1.A

**Algorithmic and Heap Sort Trading**

Algorithmic trading involves using computer algorithms to execute trades in financial markets. These algorithms analyze market data, make decisions, and execute orders automatically. One common application of algorithmic trading is **high-frequency trading (HFT)**, where traders aim to profit from small price fluctuations by executing a large number of trades in a short time

1. **Statistical Verification**: Algorithmic trading relies on statistical analysis of market data. It considers variables such as time, price, volume, and technical indicators to make trading decisions.
2. **Reducing Human Error**: Unlike human traders, algorithms don’t suffer from parallax errors or emotional biases. They execute trades based on predefined rules, minimizing errors.
3. **High-Frequency Trading (HFT)**: Algorithms can process a large number of trades quickly. HFT strategies aim to capitalize on tiny price differences, and heap sort plays a role in managing these trades efficiently.

Heap sort is an efficient sorting algorithm that uses a binary heap data structure. It has a time complexity of O(n log n), making it suitable for real-time applications. Here’s a brief overview of heap sort:

1. **Heap Data Structure**:
   * A heap is a binary tree where each parent node has a value greater (or smaller) than its children.
   * In a **max heap**, the parent node has a greater value than its children.
   * In a **min heap**, the parent node has a smaller value than its children.
2. **Heap Sort Algorithm**:
   * Build a max heap from the given array of trade prices.
   * Extract the maximum element (root) from the heap and place it at the end of the array.
   * Reduce the heap size and heapify the remaining elements.
   * Repeat the extraction and heapification until the entire array is sorted.

public class HeapSort {

public void sort(int arr[]) {

int N = arr.length;

for (int i = N / 2 - 1; i >= 0; i--)

heapify(arr, N, i);

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

int temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

heapify(arr, i, 0);

}

}

void heapify(int arr[], int N, int i) {

int largest = i;

int l = 2 \* i + 1;

int r = 2 \* i + 2;

if (l < N && arr[l] > arr[largest])

largest = l;

if (r < N && arr[r] > arr[largest])

largest = r;

if (largest != i) {

int swap = arr[i];

arr[i] = arr[largest];

arr[largest] = swap;

heapify(arr, N, largest);

}

}

static void printArray(int arr[]) {

int N = arr.length;

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, 7 };

int N = arr.length;

HeapSort ob = new HeapSort();

ob.sort(arr);

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

printArray(arr);

}

}

2.A

class ListNode {

int val;

ListNode next;

ListNode(int x) { val = x; next = null; }

}

public class Solution {

public ListNode middleNode(ListNode head) {

if (head == null || head.next == null) {

return head;

}

ListNode slow = head;

ListNode fast = head;

while (fast.next != null && fast.next.next != null) {

slow = slow.next;

fast = fast.next.next;

}

return slow;

}

}

3.A

import java.util.\*;

public class SortQueue {

public static void sortQueue(Queue<Integer> q) {

if (q.size() == 0 || q.size() == 1) {

return;

}

int n = q.size();

int count = 0;

int x = q.poll();

sortQueue(q); // recursive call to sort the rest of the queue

while (q.peek() < x && count < n) {

q.add(q.poll());

count++;

}

q.add(x); // insert the current element

if (count == n) {

return;

}

count = 0;

while (q.peek() >= x && count < n) {

q.add(q.poll());

count++;

}

}

public static void main(String[] args) {

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

queue.add(10);

queue.add(20);

queue.add(30);

queue.add(40);

queue.add(50);

System.out.println("Original Queue: " + queue);

sortQueue(queue);

System.out.println("Sorted Queue: " + queue);

}

}

Output:

Original Queue: [20, 10, 40, 30, 50]

Sorted Queue: [10, 20, 30, 40, 50]

4.A

import java.util.Stack;

public class SortStack {

public static void sortStack(Stack<Integer> stack) {

Stack<Integer> tempStack = new Stack<>();

while (!stack.isEmpty()) {

int temp = stack.pop();

while (!tempStack.isEmpty() && tempStack.peek() > temp) {

stack.push(tempStack.pop());

}

tempStack.push(temp);

}

while (!tempStack.isEmpty()) {

stack.push(tempStack.pop());

}

}

public static void main(String[] args) {

Stack<Integer> stack = new Stack<>();

stack.push(5);

stack.push(2);

stack.push(8);

stack.push(3);

stack.push(1);

