# What is a binary tree

A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. The nodes in a binary tree are connected by edges, and the topmost node is called the root. Each node in a binary tree contains a value, and the value of the left child is always less than or equal to the value of the parent, while the value of the right child is always greater than or equal to the value of the parent.

Binary trees are commonly used to represent hierarchical structures such as file systems, expression trees, and search trees. They can be traversed in several different ways, including in-order, pre-order, and post-order traversal. Binary trees can also be balanced, which means that the height of the left and right subtrees of each node differ by at most one, which can improve the efficiency of certain operations.

# What is the difference between a binary tree and a Binary Search Tree

The main difference between a binary tree and a Binary Search Tree (BST) is that in a BST, the values of the nodes follow a specific ordering rule.

In a binary tree, the values of the nodes can be arranged in any order, and there is no specific rule for organizing the nodes. In contrast, a BST is a binary tree in which the values of the left child are always less than the value of the parent, and the values of the right child are always greater than the value of the parent. This ordering rule makes it easier to search for a particular value in the tree.

The ordering rule of a BST also allows for efficient search and insertion operations, as they can be performed in O(log n) time in a balanced BST. On the other hand, searching or inserting a value in a binary tree can take up to O(n) time in the worst case, where n is the number of nodes in the tree.

In summary, a binary tree is a tree data structure in which each node has at most two children, while a BST is a binary tree in which the values of the nodes follow a specific ordering rule, which makes searching and inserting values more efficient.

# What is the possible gain of binary tree in terms of time complexity compared to linked lists

Binary trees can offer significant gains in terms of time complexity compared to linked lists for certain operations, such as searching for a specific value or inserting a new node.

In a linked list, searching for a particular value requires traversing the list from the beginning until the value is found or the end of the list is reached. This operation has a time complexity of O(n), where n is the length of the list.

In contrast, searching for a value in a binary search tree can be done in O(log n) time on average, and O(n) in the worst case scenario when the tree is unbalanced. This is because the search process can be guided by the ordering rule of the BST, which allows the search to skip over large portions of the tree.

Similarly, inserting a new node into a linked list requires traversing the list to find the correct insertion point, which also has a time complexity of O(n). In contrast, inserting a new node into a binary search tree can be done in O(log n) time on average and O(n) in the worst case scenario.

Overall, binary trees can offer significant gains in terms of time complexity compared to linked lists for certain operations, especially for searching and inserting values.

What are the depth, the height, the size of a binary tree

In a binary tree, the terms depth, height, and size refer to the characteristics of the tree:

Depth: The depth of a node in a binary tree is the number of edges from the root to that node. The depth of the root node is 0.

Height: The height of a binary tree is the maximum depth of any node in the tree. In other words, it is the length of the longest path from the root to a leaf node.

Size: The size of a binary tree is the total number of nodes in the tree, including the root node and all its descendants.

These characteristics are important in analyzing the structure and performance of binary trees. The depth and height of a binary tree can affect the time complexity of certain operations, such as searching, inserting, or deleting a node. The size of a binary tree can also affect the memory usage of the tree.

It's worth noting that the depth, height, and size of a binary tree can vary depending on the specific implementation of the tree. For example, a binary tree can be balanced or unbalanced, which affects its height and the time complexity of certain operations.

# What are the different traversal methods to go through a binary tree

There are three common traversal methods to go through a binary tree:

* In-order Traversal: In this traversal method, we visit the left subtree of the root node, then the root node, and finally the right subtree of the root node. For each subtree, we perform the same process recursively. In a binary search tree, the in-order traversal visits the nodes in ascending order.
* Pre-order Traversal: In this traversal method, we visit the root node, then the left subtree of the root node, and finally the right subtree of the root node. For each subtree, we perform the same process recursively.
* Post-order Traversal: In this traversal method, we visit the left subtree of the root node, then the right subtree of the root node, and finally the root node. For each subtree, we perform the same process recursively.
* There are also other traversal methods such as level-order traversal which visits each node in the tree level by level, starting from the root node.

Traversal methods are commonly used to perform different operations on binary trees, such as printing the values of the nodes in a certain order, searching for a specific value, or calculating the height of the tree.

# What is a complete, a full, a perfect, a balanced binary tree

Some common types of binary trees:

Complete Binary Tree: A complete binary tree is a binary tree in which all levels of the tree are fully filled, except possibly for the last level, which must be filled from left to right. In other words, all nodes are as far left as possible on each level.

Full Binary Tree: A full binary tree is a binary tree in which every node has either 0 or 2 children. In other words, every non-leaf node has exactly two children.

Perfect Binary Tree: A perfect binary tree is a binary tree in which every level of the tree is completely filled. This means that the tree has exactly 2^k-1 nodes, where k is the number of levels in the tree.

Balanced Binary Tree: A balanced binary tree is a binary tree in which the height of the left and right subtrees of any node differ by at most one. In other words, the tree is not lopsided or skewed to one side.

These types of binary trees have different characteristics and can be useful for different applications. For example, a complete binary tree can be efficiently stored in an array, while a perfect binary tree has a predictable number of nodes and a balanced binary tree offers efficient searching, insertion, and deletion operations.

Example 1.

