Name: **Muhammad Anas Baig**

Enrollment No.: **01-134152-037**

Section: **BS(CS)-4A**

LAB-JOURNAL-13

Exercise 1:

An effective way to represent Mathematical expressions is using the binary trees. The following algorithm can be employed to generate an expression tree from a given postfix expression.

```
1. Scan the postfix expression from left to right.
2. Create a node Curr
3. Get a symbol E from the expression.
4. If the symbol is an operand
        Set this operand as data member of the node Curr
        Push the node on the stack
5. If the symbol is an operator
        T2 = Pop()
        T1 = Pop()
        Attach T1 to the left and T2 to the right of Curr
        Set the operator as data member of the node Curr
        Push Curr (with child nodes attached) onto the stack
6. Repeat Steps 1-5 till end of expression
7. Pop the (only remaining) node from the stack which is a pointer to the root of the
   expression tree.
```

Consider the following class to model a node of the tree

```
class TNode{
public:
        char data;
        TNode * left;
        TNode * right;

};
```

Implement the function `TNode * constructTree(string postfix)` to convert a given postfix expression into an expression tree. Use the stack class in the STL library and create a Stack as follows.

```
stack <TNode *> s;
```

Call the function constructTree() with the postfix expression: "ab+ef*g*-"

Solution:

node.h File:

```cpp
#pragma once
#include<iostream>
using namespace std;
class node
{
public:
    node *left;
    node *right;
    char data;
public:
    node();
    node(char item);
};
```

node.cpp File:

```cpp
#include"node.h"

node::node()
{
    left = right = NULL;
}

node::node(char item)
{
    data = item;
    left = right = NULL;
}
```

bst.h File:

```cpp
#pragma once
#include<iostream>
#include"node.h"
using namespace std;

class bst
{
public:
    node * root;
public:
    bst();
    bool isempty();
    void preTraversal(node *ptr);
    void postTraversal(node *ptr);
    void inTraversal(node *ptr);
};
```

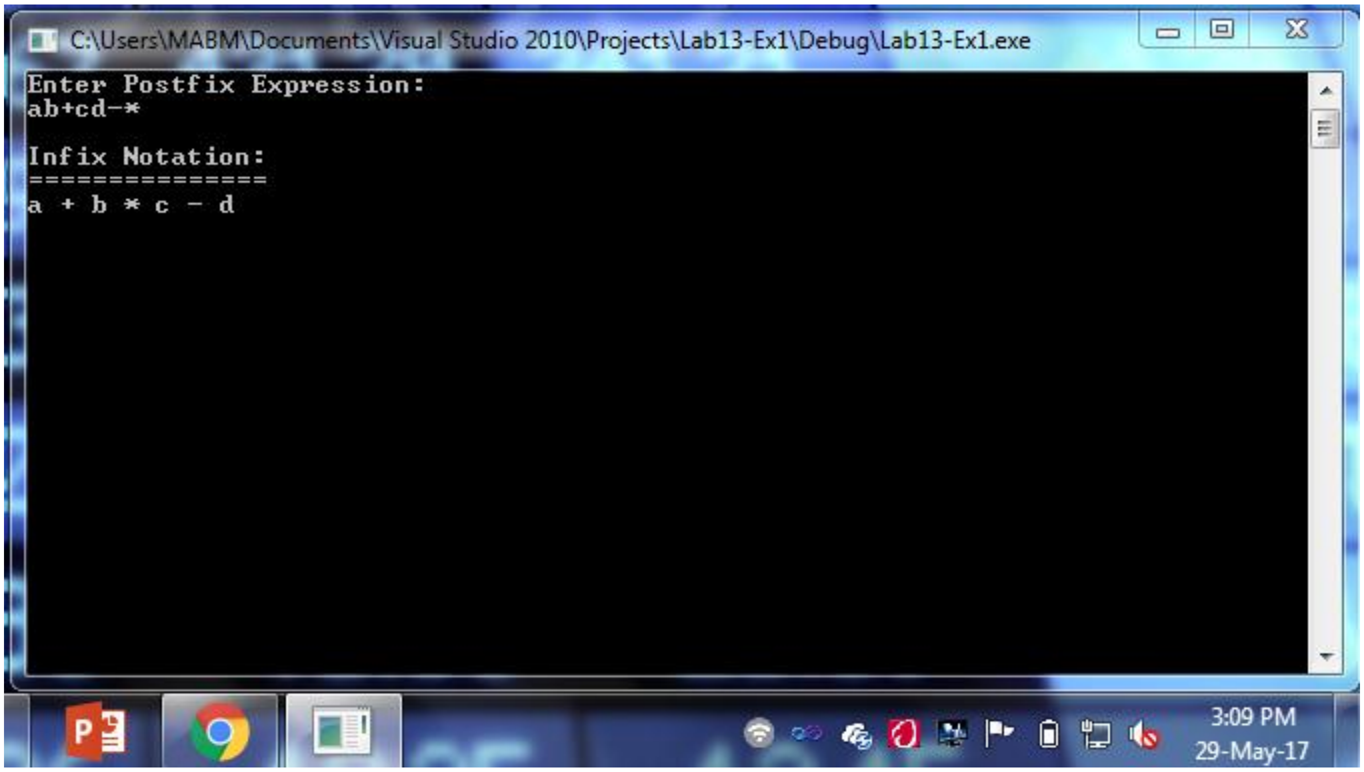
bst.cpp File:

```cpp
#include "bst.h"

bst::bst()
{
    root = NULL;
}

bool bst::isempty()
{
    if (root == NULL)
    {
        return true;
    }
    else
    {
        return false;
    }
}

void bst::preTraversal(node *ptr)
{
    if (ptr != NULL)
    {
        cout << ptr->data << "   ";
        preTraversal(ptr->left);
        preTraversal(ptr->right);
    }
}

void bst::postTraversal(node *ptr)
{
    if (ptr != NULL)
    {
        postTraversal(ptr->left);
        postTraversal(ptr->right);
        cout << ptr->data << "   ";
    }

}

void bst::inTraversal(node *ptr)
{
    if (ptr != NULL)
    {
        inTraversal(ptr->left);
        cout << ptr->data << " ";
        inTraversal(ptr->right);
    }
}
```

