# 2.1

## 2.1.1

To solve this problem, we need to find the closest shopping center with the best revenue, which placed before Xi. For each place we try to find the best revenue.

**Pseudocode:**

```
int FindBest(i)
        if i == -1
                return 0
        j = 0
        while (j < i)
                if x[i] – x[j] >= d
                        break
                j++
        return max(FindBest(i-1), r[i] + FindBest(j))
```

## 2.1.2

Recurrence formula for FindBest(N):

$T(n) = 2T(n-1) + n, \qquad n > 0$

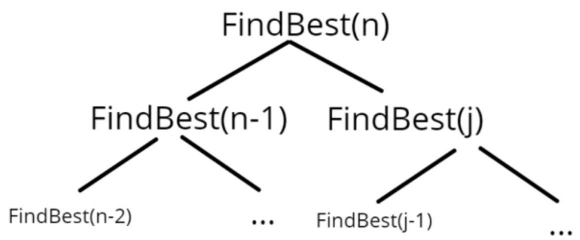
$T(1) = 1, \qquad n = 0$

$$T(n) = 2T(n-1) + n = \sum_{i=0}^{n-1} 2^i * (n-i) = n(2^0 + 2^1 + \cdots + 2^{n-1})$$

$n(2^0 + 2^1 + \cdots + 2^{n-1}) < 2^n n$

$T(n) = \theta(2^n n)$

## 2.1.3




(n-k) and (i-p) can be equal (k and p some constants). Therefore FindBest(n-k) and FindBest(i-p) can be overlap.

## 2.1.4

We can use top-down approach. Pseudocode:

| | |
|---|---|
| max_rev[N+1]<br>last [N+1]<br>max_rev[1] = r[1]<br>last[1] = -1 | $C$ |
| for i = 1 to N+1 | $N+2$ |
| best = 0<br>best_l = -1 | $N+1$ |
| for j = 1 to i -1 | $\sum_{i=1}^{N+1} i$ |
| if x[i] – x[j] >= d<br>if best < max_rev[j]<br>best = max_rev[j]<br>best_l = j | $\sum_{i=0}^{N} i$ |
| if i <= N<br>max_rev[i] = best + r[i]<br>else<br>max_rev[i] = best<br>last[i] = best_l | $N+1$ |
| ans = max_rev[N] //maximum revenue<br>locations = []<br>cur = N + 1 | $C$ |
| while last[cur] != -1 | $N+1$ |
| cur = last[cur]<br>locations.push_back(x[cur]) | $N$ |

# 2.1.5

We can see amount of operations in the table. Therefore:

$$T(n) = c + bN + \frac{(N+1)(N+2)}{2} + \frac{N(N+1)}{2} = c + bN + aN^2 = O(N^2)$$

# 2.2

## 2.2.1

In insertion sort we start from the second element and go through all elements(j – index of element). We get previous element and this index(I = index) (*all elements before j are sorted). While this element lower than previous one, we make index I lower (I = I -1) and next element for I replace by this element (We insert our j-th element on correct position in j-1 sorted elements). When our cycle stop – we replace element with index I+1 by taken element (that was with index j at the beginning) – ***You can see pseudocode in 2.2.2***

## 2.2.2

| **Insertion Sort (Pseudocode)** | **Cost** | **Times (Worst Case)** | **Times (Best Case)** |
|---|---|---|---|
| for j = 2 to A.length | C1 | $n$ | $n$ |
| key = A[j] | C2 | $n-1$ | $n-1$ |
| i = j - 1 | C3 | $n-1$ | $n-1$ |
| while i > 0 and A[i] > key | C4 | $\sum_{j=2}^{n} t_j$ | $n-2$ |
| A[i + 1] = A[i] | C5 | $\sum_{j=2}^{n}(t_j-1)$ | 0 |
| i = i – 1 | C6 | $\sum_{j=2}^{n}(t_j-1)$ | 0 |
| A[i + 1] = key | C7 | $n-1$ | $n-1$ |

**Worst**-case time complexity:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1) = \left(\frac{c_4+c_5+c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + c_7 + \frac{c_4-c_5-c_6}{2}\right)n - (c_2 + c_3 + c_4 + c_7)$$

