Murashko Artem B20-02 ar.murashko@innopolis.university

2.1 Dynamic Programming

1. ***Write down pseudocode for a recursive algorithm that solves the problem.***

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

int findMaxRevenue(vector <int> x, vector <int> r, int d, int currentRevenue,
int indexOfCurrentShop, int indexOfLastShop) {
    if (indexOfCurrentShop < x.size()) {
        int res1 = findMaxRevenue(x, r, d, currentRevenue, indexOfCurrentShop +
1, indexOfLastShop);
        int res2 = -1;
        if (indexOfLastShop == -1 || x[indexOfCurrentShop] - x[indexOfLastShop]
> d)
            res2 = findMaxRevenue(x, r, d, currentRevenue +
r[indexOfCurrentShop], indexOfCurrentShop + 1,
                                  indexOfCurrentShop);
        if (res1 > res2)
            return res1;
        else
            return res2;
    } else
        return currentRevenue;
}

int main() {
    int H, d, N;
    cin >> H >> N;

    vector <int> r(N), x(N); // Array Lists

    for (int i = 0; i < N; ++i)
        cin >> x[i];

    for (int i = 0; i < N; ++i)
        cin >> r[i];

    cin >> d;

    cout << findMaxRevenue(x, r, d, 0, 0, -1);
}
```

2. ***Provide asymptotic worst-case time complexity of the recursive algorithm.***

In the worst case, we will execute two recursive calls at each locations.
Due to that, if we have n locations, then we will have $2 * 2 * \ldots * 2 (n_{times}) = 2^n$ calls.
Thus, time complexity $O(2^n)$ worst case.

3. ***Identify overlapping subproblems.***

Let's say for maximum revenue we need to build 1, 3, 5 shopping centers.

Then, using recursion, we calculate the maximum revenue when calling from 1: {1, 3, 5}, from 3: {3, 5} and from: {5}.

Although, having counted the maximum revenue from 3, we do not need to count the maximum revenue from 5. It turns out that we do the same calculations more than once, which increases the time complexity of the algorithm.

4. ***Write down pseudocode for the optimized algorithm that solves the problem using dynamic programming (top-down or bottom-up). The algorithm should compute both the maximum estimated revenue and the specific locations where shopping centers should be placed.***

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

int main() {
    int H, d, N;
    cin >> H >> N;

    vector <int> r(H + 1), dp(H + 1), x(N); // Array Lists

    for (int i = 0; i < N; ++i)
        cin >> x[i];

    for (int i = 0; i < N; ++i)
        cin >> r[x[i]];

    cin >> d;

    dp[0] = 0;
    for (int i = 1; i <= H; ++i)
        if (i - d - 1 >= 0)
            dp[i] = max(dp[i - d - 1] + r[i], dp[i - 1]);
        else
            dp[i] = max(r[i], dp[i - 1]);

    int shopDist = H;
    vector <int> shops;
    for (int i = N - 1; i >= 0; --i) {
```

```
        if ((dp[x[i] - d - 1] + r[x[i]] == dp[x[i]] || (x[i] - d - 1 <= 0 &&
r[i] == dp[x[i]])) && x[i] <= shopDist) {
            shops.push_back(x[i]);
            shopDist = x[i] - d - 1;
        }
    }

    for (int i = shops.size() - 1; i >= 0; --i)
        cout << shops[i] << " ";

    cout << endl << dp[H - 1];

return 0;
}
```

Solution:

$0 < x_1 < x_2 < \ldots < x_n < H$
Let's go through every kilometer of the road and for each kilometer we will count the maximum revenue, this maximum revenue we will store in the array list dp.
When on x_i km we can build a shopping center, we have 2 options:

1. Build it: $dp[i] = dp[i - d - 1] + r[i]$
2. Don't build it: $dp[i] = dp[i - 1]$
   Our choice is based on whether the shopping center will increase the maximum revenue or not. $dp[0] = 0$ since we cannot build a shopping center there.

Thus, the maximum revenue will be in the $dp[H - 1]$, since $x_n < H$

Now we want get specific locations where shopping centers should be placed. Let's go through all shopping center's from the last to the first one and check the condition: if $dp[x[i] - d - 1] + r[x[i]] = dp[x[i]]$ than we have built shopping center on $x[i]$ km (If we look at the condition under which we built the shopping center it will be the same), also every time we count $shopDist = x[i] - d - 1$, which shows at what minimum distance we could build the next shopping center. If we get $shopDist < 0$, that means that we are on the last segment where can situated the first shop which we could build. So, if $dp[x[i]] = r[i]$ then it's needed shopping center. All in all, If the shop satisfy the condition, we add the store to the list, at the end we output this list from right to left.

5. ***Provide asymptotic worst-case time complexity of the dynamic programming algorithm.***

Find maxRevenue:

```
dp[0] = 0;
for (int i = 1; i <= H; ++i)
  if (i - d - 1 >= 0)
    dp[i] = max(dp[i - d - 1] + r[i], dp[i - 1]);
else
  dp[i] = max(r[i], dp[i - 1]);
```

Time complexity: $O(H)$, since we go through every km of the road.

Getting specific locations:

```
 for (int i = N - 1; i >= 0; --i) {
        if ((dp[x[i] - d - 1] + r[x[i]] == dp[x[i]] || (x[i] - d - 1 <= 0 &&
r[i] == dp[x[i]])) && x[i] <= shopDist) {
            shops.push_back(x[i]);
            shopDist = x[i] - d - 1;
        }
    }
```

Time complexity: $O(N)$, since we go through every distance of shopping centres and push_back method takes constant time in average case.

Time complexity of input/output takes $O(N)$.

All in all, time complexity of dynamic programming method takes $O(H + 3N) = O(4H) = O(H)$ worst case, since $N < H$.

Answer : $O(H)$ **worst case**

2.2 Sorting

1. ***Briefly explain how insertion sort works.***

Suppose we have a sorted array and we want to add an element to its end.
We will swap the added element with the adjacent element to the left until it takes its place.
Consider an unsorted array, then you can take its first element as the sorted part and apply the actions described above to the elements after it.

2. ***Prove that the worst and the best case running times of insertion sort are*** $\Theta(n^2)$ ***and*** $\Theta(n)$ ***, respectively.***

Worst: Suppose each added element is less than all elements to the left (Example: 5 4 3 2 1). So, we need to swap it with the adjacent elements to the left until it comes first. When we add the second element, we need to swap with 1 element. For the third element - 2 swaps, for the fourth - 3 swaps and so on. All in all, if we have array of size n algorithm will take $c * 1 + c * 2 + c * 3 + \ldots + c * (n - 1) = cn^2/2 - cn/2$ operations (c - cost of swap and comparison). Thus, time complexity: $cn^2/2 - cn/2 = \Theta(n^2)$ worst case. (This is worst case, because there are no situations, where we can do more swaps) Q.E.D.

Best: Suppose each added element is more than all elements to the left (Example: 1 2 3 4 5). So, we need only one comparison to understand, that this element on its place. All in all, adding the element will take constant time. Thus, if we have array of size n algorithm will take $c * (n - 1) = cn - c$ operations. Time complexity: $cn - c = \Theta(n)$ best case. (This is best case, because there are no more situations, where adding an element takes constant time) Q.E.D.

3. ***What is a k-sorted array? Is insertion sort fast or slow, relative to its worst-case, when applied to a k-sorted array? Justify your answer by computing the running time of insertion***

sort for a k-sorted array.

A k sorted array is an array where each element is at most k distance away from its target position in the sorted array.

Insertion sort is fast, because we will swap the added element with the adjacent element to the left until it takes its place at **most k times.** That means, $n-1$ times we should swap at most k times. So, overall time complexity $O(nk)$.

4. ***Implement a comparison-based sorting algorithm to sort the following table of data, containing information on events (code, start_date, end_date, sponsor, description). Given the code as the key, provide a pseudocode for comparing two keys that will be used in your sorting algorithm.***

```
#include <iostream>
#include <cstdlib>
#include <vector>
#include <algorithm>
using namespace std;

struct Event { // Struct event, which contains information on event
    string code, date_start, date_end, sponsor, description;
    Event(string code, string date_start, string date_end, string sponsor,
string description) {
        this -> code = code;
        this -> date_start = date_start;
        this -> date_end = date_end;
        this -> sponsor = sponsor;
        this -> description = description;
    }
};

vector <Event> events; // Array list containing objects of class Event

bool comparisonOfCodes(string code1, string code2) {
        char character1 = code1[0]; // get first symbol from code 1
        code1.erase(code1.begin()); // delete first symbol from code1
        int number1 = atoi(code1.c_str()); // get numerical part of code1
        char character2 = code2[0]; // get first symbol from code 2
        code2.erase(code2.begin()); // delete first symbol from code2
        int number2 = atoi(code2.c_str()); // get numerical part of code2

        /*
         * First of all we compare the first character
         * of the codes. If they are different, we can
         * immediately identify which code is greater.
         * If they are equal, then we need to compare
         * numerical part of the codes.
         */
```

```
        if (character1 < character2) // just compare ASCII code of char
            return false;
        else if (character1 > character2)
            return true;
        else {
            if (number1 < number2)
                return false;
            else
                return true;
        }
}

/*
 * Using regular insertion sort.
 * The difference is that we have
 * written our own comparison
 */
void insertionSort(vector <Event> &A) {
    for (int i = 1; i < A.size(); i++) {
        Event key = A[i];
        int j = i - 1;

        while (j >= 0 && comparisonOfCodes(A[j].code, key.code)) {
            A[j + 1] = A[j];
            j = j - 1;
        }

        A[j + 1] = key;
    }
}

int main() {
    // example of events
    events.push_back(Event("A001", "20-03-2021", "27-03-2021", "IU", "Desc1"));
    events.push_back(Event("A123", "20-04-2021", "20-05-2021", "DS", "Desc2"));
    events.push_back(Event("B001", "10-04-2021", "17-04-2021", "MTS",
"Desc3"));
    events.push_back(Event("A009", "12-04-2021", "15-04-2021", "GTK",
"Desc4"));

    insertionSort(events); // starting sorting

    for (const auto& event : events)
        cout << event.code << " " << event.date_start << " " << event.date_end
<< " " << event.sponsor << " " << event.description << endl;
return 0;
}
```

5. ***Write two versions, one recursive, the other iterative, of a boolean function named belongs***

which takes as arguments an array A of ordered integers, a start index from, an end index to and an integer x to search for and which returns true if x is in the array A between index from and index to, and false otherwise.

```
// Simple binary search
bool find_recursive(int A[], int from, int to, int x) {
    if (to >= from) {
        int mid = from + (to - from) / 2;

        if (A[mid] == x)
            return true;

        if (A[mid] > x)
            return find_recursive(A, from, mid - 1, x);
        else
            return find_recursive(A, mid + 1, to, x);
    }

    return false;
}

// Going through all the elements from index 'from' to index 'to'
bool find_iterative(int A[], int from, int to, int x) {
    for (int i = from; i < to; ++i) {
        if (A[i] == x)
            return true;
    }
    return false;
}
```

6. ***Write a variant of the previous function called search which takes the same arguments as belongs, and which returns the index i of the cell containing x if x appears in the array A between the index from and the index to and null otherwise.***

```
int find_recursive(int A[], int from, int to, int x) {
    if (to >= from) {
        int mid = from + (to - from) / 2;

        if (A[mid] == x)
            return mid;

        if (A[mid] > x)
            return find_recursive(A, from, mid - 1, x);
        else
            return find_recursive(A, mid + 1, to, x);
    }
```

```
    return null; // return null
}

int find_iterative(int A[], int from, int to, int x) {
    for (int i = from; i < to; ++i) {
        if (A[i] == x)
            return i;
    }
    return null; // return null
}
```

7. ***Establish the recurrence relation for the running time of search and determine the time complexity by applying Master Theorem.***

$T(n) = T(n/2) + \Theta(1)$
Work done outside the recursive call is $\Theta(1)$, because these are simple operations like return, comparison and calculation of the variable mid. Size of each subproblem is $n/2$ because each call reduces the size of the borders by half.

Applying Master Theorem:
$a = 1, b = 2, log_b a = log_2 1 = 0$.
So, here we should apply second case of Master Theorem.
Thus, time complexity $T(n) = \Theta(logn)$

8. ***Explain the benefit of using search in the context of insertion sorting. Give a version of insertion sorting using search. How does it affect the time complexity of insertion sort?***

Every time we add an element, we can search for $T(n) = \Theta(logn)$ the right place for it in the sorted part (using binary search). But, time complexity will be still $\Theta(n^2)$ worst case, because we need to shift all the elements to add a new one (swap operations).

! **But**, if we should use search from 2.2.6, it will not give us any benefit, because if the element to be inserted is not in the sorted part, then we will have to search iteratively for the place to insert.

```
int find_recursive(int A[], int from, int to, int x) {
   int mid = (to + from) / 2;

   if (to <= from) {
        if (x > A[from])
            return from + 1;
        else
            return from;
    }

    if (A[mid] == x)
        return mid + 1;

    if (A[mid] >= x)
        return find_recursive(A, from, mid - 1, x);
    else
```

```
        return find_recursive(A, mid + 1, to, x);
}

void insertionSort(int A[], int n) {
    for (int i = 1; i < n; ++i) {
        int j = i - 1;
        int currentElement = A[i];
        int locationToInsert = find_recursive(A, 0, j, currentElement);

        while (j >= locationToInsert) { // Due to that, time complexity still
O(n^2) worst case
            A[j + 1] = A[j];
            j--;
        }
        A[j + 1] = currentElement;
    }
}
```

9. ***Is Bubble Sort difficult to parallelize? Why? Provide pseudocode (and explain) a parallel version of bubble sort.***

First of all, the sequential Bubble Sort algoirthm is an algorithm that compares two adjacent elements and swaps them.

Bubble sort is difficult to parallelize since the algorithm has no concurrency (each comparison must be done sequentially, otherwise it will not make sense). But we can divide an array into odd and even and sort them one by one (Using common Bubble sort).

```
// A - array, N - size of A
void parallelBubbleSort(int A[], int N) {
    for (int i = 0; i < N; ++i) {
        if (i % 2 == 1) // if i is odd
            for (int j = 0; j < N / 2; ++j)
                if (A[2*j + 1] > A[2*j + 2]) // compare two adjacent elements
                    swap(A[2*j + 1], A[2*j + 2]); // swap them if needs

        if (i % 2 == 0) // if i is even
            for (int j = 0; j < N / 2; ++j)
                if (A[2*j] > A[2*j + 1]) // compare two adjacent elements
                    swap(A[2*j], A[2*j + 1]); // swap them if needs
    }
}
```

Since each odd and even sort requires $O(n)$ comparisons, overall time complexity will be $O(n^2)$

2.3 Balanced Binary Search Trees

1. ***Explain in words the properties of AVL trees.***

AVL tree is a Binary Search Tree where the difference between heights of left and right subtrees cannot be more than one for all nodes.

2. ***Explain in words or in pseudocode insert operation for AVL trees.***

Insertion consists of two operations: find the place where you need to insert the element (BST insert) and balancing the tree (if necessary)

1) Searching the place and insert: search the place, until there is no child and insert at that point

2) Balancing the tree: go up to the root until we find the node in which the height of the left subtree differs from the right subtree by more than 1. If we find such node, do one of the required rotations: (Left rotation / Right rotation / Left-Right rotation / Right-Left rotation)

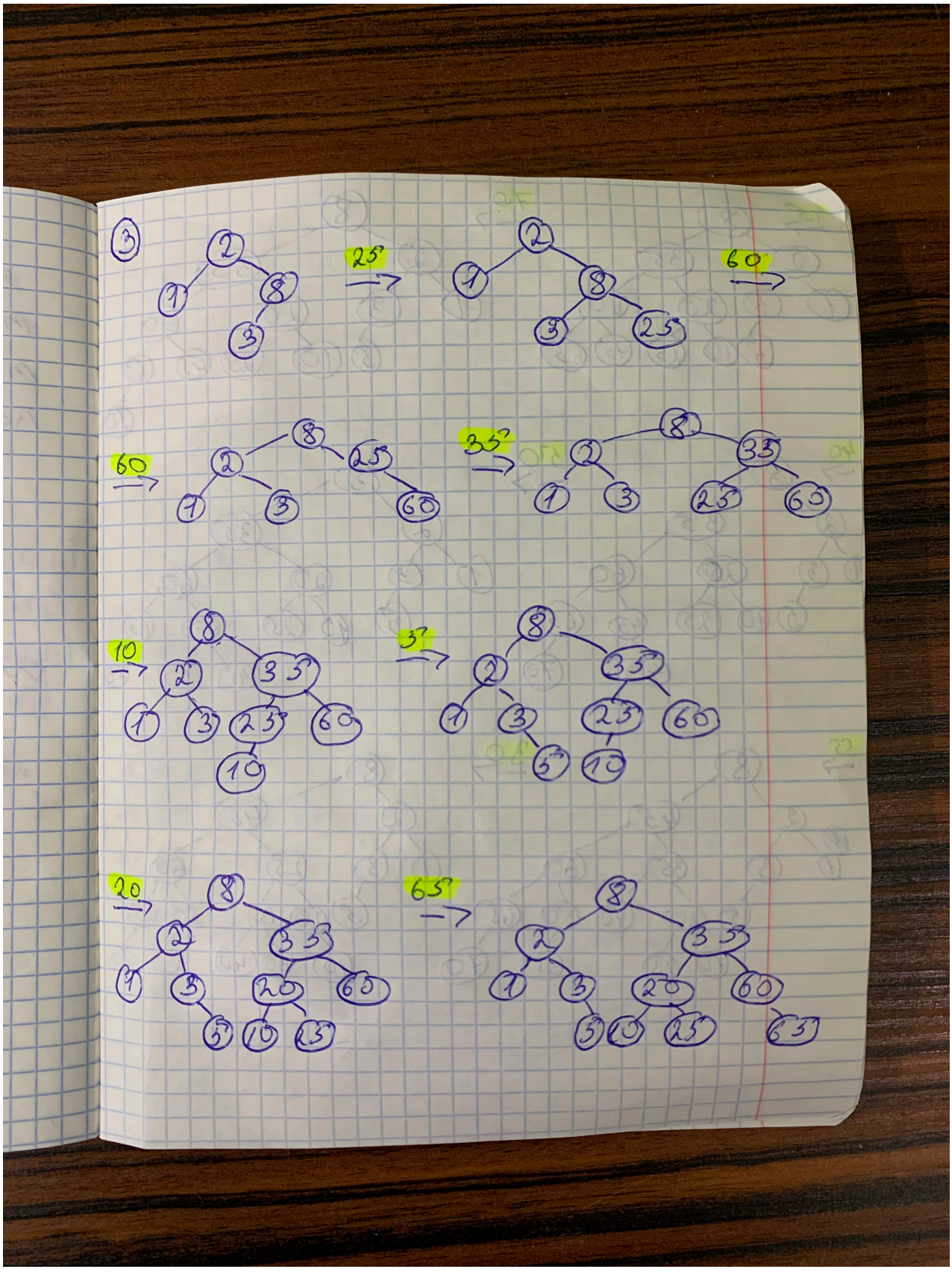
Lets Balance Factor (BF) = height of the left subree - height of the right subtree

Then:

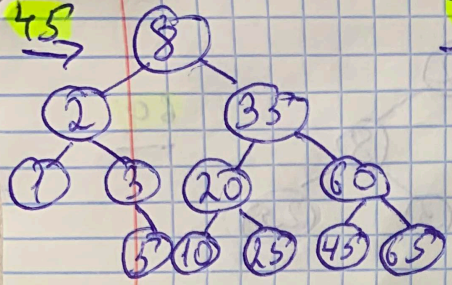
```
if balanceFactor(parent) > 1
  ifparent.left > x
    rightRotate(parent)
  else
    leftRotate(parent.left)
    rightRotate(parent)
else if balanceFactor(parent) < -1
  if parent.right < x
    leftRotate(parent)
  else
    rightRotate(parent.right)
    leftRotate(parent)
```

Till we got to the root, we should check BF for each node doint the steps above.

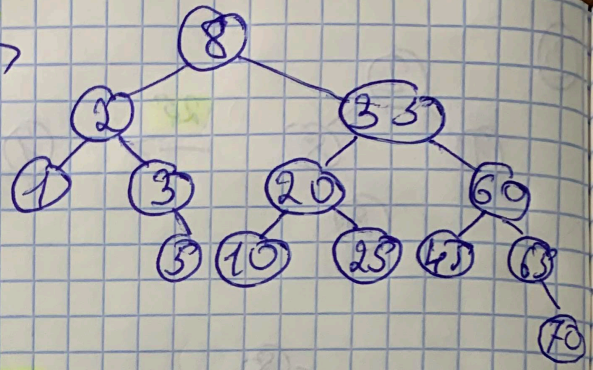
3, 4, 5.


3
2
1
8
3
25
2
1
8
3
25
60
60
8
2
25
1
3
60
35
8
2
35
1
3
25
60
10
8
2
35
1
3
25
60
10
37
8
2
35
1
3
25
60
5
10
20
8
2
35
1
3
20
60
5
10
13
63
8
2
35
1
3
20
60
5
10
25
63

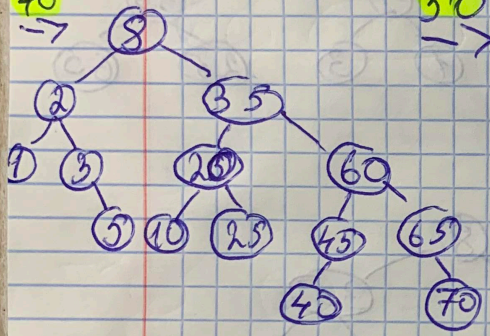
45 →



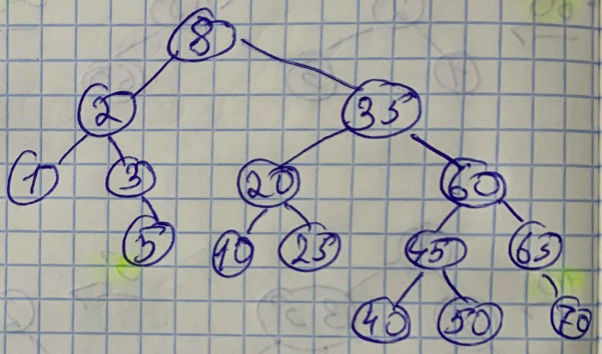
70 →



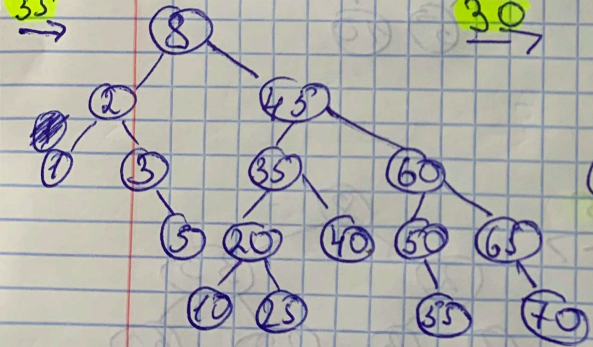
40 →



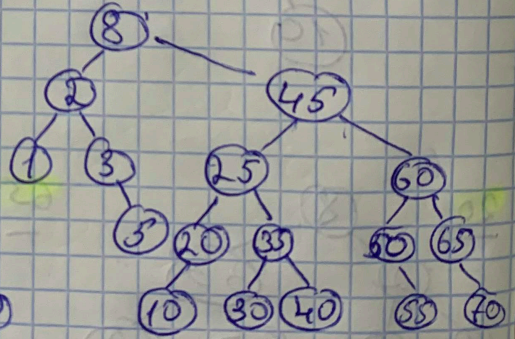
50 →

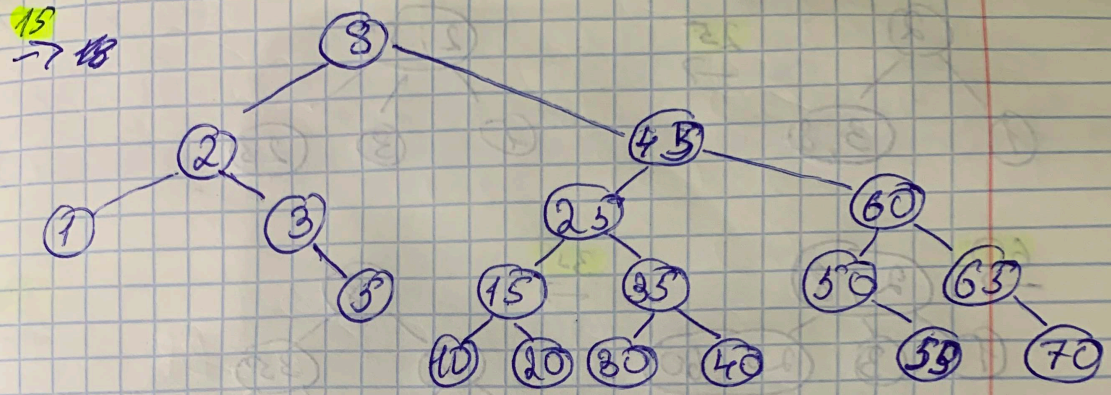


55 →



30 →



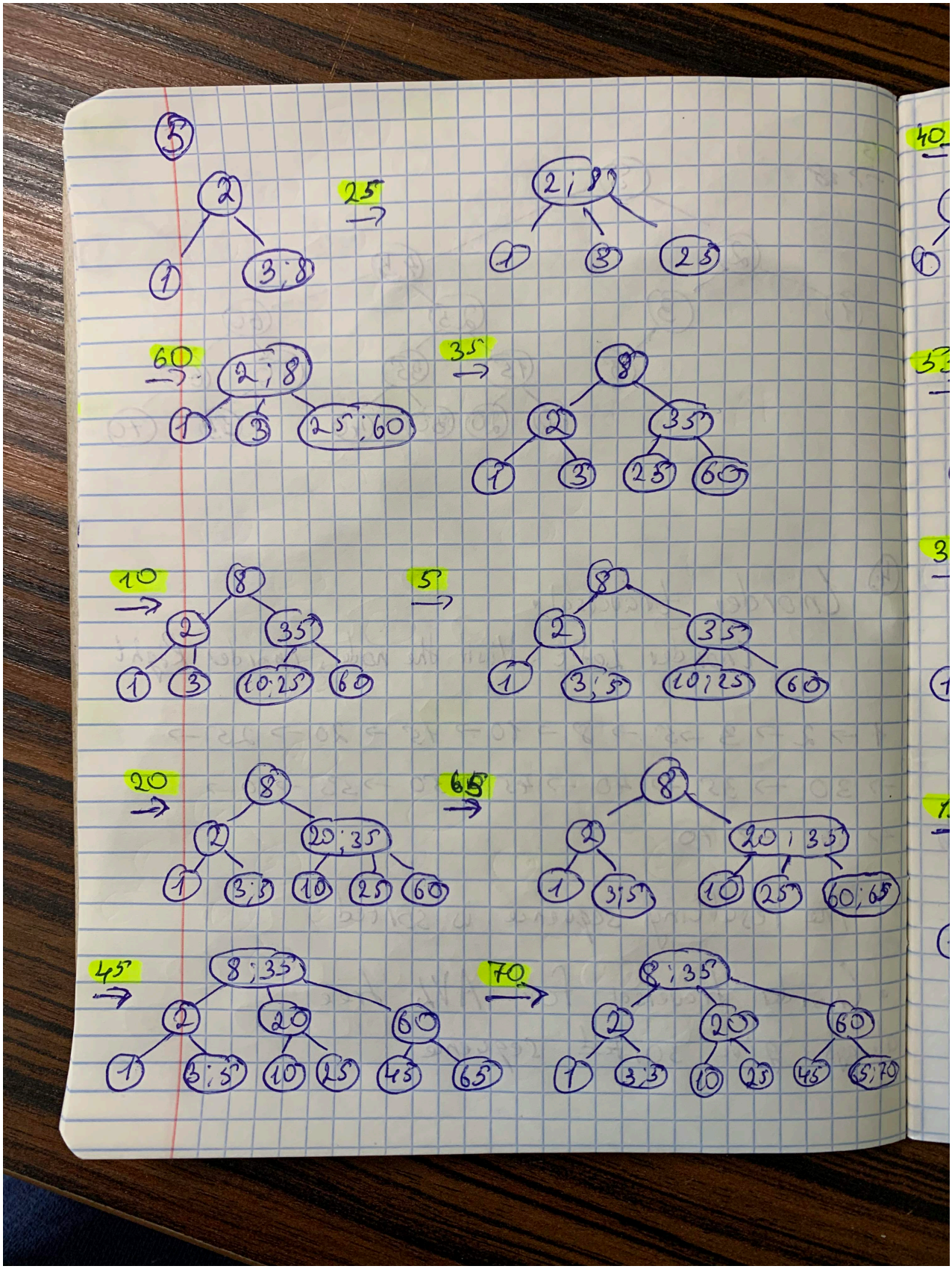


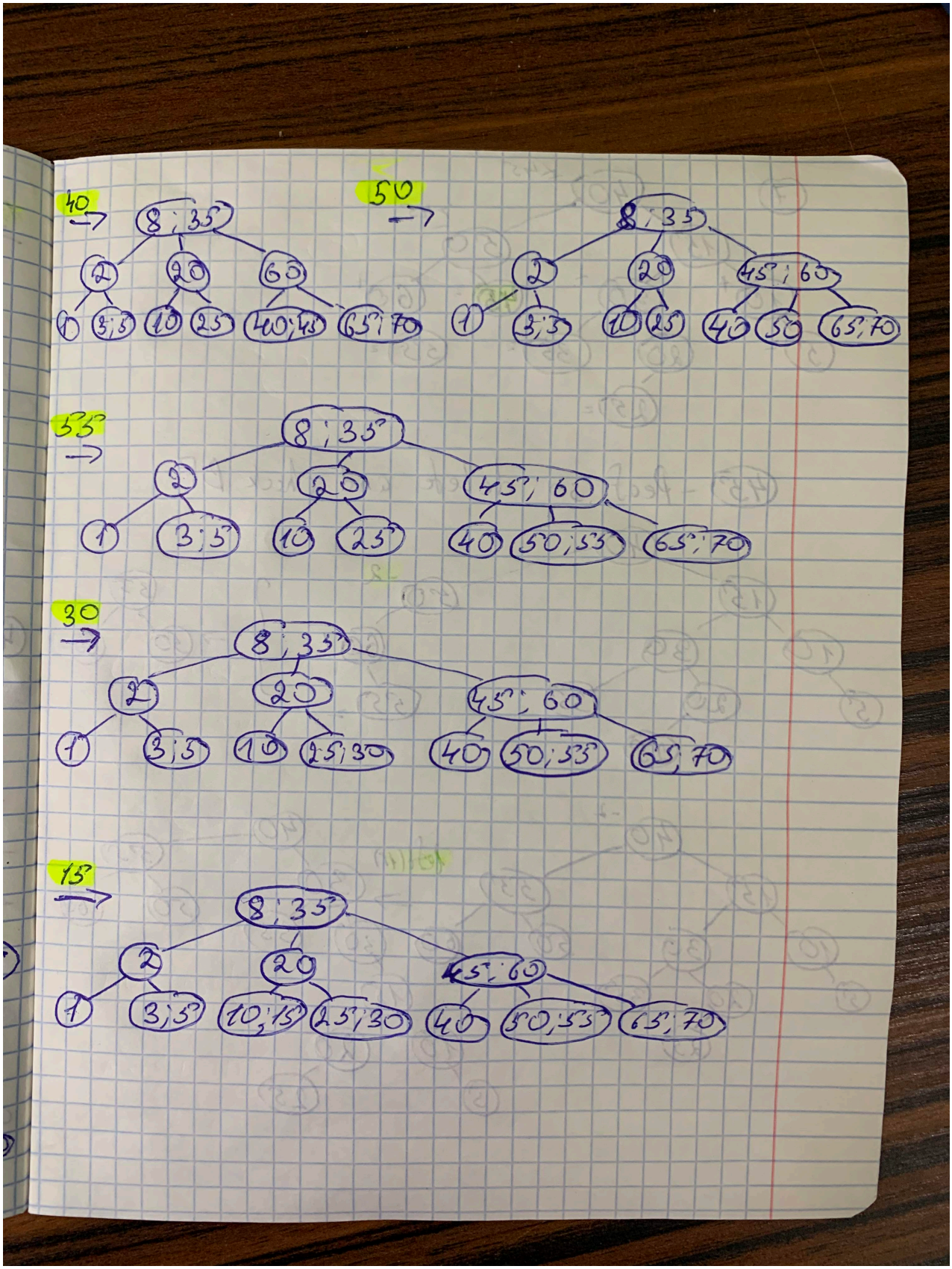
④ Inorder traversal:

Inorder Left, Visit the node, Inorder Right

1 → 2 → 3 → 5 → 8 → 10 → 15 → 20 → 25 →
→ 30 → 35 → 40 → 45 → 50 → 55 → 60 →
→ 65 → 70

- The resulting sequence is sorted
- Inorder traversal for AVL tree always give sorted sequence.

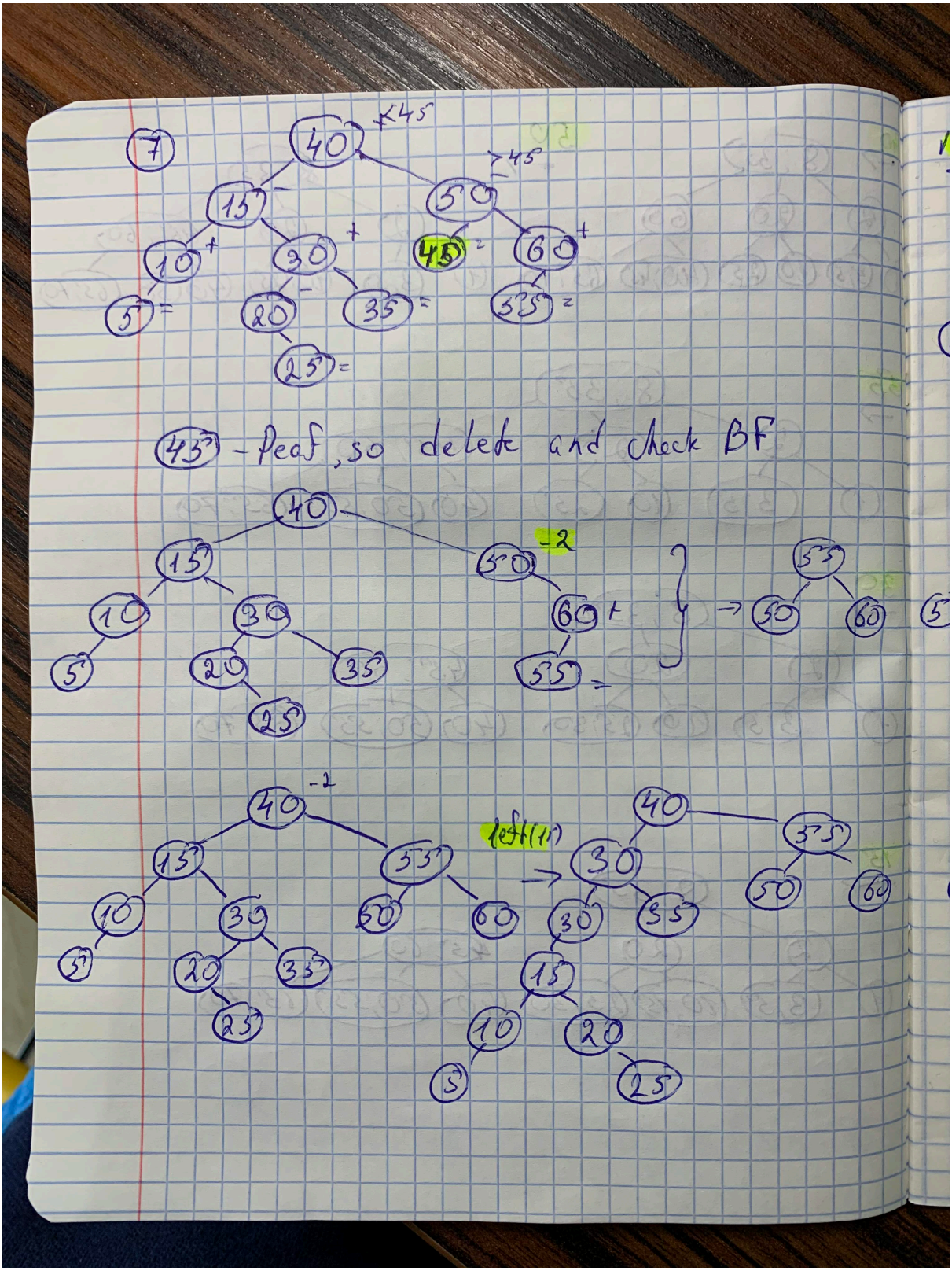

5
2
1
3;8
25
2;8
1
3
25
60
2;8
1
3
25;60
35
8
2
35
1
3
25
60
10
8
2
35
1
3
10;25
60
5
8
2
35
1
3;5
10;25
60
20
8
2
20;35
1
3;5
10
25
60
65
8
2
20;35
1
3;5
10
25
60;65
45
8;35
2
20
60
1
3;5
10
25
45
65
70
8;35
2
20
60
1
3;5
10
25
45
65;70

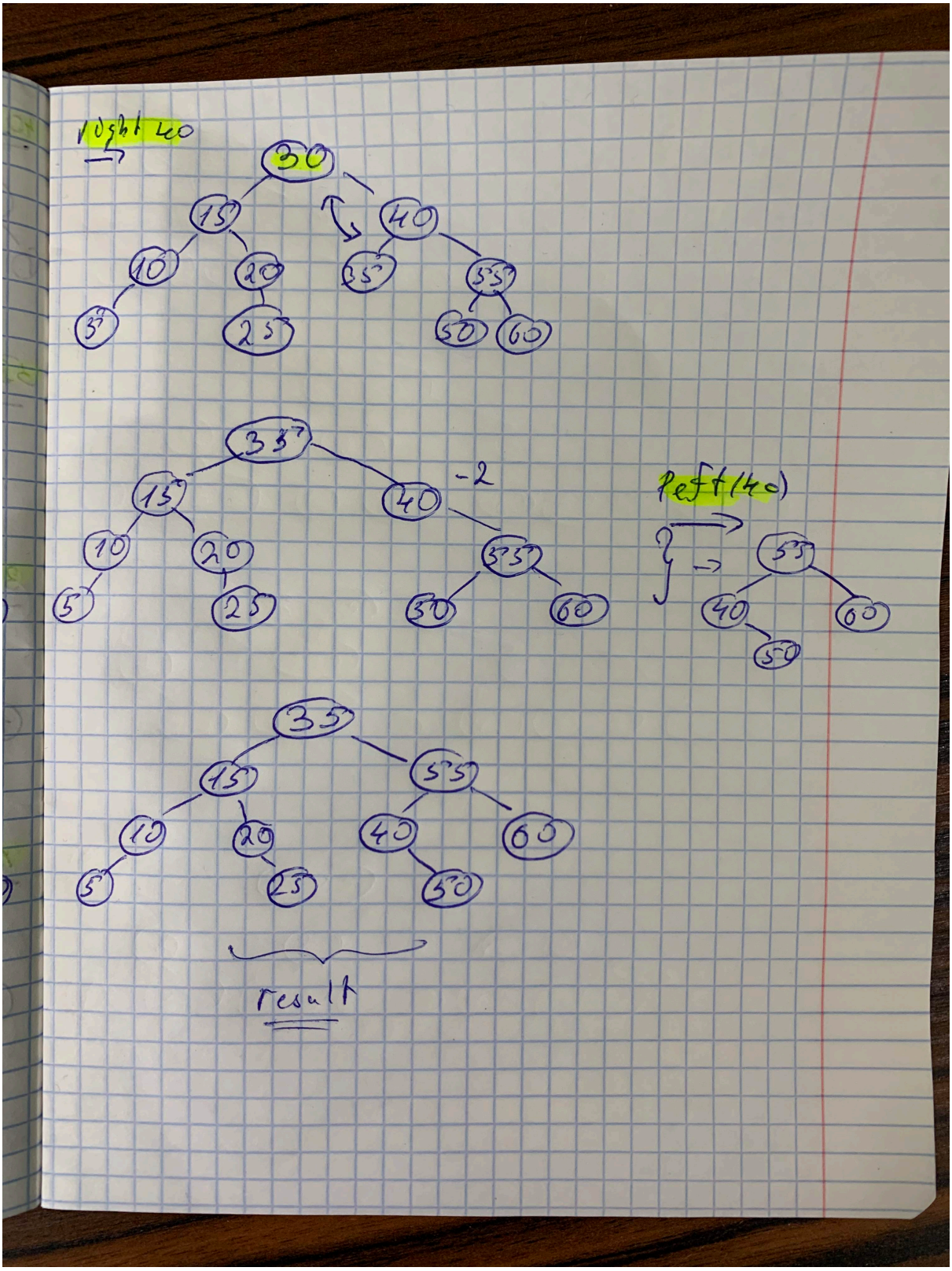


6. ***Explain in words or in pseudocode delete operation for AVL trees.***

Deletion consists of five operations: finding the node to be deleted, finding min/max in the subtree of the found node, swapping this two nodes (if the node being removed is a leaf, then we just remove it and check BF) , deletion of the required node and balancing the tree (go up to the root checking BF).

7. ***Give the trees obtained by deleting 45 and then 30 in the tree below.***





2.4 Binary Heap

Using two binary heaps describe a way to efficiently keep track of median values. That is, describe a data structure that is based on two heaps and provides the following methods:

1. **insert(k) — insert value;**
2. **remove_median() — remove median value and return it;**
3. **size() — return the size of the data structure (number of stored values);**
4. **isEmpty() — check if the structure is empty (no values).**

Write down pseudo code for these methods, using the interface of the binary heap and provide worst-case time complexity for each method.

We will use two binary heaps: one min heap, where we will store the elements greater or equal to the median and one max heap, where we will store the elements less or equal to the median.

When we need to get the value of the median, we will compare the sizes of two binary heaps. Three situations are possible:
1) Min heap contains one more element than max heap, then the median is on the top of min heap.
2) Max heap contains one more element than min heap, then the median is on the top of max heap.
3) Min heap contains the same number of elements as the max heap, then the median is on the top of min and max heap (return any).

When we insert elements, it may happen that the sizes of the two heaps differ by more than one. Then we need to extract the value from the min/max heap, in which there are more values and insert it into another heap.

Since our structure is initially empty, we will take the first input element as the median. And based on this element, we will perform the rules described above.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>

using namespace std;

class MedianFinder {
    priority_queue <int> maxHeap; // Max Heap
    priority_queue<int, vector<int>, greater<int>> minHeap;
public:
    /*
     * When we insert elements, it may happen that the sizes of the two heaps
differ by
     * more than one. Then we need to extract the value from the min/max heap,
in which
     * there are more values and insert it into another heap.
     *
     * Thus, time complexity of insert in MedianFinder equal to time complexity
     * of insert in min/max heap.
     *
     * Time complexity: O(log n) worst case
     */
    void insert(int k) {
        if (maxHeap.empty()) {
            maxHeap.push(k);
        } else if (minHeap.empty()) {
            if (k < maxHeap.top()) {
```

```
                int value = maxHeap.top();
                maxHeap.pop();

                maxHeap.push(k);
                minHeap.push(value);
            } else
                minHeap.push(k);
        }

        else {
            if (k < maxHeap.top())
                maxHeap.push(k);
            else if(k > minHeap.top()) {
                int value = minHeap.top();
                minHeap.pop();

                minHeap.push(k);
                maxHeap.push(value);
            } else
                maxHeap.push(k);

            while (maxHeap.size() > minHeap.size() + 1) {
                int value = maxHeap.top();
                maxHeap.pop();
                minHeap.push(value);
            }
        }
    }

    /*
     * 1) Min heap contains one more element  than max heap, then the median is
on the top of min heap.
     * 2) Max heap contains one more element  than min heap, then the median is
on the top of max heap.
     * 3) Min heap contains the same number of elements as the max heap, then
the median is on the top
     * of min and max heap (return any).
     *
     * Also we should delete top() element of the heap.
     *
     * Don't need to balance anything, because remove_median
     * decreases the size of the larger heap by one. So, we cannot have a
     * situation when after deletion the difference of heap sizes is more than
one.
     *
     * Thus, time complexity of remove_median equal to time complexity
     * of delete_min / delete_max in min / max heap.
     *
     * Time complexity: O(log n) worst case
```

```
     */
    double remove_median() {
        if (maxHeap.size() == minHeap.size()) {
            int median = maxHeap.top();
            maxHeap.pop();

            return median;
        } else if (maxHeap.size() > minHeap.size()) {
            int median = maxHeap.top();
            maxHeap.pop();
            return median;
        } else {
            int median = minHeap.top();
            minHeap.pop();
            return median;
        }
    }

    /*
     * The number of stored values is the number of
     * the number of values in maxHeap plus the
     * number of values in minHeap.
     *
     * Time complexity: O(1) worst case
     */
    int size() {
        return maxHeap.size() + minHeap.size();
    }

    /*
     * If method size() return 0, then
     * the data structure is empty and
     * vice versa.
     *
     * Time complexity: O(1) worst case
     */
    bool isEmpty() {
        return size() == 0;
    }
};

int main() {
  // Example
    MedianFinder m;
    m.insert(254);
    m.insert(2);
    m.insert(5);
    m.insert(100);
    m.insert(1);
```

```
    /*
     * 1 2 5 100 254 -> Median: 5
     * 1 2 100 254 -> Median: 2
     * 1 100 254 -> Median: 100
     * 1 254 -> Median: 1
     * 254 -> Median: 254
     */

    cout << m.remove_median() << endl;
    cout << m.remove_median() << endl;
    cout << m.remove_median() << endl;
    cout << m.remove_median() << endl;
    cout << m.remove_median() << endl;

    return 0;
}
```