main.cpp File:

```cpp
#include"node.h"
#include"bst.h"
#include"string.h"
#include"string"
#include<stack>
#include<iostream>
#include<conio.h>
using namespace std;

node* constructTree(string postfix)
{
    stack <node *>operand;

    node * t1;
    node * t2;

    for(int i=0; i<postfix.size(); i++)
    {
        if(postfix[i] == '+' || postfix[i] == '-' || postfix[i] == '*' || postfix[i] == '/' )
        {
            node *curr1 = new node;
            t2 = operand.top();
            operand.pop();
            t1 = operand.top();
            operand.pop();
            curr1->data = postfix[i]; //operator
            curr1->left = t1; //left operand
            curr1->right = t2; //right operand
            operand.push(curr1); //pushing node
        }
        else //operator
        {
            node *curr = new node;
            curr->data = postfix[i];
            curr->left = NULL;
            curr->right = NULL;
            operand.push(curr);
        }
    }
    node * finalRoot = operand.top();
    operand.pop();

    return finalRoot;
}

void main()
{
    string postfix;

    cout<<"Enter Postfix Expression:"<<endl;
    getline(cin, postfix);

    bst finalTree;
    cout<<endl;
    cout<<"Infix Notation:"<<endl;
    cout<<"==============="<<endl;
    finalTree.inTraversal(constructTree(postfix));

    getch();
}
```

Output:


C:\Users\MABM\Documents\Visual Studio 2010\Projects\Lab13-Ex1\Debug\Lab13-Ex1.exe
3:09 PM
29-May-17

```
Enter Postfix Expression:
ab+cd-*

Infix Notation:
===============
a + b * c - d
```

Exercise 2:

Write program that creates a binary search tree using the BST class developed in the previous lab session. Write functions to implement the following.

1. Find the minimum value in the BST
2. Find the maximum value in the BST
3. Given an integer value, find the address of its parent node.

