National University of Computer and Emerging Sciences

Karachi Campus

Data Structures (CS2001)

Sessional-II Exam

Date: Nov'4th 2024

Total Time: 1 hour

Total Points: 30

Total Questions: **03**

Course Instructor(s)

Dr. Jawwad A. Shamsi, Dr. Abdul Aziz, Dr. Farrukh Hasan, Ms. Mubashra Fayyaz, Ms. Ayesha Ali, Ms. Rafia Shaikh, Mr. Nouman Rajput

Semester: Fall-2024

Dept: CS / AI/ CYS/ SE

_____ _____ _____ _____

Student Name Roll No Section Student Signature

Do not write below this line

Attempt all questions in order. Out of order answers will not be graded.

CLO #1: Use & explain concepts related to basic and advanced data structures and describe their usage in terms of common algorithmic operations

Question 1: **[10 points]**

Write the code to use Quicksort to find the median of an unsorted array without necessarily sorting all the elements. Explain the logic of your code separately.

[Solution]

1. Determine the Median Position:
    a. For an array of n elements:
        i. For an odd number of elements: The median index is n / 2.
        ii. For an even number of elements: The median is the average of the elements at indices n / 2 - 1 and n / 2.
2. Partitioning:
    a. Choose a pivot element in the array, as in QuickSort, and partition the array around this pivot:
        i. Elements less than the pivot go to its left.
        ii. Elements greater than the pivot go to its right.
    b. After partitioning, the pivot is in its correct sorted position in the array.
3. QuickSelect Process:
    a. Check the pivot's index:
        i. If it matches the median index (or indices), we've found the median.
        ii. If the pivot index is greater than the median index, recursively apply QuickSelect on the left subarray.
        iii. If the pivot index is less than the median index, apply QuickSelect on the right subarray.
4. Continue Until Median is Found:
    a. Repeat the partitioning and selection process only on the side of the array where the median lies. This reduces the problem size by half in each step, similar to binary search.

Complexity

The average time complexity of this QuickSelect method is O(n), which is more efficient than fully sorting the array (which would take O(nlogn) time).

Code:

```
#include <iostream>
using namespace std;

// Custom swap function to swap elements
void mySwap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

// Function to partition the array around the pivot element
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choosing the pivot as the last element
    int i = low - 1; // Index of the smaller element

    for (int j = low; j < high; ++j) {
        if (arr[j] < pivot) {
            ++i;
            mySwap(arr[i], arr[j]); // Swap elements if current element is less than pivot
        }
    }
    mySwap(arr[i + 1], arr[high]); // Swap the pivot element to its correct position
    return i + 1; // Return the pivot's index
}

// Function to find the median using QuickSelect
int quickSelect(int arr[], int low, int high, int medianIndex) {
    if (low == high) {
        return arr[low]; // If there's only one element, return it
    }

    int pivotIndex = partition(arr, low, high); // Partitioning the array

    if (pivotIndex == medianIndex) {
        return arr[pivotIndex]; // If pivot index is the median index, return the median
    }
    else if (pivotIndex > medianIndex) {
        return quickSelect(arr, low, pivotIndex - 1, medianIndex); // Recursively search in the left
subarray
    }
    else {
        return quickSelect(arr, pivotIndex + 1, high, medianIndex); // Recursively search in the right
subarray
    }
}
```

```
// Function to find the median of an array
double findMedian(int arr[], int n) {
    int medianIndex;

    // Determine the median index according to the given logic
    if (n % 2 != 0) {
        medianIndex = n / 2;  // Odd case (n/2 index)
        return quickSelect(arr, 0, n - 1, medianIndex);  // Calling QuickSelect to find the median
    } else {
        int leftMedian = quickSelect(arr, 0, n - 1, n / 2 - 1);
        int rightMedian = quickSelect(arr, 0, n - 1, n / 2);
        return (leftMedian + rightMedian) / 2.0;  // Even case: Average of two middle elements
    }
}
//Driver Code
int main() {
    // Example array
    int arr[] = {12, 3, 5, 60, 7, 19};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Finding the median
    double median = findMedian(arr, n);

    // Output the result
    cout << "Median is: " << median << endl;

    return 0;
}
```

CLO 1#: Use & explain concepts related to basic and advanced data structures and describe their usage in terms of common algorithmic operations

Question 2: **[10 points]**

Given a singly linked list of n nodes sorted in ascending order, write code to construct a binary search tree (BST) from this linked list such that it is not showing Skew-ness anywhere and it's balanced. (Note: Do not use AVL or any other self-balancing tree).

