Catatan Perkuliahan

Pertemuan 8 Struktur Data - Binary Search Tree

Auzi Asfarian, SKomp, MKom

Catatan: Tidak ada slide presentasi yang diberikan pada sesi UAS ini.

1 Ikhtisar

Selama tujuh pertemuan di masa UAS, kita akan melihat beberapa bentuk struktur data hierarkis, graf, serta beberapa algoritme klasik yang menggunakan graf. Berbeda dengan tujuh pertemuan sebelumnya yang memuat struktur data sekuensial, *tree* merupakan struktur data hierarkis yang merupakan salah satu terobosan dalam struktur data. Pada 3 pertemuan pertama di masa UAS ini, kita akan melihat karakteristik dari *tree* dan beberapa varian *tree* yang dikembangkan untuk meningkatkan efisiensi penggunaan *tree*. Selain itu juga, dijelaskan pula teknik penelusuran (alias *traversal*) yang umum digunakan untuk mencari data pada *tree*.

Tree merupakan bentuk khusus dari sebuah struktur data yang lebih besar, dan memiliki banyak kegunaan, yaitu graf. Pada pertemuan 12, disajikan definisi graf, model implementasinya dalam pemrograman, serta penelusuran pada struktur data graf: DFS dan BFS. Pada pertemuan 13, kita akan melihat dua problem klasik pada graf, yaitu *minimum spanning tree* serta solusinya menggunakan algoritme Prime dan Kruskal dan *single source shortest path* serta solusinya dengan menggunakan algoritme Dijkstra. Pertemuan masa UAS akan ditutup dengan struktur data *hash*.

Secara singkat, silabus masa UAS ialah:

- Pertemuan 8 Tree dan Binary Search Tree
- Pertemuan 9 AVL Tree
- Pertemuan 10 B-Tree dan Trie
- Pertemuan 12 Graf, DFS, dan BFS
- Pertemuan 13 Minimum Spanning Tree, Single Source Shortest Path
- Pertemuan 14 Hash

Pada semester kali ini, pengajar mencoba untuk tidak mengajar dengan menggunakan media slide presentasi. Catatan kuliah ini adalah satu-satunya materi tertulis yang akan diberikan kepada siswa.

1 Tree

Sebelum masuk pada *tree*, terlebih dulu kita perlu mengingat kembali definisi graf, yaitu kumpulan *verteks* (atau node) dan *edge*. *Tree* adalah graf yang tidak memiliki *cycle* di dalamnya. Akan tetapi, pengertian *tree* dalam struktur data lebih mengarah pada sifat tree yang menyimpan data secara hierarkis. Artinya, data dalam *tree* memiliki hubungan ‘di atas’ dan ‘di bawah’ atau lebih lazim disebut sebagai hubungan *parent-child* atau *ancestor-descendant*. Perlu diketahui pula bahwa

terdapat problem yang secara alamiah memang berhirarki sehingga struktur data *tree* cocok untuk digunakan.

