Task01

What kind of collision resolution strategy is implemented in the below Hash Table ?

import java.util.\*;

class Task01 {

    LinkedList<Entry>[] data = new LinkedList[10];

    public void put(String keyval, int value) {

        int index = Math.abs(keyval.hashCode() % data.length);

        if (data[index] == null) {

            data[index] = new LinkedList<>();

        }

        for (Entry e : data[index]) {

            if (e.keyval.equals(keyval)) {

                e.value = value;

                return;

            }

        }

        data[index].add(new Entry(keyval, value));

    }

    static class Entry {

        String keyval;

        int value;

        Entry(String k, int v) {

            keyval = k;

            value = v;

        }

    }

}

is it using

1. to fill collisions is it linear probing with backtracking

or

1. Opening address by placing values at next available bucket

or

1. at each index chaining using a linked list

or

1. on each collision resizing hash table

At each index chaining using a linked list

Task 02:

Wap to take input from the user a 5 digit no and display digit by digit in the output

Hint:

If input is  456897

Output:

units digit is 7

Ones digit is 9

Hundreds digit is 8

Thousands digit is 6

10 thousands digit is 5

Lakhs digit is 4

import java.util.Scanner;

public class Task02 {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter a number (at least 5 digits): ");

int num = sc.nextInt();

int position = 0;

String[] placeNames = {

"Units digit is ",

"Ones digit is ",

"Hundreds digit is ",

"Thousands digit is ",

"10 Thousands digit is ",

"Lakhs digit is ",

"Ten Lakhs digit is ",

"Crores digit is "

};

while (num > 0 && position < placeNames.length) {

int digit = num % 10;

System.out.println(placeNames[position] + digit);

num = num / 10;

position++;

}

sc.close();

}

}

INPUT: 456897

Output:

Units digit is 7

Ones digit is 9

Hundreds digit is 8

Thousands digit is 6

10 Thousands digit is 5

Lakhs digit is 4

Task 03:

Wap to take number from the user and display the no of digit it has

HInt:

If input is:

10,000

Output:

Its a 5 digit number

import java.util.Scanner;

public class Task03 {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter a number: ");

long num = sc.nextLong(); // Supports large numbers

// Convert to positive if it's negative

num = Math.abs(num);

int count = 0;

if (num == 0) {

count = 1; // Special case: 0 has 1 digit

} else {

while (num > 0) {

num = num / 10;

count++;

}

}

System.out.println("It's a " + count + " digit number.");

sc.close();

}

}

Input: 10000

Output: It's a 5 digit number.

Tsk 04:

What are the applications of heap sort?

Heap Sort is a **comparison-based sorting algorithm** that uses a binary heap data structure. It’s particularly useful in several scenarios due to its efficiency and predictable performance.

**🔹 1. Priority Queues**

* Heap sort is based on the heap data structure, which is the backbone of **priority queues**.
* Applications:
  + Task scheduling in operating systems
  + Bandwidth management in networks
  + Job scheduling in servers

**🔹 2. Real-time Systems**

* **Heap sort has a time complexity of O(n log n)** in all cases, making it predictable (unlike QuickSort).
* In real-time systems, where timing and reliability are critical, heap sort is preferred.

**🔹 3. Embedded Systems**

* Heap sort doesn't use extra memory (it sorts in-place), which is crucial for **memory-constrained environments** like embedded systems.

**🔹 4. Graph Algorithms**

* **Dijkstra’s shortest path algorithm** and **Prim’s Minimum Spanning Tree (MST)** algorithm use **heaps** to efficiently select the next node or edge.
* Though the sorting is implicit, heap operations are vital.

**🔹 5. Sorting Large Datasets**

* Useful when data fits in memory and stability is not required.
* Example: **Offline analytics**, logs processing, etc.

**🔹 6. k-Largest or k-Smallest Elements**

* Finding the **k-th largest or smallest** element in a stream or array is efficiently done using a heap, often combined with heap sort principles.

**🔹 7. Selection Algorithms**

* Heap sort can help in **partial sorting** — e.g., finding top N items from a dataset (e.g., top 10 highest scores, top trending hashtags).

