**Q.1 Array and linked list implementation of list**

**Theory**

Arrays and linked lists are fundamental data structures used for implementing lists in computer programming.

Arrays are contiguous blocks of memory that store elements of the same data type. They provide direct access to elements using indexing, making them efficient for random access operations. However, arrays have a fixed size, which must be specified when they are created. This limitation makes resizing arrays expensive and impractical in many situations.

Linked lists, on the other hand, consist of nodes where each node contains a data element and a reference (or pointer) to the next node in the sequence. This dynamic structure allows for efficient insertion and deletion operations, as nodes can be easily added or removed without the need to shift other elements. However, linked lists do not support direct access to elements by index, requiring sequential traversal from the head node to reach a specific element.

When implementing lists, the choice between arrays and linked lists depends on the specific requirements of the application. Arrays are preferred for scenarios that require frequent random access to elements or a fixed-size collection. Linked lists are more suitable when dynamic resizing, efficient insertion/deletion, or memory efficiency is critical.

In Java, lists can be implemented using arrays through the ArrayList class, which provides dynamic resizing and random access capabilities. Alternatively, linked lists can be implemented using the LinkedList class, which offers efficient insertion and deletion operations by maintaining references between nodes. Both implementations offer their own set of advantages and trade-offs, allowing developers to choose the most appropriate data structure based on the specific needs of their application.

**Java code:**

import java.util.Scanner;

public class LinearListOperations {

    private static final int LIST\_SIZE = 30;

    private static int[] element;

    private static int top = -1;

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        while (true) {

            System.out.println("\n\n Basic Operations in a Linear List......");

            System.out.println(" 1. Create New List \t 2. Modify List \t 3. View List");

            System.out.println(" 4. Insert First \t 5. Insert Last \t 6. Insert Middle");

            System.out.println(" 7. Delete First \t 8. Delete Last \t 9. Delete Middle");

            System.out.println("Enter the Choice 1 to 10: ");

            int ch = scanner.nextInt();

            switch (ch) {

                case 1:

                    top = -1;

                    System.out.println("Enter the Limit (How many Elements):");

                    int n = scanner.nextInt();

                    element = new int[LIST\_SIZE];

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

                        System.out.print("Enter The Element [" + (i + 1) + "]: ");

                        element[++top] = scanner.nextInt();

                    }

                    break;

                case 2:

                    if (top == -1) {

                        System.out.println("Linear List is Empty:");

                        break;

                    }

                    System.out.println("Enter the Element for Modification:");

                    int moddata = scanner.nextInt();

                    int found = 0;

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

                        if (element[i] == moddata) {

                            found = 1;

                            System.out.println("Enter The New Element:");

                            element[i] = scanner.nextInt();

                            break;

                        }

                    }

                    if (found == 0)

                        System.out.println("Element " + moddata + " not found");

                    break;

                case 3:

                    if (top == -1)

                        System.out.println("\n Linear List is Empty:");

                    else if (top == LIST\_SIZE - 1)

                        System.out.println("\n Linear List is Full:");

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

                        System.out.println("Element[" + (i + 1) + "] is --> " + element[i]);

                    break;

                case 4:

                    if (top == LIST\_SIZE - 1) {

                        System.out.println("Linear List is Full:");

                        break;

                    }

                    top++;

                    for (int i = top; i > 0; i--)

                        element[i] = element[i - 1];

                    System.out.println("Enter the Element:");

                    element[0] = scanner.nextInt();

                    break;

                case 5:

                    if (top == LIST\_SIZE - 1) {

                        System.out.println("Linear List is Full:");

                        break;

                    }

                    System.out.println("Enter the Element:");

                    element[++top] = scanner.nextInt();

                    break;

                case 6:

                    if (top == LIST\_SIZE - 1)

                        System.out.println("Linear List is Full:");

                    else if (top == -1)

                        System.out.println("Linear List is Empty.");

                    else {

                        found = 0;

                        System.out.println("Enter the Element after which the insertion is to be made:");

                        int insdata = scanner.nextInt();

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

                            if (element[i] == insdata) {

                                found = 1;

                                top++;

                                for (int j = top; j > i; j--)

                                    element[j] = element[j - 1];

                                System.out.println("Enter the Element:");

                                element[i + 1] = scanner.nextInt();

                                break;

                            }

                        if (found == 0)

                            System.out.println("Element " + insdata + " Not Found");

                    }

                    break;

                case 7:

                    if (top == -1) {

                        System.out.println("Linear List is Empty:");

                        break;

                    }

                    System.out.println("Deleted Data --> Element: " + element[0]);

                    top--;

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

                        element[i] = element[i + 1];

                    break;

                case 8:

                    if (top == -1)

                        System.out.println("Linear List is Empty:");

                    else

                        System.out.println("Deleted Data --> Element: " + element[top--]);

                    break;

                case 9:

                    if (top == -1) {

                        System.out.println("Linear List is Empty:");

                        break;

                    }

                    System.out.println("Enter the Element for Deletion:");

                    int deldata = scanner.nextInt();

                    found = 0;

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

                        if (element[i] == deldata) {

                            found = 1;

                            System.out.println("Deleted Data --> Element: " + element[i]);

                            top--;

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

                                element[j] = element[j + 1];

                            break;

                        }

                    if (found == 0)

                        System.out.println("Element " + deldata + " Not Found");

                    break;

                default:

                    System.out.println("End Of Run Of Your Program.........");

                    scanner.close();

                    System.exit(0);

A screenshot of a computer program

Description automatically generated  }}}}

**Conclusion:**

Implementations of lists using arrays and linked lists offer distinct advantages and trade-offs, making them suitable for different scenarios. Arrays provide constant-time access to elements based on their indices, facilitating efficient random access. This property makes arrays ideal for scenarios where frequent element access or updates by index is required. However, arrays have fixed sizes and may require resizing operations, which can be inefficient for dynamic lists with frequent insertions or deletions. Linked lists, on the other hand, offer dynamic memory allocation, enabling efficient insertion and deletion operations without the need for resizing. Each element in a linked list is connected through pointers, allowing for constant-time insertions and deletions at the beginning or end of the list. However, linked lists do not support efficient random access, requiring linear-time traversal from the head or tail to access specific elements. Additionally, linked lists incur overhead due to the storage of pointers, which can lead to higher memory consumption compared to arrays. Ultimately, the choice between array and linked list implementations depends on the specific requirements of the application, such as the frequency of element access, insertion, and deletion operations, as well as memory constraints.

**Q.2 Stack operations and Queue operations**

1. **Stack operations using Array**

**Theory:**

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle, where elements are inserted and removed from the same end, traditionally called the "top" of the stack. This means that the last element added to the stack is the first one to be removed.

Operations on Stack using Array:

Push: This operation adds an element to the top of the stack.

Pop: This operation removes the top element from the stack.

Peek (or Top): This operation returns the top element of the stack without removing it.

isEmpty: This operation checks if the stack is empty.

isFull (for fixed-size arrays): This operation checks if the stack is full (useful for arrays with a fixed capacity).

**Code using Java**

import java.util.Scanner;

public class Stack {

    private int maxSize;

    private int[] stackArray;

    private int top;

    public Stack(int size) {

        maxSize = size;

        stackArray = new int[maxSize];

        top = -1;

    }

    public void push(int value) {

        if (isFull()) {

            System.out.println("Stack overflow! Cannot push " + value);

            return;

        }

        top++;

        stackArray[top] = value;

    }

    public int pop() {

        if (isEmpty()) {

            System.out.println("Stack underflow! Cannot pop.");

            return -1;

        }

        int value = stackArray[top];

        top--;

        return value;

    }

    public boolean isEmpty() {

        return (top == -1);

    }

    public boolean isFull() {

        return (top == maxSize - 1);

    }

    public int peek() {

        if (isEmpty()) {

            System.out.println("Stack is empty. No element to peek.");

            return -1;

        }

        return stackArray[top];

    }

    public void display() {

        if (isEmpty()) {

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

            return;

        }

        System.out.println("Stack elements:");

        for (int i = top; i >= 0; i--) {

            System.out.println(stackArray[i]);

        }

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the size of the stack: ");

        int size = scanner.nextInt();

        Stack stack = new Stack(size);

        while (true) {

            System.out.println("\nChoose an operation:");

            System.out.println("1. Push");

            System.out.println("2. Pop");

            System.out.println("3. Peek");

            System.out.println("4. Display");

            System.out.println("5. Exit");

            int choice = scanner.nextInt();

            int value;

            switch (choice) {

                case 1:

                    System.out.print("Enter value to push: ");

                    value = scanner.nextInt();

                    stack.push(value);

                    break;

                case 2:

                    value = stack.pop();

                    if (value != -1) {

                        System.out.println("Popped element: " + value);

                    }

                    break;

                case 3:

                    value = stack.peek();

                    if (value != -1) {

                        System.out.println("Peeked element: " + value);

                    }

                    break;

                case 4:

                    stack.display();

                    break;

                case 5:

                    System.out.println("Exiting...");

                    System.exit(0);

                    break;

                default:

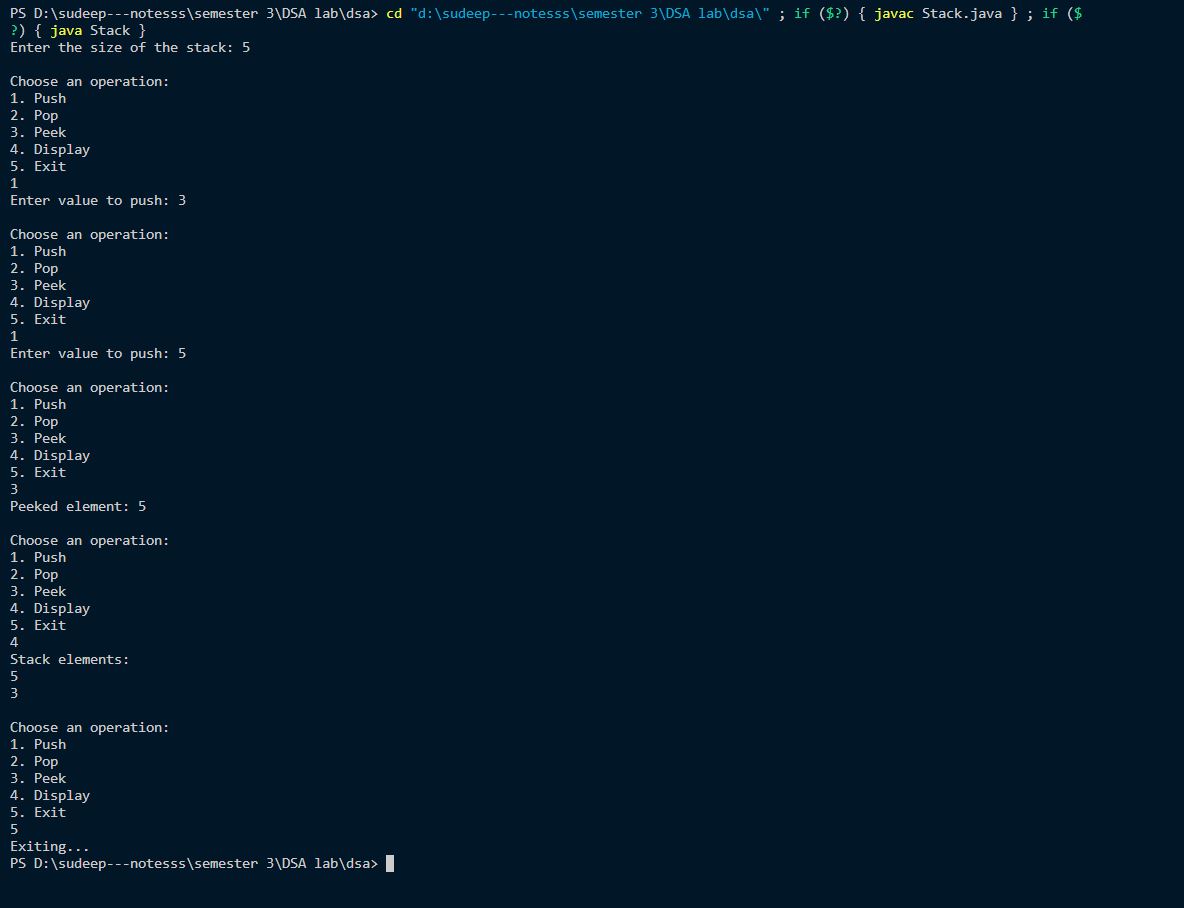
                    System.out.println("Invalid choice! Please enter a valid option.");

            }

        }

    }

}



**Conclusion:**

In conclusion, stacks implemented using arrays provide a simple and efficient way to manage data following the Last-In-First-Out (LIFO) principle. The array-based implementation offers constant time complexity for push and pop operations, making it suitable for scenarios where quick insertion and removal of elements are required. However, it's important to note that arrays have a fixed capacity, so there's a possibility of encountering stack overflow if the capacity is exceeded. Despite this limitation, stack operations using arrays are widely used in various applications, including expression evaluation, function call management, and parsing algorithms.

1. **Queue operations using Array**

**Theory:**

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle, where elements are inserted at the rear (end) and removed from the front. Think of it as a line of people waiting to be served at a counter.

Operations on Queue using Array:

Enqueue: This operation adds an element to the rear of the queue.

Dequeue: This operation removes the front element from the queue.

Front: This operation returns the front element of the queue without removing it.

Rear (or Back): This operation returns the rear element of the queue without removing it.

isEmpty: This operation checks if the queue is empty.

isFull (for fixed-size arrays): This operation checks if the queue is full (useful for arrays with a fixed capacity).

**Code using Java**

import java.util.Scanner;

public class Queue {

private static final int MAX\_SIZE = 100;

private int[] arr;

private int front;

private int rear;

public Queue() {

arr = new int[MAX\_SIZE];

front = -1;

rear = -1;

}

public boolean isEmpty() {

return front == -1;

}

public boolean isFull() {

return rear == MAX\_SIZE - 1;

}

public void enqueue(int value) {

if (isFull()) {

System.out.println("Queue is full. Cannot enqueue more elements.");

return;

}

if (isEmpty())

front = 0;

arr[++rear] = value;

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

}

public int dequeue() {

if (isEmpty()) {

System.out.println("Queue is empty. Cannot dequeue.");

return -1;

}

int value = arr[front];

if (front == rear) {

front = -1;

rear = -1;

} else {

front++;

}

System.out.println(value + " dequeued from queue.");

return value;

}

public int peek() {

if (isEmpty()) {

System.out.println("Queue is empty. Cannot peek.");

return -1;

}

return arr[front];

}

public void display() {

if (isEmpty()) {

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

return;

}

System.out.print("Queue: ");

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

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

}

System.out.println();

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

Queue queue = new Queue();

while (true) {

System.out.println("\nChoose operation:");

System.out.println("1. Enqueue");

System.out.println("2. Dequeue");

System.out.println("3. Peek");

System.out.println("4. Display");

System.out.println("5. Exit");

System.out.print("Enter your choice: ");

int choice = scanner.nextInt();

switch (choice) {

case 1:

System.out.print("Enter element to enqueue: ");

int element = scanner.nextInt();

queue.enqueue(element);

break;

case 2:

queue.dequeue();

break;

case 3:

int peekValue = queue.peek();

if (peekValue != -1)

System.out.println("Peeked element: " + peekValue);

break;

case 4:

queue.display();

break;

case 5:

System.out.println("Exiting...");

System.exit(0);

break;

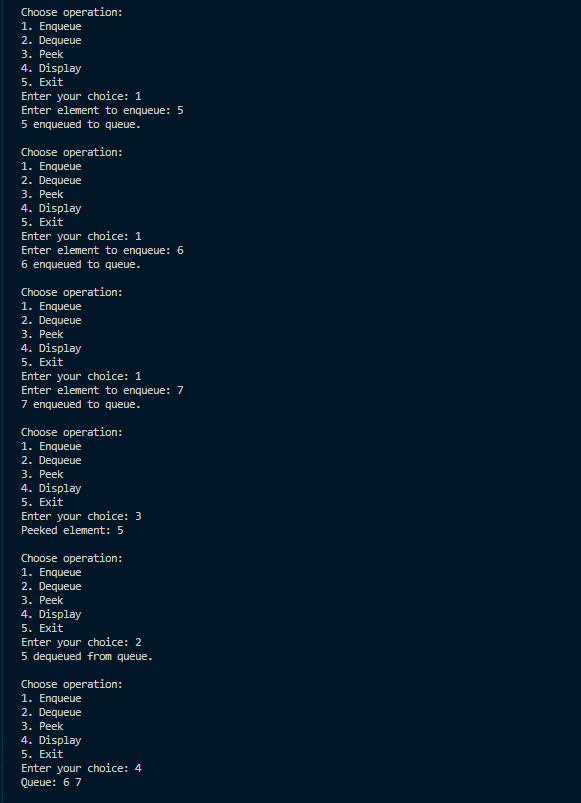
default:

System.out.println("Invalid choice. Please enter a valid choice.");

}

} }

}



**Conclusion:**

In summary, queues implemented using arrays offer an effective solution for managing data based on the First-In-First-Out (FIFO) principle. The array-based implementation provides efficient enqueue and dequeue operations with constant time complexity, making it suitable for scenarios requiring sequential data processing. However, similar to stacks, queues implemented with arrays have a fixed capacity, which may lead to queue overflow if the capacity is exceeded. Despite this limitation, array-based queues are extensively used in numerous applications, including task scheduling, message passing systems, and breadth-first search algorithms.

In both cases, while arrays provide efficient access and manipulation of elements, their fixed size can be a drawback in scenarios where dynamic resizing is required. Depending on the specific requirements and constraints of the application, alternative implementations using dynamic arrays or linked lists may be preferred to overcome the limitations associated with fixed-size arrays.

**Q3. Recursion**

**Theory:**

Recursion is a process in which the function calls itself indirectly or directly in

order to solve the problem.

* The function that performs the process of recursion is called a recursive

function. There are certain problems that can be solved pretty easily with the

help of a recursive algorithm.

1. Recursion is a problem-solving technique in which a function calls itself in

order to break down a complex problem into simpler, more manageable

subproblems

1. **WAP in java to find the Factorial of a number using recursion**

**Java code:**

import java.util.Scanner;

public class factorial {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number for factorial: ");

        int number = scanner.nextInt();

        System.out.printf("Factorial of " + number + " is " + fact(number));

        scanner.close();

    }

    public static long fact(int number) {

        if (number == 0 || number == 1) {

            return 1;

        } else {

            return number \* fact(number - 1);

        }}}

A screenshot of a computer program

Description automatically generated

1. **WAP to find the fibonacci sequence of a number using Recursion.**

**Java code:**

import java.util.Scanner;

public class FibonacciSeries {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter how many terms you want to generate: ");

        int number = scanner.nextInt();

        System.out.println("Fibonacci series up to " + number + " terms:");

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

            System.out.print(fibo(i) + "\t");

        }

        System.out.println();

    }

    public static int fibo(int number) {

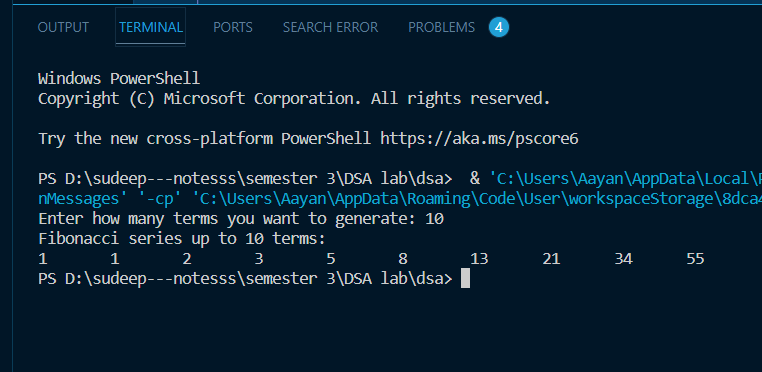
        if (number <= 1) {

            return number;

        } else {

            return fibo(number - 1) + fibo(number - 2);

        }}}



1. **WAP for finding the tower of hanoi of n disk using Recursion**

**Java code:**

import java.util.Scanner;

public class TowerOfHanoi {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of disks: ");

        int numDisks = scanner.nextInt();

        System.out.println("Steps to solve Tower of Hanoi with " + numDisks + "disks:");

        towerOfHanoi(numDisks, 'A', 'C', 'B');