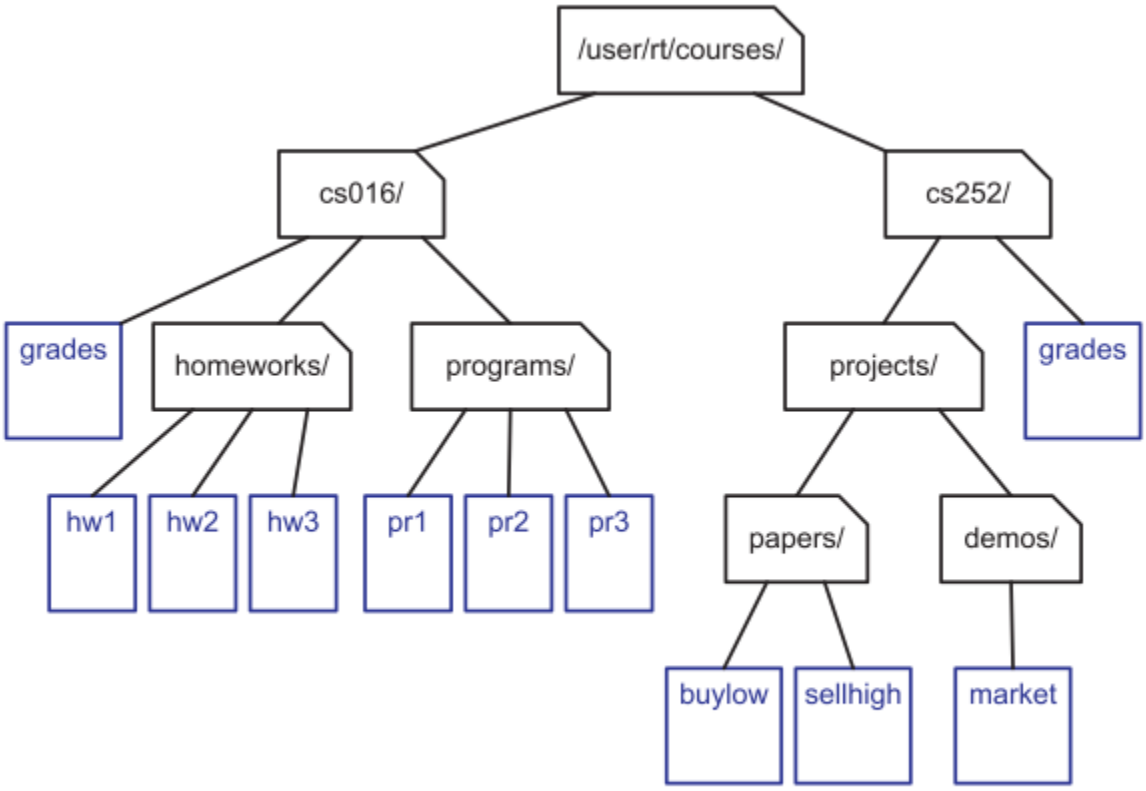


Figure 7.3: Tree representing a portion of a file system.

2 Terminologi

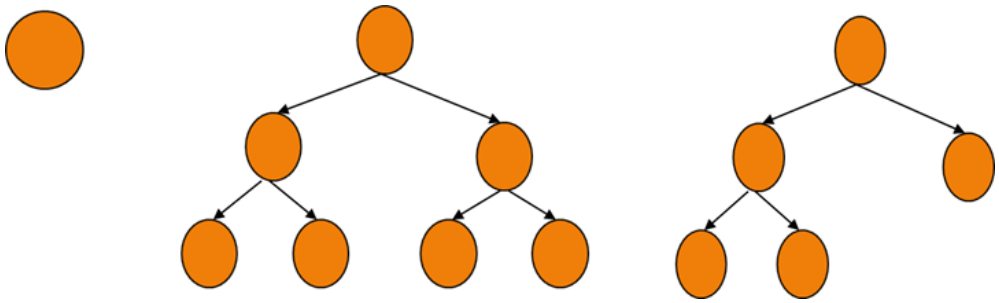
- **Tree:** sekumpulan node yang menyimpan elemen dalam hubungan *parent-child*.
- **Root:** node pada tree yang memiliki hierarki paling tinggi, dengan kata lain tidak memiliki *parent*.
- **Siblings**: node-node yang memiliki *parent* yang sama.
- **Leaf:** node yang tidak memiliki *child*.
- **Internal Node:** seluruh node yang bukan leaf (memiliki *child*).
- Predecessor: node yang berada di atas node tertentu.
- **Successor**: node yang berada di bawah node tertentu.
- **Ancestor**: node *u* adalah *ancestor* dari *v* jika *u* = *v* atau *v* adalah *ancestor* dari *parent v*.
- Descendant: *v* adalah *descendant* dari *v* jika v adalah *ancestor* dari *v*.
- **Subtree**: *tree* yang terdiri atas node *v* pada *tree* dan seluruh *descendant*-nya.
- **Edge:** pasangan dua buah node, direpresentasikan dalam bentuk garis.
- **Path**: sekuens node sehingga dua node berurutan membentuk *edge*.
- **Size:** Banyaknya node dalam suatu tree.

- **Level**: tingkatan suatu node. Root berada pada level 0, node di bawahnya bernilai 1 dan seterusnya.
- **Height**: ketinggian tree dihitung dari bawah. Leaf memiliki height 0.
- **Depth**: jumlah edge dari root sampai node tersebut.
- **Degree**: banyaknya child dari node.
- **Forest**: himpunan tree.
  Catatan: ilustrasi diberikan di papan tulis.

