**Data Structures:**

Logical relationship existing between individual elements of data

Way of organizing all data items

It doesn’t consider not only the elements stored but also their relationship to each other

Mathematical or logical model of a particular organization of data items

**Classification of Data Structures**

Linear

Array

List

Stack

Queue

Non linear

Trees

Graphs

**Linked List**

Sequence of items(objects) where every item is linked to the next

Each element is dynamically allocated and in which elements points to each other

Elements are called nodes where each node has data and pointer to the next node

Requires more memory than array

**Types of Linked List**

Single Linked list

Double Linked list

Circular Linked List

**Insert at front**

import java.util.\*;

class Node {

int data;

Node next;

}

class LinkedList {

Node head;

void push(int value) {

Node newNode = new Node();

newNode.data = value;

newNode.next = head;

head = newNode;

}

void print() {

Node temp = head;

while(temp != null) {

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

temp = temp.next;

}

}

}

public class Main {

public static void main(String[] args) {

LinkedList list = new LinkedList();

int n;

Scanner scanner = new Scanner(System.in);

while(true) {

n = scanner.nextInt();

if(n != -1) {

list.push(n);

} else {

break;

}

}

list.print();

}

}

**Insert at Back**

import java.util.Scanner;

class Node {

int data;

Node next;

Node(int data) {

this. data = data;

this. next = null;

}

}

public class Main {

static Node head = null,temp=null;;

static void insert(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

temp = newNode;

} else {

temp.next=newNode;

temp=newNode;

}

}

static void print() {

Node temp = head;

while (temp != null) {

System.out.println(temp.data);

temp = temp.next;

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int data;

while (true) {

data = scanner.nextInt();

if (data < 0) {

if (head == null) {

System.out.println("List is empty");

}

break;

}

insert(data);

}

print();

}

}

**Doubly Linked List**

Each node contains data and two pointers to point previous and next node

Main advantage is we can traverse in any direction, forward or backward

Drawback is it requires more memory compared to singly linked list because we need an extra pointer

import java.util.\*;

class Node {

public int data;

public Node prev;

public Node next;

}

public class DoublyLinkedList {

public static void push(Node head, int value) {

Node newnode = new Node();

newnode.data = value;

newnode.next = head;

newnode.prev = null;

if (head != null)

head.prev = newnode;

head = newnode;

}

public static void print(Node head) {

Node last;

System.out.println("Forward Direction");

while (head != null) {

System.out.print(head.data);

last = head;

head = head.next;

}

System.out.println();

System.out.println("Backward Direction");

while (last != null) {

System.out.print(last.data);

last = last.prev;

}

System.out.println();

}

public static void main(String[] args) {

Node head = null;

Scanner scanner = new Scanner(System.in);

int n;

while (true) {

n = scanner.nextInt();

if (n != -1) {

push(head, n);

} else {

break;

}

}

print(head);

}

}

**Circular Linked List**

Where last node points to first node

It does not contain null pointers

It has the biggest advantage of time saving when we want to go to the first node from the last node.

Directly points to first node

class Node {

public int data;

public Node next;

}

public class CircularLinkedList {

public static void push(Node head, int value) {

Node newnode = new Node();

Node temp = head;

newnode.data = value;

newnode.next = head;

if (head != null) {

while (temp.next != head) {

temp = temp.next;

}

temp.next = newnode;

} else {

newnode.next = newnode;

}

head = newnode;

}

public static void print(Node head) {

Node temp = head;

if (head != null) {

do {

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

temp = temp.next;

} while (temp != head);

}

}

public static void main(String[] args) {

Node head = null;

Scanner scanner = new Scanner(System.in);

int n;

while (true) {

n = scanner.nextInt();

if (n != -1) {

push(head, n);

} else {

break;

}

}

print(head);

}

}

**Stack – Data Structure**

**Stack**

Stacks are dynamic data structures that follow the Last In First Out (LIFO) principle.

The last item to be inserted into a stack is the first one to be deleted from it.

Both insertion and removal are allowed at only one end of Stack called Top.

push() function is used to insert new elements into the Stack and pop() function is used to remove an element from the stack.

Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.

**Applications**

Stack Using Array

Push

Pop

Inserting a value into the stack

In Array, Insert at 0 index

Deleting a value from the Stack

In Array, Delete at the end

Algorithm for PUSH operation

Check if the stack is full or not.

If the stack is full, then print an error of overflow and exit the program.

If the stack is not full, then increment the top and add the element.

**Algorithm for POP operation**

Check if the stack is empty or not.

If the stack is empty, then print an error of underflow and exit the program.

If the stack is not empty, then print the element at the top and decrement the top.

