**Binary Tree Operations**

# Insertion and Deletion in a Binary Tree

Given a binary tree and a key, insert the key into the binary tree at first position available in level order.

#include <iostream>

#include <queue>

using namespace std;

struct tree\_Node {

int value;

struct tree\_Node \* left;

struct tree\_Node \* right;

};

struct tree\_Node \* newNode(int arg\_val);

void insert(struct tree\_Node \* arg\_root, struct tree\_Node \* arg\_node);

void preorder\_traverse(struct tree\_Node \* arg\_root);

void delete\_node(struct tree\_Node \* arg\_root, int arg\_key);

void delete\_rightmost(struct tree\_Node \* arg\_root, struct tree\_Node \* rightmost); // to be used by delete\_node

int main(void) {

int n = 0;

int key = 0;

int delete\_key = 0;

struct tree\_Node \* node = NULL;

cout << "How many nodes you want to insert: ";

cin >> n;

cout << "Enter key: ";

cin >> key;

struct tree\_Node \* root = newNode(key);

cout << "Preorder Traversal: ";

preorder\_traverse(root); cout << endl;

while(0 != (n-1)) {

cout << "Enter key: ";

cin >> key;

node = NULL;

node = newNode(key);

if((NULL != root) && (NULL != node)) {

insert(root, node);

}

cout << "Preorder Traversal: ";

preorder\_traverse(root); cout << endl;

n = n - 1;

}

cout << "Enter key to delete: ";

cin >> delete\_key;

delete\_node(root, delete\_key);

cout << "Preorder Traversal: ";

preorder\_traverse(root); cout << endl;

return 0;

}

struct tree\_Node \* newNode(int arg\_val) {

struct tree\_Node \* node = new tree\_Node;

node -> value = arg\_val;

node -> left = NULL;

node -> right = NULL;

return node;

}

void preorder\_traverse(struct tree\_Node \* arg\_root) {

if(NULL == arg\_root) {

return;

}

cout << (arg\_root -> value) << " ";

preorder\_traverse(arg\_root -> left);

preorder\_traverse(arg\_root -> right);

}

void insert(struct tree\_Node \* arg\_root, struct tree\_Node \* arg\_node) {

queue<struct tree\_Node \*> qt;

qt.push(arg\_root);

while(!qt.empty()) {

struct tree\_Node \* temp = qt.front();

qt.pop();

if(NULL == (temp -> left)) {

temp -> left = arg\_node;

break;

}

else {

qt.push(temp -> left);

}

if(NULL == (temp -> right)) {

temp -> right = arg\_node;

break;

}

else {

qt.push(temp -> right);

}

} // end of while

return;

}

// this function identifies the nodes with key and rightmost node

void delete\_node(struct tree\_Node \* arg\_root, int arg\_key) {

struct tree\_Node \* temp = NULL;

struct tree\_Node \* key\_node = NULL;

struct tree\_Node \* rightmost\_node = NULL;

queue<struct tree\_Node \*> qt;

if(NULL == arg\_root) {

return;

}

qt.push(arg\_root);

while(!qt.empty()) {

temp = qt.front();

qt.pop();

if(arg\_key == (temp -> value)) {

key\_node = temp;

}

if(NULL != (temp -> left)) {

qt.push(temp -> left);

}

if(NULL != (temp -> right)) {

qt.push(temp -> right);

}

} // end of while

rightmost\_node = temp;

int x = rightmost\_node -> value; // store value of rightmost node

delete\_rightmost(arg\_root, rightmost\_node); // to delete right node

key\_node -> value = x; // replace value of key node with value of rightmost node

return;

}

void delete\_rightmost(struct tree\_Node \* arg\_root, struct tree\_Node \* rightmost) {

struct tree\_Node \* temp = NULL;

queue<struct tree\_Node \*>qt;

if(NULL == arg\_root) {

return;

}

qt.push(arg\_root);

while(!qt.empty()) {

temp = qt.front();

qt.pop();

if(NULL != (temp -> right)) {

if(rightmost == (temp -> right)) {

temp -> right = NULL;

delete(rightmost);

break;

}

else {

qt.push(temp -> right);

}

}

if(NULL != (temp -> left)) {

if(rightmost == (temp -> left)) {

temp -> left = NULL;

delete(rightmost);

break;