[Solution]

Basic idea: First, we'll have to iterate once through the linked list to count the total number of nodes (count). Then, we can define our recursive helper (treeify())using index numbers as our arguments. Even though we won't be able to access the listnodes directly by index number, we can take advantage of an inorder tree traversal to force our access to go in iterative order.

We'll need to have our list pointer (curr) have global scope in order to update properly via recursion. In an inorder traversal, we recursively process the left subtree, then process the middle node, then recursively process the right subtree. For this solution, we'll just need to make sure we move curr to curr.next at the end of processing the middle node.

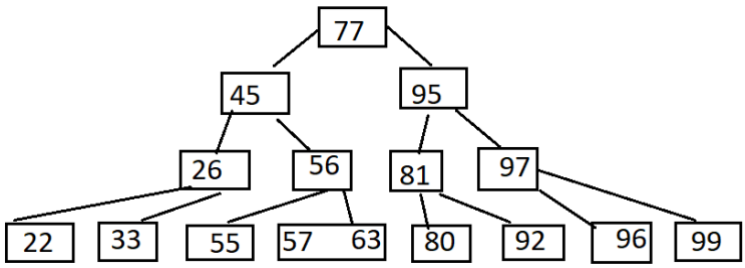
Code:

```
TreeNode* sortedListToBST(ListNode* head) {
    int count = 0;
    curr = head;
    while (curr) curr = curr->next, count++;
    curr = head;
    return treeify(1, count);
  }
TreeNode* treeify(int i, int j) {
    if (j < i) return nullptr;
    int mid = (i + j) >> 1;
    TreeNode* node = new TreeNode();
    node->left = treeify(i, mid - 1);
    node->val = curr->val, curr = curr->next;
    node->right = treeify(mid + 1, j);
    return node;
  }
```

CLO 3#: Compare different data structures in terms of their relative efficiency and design effective solutions and algorithms that make use of them.

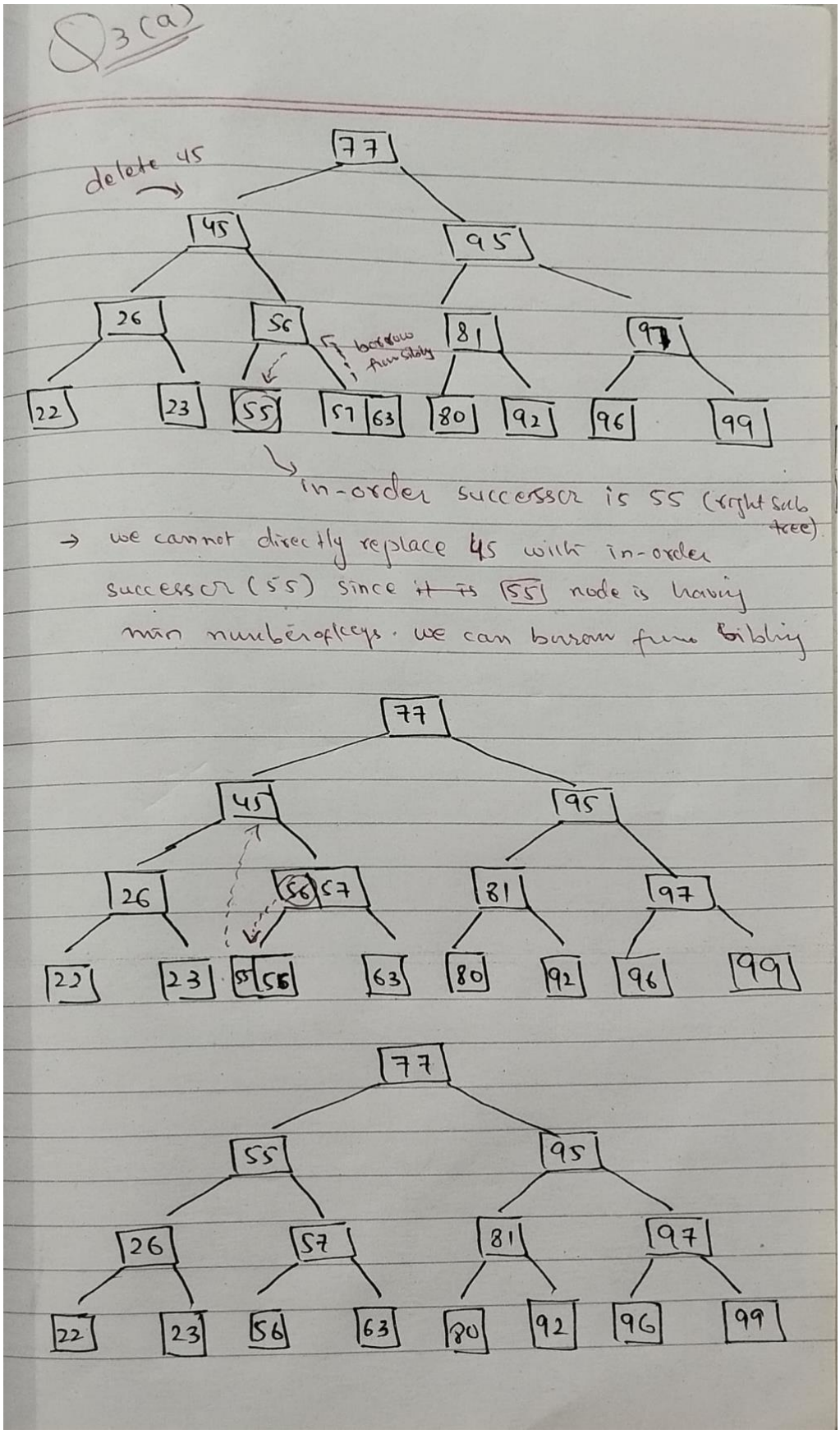
Question 3 Do as directed: **[3+3+4 = 10 points]**

