TREE DATA-STRUCTURE IN C

A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges. Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

Tree Terminologies

Node: A node is an entity that contains a key or value and pointers to its child nodes. The last nodes of each path are called **leaf nodes or external nodes** that do not contain a link/pointer to child nodes. The node having at least a child node is called an **internal node**.

Edge: It is the link between any two nodes.

Root: It is the topmost node of a tree.

Height of a Tree: The height of a Tree is the height of the root node or the depth of the deepest node.

Height of a Node: The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

Depth of a Node: The depth of a node is the number of edges from the root to the node.

Degree of a Node: The degree of a node is the total number of branches of that node.

Forest: A collection of disjoint trees is called a forest.

Types of Tree

1. [Binary Tree](https://www.programiz.com/dsa/binary-tree)
2. [Binary Search Tree](https://www.programiz.com/dsa/binary-search-tree)
3. [AVL Tree](https://www.programiz.com/dsa/avl-tree)
4. [B-Tree](https://www.programiz.com/dsa/b-tree)

Binary Tree: A binary tree is a tree data structure in which each parent node can have at most two children. Each node of a binary tree consists of three items:

* data item
* address of left child
* address of right child

## Types of Binary Tree

### 1. Full Binary Tree

A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.

### 2. Perfect Binary Tree

A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.

### 3. Complete Binary Tree

A complete binary tree is just like a full binary tree, but with two major differences

1. Every level must be completely filled
2. All the leaf elements must lean towards the left.
3. The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.

### 4. Degenerate or Pathological Tree

A degenerate or pathological tree is the tree having a single child either left or right.

### 5. Skewed Binary Tree

A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: **left-skewed binary tree** and **right-skewed binary tree**.

### 6. Balanced Binary Tree

It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1.

## Binary Tree Representation

struct node

{

int data;

struct node \*left;

struct node \*right;

};

// Tree traversal in C

#include <stdio.h>

#include <stdlib.h>

struct node {

int item;

struct node\* left;

struct node\* right;

};

// Inorder traversal

void inorderTraversal(struct node\* root) {

if (root == NULL) return;

inorderTraversal(root->left);

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

inorderTraversal(root->right);

}

// Preorder traversal

void preorderTraversal(struct node\* root) {

if (root == NULL) return;

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

preorderTraversal(root->left);

preorderTraversal(root->right);

}

// Postorder traversal

void postorderTraversal(struct node\* root) {

if (root == NULL) return;

postorderTraversal(root->left);

postorderTraversal(root->right);

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

}

// Create a new Node

struct node\* createNode(value) {

struct node\* newNode = malloc(sizeof(struct node));

newNode->item = value;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Insert on the left of the node

struct node\* insertLeft(struct node\* root, int value) {

root->left = createNode(value);

return root->left;

}

// Insert on the right of the node

struct node\* insertRight(struct node\* root, int value) {

root->right = createNode(value);

return root->right;

}

int main() {

struct node\* root = createNode(1);

insertLeft(root, 2);

insertRight(root, 3);

insertLeft(root->left, 4);

printf("Inorder traversal \n");

inorderTraversal(root);

printf("\nPreorder traversal \n");

preorderTraversal(root);

printf("\nPostorder traversal \n");

postorderTraversal(root);

}

## Binary Tree Applications

* For easy and quick access to data
* In router algorithms
* To implement [heap data structure](https://www.programiz.com/dsa/heap-data-structure)
* Syntax tree

# Binary Search Tree(BST)

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

* It is called a binary tree because each tree node has a maximum of two children.
* It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

The properties that separate a binary search tree from a regular [binary tree](https://www.programiz.com/data-structures/trees) is

1. All nodes of left subtree are less than the root node
2. All nodes of right subtree are more than the root node
3. Both subtrees of each node are also BSTs

## Search Operation

If root == NULL

return NULL;

If number == root->data

return root->data;

If number < root->data

return search(root->left)

If number > root->data

return search(root->right)

## Insert Operation

If node == NULL

return createNode(data)

if (data < node->data)

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

else if (data > node->data)

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

return node;

## Deletion Operation

There are three cases for deleting a node from a binary search tree.

### Case I

In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.

