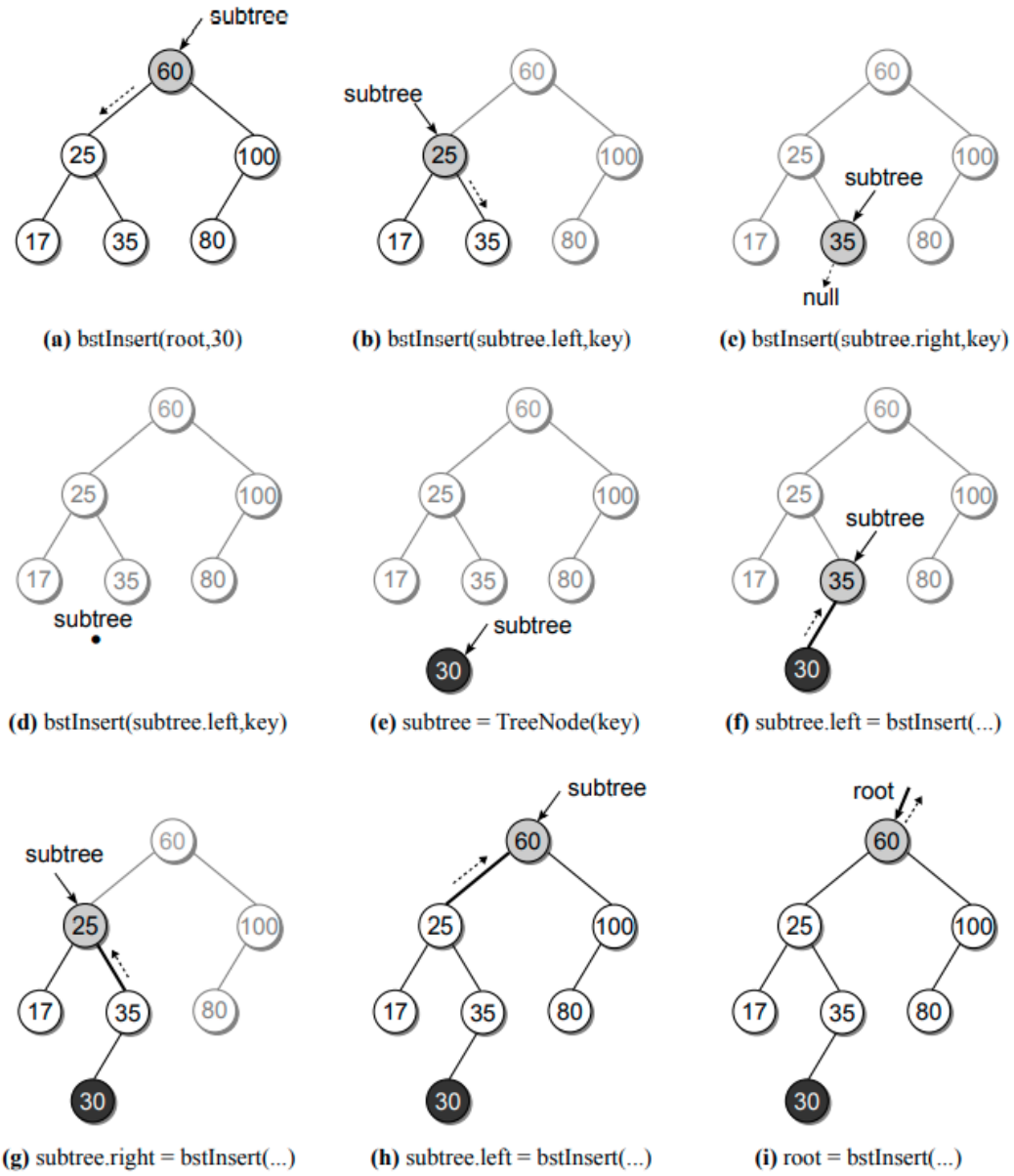


Menambahkan sebuah node baru ke binary search tree: (a) cari lokasi node dan (b) hubungkan node baru ke tree.

Kita dapat memodifikasi operasi pencarian untuk melakukan penambahan data ke binary search tree. Kode berikut adalah implementasi untuk menambahkan data baru dalam binary search tree:

```
class BST:
    #... Hanya kode yang terkait method add saja yang ditampilkan.

    # Method add menambahkan node berisi data baru ke binary search tree.
    # Method ini mengembalikan True jika data berhasil ditambahkan dan
    # mengembalikan False jika data telah ada di binary search tree.
    def add(self, data):
        # Cari terlebih dahulu data di dalam tree.
        node = self._bstSearch(self._root, data)
        # Jika terdapat data dalam tree kembalikan False.
        # Jika data tidak ada dalam tree, masukkan data ke tree dengan
        # memanggil method bantu _btsInsert.
        if node is not None:
            return False
        else:
            self._root = self._bstInsert(self._root, data)
            self._size += 1
            return True

    # Method _btsInsert adalah method bantu untuk memasukkan node baru
    # dalam tree secara rekursif.
    def _bstInsert(self, subtree, data):
        # Jika subtree kosong
        if subtree is None:
            subtree = _BSTNode(data)
        elif (data < subtree.data):
            subtree.left = self._bstInsert(subtree.left, data)
        elif (data > subtree.data):
            subtree.right = self._bstInsert(subtree.right, data)
        return subtree
```