a. Given the 2-3 tree below, delete 45. Show each step of the process clearly and the final resulting tree by drawing each step clearly.

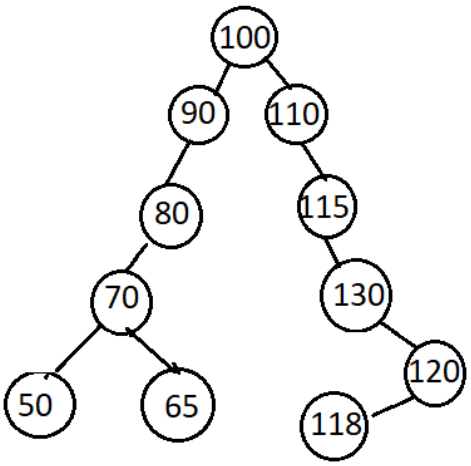


[Solution]

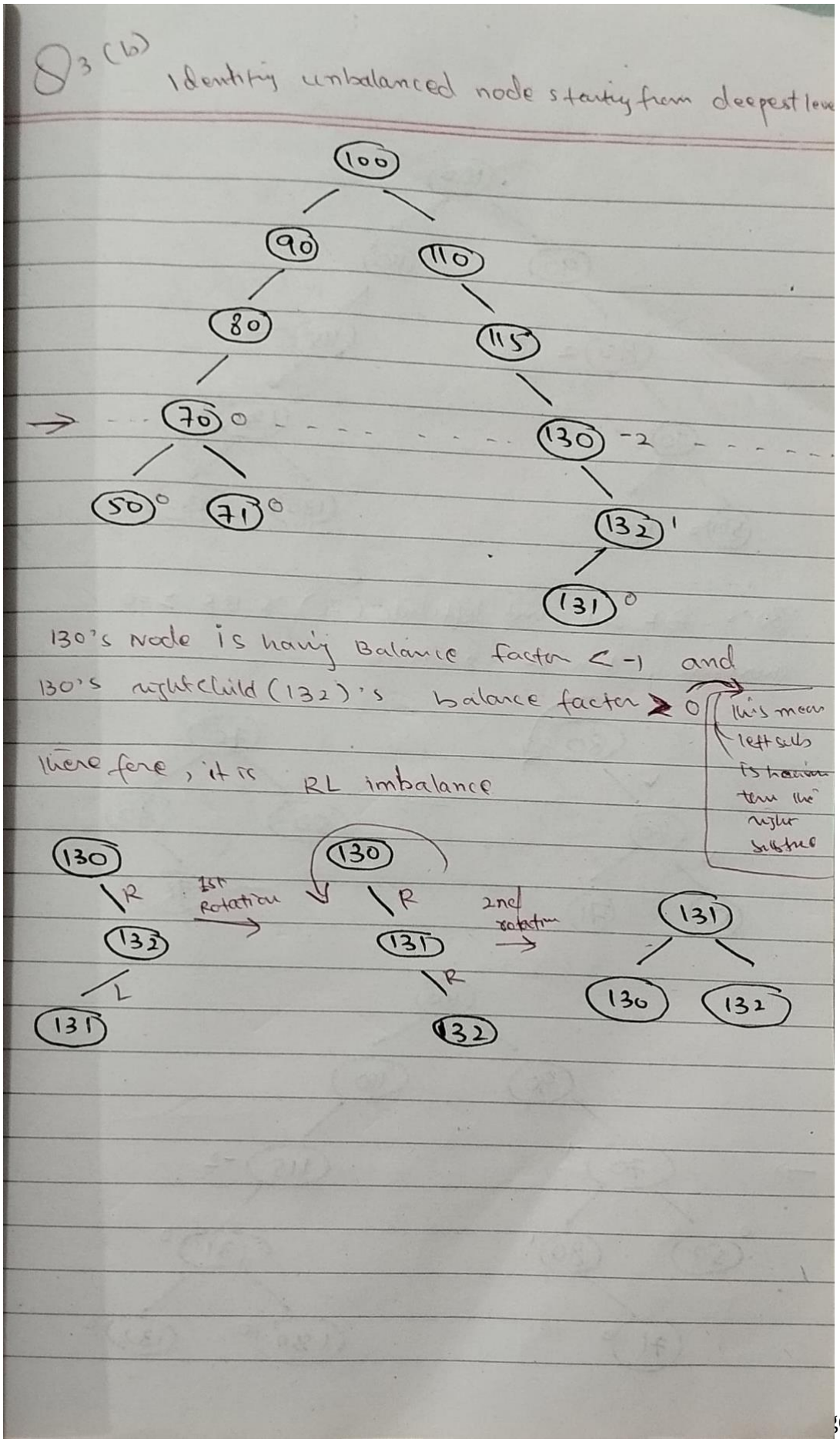
Q3 (a)

