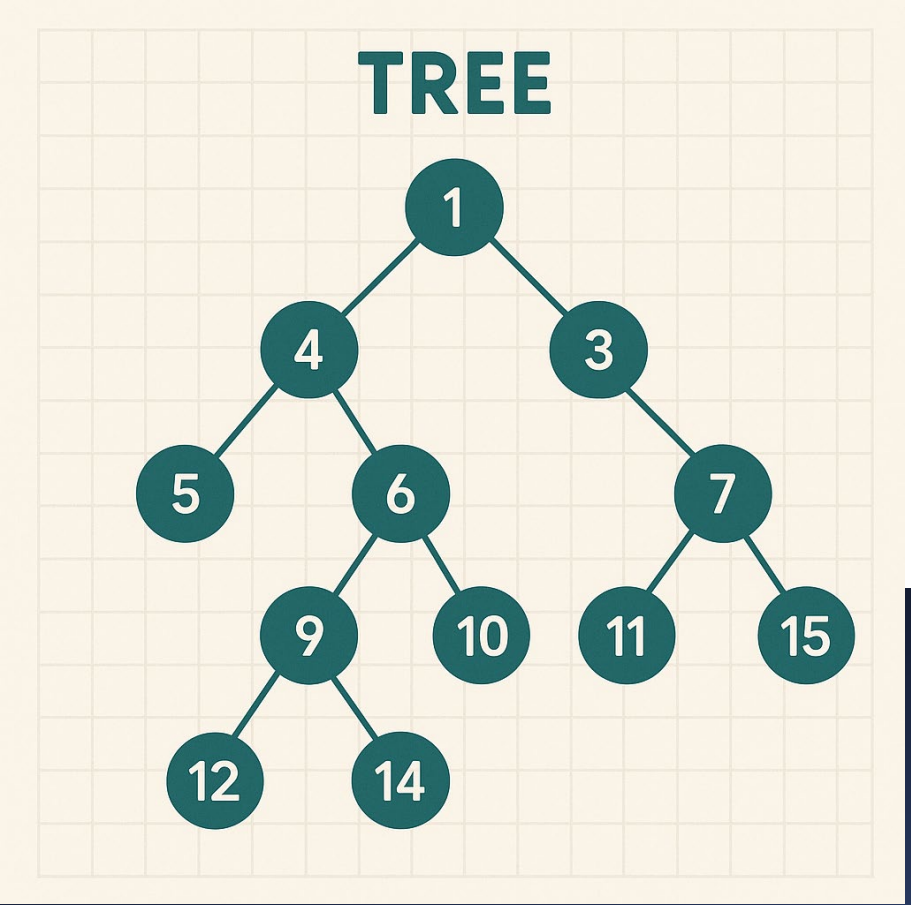
Chapter 7:Tree

BITE 1523 – COMPUTER GAME PROGRAMMING



Recap: Types of data structure: Trees

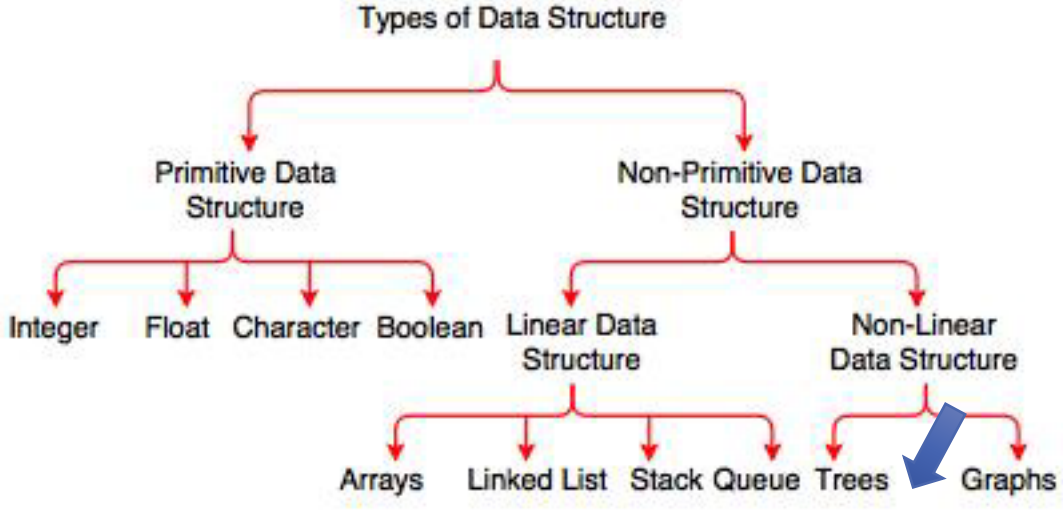


Fig. Types of Data Structure

Outline

Introduction of Trees

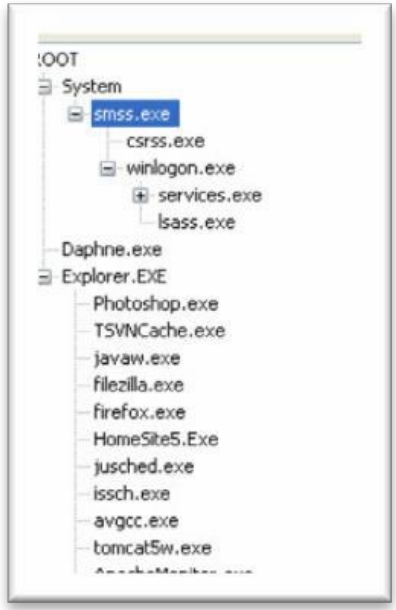
Definition

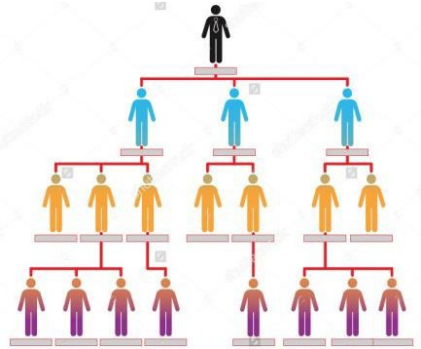
Terminologies

Type of Trees

Introduction

- In linked lists, most operations like insert, delete and find take **O(N)**.
- In contrast, trees take **O(log N)** on an average for most operations.
- A common application of trees is in storing the directory structure of computer file systems.





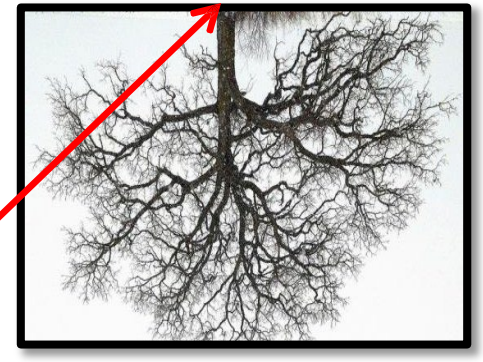
Introduction

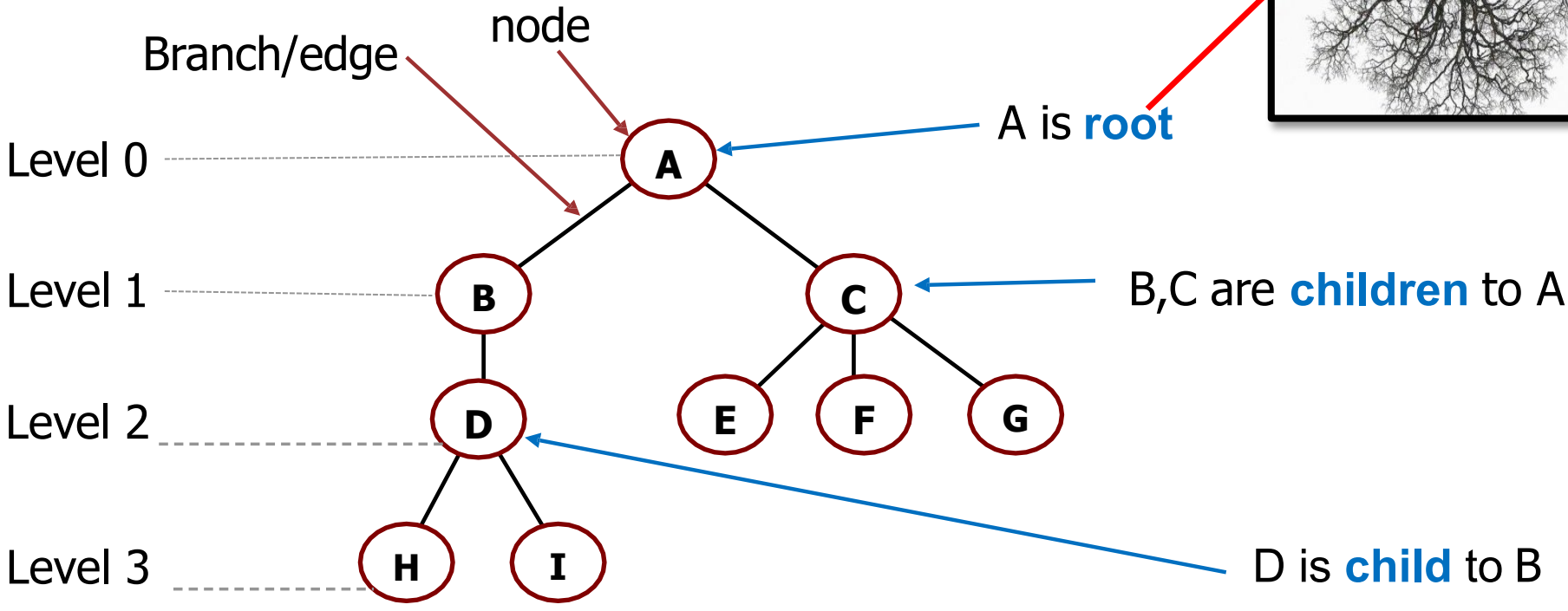
- Tree is a **nonlinear data structure.**
- Can be used to store arithmetic expressions, store data in hierarchical form such as organization chart
- Suitable for storing big and dynamic data to enable fast access to data.
- Type of trees : general tree, binary tree, binary search tree, expression tree and heap tree.

Introduction

- Trees consist of **nodes** and **branches / edges**.
- The node stores data and branches connect the nodes.
- A tree may be a null tree (contains no node) or non-null tree.
- A non-null tree must have **a root node**. Every node may have one or more nodes known as **child node.** Each node except the root node must have **one parent node**.

