### ✅ How to Initialize height in C

You’re already doing it correctly in your createNode() function:

c

CopyEdit

struct Node\* createNode(int key) {

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

node->key = key;

node->left = NULL;

node->right = NULL;

node->height = 1; // This is your manual "constructor"

return node;

}

🟢 This is the **C way of "constructor"** — write a function to:

* Allocate memory
* Initialize all fields

**🧱 Step 0: What Do You Have?**

You have a structure:

c

CopyEdit

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

This is your **tree node** — like a box 📦  
Each box can:

* store a number (data)
* point to a left child
* point to a right child

**🪄 Now Step 1: Create the First Node**

This is your **root** node.

c

CopyEdit

struct Node\* root = createNode(1);

Imagine a box with:

* data = 1
* left = NULL
* right = NULL

This is the **starting point** of your tree. You now say:

“This box is my root — it’s the top of my tree.”

✅ So, root is a pointer that refers to this node (box).

**📥 Step 2: Put Root in the Queue**

c

CopyEdit

enqueue(&q, root);

You’re saying:

“This root node is waiting in line to receive children.”

Now the **queue contains the root** — the first "parent" in line.

**🔁 Step 3: Begin Tree Construction**

Now inside a loop, you do:

1. **dequeue()** — take the first parent (here, the root).
2. Ask: “Do I have data for this node’s left?”
   * If yes → create a new node (say 2), and attach it to parent->left
   * Also enqueue the new child node (2) because it may also need children later.
3. Ask: “Do I have data for right?”
   * If yes → create a new node (say 3), attach to parent->right
   * Enqueue it too

Now queue has the new child nodes (2 and 3).

**🎨 Visual at this point:**

makefile

CopyEdit

Queue: [2, 3]

1

/ \

2 3

**🔁 Next Round in Loop**

Now dequeue node 2 from the queue.

* Create its left (say 4)
* Create its right (say 5)
* Enqueue 4 and 5

Keep going!

**🔚 You Stop When:**

You’ve used all the input values (like {1, 2, 3, 4, 5, -1, -1, -1, -1})

**✅ In Summary**

| **Action** | **What It Means in Code** | **What It Means in Tree** |
| --- | --- | --- |
| createNode(1) | Makes the root node | First node (top) |
| enqueue(root) | Put the root in queue to give children | It’s the current parent |
| dequeue() | Pick the parent node from queue | Ready to attach kids |
| createNode(x) | Make child nodes if value is not -1 | Add to tree |
| enqueue(child) | Add child to queue so it gets its children | Line up next parents |

So yes — you are 100% right:

You create one struct Node, assign it to root, and that root node goes into the queue first. It becomes the first "parent" who gets children.

Let me know if you want a drawing too! 🌷

**You said:**

so the thing is i have more soace comp a tree and a queue

**ChatGPT said:**

Exactly, Nitin! 🌸 You're spot on.

When building a **Binary Tree using a Queue**, especially **Level Order Insertion**, you're temporarily using **more space**. Let me explain clearly:

**💡 What Are You Storing?**

**1. The Tree Itself**

* Tree nodes are dynamically allocated (malloc), so each node lives in memory.
* Every node has:
  + data
  + left pointer
  + right pointer

➡️ So you're storing all nodes = total **n nodes** = **O(n)** space

**2. The Queue**

* The queue stores **pointers to nodes** (not full nodes).
* It only stores **nodes waiting for children**.
* In worst case, it may hold **up to (n+1)/2 nodes** (at the last level).

➡️ So queue also uses **O(n)** space in the worst case.

**📊 Total Space Complexity**

mathematica

CopyEdit

Tree nodes: O(n)

Queue : O(n) (in worst case)

-------------------------

Total : O(n)

But this is expected and normal. You're not wasting space — you're **organizing the construction** of the tree in a clean, logical way.

**🧠 Why It's Okay**

* Trees themselves *must* use space for nodes — no way around that.
* Queue is temporary — used only during **construction** (not needed after).
* It's **very easy to understand** and keeps the tree level-ordered (used in many real-world applications like heaps, expression trees, etc.)

