Data structures and Algorithms

Note06-Introduction to trees part01

Edited By

Arshia Gharooni

2023

0.1 Introduction

Trees are a fundamental data structure in computer science and are used to represent hierarchical relationships between data. Trees have applications in many areas of computer science, including algorithms, databases, graphics, and artificial intelligence.

A tree is a data structure that consists of nodes connected by edges. Each node in a tree can have zero or more child nodes, except for the root node, which has no parent node. The edges in a tree represent the relationships between the nodes. Trees are used to represent hierarchical relationships between data.

0.2 Some importation definitions

0.2.1 Subtree

A subtree is a subset of a tree that can be separated and treated as its own smaller tree. It consists of a node, called the root of the subtree, and all of its descendants. Formally, let T be a tree with root r, then the subtree rooted at node u is the tree $T_u = (V_u, E_u)$ where V_u is the set of nodes that are descendants of u in T, and E_u is the set of edges that connect these nodes.

0.2.2 Depth of a Node

The depth of a node u in a tree T is denoted by $d_T(u)$ and defined as the length of the path from the root of the tree to u. The length of a path is the number of edges on the path.

Trivially, we can define the depth of a node u as follows:

- $d_T(u) = 0$, if u is the root of T.
- $d_T(u) = d_T(v) + 1$, if u is a child of v in T.

0.2.3 Height of a Node

The height of a node u in a tree T is denoted by $h_T(u)$ and defined as the length of the longest path from u to a leaf node in T.

Trivially, we can define the height of a node u as follows:

- $h_T(u) = 0$, if u is a leaf node.
- $h_T(u) = \max(h_T(v)) + 1$, if u is not a leaf node and has children v_1, v_2,..., v_k in T.

0.3 Binary trees

A binary tree is a tree data structure where each node has at most two children, referred to as the left child and the right child. The left child is usually smaller than the right child in value. The nodes are connected by edges or pointers. The topmost node in a binary tree is called the root, while the nodes without children are called leaf nodes.

0.4 Tree Traversals

Tree traversal is a fundamental operation on trees that is used in many algorithms, including searching, sorting, and indexing. In this lecture note, we will discuss the three main types of tree traversal: pre-order, in-order, and post-order.

0.4.1 Tree Traversal Types

Pre-order Traversal

Pre-order traversal is a method of visiting each node in a tree in a specific order. In pre-order traversal, we visit the root node first, followed by the left subtree, and then the right subtree. In other words, we process the node before its children.

In-order Traversal(Traversal order)

In-order traversal is another method of visiting each node in a tree in a specific order. In in-order traversal, we visit the left subtree first, followed by the root node, and then the right subtree. In other words, we process the node between its children.

Post-order Traversal

Post-order traversal is the third and final method of visiting each node in a tree in a specific order. In post-order traversal, we visit the left subtree first, followed by the right subtree, and then the root node. In other words, we process the node after its children.

0.5 Tree Navigation

Tree navigation is a fundamental operation when we are talking about trees and their implementation.

0.5.1 Finding the First Node in the Traversal Order

Given a node <X>, we want to find the first node in its subtree according to some traversal order so the whole algorithm will be like:

1. If <X> has left child, recursively return the first node in the left subtree
2. Otherwise, <X> is the first node, so return it

The time complexity of this algorithm is O(h), where h is the height of the tree. This is because in the worst case, we may have to traverse the entire height of the tree to find the first node. In the best case, the first node may be the root node, and we can return it in constant time.

0.5.2 Finding the Successor of a Node

Given a node <X>, we want to find its successor in the traversal order, which is the next node we would visit if we were to continue the traversal after visiting <X>. To find the successor, we use the following algorithm:

1. If <X> has a right child, return the first node in the right subtree.
2. Otherwise, return the lowest ancestor of <X> for which <X> is in its left subtree.

The time complexity of this algorithm is also O(h), where h is the height of the tree. This is because, in the worst case, we may have to traverse the entire height of the tree to find the successor. In the best case, the successor may be the right child of <X>, and we can return it in constant time.

0.6 Dynamic Operations on Binary Search Trees

Binary Search Trees (BST) are a popular data structure that supports various dynamic operations such as insertion and deletion of nodes. This lecture note covers the dynamic operations of BST and their implementation.

0.6.1 Insertion operation

Suppose we want to insert a new node <Y> after an existing node <X> in the traversal order. We can perform the insertion operation as follows:

1. If <X> has no right child, make <Y> the right child of <X>.
2. Otherwise, make <Y> the left child of <X>'s successor (which cannot have a left child).