= $\boldsymbol{\theta(n^2)}$

**Best**-case time complexity:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-2) + c_7(n-1) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + 2c_4 + c_7)$$

= $\boldsymbol{\theta(n)}$

# 2.2.3

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

Worst-case of Insertion sort for *k*-sorted array = $\boldsymbol{\theta(nk)}$. In worst case inner cycle runs k times (Elements placed in k distance from correct places). To move every element to its correct place, at most k elements need to be moved.

| **Insertion Sort (Pseudocode)** | **Cost** | **Times (Worst Case)** |
|---|---|---|
| for j = 2 to A.length | C1 | $n$ |
| key = A[j] | C2 | $n-1$ |
| i = j - 1 | C3 | $n-1$ |
| while i > 0 and A[i] > key | C4 | $\sum_{j=2}^{n} k$ |
| A[i + 1] = A[i] | C5 | $\sum_{j=2}^{n} k-1$ |
| i = i – 1 | C6 | $\sum_{j=2}^{n} k-1$ |
| A[i + 1] = key | C7 | $n-1$ |

$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n(k-1)) + c_5((n-1)(k-1)) + c_6((n-1)(k-1)) + c_7(n-1) = (c_4 + c_5 + c_6)nk + (c_1 + c_2 + c_3 - c_4 - c_5 - c_6 + c_7)n - (c_5 + c_6)k - (c_2 + c_3 - c_5 - c_6 + c_7)$

= $\boldsymbol{\theta(nk)}$ $\boldsymbol{* \, n \geq k > 0}$

Insertion sort works faster (in worst-case) for *k*-sorted array: $\boldsymbol{\theta(nk)} \leq \boldsymbol{\theta(n^2)}$

# 2.2.4

Here you can see pseudocode with structure which represent given data, function that compare two objects (by code, date_start, date_end, sponsor and description) and function that sort array with objects

```
struct object
  string code
  string date_start
  string date_end
  string sponsor
  string description

bool a_more_than_b(object a, object b)
  if a.code.size > b.code.size
    return true
  else if a.code.size < b.code.size
    return false
  else
    for i = 1 to a.code.size
      if a.code[i] < b.code[i]
        return false
      if a.code[i] > b.code[i]
        return true

  for i = 7 to 10
    if a.date_start[i] < b.date_start[i]
      return false
    if a.date_start[i] > b.date_start[i]
      return true

  for i = 4 to 5
    if a.date_start[i] < b.date_start[i]
      return false
    if a.date_start[i] > b.date_start[i]
      return true

  for i = 1 to 2
    if a.date_start[i] < b.date_start[i]
      return false
    if a.date_start[i] > b.date_start[i]
      return true

  for i = 7 to 10
    if a.date_end[i] < b.date_end[i]
      return false
    if (a.date_end[i] > b.date_end[i]
      return true
  for i = 4 to 5
    if a.date_end[i] < b.date_end[i]
      return false
    if a.date_end[i] > b.date_end[i]
      return true

  for i = 1 to 2
    if a.date_end[i] < b.date_end[i]
      return false
    if a.date_end[i] > b.date_end[i]
      return true

  if a.sponsor.size > b.sponsor.size
    return true
  else if a.sponsor.size < b.sponsor.size
    return false
  else
    for i = 1 to a.sponsor.size
      if a.sponsor[i] < b.sponsor[i]
        return false
      if a.sponsor[i] > b.sponsor[i]
        return true

  if a.description.size > b.description.size
    return true
  else if a.description.size < b.description.size
    return false
  else
    for i = 1 to a.description.size
      if a.description[i] < b.description[i]
        return false
      if a.description[i] > b.description[i]
        return true

  return false

void sort(object A[])
  for j = 2 to A.length
    key = A[j];
    i = j - 1;
    while i > 0 and a_more_than_b(A[i], key)
      A[i + 1] = A[i]
      i = i -1
    A[i + 1] = key
```

# 2.2.5

| Recursive | Iterative |
|---|---|
| bool belongs(A, from, to, x)<br>    if (to >= from)<br>        mid = from + (to - from)/2<br>        if (A[mid] == x)<br>            return true<br>        if (A[mid] > x)<br>            return belongs(A, from, mid – 1, x)<br>        else<br>            return belongs(A, mid + 1, to, x)<br>    return false | bool belongs(A, from, to, x)<br>    while (to >= from)<br>        mid = from + (to - from)/2<br>        if (A[mid] == x)<br>            return true<br>        if (A[mid] > x)<br>            to = mid – 1<br>        else<br>            from =  mid + 1<br>    return false |

