

Department: COMPUTER SCIENCE AND ENGINEERING

Semester: Fall-2023

Program: Bachelor of Computer Science and Engineering

Course Title: Data Structure

Course Code: 0611-CSE-1207

**ASSIGNMENT 01**

|  |  |
| --- | --- |
| **Student Name:** | Md. Siam Hossain |
| **Student ID:** | 23100084 |
| **Student Batch:** | 26th |
| **Submission Date:** | 14/12/2023 |
|  |  |
| **Course Teacher:** | **Md. Mizanur Rahman**  **Assistant Professor, CSE, RPSU** |

**NARAYANGANJ 2023**

**Problem-1:**

Suppose ′P′ is the following list of 10 elements:

P: 44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66

Find the final position of all values using Quick Sort algorithm.

**Ans**:

* **Algorithm:**
* Input: an array, the index of the first element of the array (low) and the index of the last element (high).
* Output: Nothing will be returned, however. When the algorithm finishes, the input array will be sorted.
* Step 1: Return if the array has one or less elements.
* Step 2: Partition the array such that the last element of the array sits in its proper place. Let the value of that place that is the index be p.
* Step 3: Quick sort the array with index from low to p-1.
* Step 4: Quick sort the part containing the index from p+1 to high of the array.
* **Source Code:**

#include <bits/stdc++.h>

using namespace std;

// Function to find pivot element for quicksort

int pivot(int a[], int low, int high)

{

    int i, j, t;

    int piv = a[high];

    // Partitioning the array

    for (i = low - 1, j = low; j < high; j++)

    {

        if (a[j] < piv)

        {

            i++;

            t = a[j];

            a[j] = a[i];

            a[i] = t;

        }

    }

    t = a[high];

    a[high] = a[i + 1];

    a[i + 1] = t;

    return i + 1;

}

// Recursive function to perform quicksort

void quick(int a[], int low, int high)

{

    if (low >= high)

    {

        return;

    }

    int p = pivot(a, low, high);

    quick(a, low, p - 1);

    quick(a, p + 1, high);

}

int main()

{

    int n = 12;

    int a[n] = {44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66};

    int b[n];          // Copy array

    copy(a, a + n, b); // Copying array

    quick(a, 0, n - 1);

    cout << "Before Sorting\t\tAfter Quick Sort: " << endl;

    for (int i = 0; i < n; i++)

    {

        cout << "A[" << i << "] = " << b[i] << "\t\t"

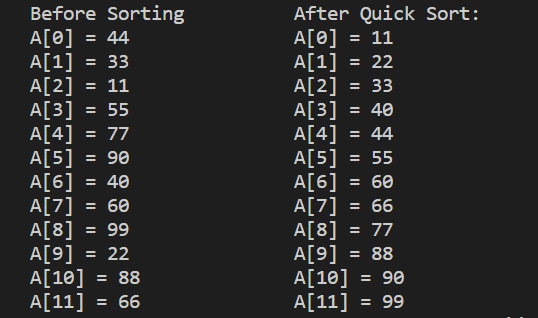
             << "A[" << i << "] = " << a[i] << endl;

    }

    return 0;

}

* **Output:**



Problem-2:

Tower of Hanoi algorithm.

**Ans**:

* **Algorithm:**
* **Step 1: Move n-1 Discs from A to B using C.**
* **Step 2: Move a Disc from A to C.**
* **Step 3: Move n-1 Discs from B to C using A.**
* **Tracing:**

****

* **Source Code:**

#include <bits/stdc++.h>

using namespace std;

void towerOfHanoi(int n, char from\_rod, char using\_rod ,char to\_rod)

{

    if (n > 0)

    {

        towerOfHanoi(n - 1, from\_rod, to\_rod, using\_rod);

        cout << "Move disk " << n << " from rod " << from\_rod

             << " to rod " << to\_rod << endl;

        towerOfHanoi(n - 1, using\_rod,from\_rod, to\_rod);

    }

}

int main()

{

    int N = 3;

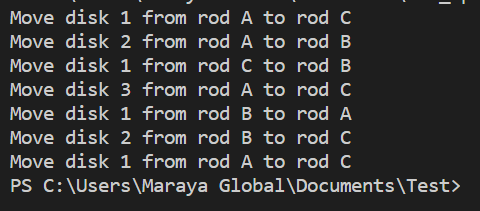
    // A, B and C are names of rods

    towerOfHanoi(N, 'A', 'B', 'C');

    return 0;

}

* **Output:**



**Problem-3:**

Binary Search Tree.

**Ans**:

A Binary Search Tree (BST) is a binary tree data structure with the following properties:

1. **Binary Tree Structure:** Each node in a BST has at most two children, referred to as the left child and the right child.
2. **Ordering Property:** For each node **n**, all nodes in its left subtree have values less than **n**, and all nodes in its right subtree have values greater than **n**. This property ensures that a binary search can be performed efficiently within the tree.

