Binary Tree

A binary tree is a structure comparing nodes, where each node has the following 3 components:

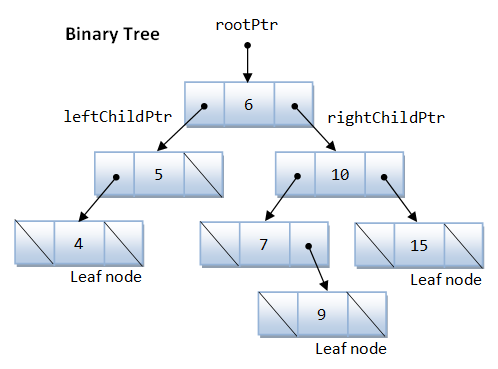
1. Data element: Stores any kind of data in the node
2. Left pointer: Points to the tree on the left side of node
3. Right pointer: Points to the tree on the right side of the node

As the name suggests, the **data** element stores any kind of data in the node.

The **left** and **right** pointers point to binary trees on the left and right side of the node respectively.

If a tree is empty, it is represented by a null pointer.

The following image explains the various components of a tree.



**Commonly-used terminologies**

* **Root**: Top node in a tree
* **Child**: Nodes that are next to each other and connected downwards
* **Parent**: Converse notion of child
* **Siblings**: Nodes with the same parent
* **Descendant**: Node reachable by repeated proceeding from parent to child
* **Ancestor**: Node reachable by repeated proceeding from child to parent.
* **Leaf**: Node with no children
* **Internal** **node**: Node with at least one child
* **External** **node**: Node with no children

**Structure code of a tree node**

In C programming, trees are declared as follows:

typedef struct treeStruct {

int data;

struct treeStruct \*left;

struct treeStruct \*right;

}NODE;

**Creating nodes**

Simple node

NODE newNode;

Pointer to a node

NODE \*newNode;

newNode = (NODE \*)malloc(sizeof(NODE));

In this case, you must explicitly allocate the memory of the node type to the pointer (preferred method).

Utility function returning node

NODE \*createNode(int data)

{

NODE \*newNode;

newNode = (NODE \*)malloc(sizeof(NODE));

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

**Maximum depth/height of a tree**

The idea is to do a post-order traversal and maintain two variables to store the left depth and right depth and return max of both the depths.

int maxDepth(NODE \*root)

{

if (NULL == root)

return 0;

else {

int leftDepth = maxDepth(root->left);

int rightDepth = maxDepth(root->right);

if (leftDepth > rightDepth)

return (leftDepth + 1);

else

return (rightDepth + 1);

}

}

**Time complexity**

O(n)

**Application of trees**

1. A manipulate hierarchical data
2. Make information easy to search (see tree traversal)
3. Manipulate sorted lists of data
4. Use as a workflow for compositing digital images for visual effects
5. Use in router algorithms

**Binary Search Tree**

Binary Search Tree, is a node-based binary tree data structure which has the following properties:

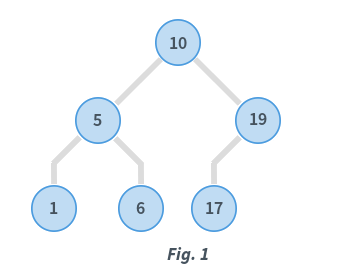
* The left subtree of a node contains only nodes with keys lesser than the node’s key.
* The right subtree of a node contains only nodes with keys greater than the node’s key.
* The left and right subtree each must also be a binary search tree.
* Three must be no duplicate nodes.

200px-Binary_search_tree.svg

The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search a given key.

For a binary tree to be a binary search tree, the data of all the nodes in the left sub-tree of the root node should be <= the data of the root. The data of all the nodes in the right subtree of the root node should be > the data of the root.

Example



In Fig. 1, consider the root node with data = 10.

* Data in the left subtree is: [5, 1, 6]. All data elements are < 10.
* Data in the right subtree is: [19, 17]. All data elements are > 10.

Also, considering the root node with **data = 5**, its children also satisfy the specified ordering. Similarly, the root node with **data = 19** also satisfies this ordering. When recursive, all subtrees satisfy the left and right subtree ordering.

The tree is known as a Binary Search Tree or BST.

**Traversing the tree**

There are mainly three types of tree traversals.

**Pre-order traversal**

In this traversal technique the traversal order is root-left-right i.e.

* Process data of root node
* First, traverse left subtree completely
* Then, traverse right subtree

void preorder(NODE \*root)

{

if (NULL == root)

return;

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

preorder(root->left);

preorder(root->right);

}

**Post-order traversal**

In this traversal technique the traversal order is left-right-root.

* Process data of left subtree
* First, traverse right subtree
* Then, traverse root node

void postorder(NODE \*root)

{

if (NULL == root)

return;

postorder(root->left);

postorder(root->right);

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

}

**In-order traversal**

In in-order traversal, do the following:

* First process left subtree (before processing root node)
* Then, process current root node
* Process right subtree

void inorder(NODE \*root)

{

if (NULL == root)

return;

inorder(root->left);

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

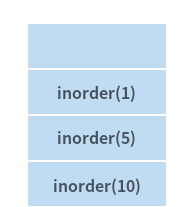
inorder(root->right);

}

Consider the in-order traversal of a sample BST

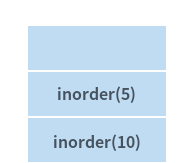
* The **“inorder()”** function is called with root equal to node with **data = 10**
* Since the node has a left subtree, **“inorder()”** is called with root equal to node with **data = 5**
* Again, the node has a left subtree, so **“inorder()”** is called with **data = 1**

The function call stack is as follows:



* Node with **data = 1** does not have a left subtree. Hence, this node is processed.
* Node with **data = 1** does not have a right subtree. Hence, nothing is done.
* **inorder(1)** gets completed and this function call is popped from the call stack.