Gamabar berikut mengilustrasikan apa yang diproses oleh *method* `_bstInsert` setiap pemanggilan rekursi dan menunjukkan perubahan pada tree saat *statement-statement* dalam *method* tersebut dieksekusi:



Gambar (a) sampai dengan (c) menunjukkan langkah-langkah rekursif dari proses pencarian untuk mendapatkan node yang akan dihubungkan dengan node baru. Node yang diwarnai abu-abu adalah node yang sedang diproses oleh pemanggilan *method* `_bstInsert`. Garis putus-putus menandakan arah yang harus diikuti untuk mendapatkan jalur yang tepat dan juga menandakan jalur kembali dari pemanggilan rekursif. Kasus base rekursif dicapai ketika subtree kosong didapatkan setelah mengambil jalur ke *child* kiri dari node 35 (gambar (d)). Sebuah node baru dibuat dan *field* data dari node baru tersebut ditetapkan dengan nilai 30 (gambar (e)). Referensi ke node baru ini kemudian dikembalikan dan proses perjalanan kembali rekursi dimulai. Langkah pertama dalam perjalanan kembali adalah ke node 35. Referensi ke node baru ini dikembalikan dan ditugaskan ke *field* `left` dari node `subtree`, yang menghasilkan node baru dihubungkan ke tree. Seiring perjalanan kembali rekursi, ditunjukkan pada gambar (f) sampai dengan (i), referensi node `subtree` dikembalikan dan dihubungkan kembali ke node *parent*nya. Ini tidak mengubah struktur tree karena referensi yang sama ditugaskan kembali. Cara ini diperlukan untuk menambahkan node baru ke *parent*-nya dan juga untuk memungkinkan penambahan node baru ke tree kosong.

Method `remove(data)`

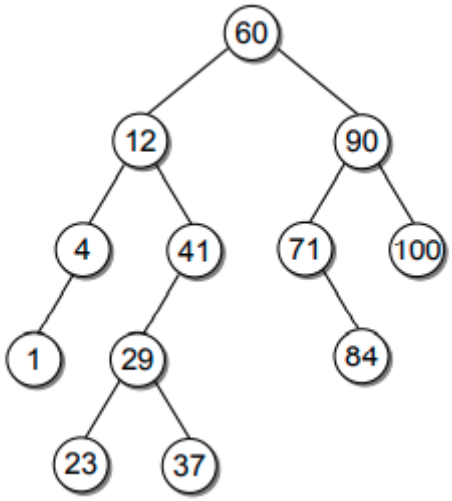
Menghapus sebuah elemen dari binary search tree adalah operasi yang paling rumit diantara operasi-operasi lainnya. Penghapusan node melibatkan pencarian untuk node yang berisi nilai target dan lalu memutuskan link dari node tersebut untuk menghapusnya dari tree. Ketika sebuah node dihapus, node-node tersisa harus mempertahankan sifat binary search tree. Terdapat tiga kasus yang perlu diperhatikan ketika node yang dicari telah ditemukan:

1. Node yang dihapus adalah node *leaf*
2. Node yang dihapus memiliki satu node *child*
3. Node yang dihapus memiliki dua node *child*

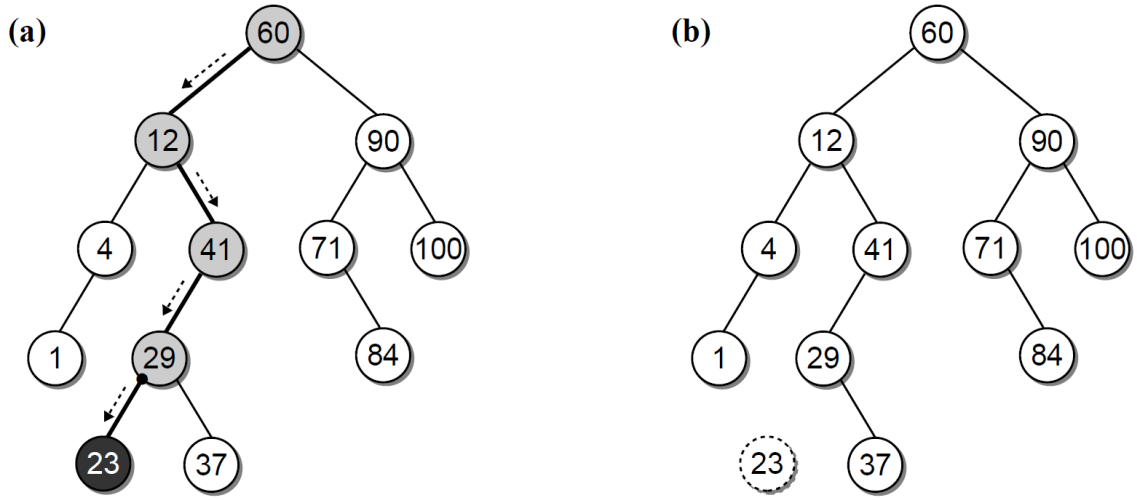
Langkah pertama untuk menghapus sebuah elemen adalah untuk menemukan node yang menyimpan data yang ingin dihapus. Ini dapat dilakukan dengan cara yang sama dengan yang kita gunakan ketika mencari lokasi untuk memasukkan elemen baru. Setelah lokasi node ditemukan, node tersebut harus dilepaskan untuk menghapusnya dari tree. Kita akan membahas tiga kasus di atas satu per satu dan mengimplementasikannya ke dalam *method* bantu rekursif `_btsRemove()`. *Method* `_bstRemove()` digunakan untuk mengimplementasikan *method* `remove()`.