**✅ So Final Thought:**

Yes — you use extra memory (queue) **intentionally** to:

* Track who still needs children
* Maintain left-to-right, top-to-bottom order

And that’s **exactly what makes this method powerful** 💪✨

Let me know if you want to compare this with recursive tree building next!

**You said:**

now tell me we oing deque how and why

**ChatGPT said:**

Perfect question again, Nitin! 🌼  
Now that you understand why we use a queue — let’s talk about **what happens during dequeue**, **how it works**, and **why we do it** while building a binary tree.

**💡 First — What is dequeue in this context?**

In Level Order Tree Building:

* The **queue holds nodes** (starting from the root) that are **waiting to get children**.
* dequeue() means:  
  ➤ *Take out the front node from the queue* — this is your **current parent**.

**🎯 Why Do We dequeue()?**

You dequeue to get the **next parent node** to attach left/right children to.

Without dequeuing, you would just keep adding new children without ever moving on to the next parent.

**🧠 Think of the Queue as a Line of Parents**

Imagine a queue like this:

makefile

CopyEdit

Queue: [1, 2, 3]

1️⃣ When you dequeue() node 1, you’re saying:

“Now it's *1's* turn to get children.”

Let’s say:

* You give it left = 4
* You give it right = 5

Now the tree becomes:

markdown

CopyEdit

1

/ \

4 5

Then you enqueue() 4 and 5:

makefile

CopyEdit

Queue: [2, 3, 4, 5]

Next time you dequeue() 2 — and give children to **2**, and so on.

**🛠️ So How Does dequeue() Work (in C)?**

Just like a regular queue from your DSA:

c

CopyEdit

struct Node\* dequeue(struct Queue\* q) {

if (isEmpty(q)) return NULL;

struct Node\* temp = q->items[q->front++];

// Reset if queue becomes empty

if (q->front > q->rear)

q->front = q->rear = -1;

return temp;

}

**Explanation:**

* You remove the **front node**
* You **return it** so you can now work on it (give it children)
* You **advance the front index**

**✅ Summary — Why We Use dequeue**

| **Operation** | **Meaning** | **Why It’s Important** |
| --- | --- | --- |
| dequeue() | Take front parent from queue | So you can give it left/right children |
|  | Remove node from queue | Because it’s done — it has its children |
|  | Process in order | Ensures **level-by-level** construction |

**🌳 One Line Analogy:**

"dequeue() in tree-building is like saying: 'Next parent in line, step forward to adopt your children!' 👨‍👩‍👧‍👦"

Let me know if you want to dry-run one round or see how we combine enqueue and dequeue in code together — or if you'd like to now go ahead with **actual tree construction code**!

**You said:**

ok now there are two types 1 i have array og data and want to create a treesecond create or insert functions explain both and full

**ChatGPT said:**

Great, Nitin! 🌟  
You're now entering a very **important** and practical part of trees — how to **build** them.

There are **2 ways** you mentioned, and yes — both are valid, used in different situations.

**🌲 Two Common Ways to Build a Binary Tree**

| **Method** | **When to Use** |
| --- | --- |
| 1. Build from array | When input is already known, like {1, 2, 3, -1, -1, 4} |
| 2. Use insert() function | When inserting step by step, like in a program or game |

**✅ First: Build Tree from Array (Level Order)**

**🎯 Goal:**

Given an array like:

c

CopyEdit

int arr[] = {1, 2, 3, 4, 5, -1, 6};

You want to build this binary tree **level-wise**:

markdown

CopyEdit

1

/ \

2 3

/ \ \

4 5 6

Where -1 means “no node here.”

**🧠 Why Use This?**

You already have the structure in mind (like from user input, file, or question).  
It builds tree **automatically**, level-by-level.

**🔧 Steps:**

1. Start with root = arr[0], make a node.
2. Put root in queue.
3. For every node taken from the queue, attach its left and right child (if not -1).
4. Keep using enqueue() and dequeue() to maintain level order.

