Assignment 5 – Graphs and Hashing

Name: Kartik Banshi Katkar

Rollno : 29

Prn - 12111367

SY-IT-A

**Q.**

**A. Implement BFS and DFS algorithm- Graph**

**B. Implement Linear probing in Hashing**

**Code:**

#include <iostream>

#include <cstdio>

#include <cstdlib>

#include <conio.h>

#include <list>

using namespace std;

const int TABLE\_SIZE = 10;

int a[20][20], reach[20], n;

class HashNode

{

    public:

        int key;

        int value;

        HashNode(int key, int value)

        {

            this->key = key;

            this->value = value;

        }

};

/\*

 \* DeletedNode Class Declaration

 \*/

class DeletedNode:public HashNode

{

    private:

        static DeletedNode \*entry;

        DeletedNode():HashNode(-1, -1)

        {}

    public:

        static DeletedNode \*getNode()

        {

            if (entry == NULL)

                entry = new DeletedNode();

            return entry;

        }

};

DeletedNode \*DeletedNode::entry = NULL;

/\*

 \* HashMap Class Declaration

 \*/

class HashMap

{

    private:

        HashNode \*\*htable;

    public:

        HashMap()

        {

            htable = new HashNode\* [TABLE\_SIZE];

            for (int i = 0; i < TABLE\_SIZE; i++)

            {

                htable[i] = NULL;

            }

        }

        //  Destructors are usually used to deallocate memory and do other cleanup for a class object and its class members when the object is destroyed. A destructor is called for a class object when that object passes out of scope or is explicitly deleted.

        ~HashMap()

        {

            for (int i = 0; i < TABLE\_SIZE; i++)

            {

                if (htable[i] != NULL && htable[i] != DeletedNode::getNode())

                    delete htable[i];

            }

            delete[] htable;

        }

        /\*

         \* Hash Function

         \*/

        int HashFunc(int key)

        {

            return key % TABLE\_SIZE;

        }

        /\*

         \* Insert Element at a key

         \*/

        void Insert(int key, int value)

        {

            int hash\_val = HashFunc(key);

            int init = -1;

            int deletedindex = -1;

            while (hash\_val != init && (htable[hash\_val]

                            == DeletedNode::getNode() || htable[hash\_val]

                            != NULL && htable[hash\_val]->key != key))

            {

                if (init == -1)

                    init = hash\_val;

                if (htable[hash\_val] == DeletedNode::getNode())

                    deletedindex = hash\_val;

                hash\_val = HashFunc(hash\_val + 1);

            }

            if (htable[hash\_val] == NULL || hash\_val == init)

            {

                if(deletedindex != -1)

                    htable[deletedindex] = new HashNode(key, value);

                else

                    htable[hash\_val] = new HashNode(key, value);

            }

            if(init != hash\_val)

            {

                if (htable[hash\_val] != DeletedNode::getNode())

                {

                    if (htable[hash\_val] != NULL)

                    {

                        if (htable[hash\_val]->key == key)

                            htable[hash\_val]->value = value;

                    }

                }

                else

                    htable[hash\_val] = new HashNode(key, value);

            }

        }

        /\*

         \* Search Element at a key

         \*/

        int Search(int key)

        {

            int hash\_val = HashFunc(key);

            int init = -1;

            while (hash\_val != init && (htable[hash\_val]

                            == DeletedNode::getNode() || htable[hash\_val]

                            != NULL && htable[hash\_val]->key != key))

            {

                if (init == -1)

                    init = hash\_val;

                hash\_val = HashFunc(hash\_val + 1);

            }

            if (htable[hash\_val] == NULL || hash\_val == init)

                return -1;

            else

                return htable[hash\_val]->value;

        }

        /\*

         \* Remove Element at a key

         \*/

        void Remove(int key)

        {

            int hash\_val = HashFunc(key);

            int init = -1;

            while (hash\_val != init && (htable[hash\_val]

                            == DeletedNode::getNode() || htable[hash\_val]

                            != NULL && htable[hash\_val]->key != key))

            {

                if (init == -1)

                    init = hash\_val;

                hash\_val = HashFunc(hash\_val + 1);

