Satyam Chaurasiya

DSA Assignment 2

Question 1.

A. Explain the implementation of Binary Heap.
B. Insert the following values (in the given order) into an initially empty min heap: 46, 34, 83, 75, 25, 57, 93, 27, 17. You need to justify the steps, but a single sentence describing the steps suffices — provided that all the key details are indeed described (you still need to show the step-by-step contents of the heap, after each insertion).
C. Dequeue(remove/delete) the elements, showing the contents of the heap after each removal (only for two elements).

Answer 1.

A. Implementation of Binary Heap: A binary heap is a binary tree data structure that satisfies the "heap property," which states that for every node N, the key of N is greater (in a max heap) or smaller (in a min heap) than or equal to the keys of its children. Binary heaps are typically implemented using an array, where the children of the node at index i are located at indices 2*i+1 (left child) and 2*i+2 (right child).

The key operations for a binary heap are "insertion" and "deletion" (usually called "enqueue" and "dequeue" for a priority queue). When inserting an element, it is added at the end of the array and then "bubbled up" (swapped with its parent) if it violates the heap property. When dequeuing, the root (the minimum or maximum element) is removed, and the last element in the array is moved to the root position and then "bubbled down" (swapped with its smallest child) if it violates the heap property.

B. Step-by-step insertion of values into a min heap:

1. Insert 46: [46]

2. Insert 34: [34, 46]

3. Insert 83: [34, 46, 83]

4. Insert 75: [34, 46, 83, 75]

5. Insert 25: [25, 34, 83, 75, 46]

6. Insert 57: [25, 34, 57, 75, 46, 83]

7. Insert 93: [25, 34, 57, 75, 46, 83, 93]

8. Insert 27: [25, 27, 57, 34, 46, 83, 93, 75]

9. Insert 17: [17, 25, 57, 27, 46, 83, 93, 75, 34]

C. Step-by-step dequeue of elements from the min heap:

1. Dequeue (remove) the root (17): [25, 27, 57, 34, 46, 83, 93, 75]

2. Dequeue (remove) the new root (25): [27, 34, 57, 75, 46, 83, 93]

After each removal, the heap is adjusted to maintain the heap property. The smallest element is removed first, and the next smallest element becomes the new root, and so on.

Questions 2.

For the following array of values: {48, 23, 17, 26, 14, 87, 53, 89}:

A. Sort the values (showing the procedure, step-by-step), using merge sort. For the recursion's base case, you may use the point where the array size reaches two, at which point sorting boils down to a conditional swap of the values.
B. Run the first iteration of quick sort, using pivot as 48. Show the step-by-step procedure.

Answer 2.

A. Merge Sort step-by-step:

Merge Sort is a divide-and-conquer sorting algorithm that recursively divides the input array into two halves, sorts them independently, and then merges the sorted halves back together. The base case for the recursion is when the array size reaches one or two, as sorting an array of one or two elements is trivial.

Given the array: {48, 23, 17, 26, 14, 87, 53, 89}, we'll go through the steps of the Merge Sort algorithm:

1. Divide the array into two halves:

   {48, 23, 17, 26} and {14, 87, 53, 89}

2. Recursively sort each half:

   {17, 23, 26, 48} and {14, 53, 87, 89}

3. Merge the two sorted halves:

   {14, 17, 23, 26, 48, 53, 87, 89}

B. Quick Sort first iteration step-by-step:

Quick Sort is also a divide-and-conquer sorting algorithm that chooses a pivot element and partitions the array around the pivot. Elements smaller than the pivot are moved to the left, and elements greater than the pivot are moved to the right. The pivot is then placed in its final sorted position.

For the array: {48, 23, 17, 26, 14, 87, 53, 89}, let's run the first iteration of Quick Sort using 48 as the pivot:

1. Select the pivot (48).

   {48, 23, 17, 26, 14, 87, 53, 89}

2. Partition the array around the pivot (48):

{17, 26, 14, 23} [48] {87, 53, 89}

3. The pivot (48) is in its final sorted position.

4. Recursively apply Quick Sort to the two partitions:

Left partition: {17, 26, 14, 23}

Right partition: {87, 53, 89}

For brevity, the subsequent iterations of Quick Sort would be required to fully sort the array.

Question 3

A. How many different orderings of the four numbers {1, 2, 3, 4} are there?

B. By considering all those possible orderings, draw all possible binary search trees of size four with nodes labeled by the four numbers {1, 2, 3, 4}. After discarding any duplicate trees, how many different binary search trees of size four are there?

C. For each different tree, state its height, how many leaf nodes it has, and whether it is perfectly balanced.

Answer 3.

A. To find the number of different orderings (permutations) of the four numbers {1, 2, 3, 4}, we use the concept of permutations. The number of permutations of n distinct objects is given by n!, which represents the factorial of n.

For n = 4:

4! = 4 x 3 x 2 x 1 = 24

So, there are 24 different orderings of the four numbers {1, 2, 3, 4}.