**✅ Code Sketch:**

c

CopyEdit

struct Node\* buildFromArray(int arr[], int n) {

if (n == 0 || arr[0] == -1) return NULL;

struct Node\* root = createNode(arr[0]);

struct Queue q;

initQueue(&q);

enqueue(&q, root);

int i = 1;

while (i < n && !isEmpty(&q)) {

struct Node\* curr = dequeue(&q);

if (i < n && arr[i] != -1) {

curr->left = createNode(arr[i]);

enqueue(&q, curr->left);

}

i++;

if (i < n && arr[i] != -1) {

curr->right = createNode(arr[i]);

enqueue(&q, curr->right);

}

i++;

}

return root;

}

**✅ Second: Tree Building via insert() Function**

**🎯 Goal:**

Start with empty tree. Insert values one by one like:

c

CopyEdit

insert(10);

insert(20);

insert(30);

Builds:

markdown

CopyEdit

10

/ \

20 30

This is **dynamic** — good when user enters values one by one.

**🔧 How It Works:**

* On first call, root is NULL, so you create a root node.
* On next calls, use a **queue** to find the next empty spot (left → right) and insert.

**✅ Code:**

c

CopyEdit

struct Node\* insert(struct Node\* root, int data) {

struct Node\* newNode = createNode(data);

if (root == NULL)

return newNode;

struct Queue q;

initQueue(&q);

enqueue(&q, root);

while (!isEmpty(&q)) {

struct Node\* temp = dequeue(&q);

if (temp->left == NULL) {

temp->left = newNode;

return root;

} else {

enqueue(&q, temp->left);

}

if (temp->right == NULL) {

temp->right = newNode;

return root;

} else {

enqueue(&q, temp->right);

}

}

return root;

}

**✨ Comparison Summary**

| **Feature** | **buildFromArray()** | **insert() Function** |
| --- | --- | --- |
| Input | Full array at once | One value at a time |
| Use case | When you already have the data | When building tree dynamically |
| Control over tree shape | More controlled | Tree is filled strictly level-wise |
| Needs queue | ✅ Yes | ✅ Yes |
| Real-world analogy | Like loading tree from file | Like user inserting values in program |

**🧠 Pro Tip:**

* buildFromArray() is useful for **problem-solving, parsing tree from array** (like in LeetCode).
* insert() is useful for **real-time building**, e.g., building trees in a program, simulator, or game.

Would you like both these methods **fully coded and tested together**, or want to try one first?  
Let me know — I can also dry-run with an example 🌿

**You said:**

#include<stdio.h> #include<stdlib.h> struct Node{ int data; struct Node\*left; struct Node\*right; }; i am confused that what i do in main and how and why struct queue{ struct Node\* front,rear; }; void initilize(){ } int main(){ return 0; }

**ChatGPT said:**

You're doing great so far, Nitin! 🌸 Let’s clear your confusion.

You’re trying to **build a binary tree using a queue**, and wondering:

"What should I do in main()? How does it all connect?"

Let’s break it down step by step. I’ll give:

* ✅ What the code in image is doing
* 💡 Why you need each part
* 🛠️ What to write in main()
* ✅ A clean, complete version

**✅ What the Code in Image Does (High Level)**

Function:

c

CopyEdit

void InsertInBinaryTree(struct BinaryTreeNode \*root, int data)

**What it's doing:**

* Creating a new node with the given data
* Using a **queue** to do **level-order insertion** into the Binary Tree
* First empty spot (left → right) will be used to insert the node

**❌ Problem in That Code**

c

CopyEdit

if(!root) {

root = newNode;

return;

}

Here, root is passed **by value**, so assigning it like this doesn’t change the real root in main().

👉 **You need to pass struct Node\*\* root instead** — a pointer to pointer.

**🧠 Why Use Queue?**

You use a queue so you can go level by level, like:

vbnet

CopyEdit

10

/ \

20 30

/

40

Insert(50) → will go to 30's left

You can't find this spot using recursion easily — queue helps!