Kasus 1: Menghapus Node Leaf

Menghapus node *leaf* adalah kasus yang paling mudah dari ketiga kasus. Misalkan kita ingin menghapus nilai 23 dari binary search tree pada gambar berikut:



Pertama yang kita lakukan adalah mencari node dengan nilai 23 dan memutus *link*-nya dengan node *parent*nya seperti dapat dilihat pada bagian (a) dari gambar di bawah. Untuk menghapus node *leaf* kita hanya perlu mengembalikan referensi *null*. Untuk kasus ini, kita dapat menggunakan teknik yang sama yang mengembalikan referensi dari setiap pemanggilan rekursif seperti pada operasi penambahan node. Dengan mengembalikan `None` ke node *parent*, sebuah referensi *null* akan ditugaskan ke *field link* yang tepat dalam node *parent*, yang berarti memutus node tersebut dari tree. Ini diilustrasikan pada bagian (b) dari gambar di bawah.

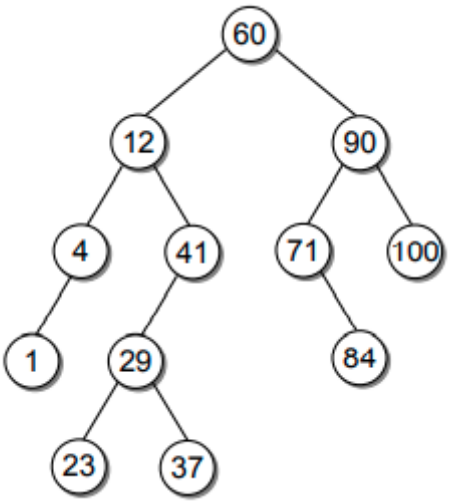


Potongan kode dari *method* `_bstRemove` untuk kasus ini dapat dituliskan berikut:

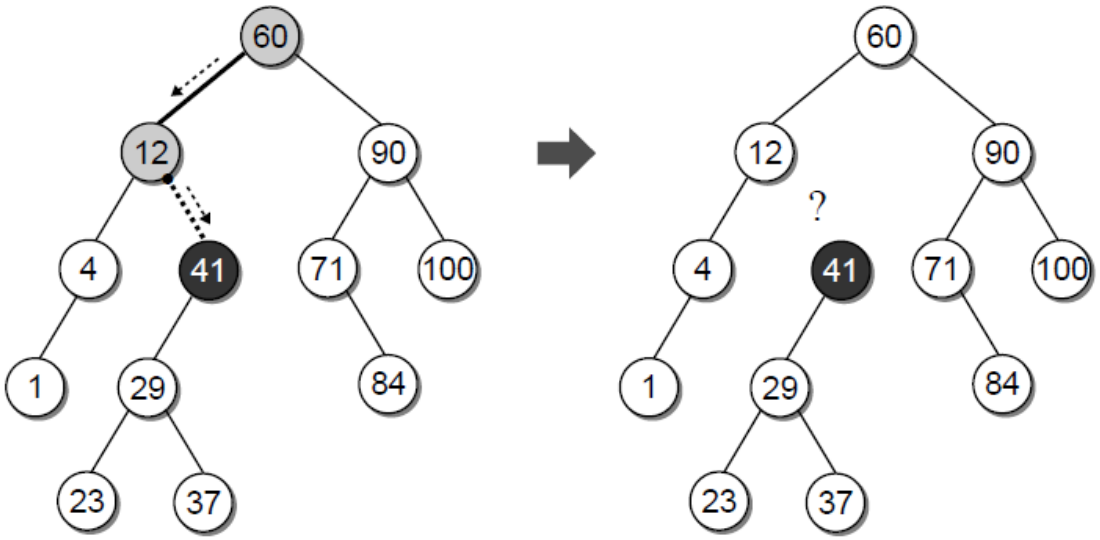
```
def _bstRemove(self, subtree, target):
    # Cari target (data dari node yang ingin dihapus) dalam tree.
    if subtree is None:
        return subtree
    elif target < subtree.data:
        subtree.left = self._bstRemove(subtree.left, target)
        return subtree
    elif target > subtree.data:
        subtree.right = self._bstRemove(subtree.right, target)
        return subtree
    # Node target ditemukan
    else:
        # Kasus 1: Node target adalah node leaf
        if subtree.left is None and subtree.right is None:
            return None

        # ... Kode Kasus 2 dan Kasus 3 tidak ditampilkan
```