**Case II**

In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:

1. Replace that node with its child node.
2. Remove the child node from its original position.

**Case III**

In the third case, the node to be deleted has two children. In such a case follow the steps below:

1. Get the inorder successor of that node.
2. Replace the node with the inorder successor.
3. Remove the inorder successor from its original position.
4. // Binary Search Tree operations in C
5. #include <stdio.h>
6. #include <stdlib.h>
7. struct node {
8. int key;
9. struct node \*left, \*right;
10. };
11. // Create a node
12. struct node \*newNode(int item) {
13. struct node \*temp = (struct node \*)malloc(sizeof(struct node));
14. temp->key = item;
15. temp->left = temp->right = NULL;
16. return temp;
17. }
18. // Inorder Traversal
19. void inorder(struct node \*root) {
20. if (root != NULL) {
21. // Traverse left
22. inorder(root->left);
23. // Traverse root
24. printf("%d -> ", root->key);
25. // Traverse right
26. inorder(root->right);
27. }
28. }
29. // Insert a node
30. struct node \*insert(struct node \*node, int key) {
31. // Return a new node if the tree is empty
32. if (node == NULL) return newNode(key);
33. // Traverse to the right place and insert the node
34. if (key < node->key)
35. node->left = insert(node->left, key);
36. else
37. node->right = insert(node->right, key);
38. return node;
39. }
40. // Find the inorder successor
41. struct node \*minValueNode(struct node \*node) {
42. struct node \*current = node;
43. // Find the leftmost leaf
44. while (current && current->left != NULL)
45. current = current->left;
46. return current;
47. }
48. // Deleting a node
49. struct node \*deleteNode(struct node \*root, int key) {
50. // Return if the tree is empty
51. if (root == NULL) return root;
52. // Find the node to be deleted
53. if (key < root->key)
54. root->left = deleteNode(root->left, key);
55. else if (key > root->key)
56. root->right = deleteNode(root->right, key);
57. else {
58. // If the node is with only one child or no child
59. if (root->left == NULL) {
60. struct node \*temp = root->right;
61. free(root);
62. return temp;
63. } else if (root->right == NULL) {
64. struct node \*temp = root->left;
65. free(root);
66. return temp;
67. }
68. // If the node has two children
69. struct node \*temp = minValueNode(root->right);
70. // Place the inorder successor in position of the node to be deleted
71. root->key = temp->key;
72. // Delete the inorder successor
73. root->right = deleteNode(root->right, temp->key);
74. }
75. return root;
76. }
77. // Driver code
78. int main() {
79. struct node \*root = NULL;
80. root = insert(root, 8);
81. root = insert(root, 3);
82. root = insert(root, 1);
83. root = insert(root, 6);
84. root = insert(root, 7);
85. root = insert(root, 10);
86. root = insert(root, 14);
87. root = insert(root, 4);
88. printf("Inorder traversal: ");
89. inorder(root);
90. printf("\nAfter deleting 10\n");
91. root = deleteNode(root, 10);
92. printf("Inorder traversal: ");
93. inorder(root);

## } Binary Search Tree Applications

1. In multilevel indexing in the database
2. For dynamic sorting
3. For managing virtual memory areas in Unix kernel

# AVL Tree

AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.

## Balance Factor

Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

Balance Factor = (Height of Left Subtree - Height of Right Subtree) or (Height of Right Subtree - Height of Left Subtree)

The self balancing property of an avl tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1.