Task 05:

Do you find any significance change between the breadthFirstSearchRecursive() approach compared to the standard BFS?

1. Will it  need for queues entirely by using a stack-based recursion?

1. Will it simplifies implementation by using queues implicitly within recursive function calls?

1. will it achieve same result but emphasizes on recursive style using the same level-order logic with explicit queue management?

or

1. will it processes nodes in post-order sequence to avoid memory allocation?

### ✅ ****Explanation:****

#### 🔹 **Standard BFS:**

* Uses an **explicit queue**.
* Processes nodes **level-by-level** (also called level-order traversal in trees).
* Typical iterative approach.

#### 🔹 **Recursive BFS:**

* Still **requires a queue** to maintain the correct order of node processing.
* But instead of using a loop, the recursion handles the traversal step-by-step.
* Queue is **passed as a parameter** or managed outside the recursive function.

#### 📌 So:

* It **does not eliminate the need for a queue**.
* It **does not use a stack-based recursion** like DFS.
* It **does not use post-order**, which is a DFS concept.
* It simply **implements level-order traversal recursively**, still using an explicit queue to maintain breadth-first behavior.

Task 06:

How does heap sort work ? explain the technique in 5 .. algorithm

**Heap Sort Technique in 5 Steps (Algorithm):**

**🔹 1. Build a Max Heap from the input array**

* Convert the input array into a **Max Heap**, where the largest element is at the root.
* This is done using the **heapify()** process from the bottom up.

📝 Example:  
Input: [4, 10, 3, 5, 1] → Max Heap: [10, 5, 3, 4, 1]

**🔹 2. Swap the root with the last element**

* The largest value is at the root (index 0).
* Swap it with the **last element** in the heap.
* After the swap, the largest element is in its correct sorted position.

📝 Example:  
Swap 10 with 1 → [1, 5, 3, 4, 10]

**🔹 3. Reduce the heap size**

* Exclude the last element (now sorted) from the heap.
* Heap size is reduced by 1.

**🔹 4. Heapify the root again**

* After swapping, the heap property may be violated.
* Apply **heapify()** on the root to restore the Max Heap structure.

📝 Example:  
Heapify [1, 5, 3, 4] → [5, 4, 3, 1]

**🔹 5. Repeat until all elements are sorted**

* Continue swapping the root with the last unsorted element, reducing the heap size, and re-heapifying.
* After all iterations, the array becomes sorted in ascending order.

**✅ Final Sorted Output:**

From [4, 10, 3, 5, 1] → Sorted: [1, 3, 4, 5, 10]

**✅ Summary of Steps:**

| **Step** | **Description** |
| --- | --- |
| 1 | Build a Max Heap from the array |
| 2 | Swap root (max) with last element |
| 3 | Reduce heap size |
| 4 | Heapify the root to restore heap property |
| 5 | Repeat steps 2–4 until sorted |

Task 07:

how can you say recursive functions maintain the state of each call during execution?

1. Each recursive call creates a new thread, and context switching maintains state.

2. Recursive functions store state in global variables accessible across calls.

3. The system call stack tracks local variables and return addresses for each recursive invocation.

4. Recursive functions replicate the heap structure to keep values between calls.

### ****Correct Answer:****

**3. The system call stack tracks local variables and return addresses for each recursive invocation.**

Task 11:

Which concept explains how recursive functions maintain the state of each call during execution?

1. Each recursive call creates a new thread, and context switching maintains state.

2. Recursive functions store state in global variables accessible across calls.

3. The system call stack tracks local variables and return addresses for each recursive invocation.

4. Recursive functions replicate the heap structure to keep values between calls.

### ****Correct Answer:****

**3. The system call stack tracks local variables and return addresses for each recursive invocation.**

Task 12:

How does this binary search function behave on unsorted arrays?

public class BinarySearch {

    public int search(int[] arr, int target) {

        int left = 0, right = arr.length - 1;

        while (left <= right) {

            int mid = left + (right - left) / 2;

            if (arr[mid] == target) {

                return mid;

            } else if (arr[mid] < target) {

                left = mid + 1;

            } else {

                right = mid - 1;

            }

        }

        return -1;

    }

}

The function will return **incorrect or unpredictable results** on **unsorted arrays**.