# 2.2.6

```
int search(A, from, to, x)
    if (to >= from)
        mid = from + (to - from)/2
        if (A[mid] == x)
            return mid
        if (A[mid] > x)
            return search(A, from, mid – 1, x)
        else
            return search(A, mid + 1, to, x)
    return null
```

# 2.2.7

$$T(n) = T \times \left(\frac{n}{2}\right) + 1$$

$$a = 1, b = 2, f(n) = 1$$

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$$

$$f(n) = n^{\log_b a} \rightarrow 2nd\ Case$$

$$T(n) = \theta\left(n^{\log_b a} \log n\right) = \theta(\log n)$$

# 2.2.8

Search (Binary search) helps to reduce the number of comparisons. We can use Search to find location to insert the selected element at each iteration. In normal insertion sort, it takes O(n) comparisons (at nth iteration) in the worst case. We can reduce it to O(log n) by using binary search.

**Pseudocode:**

```
int search(A, from, to, x)
        if (to >= from)
                mid = from + (to - from)/2
                if (A[mid] == x)
                        return mid
                if (A[mid] > x)
                        return search(A, from, mid – 1, x)
                else
                        return search(A, mid + 1, to, x)
        return null

void insertionSort(A)
        for j = 2 to A.length
        key = A[j]
        i = j – 1
        loc = search(A, 0, i, key)
        if (loc == null)
                while i > 0 and A[i] > key
                        A[i+1] = A[i]
                        i = i – 1
                A[i+1] = key

        else
                while i >= loc
                        A[i+1] = A[i]
                        i = i – 1
                A[i+1] = key
```

**Time Complexity**: The algorithm as a whole still has a running worst-case running time of $\boldsymbol{\theta(n^2)}$ because of the series of swaps required for each insertion.

# 2.2.9

Bubble sort is easy to parallelize. We need to sort subarrays in different threads and after that we should merge with 2 pointers. Execution time = $\frac{n^2}{2} + n$:

```
void bubble(A, from, to)
        for (j = from; j < to – 1; j++)
                for (i = from; i < to-1; i++)
                if A[i] > A[i+1]
                        swap(A[i], A[i+1])
void sort(A)
        int ans[A.size]
        i = 0
        j = A.size / 2
        thread1(bubble, A, 0, j)
        thread2(bubble, A, j, A.size)
        thread1.wait
```

```
        thread2.wait
        k = 0
        while i != A.size / 2 and j != A.size
                if A[i] < A[j]
                        ans[k++] = A[i++]
                else
                        ans[k++] = A[j++]
        while i != A.size / 2
                ans[k++] = A[i++]
        while j != A.size
                ans[k++] = A[j++]
        A = ans
```

# 2.3

## 2.3.1

***AVL tree's properties:***

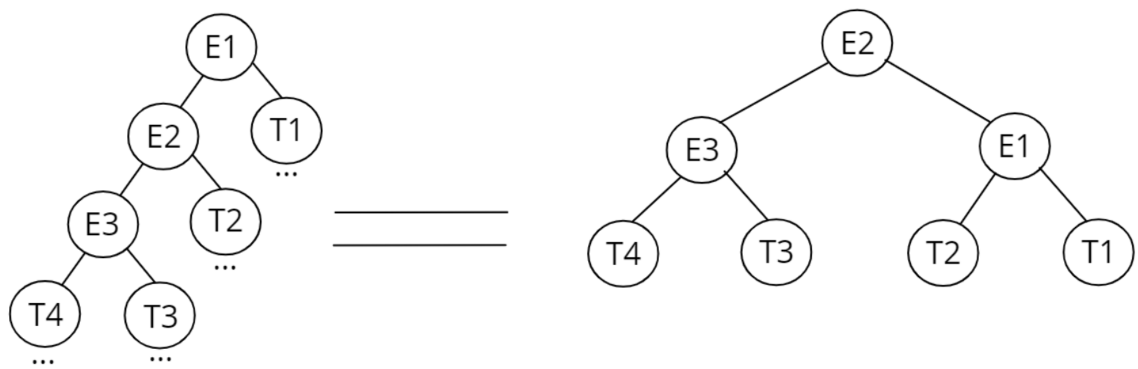
1. Each tree has a root node
2. Each node has 0, 1 or 2 child nodes
3. For each node maximum count of child = 2
4. | height(left subtree) | - | height(right subtree) | = {-1, 0, 1}