## Operations on an AVL tree

## Rotating the subtrees in an AVL Tree

### Left Rotate

### Right Rotate

### Left-Right and Right-Left Rotate

1. #include <stdio.h>
2. #include <stdlib.h>
3. // Create Node
4. struct Node {
5. int key;
6. struct Node \*left;
7. struct Node \*right;
8. int height;
9. };
10. int max(int a, int b);
11. // Calculate height
12. int height(struct Node \*N) {
13. if (N == NULL)
14. return 0;
15. return N->height;
16. }
17. int max(int a, int b) {
18. return (a > b) ? a : b;
19. }
20. // Create a node
21. struct Node \*newNode(int key) {
22. struct Node \*node = (struct Node \*)
23. malloc(sizeof(struct Node));
24. node->key = key;
25. node->left = NULL;
26. node->right = NULL;
27. node->height = 1;
28. return (node);
29. }
30. // Right rotate
31. struct Node \*rightRotate(struct Node \*y) {
32. struct Node \*x = y->left;
33. struct Node \*T2 = x->right;
34. x->right = y;
35. y->left = T2;
36. y->height = max(height(y->left), height(y->right)) + 1;
37. x->height = max(height(x->left), height(x->right)) + 1;
38. return x;
39. }
40. // Left rotate
41. struct Node \*leftRotate(struct Node \*x) {
42. struct Node \*y = x->right;
43. struct Node \*T2 = y->left;
44. y->left = x;
45. x->right = T2;
46. x->height = max(height(x->left), height(x->right)) + 1;
47. y->height = max(height(y->left), height(y->right)) + 1;
48. return y;
49. }
50. // Get the balance factor
51. int getBalance(struct Node \*N) {
52. if (N == NULL)
53. return 0;
54. return height(N->left) - height(N->right);
55. }
56. // Insert node
57. struct Node \*insertNode(struct Node \*node, int key) {
58. // Find the correct position to insertNode the node and insertNode it
59. if (node == NULL)
60. return (newNode(key));
61. if (key < node->key)
62. node->left = insertNode(node->left, key);
63. else if (key > node->key)
64. node->right = insertNode(node->right, key);
65. else
66. return node;
67. // Update the balance factor of each node and
68. // Balance the tree
69. node->height = 1 + max(height(node->left),
70. height(node->right));
71. int balance = getBalance(node);
72. if (balance > 1 && key < node->left->key)
73. return rightRotate(node);
74. if (balance < -1 && key > node->right->key)
75. return leftRotate(node);
76. if (balance > 1 && key > node->left->key) {
77. node->left = leftRotate(node->left);
78. return rightRotate(node);
79. }
80. if (balance < -1 && key < node->right->key) {
81. node->right = rightRotate(node->right);
82. return leftRotate(node);
83. }
84. return node;
85. }
86. struct Node \*minValueNode(struct Node \*node) {
87. struct Node \*current = node;
88. while (current->left != NULL)
89. current = current->left;
90. return current;
91. }
92. // Delete a nodes
93. struct Node \*deleteNode(struct Node \*root, int key) {
94. // Find the node and delete it
95. if (root == NULL)
96. return root;
97. if (key < root->key)
98. root->left = deleteNode(root->left, key);
99. else if (key > root->key)
100. root->right = deleteNode(root->right, key);
101. else {
102. if ((root->left == NULL) || (root->right == NULL)) {
103. struct Node \*temp = root->left ? root->left : root->right;
104. if (temp == NULL) {
105. temp = root;
106. root = NULL;
107. } else
108. \*root = \*temp;
109. free(temp);
110. } else {
111. struct Node \*temp = minValueNode(root->right);
112. root->key = temp->key;
113. root->right = deleteNode(root->right, temp->key);
114. }
115. }
116. if (root == NULL)
117. return root;
118. // Update the balance factor of each node and
119. // balance the tree
120. root->height = 1 + max(height(root->left),
121. height(root->right));
122. int balance = getBalance(root);
123. if (balance > 1 && getBalance(root->left) >= 0)
124. return rightRotate(root);
125. if (balance > 1 && getBalance(root->left) < 0) {
126. root->left = leftRotate(root->left);
127. return rightRotate(root);
128. }
129. if (balance < -1 && getBalance(root->right) <= 0)
130. return leftRotate(root);
131. if (balance < -1 && getBalance(root->right) > 0) {
132. root->right = rightRotate(root->right);
133. return leftRotate(root);
134. }
135. return root;
136. }
137. // Print the tree
138. void printPreOrder(struct Node \*root) {
139. if (root != NULL) {
140. printf("%d ", root->key);
141. printPreOrder(root->left);
142. printPreOrder(root->right);
143. }
144. }
145. int main() {
146. struct Node \*root = NULL;
147. root = insertNode(root, 2);
148. root = insertNode(root, 1);
149. root = insertNode(root, 7);
150. root = insertNode(root, 4);
151. root = insertNode(root, 5);
152. root = insertNode(root, 3);
153. root = insertNode(root, 8);
154. printPreOrder(root);
155. root = deleteNode(root, 3);
156. printf("\nAfter deletion: ");
157. printPreOrder(root);
158. return 0;
159. }

## AVL Tree Applications

* For indexing large records in databases
* For searching in large databases

# B-tree

B-tree is a special type of self-balancing search tree in which each node can contain more than one key and can have more than two children. It is a generalized form of the [binary search tree](https://www.programiz.com/dsa/binary-search-tree).

It is also known as a height-balanced m-way tree.

