**LAB – 6**

**NAME – TUSHAR MAROO**

**SECTION – A**

**REGISTRATION NUMBER – 190905238**

**ROLL NUMBER – 40**

**Q1.**

**CODE**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int opcount = 0;

struct Node

{

int val;

struct Node \*left, \*right;

};

struct Node \*newNode(int value)

{

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

node->val = value;

node->left = NULL;

node->right = NULL;

return (node);

}

int getNodeCount(struct Node \*tree)

{

opcount++;

if (tree == NULL)

return 0;

return 1 + getNodeCount(tree->right) + getNodeCount(tree->left);

}

int main()

{

struct Node \*root = newNode(1);

int t, i, error = 0;

printf("Enter the the number of nodes except root node : ");

scanf("%d", &t);

while (t--)

{

struct Node \*temp = root;

char str[10];

printf("Enter the location of node : ");

scanf("%s", str);

for (i = 0; i < strlen(str) - 1; i++)

{

if (str[i] == 'L' || str[i] == 'l')

{

temp = temp->left;

}

else if (str[i] == 'R' || str[i] == 'r')

{

temp = temp->right;

}

else

{

printf("Invalid String\n");

break;

}

if (temp == NULL)

{

printf("Invalid String\n");

error = 1;

break;

}

}

if (error == 1)

{

error = 0;

continue;

}

if (str[i] == 'L' || str[i] == 'l')

{

temp->left = newNode(1);

}

else if (str[i] == 'R' || str[i] == 'r')

{

temp->right = newNode(1);

}

else

{

printf("Invalid String\n");

continue;

}

printf("Node Added\n");

}

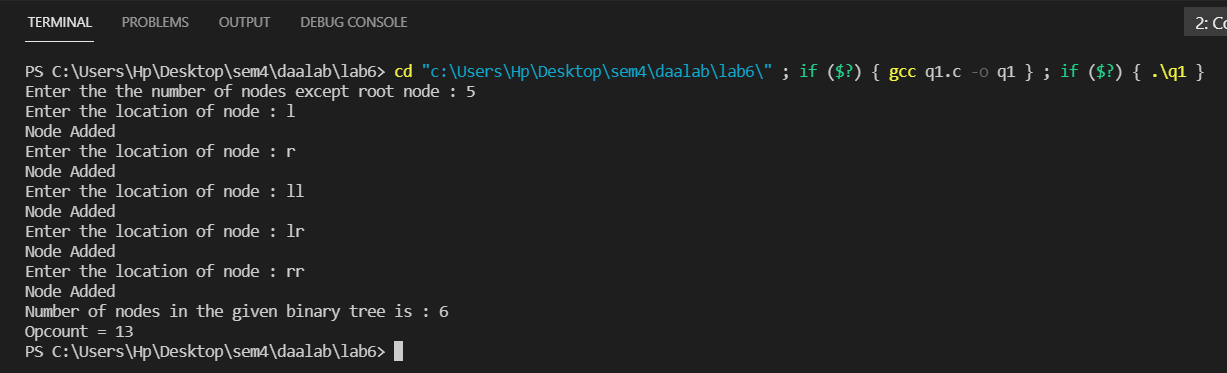
printf("Number of nodes in the given binary tree is : %d\n", getNodeCount(root));

printf("Opcount = %d\n", opcount);

return 0;

}

**OUTPUT**

****

**GRAPH**

**ANALYSIS**

We need to visit all nodes once to include them in count. Thus, the time complexity is O(n). There would be no best and worst case for this.