            }

            if (hash\_val != init && htable[hash\_val] != NULL)

            {

                delete htable[hash\_val];

                htable[hash\_val] = DeletedNode::getNode();

            }

        }

};

class Graph {

  int numVertices;

  list<int>\* adjLists;

  bool\* visited;

   public:

  Graph(int vertices);

  void addEdge(int src, int dest);

  void BFS(int startVertex);

};

// Create a graph with given vertices,

// and maintain an adjacency list

Graph::Graph(int vertices) {

  numVertices = vertices;

  adjLists = new list<int>[vertices];

}

// Add edges to the graph

void Graph::addEdge(int src, int dest) {

  adjLists[src].push\_back(dest);

  adjLists[dest].push\_back(src);

}

// BFS algorithm

void Graph::BFS(int startVertex) {

  visited = new bool[numVertices];

  for (int i = 0; i < numVertices; i++)

    visited[i] = false;

  list<int> queue;

  visited[startVertex] = true;

  queue.push\_back(startVertex);

  list<int>::iterator i;

  while (!queue.empty()) {

    int currVertex = queue.front();

    cout << "Visited " << currVertex << " ";

    queue.pop\_front();

    for (i = adjLists[currVertex].begin(); i != adjLists[currVertex].end(); ++i) {

      int adjVertex = \*i;

      if (!visited[adjVertex]) {

        visited[adjVertex] = true;

        queue.push\_back(adjVertex);

      }

    }

  }

}

void dfs(int v)

{

    int i;

    reach[v] = 1;

    for (i = 1; i <= n; i++)

        if (a[v][i] && !reach[i])

        {

            printf("\n %d->%d", v, i);

            dfs(i);

        }

}

void dfsfunction()

{

    int i, j, count = 0;

    printf("\n Enter number of vertices:");

    scanf("%d", &n);

    for (i = 1; i <= n; i++)

    {

        reach[i] = 0;

        for (j = 1; j <= n; j++)

            a[i][j] = 0;

    }

    printf("\n Enter the adjacency matrix:\n");

    for (i = 1; i <= n; i++)

        for (j = 1; j <= n; j++)

            scanf("%d", &a[i][j]);

    dfs(1);

    printf("\n");

    for (i = 1; i <= n; i++)

    {

        if (reach[i])

            count++;

    }

    if (count == n)

        printf("\n Graph is connected");

    else

        printf("\n Graph is not connected");

    getch();

}

void bfsfunction()

{

int no;

cout << "\n Enter the no of vertices\n";

    cin >> no;

    Graph g(no);

    int nedge;

    cout << "\n Enter the no of edges you want to add\n";

    cin >> nedge;

    for(int i = 0;i<nedge;i++)

    {

        int a,b;

        cout << "\n Enter the Vertices for the edges\n";

    cin >> a>>b;

         g.addEdge(a, b);

    }

int start;

cout << "\n Enter the starting vertex \n";

    cin >> start;

  g.BFS(start);

}

int linearprobing()

{

HashMap hash;

    int key, value;

    int choice;

    while(1)

    {

        cout<<"\n----------------------"<<endl;

        cout<<"Operations on Hash Table"<<endl;

        cout<<"\n----------------------"<<endl;

        cout<<"1.Insert element into the table"<<endl;

        cout<<"2.Search element from the key"<<endl;

        cout<<"3.Delete element at a key"<<endl;

        cout<<"4.Exit"<<endl;

        cout<<"Enter your choice: ";

        cin>>choice;

        switch(choice)

        {

        case 1:

            cout<<"Enter element to be inserted: ";

            cin>>value;

            cout<<"Enter key at which element to be inserted: ";

            cin>>key;

            hash.Insert(key, value);

            break;

        case 2:

            cout<<"Enter key of the element to be searched: ";

            cin>>key;

            if(hash.Search(key) == -1)

            {

                cout<<"No element found at key "<<key<<endl;

                continue;

            }

            else

            {

                cout<<"Element at key "<<key<<" : ";

                cout<<hash.Search(key)<<endl;