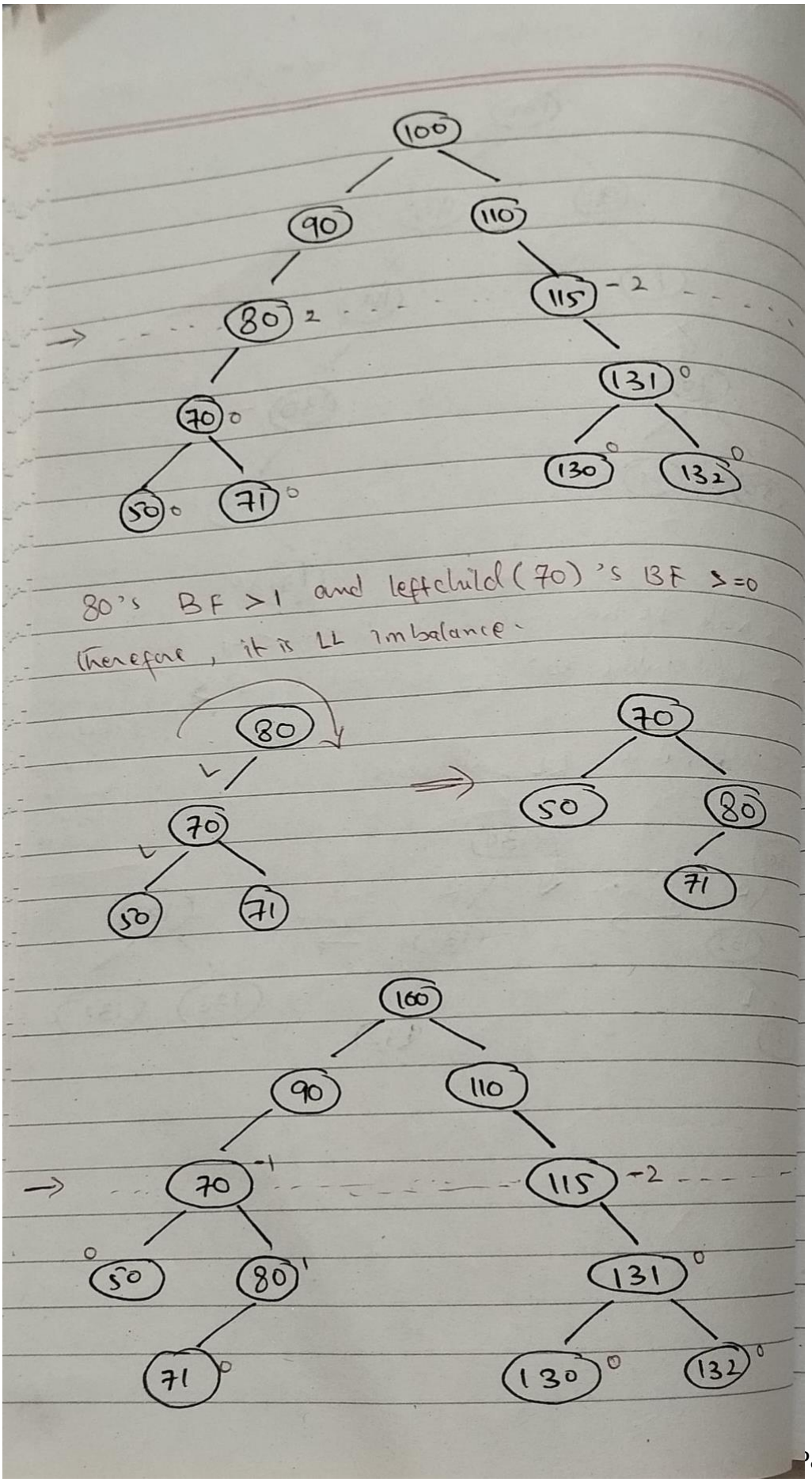

delete 45
77
45
95
26
56
81
97
22
23
55
57
63
80
92
96
99
borrow from sibling
in-order successor is 55 (right sub tree)
→ we cannot directly replace 45 with in-order successor (55) since it is 55 node is having min number of keys. we can borrow from sibling
77
45
95
26
56
57
81
97
22
23
55
56
63
80
92
96
99
77
55
95
26
57
81
97
22
23
56
63
80
92
96
99

b. Balance the following tree using AVL rotations.



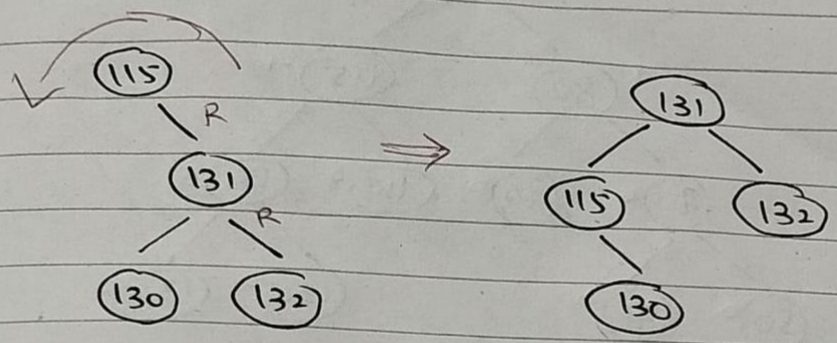
[Solution]


Q3 (b)
Identify unbalanced node starting from deepest level
100
90
110
80
115
70 0
130 -2
50 0