**Q2.**

**CODE**

#include <stdio.h>

int opcount = 0;

void swap(int \*a, int \*b)

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void quickSort(int A[], int l, int r)

{

if (l < r)

{

int i, pivot = A[r], pivotidx = l;

for (i = l; i < r; i++)

{

opcount++;

if (A[i] <= pivot)

{

swap(&A[i], &A[pivotidx]);

pivotidx++;

}

}

swap(&A[pivotidx], &A[r]);

quickSort(A, l, pivotidx - 1);

quickSort(A, pivotidx + 1, r);

}

}

int main()

{

int n, i;

printf("Enter the number of integers : ");

scanf("%d", &n);

int A[n];

printf("Enter the integers : ");

for (i = 0; i < n; i++)

scanf("%d", &A[i]);

quickSort(A, 0, n - 1);

printf("The sorted array is : ");

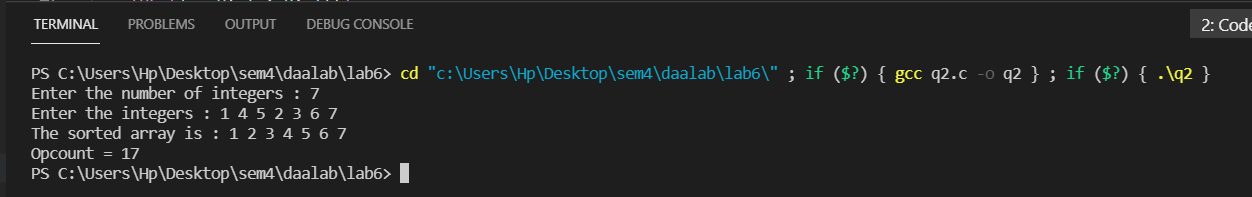
for (i = 0; i < n; i++)

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

printf("\nOpcount = %d\n", opcount);

}

**OUTPUT**

****

**GRAPH**

**ANALYSIS**

For quick sort we choose a pivot and alter the array in such a way that all the numbers smaller than or equal to pivot is on the left side of pivot and all numbers greater than pivot is on the right side of the pivot. We apply this for the left and right sides separately till we get an array of size 1. In this algorithm if we get the largest number as pivot again and again then it would be the worst case. The worst case time complexity is O(n2) and average time complexity is O(nlogn).

**Q3.**

**CODE**

#include <stdio.h>

#include <stdlib.h>

int opcount = 0;

void mergeSort(int A[], int n)

{

if (n <= 1)

return;

int L[n / 2], R[n - n / 2];

int i;

for (i = 0; i < n / 2; i++)

L[i] = A[i];

for (i = n / 2; i < n; i++)

R[i - n / 2] = A[i];

mergeSort(L, n / 2);

mergeSort(R, n - n / 2);

int lp = 0, rp = 0;

i = 0;

while (lp < n / 2 && rp < n - n / 2)

{

opcount++;

if (L[lp] < R[rp])

A[i++] = L[lp++];

else

A[i++] = R[rp++];

}

while (lp < n / 2)

{

opcount++;

A[i++] = L[lp++];

}

while (rp < n - n / 2)

{

opcount++;

A[i++] = R[rp++];

}

}

int main()

{

int n, i;

printf("Enter the number of integers : ");

scanf("%d", &n);

int A[n];

printf("Enter the integers : ");

for (i = 0; i < n; i++)

scanf("%d", &A[i]);

mergeSort(A, n);

printf("The sorted array is : ");

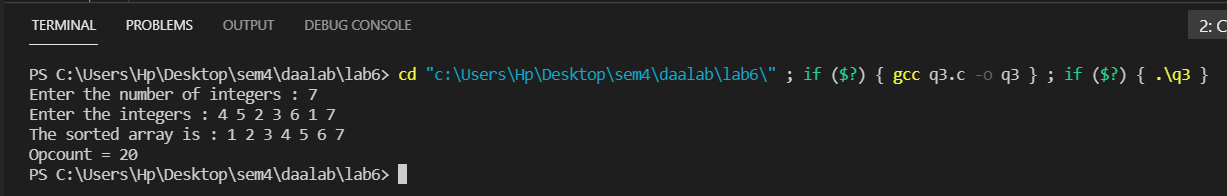
for (i = 0; i < n; i++)

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

printf("\nOpcount = %d\n", opcount);

}

**OUTPUT**

****

**GRAPH**

**ANALYSIS**

For merge sort we divide the array in two parts sort the parts as merge them to form a sorted array. We call this recursively till the size of an array is 1. The time complexity for merge sort is O(nlogn).