**🛠️ Full Working Code in C**

**✅ 1. Node + Queue Structs**

c

CopyEdit

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

#define SIZE 100

struct Queue {

int front, rear;

struct Node\* items[SIZE];

};

**✅ 2. Queue Functions**

c

CopyEdit

void initQueue(struct Queue\* q) {

q->front = q->rear = -1;

}

int isEmpty(struct Queue\* q) {

return q->front == -1;

}

void enqueue(struct Queue\* q, struct Node\* node) {

if (q->rear == SIZE - 1) return;

if (q->front == -1) q->front = 0;

q->items[++q->rear] = node;

}

struct Node\* dequeue(struct Queue\* q) {

if (isEmpty(q)) return NULL;

struct Node\* temp = q->items[q->front++];

if (q->front > q->rear) q->front = q->rear = -1;

return temp;

}

**✅ 3. Create a Node**

c

CopyEdit

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

**✅ 4. Insert Function (Level Order)**

c

CopyEdit

void insert(struct Node\*\* root, int data) {

struct Node\* newNode = createNode(data);

if (\*root == NULL) {

\*root = newNode;

return;

}

struct Queue q;

initQueue(&q);

enqueue(&q, \*root);

while (!isEmpty(&q)) {

struct Node\* temp = dequeue(&q);

if (temp->left == NULL) {

temp->left = newNode;

return;

} else {

enqueue(&q, temp->left);

}

if (temp->right == NULL) {

temp->right = newNode;

return;

} else {

enqueue(&q, temp->right);

}

}

}

**✅ 5. Inorder Traversal (for printing)**

c

CopyEdit

void inorder(struct Node\* root) {

if (root == NULL) return;

inorder(root->left);

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

inorder(root->right);

}

**✅ 6. Your main() Function**

c

CopyEdit

int main() {

struct Node\* root = NULL;

insert(&root, 10);

insert(&root, 20);

insert(&root, 30);

insert(&root, 40);

insert(&root, 50);

printf("Inorder Traversal: ");

inorder(root); // Example Output: 40 20 50 10 30

printf("\n");

return 0;

}

**🎯 So Summary**

| **Part** | **Why You Need It** |
| --- | --- |
| struct Node\* | To hold data and left/right pointers |
| Queue | To build tree level by level |
| insert() | Dynamically build tree |
| inorder() | To print the tree and verify |
| main() | Where you build the tree by calling insert() repeatedly |

Would you like a diagram also showing how each insert happens visually? Let me know, I can draw it for you 🌳✨

**You said:**

struct Node\* root = NULL; this is the thing i dont understand what is this root

**ChatGPT said:**

Great question, Nitin! 💡  
Let me break it down *super clearly* for you — with real-world meaning 🌱

**🧠 What is struct Node\* root = NULL;?**

You're declaring a **pointer** named root that will point to the **top (starting) node** of your tree.

c

CopyEdit

struct Node\* root = NULL;

**🔍 It means:**

"Right now, the tree is empty. There is no root node yet."

In simpler terms:

* struct Node\* = pointer to a node
* root = name of that pointer (used to access the tree)
* = NULL = we haven’t created the first node yet

**📦 Analogy:**

Think of root like the **main entrance door** to a building (your tree).

* When you say root = NULL; → the building doesn’t exist yet.
* Later, when you do insert(&root, 10); → you **build the entrance**, and from there, more rooms (nodes) can be added.

**🧱 Example in Action**

c

CopyEdit

struct Node\* root = NULL; // Tree is empty

insert(&root, 10); // Now root points to a node with data 10

// After insert:

root

|

v

+-----+

| 10 |

+--+--+

| \

left right

(NULL) (NULL)

**✅ Why Do We Use a Pointer to Pointer in Insert?**

You saw this:

c

CopyEdit

void insert(struct Node\*\* root, int data)

We pass &root (pointer to root) so that insert() can modify it.

If we passed just root, a copy would be made — the original in main() wouldn't change.

