Q1: create a simple tree and singly link list of five nodes with the data in it let's say you are trying to insert the name of the persons in both the tree and singly link list. Your task is to build an algorithm that extract the common element from both the data structure. When you are done with extracting the common elements put all these elements in the stack by using templates. You are required to show the output of each step in proper manner. Furthermore, perform the following operations in your program which are the core of this problem statement

1) Draw diagrams of tree and singly link list in pages. The diagrams should be same as that of your input which you are giving to your diagram. The diagrams must be pasted in your solution.
2) Give us your opinion that, is it good to use the templates in this program.
3) Extract the number of comparison while comparing the elements from tree to singly link list and from singly link list to tree and give us your analysis that which data structure is good to store data. (15)

Solution:

Code:

ListNode.h header file:

```
#pragma once
#include <iostream>

using namespace std;

class ListNode
{
public:
	string data;
	ListNode* next;

	ListNode();
};
```

List.h header file:

```
#pragma once
#include <iostream>
#include "ListNode.h"

using namespace std;
```

```
class List
{
public:
	ListNode* head;

	List();
	~List();
	void insertNode();

};
```

BinaryTree.h header file:

```
#pragma once
#include <iostream>

using namespace std;

class BinaryTree
{
public:
	string* namesArray;
	int nodes;
	int count;

	BinaryTree(int val);
	void insertNode();

};
```

Source.cpp file:

```
#pragma once
#include <iostream>
#include <stack>
#include <string>
#include "ListNode.h"
#include "List.h"
#include "BinaryTree.h"

using namespace std;

//ListNode constructor
ListNode::ListNode() : data(""), next(NULL)
{}

//Destructor for List
```

```
List::~List()
{
	if (head != NULL)
	{
		while (head != NULL)
		{
			ListNode* temp = head;
			head = head->next;
			delete temp;
		}
	}
}


//List constructor
List::List() : head(NULL)
{}

//Insert function for list
void List::insertNode()
{
	ListNode* newNode = new ListNode;
	ListNode* temp = head;

	cout << "\nEnter name: ";
	cin.ignore();
	getline(cin, newNode->data);

	if (head == NULL)
	{
		head = newNode;
	}
	else
	{
		newNode->next = head;			//insert begin functionality
		head = newNode;
	}

	cout << "\nNode inserted.\n";
}



//BinaryTree constructor
BinaryTree::BinaryTree(int val) : nodes(val), count(0)
{
	namesArray = new string[val];
}

//Insert function for binary tree
void BinaryTree::insertNode()
{
	string newData;

	if (count < nodes)
	{
		cout << "\nEnter name: ";
		cin.ignore();
```

```
                getline(cin, newData);

                namesArray[count++] = newData;
                cout << "\nNode inserted.\n";
        }
        else
        {
                cout << "\nSorry, the tree is full.\n";
        }
}



//comparing data structures and finding common values
void extractCommon(ListNode* h, BinaryTree t)
{

        stack<string> common;

        int totalComparison = 0;
        ListNode* listHead = h;

        if (h == NULL)
        {
                cout << "\nSorry, list is empty.\n";
                return;
        }
        if (t.count == 0)
        {
                cout << "\nSorry, tree is empty.\n";
                return;
        }


        while(listHead != NULL)
        {
                for (int i = 0; i < t.count; ++i)
                {
                        ++totalComparison;
                        if (listHead->data == t.namesArray[i])
                        {
                                common.push(listHead->data);
                                break;
                        }
                }
                listHead = listHead->next;
        }

        listHead = h;
        cout << "\nElements of list: \n";

        while (listHead != NULL)
        {
                cout << "\n"<< listHead->data <<"\n";
                listHead = listHead->next;
        }
```

