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

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

Code:

package July18;  
import java.util.Scanner;  
  
public class DigitPlacePrinter {  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
  
 // Take input from user  
 System.*out*.print("Enter a 6-digit number: ");  
 int number = scanner.nextInt();  
  
 if (number < 100000 || number > 999999) {  
 System.*out*.println("Please enter a valid 6-digit number.");  
 } else {  
 int units = number % 10;  
 int tens = (number / 10) % 10;  
 int hundreds = (number / 100) % 10;  
 int thousands = (number / 1000) % 10;  
 int tenThousands = (number / 10000) % 10;  
 int lakhs = number / 100000;  
  
 System.*out*.println("Units digit is " + units);  
 System.*out*.println("Ones digit is " + tens);  
 System.*out*.println("Hundreds digit is " + hundreds);  
 System.*out*.println("Thousands digit is " + thousands);  
 System.*out*.println("10 thousands digit is " + tenThousands);  
 System.*out*.println("Lakhs digit is " + lakhs);  
 }  
  
 scanner.close();  
 }  
}

output:

Enter a 6-digit number: 789456

Units digit is 6

Ones digit is 5

Hundreds digit is 4

Thousands digit is 9

10 thousands digit is 8

Lakhs digit is 7

Process finished with exit code 0

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

Code:

package July18;  
  
import java.util.Scanner;  
  
public class DigitCounter {  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
  
 // Take input  
 System.*out*.print("Enter a number: ");  
 long number = scanner.nextLong();  
  
 // Make number positive if it's negative  
 number = Math.*abs*(number);  
  
 // Special case for 0  
 if (number == 0) {  
 System.*out*.println("It's a 1-digit number");  
 } else {  
 int count = 0;  
 long temp = number;  
  
 while (temp > 0) {  
 temp = temp / 10;  
 count++;  
 }  
  
 System.*out*.println("It's a " + count + "-digit number");  
 }  
  
 scanner.close();  
 }  
}

output:

Enter a number: 258963

It's a 6-digit number

Process finished with exit code 0

Tsk 04:

What are the applications of heap sort?

Implementing priority queues

Task scheduling in operating systems

Dijkstra’s and Prim’s algorithms (graph shortest path)

Event-driven simulations (e.g., network packet scheduling)

Sorting large datasets in external memory

Finding top-K largest or smallest elements in streaming data

Memory-constrained systems due to in-place sorting

Used in Heapsort-based sorting libraries/tools

Huffman coding for data compression

Real-time analytics and dashboards with live rankings

Task 05:

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

1. Will it the 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

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

Task 06:

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

**Heap Sort Technique in 5 Steps:**

1. **Build a Max Heap** from the input array (the largest value becomes the root).
2. **Swap** the root (maximum value) with the last element of the heap.
3. **Reduce the heap size** by 1 (exclude the last element, which is now sorted).
4. **Heapify** the root again to maintain the max heap property.
5. **Repeat** steps 2–4 until the heap size becomes 1.

Algorithm

heapSort(arr[], n):

1. Build a max heap from the array

for i = n/2 - 1 down to 0:

heapify(arr, n, i)

2. One by one extract elements from the heap

for i = n - 1 down to 1:

swap arr[0] with arr[i] // move max to end

heapify(arr, i, 0) // heapify reduced heap

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.

Which property of a priority queue differentiates it most from a regular queue implementation?

1. It allows insertion and removal only from one end, similar to a stack.

2. Elements are removed based on their order of insertion rather than priority.

3. Elements are dequeued based on their priority, not their insertion order, often implemented using a binary heap.

4. It maintains a strict hierarchical structure using a self-balancing BST to enforce priority.

Task 09: recap of Quiz qn

What is the main purpose of using a binary heap in the implementation of a priority queue?

1. To maintain keys in alphabetical order for efficient string processing.

2. To ensure that the highest-priority element always bubbles to the root efficiently.

3. To guarantee constant-time insertion and logarithmic-time deletion.

4. To reduce memory consumption by flattening the tree into a linear array.

Task 15:

Write algo for radix sort

1. Find the maximum number in the array to determine the number of digits.
2. Starting from the least significant digit, do:
   * Sort all numbers by the current digit using Counting Sort.
3. Repeat step 2 for each digit place (units, tens, hundreds, etc.).