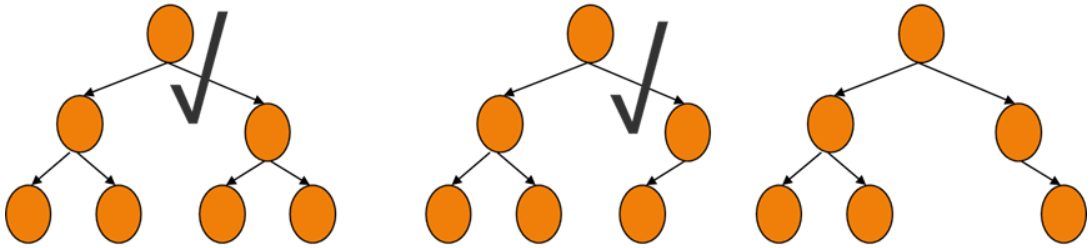
3 Binary Tree

Binary tree merupakan *tree* yang setiap nodenya memiliki *degree* maksimum 2. Terdapat beberapa varian *binary tree* yang perlu diketahui saat menganalisis struktur data berbentuk *tree*, misalnya *complete binary tree, full binary tree, perfect binary tree,* dan *skewed binary tree.*

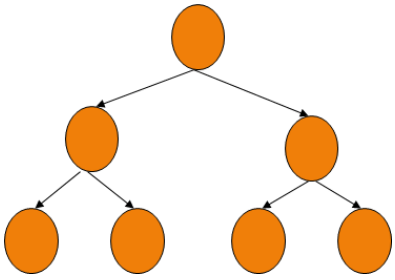
- *Full Binary Tree*: *binary tree* yang setiap node di dalamnya memiliki tepat 0 atau 2 *child.*



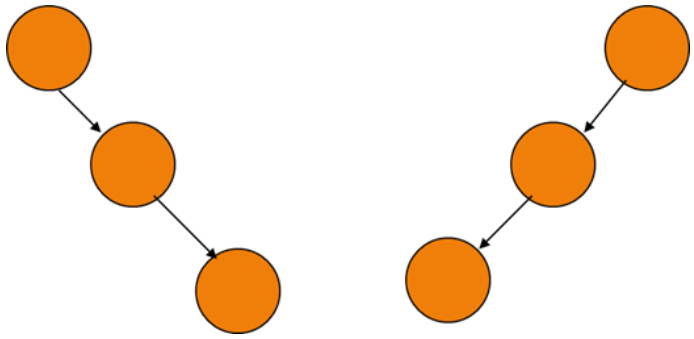
- *Complete Binary Tree*: setiap leaf berada pada *depth* ke-N atau N-1 dan setiap *leaf* pada level terendah merapat ke arah kiri.



- *Perfect Binary Tree*: setiap node internal memiliki tepat dua *child* dan seluruh *leaf* berada pada ketinggian yang sama.

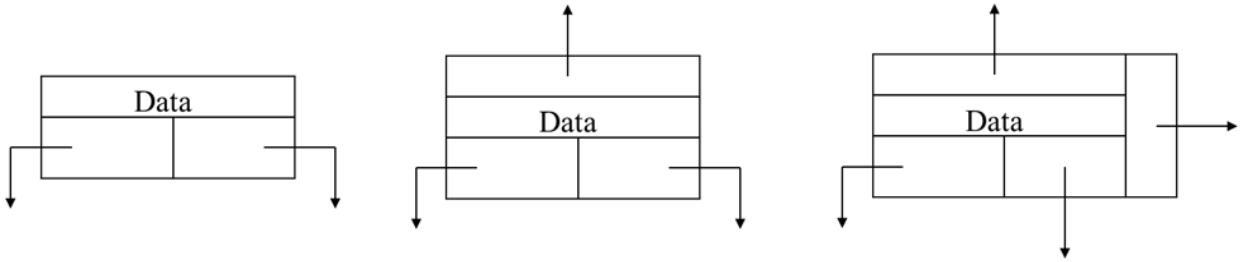


- *Skewed Binary Tree*: setiap node internal memiliki tepat satu *child*.

