



Department of IT and Computer Science
Pak-Austria Fachhochschule: Institute of Applied Sciences
and Technology, Haripur, Pakistan

COMP-201L Data Structures and Algorithms Lab

Lab Report 12

Class: **Computer Science**
Name: **Yaseen Ejaz Ahmed**
Registration No.: **B20F0283CS014**
Semester: **Third**
Submitted to: **Engr. Rafi Ullah**

Instructor Signature

Lab No. 12

Graph traversing

Objectives:

- Implementation of Depth first search for graph traversing
- Implementation of Breadth first search for graph traversing

Tools/Software Required:

C++ Compiler

Introduction:

The breadth first search (BFS) and the depth first search (DFS) are the two algorithms used for traversing and searching a node in a graph. They can also be used to find out whether a node is reachable from a given node or not.

Breadth first search (BFS)

This is a very different approach for traversing the graph nodes. The aim of BFS algorithm is to traverse the graph as close as possible to the root node. Queue is used in the implementation of the breadth first search.

Depth-first search (DFS)

A Depth-first search (DFS) algorithm begins by expanding the initial node and generating its successors. In each subsequent step, DFS expands one of the most recently generated nodes. If this node has no successors (or cannot lead to any solutions), then DFS backtracks and expands a different node. In some DFS algorithms, successors of a node are expanded in an order determined by their heuristic values. A major advantage of DFS is that its storage requirement is linear in the depth of the state space being searched.

Lab Tasks:

Lab Task 01: Write a program for the Breadth First Search for the following rules

Code:

```
#include <iostream>
using namespace std;

class node
{
    public:

    char data;
    node* next;
    node* head;
    node* ptr;

    void enqueue(char ch)
    {
        node *temp = new node();
        temp->data=ch;
        temp->next=NULL;

        if(head==NULL)
        {
            head=temp;
        }

        else
        {
            ptr=head;
            while(ptr->next!=NULL)
            {
                ptr=ptr->next;
            }
            ptr->next=temp;
        }
    }

    void dequeue()
    {
        node *temp=head;
```

```

        char ch=temp->data;
        head=temp->next;
        delete temp;
        cout<<ch;
    }

}q;

class tree
{
    public:
        tree* head=NULL;
        tree* l=NULL;
        tree* r=NULL;
        tree* m=NULL;
        char data;

    void Graph()
    {
        tree *temp=new tree;
        temp->data='S';
        head=temp;

        tree* temp1=new tree;
        temp1->data='A';

        tree* temp2=new tree;
        temp2->data='B';

        tree* temp3=new tree;
        temp3->data='C';

        tree* temp4=new tree;
        temp4->data='D';

        head->l=temp1;
        head->m=temp2;
        head->r=temp3;

        temp1->r=head;
        temp2->r=head;
        temp3->l=head;

        temp1->l=temp4;
        temp2->l=temp4;
    }
};

```

```

        temp3->r=temp4;

        temp4->l=temp1;
        temp4->m=temp2;
        temp4->r=temp2;
    }

    show()
    {
        cout<<head->data;
        cout<<head->l->data;
        cout<<head->m->data;
        cout<<head->r->data;
        cout<<head->l->l->data;
    }

    void BFS()
    {
        tree* ptr=head;
        q.enqueue(head->data);
        head->data='X';
    }

};

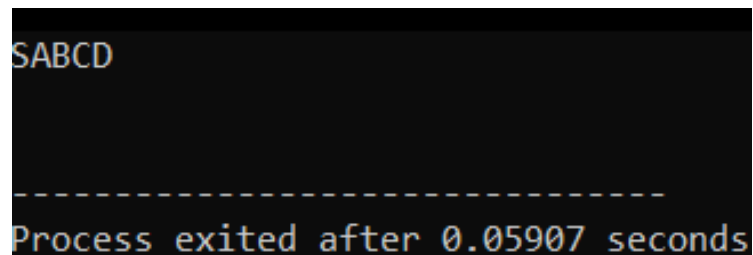
int main()
{
    tree t;
    t.Graph();
    t.show();

    cout<<endl<<endl;
    t.BFS();

}

```

Output:



```

SABCD
-----
Process exited after 0.05907 seconds

```

Results & Observations:

In this lab we have learnt about different traversing algorithms. We can use these algorithms to check each and every node in a graph. We can check the child first or the parent first depending on the problem. We can use Queues and stacks for saving the traversed nodes.