    }

    public static void towerOfHanoi(int n, char source, char destination, char auxiliary) {

        if (n == 1) {

            System.out.println("Move disk 1 from " + source + " to " + destination);

            return;

        }

        towerOfHanoi(n - 1, source, auxiliary, destination);

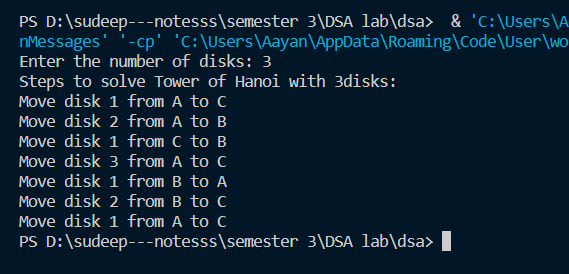
        System.out.println("Move disk " + n + " from " + source + " to " +

                destination);

        towerOfHanoi(n - 1, auxiliary, destination, source);

    }

}



**Conclusion:**

In conclusion, recursion is a powerful and elegant programming technique that simplifies complex problems by breaking them down into smaller, self-referential subproblems. It offers clarity in expressing solutions and mirrors the structure of the problems themselves, enhancing code readability and maintainability. However, careful consideration must be given to defining base cases to prevent infinite recursion and managing memory usage, as recursive solutions may lead to stack overflow for deep or nested calls. Despite these considerations, recursion remains a fundamental concept in computer science, providing intuitive solutions to a wide range of problems and serving as a cornerstone for algorithm design and problem-solving strategies.

**Q.4 Linked list implementation of Stack and Queue**

1. **Linked list implementation of stack**

**Theory:**

The linked list implementation of a stack utilizes a linked list data structure where each element of the stack is represented as a node in the list. Key points:

Node Structure: Each node holds the data and a reference to the next node.

Operations:

Push: Adds a new node at the beginning of the list.

Pop: Removes the first node from the list.

Display: Prints the elements of the stack by traversing the list.

Main Method: Provides a menu-driven interface for users to interact with the stack.

This implementation offers efficient insertion and deletion operations with a time complexity of O(1) and dynamic memory allocation.

**Java code:**

import java.util.Scanner;

class Node {

int val;

Node next;

Node(int value) {

val = value;

next = null;

}

}

public class Main {

static Node head;

static Scanner scanner = new Scanner(System.in);

public static void main(String[] args) {

int choice = 0;

System.out.println("\n\*\*\*\*\*\*\*\*\*Stack operations using linked list\*\*\*\*\*\*\*\*\*");

System.out.println("\n----------------------------------------------");

while (choice != 4) {

System.out.println("\n\nChoose one from the below options...");

System.out.println("1. Push\n2. Pop\n3. Show\n4. Exit");

System.out.print("Enter your choice: ");

choice = scanner.nextInt();

switch (choice) {

case 1 -> push();

case 2 -> pop();

case 3 -> display();

case 4 -> System.out.println("Exiting....");

default -> System.out.println("Please Enter a valid choice");

}

}

scanner.close();

}

static void push() {

int value;

System.out.print("Enter the value: ");

value = scanner.nextInt();

Node newNode = new Node(value);

if (head == null) {

head = newNode;

} else {

newNode.next = head;

head = newNode;

}

System.out.println("Item pushed: " + value);

}

static void pop() {

if (head == null) {

System.out.println("Underflow: Stack is empty");

return;

}

int item = head.val;

head = head.next;

System.out.println("Item popped: " + item);

}

static void display() {

if (head == null) {

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

return;

}

Node ptr = head;

System.out.println("Printing Stack elements: ");

while (ptr != null) {

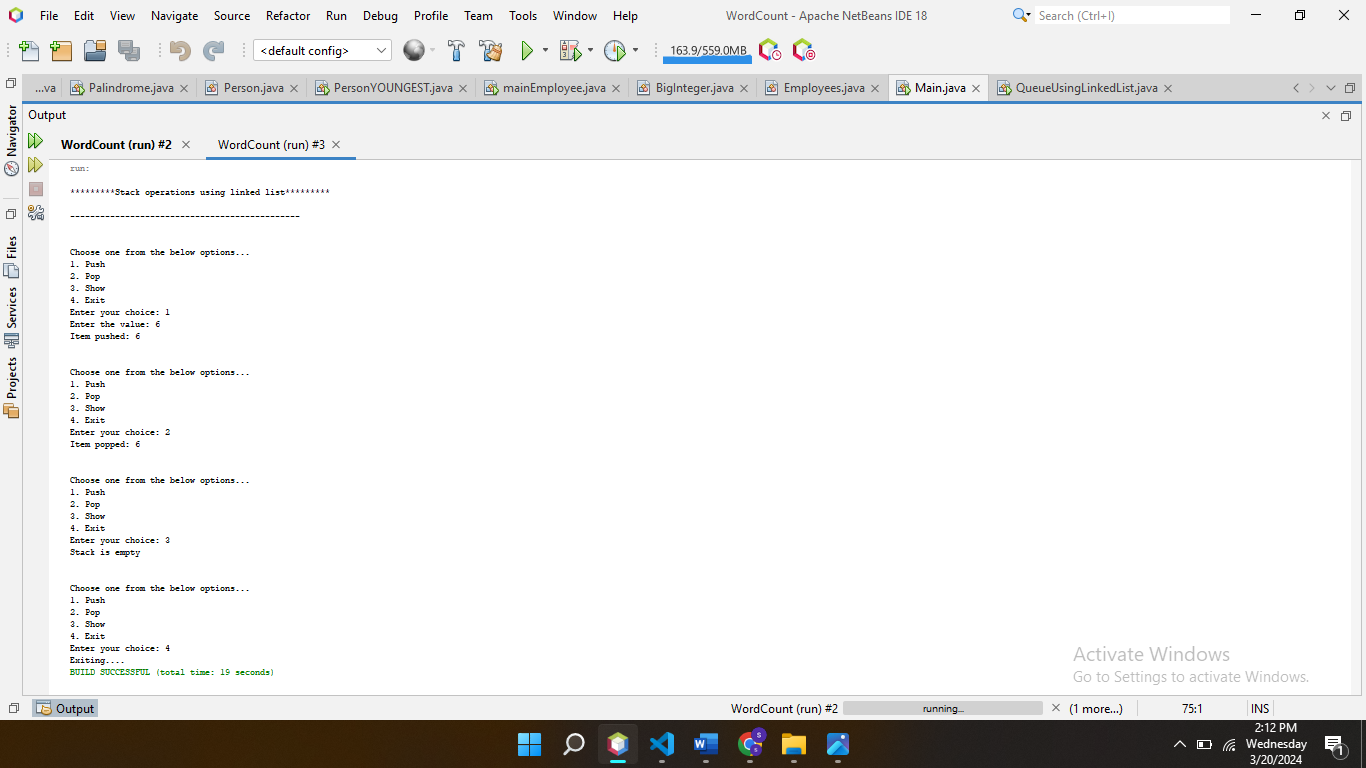
System.out.println(ptr.val);

ptr = ptr.next;

}

}

}



**Coclusion:**

In conclusion, the linked list implementation of a stack provides a flexible and efficient way to manage data. By leveraging the structure of a linked list, it offers constant-time insertion and deletion operations. This approach is particularly useful when the size of the stack is not known in advance or when memory needs to be dynamically allocated.

1. **Linked list implementation of Queue in java**

**Theory:**

The linked list implementation of a queue involves using a linked list data structure to represent the queue. Key points include:

Node Structure: Each node in the linked list holds the data and a reference to the next node.

Operations:

Enqueue: Adds an element to the rear of the queue by adding a new node at the end of the linked list.

Dequeue: Removes and returns the element at the front of the queue by removing the first node from the linked list.

Display: Prints the elements of the queue by traversing the linked list.

Main Method: Provides a menu-driven interface for users to interact with the queue, offering options to enqueue, dequeue, display, or exit. This implementation offers efficient insertion and deletion operations with a time complexity of O(1) and dynamic memory allocation. It's useful for scenarios where elements need to be managed in a first-in-first-out (FIFO) manner.

**Code:**Top of Form

import java.util.Scanner;

class Node {

int data;

Node next;

Node(int value) {

data = value;

next = null;

}

}

public class QueueUsingLinkedList {

static Node front;

static Node rear;

static Scanner scanner = new Scanner(System.in); // Declared scanner as static

public static void main(String[] args) {

int choice;

while (true) {

System.out.println("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("=================================================================");

System.out.println("1. Insert element(s)\n2. Delete an element\n3. Display the queue\n4. Exit");

System.out.print("Enter your choice: ");

choice = scanner.nextInt();

switch (choice) {

case 1:

insert();

break;

case 2:

delete();

break;

case 3:

display();

break;

case 4:

System.exit(0);

default:

System.out.println("\nEnter valid choice??");

}

}

}

static void insert() {

int itemCount;

System.out.print("Enter the number of elements to insert: ");

itemCount = scanner.nextInt();

System.out.println("Enter " + itemCount + " element(s) to insert:");

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

int item = scanner.nextInt();

Node newNode = new Node(item);

if (newNode == null) {

System.out.println("\nOVERFLOW");

return;

} else {

if (front == null) {

front = newNode;

rear = newNode;

front.next = null;

rear.next = null;

} else {

rear.next = newNode;

rear = newNode;

rear.next = null;

}

}

}

System.out.println(itemCount + " element(s) inserted successfully.");

}

static void delete() {

Node ptr;

if (front == null) {

System.out.println("\nUNDERFLOW");

return;

} else {

ptr = front;

front = front.next;

ptr = null; // Freeing the memory

System.out.println("Element deleted successfully.");

}

}

static void display() {

Node ptr = front;

if (front == null) {

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

} else {

System.out.println("\nPrinting values .....");

while (ptr != null) {

System.out.println(ptr.data);