Binary search on unsorted arrays produces **unpredictable or incorrect results**, because it relies on the array being **sorted** to function properly.

Task 13:

What is the result of performing DFS traversal in this graph implementation?

import java.util.\*;

public class DFSGraph {

     Map<Integer, List<Integer>> adj = new HashMap<>();

     Set<Integer> visited = new HashSet<>();

     public void addEdge(int u, int v) {

        adj.computeIfAbsent(u, x -> new ArrayList<>()).add(v);

    }

     public void dfs(int node) {

        if (visited.contains(node)) {

            return;

        }

        visited.add(node);

        System.out.print(node + " ");

        for (int neighbor : adj.getOrDefault(node, new ArrayList<>())) {

            dfs(neighbor);

        }

    }

}

1. DFS uses a queue to ensure order

2. DFS will return shortest path like BFS

3. DFS traverses all nodes depth-first recursively

4. DFS skips connected nodes due to reentrancy issue

Answer: 3. DFS traverses all nodes depth-first recursively

Task 14:

Why is BFS generally preferred over DFS in shortest path algorithms for unweighted graphs?

1. BFS uses random access to edges, ensuring constant-time traversal.

2. BFS explores one path to maximum depth before switching, reducing memory usage.

3. BFS ignores revisiting nodes, reducing processing time in cyclic graphs.

4. BFS explores nodes in increasing distance order from the source, ensuring shortest paths are found first.

Task 15:

Write algo for radix sort

#### **Step-by-step Algorithm:**

1. **Find the maximum number** in the array to know the number of digits.
2. **Iterate from the least significant digit to the most significant digit**:
   * Use a **stable sort** (like counting sort) to sort the array based on the current digit.
3. After processing all digit places, the array will be sorted.

Task 16:

Write pseudo code for radix sort

function radixSort(arr):

maxVal = findMaximum(arr)

exp = 1 // 1s, 10s, 100s, ...

while (maxVal / exp) > 0:

countingSortByDigit(arr, exp)

exp \*= 10

function countingSortByDigit(arr, exp):

output = array of same length as arr

count = array of size 10 (for digits 0-9)

// Count occurrences of each digit at current place (exp)

for i = 0 to arr.length - 1:

index = (arr[i] / exp) % 10

count[index] += 1

// Transform count to represent actual positions

for i = 1 to 9:

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

// Build the output array (iterate in reverse to keep it stable)

for i = arr.length - 1 down to 0:

index = (arr[i] / exp) % 10

output[count[index] - 1] = arr[i]

count[index] -= 1

// Copy output back to arr

for i = 0 to arr.length - 1:

arr[i] = output[i]

Task 17:

Write code for radix sort

public class RadixSort {

// Main radix sort method

public void radixSort(int[] arr) {

int max = getMax(arr);

// Sort the array for every digit (exp = 1, 10, 100, ...)

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

countingSortByDigit(arr, exp);

}

}

// Helper method to get the maximum value in the array

private int getMax(int[] arr) {

int max = arr[0];

for (int num : arr) {

if (num > max) {

max = num;

}

}

return max;

}

// Counting sort based on digit at current exp (1s, 10s, etc.)

private void countingSortByDigit(int[] arr, int exp) {

int n = arr.length;

int[] output = new int[n]; // Output array

int[] count = new int[10]; // Count array for digits (0-9)

// Count occurrences of digits

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

int digit = (arr[i] / exp) % 10;

count[digit]++;

}

// Change count[i] to contain actual position in output[]

for (int i = 1; i < 10; i++) {

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

}

// Build the output array (traverse from right to left to maintain stability)

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

int digit = (arr[i] / exp) % 10;

output[count[digit] - 1] = arr[i];

count[digit]--;

}

// Copy the sorted values back to original array

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

arr[i] = output[i];

}

}

// Test the radix sort

public static void main(String[] args) {

int[] arr = {170, 45, 75, 90, 802, 24, 2, 66};

RadixSort sorter = new RadixSort();

sorter.radixSort(arr);

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

for (int num : arr) {

System.out.print(num + " ");

}

}

}