Solution:

node.h File:

```cpp
#pragma once
#include<iostream>
using namespace std;
class node
{
public:
    node *Left;
    node *Right;
    int data;
    node();
    node(int item);
};
```

node.cpp File:

```cpp
#include"node.h"

node::node()
{
    Left = Right = NULL;
}

node::node(int item)
{
    data = item;
    Left = Right = NULL;
}
```

bst.h File:

```cpp
#pragma once
#include<iostream>
#include"node.h"
using namespace std;
class bst
{
public:
    node * root;
public:
    bst();
    bool isempty();
    void insert(int item);
    bool search(int item);
    void preTraversal(node *ptr);
    void postTraversal(node *ptr);
    void inTraversal(node *ptr);
    int leafCount(node *ptr);
    int nonLeafCount(node *ptr);
    int minValue(node *ptr);
    int maxValue(node *ptr);
};
```

bst.cpp File:

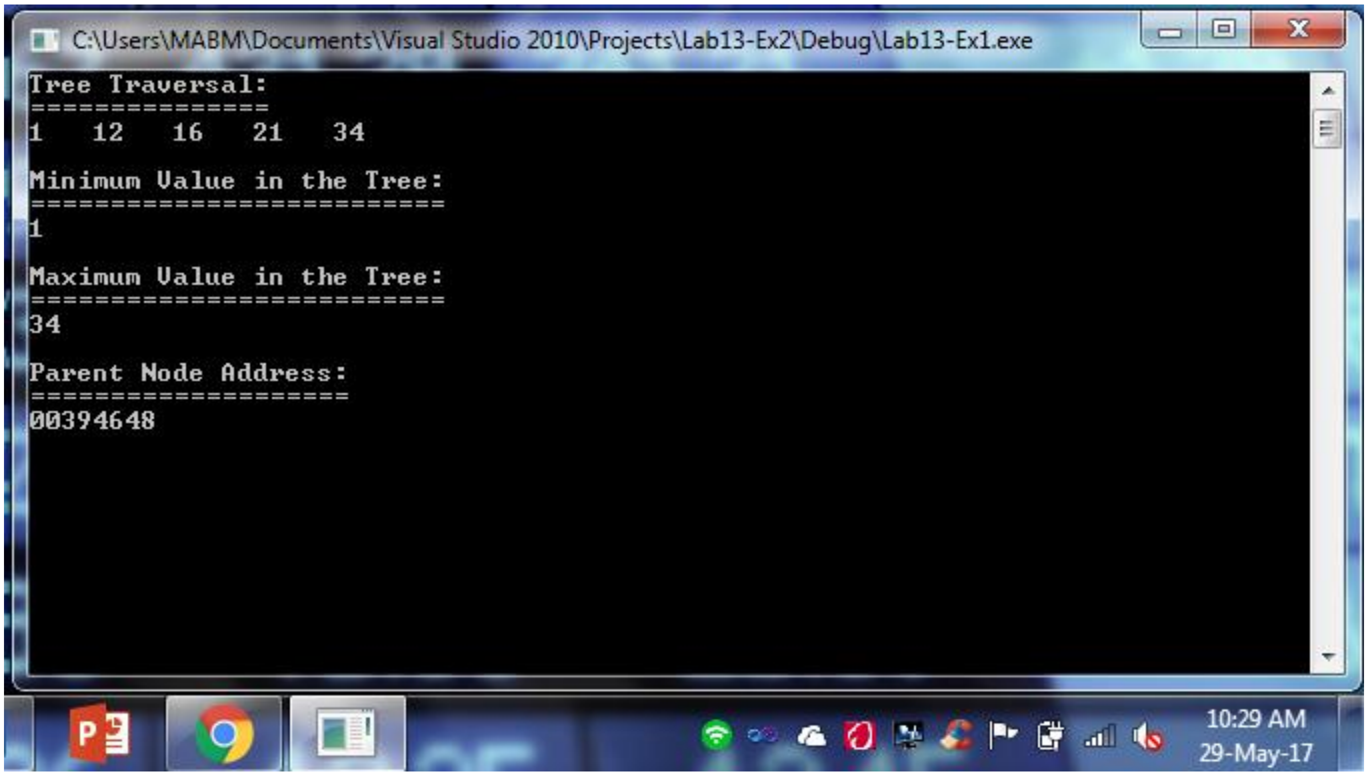
```cpp
#include "bst.h"

bst::bst()
{
    root = NULL;
}

bool bst::isempty()
{
    if (root == NULL)
    {
        return true;
    }
    else
    {
        return false;
    }
}