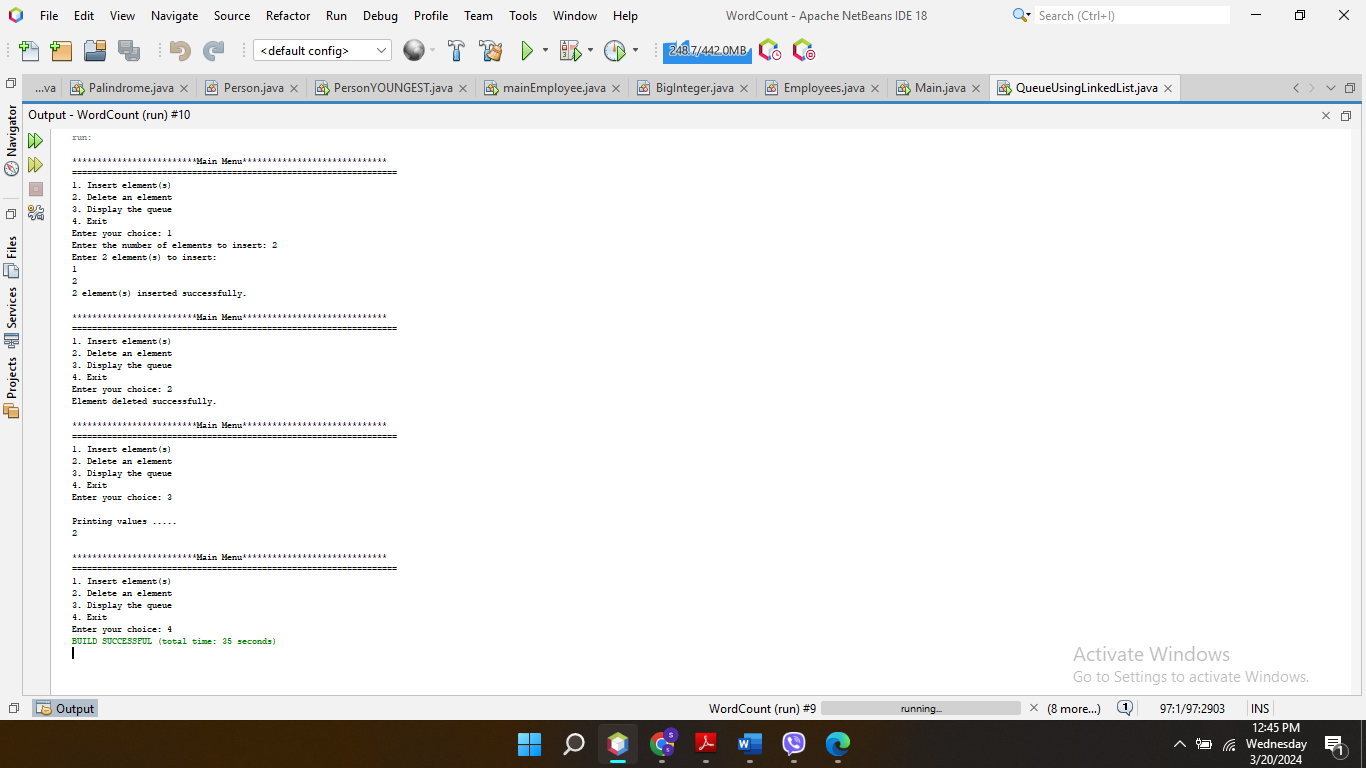
ptr = ptr.next;

}

}

}

}



**Conclusion:**

In conclusion, the linked list implementation of a queue provides an efficient and flexible way to manage data in a FIFO manner. By leveraging the structure of a linked list, it offers constant-time insertion and deletion operations. This approach is beneficial for scenarios where the size of the queue may vary dynamically, and memory allocation needs to be adjusted accordingly.

**Q5. Binary search tree**

A binary search tree (BST) is a hierarchical data structure where each node has at most two children: a left child and a right child. The key characteristic of a BST is the ordering property, where all nodes in the left subtree have values less than the current node, and all nodes in the right subtree have values greater than the current node. This property enables efficient searching, insertion, deletion, and traversal operations. Searching involves recursively traversing the tree to find the target value. Insertion involves traversing the tree to find the appropriate position for the new node and adding it while maintaining the ordering property. Deletion requires identifying the node to remove and rearranging the tree structure if necessary to maintain the ordering property. Traversal operations, including inorder, preorder, and postorder traversal, involve visiting all nodes in specific orders to process or display their values.

Operations:

1. Searching: Recursively traverse the tree to find the target value.
2. Insertion: Traverse the tree to find the appropriate position for the new node and add it while maintaining the ordering property.
3. Deletion: Identify the node to remove and rearrange the tree structure if necessary to maintain the ordering property.
4. Traversal: Visit all nodes in specific orders, such as inorder, preorder, and postorder, to process or display their values.

**Java code:**

import java.util.Scanner;

class Node {

    int key;

    Node left, right;

    public Node(int item) {

        key = item;

        left = right = null;

    }

}

public class BinarySearchTree {

    Node root;

    BinarySearchTree() {

        root = null;

    }

    void insert(int key) {

        root = insertRec(root, key);

    }

    Node insertRec(Node root, int key) {

        if (root == null) {

            root = new Node(key);

            return root;

        }

        if (key < root.key)

            root.left = insertRec(root.left, key);

        else if (key > root.key)

            root.right = insertRec(root.right, key);

        return root;

    }

    void inorder() {

        inorderRec(root);

    }

    void inorderRec(Node root) {

        if (root != null) {

            inorderRec(root.left);

            System.out.print(root.key + " ");

            inorderRec(root.right);

        }

    }

    Node search(Node root, int key) {

        if (root == null || root.key == key)

            return root;

        if (key < root.key)

            return search(root.left, key);

        return search(root.right, key);

    }

    Node minValueNode(Node node) {

        Node current = node;

        while (current.left != null)

            current = current.left;

        return current;

    }

    Node deleteNode(Node root, int key) {

        if (root == null)

            return root;

        if (key < root.key)

            root.left = deleteNode(root.left, key);

        else if (key > root.key)

            root.right = deleteNode(root.right, key);

        else {

            if (root.left == null)

                return root.right;

            else if (root.right == null)

                return root.left;

            root.key = minValueNode(root.right).key;

            root.right = deleteNode(root.right, root.key);

        }

        return root;

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        BinarySearchTree tree = new BinarySearchTree();

        while (true) {

            System.out.println("\nBinary Search Tree Operations:");

            System.out.println("1. Insert");

            System.out.println("2. Search");

            System.out.println("3. Delete");

            System.out.println("4. Inorder Traversal");

            System.out.println("5. Exit");

            System.out.print("Enter your choice: ");

            int choice = scanner.nextInt();

            switch (choice) {

                case 1:

                    System.out.print("Enter key to insert: ");

                    int insertKey = scanner.nextInt();

                    tree.insert(insertKey);

                    break;

                case 2:

                    System.out.print("Enter key to search: ");

                    int searchKey = scanner.nextInt();

                    Node searchResult = tree.search(tree.root, searchKey);

                    if (searchResult != null)

                        System.out.println("Key found");

                    else

                        System.out.println("Key not found");

                    break;

                case 3:

                    System.out.print("Enter key to delete: ");

                    int deleteKey = scanner.nextInt();

                    tree.root = tree.deleteNode(tree.root, deleteKey);

                    break;

                case 4:

                    System.out.print("Inorder Traversal: ");

                    tree.inorder();

                    break;

                case 5:

                    System.out.println("Exiting program");

                    System.exit(0);

                default:

                    System.out.println("Invalid choice");

            }

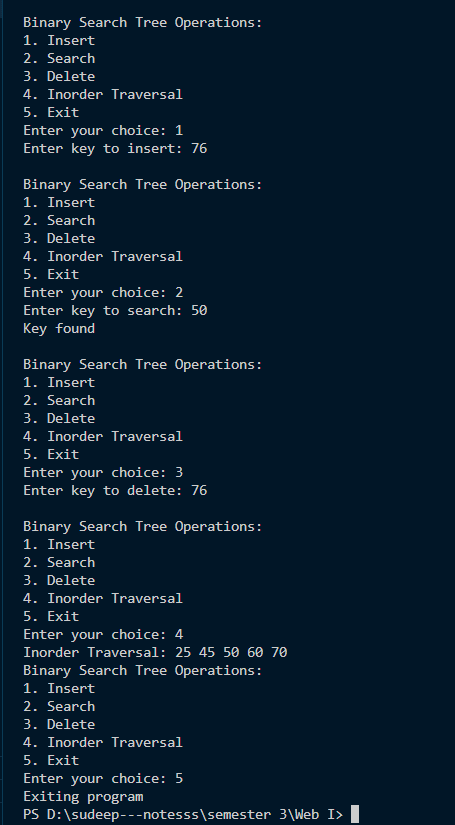
        }

    }

}

**Conclusion:**

In conclusion, binary search trees (BSTs) offer an efficient and ordered way to store and manipulate data. Their hierarchical structure and ordering property facilitate fast searching, insertion, deletion, and traversal operations. By maintaining the ordering property, BSTs ensure that data is organized in a balanced manner, optimizing performance for various tasks in computer science and software development. Despite their effectiveness, it's important to consider potential challenges such as maintaining balance in the tree to prevent performance degradation in certain scenarios. Overall, BSTs remain a fundamental and versatile data structure, widely utilized in diverse applications ranging from databases to algorithm design.



**Q5. Graph Representation**

**Theory:**

A graph serves as a fundamental structure comprising vertices and edges, providing a versatile representation for modeling relationships between entities in diverse systems, ranging from social networks to transportation networks and computational models. Graphs facilitate the analysis of connectivity, paths, and interactions among elements within a network. Different graph representations offer varying trade-offs in terms of memory efficiency, traversal speed, and suitability for different types of graphs.

Graph representation refers to the method or technique used to represent a graph data structure in computer memory. In a graph, vertices (nodes) are connected by edges (links), and the representation method determines how these vertices and edges are stored and organized in memory. Different representation techniques, such as adjacency lists, adjacency matrices, and incidence matrices, offer varying trade-offs in terms of memory usage, computational efficiency, and suitability for different types of graphs and algorithms. The choice of representation depends on factors such as the size and density of the graph, memory constraints, and the specific operations to be performed on the graph.