Terminologies in Trees

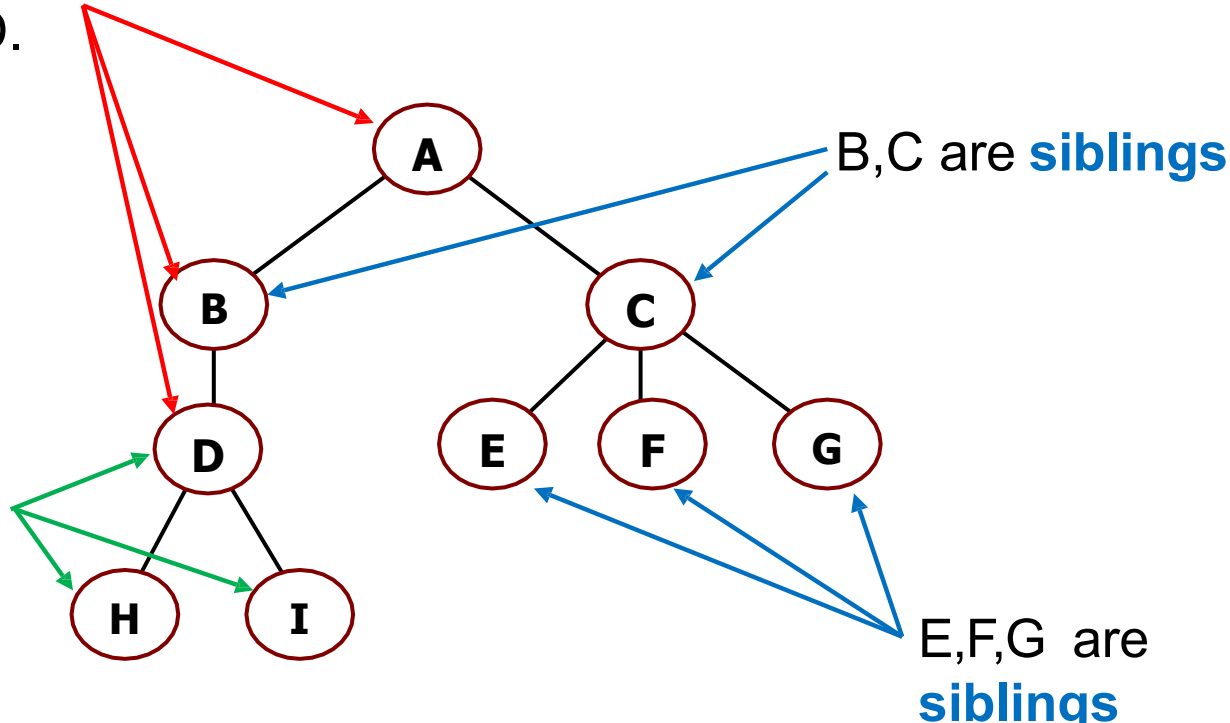




Terminologies in Trees

Ancestor nodes are nodes in the path starting from the root node until the particular node.
e.g. the **ancestor nodes for node H / I** are nodes A, B are D.

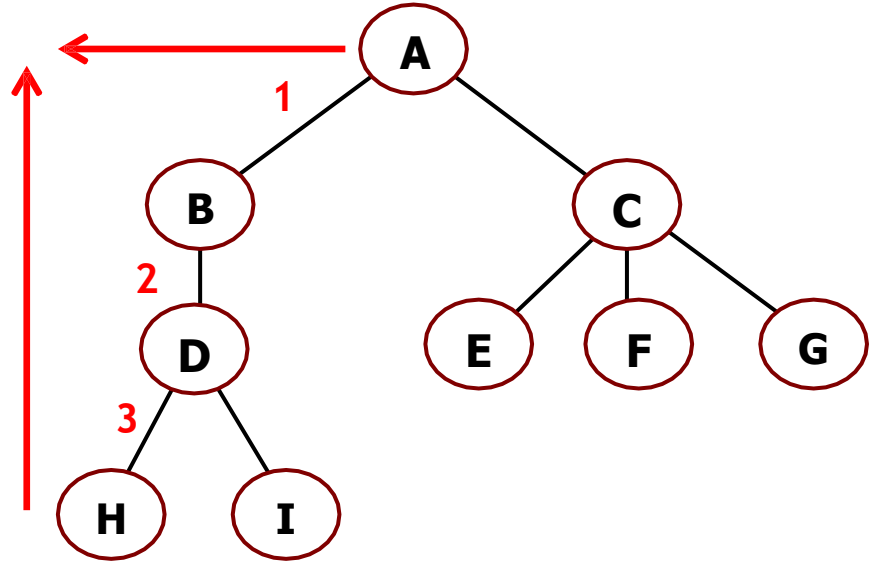
Descendent nodes are all nodes below a particular node starting from the node until the leaf nodes.
e.g. the **descendent nodes for node B** are node D, H and I.



Terminologies in Trees

The height of a tree is the **number of branches / edges** on the **longest path** between **the root** and **a leaf**. The height of a null tree is 0.

The height of the tree below is 3

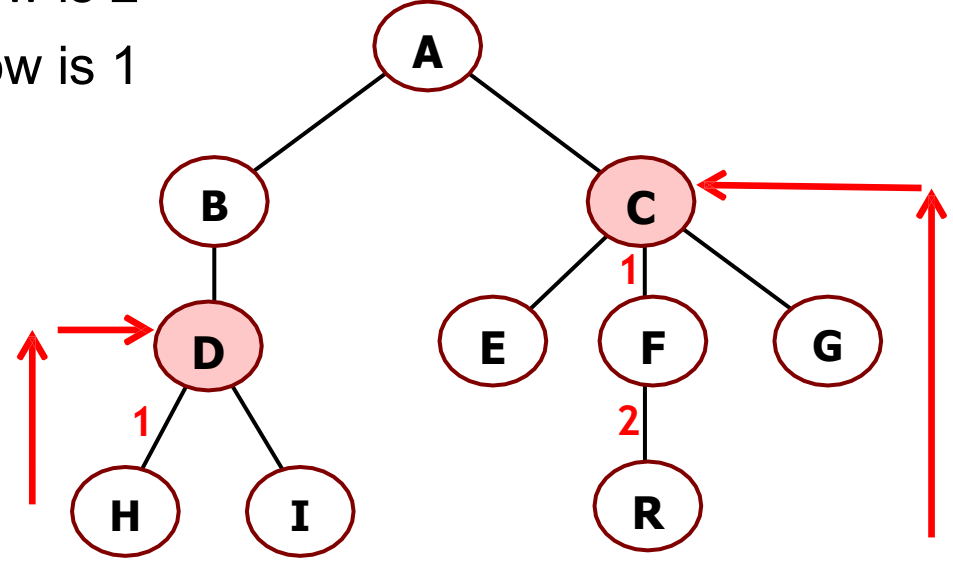


Terminologies in Trees

The height of a node is the **number of branches / edges** on the **longest path** between **that node** and **a leaf**.

The height of the node C below is 2

The height of the node D below is 1

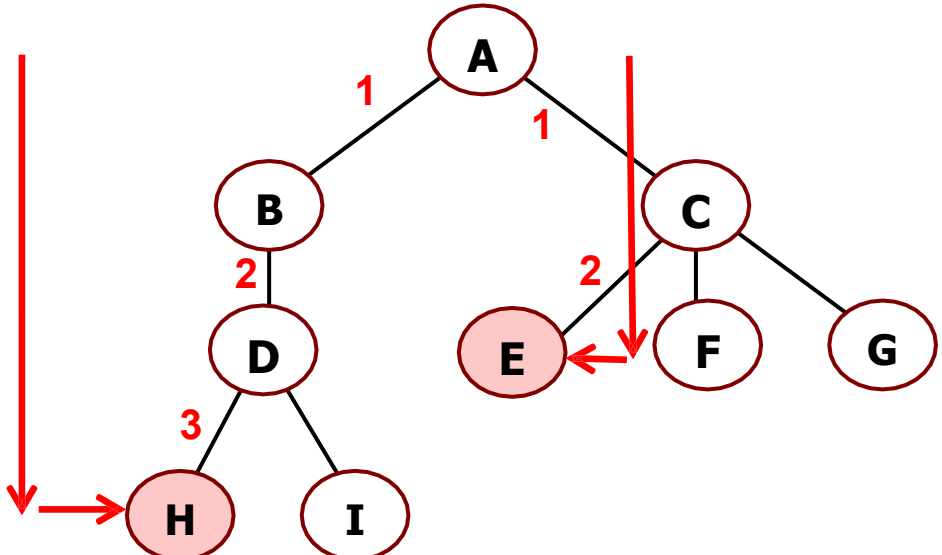


Terminologies in Trees

The depth of a node is the **number of branches / edges** from **the tree's root** to **the node**.

The depth of the node H below is 3

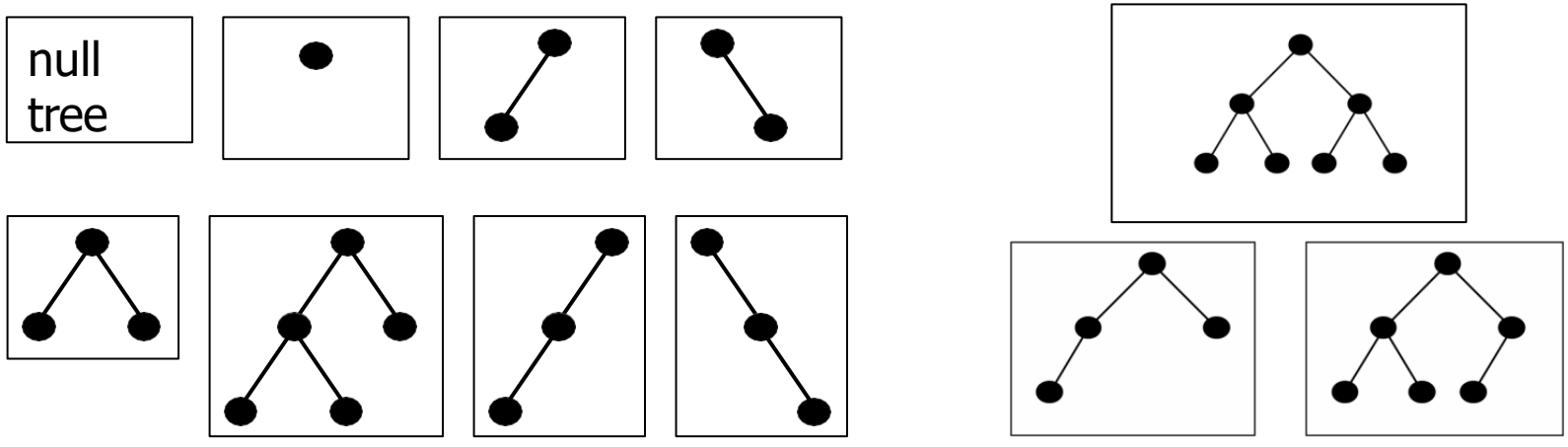
The depth of the node E below is 2



Binary Trees

A null tree OR a tree that contains the root node and a left subtree or/and a right subtree, each has no more than 2 children. A complete binary tree must have the maximum number of nodes. A nearly complete binary tree has the minimum height for its nodes and the last node is on the left of the subtree.