            }

            break;

        case 3:

            cout<<"Enter key of the element to be deleted: ";

            cin>>key;

            hash.Remove(key);

            break;

        case 4:

            return 0;

        default:

           cout<<"\nEnter correct option\n";

       }

    }

return 0;

}

int main()

{

    int choice;

    do{

    cout << "\n Menu \n 1.Linear Probing \n 2.BFS \n 3.DFS \n 4.Exit \n";

    cin >> choice;

    switch (choice)

    {

    case 1:

        linearprobing();

        break;

    case 2:

        bfsfunction();

        break;

    case 3:

        dfsfunction();

        break;

    case 4:

        cout << "\n Thank you for using the program  \n";

        break;

    default:

        cout << "\n Invalid Choice \n";

        break;

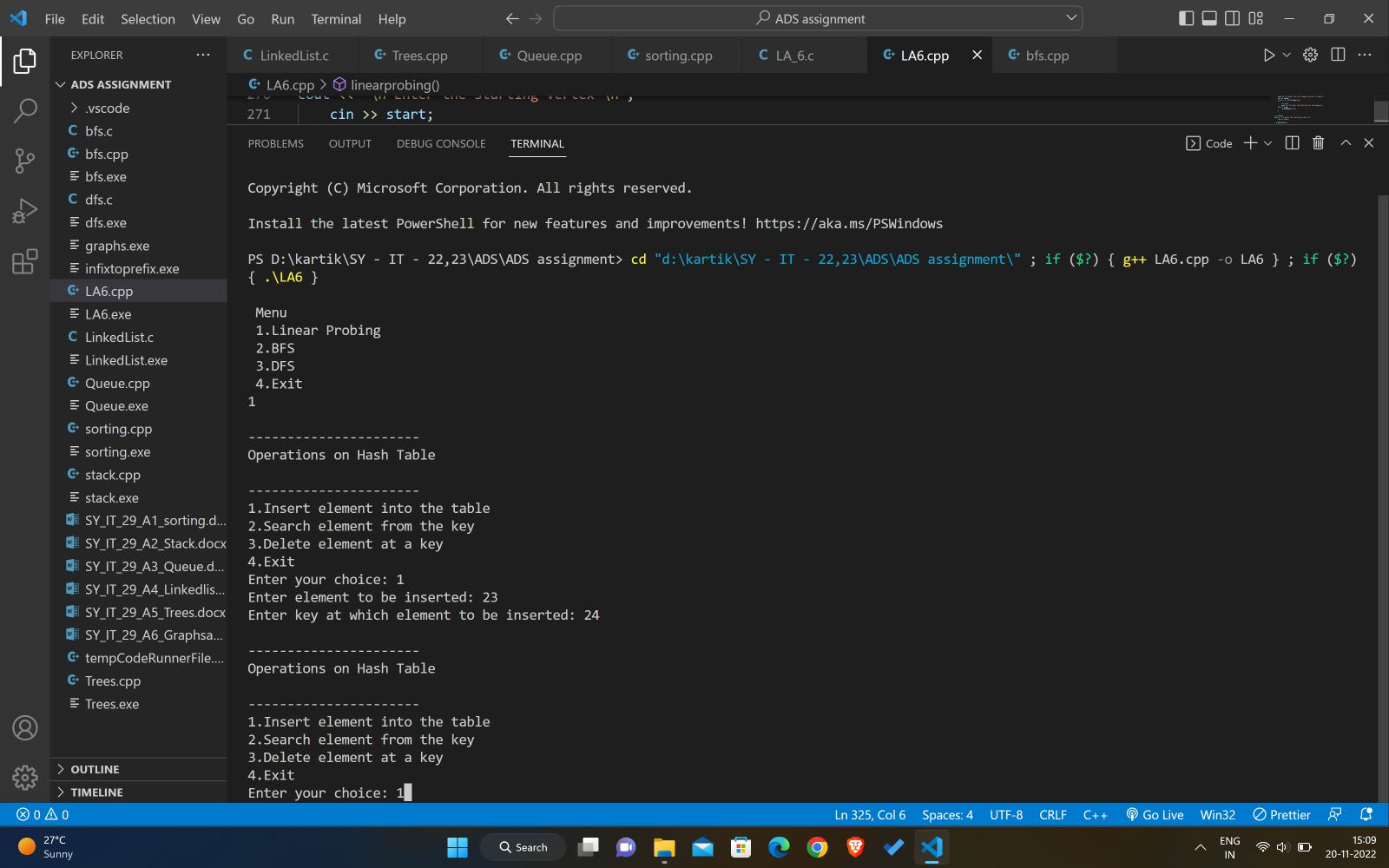
    }

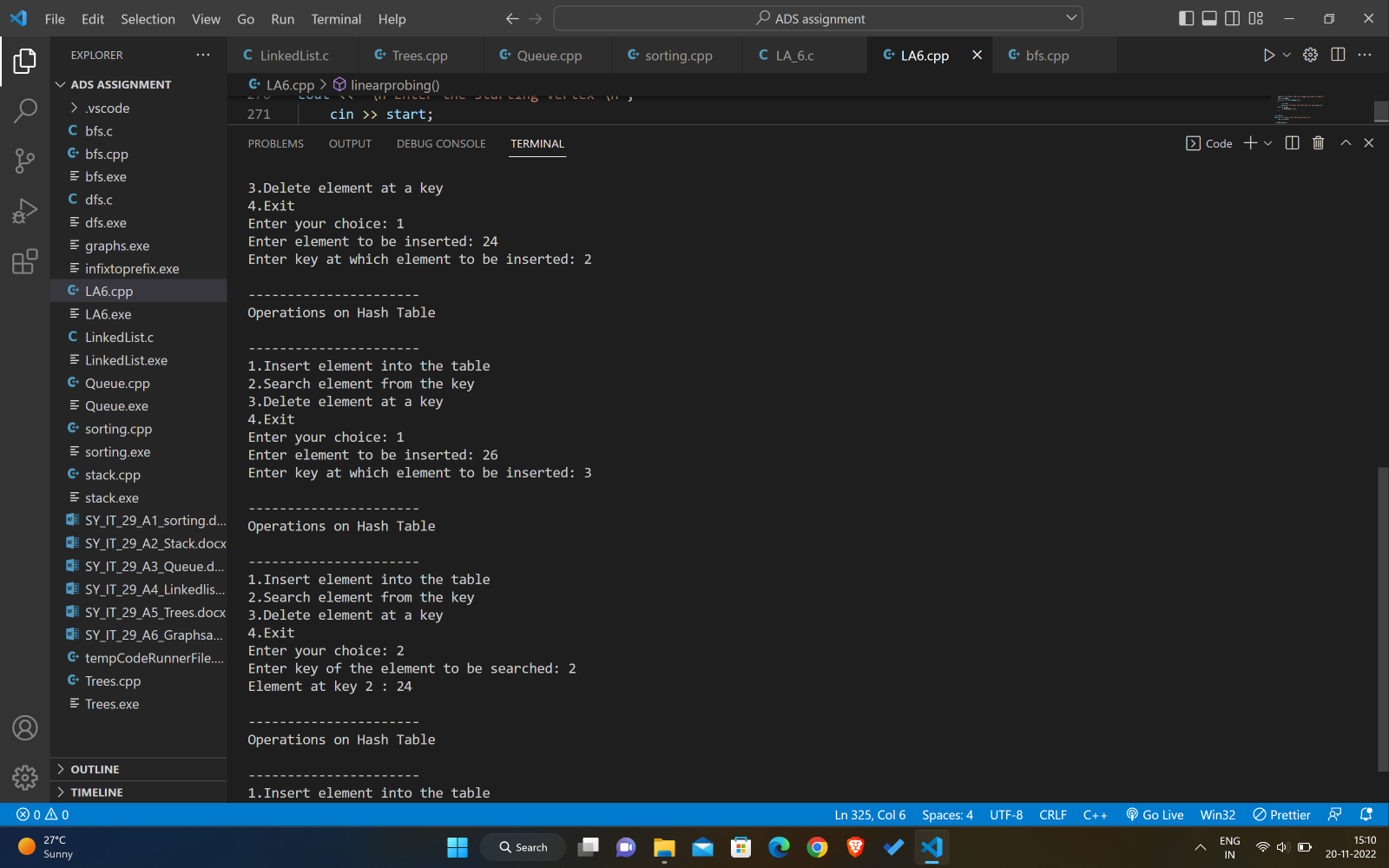
    }while(choice != 4);