1. Adjacency List: This representation involves each vertex holding a list of adjacent vertices. It's advantageous for sparse graphs due to its memory efficiency. Traversal of the adjacency list enables swift access to neighboring vertices, facilitating various operations such as graph traversal and pathfinding algorithms.
2. Adjacency Matrix: Here, a 2D array signifies edge presence between vertices. Best suited for dense graphs, it offers constant-time edge checks. However, its memory usage increases quadratically with the number of vertices, making it less efficient for large sparse graphs.
3. Incidence Matrix: Employing a 2D array where rows represent vertices and columns represent edges, this representation indicates vertex-edge connections. It's useful for graphs with attributes but is less common due to its higher memory consumption compared to other representations. Despite this, it offers a structured view of vertex-edge relationships, which can be beneficial in certain analytical contexts.

**Java Code:**

import java.util.\*;

public class Graph {

    private int V; // Number of vertices

    private LinkedList<Integer> adj[]; // Adjacency list representation

    // Constructor

    public Graph(int v) {

        V = v;

        adj = new LinkedList[v];

        for (int i = 0; i < v; ++i)

            adj[i] = new LinkedList();

    }

    // Function to add an edge into the graph

    public void addEdge(int v, int w) {

        adj[v].add(w); // Add w to v's list.

    }

    // Function to print the adjacency list representation of the graph

    public void printGraph() {

        for (int i = 0; i < V; ++i) {

            System.out.print("Adjacency list of vertex " + i + ": ");

            for (Integer node : adj[i]) {

                System.out.print(" -> " + node);

            }

            System.out.println();

        }

    }

    public static void main(String args[]) {

        Graph graph = new Graph(5); // Creating a graph with 5 vertices

        graph.addEdge(0, 1);

        graph.addEdge(0, 4);

        graph.addEdge(1, 2);

        graph.addEdge(1, 3);

        graph.addEdge(1, 4);

        graph.addEdge(2, 3);

        graph.addEdge(3, 4);

        // Print the adjacency list representation of the graph

        graph.printGraph();

    }

}

A screenshot of a computer code

Description automatically generated

**Conclusion:**

In conclusion, graph representation is a crucial aspect of graph theory and computer science, enabling the efficient storage and manipulation of graphs in computer memory. By selecting the appropriate representation technique, such as adjacency lists, adjacency matrices, or incidence matrices, developers can optimize memory usage and computational efficiency while effectively capturing the relationships and connections within a graph. Understanding the strengths and limitations of each representation method is essential for designing efficient algorithms and solving various graph-related problems in diverse fields, including networking, social media analysis, logistics, and more.

**Q.6 SPANNING TREE AND SHORTEST PATH ALGORITHMS**

1. **Spanning Tree**

**Theory:**

● A spanning tree is a subset of Graph G, which has all the vertices covered with a minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected..

● By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree.

● A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.

**Minimum Spanning Tree (MST)**

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

**Minimum Spanning-Tree Algorithm**

We shall learn about two most important spanning tree algorithms here −

Kruskal's Algorithm

Prim's Algorithm

Both are greedy algorithms

1. **KRUSKAL’S ALGORITHM**

**THEORY**

● Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph.

● The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum.

**CODE USING JAVA**

import java.util.\*;

// Class to represent an edge in the graph

class Edge implements Comparable<Edge> {

int src, dest, weight;

// Comparator to sort edges based on their weights

public int compareTo(Edge compareEdge) {

return this.weight - compareEdge.weight;

}

};

// Class to represent a subset for union-find

class Subset {

int parent, rank;

};

public class KruskalsAlgorithm {

int V, E; // V-> no. of vertices & E->no.of edges

Edge edge[]; // collection of all edges

// Creates a graph with V vertices and E edges

KruskalsAlgorithm(int v, int e) {

V = v;

E = e;

edge = new Edge[E];

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

edge[i] = new Edge();

}

// A utility function to find set of an element i

// (uses path compression technique)

int find(Subset subsets[], int i) {

if (subsets[i].parent != i)

subsets[i].parent = find(subsets, subsets[i].parent);

return subsets[i].parent;

}

// A function that does union of two sets of x and y

// (uses union by rank)

void Union(Subset subsets[], int x, int y) {

int xroot = find(subsets, x);

int yroot = find(subsets, y);

// Attach smaller rank tree under root of high rank tree

// (Union by Rank)

if (subsets[xroot].rank < subsets[yroot].rank)

subsets[xroot].parent = yroot;

else if (subsets[xroot].rank > subsets[yroot].rank)

subsets[yroot].parent = xroot;

// If ranks are the same, then make one as root and increment its rank by one

else {

subsets[yroot].parent = xroot;

subsets[xroot].rank++;

}

}

// The main function to construct MST using Kruskal's algorithm

void KruskalMST() {

Edge result[] = new Edge[V]; // This will store the resultant MST

int e = 0; // An index variable, used for result[]

int i = 0; // An index variable, used for sorted edges

for (i = 0; i < V; ++i)

result[i] = new Edge();

// Step 1: Sort all the edges in non-decreasing order of their weight.

Arrays.sort(edge);

// Allocate memory for creating V subsets

Subset subsets[] = new Subset[V];

for (i = 0; i < V; ++i)

subsets[i] = new Subset();

// Create V subsets with single elements

for (int v = 0; v < V; ++v) {

subsets[v].parent = v;

subsets[v].rank = 0;

}

i = 0; // Initialize index for the edges[] array

// Number of edges to be taken is equal to V-1

while (e < V - 1) {

// Step 2: Pick the smallest edge. Increment index for the next iteration

Edge next\_edge = new Edge();

next\_edge = edge[i++];

int x = find(subsets, next\_edge.src);

int y = find(subsets, next\_edge.dest);

// If including this edge doesn't cause a cycle, include it in the result and increment the index of result for next edge

if (x != y) {

result[e++] = next\_edge;

Union(subsets, x, y);

}

// Else discard the next\_edge

}

// Print the contents of result[] to display the built MST

System.out.println("Following are the edges in the constructed MST:");

for (i = 0; i < e; ++i)

System.out.println(result[i].src + " -- " + result[i].dest + " == " + result[i].weight);

}

public static void main(String[] args) {

/\* Let us create following weighted graph

10

0--------1

| \ |

6| 5\ |15

| \ |

2--------3

4 \*/

int V = 4; // Number of vertices in the graph

int E = 5; // Number of edges in the graph

KruskalsAlgorithm graph = new KruskalsAlgorithm(V, E);

// add edge 0-1

graph.edge[0].src = 0;

graph.edge[0].dest = 1;

graph.edge[0].weight = 10;

// add edge 0-2

graph.edge[1].src = 0;

graph.edge[1].dest = 2;

graph.edge[1].weight = 6;

// add edge 0-3

graph.edge[2].src = 0;

graph.edge[2].dest = 3;

graph.edge[2].weight = 5;

// add edge 1-3

graph.edge[3].src = 1;

graph.edge[3].dest = 3;

graph.edge[3].weight = 15;

// add edge 2-3

graph.edge[4].src = 2;

graph.edge[4].dest = 3;

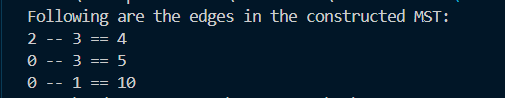
graph.edge[4].weight = 4;

// Function call

graph.KruskalMST();

}

}

****

**CONCLUSION**

Kruskal's algorithm, a fundamental approach in graph theory and optimization, efficiently finds the minimum spanning tree (MST) of a connected, weighted graph. Through a systematic process of adding edges in ascending order of weight while avoiding cycles, Kruskal's algorithm constructs an MST that spans all vertices with the least total edge weight. Its simplicity, effectiveness, and time complexity of O(E log E) or O(E log V) make it a preferred choice for many applications, including network design, circuit layout, and clustering analysis. With its ability to handle sparse graphs and its straightforward implementation, Kruskal's algorithm stands as a cornerstone in the arsenal of graph algorithms, facilitating optimal solutions in various domains.

1. **PRIM’S ALGORITHM**

**THEORY**

● Prim's Algorithm is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimised.

● Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

**CODE USING JAVA**

import java.util.\*;

class PrimAlgorithm {

// Number of vertices in the graph

private static final int V = 5;

// Function to find the vertex with minimum key value,

// from the set of vertices not yet included in MST

private int minKey(int[] key, boolean[] mstSet) {

int min = Integer.MAX\_VALUE, minIndex = -1;

for (int v = 0; v < V; v++) {

if (!mstSet[v] && key[v] < min) {

min = key[v];

minIndex = v;

}

}

return minIndex;

}

// Function to print the constructed MST stored in parent[]

private void printMST(int[] parent, int[][] graph) {

System.out.println("Edge \tWeight");

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

System.out.println(parent[i] + " - " + i + "\t" + graph[i][parent[i]]);

}

}

// Function to construct and print MST for a graph represented

// using adjacency matrix representation

void primMST(int[][] graph) {

// Array to store constructed MST

int[] parent = new int[V];

// Key values used to pick minimum weight edge in cut

int[] key = new int[V];

// To represent set of vertices not yet included in MST

boolean[] mstSet = new boolean[V];

// Initialize all keys as INFINITE

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

key[i] = Integer.MAX\_VALUE;

mstSet[i] = false;

}

// Always include first vertex in MST.

// Make key 0 so that this vertex is picked as first vertex

key[0] = 0;

parent[0] = -1; // First node is always root of MST

// The MST will have V vertices

for (int count = 0; count < V - 1; count++) {

// Pick the minimum key vertex from the set of vertices

// not yet included in MST

int u = minKey(key, mstSet);

// Add the picked vertex to the MST Set

mstSet[u] = true;

// Update key value and parent index of the adjacent vertices

// of the picked vertex. Consider only those vertices which are

// not yet included in MST

for (int v = 0; v < V; v++) {

// graph[u][v] is non zero only for adjacent vertices of m

// mstSet[v] is false for vertices not yet included in MST

// Update the key only if graph[u][v] is smaller than key[v]

if (graph[u][v] != 0 && !mstSet[v] && graph[u][v] < key[v]) {

parent[v] = u;

key[v] = graph[u][v];

}

}

}

// print the constructed MST

printMST(parent, graph);

}

public static void main(String[] args) {

/\* Let us create the following graph

2 3

(0)--(1)--(2)

| / \ |

6| 8/ \5 |7

| / \ |

(3)-------(4)

9 \*/

PrimAlgorithm t = new PrimAlgorithm();

int[][] graph = new int[][]{{0, 2, 0, 6, 0},

{2, 0, 3, 8, 5},

{0, 3, 0, 0, 7},

{6, 8, 0, 0, 9},

{0, 5, 7, 9, 0}};

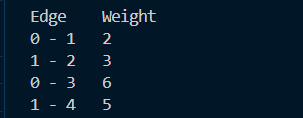
// Print the solution

t.primMST(graph);

}

}

**OUTPUT**

****

**CONCLUSION**

Prim's algorithm is a fundamental method for finding the minimum spanning tree (MST) of a weighted undirected graph. Through a systematic process of selecting edges with the lowest weights while ensuring connectivity and avoiding cycles, Prim's algorithm efficiently constructs an MST. Its simplicity, clarity, and effectiveness make it a widely used tool in various applications, including network design, transportation planning, and circuit layout. With a time complexity of O(V^2) or O(E log V) depending on the implementation, Prim's algorithm strikes a balance between computational efficiency and simplicity. Overall, Prim's algorithm stands as a cornerstone in graph theory and optimization, offering a versatile solution to the problem of finding the most cost-effective connections within a graph.

