Day 19 19th July 2025

**Task 01:**

**In a trie (prefix tree), what is the most significant benefit it provides in information retrieval systems like autocomplete?**

1. It stores keys in a hash map allowing faster lookup than string comparison.

2. It enables prefix-based searching by storing characters in a tree-like format, reducing lookup time.

3. It compresses all values into a single hash index for instant access.

4. It eliminates the need for traversal by maintaining precomputed suggestions for each node.

**Task 02:**

**What do you understand by stable and unstable sorting?**



**Stable Sorting Algorithms:**

A stable sorting algorithm preserves the relative order of elements with equal keys. If two elements have the same value, and one appears before the other in the original unsorted array, it will also appear before the other in the sorted array

**Unstable Sorting Algorithms:**

An unstable sorting algorithm does not guarantee to preserve the relative order of elements with equal keys. The order of equal elements might change after sorting.

**Task 03:**

**What is the primary purpose of reversing the pointers the linked list?**

1. To convert singly linked list into doubly linked list

2. To delete all the nodes in reverse order

3. To perform in-place reversal of the list with O(1) space

4. To traverse backwards using a stack

**Task 04:**

**What does O(log n) signify when used in the context of a binary search tree operation?**

1. The number of steps grows linearly with the size of the input.

2. The operation takes exponential time depending on tree height.

3. The number of steps grows proportionally to the logarithm of the input size, typical for balanced trees.

4. The operation performs a constant number of steps for each input regardless of size.

**Task 05:**

**What distinguishes a queue implemented with a linked list from one implemented using an array in terms of performance?**

1. Array-based queues allow two-directional traversal, making them superior for complex operations.

2. Array-based queues can expand without limit, offering better memory efficiency.

3. Linked list-based queues avoid resizing operations, providing consistent performance during enqueue and dequeue.

4. Linked list-based queues require preallocation of memory which improves speed.

**Task 06:**

**In a binary search algorithm, why must the input data be sorted before execution?**

1. Binary search modifies the array structure, so sorting prevents errors.

2. Binary search only works with integer values, which are easier to sort.

3. Sorting allows the algorithm to eliminate half of the search space in each step, achieving O(log n) time.

4. Sorting ensures that every item has a fixed memory address, improving cache locality.

**Task 07:**

**What is the significance of using a linked list to implement a stack instead of an array?**

1. Linked list implementation leads to slower operations but saves space due to non-contiguous memory.

2. Linked list stacks prevent duplicate values and automatically enforce element uniqueness.

3. Linked list-based stacks avoid overflow by dynamically growing in memory without the need for resizing arrays.

4. Linked list stacks operate using a tree-like structure for better depth analysis.

**Task 07:**

**What is the significance of using a linked list to implement a stack instead of an array?**

1. Linked list implementation leads to slower operations but saves space due to non-contiguous memory.

2. Linked list stacks prevent duplicate values and automatically enforce element uniqueness.

3. Linked list-based stacks avoid overflow by dynamically growing in memory without the need for resizing arrays.

4. Linked list stacks operate using a tree-like structure for better depth analysis.

**Task 08:**

**"What will be the result when inserting keys with the same hash in this custom hash map?**

import java.util.ArrayList;

import java.util.List;

public class HashCollision {

static class Entry {

String key;

int value;

Entry(String key, int value) {

this.key = key;

this.value = value;

}

}

List<Entry>[] table = new ArrayList[10];

public void put(String key, int val) {

int index = Math.abs(key.hashCode() % table.length);

if (table[index] == null) {

table[index] = new ArrayList<>();

}

table[index].add(new Entry(key, val));

}

}

1. Only one key-value pair will be stored due to overwriting

2. Insertion will fail due to duplicate key exception

3. Multiple values are stored in same bucket via chaining

4. Values are distributed across different buckets using linear probing

**Task 09:**

**What is the key advantage of memoization in dynamic programming?**

1. It compresses all inputs into a single hash, making the program run in O(1).

2. It avoids re-computation by caching results of expensive function calls, reducing time complexity.

3. It removes the need for recursion by using fixed-size loops.

4. It uses real-time memory access to directly jump to final results.

**Task 10:**

**What is the role of memoization in this DP stairs problem?**

import java.util.HashMap;

import java.util.Map;

public class ClimbStairsDP {

Map<Integer, Integer> memo = new HashMap<>();

public int climbStairs(int n) {

if (n <= 2) return n;

if (memo.containsKey(n)) {

return memo.get(n);

}

int result = climbStairs(n - 1) + climbStairs(n - 2);

memo.put(n, result);

return result;

}

}

1. It stores results of previous calls to avoid recomputation

2. It uses bottom-up table to optimize space

3. It converts recursion into iteration

4. It parallelizes recursive calls for performance

**Task 11:**

**What is the key reason for stack overflow in this recursive call?**

public class RecursiveLoop {

public int calculate(int n) {

if (n == 0) return 0;

return n + calculate(n);

}

public static void main(String[] args) {

System.out.println(new RecursiveLoop().calculate(5));