71 0
132 1
131 0
130's node is having Balance factor < -1 and
130's right child (132)'s balance factor ≥ 0
this means left subtree is heavier than the right subtree
Therefore, it is RL imbalance
130
R
132
L
131
1st rotation
130
R
131
R
132
2nd rotation
131
130
132

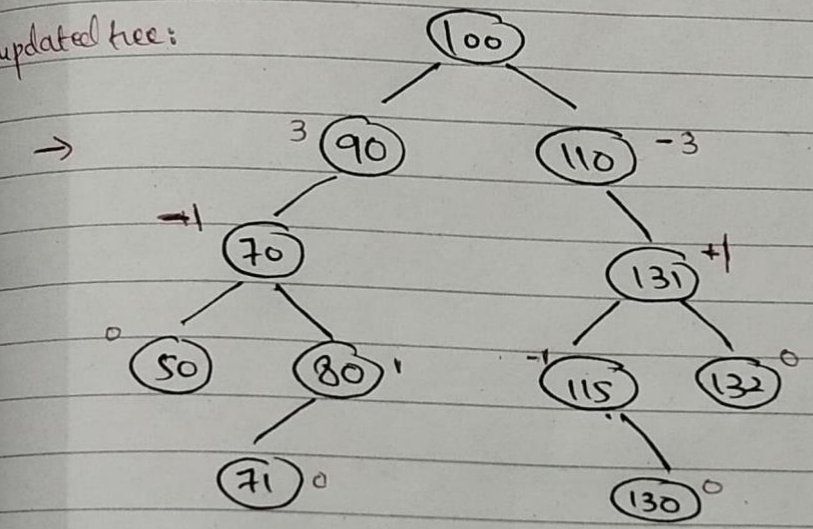

100
90
110
80
2
115
-2
70
0
131
0
50
0
71
0
130
0
132
0
80's BF >1 and leftchild (70)'s BF >=0
Therefore, it is LL imbalance.
80
L
70
L
50
71
70
50
80
71
100
90
110
70
-1
115
-2
50
0
80
1
131
0
71
0
130
0
132
0

115's BF < -1 and rightchild(131)'s BF <=0

Therefore, it is RR imbalance.