**B) Shortest-Path Problems:**

**Types, Single-Source Shortest path problem-Dijkestra’s Algorithm.**

**Shortest Path Problem**

In data structures,

● Shortest path problem is a problem of finding the shortest path(s) between vertices of a given graph.

● Shortest path between two vertices is a path that has the least cost as compared to all other existing paths.

**Shortest Path Algorithms**

Shortest path algorithms are a family of algorithms used for solving the shortest path problem.

**Applications**

Shortest path algorithms have a wide range of applications such as in-

● Google Maps

● Road Networks

● Logistics Research

**Types of Shortest Path Problem**

Various types of shortest path problem are

1. Single-pair shortest path problem

2. Single-source shortest path problem

3. Single-destination shortest path problem

4. All pairs shortest path problem

1. **Dijkstra Algorithm**

**THEORY**

● The Dijkstra Algorithm is a very famous greedy algorithm.

● It is used for solving the single source shortest path problem.

● It computes the shortest path from one particular source node to all other remaining nodes of graph.

**CODE USING JAVA**

import java.util.\*;

class PrimAlgorithm {

// Number of vertices in the graph

private static final int V = 5;

// Function to find the vertex with minimum key value,

// from the set of vertices not yet included in MST

private int minKey(int[] key, boolean[] mstSet) {

int min = Integer.MAX\_VALUE, minIndex = -1;

for (int v = 0; v < V; v++) {

if (!mstSet[v] && key[v] < min) {

min = key[v];

minIndex = v;

}

}

return minIndex;

}

// Function to print the constructed MST stored in parent[]

private void printMST(int[] parent, int[][] graph) {

System.out.println("Edge \tWeight");

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

System.out.println(parent[i] + " - " + i + "\t" + graph[i][parent[i]]);

}

}

// Function to construct and print MST for a graph represented

// using adjacency matrix representation // Key values used to pick minimum weight edge in cut

int[] key = new int[V];

// To represent set of vertices not yet in

void primMST(int[][] graph) {

// Array to store constructed MST

int[] parent = new int[V];

cluded in MST

boolean[] mstSet = new boolean[V];

// Initialize all keys as INFINITE

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

key[i] = Integer.MAX\_VALUE;

mstSet[i] = false;

}

// Always include first vertex in MST.

// Make key 0 so that this vertex is picked as first vertex

key[0] = 0;

parent[0] = -1; // First node is always root of MST

// The MST will have V vertices

for (int count = 0; count < V - 1; count++) {

// Pick the minimum key vertex from the set of vertices

// not yet included in MST

int u = minKey(key, mstSet);

// Add the picked vertex to the MST Set

mstSet[u] = true;

// Update key value and parent index of the adjacent vertices

// of the picked vertex. Consider only those vertices which are

// not yet included in MST

for (int v = 0; v < V; v++) {

// graph[u][v] is non zero only for adjacent vertices of m

// mstSet[v] is false for vertices not yet included in MST

// Update the key only if graph[u][v] is smaller than key[v]

if (graph[u][v] != 0 && !mstSet[v] && graph[u][v] < key[v]) {

parent[v] = u;

key[v] = graph[u][v];

}

}

}

// print the constructed MST

printMST(parent, graph);

}

public static void main(String[] args) {

/\* Let us create the following graph

2 3

(0)--(1)--(2)

| / \ |

6| 8/ \5 |7

| / \ |

(3)-------(4)

9 \*/

PrimAlgorithm t = new PrimAlgorithm();

int[][] graph = new int[][]{{0, 2, 0, 6, 0},

{2, 0, 3, 8, 5},

{0, 3, 0, 0, 7},

{6, 8, 0, 0, 9},

{0, 5, 7, 9, 0}};

// Print the solution

t.primMST(graph);

}

}

**A screen shot of a computer

Description automatically generated**

**CONCLUSION**

In conclusion, Dijkstra's algorithm stands as a cornerstone in graph theory and algorithm design, offering an elegant solution for finding shortest paths in weighted graphs. Its simplicity of implementation and efficiency make it indispensable for a wide array of applications, from network routing to GPS navigation. While its limitations, such as the assumption of non-negative edge weights, must be acknowledged, its enduring impact on the field of algorithms and its practical utility remain undeniable. As such, Dijkstra's algorithm continues to serve as a fundamental tool for computer scientists and programmers alike, solving shortest path problems with precision and effectiveness.

**Q.8 Sorting, Searching and Hashing Algorithm**

1. **Sorting**

**Theory:**

The arrangement of data in a preferred order is called sorting in the data Structure . By sorting data, it is easier to search through it quickly and easily. The simplest example of sorting is a dictionary. There are different types of sorting. i.e.

1. Bubble Sort: It’s a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they’re in the wrong order.
2. Insertion Sort: In Insertion Sorting, the dataset is virtually split into a sorted and an unsorted part, then the algorithm picks up the elements from the unsorted part and places them at the correct position in the sorted part.
3. Selection Sort: In Selection Sorting, we move along the data and select the smallest item, swap the selected item to the position 0,and so on.
4. Quick Sort: This sorting algorithm picks up a pivot element, then partitions the dataset into two sub-arrays, one sub-array is greater than the element and another sub-array is less than the element.
5. Merge Sort: It divides the dataset into smaller sub-arrays , sorts them individually, and then merges them to obtain the final sorted result.
6. Heap Sort: Heap Sorting involves building a Heap data structure from the given array and then utilizing the Heap to sort the array.
7. Radix Sort: It sorts the numbers by sorting each digit from left to right, resulting in the sorted data. Its time complexity is O(d\*(n+b)).
8. **Insertion sort**

**Java code:**

public class InsertionSort {

static void insertionSort(int arr[], int n) {

int i, key, j;

for (i = 1; i < n; i++) {

key = arr[i];

j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

}

static void printArray(int arr[], int n) {

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

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

System.out.println();

}

public static void main(String args[]) {

int arr[] = { 12, 11, 13, 5, 6 };

int N = arr.length;

insertionSort(arr, N);

printArray(arr, N);

}

}

Output:

InsertionSorting.png

1. **Selection sort**

import java.util.Arrays;

public class SelectionSort {

// Function for Selection sort

static void selectionSort(int arr[], int n) {

int i, j, min\_idx;

// One by one move boundary of

// unsorted subarray

for (i = 0; i < n - 1; i++) {

// Find the minimum element in

// unsorted array

min\_idx = i;

for (j = i + 1; j < n; j++) {

if (arr[j] < arr[min\_idx])

min\_idx = j;

}

// Swap the found minimum element

// with the first element

if (min\_idx != i) {

int temp = arr[min\_idx];

arr[min\_idx] = arr[i];

arr[i] = temp;

}

}

}

// Function to print an array

static void printArray(int arr[], int size) {

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

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

}

System.out.println();

}

// Driver program

public static void main(String[] args) {

int arr[] = { 64, 25, 12, 22, 11 };

int n = arr.length;

// Function Call

selectionSort(arr, n);

System.out.println("The Sorted Array: ");

printArray(arr, n);

}

}

SelectionSOrt.png

1. **Bubble sort**

import java.util.Arrays;

public class BubbleSort {

// An optimized version of Bubble Sort

static void bubbleSort(int arr[], int n) {

int i, j;

boolean swapped;

for (i = 0; i < n - 1; i++) {

swapped = false;

for (j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

// Swap elements if they are in the wrong order

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

swapped = true;

}

}

// If no two elements were swapped by inner loop, then break

if (!swapped)

break;

}

}

// Function to print an array

static void printArray(int arr[], int size) {

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

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

}

// Driver program to test above functions

public static void main(String args[]) {

int arr[] = { 64, 34, 25, 12, 22, 11, 90 };

int N = arr.length;

bubbleSort(arr, N);

System.out.println("Sorted array:");

printArray(arr, N);

}

}

Output:

Bubble Sort.png

1. **Quick Sort**

import java.util.Arrays;

public class QuickSort {

// Function to partition the array and return the index of the pivot

static int partition(int arr[], int low, int high) {

// Choose the pivot

int pivot = arr[high];

// Index of the smaller element and indicates

// the right position of the pivot found so far

int i = low - 1;

for (int j = low; j <= high; j++) {

// If the current element is smaller than the pivot

if (arr[j] < pivot) {

// Increment index of the smaller element

i++;

// Swap arr[i] and arr[j]

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

// Swap arr[i+1] and arr[high] (pivot)

int temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

return i + 1;

}

// The Quicksort function implementation

static void quickSort(int arr[], int low, int high) {

// When low is less than high

if (low < high) {

// pi is the partition index, arr[pi] is now at the right place

int pi = partition(arr, low, high);

// Recursion call

// Smaller elements than the pivot go to the left

// and higher elements go to the right

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

public static void main(String[] args) {

int arr[] = { 10, 7, 8, 9, 1, 5 };

int n = arr.length;

// Function call

quickSort(arr, 0, n - 1);

// Print the sorted array

System.out.println("Sorted Array:");

System.out.println(Arrays.toString(arr));

}

}

QuickSort.png

1. **Merge sort**

public class MergeSort {

// Function to merge the subarrays of arr[]

void merge(int arr[], int beg, int mid, int end) {

int n1 = mid - beg + 1;

int n2 = end - mid;

int LeftArray[] = new int[n1];

int RightArray[] = new int[n2];

// Copy data to temporary arrays

for (int i = 0; i < n1; ++i)

LeftArray[i] = arr[beg + i];

for (int j = 0; j < n2; ++j)

RightArray[j] = arr[mid + 1 + j];

int i = 0, j = 0;

int k = beg;

while (i < n1 && j < n2) {

if (LeftArray[i] <= RightArray[j]) {

arr[k] = LeftArray[i];

i++;

} else {

arr[k] = RightArray[j];

j++;

}

k++;

}

// Copy remaining elements of LeftArray[], if there are any

while (i < n1) {

arr[k] = LeftArray[i];

i++;

k++;

}

// Copy remaining elements of RightArray[], if there are any

while (j < n2) {

arr[k] = RightArray[j];

j++;

k++;

}

}

// Main function that sorts arr[l..r] using merge()

void mergeSort(int arr[], int beg, int end) {

if (beg < end) {

// Find the middle point

int mid = (beg + end) / 2;

// Sort first and second halves

mergeSort(arr, beg, mid);

mergeSort(arr, mid + 1, end);

// Merge the sorted halves

merge(arr, beg, mid, end);

}

}

// Function to print the array

void printArray(int arr[]) {

int n = arr.length;

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

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

System.out.println();

}

// Driver method

public static void main(String args[]) {

MergeSort ob = new MergeSort();

int arr[] = { 12, 31, 25, 8, 32, 17, 40, 42 };

int n = arr.length;

System.out.println("Before sorting array elements are:");

ob.printArray(arr);

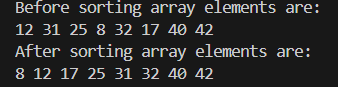
ob.mergeSort(arr, 0, n - 1);

System.out.println("After sorting array elements are:");

ob.printArray(arr);

}

}



1. **Radix sort**

import java.util.Arrays;

public class RadixSort {

// Function to get the maximum value from an array

static int getMax(int a[], int n) {

int max = a[0];

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

if (a[i] > max)

max = a[i];

}

return max;

}

// Function to perform counting sort based on the digit represented by the place parameter

static void countingSort(int a[], int n, int place) {

final int RANGE = 10;

int output[] = new int[n];

int count[] = new int[RANGE];

// Calculate count of elements

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

count[(a[i] / place) % RANGE]++;

// Calculate cumulative frequency

for (int i = 1; i < RANGE; i++)

count[i] += count[i - 1];

// Place the elements in sorted order

for (int i = n - 1; i >= 0; i--) {

output[count[(a[i] / place) % RANGE] - 1] = a[i];

count[(a[i] / place) % RANGE]--;

}

// Copy the output array to the original array

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

a[i] = output[i];

}

// Function to implement Radix Sort

static void radixSort(int a[], int n) {

int max = getMax(a, n);

// Apply counting sort to sort elements based on place value

for (int place = 1; max / place > 0; place \*= 10)

countingSort(a, n, place);

}

// Function to print array elements

static void printArray(int a[], int n) {

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

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

}

System.out.println();

}

public static void main(String[] args) {

int a[] = {181, 289, 390, 121, 145, 736, 514, 888, 122};

int n = a.length;

System.out.println("Before sorting, array elements are:");

printArray(a, n);

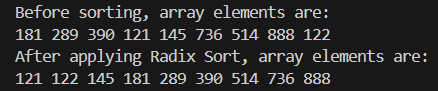
radixSort(a, n);

System.out.println("After applying Radix Sort, array elements are:");

printArray(a, n);

}

}



**Conclusion:**

In conclusion, sorting algorithms are essential tools in data structures and algorithms, enabling efficient data organization and retrieval. The variety of sorting techniques available allows for tailored implementations based on factors such as dataset size and data type. From classic comparison-based algorithms like quicksort to specialized approaches such as counting sort, the versatility of sorting methodologies ensures optimization for various scenarios. Additionally, advancements in parallel computing have led to the development of parallel sorting algorithms, addressing challenges posed by large-scale datasets and distributed environments. Thus, sorting algorithms remain integral to computational efficiency and data management across diverse contexts.

1. **SEARCHING ALGORITHM**

**Theory :**

Searching algorithms are vital tools in computer science for locating specific items within data collections. Three main types of searching algorithms exist:

* Linear Search: Also known as sequential search, this simple algorithm traverses the entire list, comparing each element with the target item until a match is found, returning its location, or NULL otherwise.
* Binary Search: Utilized in sorted arrays, binary search repeatedly divides the search interval in half, leveraging the sorted nature of the array to achieve a time complexity of O(log N).
* Interpolation Search: An enhancement over binary search, interpolation search is effective when sorted array values are uniformly distributed. It constructs new data points within the range of known data points, potentially leading to different search locations based on the value being sought.

1. **Linear searching**

import java.util.\*;

public class LinearSearch {

// Function for linear search

public static int search(int arr[], int x)

{

int n = arr.length;

// Traverse array arr[]

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

// If element found then

// return that index

if (arr[i] == x)

return i;

}

return -1;

}

// Driver Code

public static void main(String args[])

{

// Given arr[]

int arr[] = { 2, 3, 4, 10, 40 };

Scanner sc=new Scanner(System.in);

// Element to search

System.out.println("Enter the element you want to search:");

int x=sc.nextInt();

// Function Call

int result = search(arr, x);

if (result == -1)

System.out.print(

"Element is not present in array");

else

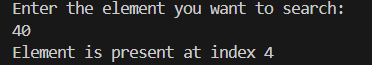
System.out.print("Element is present"

+ " at index "

+ result);

}

}



1. **Binary searching**

import java.util.Scanner;

public class BinarySearch {

// Function that returns index of

// x if it is present in arr[l, r]

int binarySearch(int arr[], int l,

int r, int x)

{

if (r >= l) {

int mid = l + (r - l) / 2;

// If the element is present

// at the middle itself

if (arr[mid] == x)

return mid;

// If element is smaller than

// mid, then it can only be

// present in left subarray

if (arr[mid] > x)

return binarySearch(arr, l,

mid - 1, x);

// Else the element can only be

// present in right subarray

return binarySearch(arr, mid + 1,

r, x);

}

// Reach here when element is

// not present in array

return -1;

}

// Driver Code

public static void main(String args[])

{

// Create object of this class

BinarySearch ob = new BinarySearch();

Scanner sc=new Scanner(System.in);

// Given array arr[]

int arr[] = { 2, 3, 4, 10, 40 };

int n = arr.length;

System.out.println("Enter the element you want to search:");

int x=sc.nextInt();

// Function Call

int result = ob.binarySearch(arr, 0,

n - 1, x);

if (result == -1)

System.out.println("Element "

+ "not present");

else

System.out.println("Element found"

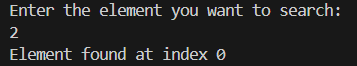
+ " at index "

+ result);

}

}

Output:



1. **Interpolation searching**

import java.util.Scanner;

public class InterpolationSearch {

public static int interpolationSearch(int[] arr, int target) {

int low = 0;

int high = arr.length - 1;

while (low <= high && target >= arr[low] && target <= arr[high]) {

// Calculate the position of the target element based on its value

int pos = low + (((target - arr[low]) \* (high - low)) / (arr[high] - arr[low]));

// Check if the target element is at the calculated position

if (arr[pos] == target) {

return pos;

}

// If the target element is less than the element at the

// calculated position, search the left half of the list

if (arr[pos] > target) {

high = pos - 1;

} else {

// If the target element is greater than the element

// at the calculated position, search the right half of the list

low = pos + 1;

}

}

return -1;

}

public static void main(String[] args) {

int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9};

Scanner sc=new Scanner(System.in);

System.out.println("Enter the element you want to search:");

int target=sc.nextInt();

int index = interpolationSearch(arr, target);

if (index == -1) {

System.out.println(target + " not found in the list");

} else {

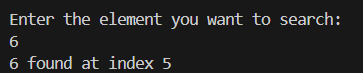
System.out.println(target + " found at index " + index);

}

}

}

Output:



**Conclusion:**

In conclusion, linear search is a basic search algorithm that can be useful in certain situations, such as when the list of elements is small, or when the list is unsorted. Binary search is a more efficient search algorithm for large, sorted lists.

1. **Hashing Algorithm**

**Theory:**

A hashing algorithm converts input into a fixed-size output, serving as an index in an array or hash table. The hash function ensures determinism, always producing the same result for a given input. Different hashing techniques include:

* Linear Probing: Used to resolve collisions in hash tables by linearly searching for the next available location if the hashed memory location is already filled. The table is considered circular for seamless traversal.
* Quadratic Probing: Similar to linear probing but utilizes a quadratic sequence (h, h+1, h+4, h+9...) to search for available locations, aiming to distribute values more evenly.
* Separate Chaining: Handles collisions by allowing multiple key-value pairs to be stored at the same index. Each index contains a linked list or another data structure holding the colliding pairs.
* Double Hashing: Resolves collisions by employing two hash functions to compute two different hash values for a key. The first function calculates the initial hash value, while the second determines the step size for the probing sequence, resulting in a lower collision rate compared to other methods.

1. **Linear Probing**

public class LinearProbingHashTable {

private static final int DEFAULT\_CAPACITY = 10;

private static final double LOAD\_FACTOR\_THRESHOLD = 0.7;

private Entry[] table;

private int size;

public LinearProbingHashTable() {

this(DEFAULT\_CAPACITY);

}

public LinearProbingHashTable(int initialCapacity) {

table = new Entry[initialCapacity];

size = 0;

}

private static class Entry {

Object key;

Object value;

boolean deleted;

Entry(Object key, Object value) {

this.key = key;

this.value = value;

this.deleted = false;

}

}

public void put(Object key, Object value) {

if ((double) size / table.length > LOAD\_FACTOR\_THRESHOLD) {

resize();

}

int index = hash(key);

while (table[index] != null && !table[index].deleted) {

if (table[index].key.equals(key)) {

table[index].value = value;

return;

}

index = (index + 1) % table.length; // Linear probing

}

table[index] = new Entry(key, value);

size++;

}

public Object get(Object key) {

int index = find(key);

return index != -1 ? table[index].value : null;

}

public void remove(Object key) {

int index = find(key);

if (index != -1) {

table[index].deleted = true;

size--;

}

}

private int find(Object key) {

int index = hash(key);

while (table[index] != null) {

if (!table[index].deleted && table[index].key.equals(key)) {

return index;

}

index = (index + 1) % table.length; // Linear probing

}

return -1;

}

private int hash(Object key) {

return Math.abs(key.hashCode()) % table.length;

}

private void resize() {

Entry[] oldTable = table;

table = new Entry[table.length \* 2];

size = 0;

for (Entry entry : oldTable) {

if (entry != null && !entry.deleted) {

put(entry.key, entry.value);

}

}

}

public static void main(String[] args) {

LinearProbingHashTable hashTable = new LinearProbingHashTable();

hashTable.put(1, "One");

hashTable.put(2, "Two");

hashTable.put(3, "Three");

hashTable.put(4, "Four");

hashTable.put(5, "Five");

hashTable.put(6, "Six");

hashTable.put(7, "Seven");

System.out.println("Value associated with key 3: " + hashTable.get(3));

hashTable.remove(3);

System.out.println("Value associated with key 3 after removal: " + hashTable.get(3));

}

}

Output:

Linearhashing.png

1. **Quadratic Probing**

public class QuadraticProbingHashTable {

private static final int DEFAULT\_CAPACITY = 10;

private HashEntry[] table;

private int size;

public QuadraticProbingHashTable() {

table = new HashEntry[DEFAULT\_CAPACITY];

size = 0;

}

public QuadraticProbingHashTable(int capacity) {

table = new HashEntry[capacity];

size = 0;

}

public void insert(int key, String value) {

if (size == table.length)

throw new IllegalStateException("Hash table is full");

int hash = hash(key);

int index = hash % table.length;

int originalIndex = index;

int i = 1;

while (table[index] != null && !table[index].isDeleted()) {

index = (originalIndex + i \* i) % table.length;

i++;

}

table[index] = new HashEntry(key, value);

size++;

}

public String search(int key) {

int hash = hash(key);

int index = hash % table.length;

int originalIndex = index;

int i = 1;

while (table[index] != null) {

if (table[index].getKey() == key && !table[index].isDeleted())

return table[index].getValue();

index = (originalIndex + i \* i) % table.length;

i++;

}

return null;

}

public void remove(int key) {

int hash = hash(key);

int index = hash % table.length;

int originalIndex = index;

int i = 1;

while (table[index] != null) {

if (table[index].getKey() == key && !table[index].isDeleted()) {

table[index].setDeleted(true);

size--;

return;

}

index = (originalIndex + i \* i) % table.length;

i++;

}

}

private int hash(int key) {

return key % table.length;

}

private static class HashEntry {

private int key;

private String value;

private boolean deleted;

public HashEntry(int key, String value) {

this.key = key;

this.value = value;

this.deleted = false;

}

public int getKey() {

return key;

}

public String getValue() {

return value;

}

public boolean isDeleted() {

return deleted;

}

public void setDeleted(boolean deleted) {

this.deleted = deleted;

}

}

public static void main(String[] args) {

QuadraticProbingHashTable hashTable = new QuadraticProbingHashTable();

hashTable.insert(1, "Value1");

hashTable.insert(2, "Value2");

hashTable.insert(3, "Value3");

hashTable.insert(4, "Value4");

System.out.println("Value for key 1: " + hashTable.search(1));

System.out.println("Value for key 2: " + hashTable.search(2));

System.out.println("Value for key 3: " + hashTable.search(3));

System.out.println("Value for key 4: " + hashTable.search(4));

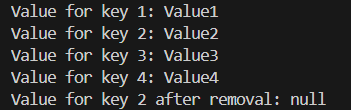
hashTable.remove(2);

System.out.println("Value for key 2 after removal: " + hashTable.search(2));

}

}

Output:



1. **Separate Chaining**

import java.util.ArrayList;

import java.util.Objects;

// A node of chains

class HashNode<K, V> {

K key;

V value;

final int hashCode;

// Reference to next node

HashNode<K, V> next;

// Constructor

public HashNode(K key, V value, int hashCode)

{

this.key = key;

this.value = value;

this.hashCode = hashCode;

}

}

// Class to represent entire hash table

class Map<K, V> {

// bucketArray is used to store array of chains

private ArrayList<HashNode<K, V> > bucketArray;

// Current capacity of array list

private int numBuckets;

// Current size of array list

private int size;

// Constructor (Initializes capacity, size and

// empty chains.

public Map()

{

bucketArray = new ArrayList<>();

numBuckets = 10;

size = 0;

// Create empty chains

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

bucketArray.add(null);

}

public int size() { return size; }

public boolean isEmpty() { return size() == 0; }

private final int hashCode (K key) {

return Objects.hashCode(key);

}

// This implements hash function to find index

// for a key

private int getBucketIndex(K key)

{

int hashCode = hashCode(key);

int index = hashCode % numBuckets;

// key.hashCode() could be negative.

index = index < 0 ? index \* -1 : index;

return index;

}

// Method to remove a given key

public V remove(K key)

{

// Apply hash function to find index for given key

int bucketIndex = getBucketIndex(key);

int hashCode = hashCode(key);

// Get head of chain

HashNode<K, V> head = bucketArray.get(bucketIndex);

// Search for key in its chain

HashNode<K, V> prev = null;

while (head != null) {

// If Key found

if (head.key.equals(key) && hashCode == head.hashCode)

break;

// Else keep moving in chain

prev = head;

head = head.next;

}

// If key was not there

if (head == null)

return null;

// Reduce size

size--;

// Remove key

if (prev != null)

prev.next = head.next;

else

bucketArray.set(bucketIndex, head.next);

return head.value;

}

// Returns value for a key

public V get(K key)

{

// Find head of chain for given key

int bucketIndex = getBucketIndex(key);

int hashCode = hashCode(key);

HashNode<K, V> head = bucketArray.get(bucketIndex);

// Search key in chain

while (head != null) {

if (head.key.equals(key) && head.hashCode == hashCode)

return head.value;

head = head.next;

}

// If key not found

return null;

}

// Adds a key value pair to hash

public void add(K key, V value)

{

// Find head of chain for given key

int bucketIndex = getBucketIndex(key);

int hashCode = hashCode(key);

HashNode<K, V> head = bucketArray.get(bucketIndex);

// Check if key is already present

while (head != null) {

if (head.key.equals(key) && head.hashCode == hashCode) {

head.value = value;

return;

}

head = head.next;

}

// Insert key in chain

size++;

head = bucketArray.get(bucketIndex);

HashNode<K, V> newNode

= new HashNode<K, V>(key, value, hashCode);

newNode.next = head;

bucketArray.set(bucketIndex, newNode);

// If load factor goes beyond threshold, then

// double hash table size

if ((1.0 \* size) / numBuckets >= 0.7) {

ArrayList<HashNode<K, V> > temp = bucketArray;

bucketArray = new ArrayList<>();

numBuckets = 2 \* numBuckets;

size = 0;

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

bucketArray.add(null);

for (HashNode<K, V> headNode : temp) {

while (headNode != null) {

add(headNode.key, headNode.value);

headNode = headNode.next;

}

}

}

}

// Driver method to test Map class

public static void main(String[] args)

{

Map<String, Integer> map = new Map<>();

map.add("this", 1);

map.add("coder", 2);

map.add("this", 4);

map.add("hi", 5);

System.out.println(map.size());

System.out.println(map.remove("this"));

System.out.println(map.remove("this"));

System.out.println(map.size());

System.out.println(map.isEmpty());

}

}

Output:



1. **Double Hashing**

public class DoubleHashingHashTable {

private static final int TABLE\_SIZE = 10;

private Entry[] table;

public DoubleHashingHashTable() {

table = new Entry[TABLE\_SIZE];

for (int i = 0; i < TABLE\_SIZE; i++) {

table[i] = null;

}

}

static class Entry {

String key;

String value;

Entry(String key, String value) {

this.key = key;

this.value = value;

}

}

// Hash function 1

private int hash1(String key) {

return Math.abs(key.hashCode()) % TABLE\_SIZE;

}

// Hash function 2

private int hash2(String key) {

// Prime number less than TABLE\_SIZE

int prime = 7;

return prime - (Math.abs(key.hashCode()) % prime);

}

// Insert key-value pair into the hash table

public void insert(String key, String value) {

int index = hash1(key);

int stepSize = hash2(key);

while (table[index] != null) {

index = (index + stepSize) % TABLE\_SIZE;

}

table[index] = new Entry(key, value);

}

// Get value associated with the key

public String get(String key) {

int index = hash1(key);

int stepSize = hash2(key);

while (table[index] != null && !table[index].key.equals(key)) {

index = (index + stepSize) % TABLE\_SIZE;

}

if (table[index] != null && table[index].key.equals(key)) {

return table[index].value;

} else {

return null;

}

}

public static void main(String[] args) {

DoubleHashingHashTable hashTable = new DoubleHashingHashTable();

// Inserting key-value pairs

hashTable.insert("John", "Doe");

hashTable.insert("Jane", "Smith");

hashTable.insert("Bob", "Jones");

// Retrieving values

System.out.println("Value associated with John: " + hashTable.get("John"));

System.out.println("Value associated with Jane: " + hashTable.get("Jane"));

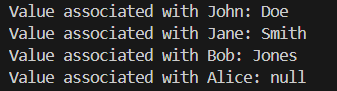
System.out.println("Value associated with Bob: " + hashTable.get("Bob"));

System.out.println("Value associated with Alice: " + hashTable.get("Alice"));

}

}

Output:



**Conclusion:**

One common technique is open addressing, where collisions are resolved by probing the table for an empty slot. Linear probing sequentially searches for the next available slot, while quadratic probing uses a quadratic function to determine the next slot to check, reducing clustering. However, both methods can suffer from clustering issues. Another approach, double hashing, uses two hash functions to calculate the step size, which helps mitigate clustering and provides better distribution of values across the table. Separate chaining is another popular technique, where each slot in the hash table contains a linked list of key-value pairs that hash to the same index. This method is simple and efficient but may incur additional memory overhead. Each hashing technique has its advantages and disadvantages, and the choice of which to use depends on factors such as the data distribution, performance requirements, and memory constraints. Overall, understanding the characteristics of different hashing techniques is essential for designing efficient and reliable data structures and algorithms.