```

```
void bst::insert(int item)
{
    node * ptr = root;
    node * prev = 0;

    while (ptr != 0)
    {
        prev = ptr;
        if (item <  ptr->data)
        {
            ptr = ptr->Left;
        }
        else if (item > ptr->data)
        {
            ptr = ptr->Right;
        }
        else
        {
            cout << "Value already exist"; return;
        }
    }
    node * temp = new node;
    temp->data = item;
    temp->Left = 0;
    temp->Right = 0;

    if (prev == 0)
    {
        root = temp;
    }
    else if (item < prev->data)
    {
        prev->Left = temp;
    }
    else
    {
        prev->Right = temp;
    }
}

bool bst::search(int item)
{
    node *ptr = root;
    bool found = false;
    for (;;)
    {
        if (found || ptr == NULL)
        {
            break;
        }
        if (item < ptr->data)
        {
            ptr = ptr->Left;
        }
        else if (item > ptr->data)
        {
            ptr = ptr->Right;
        }
        else
        {
            cout << "VALUE FOUND:" << endl;
        }
        found = true;
    }
    return found;
}

void bst::preTraversal(node *ptr)
{
    if (ptr != NULL)
    {
        cout << ptr->data << "   ";
        preTraversal(ptr->Left);
        preTraversal(ptr->Right);

    }
}

void bst::postTraversal(node *ptr)
{
            if (ptr != NULL)
            {
                postTraversal(ptr->Left);
                postTraversal(ptr->Right);
                cout << ptr->data << "   ";
            }

        }

        void bst::inTraversal(node *ptr)
        {
            if (ptr != NULL)
            {
                inTraversal(ptr->Left);
                cout << ptr->data << "   ";
                inTraversal(ptr->Right);
            }

        }

        int bst::leafCount(node  *ptr)
        {
            int count = 0;
            if (ptr == NULL)
            {
                return 0;
            }
            else if (ptr->Left == NULL && ptr->Right == NULL)
            {
                return 1;
            }
            else
            {
                count = leafCount(ptr->Left) + leafCount(ptr->Right);
                return count;
            }
        }

        int bst::nonLeafCount(node *ptr)
        {
            if (ptr == NULL || ptr->Left == NULL && ptr->Right == NULL)
            {
                return 0;

            }

            return 1 + nonLeafCount(ptr->Left) + nonLeafCount(ptr->Right);
        }

        int bst::minValue(node *ptr)
        {
            while (ptr->Left != NULL)
            {
                ptr = ptr->Left;
            }
            return(ptr->data);
        }

        int bst::maxValue(node *ptr)
        {
            while(ptr->Right!=NULL)
            {
                ptr = ptr->Right;
            }
            return(ptr->data);
        }
```

main.cpp File:

```
#include"node.h"
#include"bst.h"
#include<iostream>
#include<conio.h>
using namespace std;

node* getParent(node* curr, int num)
{
    if ((curr->Left->data == num) ||(curr->Right->data == num))
        return curr;
    else if (num < curr->data)
        return getParent(curr->Left, num);
    else
        return getParent(curr->Right, num);
}

void main()
{
    int I = 0;
    bst T;
    T.insert(12);
    T.insert(34);
    T.insert(21);
    T.insert(1);
    T.insert(16);
    cout << "Tree Traversal:" << endl;
    cout << "===============" << endl;
    T.inTraversal(T.root); //traversing root
    cout<<endl;
    cout<<endl;

    cout << "Minimum Value in the Tree:" << endl;
    cout << "==========================" << endl;
    cout<<T.minValue(T.root);
    cout << endl;
    cout<<endl;
    cout << "Maximum Value in the Tree:" << endl;
    cout << "==========================" << endl;
    cout << T.maxValue(T.root);
    cout << endl;
    cout<<endl;
    cout << "Parent Node Address:" << endl;
    cout << "====================" << endl;
    cout<<getParent(T.root, 21);

    _getch();
```

Output:



```
C:\Users\MABM\Documents\Visual Studio 2010\Projects\Lab13-Ex2\Debug\Lab13-Ex1.exe
Tree Traversal:
===============
1   12   16   21   34

Minimum Value in the Tree:
==========================
1

Maximum Value in the Tree:
==========================
34

Parent Node Address:
====================
00394648
```

Exercise 3:

Create a class Student to store the name and ID of each student. Provide appropriate constructors, get(), set() and display() methods in the class.

Prompt user to enter data for 5 students and store the entered data in a BST. Each node of the BST should contain an object of class Student. Data to be arranged with respect to student names.