## 2.3.2

1. Perform standard Binary Search tree insert for new element(we use search algorithm until no child, and put at that point) (N)
2. Count all differences between the height of left subtree and the height of right subtree, starting with the element N
    a. Find the first unbalanced node
    b. Let E1 be the first unbalanced node. E2 be the child of this node that is on the path from N to E1. E3 be the grandchild of E1 that is on the path from N to E1
    c. Rebalance tree(4 cases):
        i. Left Left Rotation (E2 is left child of E1 and E3 is left child of E2)
        ii. Left Right Rotation (E2 is left child of E1 and E3 is right child of E2)
        iii. Right Left Rotation (E2 is right child of E1 and E3 is left child of E2)
        iv. Right Right Rotation (E2 is right child of E1 and E3 is right child of E2)

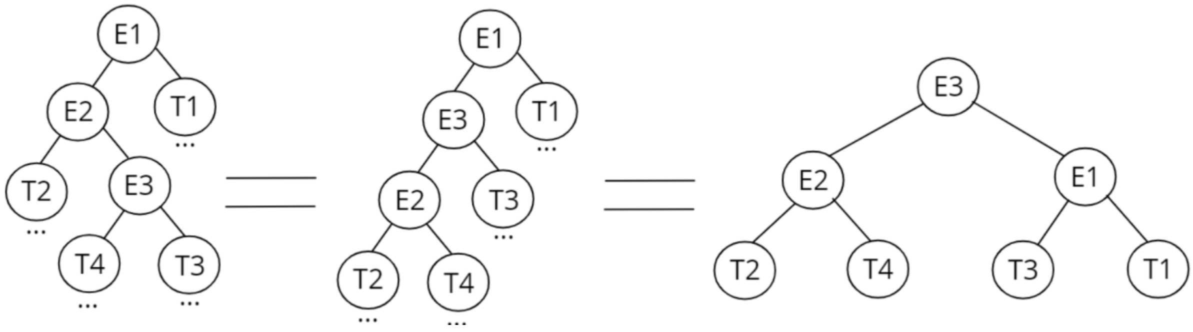
Left Left Rotation

T1, T2 … - subtrees