Kasus 2: Menghapus Node Interior dengan Satu Node Child

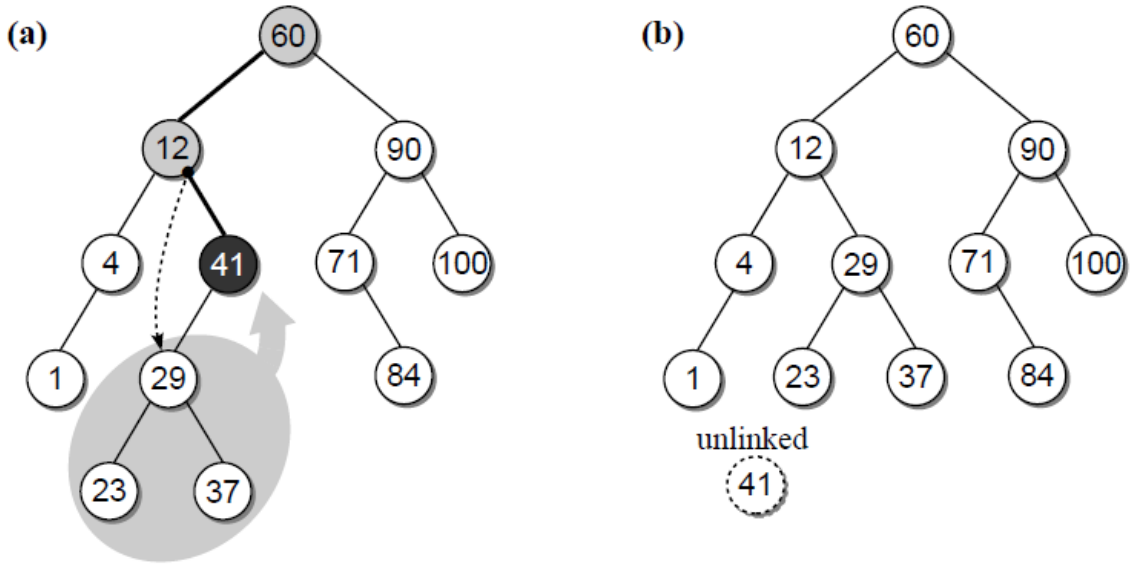
Jika node yang ingin dihapus mempunyai satu *child*, *child* tersebut dapat berupa *child* kiri ataupun *child* kanan. Misalkan kita ingin menghapus nilai 41 dari binary search tree pada gambar berikut:



Node yang berisi nilai 41 mempunyai satu subtree yang terhubung sebagai *child* kiri. Jika kita hanya mengembalikan `None` ke *parent* (Node dengan nilai 12) seperti yang kita lakukan pada kasus 1, tidak hanya node 41 yang dihapus, tetapi kita juga menghapus semua *descendant* (keturunannya) seperti dapat dilihat pada gambar berikut:



Untuk menghapus node 41, kita harus melakukan sesuatu terhadap *descendantnya*. Karena node 41 mempunyai satu *child*, maka hanya terdapat dua kemungkinan: semua *descendatnya* memiliki nilai-nilai kurang dari 41 atau semua *descendantnya* memiliki nilai-nilai lebih dari 41. Pada gambar di atas dapat dilihat, *descendant* dari node 41 terhubung ke *child* kiri, sehingga semua *descendantnya* mempunyai nilai-nilai kurang dari 41. Selain itu, karena node 41 adalah *child* kanan dari node 12, semua *descendantnya* pasti bernilai lebih dari 12. Oleh karena ini kita dapat menghubungkan langsung node 12 dengan *descendant* dari node 41 melalui *child* kanan dari node 12, seperti terlihat pada gambar berikut:



Node 29 sekarang menjadi *child* kanan dari node 12 dan semua *descendant* dari node 41 akan terhubung dengan benar tanpa kehilangan node-node lain.

Untuk melakukan penghapusan node dengan satu *child* ini dalam *method recursive*, kita hanya perlu mengubah *link* dari *field child* node *parent* untuk mereferensikan *child* dari node yang dihapus. Pemilihan *field child* dari node *parent* dilakukan secara otomatis oleh penugasan saat perjalanan kembali dari pemanggilan rekursif. Hal yang perlu kita lakukan adalah mengembalikan

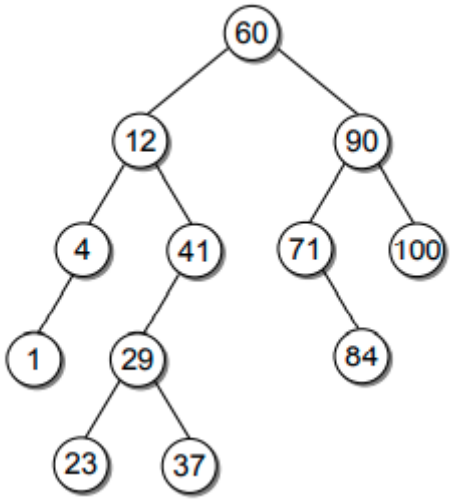
referensi ke *child* dari node yang dihapus.