Examples of binary trees

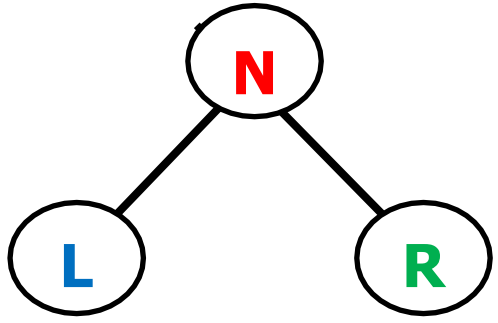


Binary Tree Traversal

- Operation that is usually done on binary trees is tree traversal.
- **Traversing** a tree is to visit all its nodes at least once starting from the root
- Three types of Binary Tree Traversal sequence (**also called depth-first tree traversal**) :-
    i. Preorder Traversal
    ii. Inorder Traversal
    iii. Postorder Traversal

Binary Tree Traversal

N – root node



L – left subtree node

R – right subtree node

Preorder : NLR
Inorder : LNR
Postorder : LRN

Binary Tree Traversal

Preorder Traversal – NLR

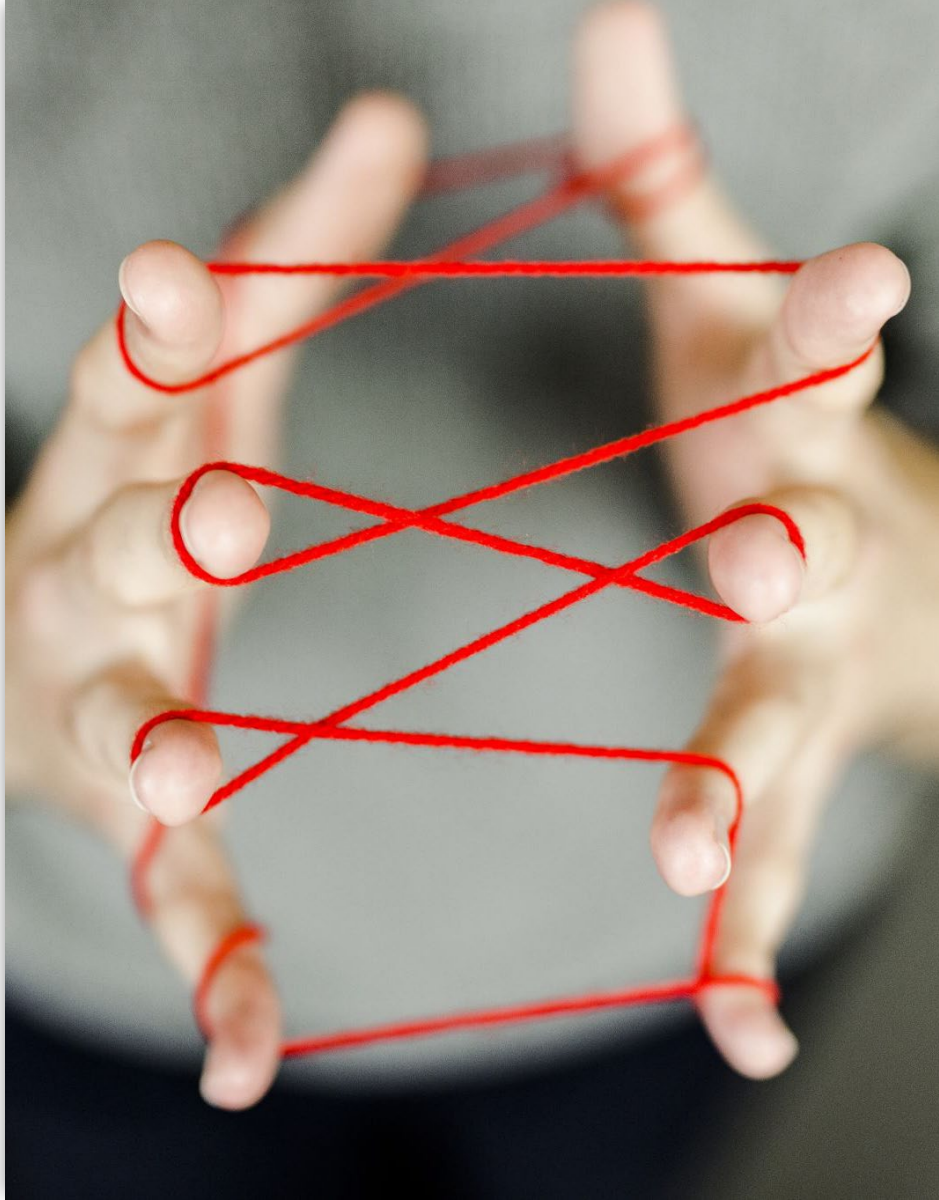
- Root node, followed by the left subtree nodes,
- then the right subtree nodes.

Inorder Traversal – LNR

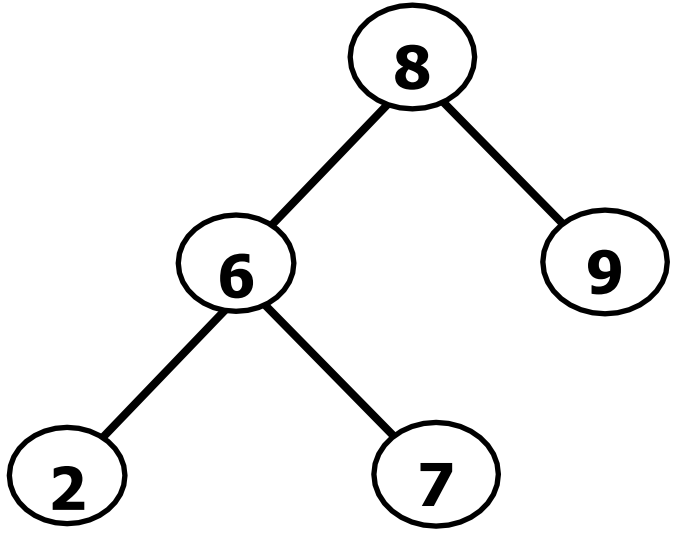
- Left subtree nodes to be processed first, followed
- by the root node, then the right subtree nodes.

Postorder Traversal – LRN

- Left subtree nodes to be processed first, followed
- by the right subtree nodes, then the root node.



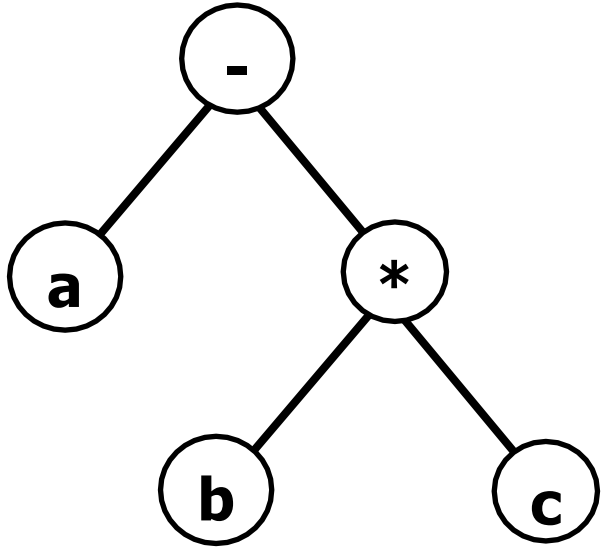
Example of Traversal

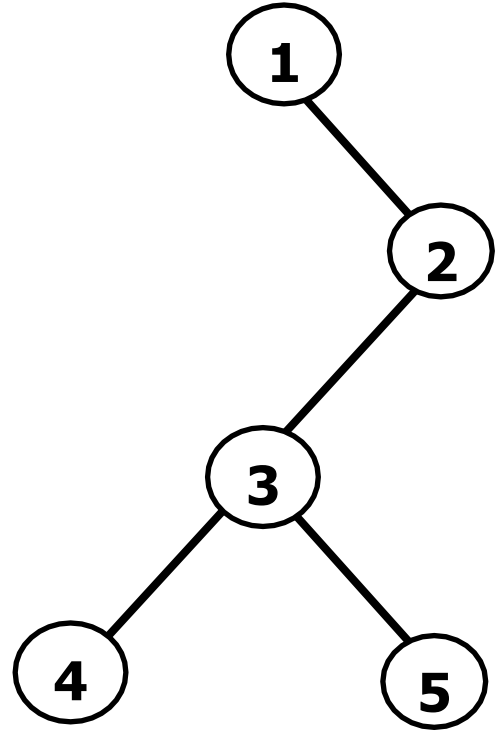


Preorder (NLR) : **8 6 2 7 9**
Inorder (LNR) : **2 6 7 8 9**
Postorder (LRN) : **2 7 6 9 8**

Exercise

What are the results of Preorder, Inorder and Postorder ?





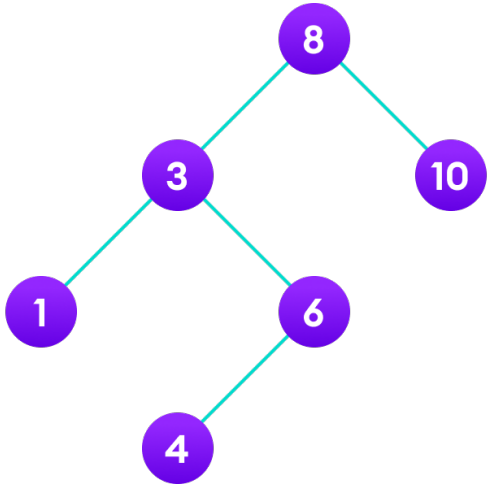
Binary Search Tree (BST)

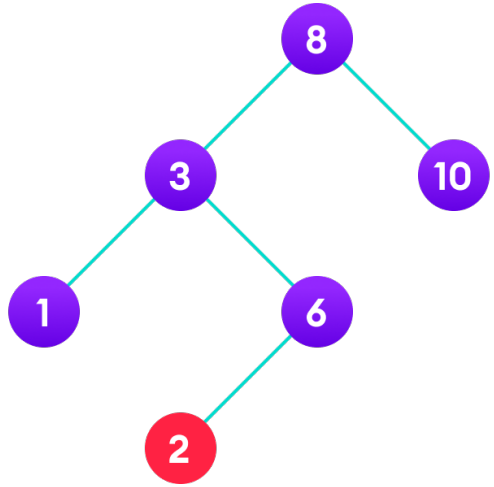
A more efficient method to find an item is by using the binary search tree.

The binary search tree has the following properties:

Value in left subtree nodes < the value in root node	Value in right subtree nodes > the value in root node.	The left and right subtrees **are also** binary search trees.	All data in the tree nodes are unique.