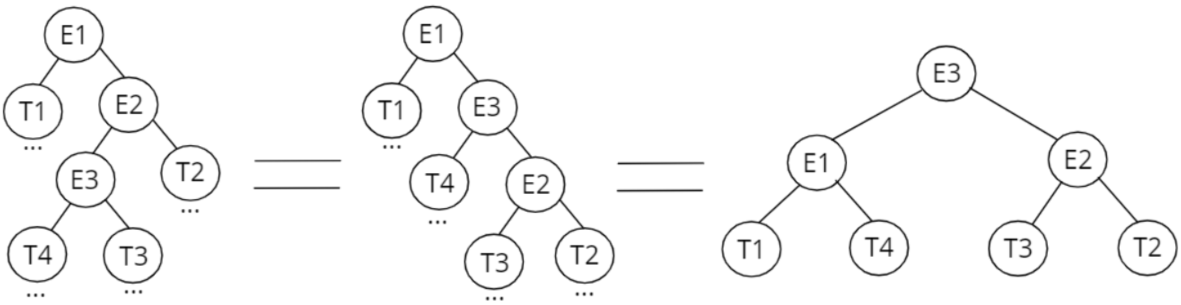
Left Right Rotation

T1, T2 ... - subtrees



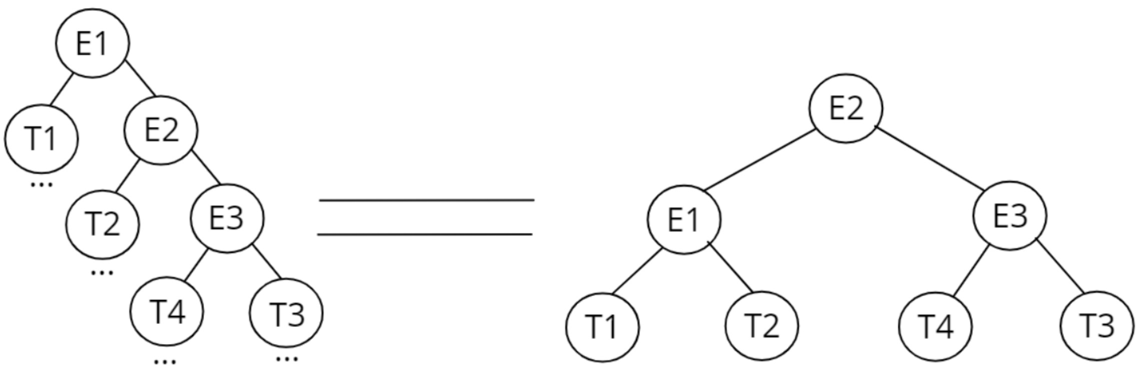

Right Left Rotation

T1, T2 ... - subtrees

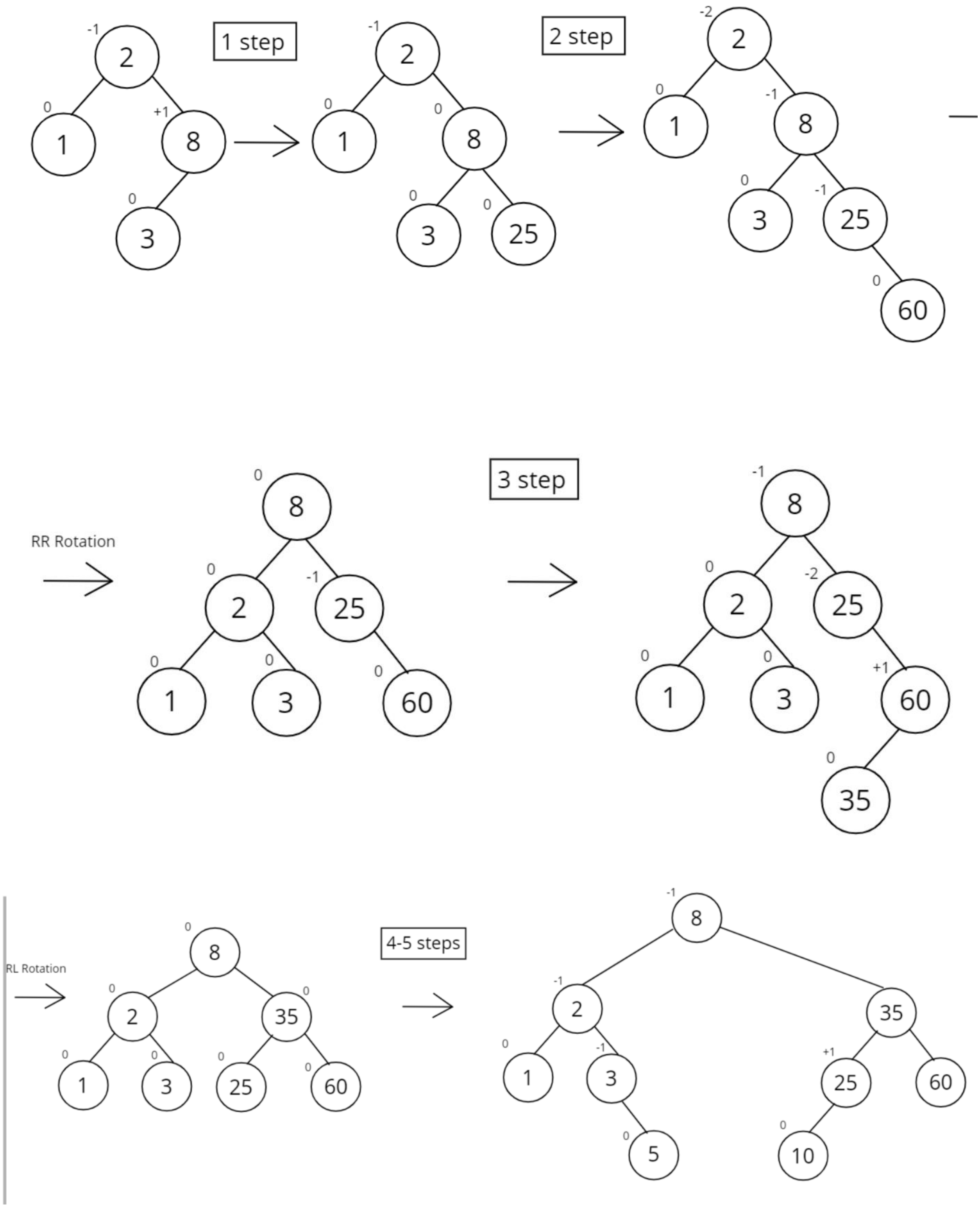



## Right Right Rotation

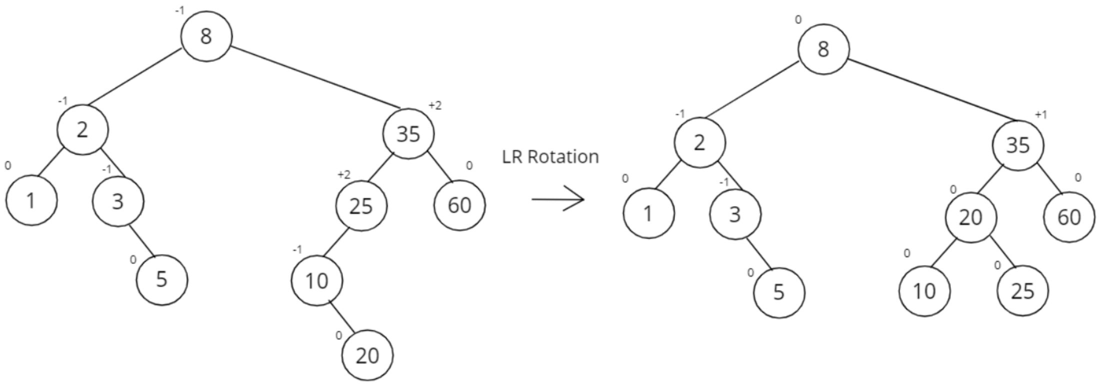
T1, T2 ... - subtrees



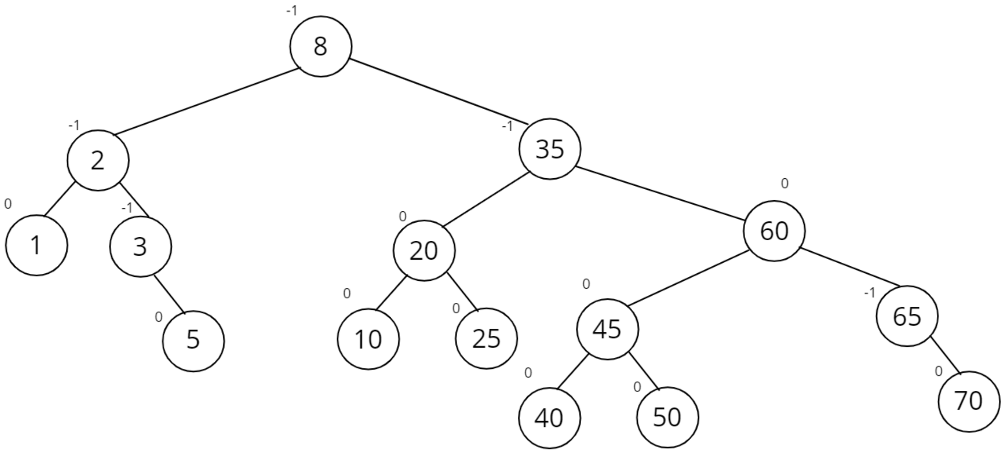