**Here is an example of Binary Search Tree:**

10

/ \

5 17

/ \ / \

3 7 12 19

/ \

1 4

**In this Assignment I will show this 3 operation:**

1. Creating and inserting the data into the node in a Binary Search Tree.
2. Searching the data from Binary Search Tree.
3. **Creating and inserting the data into the node in a Binary Search Tree:**

**How to Insert a value in a Binary Search Tree:**

A new key is always inserted at the leaf by maintaining the property of the binary search tree. We start searching for a key from the root until we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node. The below steps are followed while we try to insert a node into a binary search tree:

* Check the value(arr[i]) to be inserted (say X) with the value of the current node (say value(arr[i])) we are in:
  + If X is less than value(arr[i]) move to the left subtree.
  + Otherwise, move to the right subtree.
* Once the leaf node is reached, insert X to its right or left based on the relation between X and the leaf node’s value(arr[i]).

**Source Code:**

#include <bits/stdc++.h>

using namespace std;

typedef struct Node node;

struct Node

{

    int data;

    Node \*parent;

    Node \*left;

    Node \*right;

}; //...

Node \*create\_node(int item)

{

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

    if (new\_node == NULL)

    {

        cout << "Error! Could not create A New Node" << endl;

        exit(1);

    }

    new\_node->data = item;

    new\_node->parent = NULL;

    new\_node->left = NULL;

    new\_node->right = NULL;

    return new\_node;

} //...

void add\_left\_child(Node \*node, Node \*child)

{

    node->left = child;

    if (child != NULL)

    {

        child->parent = node;

    }

} //...

void add\_right\_child(Node \*node, Node \*child)

{

    node->right = child;

    if (child != NULL)

    {

        child->parent = node;

    }

} //...

Node \*bst\_insert(Node \*root, Node \*node)

{

    Node \*parent\_node, \*current\_node;

    if (root == NULL)

    {

        // The value of root is null means the Tree has no node

        // So if the node is added and that is only node of the Tree

        // So we make this node into root node and return from the function

        root = node;

        return root;

    }

    // We will put the node of parent into parent\_node

    parent\_node = NULL;

    current\_node = root;

    while (current\_node != NULL)

    {

        parent\_node = current\_node;

        if (node->data < current\_node->data)

        {

            current\_node = current\_node->left;

        }

        else

        {

            current\_node = current\_node->right;

        }

        if (node->data < parent\_node->data)

        {

            add\_left\_child(parent\_node, node);

        }

        else

        {

            add\_right\_child(parent\_node, node);

        }

    }

    // Now return thr root node

    return root;

} //...

Node \*create\_bst()

{

    Node \*root, \*node;

    int i;

    int arr[] = {5, 17, 3, 712, 19, 1, 4};

    // First create the Root Node

    root = create\_node(10);

    for (i = 0; i < 7; i++)

    {

        // Create a node with Arr[i]

        node = create\_node(arr[i]);

        // Then insert those created node into Binary Search Tree

        root = bst\_insert(root, node);

    }

    return root;

} //...

void in\_order(Node \*node)

{

    if (node->left != NULL)

    {

        in\_order(node->left);

    }

    cout << node->data << " ";

    if (node->right != NULL)

    {

        in\_order(node->right);

    }

}

int main()

{

    Node \*root = create\_bst();

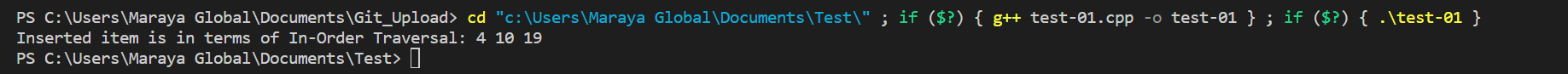
    cout << "Inserted item is in terms of In-Order Traversal: ";

    in\_order(root);

    return 0;

}

**Output:**

****

1. **Searching the data from Binary Search Tree:**

**Algorithm to search for a key in a given Binary Search Tree:**

Let’s say we want to search for the number **X,** We start at the root. Then:

* We compare the value to be searched with the value of the root.
  + If it’s equal we are done with the search if it’s smaller we know that we need to go to the left subtree because in a binary search tree all the elements in the left subtree are smaller and all the elements in the right subtree are larger.
* Repeat the above step till no more traversal is possible
* If at any iteration, key is found, return True. Else False.

**Source code:**

Node \*bst\_search(Node \*root, int item)

{

    Node \*node = root;

    while (node != NULL)

    {

        if (node->data == item)

        {

            return node;

        }

        if (item < node->data)

        {

            node = node->left;

        }

        else

        {

            node = node->right;

        }

    }

    return node;

}

int main()

{

    Node \*root = create\_bst();

    Node \*node;

    node = bst\_search(root, 4);

    if (node != NULL)

    {

        cout << "Node data is found and it is: " << node->data << endl;

    }

    else

    {

        cout << "Node data is not found!" << endl;

    }

    node = bst\_search(root, 8);

    if (node != NULL)

    {

        cout << "Node data is found and it is: " << node->data << endl;

    }

    else

    {

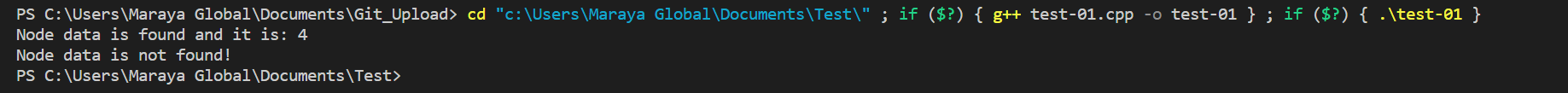
        cout << "Node data is not found!" << endl;

    }

    return 0;

}

**Output:**

****

THANK YOU!