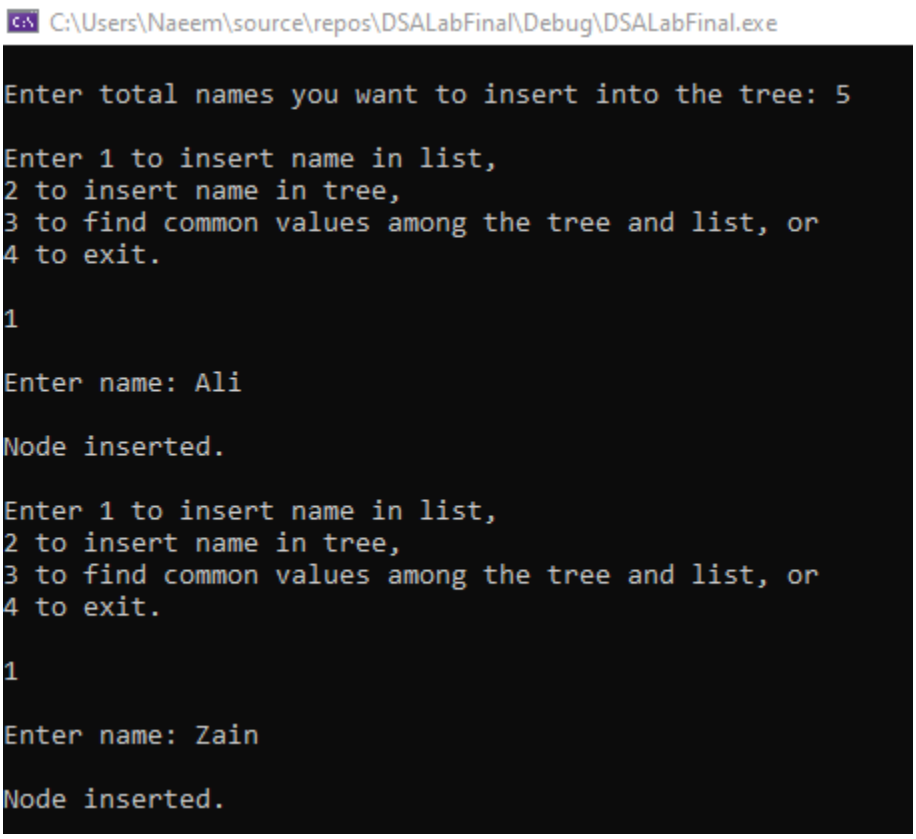
```
	cout << "\n\n\nElements of tree: \n";
	for (int i = 0; i < t.count; ++i)
	{
		cout << "\n" << t.namesArray[i] << "\n";
	}

	cout << "\n\nCommon elements (stored in stack): \n";
	while (!common.empty())
	{
		cout << "\n" << common.top() << "\n";
		common.pop();
	}

	cout << "\nTotal comparisons: " << totalComparison;
}
```

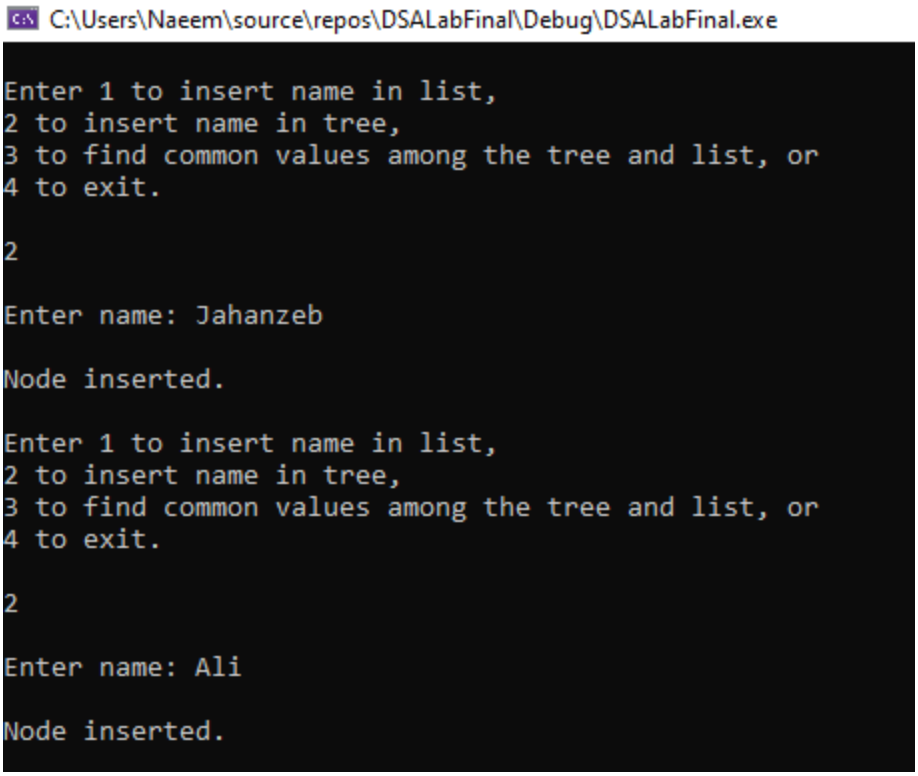
Main.cpp file:

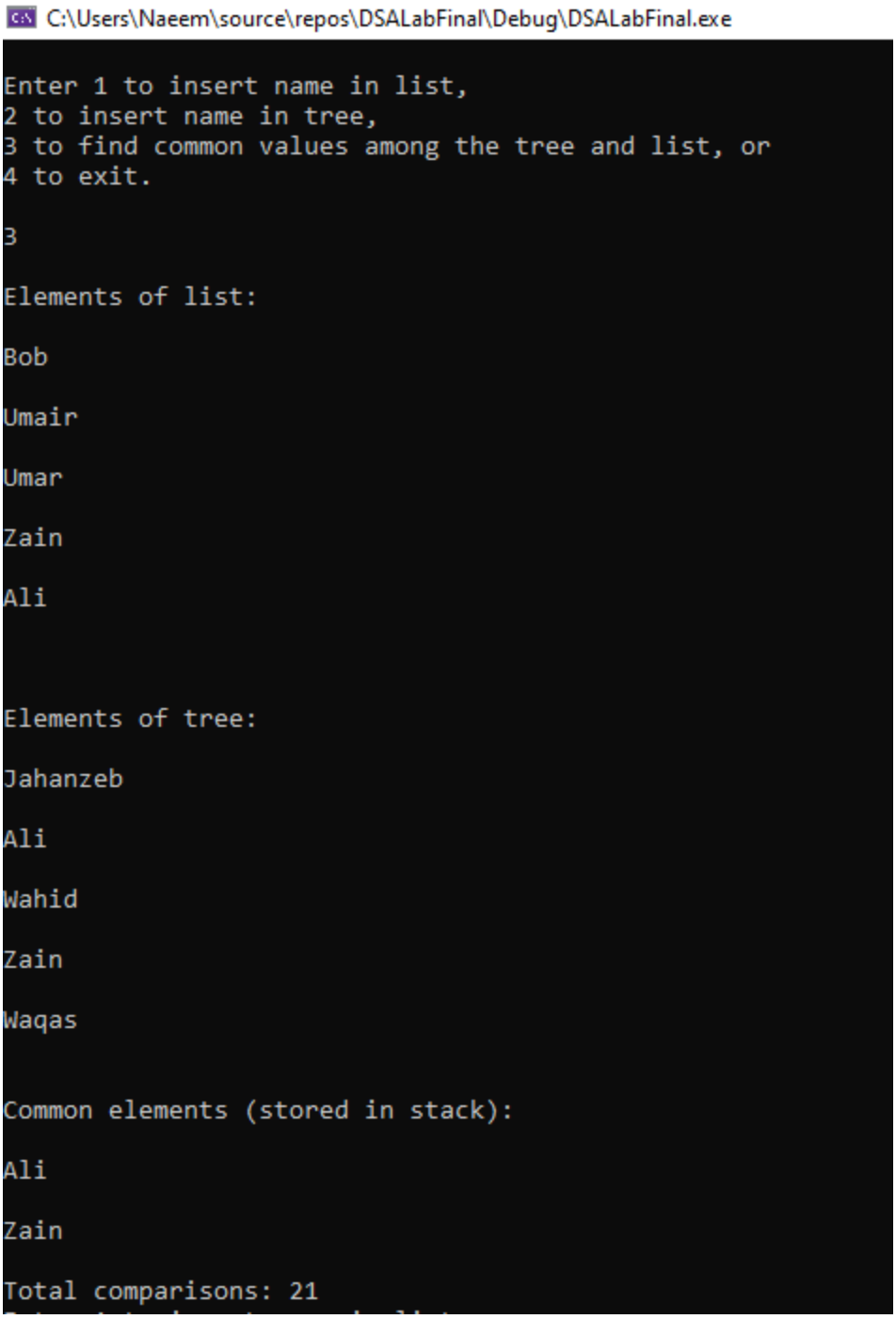
```
#pragma once
#include <iostream>
#include "List.h"
#include "BinaryTree.h"

using namespace std;

void extractCommon(ListNode* h, BinaryTree t);


int main()
{
	int option = 0;
	int value = 0;
	cout << "\nEnter total names you want to insert into the tree: ";
	cin >> value;

	BinaryTree BT(value);
	List SL;

	while (1)
	{
		cout << "\nEnter 1 to insert name in list, "
			<< "\n2 to insert name in tree, "
			<< "\n3 to find common values among the tree and list, or"
			<< "\n4 to exit.\n\n";
		cin >> option;

		if (option == 1)
		{
			SL.insertNode();
		}
		else if (option == 2)
		{
			BT.insertNode();
```

```
		}
		else if (option == 3)
		{
			::extractCommon(SL.head, BT);
		}
		else
		{
			break;
		}
	}

}
```