Potongan kode dari *method* `_bstRemove` untuk kasus 2 ini dapat dituliskan berikut:

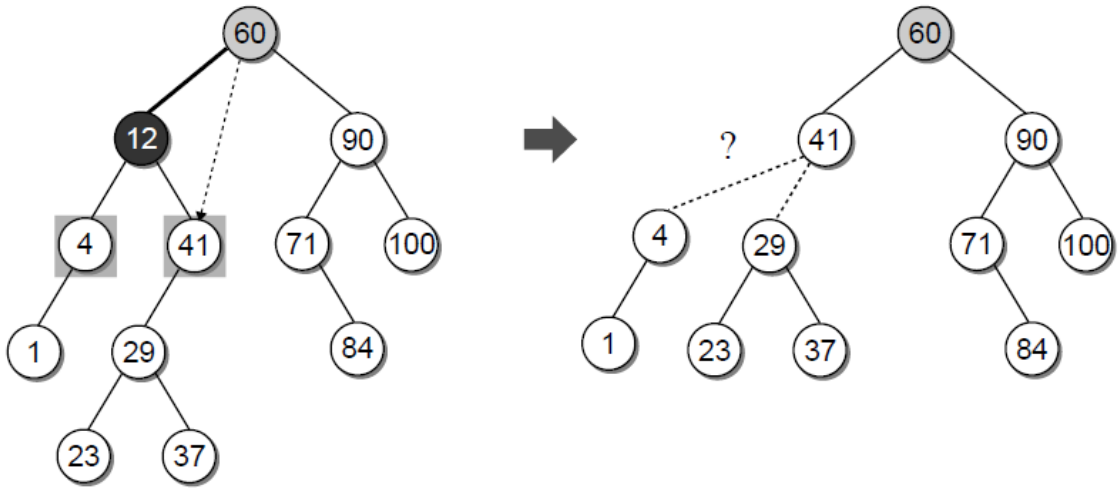
```
def _bstRemove(self, subtree, target):
    # Cari target (data dari node yang ingin dihapus) dalam tree.
    if subtree is None:
        return subtree
    elif target < subtree.data:
        subtree.left = self._bstRemove(subtree.left, target)
        return subtree
    elif target > subtree.data:
        subtree.right = self._bstRemove(subtree.right, target)
        return subtree
    # Node target ditemukan
    else:
        # Kasus 1: Node target adalah node leaf
        if subtree.left is None and subtree.right is None:
            return None
        # Kasus 2: Node target mempunyai satu child
        if subtree.left is None or subtree.right is None:
            # Kembalikan child dari node target yang dihapus
            # Jika descendant berada di child kiri, kembalikan child kiri.
            if subtree.left is not None:
                return subtree.left
            # Jika descendant berada di child kanan, kembalikan child kanan.
            else:
                return subtree.right

        # ... Kode Kasus 3 tidak ditampilkan
```

Kasus 3: Menghapus Node Interior dengan Dua Node Child

Kasus ini adalah kasus yang paling rumit diantara ketiga kasus. Ebagai contoh misalkan kita ingin menghapus node 12 dari binary search tree dalam gambar berikut:

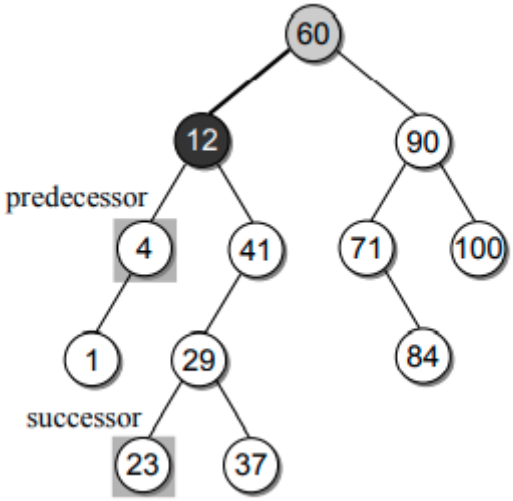


Node 12 mempunyai dua *child* dan kedua *child* tersebut adalah root dari subtreenya. Jika kita menggunakan pendekatan yang sama seperti yang digunakan saat kita menghapus node dengan satu *child*, *child* mana yang kita pilih untuk menggantikan *parent* dan apa yang terjadi pada *child* lain dan subtreeenya? Gambar berikut mengilustrasikan hasil dari menggantikan node 12 dengan *child* kanannya:



Ini akan membuat subtree kiri dari node yang dihapus tidak terhubung ke tree sehingga akan menghapus subtree tersebut dari tree. Kita dapat saja menghubungkan child kiri dan subtreenya sebagai child kiri dari node 23. Namun, ini akan membuat tree semakin tinggi dan menyebabkan operasi tree menjadi kurang efisien.

