**VISUAL PROGRAMMING LAB 07**

**WORKBOOK**

**Instructor**

**Md. Rashedul Islam**

**SUBMIT WITHIN: 12:45 PM**

**SEND PROGRAMS IMAGES TO**

**rashed.class.official17@gmail.com**

**Email Subject: LAB07\_CSC440\_ID\_SUMMER2022**

**C# program to implement stack using array**

A stack is a linear data structure that follows the principle of Last In First Out (LIFO). This means the last element inserted inside the stack is removed first.

You can think of the stack data structure as the pile of plates on top of another.



Stack representation similar to a pile of plate

Here, you can:

Put a new plate on top

Remove the top plate

And, if you want the plate at the bottom, you must first remove all the plates on top. This is exactly how the stack data structure works.

LIFO Principle of Stack

In programming terms, putting an item on top of the stack is called push and removing an item is called pop.



Stack Push and Pop Operations

In the above image, although item 2 was kept last, it was removed first. This is exactly how the LIFO (Last In First Out) Principle works.

In stack we use array to store elements, and a pointer top, that points top most element in stack.

using System;

namespace LAB07

{

class Stack

{

private int[] ele;

private int top;

private int max;

public Stack(int size)

{

ele = new int[size];

top = -1;

max = size;

}

public void push(int item)

{

if (top == max - 1)

{

Console.WriteLine("Stack Overflow");

return;

}

else

{

ele[++top] = item;

}

}

public int pop()

{

if (top == -1)

{

Console.WriteLine("Stack Underflow");

return -1;

}

else

{

Console.WriteLine("Poped element is: " + ele[top]);

return ele[top--];

}

}

public void printStack()

{

if (top == -1)

{

Console.WriteLine("Stack is Empty");

return;

}

else

{

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

{

Console.WriteLine("Item[" + (i + 1) + "]: " + ele[i]);

}

}

}

}

class Program

{

static void Main(string[] args)

{

Stack S = new Stack(5);

S.push(10);

S.push(20);

S.push(30);

S.push(40);

S.push(50);

Console.WriteLine("Items are : ");

S.printStack();

S.pop();

S.pop();

S.pop();

}

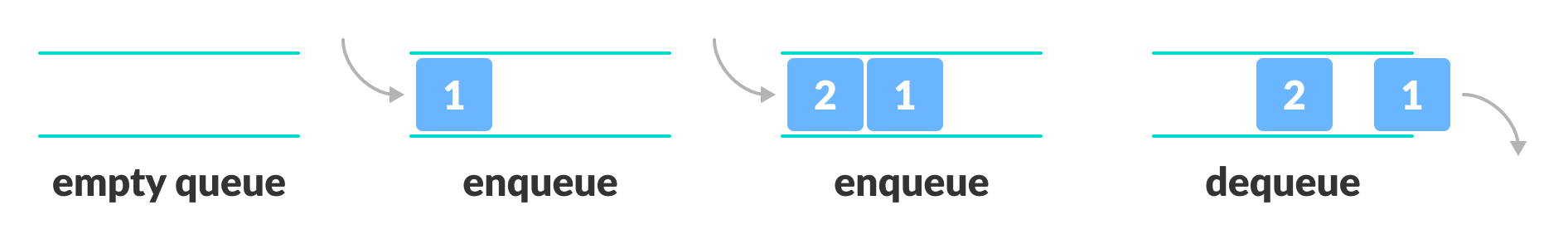
}

}

**C# program to implement linear queue using array**

A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

Queue follows the First In First Out (FIFO) rule - the item that goes in first is the item that comes out first.

FIFO Representation of Queue

In the above image, since 1 was kept in the queue before 2, it is the first to be removed from the queue as well. It follows the FIFO rule.

In programming terms, putting items in the queue is called enqueue, and removing items from the queue is called dequeue.

using System;

namespace LAB07

{

class LinearQueue

{

private int[] ele;

private int front;

private int rear;

private int max;

public LinearQueue(int size)

{

ele = new int[size];

front = 0;

rear = -1;

max = size;

}

public void insert(int item)

{

if (rear == max - 1)

{

Console.WriteLine("Queue Overflow");

return;

}

else

{

ele[++rear] = item;

}

}

public int delete()

{

if (front == rear + 1)

{

Console.WriteLine("Queue is Empty");

return -1;

}

else

{

Console.WriteLine("deleted element is: " + ele[front]);

return ele[front++];

}

}

public void printQueue()

{

if (front == rear + 1)

{

Console.WriteLine("Queue is Empty");

return;

}

else

{

for (int i = front; i <= rear; i++)

{

Console.WriteLine("Item[" + (i + 1) + "]: " + ele[i]);

}

}

}

}

class Program

{

static void Main(string[] args)

{

LinearQueue Q = new LinearQueue(5);

Q.insert(10);

Q.insert(20);

Q.insert(30);

Q.insert(40);

Q.insert(50);

Console.WriteLine("Items are : ");

Q.printQueue();

Q.delete();

Q.delete();

Console.WriteLine("Items are : ");

Q.printQueue();

}

}

}

**C# program to clear all elements from stack**

Stack.Clear() method

This is a method of 'Stack' class, it is used to clear/remove all elements of the stack.

Syntax:

void Clear();

using System;

using System.Collections;

namespace LAB07

{

class Program

{

static void Main(string[] args)

{

Stack S = new Stack(5);

S.Push(10);

S.Push(20);

S.Push(30);

S.Push(40);

S.Clear();

Console.WriteLine("Now stack is empty");

}

}

}

**C# program to traverse the singly linked list**

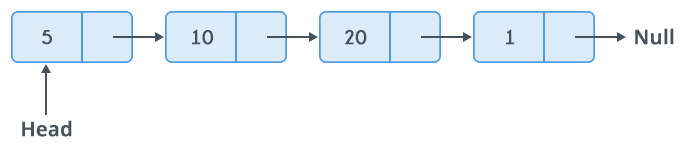
A **linked list** is a way to store a collection of elements. Like an array these can be character or integers. Each element in a linked list is stored in the form of a **node**.

**Node**:



A node is a collection of two sub-elements or parts. A **data** part that stores the element and a **next** part that stores the link to the next node.

**Linked List**:



A linked list is formed when many such nodes are linked together to form a chain. Each node points to the next node present in the order. The first node is always used as a reference to traverse the list and is called **HEAD**. The last node points to **NULL**.

using System;

using System.Collections;

namespace LAB07

{

class ListNode

{

private int item;

private ListNode next;

public ListNode(int value)

{

item = value;

next = null;

}

public ListNode AddItem(int value)

{

ListNode node = new ListNode(value);

if (this.next == null)

{

node.next = null;

this.next = node;

}

else

{

ListNode temp = this.next;

node.next = temp;

this.next = node;

}

return node;

}

public void ListTraverse()

{

ListNode node = this;

while (node != null)

{

Console.WriteLine("-->" + node.item);

node = node.next;

}

}

}

class Program

{

static void Main(string[] args)

{

ListNode StartNode = new ListNode(201);

ListNode n1;

ListNode n2;

ListNode n3;

ListNode n4;

n1 = StartNode.AddItem(202);

n2 = n1.AddItem(203);

n3 = n2.AddItem(204);

n4 = n3.AddItem(205);

Console.WriteLine("Traversing of Linked list:");

StartNode.ListTraverse();

}

}

}

**C# program to implement In-order traversal in Binary Tree**

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.



We start from A, and following in-order traversal, we move to its left subtree B. B is also traversed in-order. The process goes on until all the nodes are visited. The output of in-order traversal of this tree will be −

D → B → E → A → F → C → G

**Algorithm**

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

using System;

using System.Collections;

namespace LAB07

{

class Node

{

public int item;

public Node left\_ptr;

public Node right\_ptr;

}

class BinaryTree

{

public Node root;

public Node GetRoot()

{

return root;

}

public void Inorder\_Traverse(Node rootNode)

{

if (rootNode != null)

{

Inorder\_Traverse(rootNode.left\_ptr);

Console.Write("{0} ", rootNode.item);

Inorder\_Traverse(rootNode.right\_ptr);

}

}

public void InsertItem(int item)

{

Node curNode;

Node parentNode;

Node node = new Node();

node.item = item;

if (root != null)

{

curNode = root;

while (true)

{

parentNode = curNode;

if (item < curNode.item)

{

curNode = curNode.left\_ptr;

if (curNode == null)

{

parentNode.left\_ptr = node;

break;

}

}

else

{

curNode = curNode.right\_ptr;

if (curNode == null)

{

parentNode.right\_ptr = node;

break;

}

}

}

}

else

{

root = node;

}

}

}

class Program

{

static void Main(string[] args)

{

BinaryTree tree = new BinaryTree();

tree.InsertItem(10);

tree.InsertItem(15);

tree.InsertItem(35);

tree.InsertItem(26);

tree.InsertItem(47);

tree.InsertItem(34);

tree.InsertItem(90);

Console.WriteLine("Inorder Traversal : ");

tree.Inorder\_Traverse(tree.GetRoot());

Console.WriteLine(" ");

}

}

}

**C# program to implement selection Sort**

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of Ο(n2), where n is the number of items.

How Selection Sort Works?

Consider the following depicted array as an example.

Unsorted Array

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

Selection Sort

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

Selection Sort

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

Selection Sort

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

Selection Sort

After two iterations, two least values are positioned at the beginning in a sorted manner.

Selection Sort

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −

Now, let us learn some programming aspects of selection sort.

Algorithm

**Step 1** − Set MIN to location 0

**Step 2** − Search the minimum element in the list

**Step 3** − Swap with value at location MIN

**Step 4** − Increment MIN to point to next element

**Step 5** − Repeat until list is sorted

using System;

using System.Collections;

namespace LAB07

{

class Program

{

static void Main(string[] args)

{

int[] intArry = new int[5] { 65, 34, 23, 76, 21 };

Console.WriteLine("Array before sorting: ");

for (int i = 0; i < intArry.Length; i++)

{

Console.Write(intArry[i] + " ");

}

Console.WriteLine();

SelectionSort(ref intArry);

Console.WriteLine("Array before sorting: ");

for (int i = 0; i < intArry.Length; i++)

{

Console.Write(intArry[i] + " ");

}

Console.WriteLine();

}

static void SelectionSort(ref int[] intArr)

{

int temp = 0;

int min = 0;

int i = 0;

int j = 0;

for (i = 0; i < intArr.Length - 1; i++)

{

min = i;

for (j = i + 1; j < intArr.Length; j++)

{

if (intArr[j] < intArr[min])

{

min = j;

}

}

temp = intArr[min];

intArr[min] = intArr[i];

intArr[i] = temp;

}

}

}

}