The running time of this insertion operation is O(h), where h is the height of the tree.

0.6.2 Deletion operation

Suppose we want to remove an item from the subtree rooted at a node <X>. We can perform the deletion operation as follows:

1. If <X> is a leaf, detach it from the parent and return.
2. Otherwise, <X> has a child.
   (a) If <X> has a left child, swap items with the predecessor of <X> and recurse.
   (b) Otherwise, <X> has a right child, swap items with the successor of <X> and recurse.

The running time of this deletion operation is O(h).

0.7 Binary search tree (BST)

A Binary Search Tree (BST) is a fundamental data structure used in computer science for efficient searching, insertion, and deletion operations. It is a tree data structure in which each node has at most two children, known as left and right children. In addition, each node in a BST holds a key, which is used to order and search the tree. The key in each node is unique.

A BST is a particular type of tree where every node in the left subtree of a given node holds a smaller key value, while every node in the right subtree holds a larger key value. This ordering of keys in a BST allows for efficient searching, as the search algorithm can traverse the tree by comparing the search key with the keys in each node and following the appropriate subtree.

0.7.1 Properties of Binary Search Trees

A BST has several essential properties that make it a useful data structure for searching and sorting:

1. The key in each node is unique.
2. The left and right subtrees of a node are also BSTs.
3. All the keys in the left subtree of a node are less than the key in the node.
4. All the keys in the right subtree of a node are greater than the key in the node.

The first property is essential for the correct functioning of the BST. The second property ensures that we can apply the same search algorithm to the left and right subtrees, as they are also BSTs. The third and fourth properties provide the ordering of the keys in the tree, which enables efficient searching.

0.8 BST Operations

0.8.1 Search/find

The search operation is one of the most basic operations on a BST. It allows us to find a specific node in the tree by its value. The search operation can be implemented recursively or iteratively.

- Recursive search implementation:

```
def search(node, value):
    if node is None or node.value == value:
        return node
    if value < node.value:
        return search(node.left, value)
    else:
        return search(node.right, value)
```

- Iterative search implementation:

```
def search(node, value):
    while node is not None and node.value != value:
        if value < node.value:
            node = node.left
        else:
            node = node.right
    return node
```

0.8.2 Insertion

The insertion operation allows us to add a new node to the tree while maintaining the BST property. The insertion operation can be implemented recursively or iteratively.

- Recursive insertion implementation

```
def insert(node, value):
    if node is None:
        return Node(value)
    if value < node.value:
        node.left = insert(node.left, value)
    else:
        node.right = insert(node.right, value)
    return node
```

- Iterative insertion implementation:

```
def insert(node, value):
    if node is None:
        return Node(value)
    parent = None
    current = node
    while current is not None:
        parent = current
        if value < current.value:
            current = current.left
        else:
            current = current.right
    new_node = Node(value)
    if value < parent.value:
        parent.left = new_node
    else:
        parent.right = new_node
    return node
```

0.8.3 Deletion

The deletion operation in BSTs removes a node with a given value from the tree while maintaining the BST property. There are three cases to consider:

1. The node to be deleted is a leaf node (has no children).
2. The node to be deleted has one child.
3. The node to be deleted has two children.

In the first case, we simply remove the node from the tree. In the second case, we replace the node with its child. In the third case, we replace the node with its successor (the smallest node in its right subtree), and recursively delete the successor.

- Recursive deletion implementation

```
def delete(node, value):
    if node is None:
        return node
    if value < node.value:
        node.left = delete(node.left, value)
    elif value > node.value:
        node.right = delete(node.right, value)
    else:
        if node.left is None:
```

```
            return node.right
        elif node.right is None:
            return node.left
        else:
            successor = node.right
            while successor.left is not None:
                successor = successor.left
            node.value = successor.value
            node.right = delete(node.right, successor.value)
    return node
```

- Iterative Deletion implementation:

```
def delete(node, value):
    parent = None
    current = node
    while current is not None and current.value != value:
        parent = current
        if value < current.value:
            current = current.left
        else:
            current = current.right
    if current is None:
        return node
    if current.left is None:
        child = current.right
    elif current.right is None:
        child = current.left
    else:
        successor = current.right
        while successor.left is not None:
            successor = successor.left
        current.value = successor.value
        current = successor
        child = current.right
    if parent is None:
        node = child
    elif parent.left == current:
        parent.left = child
    else:
        parent.right = child
    return node
```