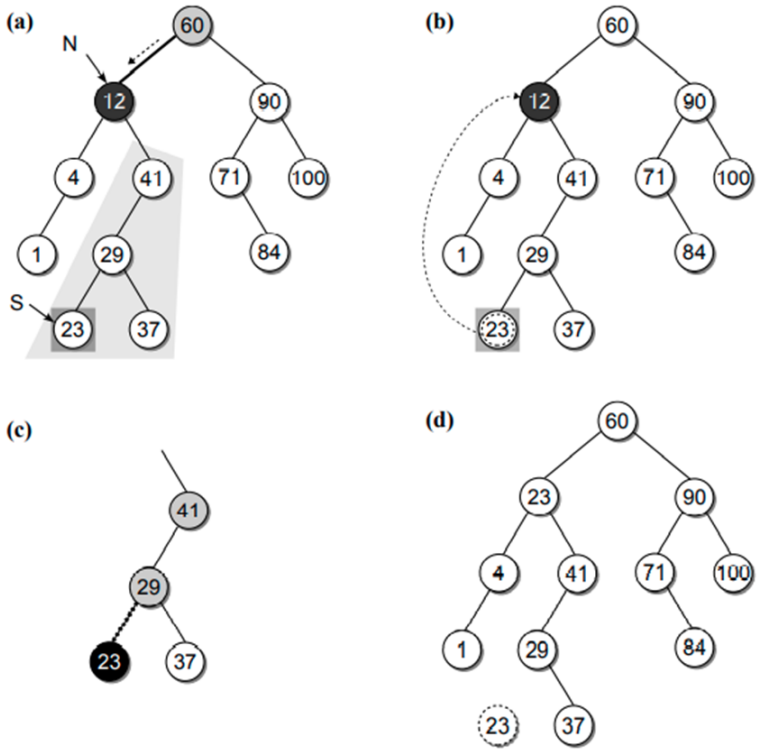
Sifat dari binary search tree mempunyai implikasi bahwa proses *inorder traversal* yang dilakukan pada binary search tree akan menghasilkan rangkaian nilai-nilai yang terurut. Sehingga, setiap node dalam binary search tree mempunyai ***predecessor*** (pendahulu) dan ***successor*** (penerus/pengganti). Untuk node 12, *predecessor*-nya adalah node 4 dan *successor*-nya adalah node 23, seperti dapat dilihat pada gambar berikut:



Ketimbang mencoba menggantikan node dengan salah satu dari kedua child-nya, kita dapat memilih predecessor atau successor dari node yang dihapus untuk menggantikan node tersebut. Menggantikan node yang dihapus dengan successor merupakan cara yang lebih mudah karena node successor dari sebuah node dengan dua child adalah node dengan nilai terkecil dari subtree kanan. Maka dapat disimpulkan, node successor dari node dengan dua child adalah node paling kiri dari subtree kanan dan hanya dapat berupa node leaf atau node dengan child kanan saja. Sehingga, proses penghapusan sebuah node interior dengan dua child dapat dilakukan dengan tiga langkah berikut:

- Cari node successor, S, dari node yang ingin dihapus, N.
- Salin nilai dari node S ke node N.
- Hapus node S dari tree.

Gambar berikut mengilustrasikan langkah-langkah di atas:



Langkah-Langkah menghapus nilai dalam binary search tree: (a) cari node, *N*, dan successornya, *S*; (b) salin nilai successor dari node *N* ke *S*; (c) buang node successor dari subtree kanan node *N*; dan (d) tree setelah menghapus nilai 12.

Langkah pertama, pencarian node successor (node S) dapat kita lakukan dengan memanggil method `_bstMinimum()` dengan argument subtree kanan dari node yang ingin dihapus (node N). Langkah kedua, kita lakukan dengan menyalin nilai field `data` dari node successor (node S) ke node yang ingin dihapus (node N). Sehingga setelah langkah kedua ini, node N yang sebelumnya mempunyai nilai data yang dihapus sekarang akan mempunyai nilai sama seperti nilai pada node successor. Pada langkah ketiga, karena method `_bstRemove()` telah mempunyai cara untuk menghapus node leaf (kasus 1) dan untuk menghapus node dengan satu child (kasus 2), maka kita dapat memanggil method `_bstRemove()` dengan argument node root dari subtree kanan. Referensi yang dikembalikan dari method `_bstRemove` pada langkah ketiga ini kita tugaskan ke child kanan dari node N untuk mempertahankan struktur subtree kanan.