B. By considering all those possible orderings, draw all possible binary search trees of size four with nodes labeled by the four numbers {1, 2, 3, 4}. After discarding any duplicate trees, how many different binary search trees of size four are there?

To construct all possible binary search trees (BSTs) with four nodes labeled {1, 2, 3, 4}, we need to consider all 24 different orderings from part A and build unique BSTs based on those orderings.

Let's draw the possible BSTs:

(Note: I'll represent the trees in a textual form, using indentation for left and right children.)

1. BST 1:

```
  2
 / \
1   4
   /
  3
```

2. BST 2:

```
```

```
  2
 / \
1   3
     \
      4
```

3. BST 3:

```
  3
 / \
2   4
/
1
```

4. BST 4:

```
  3
 / \
1   4
 \
  2
```

5. BST 5:

```
  4
 / \
2   3
/ \
1   4
```

6. BST 6:

```
  4
 / \
1   3
   / \
  2   4
```

So, there are 6 different binary search trees of size four after discarding duplicates.

C. For each different tree, state its height, how many leaf nodes it has, and whether it is perfectly balanced.

1. BST 1:

- Height: 2

- Number of leaf nodes: 2
- Perfectly balanced: No

2. BST 2:
   - Height: 2
   - Number of leaf nodes: 2
   - Perfectly balanced: No

3. BST 3:
   - Height: 2
   - Number of leaf nodes: 2
   - Perfectly balanced: No

4. BST 4:
   - Height: 2
   - Number of leaf nodes: 2
   - Perfectly balanced: No

5. BST 5:
   - Height: 3
   - Number of leaf nodes: 2
   - Perfectly balanced: No

6. BST 6:
   - Height: 3

- Number of leaf nodes: 2
- Perfectly balanced: No

A BST is considered perfectly balanced if the heights of the two subtrees of every node differ by at most one. In this case, none of the BSTs is perfectly balanced since the heights of their subtrees are different.

Question 4.

A. Construct AVL Tree for the following numbers 14, 8, 12, 46, 23, 5, 77, 88, 20
B. Is Complete binary tree a AVL tree? What are the difference between Maxheap, Complete, strict, and full binary tree?

Answer 4.

A. To construct an AVL Tree, we need to insert the given numbers in a way that maintains the AVL property (the heights of the left and right subtrees of every node differ by at most one). Here's the step-by-step construction of the AVL Tree:

1. Insert 14:

```
  14
```

2. Insert 8:

```
  14
 /
```

```
 8
```

3. Insert 12:

```
   14
  / \
 8   12
```

4. Insert 46:

```
   14
  / \
 8   46
    /
   12
```

5. Insert 23:

```
   14
  / \
 8   46
```

```
     /
  12
  /
23
```

6. Insert 5:

```
    14
   / \
 8   46
 /\   \
5  12   77
  /   /
 23
```

7. Insert 77:

```
    14
   / \
 8   46
/ \   \
5  12   77
```

```
  /   / \
 23   20  88
```

8. Insert 88:

```
    14
   / \
  8   46
 / \   \
5  12   77
  /   / \
 23   20  88
           \
            20
```

9. Insert 20:

```
    14
   / \
  8   46
 / \   \
5  12   77
```

```
  /   / \
 23   20  88
        / \
       20  23
```

The AVL Tree is now constructed, and the tree satisfies the AVL property at every node.

B.

- Complete Binary Tree:

A complete binary tree is a binary tree in which every level is completely filled, except possibly the last level, and the nodes at the last level are left-aligned. It is not necessary for the tree to be balanced in terms of heights. A complete binary tree can have fewer nodes compared to a full binary tree.

- AVL Tree:

An AVL tree is a self-balancing binary search tree where the heights of the left and right subtrees of every node differ by at most one. It ensures that the tree remains balanced, leading to efficient search, insert, and delete operations.

- Max Heap (Max Binary Heap):

A max heap is a binary tree where each parent node is greater than or equal to its children (in case of a max binary heap). The root node represents the maximum element in the heap. It is not necessary for the tree to be balanced or complete.

- Strict Binary Tree:

A strict binary tree is a binary tree in which each node has exactly two children, except for the leaf nodes. In other words, every non-leaf node has both a left child and a right child.

- Full Binary Tree:

A full binary tree is a binary tree in which every non-leaf node has exactly two children. All the leaf nodes are at the same level.

Differences:

1. AVL Tree is a self-balancing binary search tree, whereas a complete binary tree does not guarantee balanced heights.

2. A complete binary tree can have fewer nodes compared to a full binary tree.

3. A strict binary tree has exactly two children for each non-leaf node, while a full binary tree has exactly two children for all non-leaf nodes.

4. A max heap is not necessarily a balanced tree but ensures that the parent node's value is greater than or equal to its children.

Question 5.

A. Draw the binary search tree that results from inserting the items [19, 30, 36, 10, 40, 25, 33] in that order into an initially empty tree.