The stack is as follows:

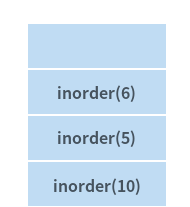


* Left subtree of node with **data = 5** is completely processed. Hence, this node gets processed.
* Right subtree of this node with **data = 5** is non-empty. Hence, the right subtree gets processed now. **‘inorder(6)’** is then called.

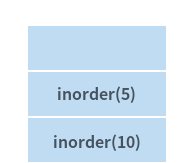
Note

**‘inorder(6)’** is only equivalent to saying inorder(pointer to node with **data = 6**). The notation has been used for brevity.

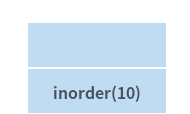
The function call stack is as follows:



Again, the node with **data = 6** has no left subtree. Therefore, it can be processed and it also has no right subtree. **‘inorder(6)’** is then completed.



Both the left and right subtree of node with **data = 5** have been completely processed. Hence, **inorder(5)** is then completed.



* Now node with **data = 10** is processed
* Right subtree of this node gets processed in a similar way as described until step 10
* After right subtree of this node is completely processed, entire traversal of the BST is complete

The order is which BST in Fig. 1 is visited is: 1, 5, 6, 10, 17, 19. The in-order traversal of a BST gives a sorted ordering of the data elements that are present in the BST. This is an important property of a BST.

**Insertion in BST**

A new key is always inserted at leaf. We start searching a key from root till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

100 100

/ \ Insert 40 / \

20 500 ---------> 20 500

/ \ / \

10 30 10 30

\

40

Consider the insertion of **data = 2** in the BST.

**Algorithm**

Parameter: Root of the BST and a data to be inserted.

Return: Root of the BST after insert the given data.

1. If the root is null:
   1. Create a new node using given data
   2. Assign the new node to root.
   3. Return root.
2. If the data of root node is greater and if a left subtree exists then repeat step 1 with root = root of the left subtree.
3. If the data of root node is smaller and if right subtree exits then repeat step 1 with root = root of the right subtree.
4. Finally return the root of the current subtree.

NODE \*insert(NODE \*root, int data)

{

if (NULL == root)

root = createNode(data);

else if (data < root->data)

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

else if (data > root->data)

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

return root;

}

**Illustration to insert 2 in below tree:**

1. Start from root.
2. Compare the inserting element with root, if less than root, then recurse for left, else recurse for right.
3. After reaching end, just insert that node at left(if less than current) else right.

bstsearch

Time Complexity: The worst case time complexity of search and insert operation is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of search and insert operation may become O(n).

Some interesting Facts:

* Inorder traversal of BST always produces sorted output.
* We can construct a BST with only Preorder or Postorder or Lever Order traversal. Note that we can always get inorder traversal by sorting the only given traversal.
* Number of unique BST’s with n distinct keys is Catalan Number.

**Search in BST**

To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root’s key, we recur for right subtree of root node. Otherwise we recur for left subtree.

int search(NODE \*root, int data)

{

if (NULL == root)

return 0;

if (data < root->data)

return search(root->left, data);

if (data > root->data)

return search(root->right, data);

return 1;

}

Illustration to search 6 in below tree:

1. Start from root.
2. Compare the inserting element with root, if less than root, then recurse for left, else recurse for right.
3. If element to search is found anywhere, return true, else return false.

bstsearch

**Delete in BST**

When we delete a node, three possibilities arise.

1. **Node to be deleted is leaf:**

Simply remove from the tree.

50 50

/ \ delete(20) / \

30 70 ---------> 30 70

/ \ / \ \ / \

20 40 60 80 40 60 80

1. **Node to be deleted has only one child:**
2. Copy the child to the node and delete the child or
3. Delete the node and assign the child node at position of deleted node.

50 50

/ \ delete(30) / \

30 70 ---------> 40 70

\ / \ / \

40 60 80 60 80

1. Node to be deleted has two children: Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

50 60

/ \ delete(50) / \

40 70 ---------> 40 70

/ \ \

60 80 80

The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

bst-delete

NODE \*deleteNode(NODE \*root, int data)

{

if (NULL == root)

return root;

if (data < root->data)

root->left = deleteNode(root->left, data);

else if (data > root->data)

root->right = deleteNode(root->right, data);

else {

if (NULL == root->left && NULL == root->right) {

free(root);

return NULL;

}

else if (NULL == root->left) {

NODE \*temp = root->right;

free(root);

return temp;

}

else if (NULL == root->right) {

NODE \*temp = root->left;

free(root);

return temp;

}

else {

NODE \*leftMost = root->right;

while (leftMost->left != NULL)

leftMost = leftMost->left;

root->data = leftMost->data;

root->right = deleteNode(root->right, root->data);

return root;

}

}

return root;

}