Once the data is entered, traverse the BST so as to display the sorted list of students (sorted on names). In addition, provide appropriate functionality to allow users search data for a given student using name as the search parameter.

Solution:

node.h File:

```
#pragma once
#include<iostream>
#include"student.h"
using namespace std;

class node
```

```
{
public:
    node *left;
    node *right;
    student data;
    node();
};
```

node.cpp File:

```
#include"node.h"

node::node()
{
    left = right = NULL;
}
```

student.h File:

```
#pragma once
#include<iostream>
#include<string>
using namespace std;

class student
{
public:
    string name;
    int id;
    student();
    void setName(string a);
    void setId(int b);
    string getName();
    int getId();
    void display();
};
```

student.cpp File:

```
#include "student.h"

student::student()
{
    name = '\0';
    id = 0;
}

void student::setName(string a)
{
    name = a;
}

void student::setId(int  b)
{
    id = b;
}

string student::getName()
{
    return name;
}
```

```

int student::getId()
{
    return id;
}

void student::display()
{
    cout << "Name:" << name<<endl;
    cout << "ID:" << id << endl;
}
```

bst.h File:

```
#pragma once
#include<iostream>
#include"node.h"
#include<string>
#include"student.h"
using namespace std;

class bst
{
public:
    node * root;
public:
    bst();
    bool isempty();
    void insert(student item);
    bool search(student item);
    void preTraversal(node *ptr);
    void postTraversal(node *ptr);
    void inTraversal(node *ptr);
};
```

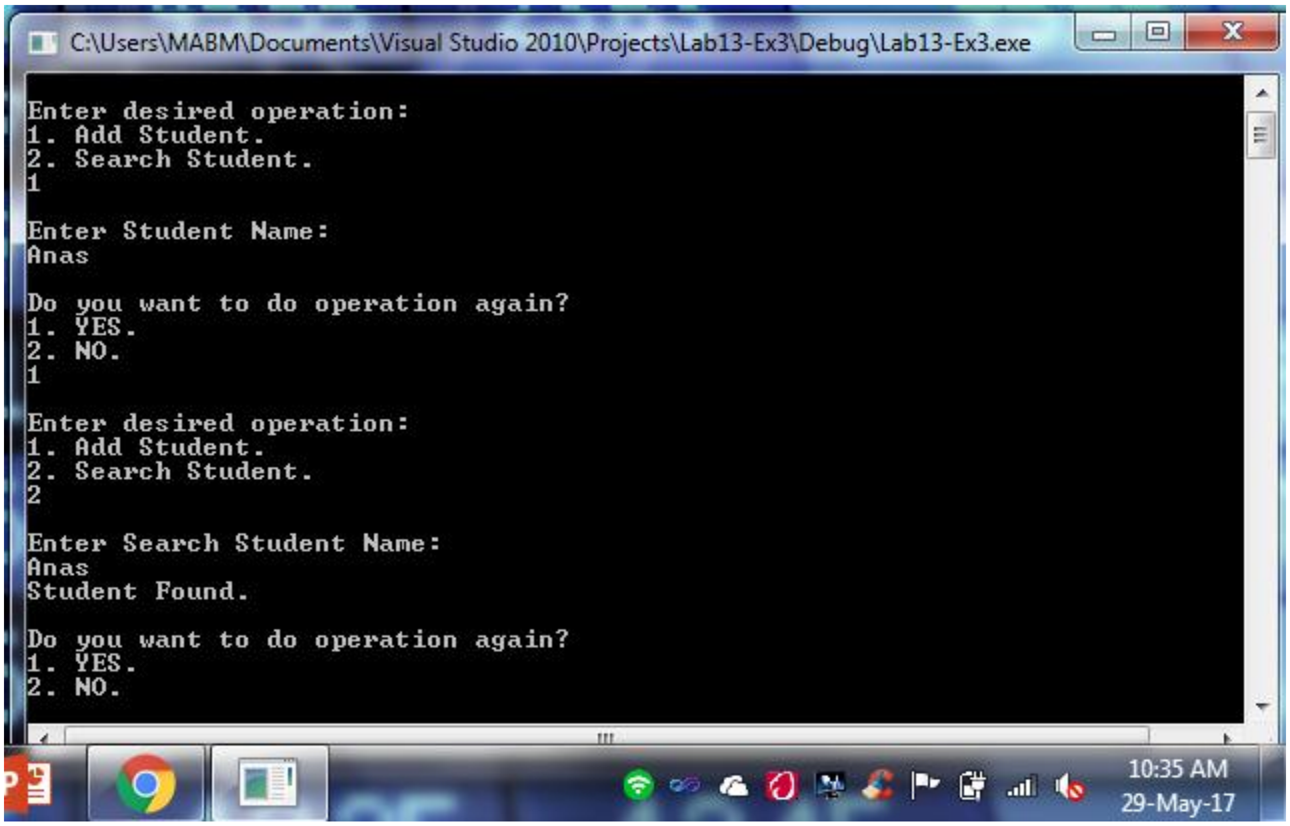
bst.cpp File:

```
#include "bst.h"

bst::bst()
{
    root = NULL;
}

bool bst::isempty()
{
    if (root == NULL)
    {
        return true;
    }
    else
    {
        return false;
    }
}

void bst::insert(student item)
{
    node * ptr = root;
    node * prev = 0;

    while (ptr != 0)
    {
        prev = ptr;
        if (item.getName() < ptr->data.getName())
            ptr = ptr->left;
        else if (item.getName() > ptr->data.getName())
            ptr = ptr->right;
        else {
            cout << "Student Already Exists."; return;
        }
    }

    node * temp = new node;
    temp->data = item;
    temp->left = 0;
    temp->right = 0;

    if (prev == 0)
    {
        root = temp;
    }
    else if (item.getName() < prev->data.getName())
    {
        prev->left = temp;
    }
    else
    {
        prev->right = temp;
    }
}

bool bst::search(student item)
{
    node *ptr = root;
    bool found = false;

    for (;;)
    {
        if (found || ptr == NULL)
        {
            break;
        }
        else if (item.getName() < ptr->data.getName())
        {
            ptr = ptr->left;
        }
        else if (item.getName() > ptr->data.getName())
        {
            ptr = ptr->right;
        }
        else
        {
            found = true;
        }
    }
    return found;
}

void bst::preTraversal(node *ptr)
{
    if (ptr != NULL)
    {
        ptr->data.display();
        preTraversal(ptr->left);
        preTraversal(ptr->right);

    }
}

void bst::postTraversal(node *ptr)
{
    if (ptr != NULL)
    {
        postTraversal(ptr->left);
        postTraversal(ptr->right);
                ptr->data.display();
            }
        }

        void bst::inTraversal(node *ptr)
        {
            if (ptr != NULL)
            {
                inTraversal(ptr->left);
                ptr->data.display();
                inTraversal(ptr->right);
            }
        }
```