B. Show how the tree rotation approach can be used to balance that tree.

Answer.

A. To construct the binary search tree (BST) resulting from inserting the items [19, 30, 36, 10, 40, 25, 33] in that order into an initially empty tree, follow the insertion process for each element:

1. Insert 19:

```
   19
```

2. Insert 30:

```
   19
     \
      30
```

3. Insert 36:

```
   19
     \
      30
        \
         36
```

4. Insert 10:

```
```

```
        19
        / \
     10    30
             \
              36
```

5. Insert 40:

```
      19
      / \
   10    30
           \
           36
             \
              40
```

6. Insert 25:

```
     19
    /  \
 10    30
       /  \
```

```
    25   36
          \
           40
```

7. Insert 33:

```
    19
   /  \
 10    30
      /  \
     25   36
        / \
       33  40
```

The binary search tree after inserting all the items is now balanced and maintains the BST property, where the left subtree of a node contains only nodes with keys less than the node's key, and the right subtree contains only nodes with keys greater than the node's key.

B. Tree Rotation Approach to Balance the Tree:

To balance the tree, we use rotation operations to maintain the AVL property (the heights of the left and right subtrees of every node differ by at most one). In this

case, the AVL tree is already balanced, so we do not need to perform any rotation operations.

If the tree were not balanced, and AVL property was violated (the difference in heights of left and right subtrees exceeded 1), we would use rotation operations like "left rotation," "right rotation," "left-right rotation," or "right-left rotation" to bring the tree back into balance. These rotations help to maintain the AVL property and ensure efficient search, insert, and delete operations in the AVL tree.

Question 6.

Consider the polynomial 5y2 − 3y + 2.

i. Write the polynomial as an expression tree that obeys the usual ordering of operations. Hint: to clarify the ordering for yourself, first write the expression with brackets indicating the order of operations.

ii. Write the polynomial as postfix expression.

Answer.

i. Expression Tree for the Polynomial 5y^2 - 3y + 2:

To represent the polynomial as an expression tree, we need to arrange the terms and operations according to the usual order of operations (PEMDAS/BODMAS). Let's first write the polynomial with brackets to indicate the order of operations:

Expression with brackets: 5 * y^2 - 3 * y + 2

Now, we can construct the expression tree:

```
        -
       / \
      *   2
     /\
    5  y^2
       \
        y
```

In this expression tree, each node represents an operation or a constant term, and the branches represent the operands. The tree is constructed in such a way that the operations are performed following the usual order of operations: first, exponentiation (y^2), then multiplication (5 * y^2), followed by subtraction (-3 * y), and finally addition (5 * y^2 - 3 * y + 2).

ii. Postfix Expression for the Polynomial 5y^2 - 3y + 2:

To convert the polynomial into postfix notation, we use the Reverse Polish Notation (RPN) approach. In postfix notation, the operators are written after their operands.

The original expression is: 5 * y^2 - 3 * y + 2

The postfix expression would be: 5 y^2 * 3 y * - 2 +

So, the polynomial 5y^2 - 3y + 2 in postfix notation is: 5 y^2 * 3 y * - 2 +.

Question 7.

A B-tree of order 4 is built from scratch by 10 successive insertions [10, 20, 30, 40, 25, 15, 05, 02, 07, 1]. What is the maximum number of node splitting operations that may take place?

Count of node splitting in B/B+ tree and show step by step.

Answer.

In a B-tree of order m, each node (except the root) can have at most m-1 keys and m children. When a node overflows due to an insertion, it needs to split into two nodes. The maximum number of node splitting operations that may take place during 10 successive insertions in a B-tree of order 4 can be calculated as follows:

1. Insert 10:

```
    10
```

2. Insert 20:

```
    10, 20
```

3. Insert 30:

```
      10, 20, 30
```

4. Insert 40:

```
       20
     /   \
   10     30, 40
```

5. Insert 25 (Node Splitting Operation 1):

```
       20
     /    \
   10, 15   30, 40
      \
       25
```

6. Insert 15 (Node Splitting Operation 2):

```
        15, 20
       /     \
```

```
   10       30, 40
     \
      25
```

7. Insert 05 (Node Splitting Operation 3):

```
        15, 20
       /      \
   05, 10      30, 40
       \
        25
```

8. Insert 02 (Node Splitting Operation 4):

```
        10, 15, 20
       /          \
   02, 05      25, 30, 40
```

9. Insert 07 (Node Splitting Operation 5):

```
        10, 15, 20
       /          \
```

```
   02, 05, 07   25, 30, 40
```

10. Insert 1 (Node Splitting Operation 6):

```
            10, 15, 20
           /        \
      02, 05, 07   25, 30, 40
     /
   1
```

After the 10 successive insertions, a total of 6 node splitting operations have taken place in the B-tree of order 4. Each time a node splits, it requires an additional level to be added to the tree, which helps maintain the balance and order of the B-tree.