Potongan kode dari method `_bstRemove` untuk kasus 3 ini dapat dituliskan berikut:

```
def _bstRemove(self, subtree, target):
    # Cari target (data dari node yang ingin dihapus) dalam tree.
    if subtree is None:
        return subtree
    elif target < subtree.data:
        subtree.left = self._bstRemove(subtree.left, target)
        return subtree
    elif target > subtree.data:
        subtree.right = self._bstRemove(subtree.right, target)
        return subtree
    # Node target ditemukan
    else:
        # Kasus 1: Node target adalah node leaf
        if subtree.left is None and subtree.right is None:
            return None
        # Kasus 2: Node target mempunyai satu child
```

```
        if subtree.left is None or subtree.right is None:
            # Kembalikan child dari node target yang dihapus
            # Jika descendant berada di child kiri, kembalikan child kiri.
            if subtree.left is not None:
                return subtree.left
            # Jika descendant berada di child kanan, kembalikan child kanan.
            else:
                return subtree.right
        # Kasus 3: Node target mempunyai dua child.
        else:
            successor = self._bstMinimum(subtree.right)
            subtree.data = successor.data
            subtree.right = self._bstRemove(subtree.right, successor.data)
            return subtree
```

Implementasi Lengkap Method `remove(data)`

Kode berikut adalah implementasi dari method `_bstRemove()` dan method `remove()`:

```
class BSTMap:
    # ... Hanya kode-kode terkait method remove saja yang ditampilkan.

    # Method remove menghapus node dengan data yang diberikan sebagai argument.
    # Method ini menggunakan method bantu _bstRemove.
    def remove(self, dataDihapus):
        if dataDihapus not in self:
            raise Exception('Data tidak ada dalam tree.')
        else:
            self._root = self._bstRemove(self._root, dataDihapus)
            self._size -= 1

    # Method _bstRemove adalah method bantu yang menghapus node yang berada
    # di dalam argument subtree dan mempunyai nilai sama dengan argument target
    # yang diberikan.
    def _bstRemove(self, subtree, target):
        # Cari target (data dari node yang ingin dihapus) dalam tree.
        if subtree is None:
            return subtree
        elif target < subtree.data:
            subtree.left = self._bstRemove(subtree.left, target)
            return subtree
        elif target > subtree.data:
            subtree.right = self._bstRemove(subtree.right, target)
            return subtree
        # Node target ditemukan
        else:
            # Kasus 1: Node target adalah node leaf
            if subtree.left is None and subtree.right is None:
                return None
            # Kasus 2: Node target mempunyai satu child
            if subtree.left is None or subtree.right is None:
                # Kembalikan child dari node target yang dihapus
                # Jika descendant berada di child kiri, kembalikan child kiri.
                if subtree.left is not None:
                    return subtree.left
                # Jika descendant berada di child kanan, kembalikan child kanan.
                else:
```

```
                return subtree.right
            # Kasus 3: Node target mempunyai dua child.
            else:
                successor = self._bstMinimum(subtree.right)
                subtree.data = successor.data
                subtree.right = self._bstRemove(subtree.right, successor.data)
                return subtree
```

Method `remove(data)` kita tuliskan dengan kondisional yang menguji apakah `data` yang ingin dihapus berada di dalam tree. Jika `data` tidak ada di dalam tree kita meng-raise sebuah eksepsi. Jika `data` ada di dalam tree, method ini memanggil method `_bstRemove()` dengan argument node root dan `data`. Lalu, field `_size` didekrementasi.

Implementasi Lengkap ADT Binary Search Tree

Berikut adalah kode lengkap dari implementasi ADT Binary Search Tree:

Module *`binarysearchtree.py`*

```
class BST:
    # Constructor untuk membuat binary search tree baru
    # Field _root: menyimpan referensi ke node root dari tree.
    # Field _size: menyimpan ukuran tree (banyaknya node).
    def __init__(self):
        self._root = None
        self._size = 0

    # Method __len__ mengembalikan ukuran (nilai field _size)
    # dari binary search tree.
    # Method ini diakses menggunakan fungsi len().
    def __len__(self):
        return self._size

    # Method __contains__ (diakses dengan operator in) menerima
    # sebuah nilai target yang dicari.
    # Method ini mengembalikan true jika nilai target ditemukan
    # dan mengembalikan false jika nilai target tidak ditemukan.
    # Method ini menggunakan method bantu _bstSearch.
    def __contains__(self, nilaiDicari):
        return self._bstSearch(self._root, nilaiDicari) is not None

    # Method _bstSearch adalah method bantu yang secara rekursif
    # mencari nilai target dalam tree.
    def _bstSearch(self, subtree, target):
        if subtree is None :
            return None
        elif target < subtree.data: # Target berada di subtree kiri.
            return self._bstSearch(subtree.left, target)
        elif target > subtree.data: # Target berada di subtree kanan
            return self._bstSearch(subtree.right, target)
        else:
            return subtree

    # Method min mengembalikan nilai minimum dalam binary search tree
    def min(self):
        nodeMinimum = self._bstMinimum(self._root)
        if nodeMinimum is None:
```

```
            return 0
        return nodeMinimum.data

    # Method _bstMinimum melakukan pencarian node paling kiri dengan
    # melakukan traverse arah kiri secara rekursif.
    def _bstMinimum(self, subtree):
        if subtree is None:
            return None
        elif subtree.left is None:
            return subtree
        else:
            return self._bstMinimum(subtree.left)

    # Method add menambahkan node berisi data baru ke binary search tree.
    # Method ini mengembalikan True jika data berhasil ditambahkan dan
    # mengembalikan False jika data telah ada di binary search tree.
    def add(self, data):
        # Cari terlebih dahulu data di dalam tree.
        node = self._bstSearch(self._root, data)
        # Jika terdapat data dalam tree kembalikan False.
        # Jika data tidak ada dalam tree, masukkan data ke tree dengan
        # memanggil method bantu _btsInsert.
        if node is not None:
            return False
        else:
            self._root = self._bstInsert(self._root, data)
            self._size += 1
            return True

    # Method _btsInsert adalah method bantu untuk memasukkan node baru
    # dalam tree secara rekursif.
    def _bstInsert(self, subtree, data):
        # Jika subtree kosong
        if subtree is None:
            subtree = _BSTNode(data)
        elif (data < subtree.data):
            subtree.left = self._bstInsert(subtree.left, data)
        elif (data > subtree.data):
            subtree.right = self._bstInsert(subtree.right, data)
        return subtree

    # Method remove menghapus node dengan data yang diberikan sebagai argument.
    # Method ini menggunakan method bantu _bstRemove.
    def remove(self, dataDihapus):
        if dataDihapus not in self:
            raise Exception('Data tidak ada dalam tree.')
        else:
            self._root = self._bstRemove(self._root, dataDihapus)
            self._size -= 1

    # Method _bstRemove adalah method bantu yang menghapus node yang berada
    # di dalam argument subtree dan mempunyai nilai sama dengan argument target
    # yang diberikan.
    def _bstRemove(self, subtree, target):
        # Cari target (data dari node yang ingin dihapus) dalam tree.
        if subtree is None:
            return subtree
        elif target < subtree.data:
```

```
            subtree.left = self._bstRemove(subtree.left, target)
            return subtree
        elif target > subtree.data:
            subtree.right = self._bstRemove(subtree.right, target)
            return subtree
        # Node target ditemukan
        else:
            # Kasus 1: Node target adalah node leaf
            if subtree.left is None and subtree.right is None:
                return None
            # Kasus 2: Node target mempunyai satu child
            if subtree.left is None or subtree.right is None:
                # Kembalikan child dari node target yang dihapus
                # Jika descendant berada di child kiri, kembalikan child kiri.
                if subtree.left is not None:
                    return subtree.left
                # Jika descendant berada di child kanan, kembalikan child kanan.
                else:
                    return subtree.right
            # Kasus 3: Node target mempunyai dua child.
            else:
                successor = self._bstMinimum(subtree.right)
                subtree.data = successor.data
                subtree.right = self._bstRemove(subtree.right, successor.data)
                return subtree

# Class _BSTNode merepresentasikan node dalam binary search tree.
class _BSTNode:
    # Constructor untuk membuat node binary search tree baru.
    # Field data: Menyimpan nilai pada node.
    # Field left: Referensi ke node child kiri.
    # Field right: Referensi ke node child kanan.
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

Menguji Implementasi Binary Search Tree

Kita dapat menggunakan kode berikut untuk menguji implementasi Binary Search Tree:

```
from binarytree import inorderTrav
from binarysearchtree import BST

# Konstruksi binary search tree dari
# list [60, 12, 90, 4, 41, 71, 100, 1, 29, 84, 23, 37]
binSearchTree = BST()
data = [60, 12, 90, 4, 41, 71, 100, 1, 29, 84, 23, 37]

# Tambahkan elemen-elemen data ke binary search tree
for datum in data:
    binSearchTree.add(datum)

# Print node-node tree
print('BST pertama kali dibangun:')
inorderTrav(binSearchTree._root)
print()
```

```
# Tambahkan 30 ke tree
binSearchTree.add(30)

# Tampilkan ukuran tree
print('Ukuran tree = ', len(binSearchTree))

# Cek apakah 32 ada di tree
print('32 ada di tree?')
if (32 in binSearchTree):
    print('32 ada di tree.')
else:
    print('32 tidak ada di tree.')

# Print node-node tree setelah penambahan
print('BST setelah penambahan nilai 30:')
inorderTrav(binSearchTree._root)
print()

# Min
print('Min = ', binSearchTree.min())

# Remove 23 (node leaf)
binSearchTree.remove(23)

# Remove 71 (node dengan 1 child)
binSearchTree.remove(71)

# Remove 12 (node dengan 2 child)
binSearchTree.remove(12)

# Print tree terakhir
print('BST akhir:')
inorderTrav(binSearchTree._root)
print()

# Tampiklan ukuran tree
print('Ukuran tree akhir = ', len(binSearchTree))```
```

Output dari kode di atas:

```
BST pertama kali dibangun:
1 4 12 23 29 37 41 60 71 84 90 100
Ukuran tree =  13
32 ada di tree?
32 tidak ada di tree.
BST setelah penambahan nilai 30:
1 4 12 23 29 30 37 41 60 71 84 90 100
Min =  1
BST akhir:
1 4 29 30 37 41 60 84 90 100
Ukuran tree akhir =  10
```

REFERENSI:

[1] Gaddis, Tony. 2012. *Starting Out With Python Second Edition*. United States of America: Addison-Wesley.