**Arrays in C Programming**

What is an Array?

* Array is an collection of data (a folder as multiple files)
* An array in C is a collection of elements of the same type stored in contiguous memory locations. Each element can be accessed using an index, with the index starting at 0. Arrays are used to store multiple values of the same type in a single variable, making it easier to manage and manipulate collections of data.

Why Use Arrays?

* **Efficient Data Management**: Arrays allow you to store and manage multiple values in a single variable.
* **Ease of Access:** Elements can be accessed directly using an index, which provides a quick way to retrieve or update values.
* **Compact Code:** Arrays help in writing compact and readable code, especially when dealing with a large number of similar items.

**Applications of Arrays**

1. Storing Data: Arrays are commonly used to store lists of items such as integers, floating-point numbers, or characters.

2. Matrix Operations: Arrays can represent matrices in mathematical computations.

3. Data Processing: Useful in algorithms that require sorting, searching, or manipulating a sequence of values.

4. Tables and Records: Arrays are used to manage tables, grids, or other structures where data is organized in rows and columns.

**Declaration & Initialization of Arrays**

Declaration:

To declare an array in C, you specify the type of the elements followed by the array name and the size of the array in square brackets.

Syntax:

type arrayName[arraySize];

**Initialization:**

You can initialize an array at the time of declaration by providing a list of values in curly braces. If the number of values is less than the size specified, the remaining elements are automatically initialized to zero.

Syntax:

type arrayName[arraySize] = {value1, value2, value3, ...};

Example:

#include <stdio.h>

int main() {

// Declaration and initialization of an integer array with 5 elements

int numbers[5] = {10, 20, 30, 40, 50};

// Accessing and printing array elements

for (int i = 0; i < 5; i++) {

printf("Element at index %d: %d\n", i, numbers[i]);

}

return 0;

}

**Output:**

Element at index 0: 10

Element at index 1: 20

Element at index 2: 30

Element at index 3: 40

Element at index 4: 50

**Explanation:**

- `int numbers[5]` declares an integer array named `numbers` with space for 5 elements.

- `{10, 20, 30, 40, 50}` initializes the array with specific values.

- The `for` loop iterates through the array indices (0 to 4) and prints each element.

Accessing Array Elements

In C, you access elements of an array using indices. The index is an integer value that specifies the position of the element within the array. Array indices start from 0, meaning the first element is accessed with index 0, the second with index 1, and so on.

**Syntax:**

arrayName[index]

**Example:**

#include <stdio.h>

int main() {

int numbers[5] = {10, 20, 30, 40, 50};

// Accessing and printing specific array elements

printf("Element at index 0: %d\n", numbers[0]); // Output: 10

printf("Element at index 2: %d\n", numbers[2]); // Output: 30

return 0;

}

**Output:**

Element at index 0: 10

Element at index 2: 30

**Explanation:**

numbers[0] accesses the first element of the array.

numbers[2] accesses the third element of the array.

**Bounds Checking:**

Bounds checking refers to the practice of ensuring that you do not access elements outside the valid range of an array. Accessing elements beyond the declared size of an array leads to undefined behavior and can cause program crashes or corruption of data.

Always ensure that your indices are within the bounds of the array (i.e., between 0 and `arraySize - 1`).

**Example:**

#include <stdio.h>

int main() {

int numbers[3] = {1, 2, 3};

// Correct access

for (int i = 0; i < 3; i++) {

printf("Element at index %d: %d\n", i, numbers[i]);

}

// Incorrect access (out of bounds)

// Uncommenting the following line will cause undefined behavior

// printf("Element at index 5: %d\n", numbers[5]);

return 0;

}

**Explanation:**

- Accessing indices from 0 to 2 is safe.

- Accessing indices beyond the size of the array (e.g., `numbers[5]`) leads to undefined behavior.

**Passing an Array as an Argument to a Function:**

In C, arrays are passed to functions by passing the array's name. The array name acts as a pointer to the first element of the array. Thus, functions can modify the contents of the array.

**Syntax:**

void functionName(type arrayName[], int size);

**Example:**

#include <stdio.h>

void printArray(int arr[], int size) {

for (int i = 0; i < size; i++) {

printf("Element at index %d: %d\n", i, arr[i]);

}

}

int main() {

int numbers[5] = {10, 20, 30, 40, 50};

// Passing array to function

printArray(numbers, 5);

return 0;

}

**Output:**

Element at index 0: 10

Element at index 1: 20

Element at index 2: 30

Element at index 3: 40

Element at index 4: 50

**Explanation:**

- `printArray` is a function that takes an array and its size as arguments.

- The `arr` parameter in `printArray` acts as a pointer to the array `numbers` in `main`.

- The function prints each element of the array.

**Searching Algorithms in C**

**Linear Search**

Linear search, also known as sequential search, is a method for finding a target value within an array by checking each element sequentially from the beginning to the end.

**Complexity:**

Time Complexity: O(n) where n is the number of elements in the array.

Space Complexity: O(1)

**Example:**

#include <stdio.h>

int linearSearch(int arr[], int size, int target) {

for (int i = 0; i < size; i++) {

if (arr[i] == target) {

return i; // Return index of the target element

}

}

return -1; // Target not found

}

int main() {

int numbers[] = {10, 20, 30, 40, 50};

int size = sizeof(numbers) / sizeof(numbers[0]);

int target = 30;

int index = linearSearch(numbers, size, target);

if (index != -1) {

printf("Element %d found at index %d.\n", target, index);

} else {

printf("Element %d not found.\n", target);

}

return 0;

}

**Output:**

Element 30 found at index 2.

**Explanation:**

* linearSearch function iterates through the array and compares each element with the target value.
* It returns the index if found, otherwise -1 if the target is not in the array.

**Binary Search**

Binary search is an efficient algorithm for finding a target value within a sorted array by repeatedly dividing the search interval in half.

**Complexity:**

Time Complexity: O(log n) where n is the number of elements in the array.

Space Complexity: O(1)

**Example:**

#include <stdio.h>

int binarySearch(int arr[], int size, int target) {

int low = 0;

int high = size - 1;

// Initializes **`low`** to **`0`** and **`high`** to **`size - 1`**

while (low <= high) {

int mid = low + (high - low) / 2;

if (arr[mid] == target) {

return mid; // Return index of the target element

}

if (arr[mid] < target) {

low = mid + 1; **// If `target` is less, adjusts `high` to search the left half.**

} else {

high = mid - 1; // If **`target`** is less, adjusts **`high`** to search the left half

}

}

return -1; // Target not found

}

int main() {

// Different sorted array for demonstration

int numbers[] = {5, 12, 23, 34, 45, 56, 67, 78, 89, 100};

int size = sizeof(numbers) / sizeof(numbers[0]);

int target = 56; // sets New target value

int index = binarySearch(numbers, size, target); //calling binary search function to find the target

if (index != -1) {

printf("Element %d found at index %d.\n", target, index);

} else {

printf("Element %d not found.\n", target);

}

return 0;

}

**Output:**

Element 56 found at index 5.

**Sorting Algorithms in C**

**Bubble Sort**

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted. (ascending or descending order)

**Complexity:**

Time Complexity: O(n^2) in the worst and average cases.

Space Complexity: O(1)

**Example:**

**Ascending order:**

#include <stdio.h>

void bubbleSort(int arr[], int size) {

for (int i = 0; i < size - 1; i++) {

for (int j = 0; j < size - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

// Swap elements

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

int main() {

int numbers[] = {64, 34, 25, 12, 22};

int size = sizeof(numbers) / sizeof(numbers[0]);

bubbleSort(numbers, size);

printf("Sorted array: ");

for (int i = 0; i < size; i++) {

printf("%d ", numbers[i]);

}

printf("\n");

return 0;

}

**Output:**

Sorted array: 12 22 25 34 64

**Descending order:**

#include <stdio.h>

void bubbleSort(int arr[], int size) {

for (int i = 0; i < size - 1; i++) {

for (int j = 0; j < size - i - 1; j++) {

if (arr[j] < arr[j + 1]) { // Change the comparison here for descending order

// Swap elements

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

int main() {

int numbers[] = {64, 34, 25, 12, 22};

int size = sizeof(numbers) / sizeof(numbers[0]);

bubbleSort(numbers, size);

printf("Sorted array in descending order: ");

for (int i = 0; i < size; i++) {

printf("%d ", numbers[i]);

}

printf("\n");

return 0;

}

**Output:**

Sorted array in descending order: 64 34 25 22 12

**Selection Sort**

Selection sort is a simple sorting algorithm that divides the input list into two parts: the sorted part and the unsorted part. It repeatedly selects the smallest (or largest) element from the unsorted part and moves it to the end of the sorted part.

**Complexity:**

Time Complexity: O(n^2) in the worst and average cases.

Space Complexity: O(1)

**Example:**

#include <stdio.h>

void selectionSort(int arr[], int size) {

for (int i = 0; i < size - 1; i++) {

int minIndex = i;

for (int j = i + 1; j < size; j++) {

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

// Swap the found minimum element with the first element

int temp = arr[minIndex];

arr[minIndex] = arr[i];

arr[i] = temp;

}

}

int main() {

int numbers[] = {64, 25, 12, 22, 11};

int size = sizeof(numbers) / sizeof(numbers[0]);

selectionSort(numbers, size);

printf("Sorted array: ");

for (int i = 0; i < size; i++) {

printf("%d ", numbers[i]);

}

printf("\n");

return 0;

}

**Output:**

Sorted array: 11 12 22 25 64

**Multidimensional Arrays (2-D ARRAY)**

A multidimensional array is an array of arrays. In C, the most common type of multidimensional array is the two-dimensional array, which can be thought of as a matrix with rows and columns.

**Syntax:**

type arrayName[rows][columns];

**Example:**

#include <stdio.h>

int main() {

// Declaration and initialization of a 2x3 matrix

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

// Accessing and printing elements of the matrix

// The nested `for` loops iterate through each row and column to print the elements.

for (int i = 0; i < 2; i++) {

for (int j = 0; j < 3; j++) {

printf("Element at [%d][%d]: %d\n", i, j, matrix[i][j]);

}

}

return 0;}

**Output :**

Element at [0][0]: 1

Element at [0][1]: 2

Element at [0][2]: 3

Element at [1][0]: 4

Element at [1][1]: 5

Element at [1][2]: 6

**Overview**

* Pointer and Arrays: Pointers can access array elements, and the array name acts as a pointer to its first element.
* -Array of Pointers: Useful for storing strings or multiple memory blocks.
* Dynamic Memory Allocation: Use `malloc()`, `calloc()`, and `realloc()` for dynamic memory management. Always `free()` the memory.
* Double and Triple Pointers: Used for dynamically allocating multidimensional arrays or managing complex pointer structures.

**Pointer and Arrays**

Pointers and arrays are closely related in C. A pointer can be used to access the elements of an array, and the array name itself acts as a pointer to the first element.

Example:

#include <stdio.h>

int main() {

int arr[5] = {10, 20, 30, 40, 50};

int \*ptr = arr; // Pointing to the first element of the array

// Accessing array elements using pointer

for (int i = 0; i < 5; i++) {

printf("Element %d: %d\n", i, \*(ptr + i));

}

return 0;

}

**Explanation:**

- `ptr = arr;` assigns the address of the first element of `arr` to the pointer `ptr`.

- `\*(ptr + i)` accesses the `i`-th element of the array.

**Array of Pointers**

An array of pointers is an array where each element is a pointer. This can be useful for storing a list of strings or dynamically allocated memory blocks.

Example:

#include <stdio.h>

int main() {

const char \*arr[3] = {"Hello", "World", "C Programming"};

for (int i = 0; i < 3; i++) {

printf("%s\n", arr[i]);

}

return 0;

}

**Explanation:**

`arr` is an array of three pointers to `char`, each pointing to the first character of a string.

**Dynamic Memory Allocation (DMA)**

Dynamic memory allocation allows you to allocate memory during runtime using functions like `malloc()`, `calloc()`, and `realloc()`. Memory allocated dynamically must be freed using `free()` to avoid memory leaks.

**1 `malloc()`**

Allocates a block of memory of a specified size and returns a pointer to the first byte.

Example:

#include <stdio.h>

#include <stdlib.h>

int main() {

int \*arr = (int \*)malloc(5 \* sizeof(int)); // Allocate memory for 5 integers

if (arr == NULL) {

printf("Memory allocation failed\n");

return 1;

}

// Initialize and display the array

for (int i = 0; i < 5; i++) {

arr[i] = i + 1;

printf("%d ", arr[i]);

}

free(arr); // Free allocated memory

return 0;

}

Explanation:

* `malloc(5 \* sizeof(int))` allocates memory for 5 integers.
* `free(arr);` deallocates the memory.

**`calloc()`**

Allocates memory for an array of elements and initializes all bytes to zero.

Example:

#include <stdio.h>

#include <stdlib.h>

int main() {

int \*arr = (int \*)calloc(5, sizeof(int)); // Allocate and initialize memory for 5 integers

if (arr == NULL) {

printf("Memory allocation failed\n");

return 1;

}

// Display the array (all elements initialized to 0)

for (int i = 0; i < 5; i++) {

printf("%d ", arr[i]);

}

free(arr); // Free allocated memory

return 0;

}

Explanation:

- `calloc(5, sizeof(int))` allocates memory for 5 integers and initializes them to `0`.

**`realloc()`**

Changes the size of the previously allocated memory block.

**Example:**

#include <stdio.h>

#include <stdlib.h>

int main() {

int \*arr = (int \*)malloc(5 \* sizeof(int));

if (arr == NULL) {

printf("Memory allocation failed\n");

return 1;

}

// Initialize and display the array

for (int i = 0; i < 5; i++) {

arr[i] = i + 1;

printf("%d ", arr[i]);

}

// Reallocate memory for 10 integers

arr = (int \*)realloc(arr, 10 \* sizeof(int));

if (arr == NULL) {

printf("Memory reallocation failed\n");

return 1;

}

// Initialize the new elements

for (int i = 5; i < 10; i++) {

arr[i] = i + 1;

}

// Display the array

printf("\nAfter reallocation: ");

for (int i = 0; i < 10; i++) {

printf("%d ", arr[i]);

}

free(arr); // Free allocated memory

return 0;

}

Explanation:

- `realloc(arr, 10 \* sizeof(int))` changes the size of the array to hold 10 integers.

**`free()`**

Deallocates the memory previously allocated by `malloc()`, `calloc()`, or `realloc()`.

**Double Pointer & Triple Pointer**

Double pointers (pointers to pointers) and triple pointers (pointers to double pointers) are used for more complex data structures, such as dynamically allocated multidimensional arrays or managing pointer arrays.

**Double Pointer Example:**

#include <stdio.h>

#include <stdlib.h>

int main() {

int \*\*arr = (int \*\*)malloc(3 \* sizeof(int \*)); // Allocate memory for 3 integer pointers

for (int i = 0; i < 3; i++) {

arr[i] = (int \*)malloc(3 \* sizeof(int)); // Allocate memory for 3 integers for each pointer

}

// Initialize and display the 2D array

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

arr[i][j] = i + j;

printf("%d ", arr[i][j]);

}

printf("\n");

}

// Free allocated memory

for (int i = 0; i < 3; i++) {

free(arr[i]);

}

free(arr);

return 0;

}

Explanation:

- `int \*\*arr` is a double pointer used to dynamically allocate a 2D array.

**Triple Pointer Example:**

Triple pointers are often used in more complex scenarios, such as dynamic memory allocation for 3D arrays.

#include <stdio.h>

#include <stdlib.h>

int main() {

int \*\*\*arr = (int \*\*\*)malloc(2 \* sizeof(int \*\*)); // Allocate memory for 2 double pointers

for (int i = 0; i < 2; i++) {

arr[i] = (int \*\*)malloc(2 \* sizeof(int \*)); // Allocate memory for 2 integer pointers for each double pointer

for (int j = 0; j < 2; j++) {

arr[i][j] = (int \*)malloc(2 \* sizeof(int)); // Allocate memory for 2 integers for each pointer

}

}

// Initialize and display the 3D array

for (int i = 0; i < 2; i++) {

for (int j = 0; j < 2; j++) {

for (int k = 0; k < 2; k++) {

arr[i][j][k] = i + j + k;

printf("%d ", arr[i][j][k]);

}

printf("\n");

}

printf("\n");

}

// Free allocated memory

for (int i = 0; i < 2; i++) {

for (int j = 0; j < 2; j++) {

free(arr[i][j]);

}

free(arr[i]);

}

free(arr);

return 0;

}

Explanation:

- `int \*\*\*arr` is a triple pointer used to dynamically allocate a 3D array.

**Overview of pointers:**

* **Pointers** store the memory address of another variable, allowing direct memory access and manipulation.
* **Pointer Notation** includes declaring, assigning, and dereferencing pointers.
* **Call by Value** passes a copy of the variable to the function, leaving the original value unchanged.
* **Call by Reference** passes a reference (address) to the function, allowing the original value to be modified directly.

**What is a Pointer? Why Pointer?**

Pointer in C is a variable that stores the memory address of another variable. Pointers are a powerful feature of C that allow you to directly access and manipulate memory, enabling dynamic memory allocation, array management, and passing by reference.

**Why use Pointers?**

- Memory Management: Pointers allow for dynamic memory allocation using functions like `malloc()`, `calloc()`, and `realloc()`.

- Efficient Passing of Large Data: Instead of copying large structures, you can pass a pointer to the data, saving time and memory.

- Array and String Management: Pointers enable efficient handling of arrays and strings.

- Accessing Hardware: Pointers can be used to directly access and manipulate hardware registers.

**Example:**

#include <stdio.h>

int main() {

int var = 10; // Normal variable

int \*ptr = &var; // Pointer to an integer, storing the address of `var`

printf("Value of var: %d\n", var);

printf("Address of var: %p\n", &var);

printf("Value stored in ptr (address of var): %p\n", ptr);

printf("Value pointed to by ptr: %d\n", \*ptr);

return 0;

}

**Output:**

Value of var: 10

Address of var: 0x7ffeefbff5ac // This address may vary

Value stored in ptr (address of var): 0x7ffeefbff5ac

Value pointed to by ptr: 10

**Explanation:**

- `ptr` is a pointer that stores the address of the variable `var`.

- `\*ptr` dereferences the pointer, giving access to the value stored at the address.

**Pointer Notation**

Pointer notation is how pointers are declared, assigned, and dereferenced.

**Declaration:**

int \*ptr; // Declares a pointer to an integer

**Assignment:**

ptr = &var; // Assigns the address of `var` to `ptr`

**Dereferencing:**

\*ptr = 20; // Changes the value of `var` to 20

**Example:**

#include <stdio.h>

int main() {

int a = 5;

int \*p = &a; // p is a pointer to the variable a

printf("Value of a: %d\n", a); // Output: 5

printf("Address of a: %p\n", &a); // Outputs address of a

printf("Value stored in p: %p\n", p); // Same as address of a

printf("Value pointed by p: %d\n", \*p); // Outputs 5, the value of a

\*p = 10; // Changes the value of a to 10

printf("New value of a: %d\n", a); // Output: 10

return 0;

}

**Output:**

Value of a: 5

Address of a: 0x7ffeefbff5ac

Value stored in p: 0x7ffeefbff5ac

Value pointed by p: 5

New value of a: 10

**Call by Value vs. Call by Reference**

**call by Value:**

* In Call by Value, a copy of the actual parameter's value is passed to the function.
* Changes made to the parameter inside the function do not affect the actual variable.

**Call by Reference:**

* In Call by Reference, a reference (or address) to the actual parameter is passed.
* Changes made to the parameter inside the function affect the actual variable.

**Example of Call by Value:**

#include <stdio.h>

void addTen(int num) {

num += 10; // This change will not affect the actual argument

printf("Value inside function (Call by Value): %d\n", num);

}

int main() {

int number = 5;

addTen(number);

printf("Value after function call: %d\n", number); // Original value remains unchanged

return 0;

}

**Output:**

Value inside function (Call by Value): 15

Value after function call: 5

**Explanation:**

- The value of `number` inside `main()` remains `5` because only a copy was modified inside `addTen()`.

**Example of Call by Reference:**

#include <stdio.h>

void addTen(int \*num) {

\*num += 10; // This change will affect the actual argument

printf("Value inside function (Call by Reference): %d\n", \*num);

}

int main() {

int number = 5;

addTen(&number); // Passing the address of number

printf("Value after function call: %d\n", number); // Value is changed to 15

return 0;

}

**Output:**

Value inside function (Call by Reference): 15

Value after function call: 15

**Explanation:**

- The value of `number` inside `main()` changes to `15` because the function directly modifies the memory location of `number` using a pointer.

**Binary Search Tree**

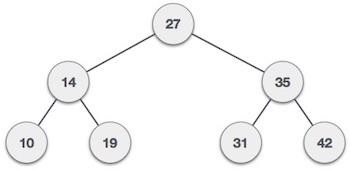
A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentionedproperties

* The left sub-tree of a node has a key less than or equal to its parent node's key.
* The right sub-tree of a node has a key greater than or equal to its parent node's key.

That is -:

left\_subtree (keys) ≤ node (key) ≤ right\_subtree (keys)

Example Binary Tree Representation



27 = parent node

**Basic Operations in BST**

* **Search − Searches an element in a tree.**
* **Insert − Inserts an element in a tree.**
* **Pre-order Traversal − Traverses a tree in a pre-order manner.**
* **In-order Traversal − Traverses a tree in an in-order manner.**
* **Post-order Traversal − Traverses a tree in a post-order manner.**

Defining a Node

Define a node that stores some data, and references to its left and right child nodes.

**Syntax or format to create node in bst**

struct node {

int data;

struct node \*leftChild;

struct node \*rightChild;

};

**Search Operation**

An element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree.

**Algorithm for search in BST**

1. START

2. Check whether the tree is empty or not

3. If the tree is empty, search is not possible

4. Otherwise, first search the root of the tree.

5. If the key does not match with the value in the root,

search its subtrees.

6. If the value of the key is less than the root value,

search the left subtree

7. If the value of the key is greater than the root value,

search the right subtree.

8. If the key is not found in the tree, return unsuccessful search.

9. END

**Implementation:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

struct node

{

int data;

struct node \*left;

struct node \*right;

};

struct node \*newNode (int data)

{

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

node->data = data;

node->left = NULL;

node->right = NULL;

return (node);

}

bool check\_element (struct node \* root, int key)

{

// If root is null, element is not found:Backtrack

if (root == NULL)

{

return false;

}

// Check whether same or not

if (root->data == key)

{

return true;

}

if (root->data > key)

{

// Traverse the left subtree

return check\_element (root->left, key);

}

// Else Traverse the right subtree

else

{

return check\_element (root->right, key);

}

}

int main ()

{

struct node \*root = newNode (80);

root->left = newNode (60);

root->right = newNode (90);

if (check\_element (root, 90))

{

printf ("Found\n");

}

else

{

printf ("Not Found\n");

}

return 0;

}

**Insertion Operation**

An element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

**Algorithm**

1. START

2. If the tree is empty, insert the first element as the root node of the

tree. The following elements are added as the leaf nodes.

3. If an element is less than the root value, it is added into the left

subtree as a leaf node.

4. If an element is greater than the root value, it is added into the right

subtree as a leaf node.

5. The final leaf nodes of the tree point to NULL values as their

child nodes.

6. END

**Implementation**

#include<stdio.h>

#include<stdlib.h>

// Basic struct of Tree

struct node

{

int val;

struct node \*left, \*right;

};

// Function to create a new Node

struct node\* newNode(int item)

{

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

temp->val = item;

temp->left = temp->right = NULL;

return temp;

}

// Function print the node in inorder format, when insertion is complete

void inorder(struct node\* root)

{

if (root != NULL)

{

inorder(root->left);

printf("%d \n", root->val);

inorder(root->right);

}

}

// Here we are finding where to insert the new node so BST is followed

struct node\* insert(struct node\* node, int val)

{

/\* If the tree(subtree) is empty, return a new node by calling newNode func.\*/

if (node == NULL) return newNode(val);

/\* Else, we will do recursion down the tree to further subtrees \*/

if (val < node->val)

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

else if (val > node->val)

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

/\* (Safety) return the node's pointer which is unchanged \*/

return node;

}

int main()

{

struct node\* root = NULL;

root = insert(root, 100);

insert(root, 60);

insert(root, 40);

insert(root, 140);

insert(root, 120);

insert(root, 160);

// Finally printing the tree using inorder

inorder(root);

return 0;

}

**Preorder Traversal**

The preorder traversal operation in a Binary Search Tree visits all its nodes. The root node in it is first printed, followed by its left subtree and then its right subtree.

**Algorithm**

1. START

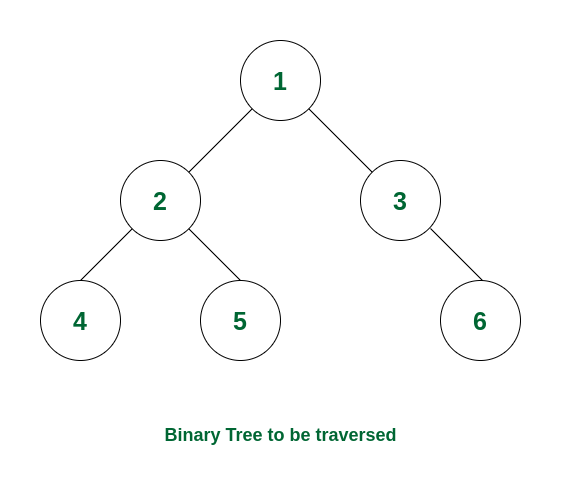
2. Traverse the root node first.

3. Then traverse the left subtree

4. Later, traverse the right subtree

5. END

**Process:**



**Implementation**

#include *<stdio.h>*

#include *<stdlib.h>*

**struct** **Node** {

int data;

**struct** **Node**\* left;

**struct** **Node**\* right;

};

*// Function to perform preorder traversal*

void preorderTraversal(**struct** **Node**\* root) {

*// Base case*

**if** (root == NULL)

**return**;

*// Visit the current node*

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

*// Recur on the left subtree*

preorderTraversal(root->left);

*// Recur on the right subtree*

preorderTraversal(root->right);

}

**struct** **Node**\* newNode(int data) {

**struct** **Node**\* node =

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

node->data = data;

node->left = NULL;

node->right = NULL;

**return** node;

}

int main() {

**struct** **Node**\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

preorderTraversal(root);

**return** 0;

}

**Postorder Traversal**

The other traversals, postorder traversal also visits all the nodes in a Binary Search Tree and displays them. However, the left subtree is printed first, followed by the right subtree and lastly, the root node.

**Algorithm**

1. START

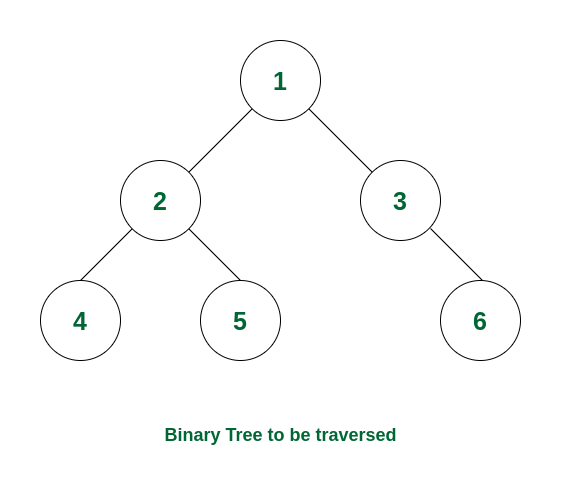
2. Traverse the left subtree

3. Traverse the right subtree

4. Then, traverse the root node

5. END

**Process**



**Implementation**

#include <stdio.h>

#include <stdlib.h>

// Structure for a binary tree node

struct Node {

int data;

struct Node \*left;

struct Node \*right;

};

// Function to create a new node

struct Node\* newNode(int data) {

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

node->data = data;

node->left = NULL;

node->right = NULL;

return node;

}

// Function to perform postorder traversal

void postorderTraversal(struct Node\* node) {

if (node == NULL) {

return;

}

// Recursively traverse left subtree

postorderTraversal(node->left);

// Recursively traverse right subtree

postorderTraversal(node->right);

// Visit the current node

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

}

int main() {

// Construct a binary tree

struct Node\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

// Perform postorder traversal

printf("Postorder traversal: ");

postorderTraversal(root);

printf("\n");

return 0;

}

**Inorder Traversal**

* The direction of traversal for inorder is anti-clockwise
* Rule followed is LCR (Left-Center-Right)

This basically means, that we first try to visit bottommost, the left node then central node and then right and then move our way up to the tree.

**Algorithm**

1. START

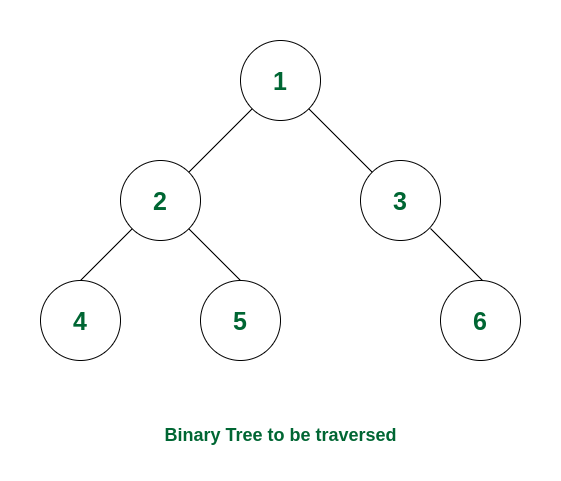
2. Traverse the left subtree

3. Traverse the root node

4. Then, Traverse the right subtree.

5. END

**Process**



**Implementation:**

#include <stdio.h>

#include <stdlib.h>

// Structure of a Binary Tree Node

struct Node {

int data;

struct Node \*left, \*right;

};

// Function to print inorder traversal

void printInorder(struct Node\* node) {

if (node == NULL)

return;

// First recur on left subtree

printInorder(node->left);

// Now deal with the node

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

// Then recur on right subtree

printInorder(node->right);

}

// Function to create a new node

struct Node\* newNode(int v) {

struct Node\* node =

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

node->data = v;

node->left = node->right = NULL;

return node;

}

// Driver code

int main() {

struct Node\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

root->right->right = newNode(6);

// Function call

printf("Inorder traversal of binary tree is: \n");

printInorder(root);

return 0;

}