## B-tree Properties

1. For each node x, the keys are stored in increasing order.
2. In each node, there is a boolean value x.leaf which is true if x is a leaf.
3. If n is the order of the tree, each internal node can contain at most n - 1 keys along with a pointer to each child.
4. Each node except root can have at most n children and at least n/2 children.
5. All leaves have the same depth (i.e. height-h of the tree).
6. The root has at least 2 children and contains a minimum of 1 key.
7. If n ≥ 1, then for any n-key B-tree of height h and minimum degree t ≥ 2, h ≥ logt (n+1)/2.
8. #include <stdio.h>
9. #include <stdlib.h>
10. #define MAX 3
11. #define MIN 2
12. struct BTreeNode {
13. int val[MAX + 1], count;
14. struct BTreeNode \*link[MAX + 1];
15. };
16. struct BTreeNode \*root;
17. // Create a node
18. struct BTreeNode \*createNode(int val, struct BTreeNode \*child) {
19. struct BTreeNode \*newNode;
20. newNode = (struct BTreeNode \*)malloc(sizeof(struct BTreeNode));
21. newNode->val[1] = val;
22. newNode->count = 1;
23. newNode->link[0] = root;
24. newNode->link[1] = child;
25. return newNode;
26. }
27. // Insert node
28. void insertNode(int val, int pos, struct BTreeNode \*node,
29. struct BTreeNode \*child) {
30. int j = node->count;
31. while (j > pos) {
32. node->val[j + 1] = node->val[j];
33. node->link[j + 1] = node->link[j];
34. j--;
35. }
36. node->val[j + 1] = val;
37. node->link[j + 1] = child;
38. node->count++;
39. }
40. // Split node
41. void splitNode(int val, int \*pval, int pos, struct BTreeNode \*node,
42. struct BTreeNode \*child, struct BTreeNode \*\*newNode) {
43. int median, j;
44. if (pos > MIN)
45. median = MIN + 1;
46. else
47. median = MIN;
48. \*newNode = (struct BTreeNode \*)malloc(sizeof(struct BTreeNode));
49. j = median + 1;
50. while (j <= MAX) {
51. (\*newNode)->val[j - median] = node->val[j];
52. (\*newNode)->link[j - median] = node->link[j];
53. j++;
54. }
55. node->count = median;
56. (\*newNode)->count = MAX - median;
57. if (pos <= MIN) {
58. insertNode(val, pos, node, child);
59. } else {
60. insertNode(val, pos - median, \*newNode, child);
61. }
62. \*pval = node->val[node->count];
63. (\*newNode)->link[0] = node->link[node->count];
64. node->count--;
65. }
66. // Set the value
67. int setValue(int val, int \*pval,
68. struct BTreeNode \*node, struct BTreeNode \*\*child) {
69. int pos;
70. if (!node) {
71. \*pval = val;
72. \*child = NULL;
73. return 1;
74. }
75. if (val < node->val[1]) {
76. pos = 0;
77. } else {
78. for (pos = node->count;
79. (val < node->val[pos] && pos > 1); pos--)
80. ;
81. if (val == node->val[pos]) {
82. printf("Duplicates are not permitted\n");
83. return 0;
84. }
85. }
86. if (setValue(val, pval, node->link[pos], child)) {
87. if (node->count < MAX) {
88. insertNode(\*pval, pos, node, \*child);
89. } else {
90. splitNode(\*pval, pval, pos, node, \*child, child);
91. return 1;
92. }
93. }
94. return 0;
95. }
96. // Insert the value
97. void insert(int val) {
98. int flag, i;
99. struct BTreeNode \*child;
100. flag = setValue(val, &i, root, &child);
101. if (flag)
102. root = createNode(i, child);
103. }
104. // Search node
105. void search(int val, int \*pos, struct BTreeNode \*myNode) {
106. if (!myNode) {
107. return;
108. }
109. if (val < myNode->val[1]) {
110. \*pos = 0;
111. } else {
112. for (\*pos = myNode->count;
113. (val < myNode->val[\*pos] && \*pos > 1); (\*pos)--)
114. ;
115. if (val == myNode->val[\*pos]) {
116. printf("%d is found", val);
117. return;
118. }
119. }
120. search(val, pos, myNode->link[\*pos]);
121. return;
122. }
123. // Traverse then nodes
124. void traversal(struct BTreeNode \*myNode) {
125. int i;
126. if (myNode) {
127. for (i = 0; i < myNode->count; i++) {
128. traversal(myNode->link[i]);
129. printf("%d ", myNode->val[i + 1]);
130. }
131. traversal(myNode->link[i]);
132. }
133. }
134. int main() {
135. int val, ch;
136. insert(8);
137. insert(9);
138. insert(10);
139. insert(11);
140. insert(15);
141. insert(16);
142. insert(17);
143. insert(18);
144. insert(20);
145. insert(23);
146. traversal(root);
147. printf("\n");
148. search(11, &ch, root);
149. }

## B Tree Applications

* databases and file systems
* to store blocks of data (secondary storage media)
* multilevel indexing

## Tree Traversal

In order to perform any operation on a tree, you need to reach to the specific node. The tree traversal algorithm helps in visiting a required node in the tree.

## Tree Applications

* Binary Search Trees(BSTs) are used to quickly check whether an element is present in a set or not.
* Heap is a kind of tree that is used for heap sort.
* A modified version of a tree called Tries is used in modern routers to store routing information.
* Most popular databases use B-Trees and T-Trees, which are variants of the tree structure we learned above to store their data
* Compilers use a syntax tree to validate the syntax of every program you write.