Example of BST





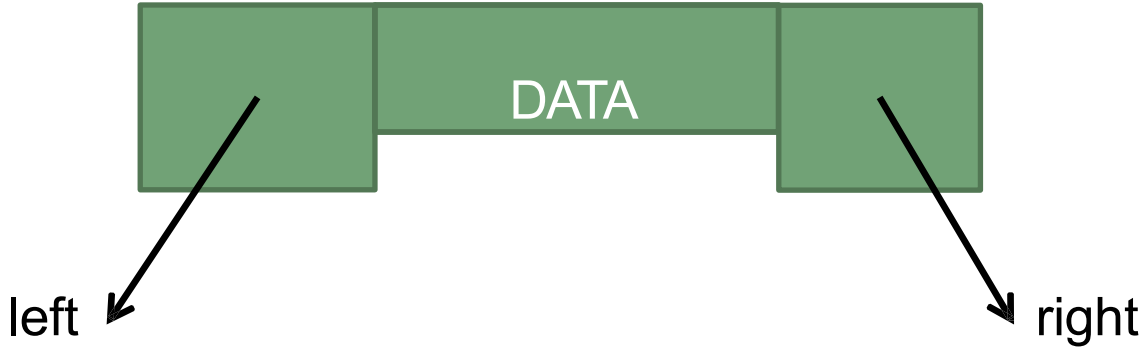
The binary tree on the right isn't a binary search tree because the right subtree of the node "3" contains a value smaller than it.

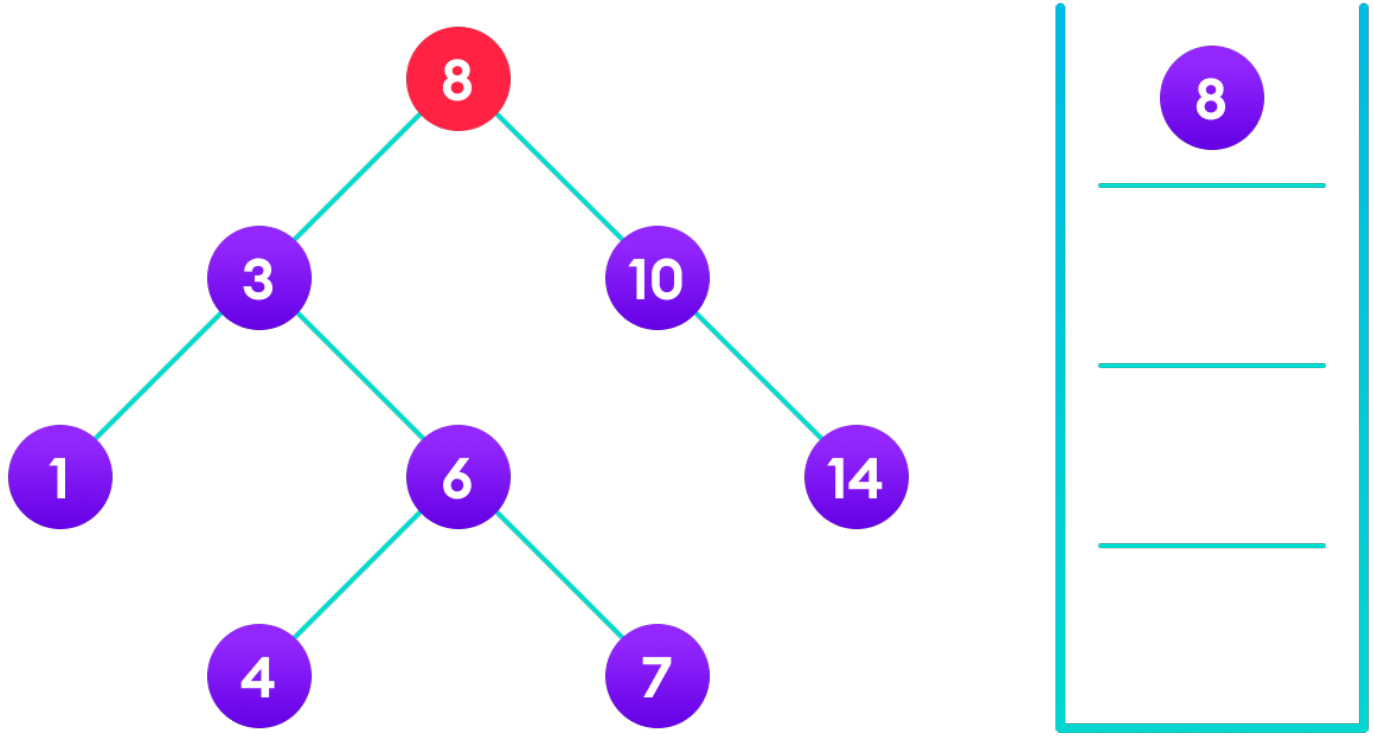
There are two basic operations that you can perform on a binary search tree:

Implementation of BST

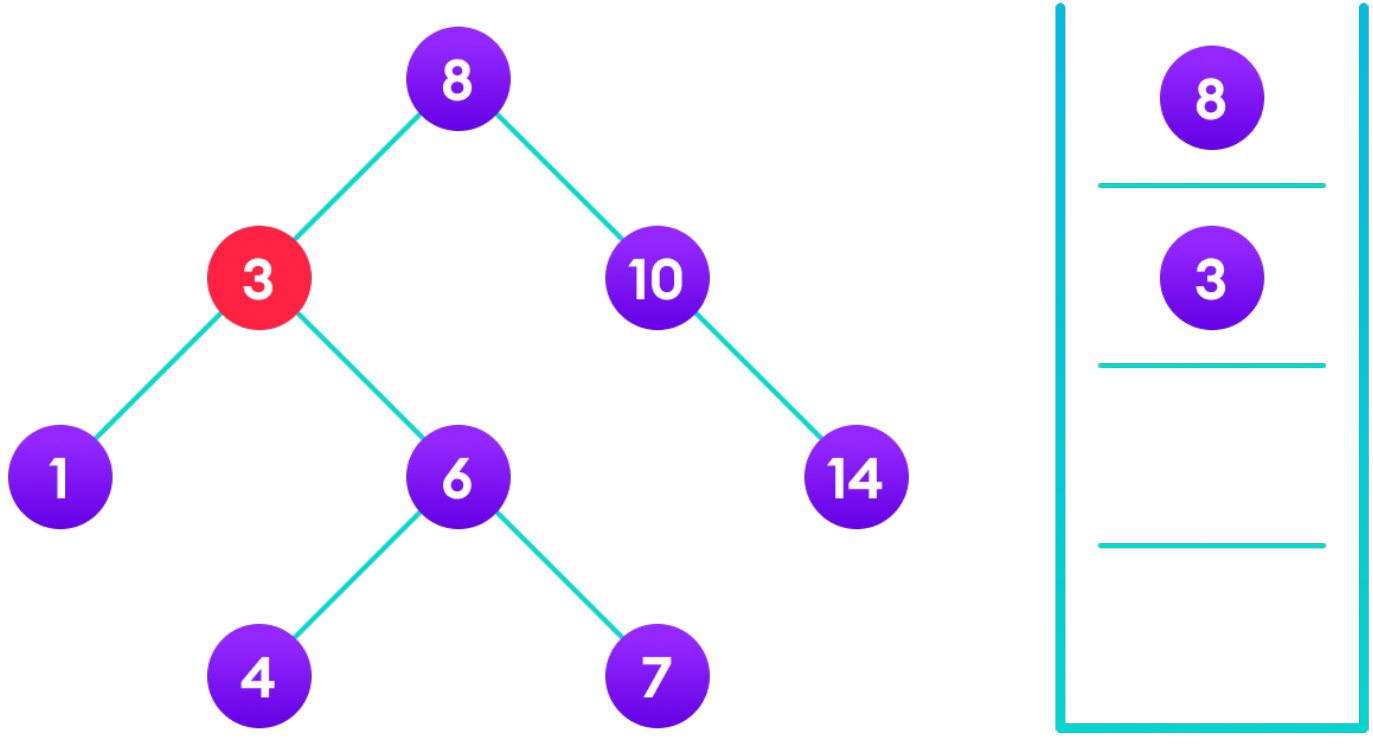
Every node in a tree must have at least 3 components：

- the data
- pointer to the left child.
- pointer to the right child.