**Stack – Operations**

push()

pop()

topElement()

isEmpty()

size()

public class Stack {

static int top = -1;

static int[] stack = new int[3];

static void push(int x, int n) {

if (top == n - 1) {

System.out.println("Stack Overflow condition!");

} else {

top = top + 1;

stack[top] = x;

}

}

static boolean isEmpty() {

return top == -1;

}

static void pop() {

if (isEmpty()) {

System.out.println("Stack Underflow condition!");

} else {

top = top - 1;

}

}

static int size() {

return top + 1;

}

static int topElement() {

return stack[top];

}

public static void main(String[] args) {

push(5, 3);

System.out.println("Current size of stack is " + size());

push(10, 3);

push(15, 3);

System.out.println("Current size of stack is " + size());

push(20, 3);

System.out.println("The current top element in stack is " + topElement());

for (int i = 0; i < 3; i++) {

pop();

}

System.out.println("Current size of stack is " + size());

pop();

}

}

**OUTPUT**

Current size of stack is 1 Current size of stack is 3 Stack Overflow conditions!

The current top element in stack is 15 Current size of stack is 0

Stack Underflow condition

Stack Using Linked List

Dynamic memory allocation

Flexible size

class Node {

int data;

Node link;

}

class Stack {

Node top;

void push(int data) {

Node temp = new Node();

if (temp == null) {

System.out.println("\nHeap Overflow");

System.exit(1);

}

temp.data = data;

temp.link = top;

top = temp;

}

boolean isEmpty() {

return top == null;

}

int peek() {

if (!isEmpty())

return top.data;

else {

System.out.println("\nStack Underflow");

System.exit(1);

return -1; // This return statement is required to satisfy the compiler

}

}

void pop() {

if (top == null) {

System.out.println("\nStack Underflow");

System.exit(1);

} else {

Node temp = top;

top = top.link;

temp.link = null;

temp = null;

}

}

void display() {

if (top == null) {

System.out.println("\nStack Underflow");

System.exit(1);

} else {

Node temp = top;

while (temp != null) {

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

temp = temp.link;

}

}

}

public static void main(String[] args) {

Stack stack = new Stack();

stack.push(11);

stack.push(22);

stack.push(33);

stack.push(44);

stack.display();

System.out.println("\nTop element is " + stack.peek());

stack.pop();

stack.pop();

stack.display();

System.out.println("\nTop element is " + stack.peek());

}

}

**OUTPUT:**

44 33 22 11

Top element is 44 22 11

Top element is 22

**Program to evaluate the given expression and print the answer.**

**Sample Input:**

1 + 2

( 1 + 2 ) / 5

10 \* 20

Sample Output:

3

0

200

import java.util.\*;

class EvaluateExpression {

static int precedence(char op) {

if (op == '+' || op == '-') return 1;

if (op == '\*' || op == '/') return 2;

return 0;

}

static int applyOp(int a, int b, char op) {

switch(op) {

case '+': return a + b;

case '-': return a - b;

case '\*': return a \* b;

case '/': return a / b;

}

return 0;

}

static int evaluate(String tokens) {

Stack<Integer> values = new Stack<>();

Stack<Character> ops = new Stack<>();

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

if (tokens.charAt(i) == ' ') continue;

else if (tokens.charAt(i) == '(') ops.push(tokens.charAt(i));

else if (Character.isDigit(tokens.charAt(i))) {

int val = 0;

while (i < tokens.length() && Character.isDigit(tokens.charAt(i))) {

val = (val \* 10) + (tokens.charAt(i) - '0');

i++;

}

values.push(val);

} else if (tokens.charAt(i) == ')') {

while (!ops.isEmpty() && ops.peek() != '(') {

int val2 = values.pop();

int val1 = values.pop();

char op = ops.pop();

values.push(applyOp(val1, val2, op));

}

if (!ops.isEmpty()) ops.pop();

} else {

while (!ops.isEmpty() && precedence(ops.peek()) >= precedence(tokens.charAt(i))) {

int val2 = values.pop();

int val1 = values.pop();

char op = ops.pop();

values.push(applyOp(val1, val2, op));

}

ops.push(tokens.charAt(i));

}

}

while (!ops.isEmpty()) {

int val2 = values.pop();

int val1 = values.pop();

char op = ops.pop();

values.push(applyOp(val1, val2, op));

}

return values.pop();

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

for (int i = 1; i <= 3; i++) {

String exp = scanner.nextLine();

System.out.println(evaluate(exp));

}

}

}

**Queue – Data Structure Queue**

The first element is inserted from one end called REAR and deleted from the other end called FRONT.

Front points to the beginning of the queue and Rear points to the end of the queue.

Queue follows the FIFO (First - In - First Out) structure.

In a queue, one end is always used to insert data (enqueue) and the other is used to delete data (dequeue), because the queue is open at both its ends.

Applications

Serving requests on a single shared resource, like a printer, CPU task scheduling etc.

In real life scenarios, Call Center phone systems use Queues to hold people calling them in an order, until a service representative is free.

Queue – Operations

enqueue()

dequeue()

display()

**Algorithm for ENQUEUE operation**

Check if the queue is full or not.

If the queue is full, then print an overflow error and exit the program.

If the queue is not full, then increment the tail and add the element.

**Algorithm for DEQUEUE operation**

Check if the queue is empty or not.

If the queue is empty, then print an underflow error and exit the program.

If the queue is not empty, then print the element at the head and increment the head.

class Queue {

private final int SIZE = 5;

private int[] items = new int[SIZE];

private int front, rear;

public Queue() {

front = -1;

rear = -1;

}

public boolean isFull() {

return front == 0 && rear == SIZE - 1;

}

public boolean isEmpty() {

return front == -1;

}

public void enQueue(int element) {

if (isFull()) {

System.out.println("Queue is full");

} else {

if (front == -1) front = 0;

rear++;

items[rear] = element;

System.out.println("Inserted " + element);

}

}

public int deQueue() {

int element;

if (isEmpty()) {

System.out.println("Queue is empty");

return -1;

} else {

element = items[front];

if (front >= rear) {

front = -1;

rear = -1;

} else {

front++;

}

System.out.println("Deleted -> " + element);

return element;

}

}

public void display() {

int i;

if (isEmpty()) {

System.out.println("\nEmpty Queue");

} else {

System.out.println("\nFront -> " + front);

System.out.print("Items -> ");

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

System.out.print(items[i] + "\t");

}

System.out.println("\nRear -> " + rear);

}

}

}

public class Main {

public static void main(String[] args) {

Queue q = new Queue();

q.deQueue();

for (int i = 1; i <= 5; i++) {

q.enQueue(i);

}

q.enQueue(6);

q.display();

q.deQueue();

q.display();

}

}

**OUTPUT:**

Queue is empty Inserted 1

Inserted 2

Inserted 3

Inserted 4

Inserted 5 Queue is full Front -> 0

Items -> 1

Rear -> 4

2

3

4

5

Deleted -> 1

Front -> 1

Items -> 2

Rear -> 4

3

4

5

**Queue Using Linked List**

Using dynamic memory allocation

Flexible size

class Node {

public int data;

public Node next;

}

class Queue {

private Node head = null;

private Node tail = null;

public void enqueue(int value) {

Node newNode = new Node();

newNode.data = value;

newNode.next = null;

if (head == null) {

head = newNode;

tail = newNode;

} else {

tail.next = newNode;

tail = newNode;

}

}

public void dequeue() {

if (head != null) {

Node temp = head;

head = head.next;

temp.next = null;

}

}

public void print() {

Node temp = head;

while (temp != null) {

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

temp = temp.next;

}

System.out.println();

}

}

public class Main {

public static void main(String[] args) {

Queue queue = new Queue();

while (true) {

int n = /\* Input from user \*/; // You need to add input handling here

if (n != -1) {

queue.enqueue(n);

} else {

break;

}

}

queue.print();

queue.dequeue();

queue.print();

}

}

**Types of Queue**

Simple Queue

Circular Queue

Priority Queue

Double Ended Queue

**Circular Queue**

Unlike the simple queues, in a circular queue each node is connected to the next node in sequence but the last node’s pointer is also connected to the first node’s address.

Hence, the last node and the first node also gets connected making a circular link overall

**Priority Queue**

Priority queue makes data retrieval possible only through a pre- determined priority number assigned to the data items.

While the deletion is performed in accordance to priority number (the data item with highest priority is removed first), insertion is performed only in the order.

**Double Ended Queue – Dequeue**

The doubly ended queue or deque allows the insert and delete operations from both ends (front and rear) of the queue.

Queues are an important concept of the data structures and understanding their types is very necessary for working appropriately with them

**GRAPHS**

A graph is a pictorial representation of a set of objects connected by links.

**Vertices**

The objects are represented by points termed as vertices.

**Edges**

The links that connect the vertices are called edges.

**Path**

Path represents a sequence of edges between the two vertices.

Adjacency

Two node or vertices are adjacent if they are connected to each other through an edge

(1,2) are adjacent.

(1,3) are adjacent.

(2,4) are adjacent.

(3,4) are adjacent

But, (2,3) are not adjacent as it is not connected by edges

Types of Graph

**DIRECTED GRAPH**

B) UNDIRECTED GRAPH:

c) CYCLIC GRAPH

D) ACYCLIC GRAPH :

E) WEIGHTED GRAPH:

F) UNWEIGHTED GRAPH

**Graph and its Representations**

Adjacency Matrix

Adjacency List

Adjacency Matrix

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph.

Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric.

Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

Adjacency Matrix

All the edges have been marked. This is the adjacency matrix for the given graph

Adjacency List

An array of linked lists is used.

Size of the array is equal to number of vertices. Let the array be array[].

An entry array[i] represents the linked list of vertices adjacent to the ith vertex.

This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists.

Adjacency list

This is the adjacency list for the given graph

Types of Traversals

Depth First Search (DFS) algorithm.

Breadth First Search (BFS) algorithm

**DFS**

1)(DFS) algorithm traverses a graph in a depth ward motion

2)Use a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

**DFS Traversal Rules**

Rule 1 − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

Rule 2 − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

Rule 3 − Repeat Rule 1 and Rule 2 until the stack is empty.

**PROGRAMS**

1.

SAMPLE INPUT & OUTPUT 1:

Please enter the number of nodes in the graph

4

Please enter the number of edges in the graph

4

Is the graph directed

no

Enter the start node, end node and weight of edge no 0

0 1 7

Enter the start node, end node and weight of edge no 1

2 3 5

Enter the start node, end node and weight of edge no 2

1 2 3

Enter the start node, end node and weight of edge no 3

3 0 6

Adjacency Matrix Representation:

0 7 0 6

7 0 3 0

0 3 0 5

6 0 5 0

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int[][] graph;

int node, edge, f = 0, sn, en, we;

String dir;

System.out.println("Please enter the number of nodes in the graph");

node = scanner.nextInt();

System.out.println("Please enter the number of edges in the graph");

edge = scanner.nextInt();

graph = new int[node][edge];

System.out.println("Is the graph directed (yes/no)");

dir = scanner.next();

if (dir.equals("yes")) {

f = 1;

}

for (int i = 0; i < edge; i++) {

System.out.printf("Enter the start node, end node and weight of edge no %d\n", i);

sn = scanner.nextInt();

en = scanner.nextInt();

we = scanner.nextInt();

graph[sn][en] = we;

if (f == 0) {

graph[en][sn] = we;

}

}

System.out.println("Adjacency Matrix Representation:");

for (int i = 0; i < node; i++) {

for (int j = 0; j < edge; j++) {

System.out.print(graph[i][j] + " ");

}

System.out.println();

}

}

}

2.

**SAMPLE INPUT & OUTPUT:**

Please enter the number of nodes in the graph

4

Please enter the number of edges in the graph

4

Is the graph directed

no

Enter the start node, end node and weight of edge no 0

0 1 7

Enter the start node, end node and weight of edge no 1

2 3 5

Enter the start node, end node and weight of edge no 2

1 2 4

Enter the start node, end node and weight of edge no 3

3 0 6

Adjacency Matrix Representation:

0 7 0 6

7 0 4 0

0 4 0 5

6 0 5 0

Adjacency List Representation:

Node 0 is connected to the following nodes:

Node 1 with edge weight 7

Node 3 with edge weight 6

Node 1 is connected to the following nodes:

Node 0 with edge weight 7

Node 2 with edge weight 4

Node 2 is connected to the following nodes:

Node 1 with edge weight 4

Node 3 with edge weight 5

Node 3 is connected to the following nodes:

Node 0 with edge weight 6

Node 2 with edge weight 5

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int[][] graph;

int node, edge, f = 0, sn, en, we;

String dir;

System.out.println("Please enter the number of nodes in the graph");

node = scanner.nextInt();

System.out.println("Please enter the number of edges in the graph");

edge = scanner.nextInt();

graph = new int[node][edge];

System.out.println("Is the graph directed (yes/no)");

dir = scanner.next();

if (dir.equals("yes")) {

f = 1;

}

for (int i = 0; i < edge; i++) {

System.out.printf("Enter the start node, end node and weight of edge no %d\n", i);

sn = scanner.nextInt();

en = scanner.nextInt();

we = scanner.nextInt();

graph[sn][en] = we;

if (f == 0) {

graph[en][sn] = we;

}

}

System.out.println("Adjacency Matrix Representation:\n");

for (int i = 0; i < node; i++) {

for (int j = 0; j < edge; j++) {

System.out.print(graph[i][j] + " ");

}

System.out.println();

}

System.out.println("\nAdjacency List Representation:");

for (int i = 0; i < node; i++) {

System.out.printf("Node %d is connected to the following nodes:\n", i);

for (int j = 0; j < edge; j++) {

if (graph[i][j] != 0) {

System.out.printf("Node %d with edge weight %d\n", j, graph[i][j]);

}

}

}

}

}