}

else {

qt.push(temp -> left);

}

}

} // end of while

return;

}

Output:

How many nodes you want to insert: 7

Enter key: 1

Preorder Traversal: 1

Enter key: 2

Preorder Traversal: 1 2

Enter key: 3

Preorder Traversal: 1 2 3

Enter key: 4

Preorder Traversal: 1 2 4 3

Enter key: 5

Preorder Traversal: 1 2 4 5 3

Enter key: 6

Preorder Traversal: 1 2 4 5 3 6

Enter key: 7

Preorder Traversal: 1 2 4 5 3 6 7

Enter key to delete: 2

Preorder Traversal: 1 7 4 5 3 6

# Binary Tree (Array implementation)

If a node has an index i, its children are found at indices

|  |
| --- |
| **Left child 2i+1** |
| **Right child 2i+2** |

while its parent (if any) is found at index

|  |
| --- |
| **parent index = (i - 1) / 2 (assuming the root has index zero)** |

#include <stdio.h>

#include <stdlib.h>

#define SUCCESS 0

#define FAILURE -1

#define TREE\_ELEMENTS 16

int tree\_arr[TREE\_ELEMENTS];

int set\_root(const int root, const int key);

int set\_left(const int root, const int key);

int set\_right(const int root, const int key);

int print\_BFS(void);

int main(void) {

int index = 0;

int root = 0;

for(index = 0; index < TREE\_ELEMENTS; index++) {

tree\_arr[index] = -1;

}

set\_root(root, 1);

set\_left(0, 2);

set\_right(0, 3);

set\_left(1, 4);

set\_right(1, 5);

set\_left(2, 6);

set\_right(2, 7);

print\_BFS();

return 0;

}

int set\_root(const int root, const int key) {

tree\_arr[root] = key;

return SUCCESS;

}

int set\_left(const int root, const int key) {

int index = (root \* 2) + 1;

if(-1 == tree\_arr[root])

return FAILURE;

else

tree\_arr[index] = key;

return SUCCESS;

}

int set\_right(const int root, const int key) {

int index = (root \* 2) + 2;

if(-1 == tree\_arr[root])

return FAILURE;

else

tree\_arr[index] = key;

return SUCCESS;

}

int print\_BFS(void) {

int index = 0;

for(index = 0; index < TREE\_ELEMENTS; index++) {

if(-1 != tree\_arr[index])

printf("%d", tree\_arr[index]);

else

printf("%c", '-');

}

return SUCCESS;

}

Output:

1234567---------

# Print leaf nodes of a Binary tree

#include <iostream>

#include <queue>

using namespace std;

struct tree\_Node {

int value;

struct tree\_Node \* left;

struct tree\_Node \* right;

};

struct tree\_Node \* newNode(int arg\_val);

void insert(struct tree\_Node \* arg\_root, struct tree\_Node \* arg\_node);

void preorder\_traverse(struct tree\_Node \* arg\_root);

void print\_leafNodes(struct tree\_Node \* arg\_root);

int main(void) {

int n = 0;

int key = 0;

struct tree\_Node \* node = NULL;

cout << "How many nodes you want to insert: ";

cin >> n;

cout << "Enter key: ";

cin >> key;

struct tree\_Node \* root = newNode(key);

cout << "Preorder Traversal: ";

preorder\_traverse(root); cout << endl;

while(0 != (n-1)) {

cout << "Enter key: ";

cin >> key;

node = NULL;

node = newNode(key);

if((NULL != root) && (NULL != node)) {

insert(root, node);

}

cout << "Preorder Traversal: ";

preorder\_traverse(root); cout << endl;

n = n - 1;

}

cout << "Preorder Traversal: ";

preorder\_traverse(root); cout << endl;

cout << endl;

cout << "Leaf nodes: ";

print\_leafNodes(root); cout << endl;

return 0;

}

struct tree\_Node \* newNode(int arg\_val) {

struct tree\_Node \* node = new tree\_Node;

node -> value = arg\_val;

node -> left = NULL;

node -> right = NULL;

return node;

}

void preorder\_traverse(struct tree\_Node \* arg\_root) {

if(NULL == arg\_root) {

return;

}

cout << (arg\_root -> value) << " ";

preorder\_traverse(arg\_root -> left);

preorder\_traverse(arg\_root -> right);