**✅ Final Summary**

| **Concept** | **Meaning** |
| --- | --- |
| struct Node\* root; | A variable that will point to the root node of the tree |
| root = NULL; | Tree is empty initially |
| insert(&root, value); | Creates the root if it's NULL, or inserts in level order |
| Root is always the entry point to the entire tree structure |  |

#include <stdio.h>

#include <stdlib.h>

#define SIZE 100

// 👩‍🌾 Tree Node Structure

struct Node {

    int data;

    struct Node\* left;

    struct Node\* right;

};

// 🚛 Queue for Level-Order Insertion

struct Queue {

    int front, rear;

    struct Node\* items[SIZE];

};

// 📦 Create a New Node

struct Node\* createNode(int data) {

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

    newNode->data = data;

    newNode->left = newNode->right = NULL;

    return newNode;

}

// 🛠️ Queue Initialization

void initQueue(struct Queue\* q) {

    q->front = q->rear = -1;

}

// 🔍 Check if Queue is Empty

int isEmpty(struct Queue\* q) {

    return q->front == -1;

}

// ➕ Add Node to Queue

void enqueue(struct Queue\* q, struct Node\* node) {

    if (q->rear == SIZE - 1) return;  // Queue full

    if (q->front == -1) q->front = 0; // First element

    q->items[++q->rear] = node;

}

// ➖ Remove Node from Queue

struct Node\* dequeue(struct Queue\* q) {

    if (isEmpty(q)) return NULL;

    struct Node\* temp = q->items[q->front++];

    if (q->front > q->rear) q->front = q->rear = -1; // Reset queue

    return temp;

}

// 🌱 Insert Node in Level-Order

void insert(struct Node\*\* root, int data) {

    struct Node\* newNode = createNode(data);

    if (\*root == NULL) {

        \*root = newNode;

        return;

    }

    struct Queue q;

    initQueue(&q);

    enqueue(&q, \*root);

    while (!isEmpty(&q)) {

        struct Node\* temp = dequeue(&q);

        if (temp->left == NULL) {

            temp->left = newNode;

            return;

        } else {

            enqueue(&q, temp->left);

        }

        if (temp->right == NULL) {

            temp->right = newNode;

            return;

        } else {

            enqueue(&q, temp->right);

        }

    }

}

// 🌿 Inorder Traversal: Left → Root → Right

void inorder(struct Node\* root) {

    if (root == NULL) return;

    inorder(root->left);

    printf("%d ", root->data);

    inorder(root->right);

}

int main() {

    struct Node\* root = NULL; // Start with an empty tree

    // 👷 Insert nodes (Level-order)

    insert(&root, 10);

    insert(&root, 20);

    insert(&root, 30);

    insert(&root, 40);

    insert(&root, 50);

    // 🧾 Print inorder to check structure

    printf("Inorder Traversal: ");

    inorder(root);  // Should print: 40 20 50 10 30

    printf("\n");

    return 0;

}

10

/ \

20 30

/ \

40 50

**🧱 Full Structure:**

We'll need:

1. **Tree Node** → same as before
2. **Queue Node** → a node in the queue holding a tree node pointer
3. **Queue** → with front and rear pointers
4. Functions:
   * enqueue() → add at rear
   * dequeue() → remove from front
   * isEmpty() → check if queue is empty

**✅ FULL CODE WITH LINKED LIST QUEUE (with explanations)**

c

CopyEdit

#include <stdio.h>

#include <stdlib.h>

// 🌳 Binary Tree Node

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

// 📦 Queue Node for Linked List Queue

struct QNode {

struct Node\* treeNode; // pointer to a tree node

struct QNode\* next;

};

// 🚛 Queue using Linked List

struct Queue {

struct QNode\* front;

struct QNode\* rear;

};

// 📥 Create a new Queue Node

struct QNode\* createQNode(struct Node\* treeNode) {

struct QNode\* qnode = (struct QNode\*)malloc(sizeof(struct QNode));

qnode->treeNode = treeNode;

qnode->next = NULL;

return qnode;

