**Assignment 3**

21AIE111

Data Structure and Algorithms – SEM-II

Professor – Dr. Sachin Sir

Submitted By: Vikhyat Bansal [CB.EN.U4AIE21076]



1. Write java code to print Fibonacci series, upto 10 positions, without recursion

(Fibonacci series: 0 1 1 2 3 5 8 13…).

CODE:

Method 1

// *Display Fibonacci series without recursion*

import *java*.*util*.*Scanner*;

*public* *class* Fibonacci

{

*public* *static* void *main*(String[] args)

    {

        Scanner input = *new* *Scanner*(System.*in*);

        System.*out*.*print*("Enter the number of terms in the Fibonacci series: ");

        int n = input.*nextInt*();

        int f1 = 0;

        int f2 = 1;

        int f3 = 0;

        System.*out*.*print*(f1 + " " + f2 + " ");

// *A program is called non recursive when it contains a loop*

*for* (int i = 2; i < n; i++) {

            f3 = f1 + f2;

            f1 = f2;

            f2 = f3;

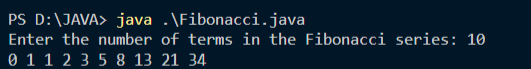
            System.*out*.*print*(f3 + " ");

        }

    }

}

OUPUT:



Method 2

// *Display Fibonacci series without recursion*

*public* *class* Fibonacci {

*public* *static* void *main*(String[] args) {

        int n = 10;

        int f1 = 0;

        int f2 = 1;

        int f3 = 0;

        System.*out*.*print*(f1 + " " + f2 + " ");

// *A program is called non recursive when it contains a loop*

*for* (int i = 2; i < n; i++) {

            f3 = f1 + f2;

            f1 = f2;

            f2 = f3;

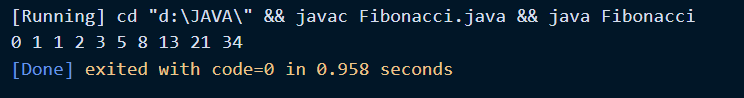
            System.*out*.*print*(f3 + " ");

        }

    }

}

OUTPUT:

Explanation:

* Fibonacci series is a series of numbers where the next number is the sum of previous two numbers starting with 0 and 1.
* We have taken two initial values f1 and f2 which will start the Fibonacci series and values in f1 and f2 initially will be printed to initialize the series and then f3 will store the output (i.e., sum of f1 and f2) in form of loop where (i<n).
* We have created a code in such a way that takes input in form of number of elements(n) in method 1.
* We have taken n = 10 as an input to number of elements in method 2.

2. Write java code to print Fibonacci series, upto 10 positions, with recursion.

CODE:

Method 1:

*public* *class* Question2 {

    int a=0, b =1, c =0;

    void *printFibonacci*(int size,int d){

*if*(size>0){

    /\* *If d == 0 (the Fibonacci series is called for the first time)*

*Do size = size - 2* \*/

*if*(d == 0){

    size = size - 2;

    // *To print 0 and 1*

    System.*out*.*print*(a+ " " + b);

    }

    c = a + b;

    a = b;

    b = c;

    d = 1;

    System.*out*.*print*(" "+ c);

*printFibonacci*(size-1,c);

    }

    }

*public* *static* void *main*(String[] args){

    Question2 Withrecursion = *new* *Question2*();

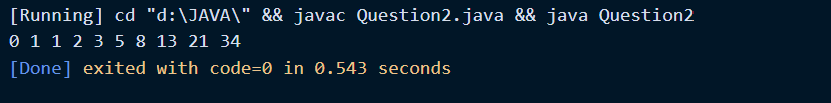
    int count=10;

    Withrecursion.*printFibonacci*(count, 0);

    }

    }

OUTPUT:



Explanation:

* The concept is same as the iterative approach but instead the function calls itself until size becomes 0.
* d is called to add the first two elements of Fibonacci series i.e., 0 & 1 and reduce the number of elements by 2 as 0 & 1 are already added.