}

void print\_leafNodes(struct tree\_Node \* arg\_root) {

queue<struct tree\_Node \*>qt;

struct tree\_Node \* temp = NULL;

int noLeft = 0;

int noRight = 0;

if(NULL == arg\_root) {

return;

}

qt.push(arg\_root);

while(!qt.empty()) {

temp = qt.front();

qt.pop();

if(NULL == (temp -> left)) {

noLeft = 1;

}

else {

qt.push(temp -> left);

}

if(NULL == (temp -> right)) {

noRight = 1;

}

else {

qt.push(temp -> right);

}

if(noLeft && noRight) {

cout << (temp ->value) << " ";

}

noLeft = 0;

noRight = 0;

} // end of while

return;

}

void insert(struct tree\_Node \* arg\_root, struct tree\_Node \* arg\_node) {

queue<struct tree\_Node \*> qt;

qt.push(arg\_root);

while(!qt.empty()) {

struct tree\_Node \* temp = qt.front();

qt.pop();

if(NULL == (temp -> left)) {

temp -> left = arg\_node;

break;

}

else {

qt.push(temp -> left);

}

if(NULL == (temp -> right)) {

temp -> right = arg\_node;

break;

}

else {

qt.push(temp -> right);

}

} // end of while

return;

}

Output:

How many nodes you want to insert: 7

Enter key: 1

Preorder Traversal: 1

Enter key: 2

Preorder Traversal: 1 2

Enter key: 3

Preorder Traversal: 1 2 3

Enter key: 4

Preorder Traversal: 1 2 4 3

Enter key: 5

Preorder Traversal: 1 2 4 5 3

Enter key: 6

Preorder Traversal: 1 2 4 5 3 6

Enter key: 7

Preorder Traversal: 1 2 4 5 3 6 7

Preorder Traversal: 1 2 4 5 3 6 7

Leaf nodes: 4 5 6 7

# Symmetric Tree (Mirror Image of itself)

Given a binary tree, check whether it is a mirror of itself.



#include <iostream>

#include <queue>

using namespace std;

struct tree\_Node {

int value;

struct tree\_Node \* left;

struct tree\_Node \* right;

};

struct tree\_Node \* newNode(int arg\_val);

void insert(struct tree\_Node \* arg\_root, struct tree\_Node \* arg\_node);

void preorder\_traverse(struct tree\_Node \* arg\_root);

int is\_symmetric(struct tree\_Node \* arg\_root);

int check\_symmetric(struct tree\_Node \* arg\_root1, struct tree\_Node \* arg\_root2);

int main(void) {

int n = 0;

int key = 0;

struct tree\_Node \* node = NULL;

int is\_symtr = 0;

cout << "How many nodes you want to insert: ";

cin >> n;

cout << "Enter key: ";

cin >> key;

struct tree\_Node \* root = newNode(key);

cout << "Preorder Traversal: ";

preorder\_traverse(root); cout << endl;

while(0 != (n-1)) {

cout << "Enter key: ";

cin >> key;

node = NULL;

node = newNode(key);

if((NULL != root) && (NULL != node)) {

insert(root, node);

}

cout << "Preorder Traversal: ";

preorder\_traverse(root); cout << endl;

n = n - 1;

}

cout << "Preorder Traversal: ";

preorder\_traverse(root); cout << endl;

cout << endl;

is\_symtr = is\_symmetric(root);

if(1 == is\_symtr) {

cout << "Tree is symmetric" << endl;

}

else {

cout << "Tree is not symmetric" << endl;

}

return 0;

}

struct tree\_Node \* newNode(int arg\_val) {

struct tree\_Node \* node = new tree\_Node;

node -> value = arg\_val;

node -> left = NULL;

node -> right = NULL;

return node;

}

void preorder\_traverse(struct tree\_Node \* arg\_root) {

if(NULL == arg\_root) {

return;

}

cout << (arg\_root -> value) << " ";

preorder\_traverse(arg\_root -> left);

preorder\_traverse(arg\_root -> right);

}

int is\_symmetric(struct tree\_Node \* arg\_root) {

return check\_symmetric(arg\_root, arg\_root);

}

int check\_symmetric(struct tree\_Node \* arg\_root1, struct tree\_Node \* arg\_root2) {

if((NULL == arg\_root1) && (NULL == arg\_root2)) {

return 1;