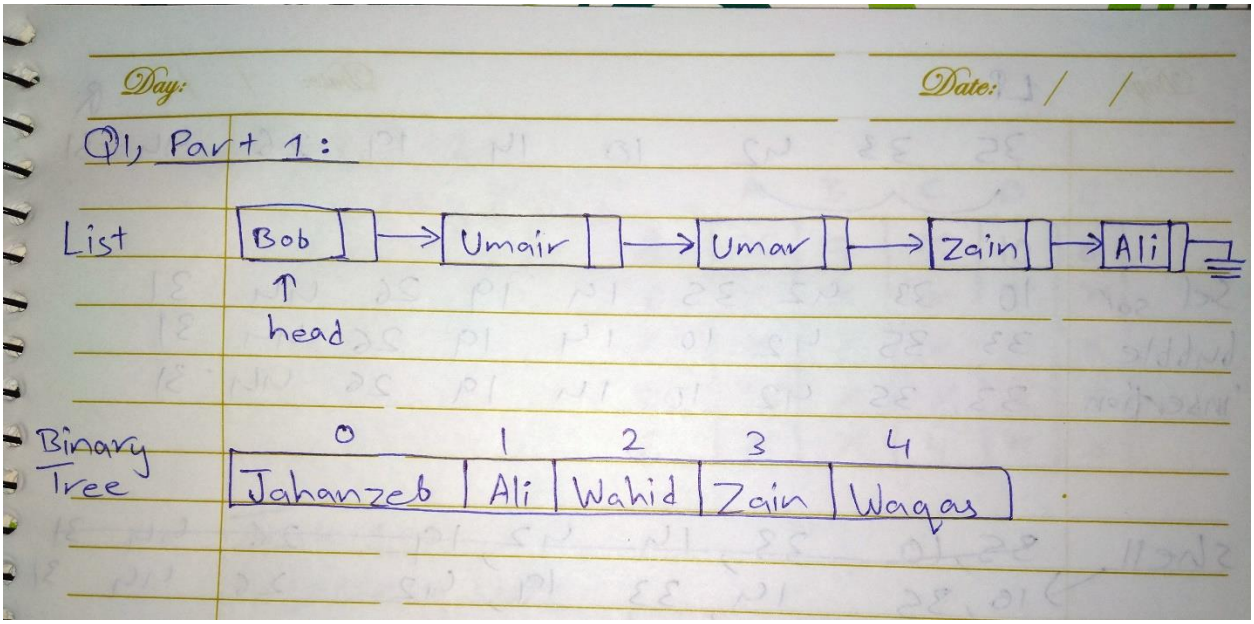
Screenshots:




C:\Users\Naeem\source\repos\DSALabFinal\Debug\DSALabFinal.exe
Enter 1 to insert name in list,
2 to insert name in tree,
3 to find common values among the tree and list, or
4 to exit.
2
Enter name: Jahanzeb
Node inserted.
Enter 1 to insert name in list,
2 to insert name in tree,
3 to find common values among the tree and list, or
4 to exit.
2
Enter name: Ali
Node inserted.


C:\Users\Naeem\source\repos\DSALabFinal\Debug\DSALabFinal.exe
Enter 1 to insert name in list,
2 to insert name in tree,
3 to find common values among the tree and list, or
4 to exit.
3
Elements of list:
Bob
Umair
Umar
Zain
Ali
Elements of tree:
Jahanzeb
Ali
Wahid
Zain
Waqas
Common elements (stored in stack):
Ali
Zain
Total comparisons: 21

Q1 Part 1:



Q1 Part 2:

As we are working with strings in this program and only want to use the push, pop and empty functions of the stack, it is okay to use template. In fact, it could be argued that it is better to use template in this case.

But, if we wanted to add more functionality to Stack class or modify its functions, in that case it would make sense to start from scratch instead of using templates.

Q1 Part 3:

In the code pasted above, we were comparing **each list node to every node in the tree**. And, as it can be seen in the screenshot about, the **total comparisons were 21**.

In the following code we are comparing **each tree node to every node in the list** and it can be seen that **the total comparisons turned out to be 24**.

Hence, it is my opinion that lists are better for storing data (only storing!). But, the choice depends on the situation more than anything else.

```
for (int i = 0; i < t.count; ++i)
{
	listHead = h;
	while (listHead != NULL)
	{
		++totalComparison;
		if (listHead->data == t.namesArray[i])
		{
			common.push(listHead->data);
			break;
		}
		listHead = listHead->next;
	}
}
```

```
Enter 1 to insert name in list,
2 to insert name in tree,
3 to find common values among the tree and list, or
4 to exit.

3

Elements of list:

Bob

Umair

Umar

Zain

Ali


Elements of tree:

Jahanzeb

Ali

Zain

Wahid

Waqas


Common elements (stored in stack):

Zain

Ali

Total comparisons (tree to list): 24
```

Q2: Implement chain Hashing using List. Your hash table must be able to store names of students in Lists. Write insert, delete and search functions. Your program must also be able to print the list with maximum number of nodes. Print separate outputs of all 4 functions. Minus one mark for each output missing. Last function has the maximum marks. (10)

Solution:

Code:

ListNode.h header file:

```
#pragma once
#include <iostream>

using namespace std;

class ListNode
{
public:
	string data;
	ListNode* next;

	ListNode();
};
```

HashTable.h header file:

```
#pragma once
#include <iostream>
#include "ListNode.h"

using namespace std;

class HashTable
{
public:
	int size;
	ListNode** table;

	HashTable(int cap);
	int hashKey(string nm);
	void insertKey(string nm);
```

```
        void deleteKey(string nm);
        bool searchKey(string nm);
        void printLongestList();
};
```

Source.cpp file:

```
#pragma once
#include <iostream>
#include <string>
#include "ListNode.h"
#include "HashTable.h"

using namespace std;


ListNode::ListNode() : data(""), next(NULL)
{}


//constructor for hash table
HashTable::HashTable(int cap) : size(cap)
{
        table = new ListNode* [cap];

        for (int i = 0; i < cap; ++i)
        {
                table[i] = NULL;
        }
}

//function to find hash value
int HashTable::hashKey(string nm)
{
        int sum = 0;
        for (int i = 0; i < nm.length(); ++i)
        {
                sum += (int)nm[i];
        }
        return (sum % size);
}

//insert function
void HashTable::insertKey(string nm)
{
        ListNode* newNode = new ListNode;
        newNode->data = nm;

        int index = hashKey(nm);

        if (table[index] == NULL)
        {
                table[index] = newNode;
```

```
	}
	else                                                    //insert begin
functionality
	{
		newNode->next = table[index];
		table[index] = newNode;
	}
	cout << "\nName inserted.\n";
}

//delete function
void HashTable::deleteKey(string nm)
{
	int index = hashKey(nm);

	ListNode* temp = table[index];

	if (temp == NULL)
	//empty list
	{
		cout << "\nSpecified name doesn't exist.\n";
	}
	else if (temp->data == nm)                                          //deleting
first node
	{
		table[index] = temp->next;
		delete temp;
		cout << "\nName deleted.\n";
	}
	else
	//deleting node from middle or end
	{
		ListNode* prev = NULL;

		while (temp != NULL)
		{
			if (temp->data == nm)
			{
				break;
			}
			prev = temp;
			temp = temp->next;
		}

		if (temp == NULL)
		{
			cout << "\nSpecified name doesn't exist.\n";
		}
		else
		{
			prev->next = temp->next;
			delete temp;
			cout << "\nName deleted.\n";
		}
	}
}

//search function
```

```
bool HashTable::searchKey(string nm)
{
	bool check = false;
	int index = hashKey(nm);

	ListNode* temp = table[index];

	while (temp != NULL)
	{
		if (temp->data == nm)
		{
			check = true;
			break;
		}
		temp = temp->next;
	}

	return check;
}

void HashTable::printLongestList()
{
	int longestIndex = 0;
	int longestListNodes = 0;


	for (int i = 0; i < size; ++i)                                  //going through
entire table to find longest list
	{
		int nodeCount = 0;
		ListNode* temp = table[i];

		if (temp != NULL)                                                  //counting
nodes
		{
			while (temp != NULL)
			{
				++nodeCount;
				temp = temp->next;
			}
		}

		if (nodeCount > longestListNodes)
		{
			longestListNodes = nodeCount;
			longestIndex = i;
		}
	}

	cout << "\nContents of longest list in table: \n";
	ListNode* curr = table[longestIndex];

	while (curr != NULL)
	{
		cout << curr->data << endl;
		curr = curr->next;
	}
}
```

Main.cpp file:

```
#pragma once
#include <iostream>
#include <string>
#include "HashTable.h"

using namespace std;


int main()
{
	int option = 0;

	HashTable table(3);


	while (1)
	{
		cout << "\nEnter 1 to insert name in table, "
			<< "\n2 to delete name front table, "
			<< "\n3 to search name in table, "
			<< "\n4 to print the content of the longest list, or"
			<< "\n5 to exit.\n\n";
		cin >> option;

		if (option == 1)
		{
			string temp;
			cout << "\nEnter name: ";
			cin.ignore();
			getline(cin, temp);
			table.insertKey(temp);
		}
		else if (option == 2)
		{
			string temp;
			cout << "\nEnter name: ";
			cin.ignore();
			getline(cin, temp);
			table.deleteKey(temp);
		}
		else if (option == 3)
		{
			string temp;
			cout << "\nEnter name: ";
			cin.ignore();
			getline(cin, temp);
			bool check = table.searchKey(temp);
```

```
                        if (check == true)
                        {
                                cout << "\nEntered name exists in table.\n";
                        }
                        else
                        {
                                cout << "\nEntered name does not exist in table.\n";
                        }
                }
                else if (option == 4)
                {
                        table.printLongestList();
                }
                else
                {
                        break;
                }
        }

}
```

Screenshots:

C:\Users\Naeem\source\repos\DSALabFinal\Debug\DSALabFinal.exe

```
Enter 1 to insert name in table,
2 to delete name front table,
3 to search name in table,
4 to print the content of the longest list, or
5 to exit.

1

Enter name: jahanzeb

Name inserted.

Enter 1 to insert name in table,
2 to delete name front table,
3 to search name in table,
4 to print the content of the longest list, or
5 to exit.

1

Enter name: naeem

Name inserted.

Enter 1 to insert name in table,
2 to delete name front table,
3 to search name in table,
4 to print the content of the longest list, or
5 to exit.

1

Enter name: akhtar

Name inserted.
```

```
Enter 1 to insert name in table,
2 to delete name front table,
3 to search name in table,
4 to print the content of the longest list, or
5 to exit.

2

Enter name: naeem

Name deleted.
```

C:\Users\Naeem\source\repos\DSALabFinal\Debug\DSALabFinal.exe

```
Enter 1 to insert name in table,
2 to delete name front table,
3 to search name in table,
4 to print the content of the longest list, or
5 to exit.

3

Enter name: zain

Entered name exists in table.
```

C:\Users\Naeem\source\repos\DSALabFinal\Debug\DSALabFinal.exe

```
Enter 1 to insert name in table,
2 to delete name front table,
3 to search name in table,
4 to print the content of the longest list, or
5 to exit.

4

Contents of longest list in table:
shabbir
baloch
asif
zain
akhtar
naeem
```

Q3: Perform an analysis of all the data structure we have discussed in our entire semester and give us an conclusion which data structure is good to build the software and why? Your analysis should be brief and understandable. (5)

Solution:

Lists: with lists, we can keep on adding nodes in the end and this allows user to add as much data as they want to. The problem with lists is that they are linear and hence, functions like InsertEnd, Delete, Edit and Search require traversal of the list. Having to go through the list repeatedly can slow down things and make the program inefficient; it can also get a little tricky to deal with its code.

Stack: stacks are also a form of list but the difference is that they rely on FILO and hence, we can only add and remove values from one end. Stack can be useful in situations where backtracking is required, for example when clicking the "undo" button in MS Word or going back in a web browser. However, due to its structure, stacks can't be used for anything other than backtracking and that makes their functionality limited.

Queues: queues are also a form of list but the special thing about queues is that we keep a track of first as well as the last node. Tracking both ends of queues makes some functions easier but functions like Edit and Search will still require traversal and that can limit this data structure's functionality. Queue relies on FIFO and allows removal only from one end and insertion only on one end.

Trees: being a non-linear data structure, trees allow users to quickly go through the data and find what they are looking for. Due to its structure, data can also be organized during insertion. However, the problem with trees is that due to its structures, recursion has to be used in many functions. Recursion not only makes the code more complex but it also makes the program a little inefficient.

Graphs: graphs are one of the most useful data structures being used today. What makes graphs so interesting is that they can help us keep track of all the connections between different nodes and thanks to adjacency matrix; we can also keep track of the weights of their connections/edges.

Which kind of data structure is best for software development depends on what kind of software we are building.

However, in case we do not know anything about the software but we must choose a data structure then in that case, I would choose **linked lists** because their simple structure allows a lot of freedom to modify the structure and increase functionality.