}

}

1. The method calculates correctly and stops after 5 iterations

2. Recursion ends correctly but prints wrong result

3. The stack is manually cleared causing error

4. The recursive method is missing a base case decrement

**Task 12:**

**What will be the time complexity of the given merge operation between two sorted arrays?**

public class MergeSortedArrays {

public int[] merge(int[] a, int[] b) {

int[] merged = new int[a.length + b.length];

int i = 0, j = 0, k = 0;

while (i < a.length && j < b.length) {

if (a[i] < b[j]) {

merged[k++] = a[i++];

} else {

merged[k++] = b[j++];

}

}

while (i < a.length) {

merged[k++] = a[i++];

}

while (j < b.length) {

merged[k++] = b[j++];

}

return merged;

}

}

1. O(n + m) since we traverse both arrays once where n and m are lengths of arrays

2. O(n log m) due to nested comparisons between arrays

3. O(n^2) because we compare each element with all in second array

4. O(log n) as merging halves is logarithmic

**Task 13:**

**How does the binary tree traversal logic work in this BFS example?**

import java.util.\*;

// Definition of TreeNode (assumed to be available)

// class TreeNode {

// int val;

// TreeNode left;

// TreeNode right;

// TreeNode(int val) { this.val = val; }

// }

public class BinaryTreeBFS {

public List<List<Integer>> levelOrder(TreeNode root) {

List<List<Integer>> result = new ArrayList<>();

Queue<TreeNode> queue = new LinkedList<>();

if (root == null) return result;

queue.add(root);

while (!queue.isEmpty()) {

int size = queue.size();

List<Integer> level = new ArrayList<>();

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

TreeNode node = queue.poll();

level.add(node.val);

if (node.left != null) {

queue.add(node.left);

}

if (node.right != null) {

queue.add(node.right);

}

}

result.add(level);

}

return result;

}

}"

1. It visits all right nodes first and then left nodes

2. It performs in-order traversal level by level

3. It performs level-by-level traversal using a queue

4. It uses recursion for pre-order traversal

**Task 14:**

**What is the significance of using a linked list to implement a stack instead of an array?**

1. Linked list implementation leads to slower operations but saves space due to non-contiguous memory.

2. Linked list stacks prevent duplicate values and automatically enforce element uniqueness.

3. Linked list-based stacks avoid overflow by dynamically growing in memory without the need for resizing arrays.

4. Linked list stacks operate using a tree-like structure for better depth analysis.

**Task 15:**

**What is one key difference in memory usage between recursive and iterative implementations of the same algorithm?**

1. Recursive implementations always consume less memory due to reuse of base cases.

2. Iterative implementations use less memory as they do not require stack frames for each call.

3. Recursive implementations use shared memory across calls to avoid duplication.

4. Iterative algorithms require storing all intermediary steps, leading to increased memory usage.

**Task 16:**

**"What is the traversal type in this BST in-order function?**

class TreeNode {

int val;

TreeNode left, right;

TreeNode(int v) {

val = v;

}

}

public class BSTInOrder {

public void inorder(TreeNode root) {

if (root == null) {

return;

}

inorder(root.left);

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

inorder(root.right);

}

}"

1. Pre-order traversal where root is visited first

2. In-order traversal resulting in sorted order for BST

3. Post-order traversal used for deleting nodes

4. Level-order traversal using recursion

**Task 17:**

**Which constraint is critical when implementing a stack using an array?**

public class StackArray {

int[] stack;

int top;

public StackArray(int size) {

stack = new int[size];

top = -1;

}

public void push(int val) {

if (top < stack.length - 1) {

stack[++top] = val;

} else {

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

}

}

public int pop() {

if (top >= 0) {

return stack[top--];

} else {

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

return -1;

}

}

}

1. You must avoid using index zero to maintain overflow safety

2. The array must always be reallocated on overflow

3. Stack must be synchronized for thread safety

4. You need to ensure bounds are checked to avoid overflow/underflow

**Task 18:**

**"What will be printed by BFS on the following graph with starting node 1?**

import java.util.\*;

public class GraphBFS {

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

public void addEdge(int u, int v) {

graph.computeIfAbsent(u, k -> new ArrayList<>()).add(v);

}

public void bfs(int start) {

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

Queue<Integer> queue = new LinkedList<>();

queue.add(start);

visited.add(start);

while (!queue.isEmpty()) {

int current = queue.poll();

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

for (int neighbor : graph.getOrDefault(current, new ArrayList<>())) {

if (!visited.contains(neighbor)) {

queue.add(neighbor);

visited.add(neighbor);

}

}

}

}

}

1. Depth-first traversal order from node 1

2. Level-order traversal of all connected nodes from node 1

3. Nodes printed in reverse due to stack usage

4. Only prints the root node as others are skipped

**Task 19:**

**What is the main reason to use dynamic arrays over static arrays in modern applications?**

1. Dynamic arrays are implemented using trees, which provide better lookup speed.

2. Dynamic arrays support variable-length keys, allowing nested data storage.

3. Dynamic arrays automatically resize when the capacity is exceeded, offering flexibility for unpredictable input sizes.

4. Dynamic arrays preallocate more memory than needed, ensuring faster access through pointer arithmetic.

Add ons