**implementation of a binary search tree in C, with the basic operations of insertion, searching, and traversal:**

c

Copy code

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a binary search tree node

struct bst\_node {

int value;

struct bst\_node\* left;

struct bst\_node\* right;

};

// Define the function to create a new node

struct bst\_node\* create\_node(int value) {

struct bst\_node\* node = (struct bst\_node\*) malloc(sizeof(struct bst\_node));

node->value = value;

node->left = NULL;

node->right = NULL;

return node;

}

// Define the function to insert a new value into the binary search tree

struct bst\_node\* insert(struct bst\_node\* root, int value) {

if (root == NULL) {

return create\_node(value);

}

if (value < root->value) {

root->left = insert(root->left, value);

} else if (value > root->value) {

root->right = insert(root->right, value);

}

return root;

}

// Define the function to search for a value in the binary search tree

struct bst\_node\* search(struct bst\_node\* root, int value) {

if (root == NULL || root->value == value) {

return root;

}

if (value < root->value) {

return search(root->left, value);

}

return search(root->right, value);

}

// Define the function to perform an in-order traversal of the binary search tree

void inorder\_traversal(struct bst\_node\* root) {

if (root != NULL) {

inorder\_traversal(root->left);

printf("%d ", root->value);

inorder\_traversal(root->right);

}

}

// Define the main function to test the binary search tree implementation

int main() {

struct bst\_node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

printf("In-order Traversal: ");

inorder\_traversal(root);

printf("\n");

int search\_value = 40;

struct bst\_node\* search\_result = search(root, search\_value);

if (search\_result == NULL) {

printf("%d not found in the tree\n", search\_value);

} else {

printf("%d found in the tree\n", search\_value);

}

return 0;

}

This implementation creates a binary search tree with seven nodes, performs an in-order traversal of the tree, and searches for a specific value in the tree. The insert() function inserts a new node with the given value into the tree, while the search() function searches for a node with the given value in the tree. The inorder\_traversal() function performs an in-order traversal of the tree, printing the values of each node in ascending order.

Example 2.

\*/

#include <stdio.h>

#include <stdlib.h>

struct bt

{

int data;

struct bt \*left;

struct bt \*right;

};

struct bt \* insert(struct bt \*q, int val, char dir)

{

struct bt \*tmp;

tmp = (struct bt \*)malloc(sizeof(struct bt));

tmp->data=val;

tmp->left=tmp->right=NULL;

if(dir=='l')

q->left=tmp;

else

q->right = tmp;

}

void inorder(struct bt \*q)

{

/\* Left, print, right (LPR)\*/

if(q==NULL)

{

return;

}

inorder(q->left);

printf(" %d ", q->data);

inorder(q->right);

}

void preorder(struct bt \*q)

{

/\*Print, Left, Right (PLR)\*/

if(q!=NULL)

{

printf(" %d ", q->data);

preorder(q->left);

preorder(q->right);

}

}

void postorder(struct bt \*q)

{

/\*Left, Right, Print \*/

if(q!=NULL)

{

postorder(q->left);

postorder(q->right);

printf(" % d", q->data);

}

}

int main(int argc, int \*\*argv)

{

struct bt \*root=(struct bt \*)malloc(sizeof(struct bt));

root->data=1;

insert(root,3,'l');

insert(root,2,'r');

insert(root->left,5,'l');

insert(root->right,7,'l');

printf("\n display element in order: ");

inorder(root);

printf("\n display element in preorder: ");

preorder(root);

printf("\n display element o=in postorder: ");

postorder(root);

return 0;

}

This code defines a binary tree data structure using a structure named bt, which has three fields: an integer data field, and two pointers to bt structs, named left and right, which represent the left and right children of a node.

The code defines a function insert that takes a pointer to a bt struct q, an integer val, and a character dir. The function creates a new bt struct, sets its data field to val, and sets its left and right pointers to NULL. If dir is 'l', the new node is added as the left child of q; otherwise, it is added as the right child.

The code also defines three functions for tree traversal: inorder, preorder, and postorder. Each of these functions takes a pointer to a bt struct and recursively traverses the tree, printing out the node values in a different order.

The main function creates a binary tree with five nodes using the insert function, and then calls each of the traversal functions to print out the values of the nodes in the tree in different orders. Specifically, it prints out the values of the nodes in inorder, preorder, and postorder.

**we can call the inorder() function which performs the in-order traversal of the binary tree and prints the data of each node.**

Here's an example code

#include <stdio.h>

#include <stdlib.h>

struct bt

{

int data;

struct bt \*left;

struct bt \*right;

};

// function to insert a node in the binary tree

struct bt \* insert(struct bt \*q, int val, char dir)

{

struct bt \*tmp;

tmp = (struct bt \*)malloc(sizeof(struct bt));

tmp->data=val;

tmp->left=tmp->right=NULL;

if(dir=='l')

q->left=tmp;

else

q->right = tmp;

}

// function to print the data of the binary tree

void printTree(struct bt \*root) {

printf("\nDisplay element in order: ");

inorder(root);

}

// function to perform in-order traversal of the binary tree

void inorder(struct bt \*q)

{

/\* Left, print, right (LPR)\*/

if(q==NULL)

{

return;

}

inorder(q->left);

printf(" %d ", q->data);

inorder(q->right);

}

int main(int argc, int \*\*argv)

{

struct bt \*root=(struct bt \*)malloc(sizeof(struct bt));

root->data=1;

insert(root,3,'l');

insert(root,2,'r');

insert(root->left,5,'l');

insert(root->right,7,'l');

// print the data of the binary tree

printTree(root);

return 0;

}

Display element in order: 5 3 1 2 7