Task 16:

Write pseudo code for radix sort

RADIX\_SORT(arr[], n)

1. maxNum ← maximum number in arr[]

2. exp ← 1 // exponent representing digit place (1 for units, 10 for tens, etc.)

3. while (maxNum / exp > 0)

a. COUNTING\_SORT\_BY\_DIGIT(arr[], n, exp)

b. exp ← exp × 10

Task 17:

Write code for radix sort

Code:

package July18;  
  
public class RadixSort {  
  
 // Function to get the maximum value in the array  
 static int getMax(int[] arr) {  
 int max = arr[0];  
 for (int num : arr) {  
 if (num > max)  
 max = num;  
 }  
 return max;  
 }  
  
 // A function to do counting sort based on the digit represented by exp  
 static void countingSort(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 to 9  
  
 // Count occurrences of digits  
 for (int i = 0; i < n; i++) {  
 int digit = (arr[i] / exp) % 10;  
 count[digit]++;  
 }  
  
 // Update count[i] so that it contains actual positions  
 for (int i = 1; i < 10; i++)  
 count[i] += count[i - 1];  
  
 // Build the output array (stable sort - go backwards)  
 for (int i = n - 1; i >= 0; i--) {  
 int digit = (arr[i] / exp) % 10;  
 output[count[digit] - 1] = arr[i];  
 count[digit]--;  
 }  
  
 // Copy the output array to original array  
 for (int i = 0; i < n; i++)  
 arr[i] = output[i];  
 }  
  
 // Main function to do radix sort  
 static void radixSort(int[] arr) {  
 int max = *getMax*(arr); // Find the max number  
  
 // Apply counting sort for every digit  
 for (int exp = 1; max / exp > 0; exp \*= 10)  
 *countingSort*(arr, exp);  
 }  
  
 // Utility function to print array  
 static void printArray(int[] arr) {  
 for (int num : arr)  
 System.*out*.print(num + " ");  
 System.*out*.println();  
 }  
  
 // Main method  
 public static void main(String[] args) {  
 int[] arr = {170, 45, 75, 90, 802, 24, 2, 66};  
  
 System.*out*.println("Original array:");  
 *printArray*(arr);  
  
 *radixSort*(arr);  
  
 System.*out*.println("Sorted array:");  
 *printArray*(arr);  
 }  
}

output:

Original array:

170 45 75 90 802 24 2 66

Sorted array:

2 24 45 66 75 90 170 802

Process finished with exit code 0

Home task

Task 17: display all corner elements of a tree

import java.util.\*;

class Node {

int data;

Node left, right;

Node(int val) {

data = val;

left = right = null;

}

}

public class CornerElements {

// Function to print corner elements at each level

static void printCornerElements(Node root) {

if (root == null) return;

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

queue.add(root);

while (!queue.isEmpty()) {

int levelSize = queue.size();

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

Node curr = queue.poll();

// Print corner elements

if (i == 0 || i == levelSize - 1) {

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

}

if (curr.left != null) queue.add(curr.left);

if (curr.right != null) queue.add(curr.right);

}

System.out.println(); // Newline for next level

}

}

// Build the tree as per the image

public static void main(String[] args) {

Node root = new Node(1);

root.left = new Node(2);

root.right = new Node(3);

root.left.left = new Node(4);

root.left.right = new Node(5);

root.right.left = new Node(6);

root.right.right = new Node(7);

root.left.left.left = new Node(8);

root.left.left.right = new Node(9);

root.left.right.left = new Node(10);

root.left.right.right = new Node(11);

root.right.left.left = new Node(12);

root.right.left.right = new Node(13);

root.right.right.left = new Node(14);

root.right.right.right = new Node(15);

System.out.println("Corner elements of each level:");

printCornerElements(root);

}

}

Output:

Corner elements of each level:

1

2 3

4 7

8 15

Process finished with exit code 0

Task 18  Reverse Alternate levels:

package July18;  
  
import java.util.\*;  
  
class Node {  
 char key;  
 Node left, right;  
  
 Node(char key) {  
 this.key = key;  
 }  
}  
  
public class ReverseOddLevels {  
  
 Node root;  
  
 // Function to reverse nodes at alternate odd levels (root is Level 1)  
 void reverseOddLevels(Node left, Node right, int level) {  
 if (left == null || right == null)  
 return;  
  