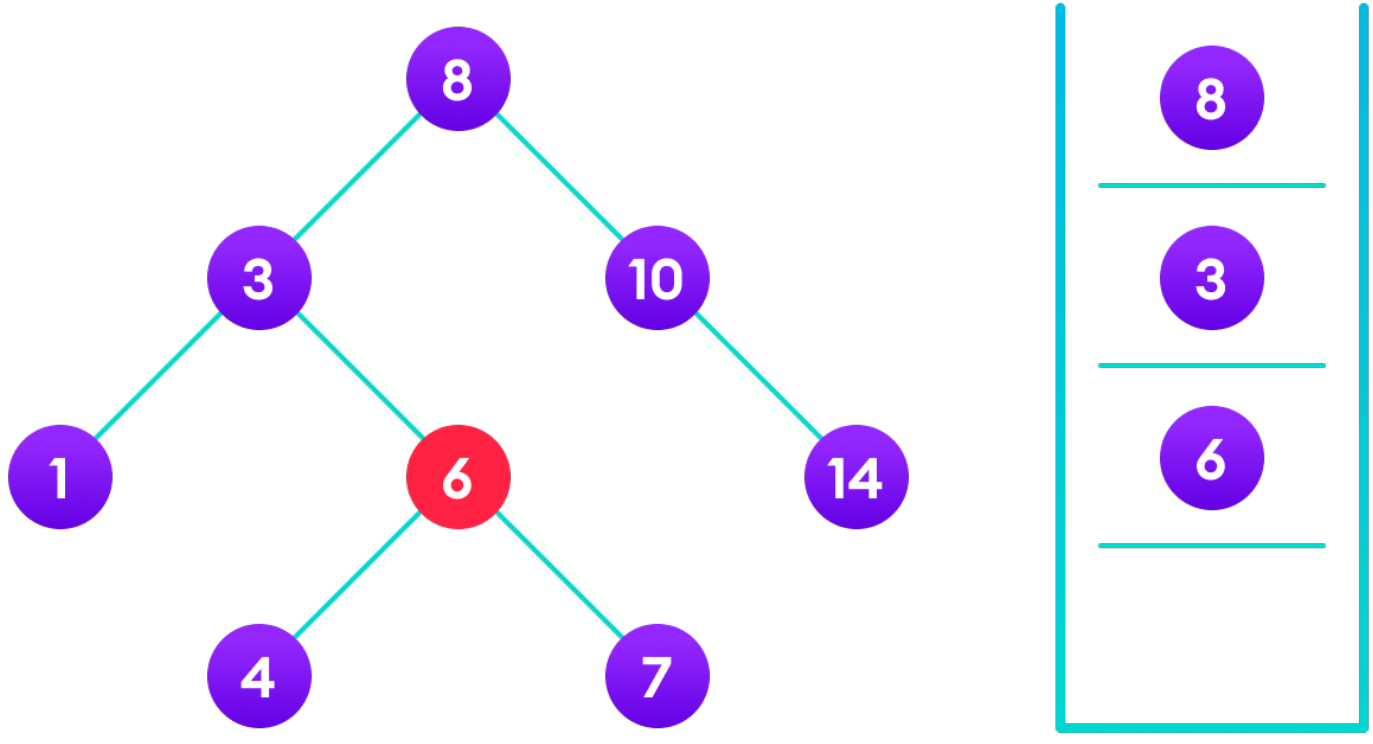




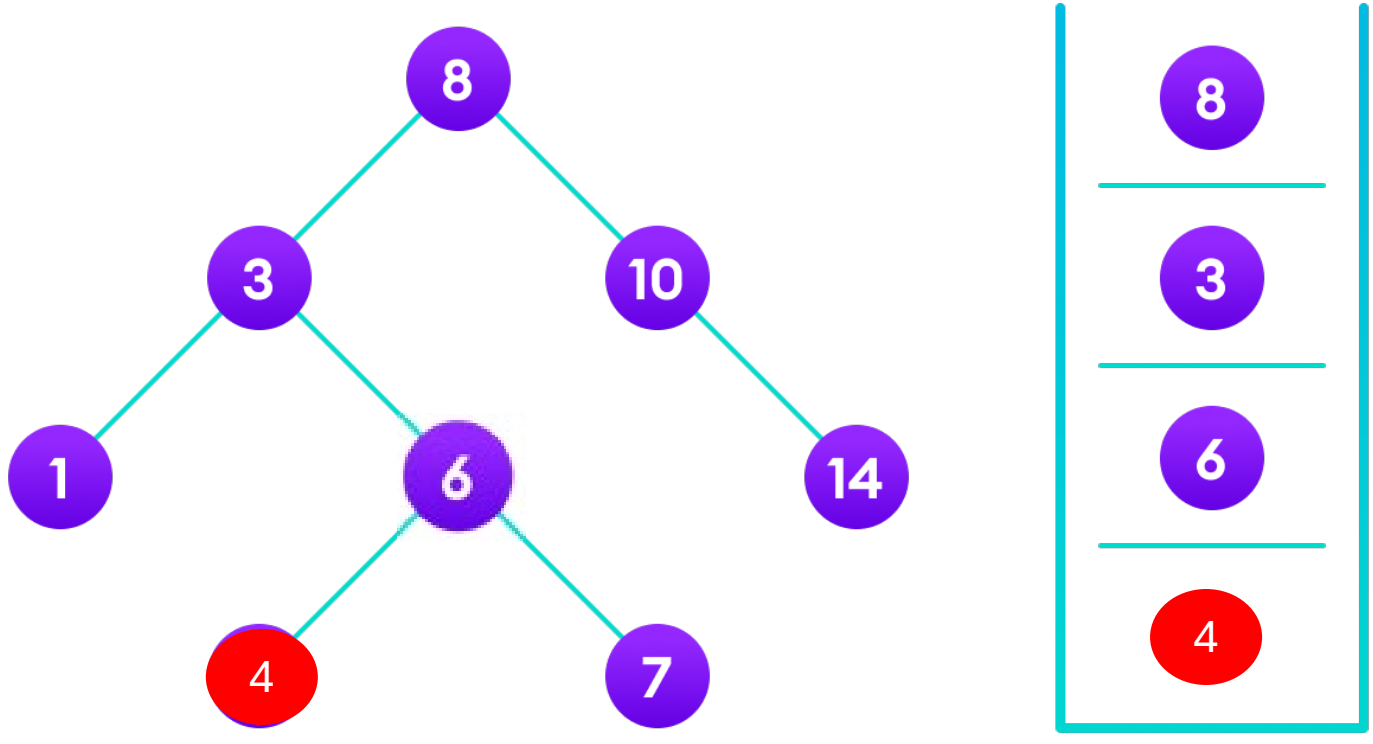
4 is not found so, traverse through the left subtree of 8



4 is not found so, traverse through the right subtree of 3



4 is not found so, traverse through the left subtree of 6



4 is found

Basic Operations of BST


THE MOST BASIC
BINARY SEARCH TREE

code

```cpp
#include <iostream>
using namespace std;

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

// Fungsi untuk cipta node baru
Node* createNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->left = newNode->right = nullptr;
    return newNode;
}

// Fungsi untuk masukkan nilai ke dalam BST
Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return createNode(value);
    }

    if (value < root->data) {
        root->left = insert(root->left, value);
    }
    else {
        root->right = insert(root->right, value);
    }

    return root;
}
```

```cpp
// Fungsi inorder traversal (kiri - root - kanan)
void inorder(Node* root) {
    if (root != nullptr) {
        inorder(root->left);
        cout << root->data << " ";
        inorder(root->right);
    }
}

int main() {
    Node* root = nullptr;

    root = insert(root, 8);
    insert(root, 3);
    insert(root, 10);
    insert(root, 1);
    insert(root, 6);

    cout << "Inorder traversal: ";
    inorder(root);
    cout << endl;

    return 0;
}
```

code

```cpp
#include <iostream>
using namespace std;

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

// Fungsi untuk cipta node baru
Node* createNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->left = newNode->right = nullptr;
    return newNode;
}

// Fungsi untuk masukkan nilai ke dalam BST
Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return createNode(value);
    }

    if (value < root->data) {
        root->left = insert(root->left, value);
    }
    else {
        root->right = insert(root->right, value);
    }

    return root;
}
```

- Ini struktur asas untuk satu node dalam BST.
- Setiap node simpan nilai (data) dan ada dua cabang: kiri (left) dan kanan (right).
- Ini memang kena ada, dah mmg macam ni cara nak implementnya. Ikut aje

code

```cpp
#include <iostream>
using namespace std;

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

// Fungsi untuk cipta node baru
Node* createNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->left = newNode->right = nullptr;
    return newNode;
}

// Fungsi untuk masukkan nilai ke dalam BST
Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return createNode(value);
    }

    if (value < root->data) {
        root->left = insert(root->left, value);
    }
    else {
        root->right = insert(root->right, value);
    }

    return root;
}
```

- Buat node baru dengan nilai **value**.
  - Kita dapat nilai value ni dari data yang dimasukkan dekat main
  - So kena create new node
  - Bila dah create node kita simpan la data tu takkan la tak simpan. So simpan dalam node newNode->data tu
  - Tapi left and right tak set lagi sebab setiap node baru celah mane ada left and right lagi, tunggu dulu. Sebab kita akan kembali ke function insert balik.
- Inisialisasi left dan right kepada **nullptr**.

code

```cpp
#include <iostream>
using namespace std;

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

// Fungsi untuk cipta node baru
Node* createNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->left = newNode->right = nullptr;
    return newNode;
}

// Fungsi untuk masukkan nilai ke dalam BST
Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return createNode(value);
    }

    if (value < root->data) {
        root->left = insert(root->left, value);
    }
    else {
        root->right = insert(root->right, value);
    }

    return root;
}
```

- Kalau pokok kosong **(root == nullptr),** cipta node baru.
- Kalau **value** lebih kecil daripada **data**, masuk ke kiri.
- Kalau lebih besar atau sama, masuk ke kanan.
- Fungsi ini rekursif — dia akan panggil diri sendiri sampai jumpa tempat yang sesuai untuk masukkan nilai tu.

step-by-step macam mana fungsi insert() tu berfungsi

```
root = insert(root, 8);
insert(root, 3);
insert(root, 10);
insert(root, 1);
insert(root, 6);
```

📥 Proses insert(root, 3)

- Kita tengok macam mana fungsi rekursif ni bekerja.
- Panggil insert(root, 3)
  - root sekarang ialah 8
  - 3 < 8 → pergi left
- Panggil insert(root->left, 3)
  - root->left sekarang nullptr
  - Jadi cipta createNode(3) dan letak sebagai root->left

📥 Proses insert(root, 3)

insert(8, 3)
└── insert(NULL, 3) → createNode(3)

Proses insert(root, 10)

- Panggil insert(root, 10)
  - 10 > 8 → pergi right
- Panggil insert(root->right, 10)
  - nullptr → cipta node baru

```
insert(8, 10)
└── insert(NULL, 10) → createNode(10)
```

📥 Proses insert(root, 1)

- 1 < 8 → pergi left
- 1 < 3 → pergi left
- root->left->left = nullptr → createNode(1)

```
insert(8, 1)
└── insert(3, 1)
    └── insert(NULL, 1) → createNode(1)
```

Proses insert(root, 6)

- 6 < 8 → kiri
- 6 > 3 → kanan
- root->left->right = nullptr → createNode(6)

```
insert(8, 6)
└── insert(3, 6)
    └── insert(NULL, 6) → createNode(6)
```

💡 Visual: Setiap insert() ibarat satu "kotak atas kotak" dalam rekursif stack

insert(8, 6)

↓

insert(3, 6)

↓

insert(NULL, 6)

↑ return Node(6)

↑ update root->right

↑ update root

Apa yang berlaku

Langkah	Nilai Masuk	Root Sekarang	Perbandingan	Arah	Tindakan
1	8	nullptr	-	-	createNode(8) jadi root
2	3	8	3 < 8	kiri	pergi insert(3) kat kiri
3		nullptr	-	-	createNode(3) kat kiri 8
4	10	8	10 > 8	kanan	pergi insert(10) kat kanan
5		nullptr	-	-	createNode(10) kat kanan 8
6	1	8	1 < 8	kiri	pergi ke anak kiri (3)
7		3	1 < 3	kiri	pergi ke anak kiri (nullptr)
8		nullptr	-	-	createNode(1) kat kiri 3
9	6	8	6 < 8	kiri	pergi ke anak kiri (3)
10		3	6 > 3	kanan	pergi ke anak kanan (nullptr)
11		nullptr	-	-	createNode(6) kat kanan 3

code

- Cara paling biasa untuk cetak BST dalam urutan menaik.
- Guna rekursi untuk lalu:
  - Subtree kiri
  - Root
  - Subtree kanan

```
// Fungsi inorder traversal (kiri - root - kanan)
void inorder(Node* root) {
    if (root != nullptr) {
        inorder(root->left);
        cout << root->data << " ";
        inorder(root->right);
    }
}
```

```
int main() {
    Node* root = nullptr;

    root = insert(root, 8);
    insert(root, 3);
    insert(root, 10);
    insert(root, 1);
    insert(root, 6);

    cout << "Inorder traversal: ";
    inorder(root);
    cout << endl;

    return 0;
}
```

code

- Mula dengan pokok kosong (root = nullptr)
- Masukkan nilai satu per satu ke dalam BST.
- Cetak hasil inorder traversal, yang akan bagi output dalam bentuk **1 3 6 8 10**.

```cpp
// Fungsi inorder traversal (kiri - root - kanan)
void inorder(Node* root) {
    if (root != nullptr) {
        inorder(root->left);
        cout << root->data << " ";
        inorder(root->right);
    }
}

int main() {
    Node* root = nullptr;

    root = insert(root, 8);
    insert(root, 3);
    insert(root, 10);
    insert(root, 1);
    insert(root, 6);

    cout << "Inorder traversal: ";
    inorder(root);
    cout << endl;

    return 0;
}
```

Other Operations?