System.out.println("Original Stack: " + stack);

sortStack(stack);

System.out.println("Sorted Stack: " + stack);

}

}

```

Output:

Original Stack: [5, 2, 8, 3, 1]

Sorted Stack: [1, 2, 3, 5, 8]

5.A

class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

this.next = null;

}

}

public class RemoveDuplicatesFromSortedList {

public static ListNode deleteDuplicates(ListNode head) {

ListNode current = head;

while (current != null && current.next != null) {

if (current.val == current.next.val) {

current.next = current.next.next;

} else {

current = current.next;

}

}

return head;

}

public static void printList(ListNode head) {

ListNode current = head;

while (current != null) {

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

current = current.next;

}

System.out.println();

}

public static void main(String[] args) {

ListNode head = new ListNode(1);

head.next = new ListNode(1);

head.next.next = new ListNode(2);

head.next.next.next = new ListNode(3);

head.next.next.next.next = new ListNode(3);

System.out.println("Original List:");

printList(head);

ListNode newList = deleteDuplicates(head);

System.out.println("List after removing duplicates:");

printList(newList);

}

}

6.A

import java.util.Stack;

public class SequenceInStack {

public static boolean isSequenceInStack(Stack<Integer> stack, int[] sequence) {

Stack<Integer> sequenceStack = new Stack<>();

for (int i : sequence) {

sequenceStack.push(i);

}

while (!stack.isEmpty() || !sequenceStack.isEmpty()) {

if (!sequenceStack.isEmpty()) {

int sequenceTop = sequenceStack.pop();

while (!stack.isEmpty() && stack.peek() == sequenceTop) {

stack.pop();

sequenceTop--;

if (sequenceTop == 0) {

break;

}

}

} else {

stack.pop();

}

}

return sequenceStack.isEmpty();

}

public static void main(String[] args) {

Stack<Integer> stack = new Stack<>();

stack.push(1);

stack.push(2);

stack.push(3);

stack.push(4);

stack.push(5);

int[] sequence = {2, 3};

System.out.println(isSequenceInStack(stack, sequence)); // Output: true

sequence = new int[]{1, 2, 3, 4, 5};

System.out.println(isSequenceInStack(stack, sequence)); // Output: false

}

}

Output:

true

false

7.A

class ListNode {

int val;

ListNode next;

ListNode(int x) { val = x; next = null; }

}

public class MergeLists {

public static ListNode mergeLists(ListNode head1, ListNode head2) {

if (head1 == null) {

return head2;

}

if (head2 == null) {

return head1;

}

ListNode mergedHead = null;

ListNode current = null;

if (head1.val < head2.val) {

mergedHead = head1;

head1 = head1.next;

} else {

mergedHead = head2;

head2 = head2.next;

}

current = mergedHead;

while (head1 != null && head2 != null) {

if (head1.val < head2.val) {

current.next = head1;

head1 = head1.next;

} else {

current.next = head2;

head2 = head2.next;

}

current = current.next;

}

if (head1 != null) {

current.next = head1;

} else {

current.next = head2;

}

return mergedHead;

}

public static void main(String[] args) {

ListNode head1 = new ListNode(1);

head1.next = new ListNode(3);

head1.next.next = new ListNode(5);

ListNode head2 = new ListNode(2);

head2.next = new ListNode(4);

ListNode mergedHead = mergeLists(head1, head2);

while (mergedHead != null) {

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

mergedHead = mergedHead.next;

}

}

}

Output:

1 2 3 4 5

8.A

public class CircularQueueBinarySearch {

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

int left = 0;

int right = arr.length - 1;

// Find the rotation index

while (left < right) {

int mid = (left + right) / 2;

if (arr[left] == arr[mid]) {

left = mid + 1;

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

left = mid + 1;

} else {

right = mid;

}

}

// Perform modified binary search

left = 0;

right = arr.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

if (arr[mid] == target) {

return mid;

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

left = mid + 1;

} else {

right = mid - 1;

}

}

return -1; // Target element not found

}

public static void main(String[] args) {

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

int target = 5;

int result = binarySearch(arr, target);

if (result != -1) {

System.out.println("Target element found at index " + result);

} else {

System.out.println("Target element not found");

}

}

}

Output:

Target element found at index 4