}

if((arg\_root1 -> value) == (arg\_root2 -> value)) {

return (check\_symmetric((arg\_root1 -> left), (arg\_root2 -> right)) && check\_symmetric((arg\_root1 -> right), (arg\_root2 -> left)));

}

return 0;

}

void insert(struct tree\_Node \* arg\_root, struct tree\_Node \* arg\_node) {

queue<struct tree\_Node \*> qt;

qt.push(arg\_root);

while(!qt.empty()) {

struct tree\_Node \* temp = qt.front();

qt.pop();

if(NULL == (temp -> left)) {

temp -> left = arg\_node;

break;

}

else {

qt.push(temp -> left);

}

if(NULL == (temp -> right)) {

temp -> right = arg\_node;

break;

}

else {

qt.push(temp -> right);

}

} // end of while

return;

}

Output 1:

How many nodes you want to insert: 7

Enter key: 1

Preorder Traversal: 1

Enter key: 2

Preorder Traversal: 1 2

Enter key: 2

Preorder Traversal: 1 2 2

Enter key: 3

Preorder Traversal: 1 2 3 2

Enter key: 4

Preorder Traversal: 1 2 3 4 2

Enter key: 4

Preorder Traversal: 1 2 3 4 2 4

Enter key: 3

Preorder Traversal: 1 2 3 4 2 4 3

Preorder Traversal: 1 2 3 4 2 4 3

Tree is symmetric

Output 2:

How many nodes you want to insert: 7

Enter key: 1

Preorder Traversal: 1

Enter key: 2

Preorder Traversal: 1 2

Enter key: 2

Preorder Traversal: 1 2 2

Enter key: 3

Preorder Traversal: 1 2 3 2

Enter key: 4

Preorder Traversal: 1 2 3 4 2

Enter key: 5

Preorder Traversal: 1 2 3 4 2 5

Enter key: 3

Preorder Traversal: 1 2 3 4 2 5 3

Preorder Traversal: 1 2 3 4 2 5 3

Tree is not symmetric

# Level order traversal line by line

Given a Binary Tree, print the nodes level wise, each level on a new line.

Following methods can be use:

1. Use function to print each level
2. Use Queue
3. Use two queue
4. Use queue with delimiter NULL

## Use function to print each level

Modify method Use function to print a given level O(n^2) to print each level at different line by inserting a new line

void levelOrderTraverse(struct tree\_Node \* arg\_root) {

int tHeight = 0;

if(NULL == arg\_root) {

return;

}

tHeight = tree\_height(arg\_root);

for(int i = 1; i <= tHeight; i++) {

printGivenLevel(arg\_root, i);

**cout << endl; // added to print each level at new line**

}

return;

}

void printGivenLevel(struct tree\_Node \* arg\_node, int arg\_level) {

if(NULL == arg\_node) {

return;

}

if(1 == arg\_level) {

cout << (arg\_node -> value) << " ";

}

else if(1 < arg\_level){

printGivenLevel((arg\_node -> left), arg\_level - 1);

printGivenLevel((arg\_node -> right), arg\_level - 1);

}

else {

// do nothing

}

return;

}

Output:

Level Order Traversal:

1

2 3

4 5 6 7

**Time complexity:** O(n2)

## Use Queue

Modify method Use Queue O(n) to print each level at new line with the help of no of elements present in queue

void levelOrderTraverse(struct tree\_Node \* arg\_root) {

queue<struct tree\_Node \*>qt;

struct tree\_Node \* temp = NULL;

int count = 0;

if(NULL == arg\_root) {

return;

}

qt.push(arg\_root);

while(1) {

count = qt.size();

if(0 == count) {

break;

}

while(0 != count) {

temp = qt.front();

qt.pop();

cout << (temp -> value) << " ";

if(NULL != (temp -> left)) {

qt.push(temp -> left);

}

if(NULL != (temp -> right)) {

qt.push(temp -> right);

}

count = count - 1;

} // end of while(0 != count)

cout << endl;

} // end of while(1)

return;

}

Level Order Traversal:

1

2 3

4 5 6 7

**Time complexity:** O(n)

## Use two queues

We can insert the first level in first queue and print it and while popping from the first queue insert its left and right nodes into the second queue. Now start printing the second queue and before popping insert its left and right child nodes into the first queue. Continue this process till both the queues become empty.