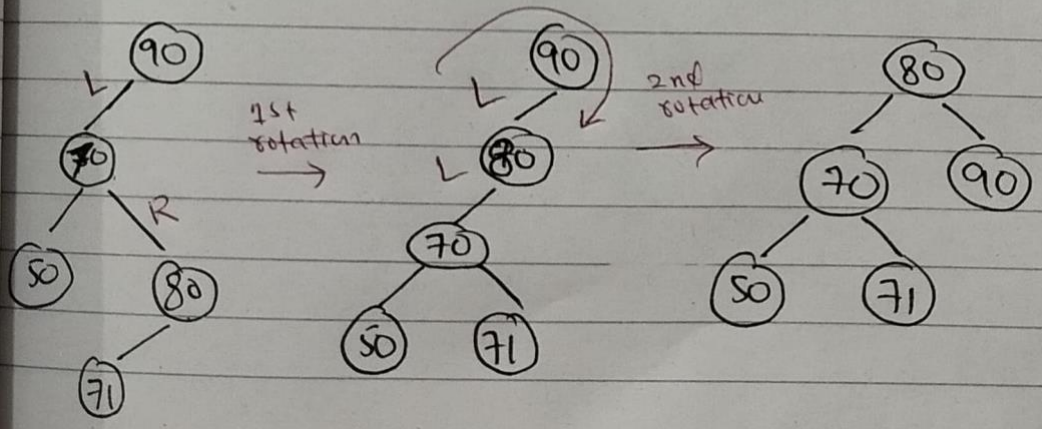


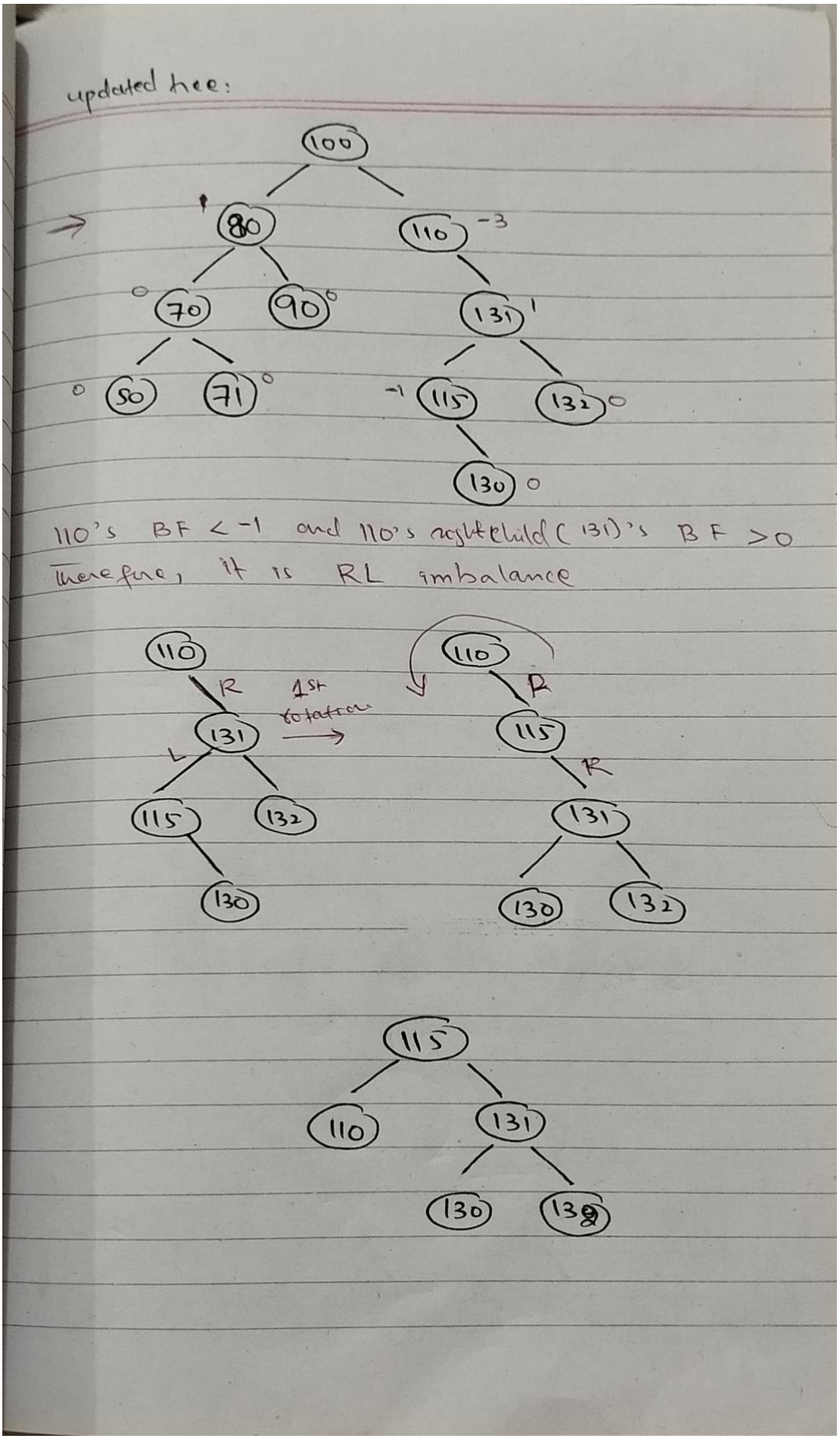
updated tree:

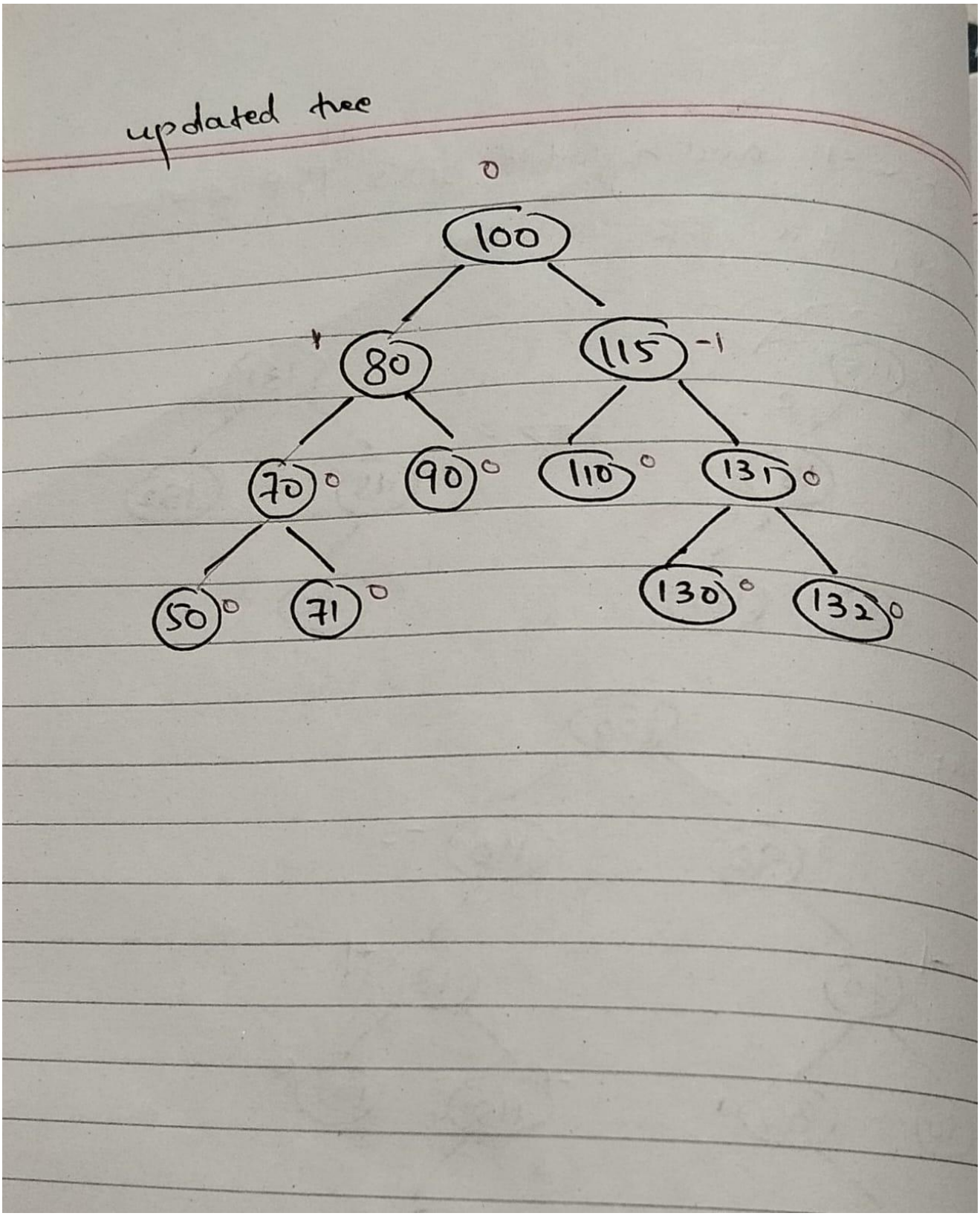


90's BF >1 and 90's leftchild(70)'s BF < 0

Therefore, it is LR imbalance




updated tree:
100
80
110
-3
70
90
131
1
50
71
-1
115
132
130
110's BF < -1 and 110's right child (131)'s BF > 0
Therefore, it is RL imbalance
110
R
131
L
115
132
130
1st rotation
110
R
115
R
131
130
132
115
110
131
130
139



c. Using a B-tree of order m=5, show step by step insertion for the following sequence:

IDs: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100.

[Solution]

Q 3.(c)

order $m = 5$

IDs : 10, 20, 30, 40, 50, 60, 70, 80, 90, 100

max no. of children nodes = $m \Rightarrow 5$

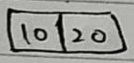
min no. of children nodes = $\lceil \frac{m}{2} \rceil = \lceil \frac{5}{2} \rceil = 3$ (for non-root nodes)

max no. of keys = $m - 1 = 5 - 1 \Rightarrow 4$

