**Project 2 Report**

**Overview**

This project involves the implementation of a Binary Search Tree (BST) that reads non-negative integers from an input file, constructs a BST, and allows for key value searches. The BST's depth levels, and node counts are printed, and specific details about each key searched by the user are provided. The program ensures the BST contains unique, non-negative integers and has a minimum of 16 keys.

**Main Function**

The `main` function handles file operations, reads input data, validates the input, constructs the BST, and manages user interactions for searching key values. The steps are as follows:

**1. File Handling:** Opens the input file and reads integers into a temporary array.

**2. Input Validation:** Ensures the integers are non-negative, unique, and at least 16 in number.

**3. Array Sorting and BST Construction**: Sorts the array and constructs the BST.

**4. Printing BST Depth Levels:** Prints the depth levels of the BST.

**5. Key Value Search:** Continuously prompts the user for key values to search in the BST and prints the node number and depth level of the key.

int main() {

FILE \*input\_file;

int count = 0;

Node \*root = NULL;

input\_file = fopen("C:\\Users\\TAHA\\Desktop\\input.txt", "r");

if (input\_file == NULL) {

printf("Error!");

exit(0);

}

int i = 0;

int tempArray[100];

while (1) {

fscanf(input\_file, "%d", &tempArray[i++]);

++count;

if (feof(input\_file)) {

break;

}

}

int numbers[count];

for (i = 0; i < count; i++) {

numbers[i] = tempArray[i];

if (numbers[i] <= 0) {

printf("The BST should contain non-negative keys!");

exit(1);

}

int j;

for (j = 0; j < i; j++) {

if (numbers[j] == numbers[i]) {

printf("The BST should have non-replicated keys!");

exit(0);

}

}

}

if (count < 16) {

printf("The BST should have at least 16 non-negative keys!");

exit(0);

}

sorting\_array(numbers, count);

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

root = array\_to\_tree(numbers, 0, size - 1);

printf("Depth level of BST is %d\n", depth\_level(array\_to\_tree(numbers, 0, size - 1)) + 1);

print\_depth\_levels(root);

int key\_value;

int \*key\_value\_ptr;

key\_value\_ptr = &key\_value;

do {

printf("Key value to be searched (Enter 0 to exit): ");

scanf("%d", &key\_value);

if (key\_value == 0) {

printf("Exit");

exit(1);

}

print\_current\_number(root, key\_value\_ptr);

} while (key\_value != 0);

fclose(input\_file);

return 0;

}

**Functions**

**1. Sorting Array**

This function sorts an array using insertion sort.

void sorting\_array(int arr[], int n) {

if (n <= 1) {

return;

}

sorting\_array(arr, n - 1);

int last = arr[n - 1];

int j = n - 2;

while (j >= 0 && arr[j] > last) {

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

j--;

}

arr[j + 1] = last;

}

**2. Array to BST**

This function constructs a balanced BST from a sorted array.

Node \*array\_to\_tree(int arr[], int start, int end) {

if (start > end) {

return NULL;

}

int mid = (start + end) / 2;

Node \*root = new\_node(arr[mid]);

root->left = array\_to\_tree(arr, start, mid - 1);

root->right = array\_to\_tree(arr, mid + 1, end);

return root;

}

**3. New Node**

This function creates a new node for the BST.

Node \*new\_node(int val) {

Node \*new\_node = (Node \*) malloc(sizeof(Node));

new\_node->val = val;

new\_node->left = NULL;

new\_node->right = NULL;

return new\_node;

}

**4. Depth Level**

This function calculates the depth level of the BST.

int depth\_level(Node \*node) {

if (node == NULL) {

return -1;

} else {

int left\_depth = depth\_level(node->left);

int right\_depth = depth\_level(node->right);

return (left\_depth > right\_depth) ? (left\_depth + 1) : (right\_depth + 1);

}

}

**5. Print Depth Levels**

This function prints the number of nodes at each depth level of the BST.

void print\_depth\_levels(Node \*root) {

int height = depth\_level(root);

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

int node\_counter = 0;

print\_node\_numbers(root, i, &node\_counter);

printf("Depth level %d -> %d\n", i, node\_counter);

}

}

**6. Print Node Numbers**

This function counts the number of nodes at a given depth level.

void print\_node\_numbers(Node \*root, int level, int \*node\_counter) {

if (root == NULL) {

return;

}

if (level == 0) {

(\*node\_counter)++;

} else if (level > 0) {

print\_node\_numbers(root->left, level - 1, node\_counter);

print\_node\_numbers(root->right, level - 1, node\_counter);

}

}

**7. Print Current Number**

This function prints the node number and depth level of a searched key.

void print\_current\_number(Node \*root, int \*key\_value\_ptr) {

int height = depth\_level(root);

int search = 0;

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

int node\_counter = 0;

print\_node\_level(root, i, &node\_counter, key\_value\_ptr, &search, i);

if (search == 1) {

break;

}

}

if (search == 0) {

printf("%d is not found in BST\n", \*key\_value\_ptr);

}

}

**8. Print Node Level**

This function counts the element number of the key and prints its depth level.

void print\_node\_level(Node \*root, int level, int \*node\_counter, int \*key\_value\_ptr, int \*search\_ptr, int node\_level) {

if (root == NULL) {

return;

}

if (level == 0) {

(\*node\_counter)++;

if (\*key\_value\_ptr == root->val) {

\*search\_ptr = 1;

if (\*node\_counter == 1) {

printf("At Depth level %d, 1st element\n", node\_level);

} else if (\*node\_counter == 2) {

printf("At Depth level %d, 2nd element\n", node\_level);

} else if (\*node\_counter == 3) {

printf("At Depth level %d, 3rd element\n", node\_level);

} else {

printf("At Depth level %d, %dth element\n", node\_level, \*node\_counter);

}

return;

}

} else if (level > 0) {

print\_node\_level(root->left, level - 1, node\_counter, key\_value\_ptr, search\_ptr, node\_level);

print\_node\_level(root->right, level - 1, node\_counter, key\_value\_ptr, search\_ptr, node\_level);

}

}

**Conclusion**

The program successfully constructs a balanced BST from a set of unique, non-negative integers read from an input file. It validates the input to ensure compliance with the constraints and provides functionality to search for keys within the BST, printing relevant details about their position and depth level. This implementation demonstrates efficient handling of BST operations and user interactions in C.