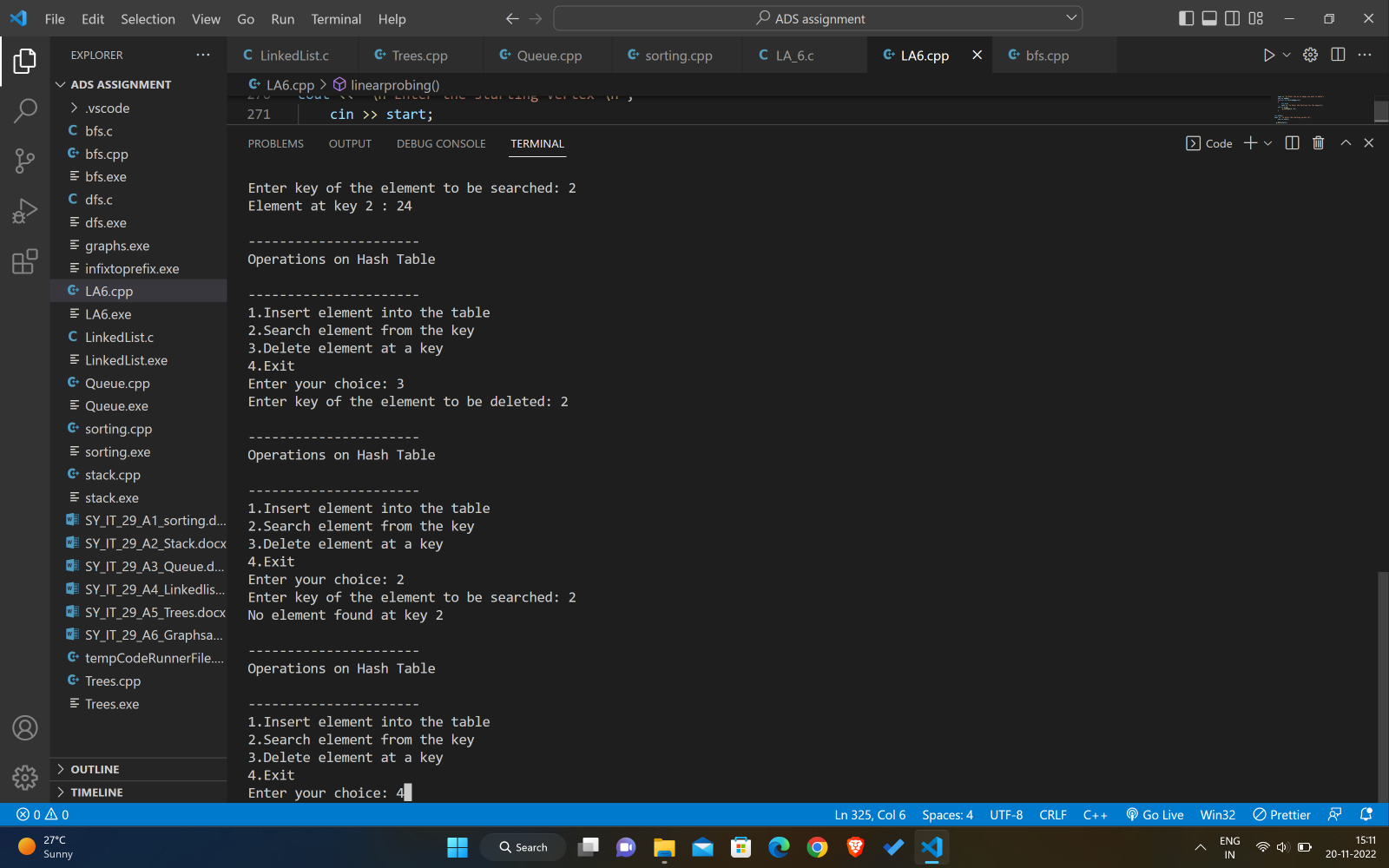
    return 0;

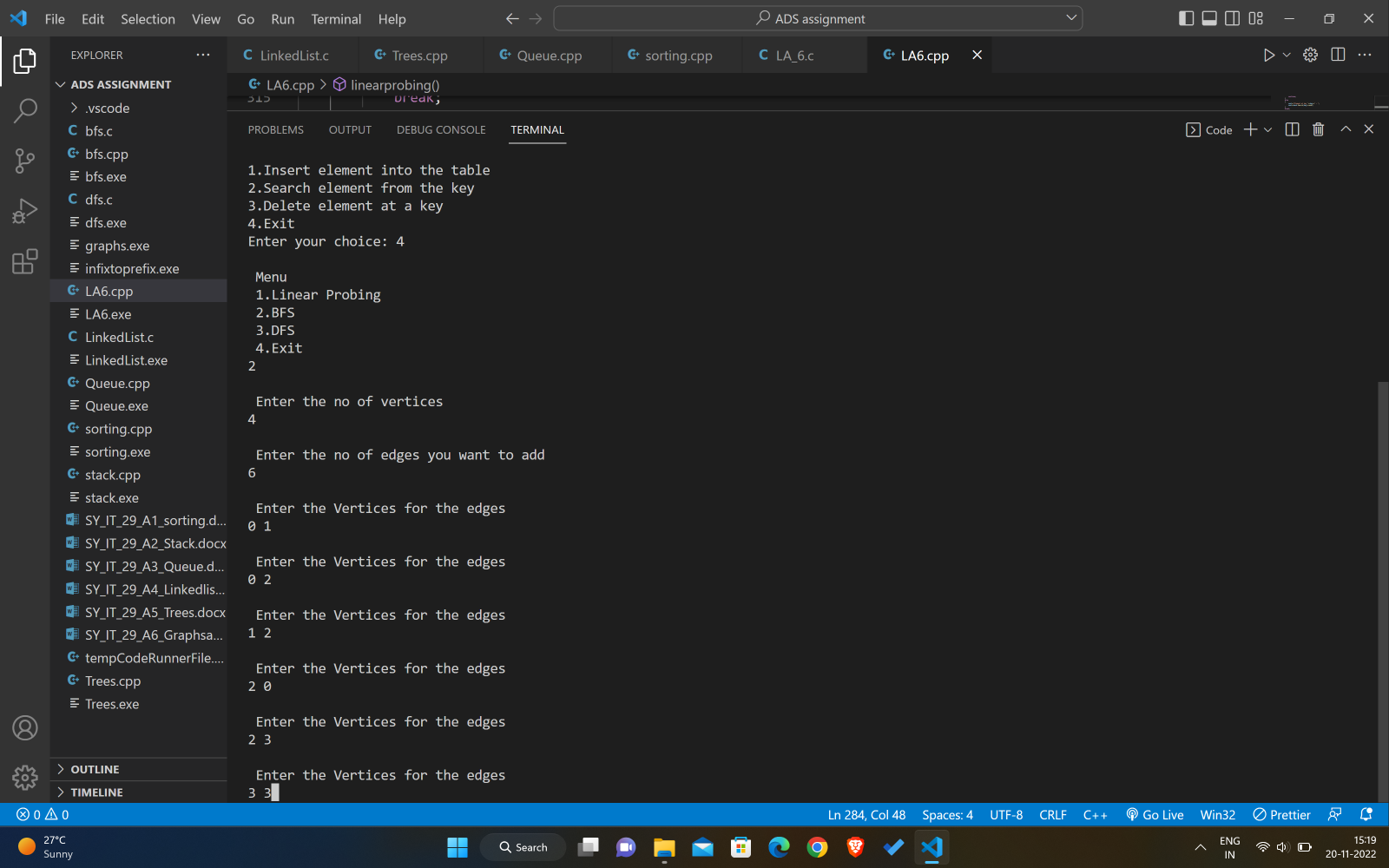
}

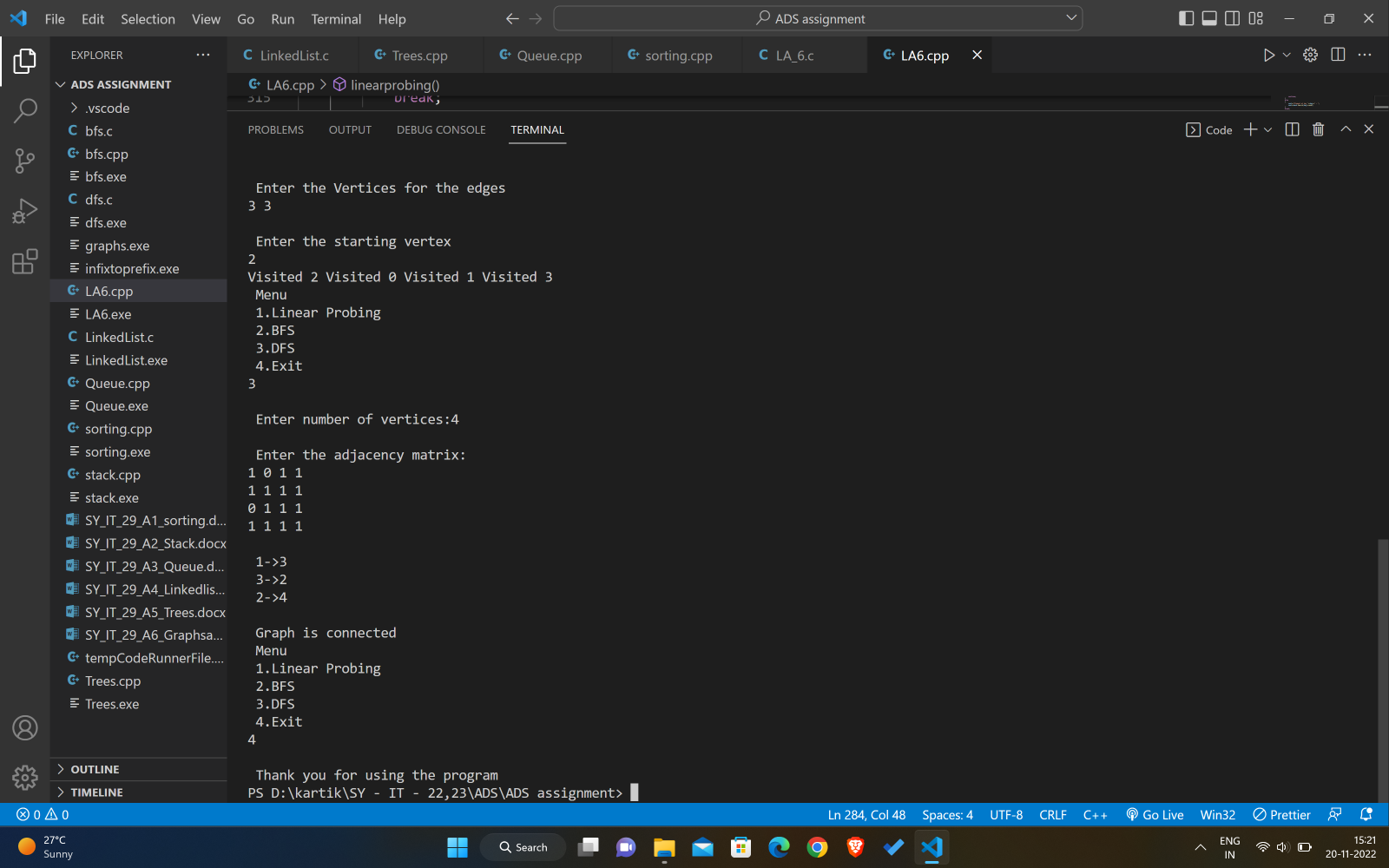
**Input / output:**

****

****

****

****

****