Method 2:

// *Show elements of Fibonacci series with recursion*

// *Take input as some element in form of n and put if condition in such a way that when n is 0,1 or 2 then it will print null,0 and 1 respectively and when n is greater than 1 then it will print Fibonacci series till nth element.*

import *java*.*util*.*Scanner*;

*public* *class* Question2 {

*public* *static* void *main*(String[] args) {

        Scanner sc = *new* *Scanner*(System.*in*);

        System.*out*.*println*("Enter a number: ");

        int n = sc.*nextInt*();

*if*(n == 0) {

            System.*out*.*println*("null");

        }

*else* *if*(n == 1) {

            System.*out*.*println*(0);

        }

*else* *if*(n == 2) {

            System.*out*.*println*(1);

        }

*else* {

            int a = 0;

            int b = 1;

            int c;

            System.*out*.*print*(a + " " + b + " ");

*for* (int i = 2; i < n; i++) {

                c = a + b;

                a = b;

                b = c;

                System.*out*.*print*(c + " ");

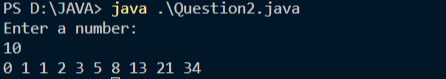
            }

    }

}

}

OUTPUT:



Explanation:

* We have created a code in such a way that takes input in form of number of elements(n) in method 1.
* We created multiple conditional statements which will give us output in form of recursion that is calling the function itself again and again.

3. Write a java code to reverse an array using recursion.

CODE:

// *Reverse an array using recursion*

*public* *class* arrayReverse {

*public* *static* int[] *reverseArray*(int[] t,int i,int j){

*if*(i<j){

            //*swap elements t[i],t[j]*

            //*t[i] = first element of array*

            //*t[j] = last element of array*

            int temp=t[i];

            t[i]=t[j];

            t[j]=temp;

*reverseArray*(t, i+1, j-1);

        }

*return* t;

    }

*public* *static* void *main*(String[] args) {

        //*input    : arr={0,1,2,3,4,5}*

        //*output   : arr={5,4,3,2,1,0}*

        int[] arr={0,1,2,3,4,5};

        int length=arr.*length*;

        //*length-1 as indexing starts from 0*

        int[] revArray=*reverseArray*(arr,0,length-1);

        //*loop through array for display.*

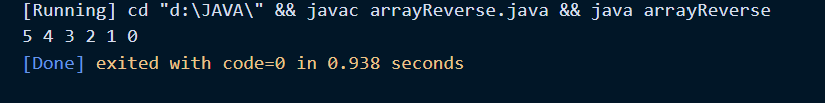
*for*(int i*:*revArray)

            System.*out*.*print*(i+" ");

    }

}

OUTPUT:



Explanation:

* We have created a method named reverseArray with three input variables.
* If conditional statement is created which will swap elements if t[i] != t[j],  
  elements will not be swapped if t[i] = t[j].
* Finally, we will increase ith index and decrease jth index by one as we move ahead and will return our array t.
* We create a main class in which we will give an input of which we get output.

4. Write a java code to merge two single linked list. Consider the linked list with atleast 5 nodes.

CODE: Using String

*public* *class* MergeSLL {

    Node head;

    Node tail;

*static* *class* Node{

    String data;

    Node next;

    // *Constructor is used to create new nodes*

*Node*(String d){

    data = d;

    next = null;

    }

    }

    // *To display the output to terminal*

*public* void *DisplayList*(){

    // *Start from head*

    Node node = head;

    /\* *While loop to traverse through the linked list.*

*It will stop execution once the node is null.*

\*/

*while*(node!=null){

    // *node.data is used to access the data inside node*

    System.*out*.*print*(node.*data* + " ");

    // *proceed to next node*

    node = node.*next*;

    }

    }

    // *Method to add nodes to linked list*

    void *add*(String data)

    {

    // *Create a temp node to hold data*

    Node temp = *new* *Node*(data);

    /\* *If head is null, then the temp node is supposed to be the head.*

*The head is only null if the linked list is empty*

\*/

*if*(head == null){

    head = temp;

    }

    /\*

*Append nodes to the tail, than traversing through the whole linked list the adding to final node.*

*Appending by traversal will result in o(n) time complexity.*

\*/

*else* *if*(tail != null){

    tail.*next* = temp;

    tail = tail.*next*;

    }

*else* {

    tail = head.*next* = temp;

    }

    }

*public* *static* void *mergeLL*(MergeSLL LList, MergeSLL LList2){

    // *Create node and assign the second Linked List’s head to it.*

    Node node = LList2.*head*;

    /\* *This will add nodes of LList2 to LList1 and exit once*

*All the elements are added*\*/

*while*(node != null){

    // *Add the value from node of 2nd LList to tail of 1st LList*

    LList.*tail*.*next* = node;

    LList.*tail* = LList.*tail*.*next*;

    node = node.*next*;

    }// *Each Linked list contains atleast 5 nodes*

    }

*public* *static* void *main*(String[] args) {

    MergeSLL LList = *new* *MergeSLL*();

    MergeSLL LList2 = *new* *MergeSLL*();

    LList.*add*("hi");

    LList.*add*("ABC");

    LList.*add*("how");

    LList.*add*("are");

    LList.*add*("you?");

    LList2.*add*("What");

    LList2.*add*("is");

    LList2.*add*("your");

    LList2.*add*("name");

    LList2.*add*("??");

*mergeLL*(LList, LList2);

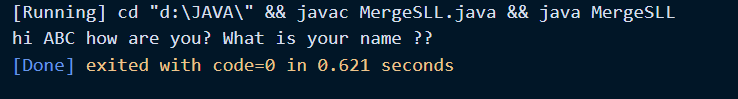
    LList2 = null;

    LList.*DisplayList*();

    }

    }

OUTPUT:



Explanation:

* A constructor is created which will be used to create linked list.
* A method (add) is created which will be used to add data in linked list.
* mergeLL method is used to point tail of linked list 1 to head of linked list 2.
* Using while loop we will append nodes in linked list 1 from linked list 2.
* Finally, we make LList2 data null so the memory is freed and LLists got merged.

CODE: Using integer

*public* *class* MergeSLL {

    Node head;

    Node tail;

*static* *class* Node{

    int data;

    Node next;

    // *Constructor is used to create new nodes*

*Node*(int data2){

    data = data2;

    next = null;

    }

    }

    // *To display the output to terminal*

*public* void *DisplayList*(){

    // *Start from head*

    Node node = head;

    /\* *While loop to traverse through the linked list.*

*It will stop execution once the node is null.*

\*/

*while*(node!=null){

    // *node.data is used to access the data inside node*

    System.*out*.*print*(node.*data* + " ");

    // *proceed to next node*

    node = node.*next*;

    }

    }

    // *Method to add nodes to linked list*

    void *add*(int data)

    {

    // *Create a temp node to hold data*

    Node temp = *new* *Node*(data);

    /\* *If head is null, then the temp node is supposed to be the head.*

*The head is only null if the linked list is empty*

\*/

*if*(head == null){

    head = temp;

    }

    /\*

*Append nodes to the tail, than traversing through the whole linked list then*

*adding to final node.*

*This approach will help to append new nodes with o(1) time complexity.*

*Appending by traversal will result in o(n) time complexity.*

\*/

*else* *if*(tail != null){

    tail.*next* = temp;

    tail = tail.*next*;

    }

*else* {

    tail = head.*next* = temp;

    }

    }

*public* *static* void *mergeLL*(MergeSLL LList, MergeSLL LList2){

    // *Create node and assign the second Linked List’s head to it.*

    Node node = LList2.*head*;

    /\* *This will add nodes of LList2 to LList1 and exit once*

*All the elements are added*\*/

*while*(node != null){

    // *Add the value from node to tail of 1st LList*

    LList.*tail*.*next* = node;

    LList.*tail* = LList.*tail*.*next*;

    node = node.*next*;

    }

    }

*public* *static* void *main*(String[] args) {

    MergeSLL LList = *new* *MergeSLL*();

    MergeSLL LList2 = *new* *MergeSLL*();

    LList.*add*(5);

    LList.*add*(12);

    LList.*add*(69);

    LList.*add*(55);

    LList.*add*(21);

    LList2.*add*(1);

    LList2.*add*(3);

    LList2.*add*(5);

    LList2.*add*(7);

    LList2.*add*(9);

*mergeLL*(LList, LList2);

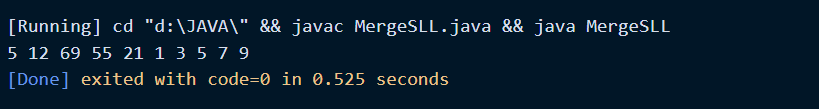
    LList2 = null;

    LList.*DisplayList*();

    }

    }

OUTPUT:



Explanation:

* Same as done with data type string, in this case data type is changed to int.

5. Write a java code to merge two double linked list. Consider the linked list with atleast 5 nodes.

CODE: Using String

// *Merge two doubly linked lists and return the merged list.*

*public* *class* DoublyLinkedList {

    Node head;

    Node tail;

*static* *class* Node{

        String data;

        Node prev;

        Node next;

*Node*(String d){

            data=d;

        }

    }

*public* void *add*(String d){

        Node newNode = *new* *Node*(d);

        newNode.*prev* = null;

        newNode.*next* = head;

*if*(head == null) {

            //*Both head and tail will point to newNode*

            head = tail = newNode;

            //*head's previous will point to null*

            head.*prev* = null;

            //*tail's next will point to null, as it is the last node of the list*

            tail.*next* = null;

        }

        //*Add newNode as new tail of the list*

*else* {

            //*newNode will be added after tail such that tail's next will point to newNode*

            tail.*next* = newNode;

            //*newNode's previous will point to tail*

            newNode.*prev* = tail;

            //*newNode will become new tail*

            tail = newNode;

            //*As it is last node, tail's next will point to null*

            tail.*next* = null;

        }

    }

*public* void *DisplayList*(){

        // *Start from head*

        Node node = head;

        /\* *While loop to traverse through the linked list.*

*It will stop execution once the node is null.*

\*/

*while*(node!=null){

        // *node.data is used to access the data inside node*

        System.*out*.*print*(node.*data* + " ");

        // *proceed to next node*

        node = node.*next*;

        }

        }

*public* *static* void *mergeLL*(DoublyLinkedList LList, DoublyLinkedList LList2){

        Node node  = LList2.*head*;

*while*(node != null){

            LList.*tail*.*next* = node;

            LList.*tail*.*next*.*prev* = LList.*tail*;

            LList.*tail* = LList.*tail*.*next*;

            node = node.*next*;

        }

    }

*public* *static* void *main*(String[] args) {

        DoublyLinkedList LList = *new* *DoublyLinkedList*();

        DoublyLinkedList LList2 = *new* *DoublyLinkedList*();

        LList.*add*("Hi");

        LList.*add*("XYZ");

        LList.*add*("have");

        LList.*add*("you");

        LList.*add*("completed");

        LList2.*add*("D");

        LList2.*add*("S");

        LList2.*add*("A");

        LList2.*add*("assignment");

        LList2.*add*("?");

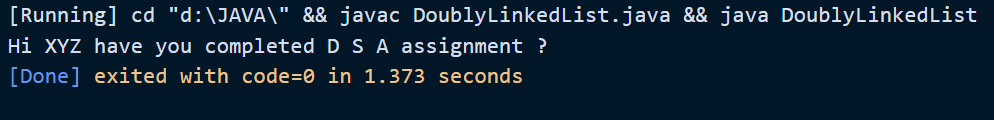
*mergeLL*(LList, LList2);

       LList.*DisplayList*();

    }

}

OUTPUT:



Explanation:

* A constructor is created which will be used to create linked list.
* A method (add) is created which will be used to add data in linked list.
* mergeLL method is used to point next of tail of linked list 1 to head of linked list 2 and previous of head of linked list 2 to tail of linked list 1.
* Using while loop we will append nodes in linked list 1 from linked list 2.

CODE:

Using Integer

// *Merge two doubly linked lists and return the merged list.*

*public* *class* DoublyLinkedList {

    Node head;

    Node tail;

*static* *class* Node{

        int data;

        Node prev;

        Node next;

*Node*(int d){

            data=d;

        }

    }

*public* void *add*(int d){

        Node newNode = *new* *Node*(d);

        newNode.*prev* = null;

        newNode.*next* = head;

*if*(head == null) {

            //*Both head and tail will point to newNode*

            head = tail = newNode;

            //*head's previous will point to null*

            head.*prev* = null;

            //*tail's next will point to null, as it is the last node of the list*

            tail.*next* = null;

        }

        //*Add newNode as new tail of the list*

*else* {

            //*newNode will be added after tail such that tail's next will point to newNode*

            tail.*next* = newNode;

            //*newNode's previous will point to tail*

            newNode.*prev* = tail;

            //*newNode will become new tail*

            tail = newNode;

            //*As it is last node, tail's next will point to null*

            tail.*next* = null;

        }

    }

*public* void *DisplayList*(){

        // *Start from head*

        Node node = head;

*while*(node!=null){

        // *node.data is used to access the data inside node*

        System.*out*.*print*(node.*data* + " ");

        node = node.*next*;

        }

        }

*public* *static* void *mergeLL*(DoublyLinkedList LList, DoublyLinkedList LList2){

        Node node  = LList2.*head*;

*while*(node != null){

            LList.*tail*.*next* = node;

            LList.*tail*.*next*.*prev* = LList.*tail*;

            LList.*tail* = LList.*tail*.*next*;

            node = node.*next*;

        }

    }

*public* *static* void *main*(String[] args) {

        DoublyLinkedList LList = *new* *DoublyLinkedList*();

        DoublyLinkedList LList2 = *new* *DoublyLinkedList*();

        LList.*add*(10);

        LList.*add*(20);

        LList.*add*(30);

        LList.*add*(40);

        LList.*add*(50);

        LList2.*add*(99);

        LList2.*add*(299);

        LList2.*add*(69);

        LList2.*add*(55);

        LList2.*add*(100);

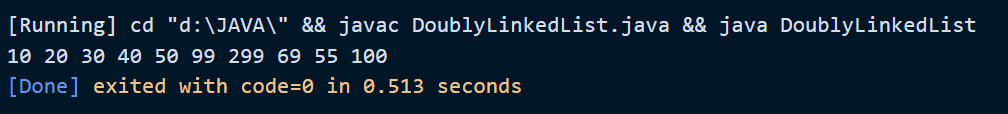
*mergeLL*(LList, LList2);

       LList.*DisplayList*();

    }

}

OUTPUT:



6. Write a java code to arrange the elements of stack in increasing order.

|  |  |
| --- | --- |
| 5 | 3 |
| 4 | 12 |
| 3 | 4 |
| 2 | 10 |
| 1 | 0 |
| 0 | 20 |

First column indicates the index location and second column indicates the value in the stack.

CODE: Using Array

// *Sort the stack elements in increasing order.*

*public* *class* SortedStack {

    // *The stack is created using arrays*

*public* int [] array;

*public* int top;

*public* int length;

    // *Constructor to create stack of dimension provided as argument.*

*SortedStack*(int dim){

    array = *new* int [dim];

    length = dim;

    // *Top is maintained, the default value is -1 (no elements)*

    top = -1;

    }

    // *Method for pushing element to Stack*

*public* void *push*(int data){

    // *Check if Stack is full. If yes, exit*

*if*(*isFull*()){

    System.*out*.*println*("Stack Full");

    System.*exit*(1);

    }

    // *Increment the Stack pointer*

    top = top+1;

    array[top] = data;

    }

    // *Method to pop element from Stack*

*public* int *pop*(){

    // *Check if stack is empty. If yes, exit.*

*if*(*isEmpty*()){

    System.*out*.*println*("Stack Empty");

    System.*exit*(1);

    }

    // *Decrement the stack pointer*

    top = top - 1;

    // *Return the popped value*

*return* array[top+1];

    }

    // *Method to check if the Stack is full*

*public* boolean *isFull*(){

    /\* *If Stack Pointer is = length -1 then the Stack is full*

*and this method will return true.* \*/

*return* top==(length-1);

    }

    // *Method to check if the Stack is full*

*public* boolean *isEmpty*(){

    // *If Stack Pointer is -1, the Stack is empty.*

*return* top==-1;

    }

    // *Method to print the values in Stack*

*public* void *displayStack*(){

    // *Same as traversing an array*

*for*(int i = 0; i < top+1; i++){

    System.*out*.*println*(array[i]);

    }

    }

*public* *static* void *sortStack*(SortedStack stack){

    // *Create an array of stack's length*

    int [] array = *new* int[stack.*length*];

    // *An array index counter*

    int counter = stack.*length*-1;

    // *While stack is not empty, execute this loop*

*while*(!stack.*isEmpty*()){

    // *Pop element from stack and put it into array*

    array[counter] = stack.*pop*();

    counter--;

    }

    // *For loops to sort array*

*for* (int i = 0; i < array.*length*; i++){

*for* (int j = i + 1; j < array.*length*; j++){

    int temp = 0;

*if* (array[i] < array[j]){

    temp = array[i];

    array[i] = array[j];

    array[j] = temp;

    }

    }

    }

    // *Push the elements from sorted array to the stack*

*for*(int i = array.*length*-1; i > -1; i--){

    stack.*push*(array[i]);

    }

    }

*public* *static* void *main*(String[] args){

    SortedStack s1 = *new* *SortedStack*(6);

    s1.*push*(20);

    s1.*push*(0);

    s1.*push*(10);

    s1.*push*(4);

    s1.*push*(12);

    s1.*push*(3);

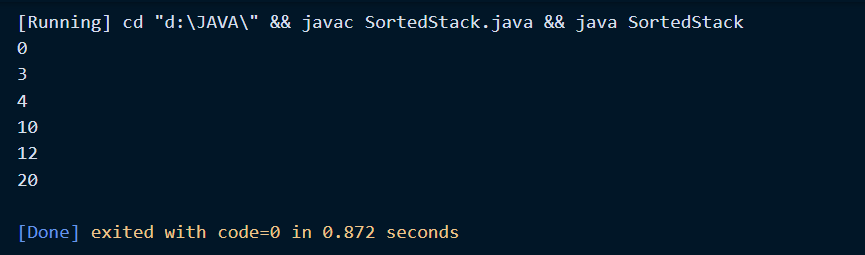
*sortStack*(s1);

    s1.*displayStack*();

    }

    }

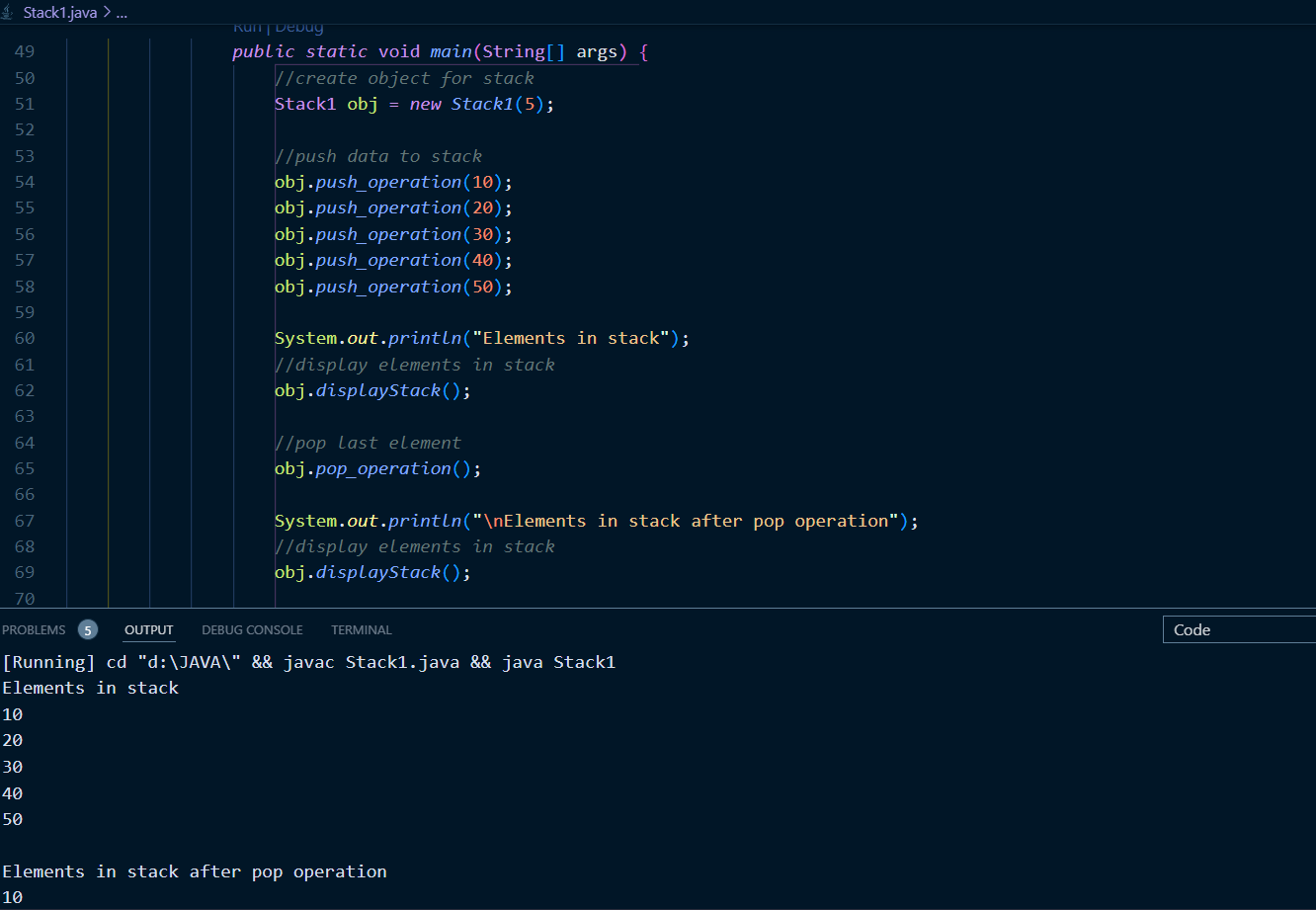
OUTPUT:



Explanation:

* We create a stack using array method, where we initialize push, pop, if full, if empty, display method.
* We create our required method that is sorting the elements of stack in ascending order (increasing order).
* In sorting method, we create an array to store the elements of input stack in sorted form and then that method after sorting will put the elements back into the stack.

NOTE: Stack output is given in form where 0th index element comes topmost and so on. Take the image posted below as reference:



7. Write a java code to reverse single linked list using recursion. Explain each step-in recursion graphically.

CODE:

// *Recursive Java program to reverse a linked list*

*class* ReverseLL {

*static* Node head; // *head of list*

*static* *class* Node {

        int data;

        Node next;

*Node*(int d)

        {

            data = d;

            next = null;

        }

    }

*static* Node *reverse*(Node head)

    {

*if* (head == null || head.*next* == null)

*return* head;

        /\* *reverse the rest list and put*

*the first element at the end* \*/

        Node rest = *reverse*(head.*next*);

        head.*next*.*next* = head;

        head.*next* = null;

        /\* *fix the head pointer* \*/

*return* rest;

    }

    /\* *Function to print linked list, we created a temporary list* \*/

*static* void *print*()

    {

        Node temp = head;

*while* (temp!= null) {

            System.*out*.*print*(temp.*data* + " ");

            temp = temp.*next*;

        }

        System.*out*.*println*();

    }

*static* void *push*(int data)

    {

        Node temp = *new* *Node*(data);

        temp.*next* = head;

        head = temp;

    }

*public* *static* void *main*(String args[])

{

    /\* *Start with the empty list* \*/

*push*(10);

*push*(84);

*push*(69);

*push*(77);

*push*(99);

    System.*out*.*println*("Given linked list");

*print*();

    head = *reverse*(head);

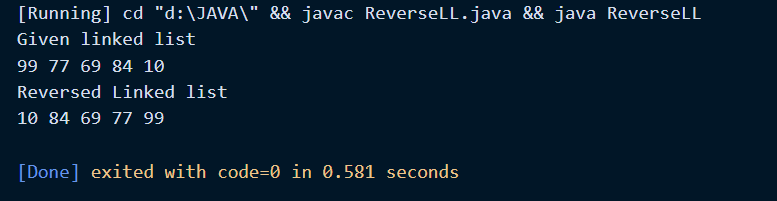
    System.*out*.*println*("Reversed Linked list");

*print*();

}

}

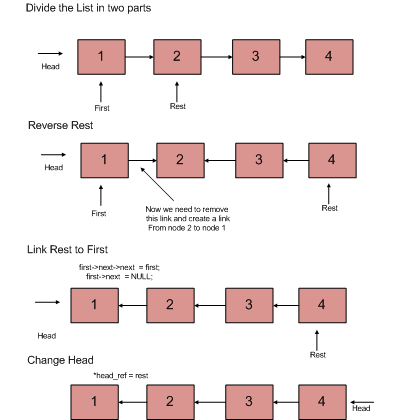
OUTPUT:



Explanation:

* We initialized a single linked list with integer as data type.
* We divided our single linked list into two parts, one is first node and other being rest of the list.
* We fixed the head and then We reverse the pointer of second part of division i.e., nodes other than first.
* We link rest of the nodes to first after reversing and then finally give the head to the last node.
* We created a temporary linked list and printed the data after reversing it recursively.

Graphical Image:



**THANK YOU!**