void levelOrderTraverse(struct tree\_Node \* arg\_root) {

queue<struct tree\_Node \*>qt1;

queue<struct tree\_Node \*>qt2;

struct tree\_Node \* temp = NULL;

if(NULL == arg\_root) {

return;

}

qt1.push(arg\_root);

while(!qt1.empty() || !qt2.empty()) {

while(!qt1.empty()) {

temp = qt1.front();

qt1.pop();

cout << (temp -> value) << " ";

if(NULL != (temp -> left)) {

qt2.push(temp -> left);

}

if(NULL != (temp -> right)) {

qt2.push(temp -> right);

}

} // end of while(!qt1.empty())

cout << endl;

while(!qt2.empty()) {

temp = qt2.front();

qt2.pop();

cout << (temp -> value) << " ";

if(NULL != (temp -> left)) {

qt1.push(temp -> left);

}

if(NULL != (temp -> right)) {

qt1.push(temp -> right);

}

} // end of while(!qt2.empty())

cout << endl;

} // end of while(!qt1.empty() || !qt2.empty())

return;

}

Level Order Traversal:

1

2 3

4 5 6 7

**Time complexity:** O(n)

## Use queue with delimiter NULL

First insert the root and a null element into the queue. This null element acts as a delimiter. Next pop from the top of the queue and add its left and right nodes to the end of the queue and then print the top of the queue. Continue this process till the queues become empty.

void levelOrderTraverse(struct tree\_Node \* arg\_root) {

queue<struct tree\_Node \*>qt;

struct tree\_Node \* temp = NULL;

if(NULL == arg\_root) {

return;

}

qt.push(arg\_root);

qt.push(NULL);

while(!qt.empty()) {

temp = qt.front();

qt.pop();

if(NULL == temp) {

if(!qt.empty()) {

qt.push(NULL);

}

cout << endl;

}

else {

cout << (temp -> value) << " ";

if(NULL != (temp -> left)) {

qt.push(temp -> left);

}

if(NULL != (temp -> right)) {

qt.push(temp -> right);

}

}

} // end of while

return;

}

Level Order Traversal:

1

2 3

4 5 6 7

# Reverse Level Order Traversal

We have discussed level order traversal of a post in previous post. The idea is to print last level first, then second last level, and so on. Like Level order traversal, every level is printed from left to right.

## Recursive function to print a given level

void reverse\_levelOrderTraverse(struct tree\_Node \* arg\_root) {

int tHeight = 0;

if(NULL == arg\_root) {

return;

}

tHeight = tree\_height(arg\_root);

for(int i = tHeight; i >= 1; i--) {

printGivenLevel(arg\_root, i);

}

return;

}

void printGivenLevel(struct tree\_Node \* arg\_node, int arg\_level) {

if(NULL == arg\_node) {

return;

}

if(1 == arg\_level) {

cout << (arg\_node -> value) << " ";

}

else if(1 < arg\_level){

printGivenLevel((arg\_node -> left), arg\_level - 1);

printGivenLevel((arg\_node -> right), arg\_level - 1);

}

else {

// do nothing

}

return;

}

Preorder Traversal: 1 2 4 5 3 6 7

Reverse Level Order Traversal: 4 5 6 7 2 3 1

## Using Queue and Stack O(n)

void reverse\_levelOrderTraverse(struct tree\_Node \* arg\_root) {

queue<struct tree\_Node \*>qt;

stack<struct tree\_Node \*>st;

struct tree\_Node \* temp = NULL;

if(NULL == arg\_root) {

return;

}

qt.push(arg\_root);

while(!qt.empty()) {

temp = qt.front();

qt.pop();

st.push(temp);

// FIRST VISIT RIGHT NODE SO THAT IT WILL BE POP AFTER LEFT NODE FROM STACK

if(NULL != (temp -> right)) {

qt.push(temp -> right);

}

if(NULL != (temp -> left)) {

qt.push(temp -> left);

}

}

while(!st.empty()) {

temp = st.top();

cout << (temp -> value) << " ";

st.pop();

}

cout << endl;

return;

}

Output:

Preorder Traversal: 1 2 4 5 3 6 7

Reverse Level Order Traversal: 4 5 6 7 2 3 1

# END