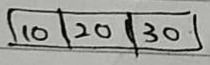
min no. of key = $\lceil \frac{m}{2} \rceil - 1 \Rightarrow \lceil \frac{5}{2} \rceil - 1 \Rightarrow 2$ (for non-root nodes)

1- insert 10 :

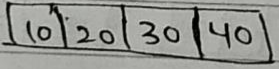
2- insert 20 :



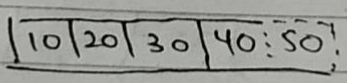
3- insert 30 :



4- insert 40 :



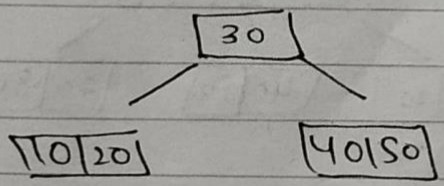
5- insert 50 :



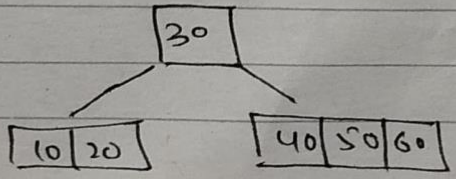
→ overflow; exceed the max limit of key

→ Split node at the median (30)

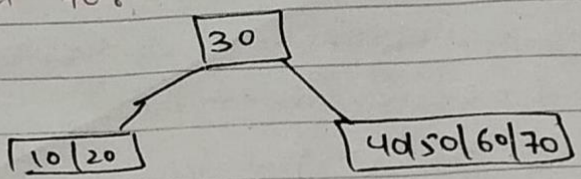
→ We promote 30 to the root.



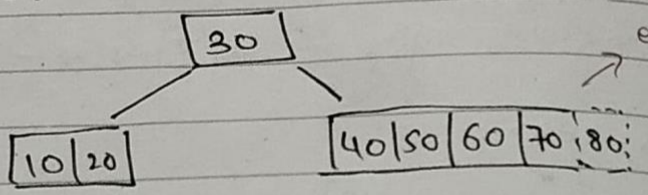
6- insert 60 :



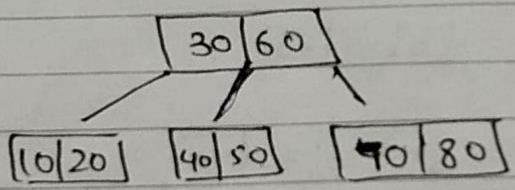
7: insert 70:



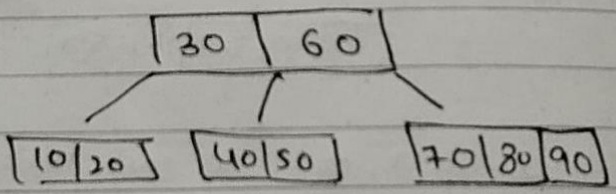
8- insert 80:



→ split node at median (60)
→ then promote 60 to the root.



9- insert 90:



10- insert 100: