**BAHRIA UNIVERSITY, Karachi Campus)**

# Department of Software Engineering

# ASSIGNMENT # 04 – Fall 2022

**CLO 4**

|  |  |  |
| --- | --- | --- |
| Course Title: **Software Construction** |  | Course Code: **SEC-311** |
| Class: **BSE – 5(B)** |  | Shift: **Morning** |
| Course Instructor: **Engr. Misbah Perveen** |  | Date: **30th Dec 2022** |
| Due Date: **08th Jan 2023**  **Name: M Muaz Shahzad** |  | Max. Marks: **5.0 Marks** |
|  |  |  |

**Question No. 1: [CLO#04, 5.0 marks]**

Considering the Abstract data type concept used in object-oriented programming language, and implement through design model. Give five examples to support your answer.

**Solution**

An abstract data type (ADT) is a logical description of a data type, without specifying how it will be implemented. It specifies a set of values and a set of operations that can be performed on those values.

An ADT is often implemented as a class or a set of classes in an object-oriented programming language. The implementation of the ADT's operations is referred to as the ADT's representation.

Here are five ADTs that are mostly used in object-oriented programming languages:

1. **Stack ADT**

A stack is an Abstract Data Type (ADT) that is found in almost all programming languages. It is called a stack because it behaves like a real-world stack, such as a deck of cards or a pile of dishes.

A real-world stack can only perform operations at one end. For instance, we can only place or remove a card or plate from the top of the stack. Similarly, Stack ADT permits all data operations at only one end. We can only access the top element of a stack at any given time.

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

* **push()** − Pushing (storing) an element on the stack.
* **pop()** − Removing (accessing) an element from the stack.
* **peek()** − get the top data element of the stack, without removing it.
* **isFull()** − check if stack is full.
* **isEmpty()** − check if stack is empty.

**Example of Code Implementation using Stack**

// Stack implementation in Java

class Stack {

  // store elements of stack

  private int arr[];

  // represent top of stack

  private int top;

  // total capacity of the stack

  private int capacity;

  // Creating a stack

  Stack(int size) {

    // initialize the array

    // initialize the stack variables

    arr = new int[size];

    capacity = size;

    top = -1;

  }

  // push elements to the top of stack

  public void push(int x) {

    if (isFull()) {

      System.out.println("Stack OverFlow");

      // terminates the program

      System.exit(1);

    }

    // insert element on top of stack

    System.out.println("Inserting " + x);

    arr[++top] = x;

  }

  // pop elements from top of stack

  public int pop() {

    // if stack is empty

    // no element to pop

    if (isEmpty()) {

      System.out.println("STACK EMPTY");

      // terminates the program

      System.exit(1);

    }

    // pop element from top of stack

    return arr[top--];

  }

  // return size of the stack

  public int getSize() {

    return top + 1;

  }

  // check if the stack is empty

  public Boolean isEmpty() {

    return top == -1;

  }

  // check if the stack is full

  public Boolean isFull() {

    return top == capacity - 1;

  }

  // display elements of stack

  public void printStack() {

    for (int i = 0; i <= top; i++) {

      System.out.print(arr[i] + ", ");

    }

  }

  public static void main(String[] args) {

    Stack stack = new Stack(5);

    stack.push(1);

    stack.push(2);

    stack.push(3);

    System.out.print("Stack: ");

    stack.printStack();

    // remove element from stack

    stack.pop();

    System.out.println("\nAfter popping out");

    stack.printStack();

  }

}

1. **Queue ADT**

The queue abstract data type (ADT) is similar to the stack abstract data type in design. Each node has a void pointer to the data as well as a link pointer to the next queue member. The program is responsible for allocating memory for data storage.

These are rhe following operations that are performed using Queue.

* enqueue() – Insert an element at the end of the queue.
* dequeue() – Remove and return the first element of the queue, if the queue is not empty.
* peek() – Return the element of the queue without removing it, if the queue is not empty.
* size() – Return the number of elements in the queue.
* isEmpty() – Return true if the queue is empty, otherwise return false.
* isFull() – Return true if the queue is full, otherwise return false.

**Example of Code Implementation using Queue**

import java.util.NoSuchElementException;

public class Queue {

  private int front;

  private int rear;

  private int size;

  private int capacity;

  private int[] queue;

  public Queue(int capacity) {

    this.front = this.size = 0;

    this.capacity = capacity;

    this.rear = capacity - 1;

    this.queue = new int[this.capacity];

  }

  // Queue is full when size becomes equal to the capacity

  public boolean isFull() {

    return this.size == this.capacity;

  }

  // Queue is empty when size is 0

  public boolean isEmpty() {

    return this.size == 0;

  }

  // Method to add an item to the queue.

  // It changes rear and size

  public void enqueue(int item) {

    if (isFull()) {

      return;

    }

    this.rear = (this.rear + 1) % this.capacity;

    this.queue[this.rear] = item;

    this.size = this.size + 1;

    System.out.println(item + " enqueued to queue");

  }

  // Method to remove an item from queue.

  // It changes front and size

  public int dequeue() {

    if (isEmpty()) {

      throw new NoSuchElementException("Underflow Exception");

    }

    int item = this.queue[this.front];

    this.front = (this.front + 1) % this.capacity;

    this.size = this.size - 1;

    return item;

  }

  // Method to get front of queue

  public int front() {

    if (isEmpty()) {

      throw new NoSuchElementException("Underflow Exception");

    }

    return this.queue[this.front];

  }

  // Method to get rear of queue

  public int rear() {

    if (isEmpty()) {

      throw new NoSuchElementException("Underflow Exception");

    }

    return this.queue[this.rear];

  }

}

1. **Linked List ADT**

A Linked List is an Abstract Data Type (ADT) that contains a collection of Nodes that can be accessed sequentially. A Linked List does not allow for random access to a Node.

These are the operations that are performed using Linked Lists.

* [Traversal](https://www.programiz.com/dsa/linked-list-operations#traverse) - access each element of the linked list
* [Insertion](https://www.programiz.com/dsa/linked-list-operations#add) - adds a new element to the linked list
* [Deletion](https://www.programiz.com/dsa/linked-list-operations#delete) - removes the existing elements
* [Search](https://www.programiz.com/dsa/linked-list-operations#search) - find a node in the linked list
* [Sort](https://www.programiz.com/dsa/linked-list-operations#sort) - sort the nodes of the linked list

**Example of Code Implementation using Linked List**

public class LinkedList {

    private Node head;

    private int size;

    public LinkedList() {

        this.head = null;

        this.size = 0;

    }

    public void add(int value) {

        Node newNode = new Node(value);

        newNode.setNext(this.head);

        this.head = newNode;

        this.size++;

    }

    public int size() {

        return this.size;

    }

    public boolean isEmpty() {

        return this.size == 0;

    }

    public void remove(int value) {

        Node current = this.head;

        Node previous = null;

        while (current != null) {

            if (current.getValue() == value) {

                if (previous == null) {

                    this.head = current.getNext();

                } else {

                    previous.setNext(current.getNext());

                }

                this.size--;

                return;

            }

            previous = current;

            current = current.getNext();

        }

    }

    public void print() {

        Node current = this.head;

        while (current != null) {

            System.out.print(current.getValue() + " ");

            current = current.getNext();

        }

        System.out.println();

    }

    private class Node {

        private int value;

        private Node next;

        public Node(int value) {

            this.value = value;

            this.next = null;

        }

        public int getValue() {

            return this.value;

        }

        public void setNext(Node next) {

            this.next = next;

        }

        public Node getNext() {

            return this.next;

        }

    }

}

1. **Graph ADT**

A graph is a non-linear data structure that consists of nodes (also known as vertices) and edges that connect these nodes. An adjacency list or an adjacency matrix can be used to represent a graph.

**Example of Code Implementation using Graph**

Here's an example of a simple Graph class in Java that makes use of an adjacency list

import java.util.ArrayList;

import java.util.List;

class Graph {

    // number of vertices in the graph

    private final int V;

    // adjacency list

    private final List<List<Integer>> adj;

    public Graph(int V) {

        this.V = V;

        adj = new ArrayList<>(V);

        for (int i = 0; i < V; i++)

            adj.add(new ArrayList<>());

    }

    public void addEdge(int v, int w) {

        adj.get(v).add(w);

        adj.get(w).add(v);

    }

    public Iterable<Integer> adj(int v) {

        return adj.get(v);

    }

}

1. **Tree ADT**

A tree is a data structure made up of nodes arranged in a hierarchy. Each node in a tree has zero or more child nodes (referred to as its children) and a single parent node (referred to as its parent). The root is the topmost node in the tree.

**Example of Code Implementation using Tree**

class Tree {

    TreeNode root;

    public Tree(int value) {

        this.root = new TreeNode(value);

    }

    public void insert(int value) {

        insertRecursive(root, value);

    }

    private void insertRecursive(TreeNode current, int value) {

        if (value < current.value) {

            if (current.left == null) {

                current.left = new TreeNode(value);

            } else {

                insertRecursive(current.left, value);

            }

        } else {

            if (current.right == null) {

                current.right = new TreeNode(value);

            } else {

                insertRecursive(current.right, value);

            }

        }

    }

}