 // If current level is odd, swap values  
 if (level % 2 == 1) {  
 char temp = left.key;  
 left.key = right.key;  
 right.key = temp;  
 }  
  
 // Recurse deeper  
 reverseOddLevels(left.left, right.right, level + 1);  
 reverseOddLevels(left.right, right.left, level + 1);  
 }  
  
 // Function to print tree level by level  
 void printLevelOrder(Node root) {  
 if (root == null) return;  
  
 Queue<Node> q = new LinkedList<>();  
 q.add(root);  
  
 while (!q.isEmpty()) {  
 int levelSize = q.size();  
 for (int i = 0; i < levelSize; i++) {  
 Node temp = q.poll();  
 System.*out*.print(temp.key + " ");  
 if (temp.left != null) q.add(temp.left);  
 if (temp.right != null) q.add(temp.right);  
 }  
 System.*out*.println(); // new line for new level  
 }  
 }  
  
 public static void main(String[] args) {  
 ReverseOddLevels tree = new ReverseOddLevels();  
  
 // Level 1  
 tree.root = new Node('A');  
  
 // Level 2  
 tree.root.left = new Node('B');  
 tree.root.right = new Node('C');  
  
 // Level 3  
 tree.root.left.left = new Node('D');  
 tree.root.left.right = new Node('E');  
 tree.root.right.left = new Node('F');  
 tree.root.right.right = new Node('G');  
  
 // Level 4  
 tree.root.left.left.left = new Node('H');  
 tree.root.left.left.right = new Node('I');  
 tree.root.left.right.left = new Node('J');  
 tree.root.left.right.right = new Node('K');  
 tree.root.right.left.left = new Node('L');  
 tree.root.right.left.right = new Node('M');  
 tree.root.right.right.left = new Node('N');  
 tree.root.right.right.right = new Node('O');  
  
 System.*out*.println("Before reversal (Level Order):");  
 tree.printLevelOrder(tree.root);  
  
 // Perform alternate level reversal (odd levels only)  
 tree.reverseOddLevels(tree.root.left, tree.root.right, 2); // Start with level 2 (children of level 1)  
  
 System.*out*.println("\nAfter reversing odd levels (Level Order):");  
 tree.printLevelOrder(tree.root);  
 }  
}

output:

Before reversal (Level Order):

A

B C

D E F G

H I J K L M N O

After reversing odd levels (Level Order):

A

B C

G F E D

H I J K L M N O

Process finished with exit code 0

<https://leetcode.com/problems/binary-tree-right-side-view/description/>

package July18;  
  
import java.util.\*;  
  
class TreeNode {  
 int val;  
 TreeNode left, right;  
  
 TreeNode(int val) {  
 this.val = val;  
 }  
}  
  
public class RightSideView {  
  
 public List<Integer> rightSideView(TreeNode root) {  
 List<Integer> result = new ArrayList<>();  
 if (root == null) return result;  
  
 Queue<TreeNode> queue = new LinkedList<>();  
 queue.add(root);  
  
 while (!queue.isEmpty()) {  
 int levelSize = queue.size();  
 TreeNode rightMost = null;  
  
 for (int i = 0; i < levelSize; i++) {  
 TreeNode current = queue.poll();  
  
 // The last node in this level is the rightmost one  
 if (i == levelSize - 1)  
 result.add(current.val);  
  
 if (current.left != null) queue.add(current.left);  
 if (current.right != null) queue.add(current.right);  
 }  
 }  
  
 return result;  
 }  
  
 // Helper method to manually construct tree from array  
 public static TreeNode buildExampleTree1() {  
 TreeNode root = new TreeNode(1);  
 root.left = new TreeNode(2);  
 root.right = new TreeNode(3);  
 root.left.right = new TreeNode(5);  
 root.right.right = new TreeNode(4);  
 return root;  
 }  
  
 public static void main(String[] args) {  
 RightSideView solution = new RightSideView();  
  
 TreeNode example1 = *buildExampleTree1*();  
 System.*out*.println("Right side view: " + solution.rightSideView(example1)); // Output: [1, 3, 4]  
 }  
}

output:

Before reversal (Level Order):

A

B C

D E F G

H I J K L M N O

After reversing odd levels (Level Order):

A

B C

G F E D

H I J K L M N O

Process finished with exit code 0