main.cpp File:

```
#include<iostream>
#include<conio.h>
#include"bst.h"
#include"student.h"
#include"node.h"
#include"conio.h"
using namespace std;

void main()
{
    string name, search;
    student s;
    bst t;
    int choice1, choice2;

    do
    {
        cout<<endl;
        cout<<"Enter desired operation:"<<endl;
        cout<<"1. Add Student."<<endl;
        cout<<"2. Search Student."<<endl;
        cin>>choice1;
        if(choice1 == 1)
        {
            cout<<endl;
            cout<<"Enter Student Name:"<<endl;
            cin>>name;
            s.setName(name);
            t.insert(s);
        }
        else if(choice1 == 2)
        {
            cout<<endl;
            cout<<"Enter Search Student Name:"<<endl;
            cin>>search;
            s.setName(search);
            if(t.search(s))
            {
                cout<<"Student Found."<<endl;
            }
            else
            {
                cout<<"Student Not Found."<<endl;
            }
        }
        else
        {
            cout<<"Invalid Input"<<endl;
        }
        cout<<endl;
        cout<<"Do you want to do operation again?"<<endl;
        cout<<"1. YES."<<endl;
        cout<<"2. NO."<<endl;
        cin>>choice2;
    }
    while(choice2 == 1);

    getch();
}
```

Output:



C:\Users\MABM\Documents\Visual Studio 2010\Projects\Lab13-Ex3\Debug\Lab13-Ex3.exe

```
Enter desired operation:
1. Add Student.
2. Search Student.
1

Enter Student Name:
Anas

Do you want to do operation again?
1. YES.
2. NO.
1

Enter desired operation:
1. Add Student.
2. Search Student.
2

Enter Search Student Name:
Anas
Student Found.

Do you want to do operation again?
1. YES.
2. NO.
```