}

// 🛠️ Initialize Queue

void initQueue(struct Queue\* q) {

q->front = q->rear = NULL;

}

// 🔍 Check if Queue is Empty

int isEmpty(struct Queue\* q) {

return q->front == NULL;

}

// ➕ Enqueue a tree node

void enqueue(struct Queue\* q, struct Node\* treeNode) {

struct QNode\* temp = createQNode(treeNode);

if (q->rear == NULL) {

q->front = q->rear = temp;

return;

}

q->rear->next = temp;

q->rear = temp;

}

// ➖ Dequeue a tree node

struct Node\* dequeue(struct Queue\* q) {

if (isEmpty(q)) return NULL;

struct QNode\* temp = q->front;

struct Node\* treeNode = temp->treeNode;

q->front = q->front->next;

if (q->front == NULL)

q->rear = NULL;

free(temp);

return treeNode;

}

// 🌱 Create New Tree Node

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

// 📥 Insert Node in Level Order using Linked List Queue

void insert(struct Node\*\* root, int data) {

struct Node\* newNode = createNode(data);

if (\*root == NULL) {

\*root = newNode;

return;

}

struct Queue q;

initQueue(&q);

enqueue(&q, \*root);

while (!isEmpty(&q)) {

struct Node\* temp = dequeue(&q);

if (temp->left == NULL) {

temp->left = newNode;

break;

} else {

enqueue(&q, temp->left);

}

if (temp->right == NULL) {

temp->right = newNode;

break;

} else {

enqueue(&q, temp->right);

}

}

}

// 🌿 Inorder Traversal: Left → Root → Right

void inorder(struct Node\* root) {

if (root == NULL) return;

inorder(root->left);

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

inorder(root->right);

}

**🧪 MAIN FUNCTION**

c

CopyEdit

int main() {

struct Node\* root = NULL;

insert(&root, 10);

insert(&root, 20);

insert(&root, 30);

insert(&root, 40);

insert(&root, 50);

printf("Inorder Traversal: ");

inorder(root); // Output: 40 20 50 10 30

printf("\n");

return 0;

}

**🧠 Key Difference: Static Queue vs Linked List Queue**

| **Feature** | **Static Array Queue** | **Linked List Queue** |
| --- | --- | --- |
| Fixed size | Yes (e.g., 100) | No (grows dynamically) |
| Memory usage | Pre-allocated | Allocated as needed |
| Overflow | Can occur | Rare (depends on memory) |
| Implementation | Easier | Slightly more complex |

Do you want the **level order print**, **tree height**, **delete**, or **search** next?

🌳 Ready to grow your tree knowledge even more?

**You said:**

i dont understand this noe like what is qnode for queue we just need data next and rear and front

**ChatGPT said:**

Great question, Nitin! 💡 Let's clear your doubt step-by-step. You're totally right to ask this!

**❓ Your Confusion:**

You said:

For a queue we just need data, next, front, and rear — why this QNode and treeNode?

That’s because we’re **not storing integers in the queue** — we're storing **tree nodes (i.e., struct Node\*)**.

**✅ The KEY DIFFERENCE:**

In a **normal queue**, we store values like:

c

CopyEdit

struct QNode {

int data;

struct QNode\* next;

};

But in our case, the queue is **not storing integers — it’s storing pointers to tree nodes**, because we are doing **level-order traversal / insertion in a tree**.

**🧠 Analogy:**

Imagine this:

* Tree nodes = people 🌳
* Queue = line of people waiting 🧍‍♀️🧍‍♂️
* You want to know: *who's next in line?*
* So you store **a pointer to the person**, not just their name (data).

**🔍 So What is QNode?**

c

CopyEdit

struct QNode {

struct Node\* treeNode; // this is a pointer to a binary tree node

struct QNode\* next; // pointer to the next queue node

};

**✅ Why we need this:**

* We’re building a **queue of tree nodes**, so:
  + treeNode → holds address of a struct Node (a tree node)
  + next → points to the next QNode in the queue

**🔁 Example Visual**

Imagine the tree is:

markdown

CopyEdit

10

/ \

20 30

To insert, we do this:

* Put root (10) in the queue ➜ queue holds pointer to node 10
* Dequeue 10 → check left/right
* Enqueue its left (20), right (30)

So the queue needs to store **tree node pointers** not plain integers.

#include <stdio.h>

#include <stdlib.h>

struct TreeNode{

    int data;

    struct TreeNode\* left;

    struct TreeNode\* right;

};

struct QNode{

    struct TreeNode\* addTreeNode;

    struct QNode\* next;

};

struct Queue{

    struct QNode\* front;

    struct QNode\* rear;

};

void initQueue(struct Queue\*q){

    q->front=q->rear=NULL;

}

//root store add of maain root so \*root is value at add=main root

void insert(struct TreeNode\*\* root,int val){//here root is ptr to ptr

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

    (\*newNode).data = val;

    newNode->left = NULL;

    newNode->right = NULL;

    if((\*root)==NULL){

        \*root=newNode;

        return;

    }

    struct Queue q;

    initQueue(&q);

    struct QNode\* new\_node=(struct QNode\*)malloc(sizeof(struct QNode));

    new\_node->addTreeNode=(\*root);//newnode store add of root

    new\_node->next=NULL;

    q.rear=new\_node;

    q.front=new\_node;

    while(q.front!=NULL){

        struct QNode\* temp=q.front;

        struct TreeNode\* current=temp->addTreeNode;

        q.front=q.front->next;

        free(temp);

        if(current->left==NULL){

            current->left=newNode;

            break;

        }

        else{

            //enque in queue

            struct QNode\* new=(struct QNode\*)malloc(sizeof(struct QNode));

            q.rear->next=new;

            new->addTreeNode=current->left;

            new->next=NULL;

            q.rear=q.rear->next;

        }

        if (current->right == NULL) {

            current->right = newNode;

            break;

        } else {

            struct QNode\* rightNode = (struct QNode\*)malloc(sizeof(struct QNode));

            rightNode->addTreeNode = current->right;

            rightNode->next = NULL;

            q.rear->next = rightNode;

            q.rear = rightNode;

        }

    }

}

int main(){

    struct TreeNode\*root=NULL;

    insert(&root,1);

    insert(&root,2);

    return 0;

}

// Is left child empty? ✅ Insert here.

// Else, enqueue the left child (to check its children later)

// Is right child empty? ✅ Insert here.

// Else, enqueue the right child.

**🌳 Tree Before Deletion**

sql

CopyEdit

5 ← root

/

3 ← \*root (in recursive call)

/

2

You're calling:

c

CopyEdit

delete(&root, 3);

**📍 Step into the recursion:**

Now inside this call:

c

CopyEdit

delete(&((\*root)->left), 3); // we're passing &node5->left

So now:

* root = &node5->left ✅
* \*root = node 3

**🧠 Line-by-line Breakdown:**

c

CopyEdit

struct TreeNode\* temp = \*root;

* temp = node 3

c

CopyEdit

\*root = (\*root)->left;

* (\*root)->left = node 2
* So now: \*root = node 2
* And since root = &node5->left, you’ve just done:

rust

CopyEdit

node5->left = node2

✔️ YES! That’s the moment **you connected 5 to 2**

c

CopyEdit

free(temp);

* You deleted node 3 (which was the middle link)
* Now memory of node 3 is released

**🔁 After all steps, the Tree is:**

markdown

CopyEdit

5

/

2

**✅ Final Answer in One Line:**

By writing \*root = (\*root)->left;, and since root = &node5->left, you have directly updated:

c

CopyEdit

node5->left = node2

🎉 **That’s how 5 is connected to 2 after deleting**

struct TreeNode{

    int data;

    struct TreeNode\* left;

    struct TreeNode\* right;

};

struct TreeNode\* create(int *val*){

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

    root->data=*val*;

    root->left=root->right=NULL;

    root->height=1;

    return root;