IsEmpty()

CountNodes()

Search()

FindMin()

FindMax()

Delete()

IsEmpty()

```
// Semak jika BST kosong
bool IsEmpty(Node* root) {
    return root == nullptr;
}
```

DETAILS

Baris 1:

- ❑ Fungsi ini ambil satu parameter: root — iaitu root node pokok.
- ❑ return root == nullptr; — semak sama ada root kosong atau tidak.
  - Kalau root == nullptr, maksudnya pokok kosong, return true.
  - Kalau tak kosong, return false.

CountNodes()

```
// Cari jumlah node dalam BST
int CountNodes(Node* root) {
    if (root == nullptr) return 0;
    return 1 + CountNodes(root->left) +
CountNodes(root->right);
}
```

DETAILS

Baris 1:

❑ Fungsi ambil root node, dan return bilangan node dalam keseluruhan pokok.

Baris 2:

❑ Kalau root kosong, return 0.

Baris 3:

❑ Kalau tak kosong, kita kira:

- 1 untuk root sekarang.
- CountNodes(root->left) untuk subtree kiri.
- CountNodes(root->right) untuk subtree kanan.

❑ Semua ditambah untuk dapat total.

Search()

```
// Cari nilai dalam BST
bool Search(Node* root, int item) {
    if (root == nullptr) return false;
    if (item == root->data) return true;
    else if (item < root->data) return Search(root->left, item);
    else return Search(root->right, item);
}
```

DETAILS

Baris 1:

❑ Ambil root dan nilai item yang nak dicari.

Baris 2:

❑ Kalau root kosong → kita tak jumpa nilai → return false.

Baris 3:

❑ Kalau item sama dengan data node sekarang → jumpa → return true.

Baris 4–5:

❑ Kalau item lebih kecil → pergi kiri (root->left).

❑ Kalau lebih besar → pergi kanan (root->right).

Fungsi ni guna rekursi untuk menyelam ke bawah pokok ikut logik BST.

FindMin()

```
// Cari nilai minimum dalam BST
int FindMin(Node* root) {
    if (root == nullptr) {
        cout << "Tree is empty.\n";
        return -1;
    }
    while (root->left != nullptr) {
        root = root->left;
    }
    return root->data;
}
```

DETAILS

Baris 1–3:

- ❑ Kalau pokok kosong, cetak mesej dan return -1 (atau boleh tukar ikut kesesuaian).

Baris 4–6:

- ❑ Loop ke kiri selagi ada node kiri, sebab dalam BST, nilai paling kecil berada paling kiri.

Baris 7:

- ❑ Bila dah sampai paling kiri, root->data itulah nilai terkecil.

FindMax()

```
// Cari nilai maksimum dalam BST
int FindMax(Node* root) {
    if (root == nullptr) {
        cout << "Tree is empty.\n";
        return -1;
    }
    while (root->right != nullptr) {
        root = root->right;
    }
    return root->data;
}
```

DETAILS

Baris 1–3:

- ❑ Kalau pokok kosong, cetak mesej dan return -1 (atau boleh tukar ikut kesesuaian).

Baris 4–6:

- ❑ Loop ke kanan selagi ada node kanan, sebab dalam BST, nilai paling BESAR berada paling kanan.

Baris 7:

- ❑ Bila dah sampai paling kanan, root->data itulah nilai terkecil.

Delete()

```
// Padam node dari BST
Node* Delete(Node* root, int item) {
    if (root == nullptr) return root;

    if (item < root->data) {
        root->left = Delete(root->left, item);
    }
    else if (item > root->data) {
        root->right = Delete(root->right, item);
    }
    else {
        // 0 atau 1 anak
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }

        // 2 anak - guna nilai terkecil dari subtree kanan
        int minValue = FindMin(root->right);
        root->data = minValue;
        root->right = Delete(root->right, minValue);
    }

    return root;
}
```

Apa dia buat:

Padam node dari BST. Ada 3 kes:

- Node tiada anak (leaf)
- Node ada satu anak
- Node ada dua anak — ganti dengan min value di right subtree

Delete()

```
// Padam node dari BST
Node* Delete(Node* root, int item) {
    if (root == nullptr) return root;

    if (item < root->data) {
        root->left = Delete(root->left, item);
    }
    else if (item > root->data) {
        root->right = Delete(root->right, item);
    }
    else {
        // 0 atau 1 anak
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }

        // 2 anak - guna nilai terkecil dari subtree kanan
        int minValue = FindMin(root->right);
        root->data = minValue;
        root->right = Delete(root->right, minValue);
    }

    return root;
}
```

DETAILS

Penjelasan Baris demi Baris:

Baris 1:

- ❑ Fungsi ambil root dan nilai yang nak dipadam (item).

Baris 2:

- ❑ Kalau root kosong, return nullptr.

Baris 3–6:

- ❑ Kalau item lebih kecil → terus padam di kiri.Kalau lebih besar → padam di kanan.

Baris 7:

- ❑ Kalau item == root->data, bermakna node ni yang nak dipadam.

Sekarang kita kena handle 3 kes:

Delete()

Kes 1 – Tiada Anak Kiri

Kita simpan anak kanan dalam temp, delete node sekarang, dan return anak kanan.

```
// Padam node dari BST
Node* Delete(Node* root, int item) {
    if (root == nullptr) return root;

    if (item < root->data) {
        root->left = Delete(root->left, item);
    }
    else if (item > root->data) {
        root->right = Delete(root->right, item);
    }
    else {
        // 0 atau 1 anak
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }

        // 2 anak - guna nilai terkecil dari subtree kanan
        int minValue = FindMin(root->right);
        root->data = minValue;
        root->right = Delete(root->right, minValue);
    }

    return root;
}
```

Delete()

Kes 2 – Tiada Anak Kanan

Sama juga, tapi kita simpan anak kiri, delete root, dan return anak kiri.

```
// Padam node dari BST
Node* Delete(Node* root, int item) {
    if (root == nullptr) return root;

    if (item < root->data) {
        root->left = Delete(root->left, item);
    }
    else if (item > root->data) {
        root->right = Delete(root->right, item);
    }
    else {
        // 0 atau 1 anak
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }

        // 2 anak - guna nilai terkecil dari subtree kanan
        int minValue = FindMin(root->right);
        root->data = minValue;
        root->right = Delete(root->right, minValue);
    }

    return root;
}
```

Delete()

Kes 3 – Ada dua anak

- Kita cari nilai terkecil di right subtree (FindMin).
- Gantikan data node sekarang dengan nilai itu.
- Padam node yang ada nilai tersebut di right subtree (supaya tak duplicate).

```
// Padam node dari BST
Node* Delete(Node* root, int item) {
    if (root == nullptr) return root;

    if (item < root->data) {
        root->left = Delete(root->left, item);
    }
    else if (item > root->data) {
        root->right = Delete(root->right, item);
    }
    else {
        // 0 atau 1 anak
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }

        // 2 anak - guna nilai terkecil dari subtree kanan
        int minValue = FindMin(root->right);
        root->data = minValue;
        root->right = Delete(root->right, minValue);
    }

    return root;
}
```

Delete()

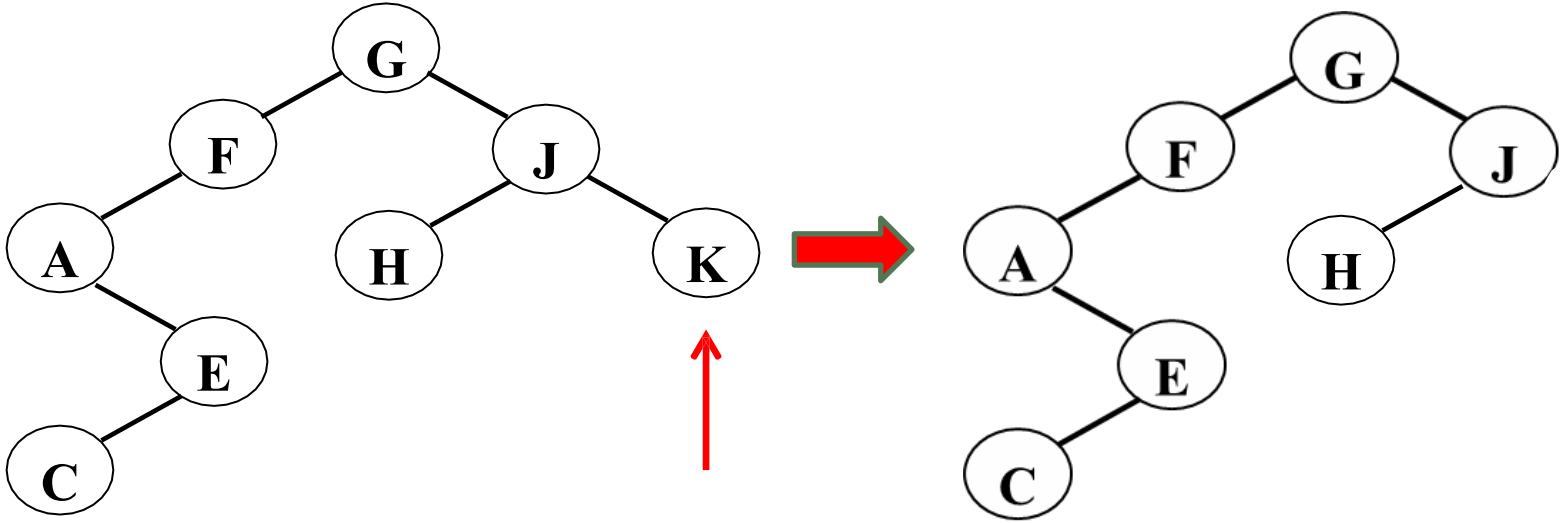
```
// Padam node dari BST
Node* Delete(Node* root, int item) {
    if (root == nullptr) return root;

    if (item < root->data) {
        root->left = Delete(root->left, item);
    }
    else if (item > root->data) {
        root->right = Delete(root->right, item);
    }
    else {
        // 0 atau 1 anak
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }

        // 2 anak - guna nilai terkecil dari subtree kanan
        int minValue = FindMin(root->right);
        root->data = minValue;
        root->right = Delete(root->right, minValue);
    }

    return root;
}
```

⬅ END return root;

- Pastikan node yang kita delete telah diproses dan root yang terkini dikembalikan untuk sambung binaan pokok.

Example of Deletion

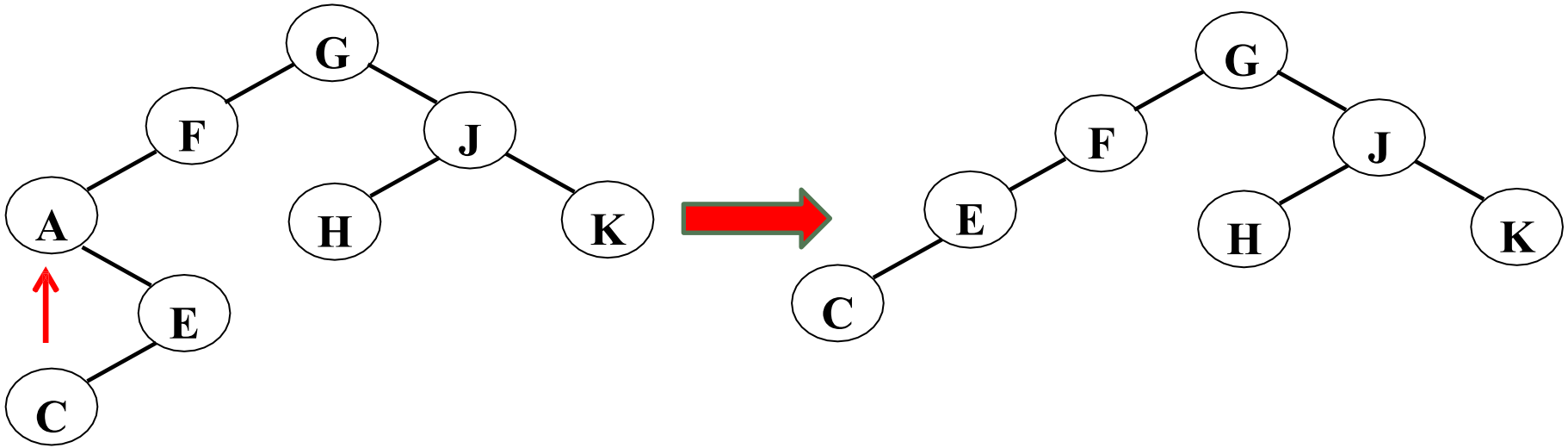
- If the node to be deleted is **a leaf node** - has no children, the process is simple.



Delete node K

Example of Deletion

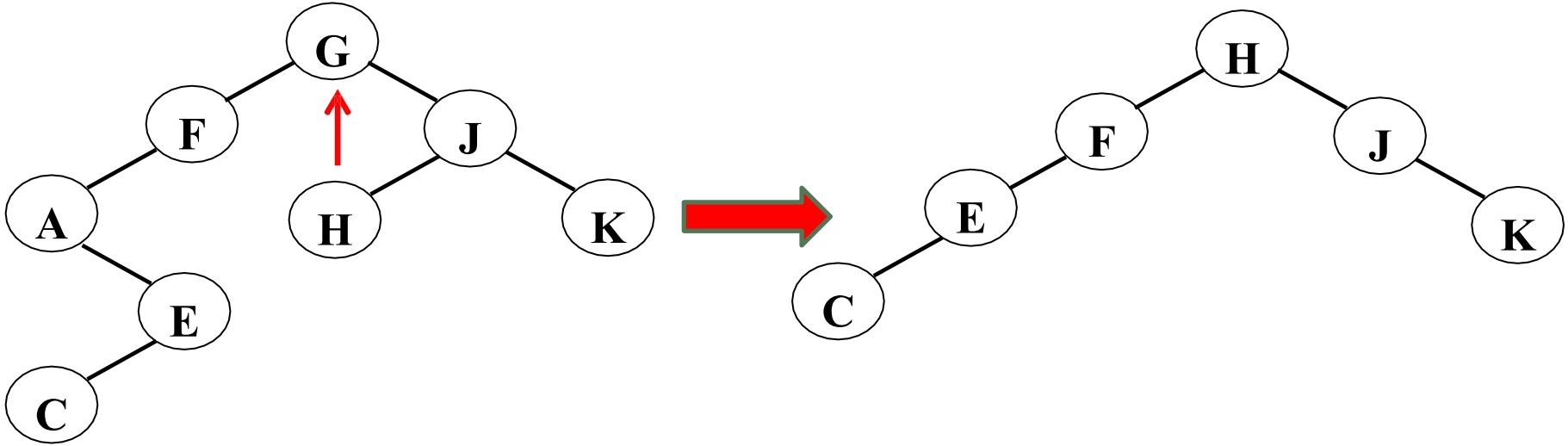
- If the node to be deleted **has one child**, the subtree for its child will be pushed up



Delete node A

Example of Deletion

- If the node to be deleted **has two children,** move up the smallest node in the right subtree to replace the node to be deleted



Delete node G

All Operations

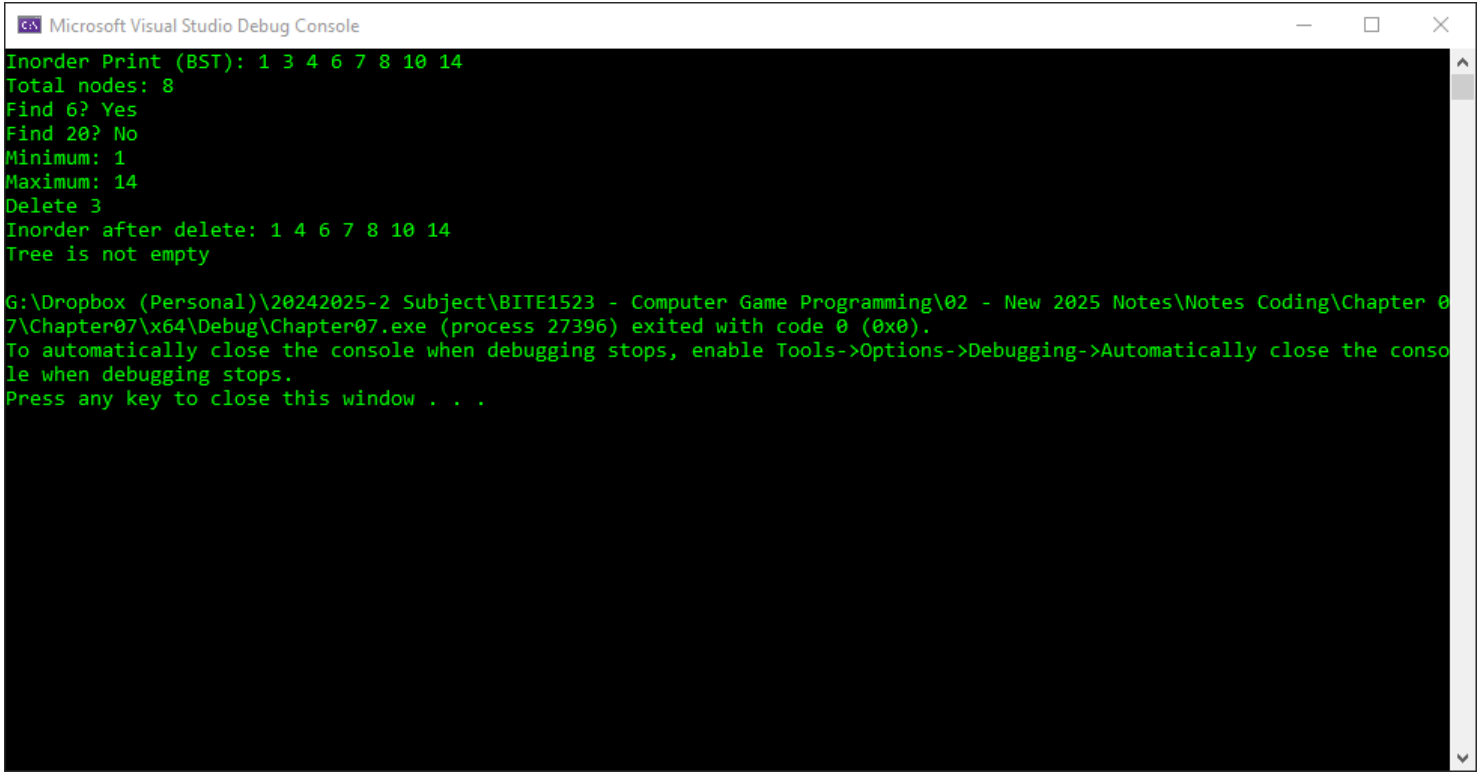
Fungsi	Status	Tujuan
Insert()	✅ Dah ada	Masukkan nilai dalam BST
Print() (Inorder)	✅ Dah ada	Cetak isi BST dari kecil ke besar
IsEmpty()	✅ Dah ada	Semak sama ada pokok kosong
CountNodes()	✅ Dah ada	Kira bilangan node
Search()	✅ Dah ada	Cari nilai tertentu dalam BST
FindMin()	✅ Dah ada	Cari nilai paling kecil dalam BST
FindMax()	✅ Dah ada	Cari nilai paling besar dalam BST
Delete()	✅ Dah ada	Padam node dari BST mengikut peraturan

Full Source Code BST – Download at https://asyrani.com/bite1523-computer-game-programming/

Let say main function:

```
int main() {
    Node* root = nullptr;

    // Insert
    root = Insert(root, 8);
    Insert(root, 3);
    Insert(root, 10);
    Insert(root, 1);
    Insert(root, 6);
    Insert(root, 14);
    Insert(root, 4);
    Insert(root, 7);

    cout << "Inorder Print (BST): ";
    Print(root);
    cout << endl;

    cout << "Total nodes: " << CountNodes(root) << endl;

    cout << "Find 6? " << (Search(root, 6) ? "Yes" : "No") << endl;
    cout << "Find 20? " << (Search(root, 20) ? "Yes" : "No") << endl;

    cout << "Minimum: " << FindMin(root) << endl;
    cout << "Maximum: " << FindMax(root) << endl;

    cout << "Delete 3" << endl;
    root = Delete(root, 3);
    cout << "Inorder after delete: ";
    Print(root);
    cout << endl;

    cout << "Tree is " << (IsEmpty(root) ? "empty" : "not empty") << endl;

    return 0;
}
```

Output


Microsoft Visual Studio Debug Console
Inorder Print (BST): 1 3 4 6 7 8 10 14
Total nodes: 8
Find 6? Yes
Find 20? No
Minimum: 1
Maximum: 14
Delete 3
Inorder after delete: 1 4 6 7 8 10 14
Tree is not empty
G:\Dropbox (Personal)\20242025-2 Subject\BITE1523 - Computer Game Programming\02 - New 2025 Notes\Notes Coding\Chapter 07\Chapter07\x64\Debug\Chapter07.exe (process 27396) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

✅ Traversal Tambahan dalam BST

1. Preorder Traversal (Root → Left → Right)
2. Postorder Traversal (Left → Right → Root)
3. Level-order Traversal (By level, kiri ke kanan guna Queue)

🧠 Apa Kegunaan Masing-masing?

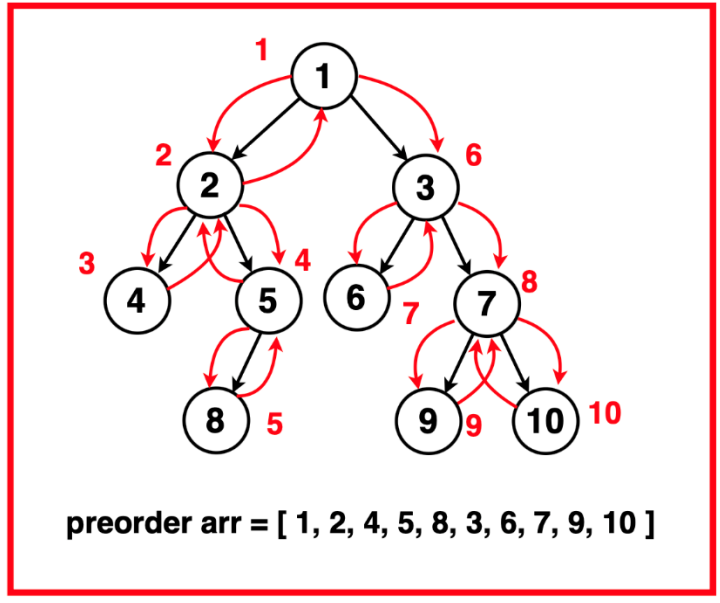
Jenis Traversal	Urutan Lalu	Kegunaan
Inorder	Left → Root → Right	Dapat susunan menaik dalam BST
Preorder	Root → Left → Right	Guna untuk **salin pokok** atau **print struktur**
Postorder	Left → Right → Root	Guna untuk **padam pokok** dari bawah ke atas
Level-order	Baris demi baris (queue)	Guna untuk **analisis level**, **breadth-first traversal**

✅ Kod Tambahan: Preorder, Postorder, Level-order (Tambah Dalam Kod Tadi)

- #include <queue> // Untuk Level-order

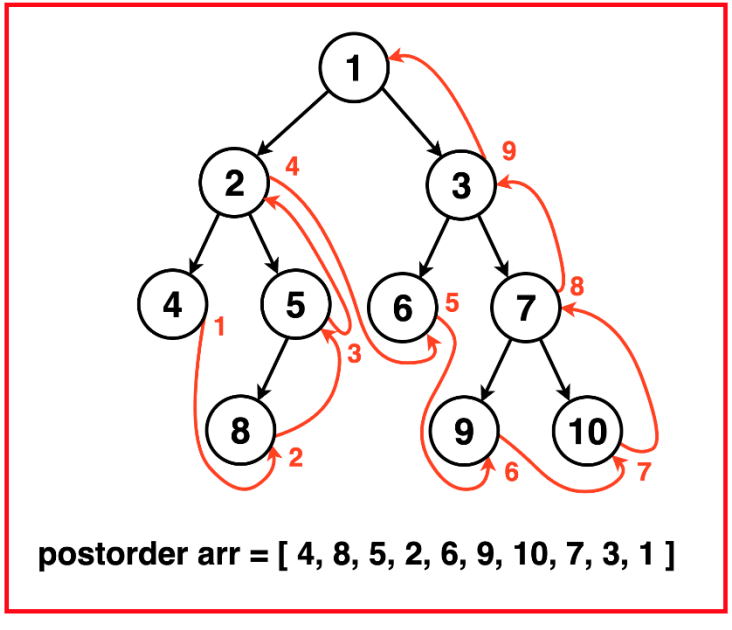
void Preorder(Node* root)

```
// Preorder: Root - Left - Right
void Preorder(Node* root) {
    if (root != nullptr) {
        cout << root->data << " ";
        Preorder(root->left);
        Preorder(root->right);
    }
}
```



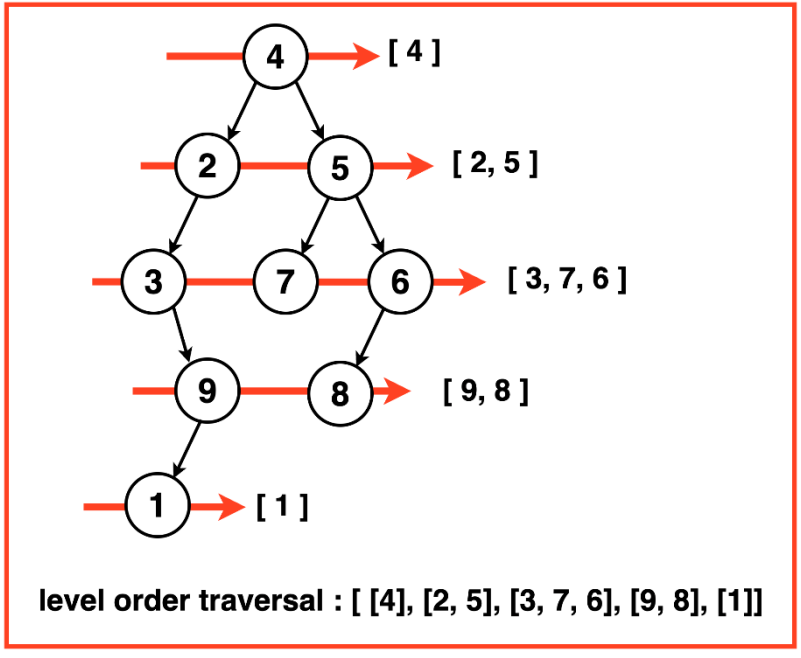
void Postorder(Node* root)

```
// Postorder: Left - Right - Root
void Postorder(Node* root) {
    if (root != nullptr) {
        Postorder(root->left);
        Postorder(root->right);
        cout << root->data << " ";
    }
}
```



void LevelOrder(Node* root)

```
// Level-order: Guna Queue
void LevelOrder(Node* root) {
    if (root == nullptr) return;

    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
        Node* current = q.front();
        q.pop();

        cout << current->data << " ";

        if (current->left != nullptr)
            q.push(current->left);
        if (current->right != nullptr)
            q.push(current->right);
    }
}
```



void LevelOrder(Node* root)

```
// Level-order: Guna Queue
void LevelOrder(Node* root) {
    if (root == nullptr) return;

    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
        Node* current = q.front();
        q.pop();

        cout << current->data << " ";

        if (current->left != nullptr)
            q.push(current->left);
        if (current->right != nullptr)
            q.push(current->right);
    }
}
```

Kalau pokok kosong, kita tak buat apa-apa — terus keluar.

- Kita cipta queue q yang akan simpan node yang belum dilawat.
- Masukkan root dulu (node paling atas/paling awal).

void LevelOrder(Node* root)

```
// Level-order: Guna Queue
void LevelOrder(Node* root) {
    if (root == nullptr) return;

    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
        Node* current = q.front();
        q.pop();

        cout << current->data << " ";

        if (current->left != nullptr)
            q.push(current->left);
        if (current->right != nullptr)
            q.push(current->right);
    }
}
```

Selagi masih ada node dalam queue (belum habis traverse), kita teruskan.

- Ambil node paling depan dalam **queue → simpan** dalam **current**
- Keluarkan dari **queue** (sebab kita akan proses dia sekarang)

void LevelOrder(Node* root)

```
// Level-order: Guna Queue
void LevelOrder(Node* root) {
    if (root == nullptr) return;

    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
        Node* current = q.front();
        q.pop();

        cout << current->data << " ";

        if (current->left != nullptr)
            q.push(current->left);
        if (current->right != nullptr)
            q.push(current->right);
    }
}
```

Cetak nilai node sekarang

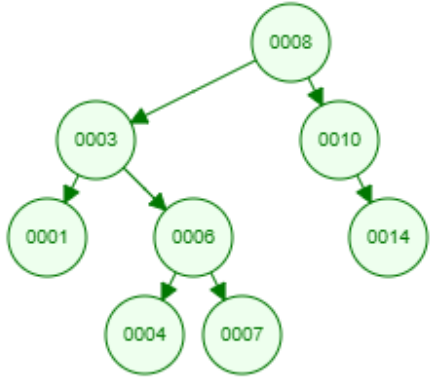
- Kalau ada anak kiri, masuk dalam queue untuk dilawat kemudian.

void LevelOrder(Node* root)

```
// Level-order: Guna Queue
void LevelOrder(Node* root) {
    if (root == nullptr) return;

    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
        Node* current = q.front();
        q.pop();

        cout << current->data << " ";

        if (current->left != nullptr)
            q.push(current->left);
        if (current->right != nullptr)
            q.push(current->right);
    }
}
```

- Sama juga, masuk anak kanan jika wujud.

void LevelOrder(Node* root)



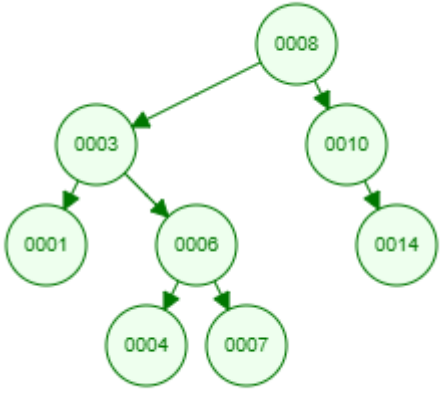

```
// Level-order: Guna Queue
void LevelOrder(Node* root) {
    if (root == nullptr) return;

    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
        Node* current = q.front();
        q.pop();

        cout << current->data << " ";

        if (current->left != nullptr)
            q.push(current->left);
        if (current->right != nullptr)
            q.push(current->right);
    }
}
```

Queue: [8] Print: -
→ Pop 8 → Print 8 Queue: [3, 10]
→ Pop 3 → Print 3 Queue: [10, 1, 6]
→ Pop 10 → Print 10 Queue: [1, 6, 14]
→ Pop 1 → Print 1 Queue: [6, 14]
→ Pop 6 → Print 6 Queue: [14]
→ Pop 14 → Print 14 Queue: []
DONE ✓

📌 Contoh penggunaan dalam main():

```
cout << "Preorder: ";
Preorder(root);
cout << endl;

cout << "Postorder: ";
Postorder(root);
cout << endl;

cout << "Level-order: ";
LevelOrder(root);
cout << endl;
```

Contoh Output Kalau Tree:



Inorder: 1 3 4 6 7 8 10 14
Preorder: 8 3 1 6 4 7 10 14
Postorder: 1 4 7 6 3 14 10 8
Level-order: 8 3 10 1 6 14 4 7

https://www.cs.usfca.edu/~galles/visualization/BST.html