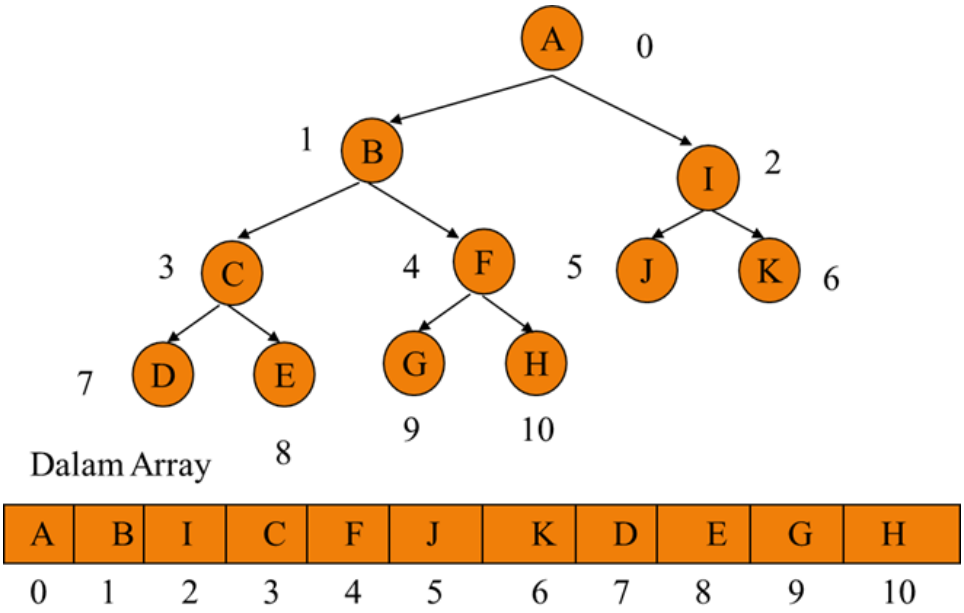


4 Representasi Binary Tree

Setiap node dapat dibuat dalam bentuk *struct*.



Atau dengan menggunakan *array* (khusus CBT dan PBT, mengapa?):



5 Traversal

Traversal adalah cara terstruktur untuk mengunjungi (serta memeriksa atau mengoperasikan) setiap node pada *tree* tepat satu kali. Terdapat tiga varian:

- Preorder : root, kiri, kanan
- Inorder : kiri, root, kanan
- Postorder : kiri, kanan root

Traversal dapat diimplementasikan dalam bentuk kode dengan menggunakan fungsi rekursif. Faktanya, pada *tree*, rekursif akan sering digunakan. Dengan mengasumsikan x adalah node *root* pada *tree/subtree*, pseudokode untuk masing-masing algoritme traversal ialah:

Preorder(node)

```
1. if node != NULL then
2.      visit(node)
3.      Preorder(node->left)
4.      Preorder(node->right)
```

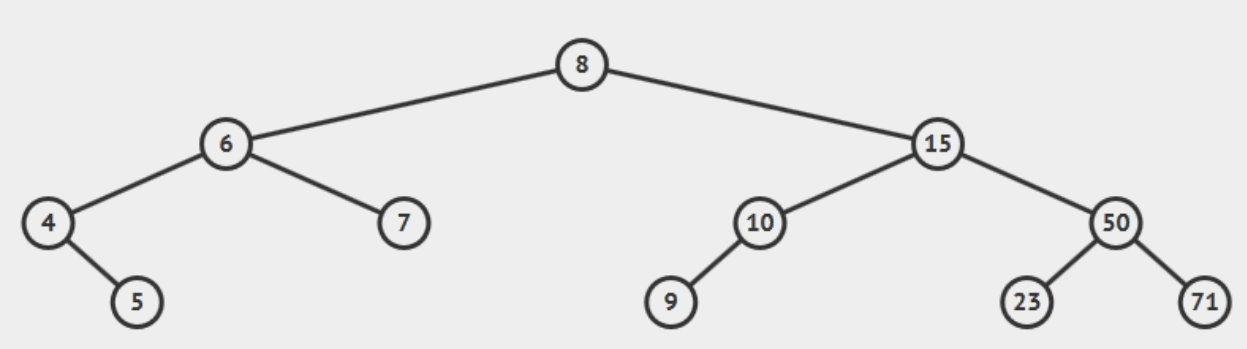
Inorder(x)

```
1. if node != NULL then
2.      Inorder(node->left)
3.      visit(node)
4.      Inorder(node->right)
```

Postorder(x)

```
1. if node != NULL then
2.      Postorder(node->left)
3.      Postorder(node->right)
4.      visit(node)
```

Misalkan diberikan BST seperti di bawah ini:



(Gambar sementara, dibuat menggunakan visualgo.net)

Keluaran dari masing-masing algoritme traversal ialah:

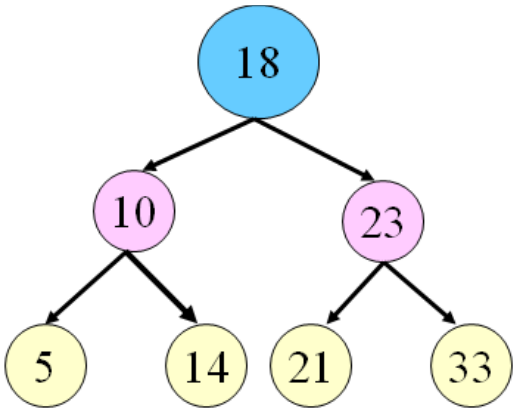
- Preorder: 8 - 6 - 4 - 5 - 7 - 15 - 10 - 9 - 50 - 23 - 71
- Inorder: 4 - 5 - 6 - 7 - 8 - 9 - 10 - 15 - 23 - 50 - 72

- Postorder: 5 - 4 - 7 - 6 - 9 - 10 - 23 - 71 - 50 - 15 - 8

Pada pertemuan 12, kita akan melihat bentuk umum dari algoritme traversal: *depth first search* dan *breadth first search*.

6 Ordered Tree

Sebuah *tree* dikatakan *ordered* jika terdapat skema pengurutan tertentu yang digunakan untuk menyimpan *children* pada setiap node. Misal dalam tree berikut, *left child* dari setiap node bernilai lebih kecil daripada node tersebut dan *right child* dari node bernilai lebih besar daripada node tersebut.



7 Binary Search Tree

Binary search tree (BST) adalah *ordered binary tree* dengan urutan *subtree* kiri < root < *subtree* kanan. Beberapa implementasi fungsi pada *binary search tree* mengalami perubahan agar urutan tersebut tetap terjaga. Kita mengasumsikan tidak ada *key* yang duplikat pada *tree*.

Pencarian

Pencarian pada BST menjadi lebih mudah dilakukan dengan adanya pengurutan. Bermula pada *root*, kita cukup membandingkan nilai *k* yang kita cari dengan nilai *root*. Apabila nilai nilai pada root lebih kecil daripada *k,* pencarian dilanjutkan ke *subtree* kiri, sedangkan jika lebih besar pencarian dilanjutkan *subtree* kanan. Sifat ini mirip dengan algoritme *binary search*. Algoritme pencarian dapat ditulis sebagai berikut:

```
tree_node* BST-Search(node, k)
   1. if node == NULL then
   2.     return node
   3. if node->key == key
   4.     return node
   5. if key < node->key then
```

```
6.     search(node->left, key)
7. else
8.     search(node->right, key)
```

Maksimum dan Minimum

Nilai minimum pada BST berada pada *leaf* yang paling kiri, sedangkan nilai maksimum pada BST berada pada *leaf* yang paling kanan.

Suksesor dan Predesesor

Suksesor dari sebuah node bernilai *k* pada *search tree* adalah node dengan nilai terkecil yang lebih besar daripada *k* di dalam *tree* tersebut. Sebaliknya, predesesor dari sebuah node bernilai *k* adalah node dengan nilai terbesar yang lebih kecil daripada *k* di dalam *tree* tersebut.

Penyisipan

Penyisipan BST dilakukan mirip dengan pencarian. Penelusuran dimulai dari *root* hingga ke node yang anak kanan atau anak kirinya dapat diisi dengan nilai yang disisipkan. Algoritme penyisipan pada BST ialah sebagai berikut:

```
tree_node* BST-Insert(node, k)
1. if node == NULL
2.      root = newNode(k)
3. if key < node->key then
4.      node->left = insert(node->left, key)
5. else
6.      node->right = insert(node->right, key)
7. return node
```

Penghapusan

Penghapusan pada BST lebih kompleks dibandingkan fungsi yang lain. Misal kita ingin menghapus sebuah node *z*, maka terdapat tiga buah kemungkinan:

- Jika *z* tidak memiliki *child*, maka *z* diisi dengan NULL.
- Jika *z* hanya memiliki satu *child*, maka *child* tersebut akan naik menggantikan *z.*
- Jika *z* memiliki dua *child*, maka *z*_digantikan dengan suksesornya. Anak kanan suksesor kemudian dihapus dengan menggunakan algoritme penghapusan yang sama.

```
tree_node* BST-Delete(node, k)
1. if node == NULL then
2.      return node
3. if  key < node->key then
4.      node->left = deleteNode(node->left, key)
5. else if key > node->key then
6.      node->right = deleteNode(node->right, key)
7. else
```

```
8.      //Hanya satu anak
9.       if node->left == NULL then
10.                 tree_node *temp = node->right
11.                 node = NULL
12.                 return temp
13.         else if node->right == NULL then
14.                 tree_node *temp = node->left
15.                 node = NULL
16.                 return temp
17.
18.         // Ada dua anak
19.         tree_node* temp = getSuccessorNode(node)
20.         node->key = temp->key
21.
22.         // Hapus suksesor dengan algoritme yang sama
23.         node->right = deleteNode(node->right, temp->key)
24.      return node;
```

8 Analisis

Jumlah pengecekan yang diperlukan untuk mencari nilai pada BST ialah O(*h*) dengan *h* adalah ketinggian *tree*. Agar nilai *h* dapat optimal, *tree* haruslah berbentuk PBT atau setidaknya CBT yang mana pada saat demikian *h* bernilai $\log_2 n$ dengan *n* adalah jumlah elemen pada *tree*. Akan tetapi, ketinggian *tree* paling besar ketika elemen dimasukkan secara berurut atau hampir terurut. Hal ini akan menyebabkan *tree* menjadi berbentuk *skewed*. Oleh karena itu, kompleksitas pencarian pada BST ialah O(*n*). Karena penyisipan dan penghapusan elemen juga melibatkan pencarian, kompleksitas keduanya pun juga menjadi O(*n*).

Pada bagian selanjutnya, kita akan melihat jenis *tree* baru (AVL Tree) yang memiliki mekanisme pengaturan keseimbangan *tree* secara otomatis sehingga *tree* selalu seimbang.

9 Eksplorasi

Visualgo.net (visualgo.net/bst.html) menyediakan visualisasi algoritme pada BST dan beberapa struktur data lainnya. Apabila Anda kesulitan memahami pseudokode yang disediakan, Anda dapat mencoba mengamati visualisasi yang diberikan sembari membuat implementasi kode dari BST.

10 Latihan

1. Buatlah algoritme BST-Minimum(x) dan BST-Maximum(x) untung mencari nilai minimum dan nilai maksimum

2. Buatlah pseudocode atau kode dari fungsi `isBST` yang mengecek apakah sebuah *binary tree* yang diberikan merupakan BST.
3. Traversal inorder pada BST akan mencetak elemen pada BST terurut secara menaik. Rancanglah sebuah algoritme traversal yang dapat mencetak elemen pada BST terurut secara menurun.
4. Hampir seluruh algoritme yang diberikan dapat dibuat dalam bentuk iteratif. Buatlah implementasinya!

Praktikum

Kegiatan 1:
Berdasarkan algoritme dan ilustrasi yang diberikan pada sesi perkuliahan, lengkapilah kode kerangka berikut sehingga kelas BST dapat digunakan.

```
struct tree_node
{
      int key;
      tree_node *left, *right;
};

class BST
{
public:
      tree_node *root;
      BST(){root = NULL;};
      tree_node* newNode(int item);
      tree_node* search(tree_node* node, int key);
      tree_node* insert(tree_node* node, int key);
      tree_node* deleteNode(struct tree_node* root, int key);
      void visit(tree_node* node);
      void preorder(tree_node *node);
      void inorder(tree_node *node);
      void postorder(tree_node *node);
      tree_node * getMinimumNode(tree_node* node);
      tree_node * getMaximumNode(tree_node* node);
      tree_node * getSuccessorNode(tree_node* node);
      tree_node * getPredecessorNode(tree_node* node);
};
```

Kegiatan 2:
Buatlah kode dari fungsi `isBST` yang mengecek apakah sebuah *binary tree* yang diberikan merupakan BST.

Kegiatan 3:
Implementasikan sebuah fungsi traversal yang dapat mencetak elemen pada BST terurut secara menurun (soal tersedia di LX: Latihan Struktur Data 8).

Kegiatan 4 (Tugas):

Gunakanlah BST untuk membantu kasus pencarian buku pada perpustakaan. Nilai yang disimpan pada setiap node ialah String (soal tersedia di LX: Tugas Struktur Data 8; Penilaian dilakukan manual).