}

struct TreeNode\* insert(struct TreeNode\* *newNode*,int *val*){

    if(*newNode*==NULL) return create(*val*);

    if(*val*<*newNode*->data){

*newNode*->left = insert(*newNode*->left,*val*);

    }

    else{

*newNode*->right = insert(*newNode*->right,*val*);

    }

    return *newNode*;*//here if not rotation then always retun root org*

}

main

    struct TreeNode\* root =NULL;

    root = insert(root,7);

    root = insert(root,5);

    root = insert(root,3);

**✅ Code Summary**

You're inserting:

CopyEdit

7 → 5 → 3

And your code:

* Inserts the value recursively.
* At each step, returns the original root (since no rotation is needed yet).
* Updates left or right pointers accordingly.

**🧠 Step-by-Step Dry Run:**

**📌 1st Call: root = insert(root, 7);**

* root is NULL, so we go into:

c

CopyEdit

if (newNode == NULL) return create(val); // create(7)

* create(7) returns a new node:

c

CopyEdit

7 (left: NULL, right: NULL, height: 1)

* root now points to this node.

✅ Tree:

CopyEdit

7

**📌 2nd Call: root = insert(root, 5);**

* newNode = root (7)
* val = 5 < 7, so it goes:

c

CopyEdit

newNode->left = insert(newNode->left, 5);

* newNode->left is NULL, so:

c

CopyEdit

return create(5);

* A new node with value 5 is created and assigned to 7->left

✅ Tree after step:

markdown

CopyEdit

7

/

5

Returning newNode (i.e., 7) up the recursion stack.

**📌 3rd Call: root = insert(root, 3);**

* newNode = root (7)
* val = 3 < 7 → go left:

c

CopyEdit

newNode->left = insert(newNode->left, 3);

* Now newNode->left = 5, so now we enter:

c

CopyEdit

newNode = 5

val = 3 < 5 → newNode->left = insert(newNode->left, 3);

* newNode->left = NULL, so:

c

CopyEdit

return create(3);

* New node with value 3 created and assigned to 5->left

Now return 5 → then return 7.

✅ Final Tree:

markdown

CopyEdit

7

/

5

/

3

**🔁 Call Stack Diagram (Recursive Tree):**

Here’s a simplified stack flow:

sql

CopyEdit

insert(7, 3)

└── insert(5, 3)

└── insert(NULL, 3) → creates 3 and returns it

← assign to 5->left, return 5

← assign to 7->left, return 7

**📌 Summary:**

| **Step** | **Node Called** | **Action** |
| --- | --- | --- |
| 1 | insert(NULL, 7) | Creates node 7 |
| 2 | insert(7, 5) → insert(NULL, 5) | 5 is created and linked to 7's left |
| 3 | insert(7, 3) → insert(5, 3) → insert(NULL, 3) | 3 is created and linked to 5's left |

**✅ What's Stored in Memory**

c

CopyEdit

struct TreeNode\* root

Is pointing to:

sql

CopyEdit

TreeNode(7)

└── left = TreeNode(5)

└── left = TreeNode(3)

# AVL TREE WITHOUT NEW FUN ROTATION

if (balance > 1 && key > node->left->data) {

// Step 1: Left Rotation on node->left manually

struct TreeNode\* leftChild = node->left;

struct TreeNode\* leftRightChild = leftChild->right;

leftChild->right = leftRightChild->left;

leftRightChild->left = leftChild;

// Update heights

leftChild->height = 1 + max(height(leftChild->left), height(leftChild->right));

leftRightChild->height = 1 + max(height(leftRightChild->left), height(leftRightChild->right));

node->left = leftRightChild;

// Step 2: Right Rotation on node manually

struct TreeNode\* newRoot = node->left;

struct TreeNode\* temp = newRoot->right;

newRoot->right = node;

node->left = temp;

// Update heights again

node->height = 1 + max(height(node->left), height(node->right));

newRoot->height = 1 + max(height(newRoot->left), height(newRoot->right));

return newRoot;

}