## 2.3.3



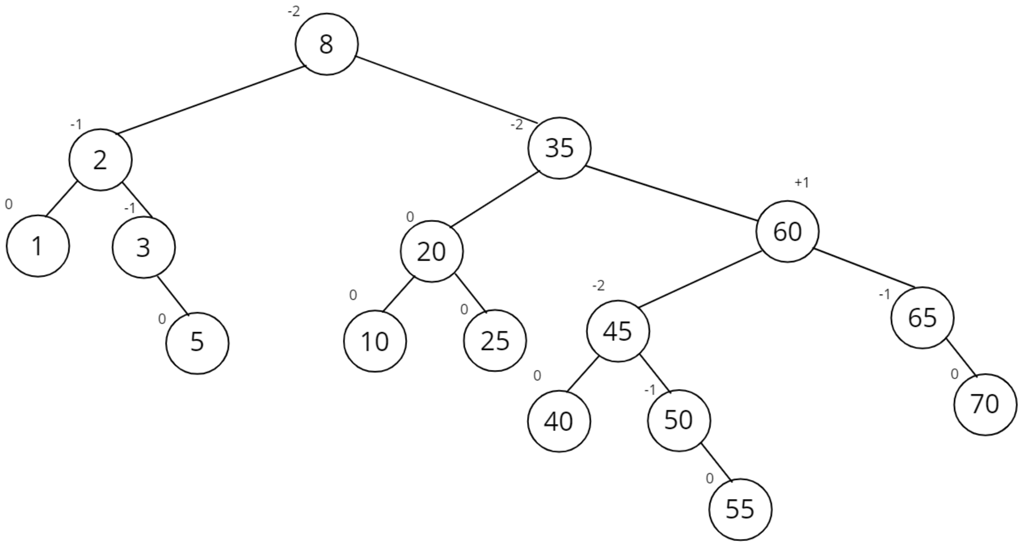
6 step



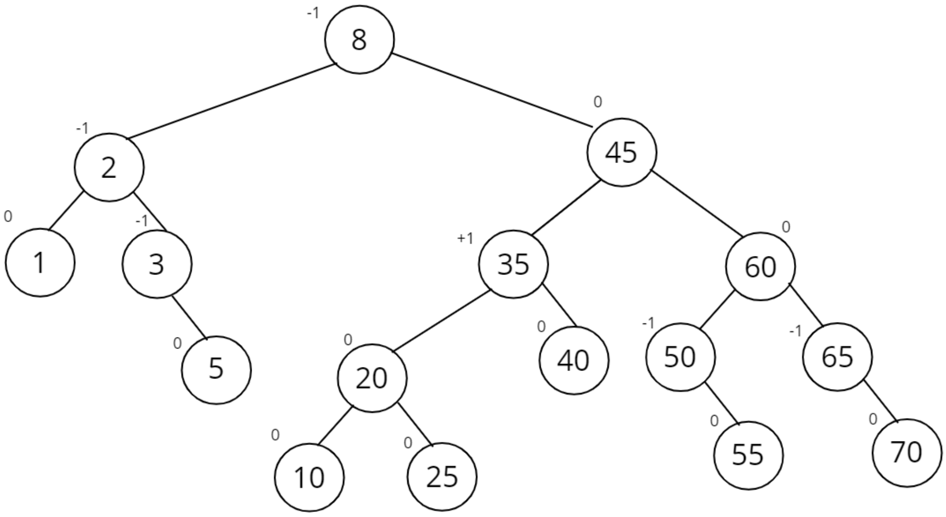

7-11 steps



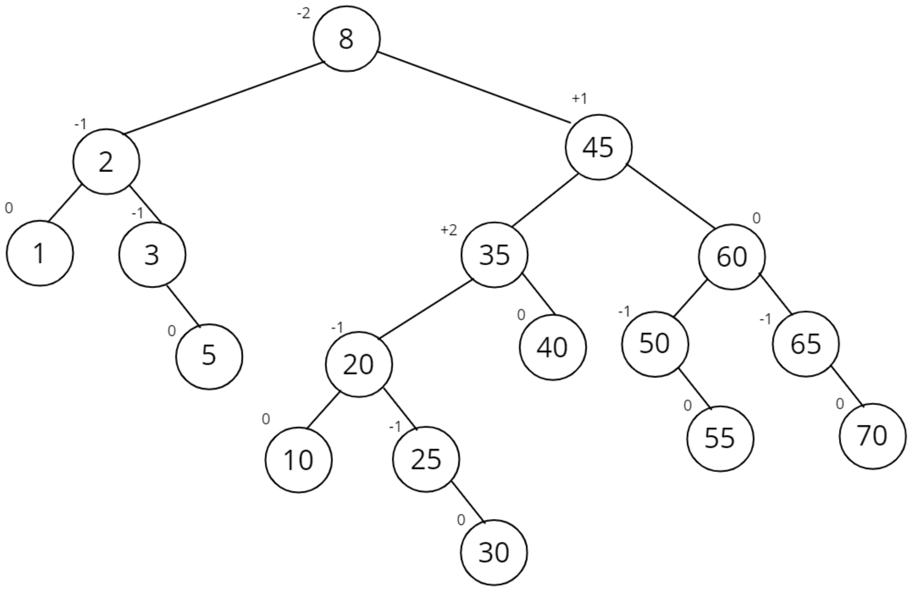
12 step



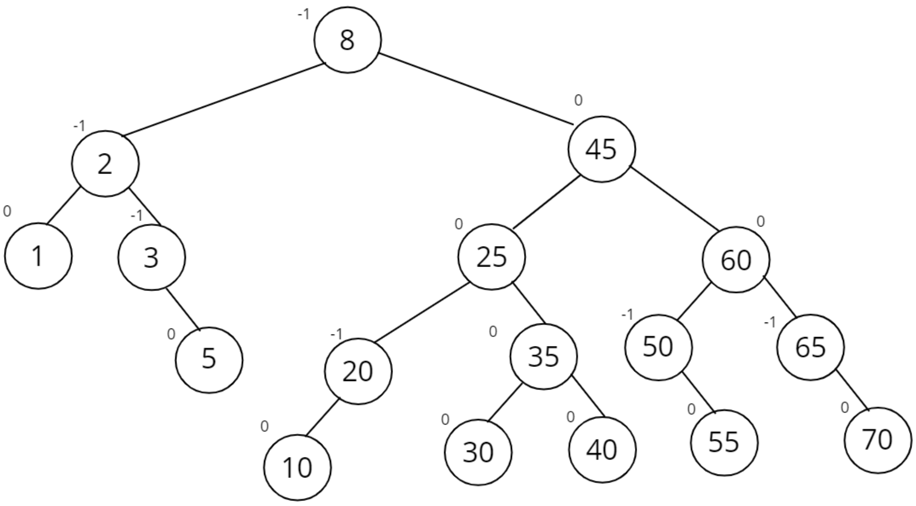
RL Rotation



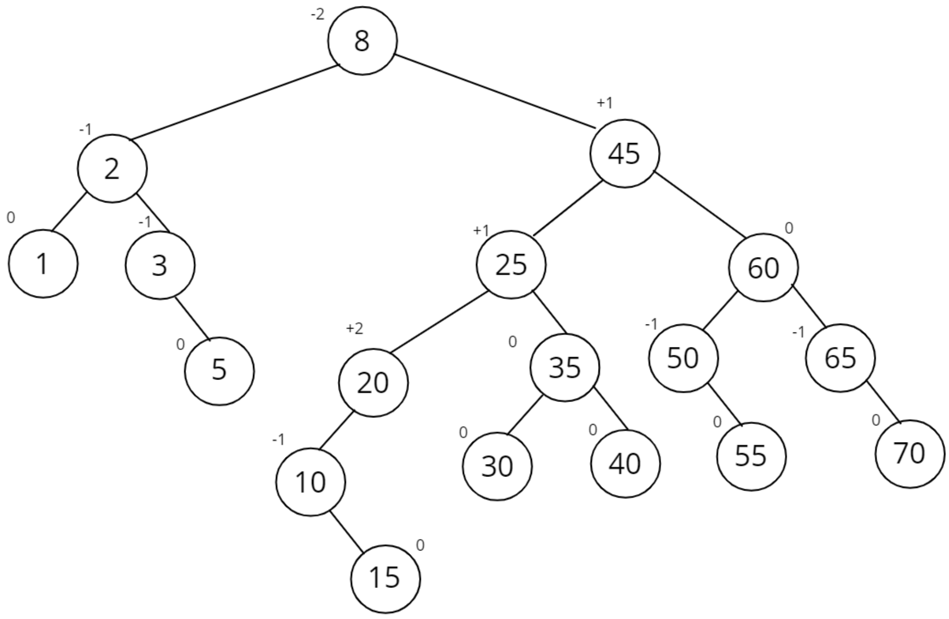

13 step




LR Rotation

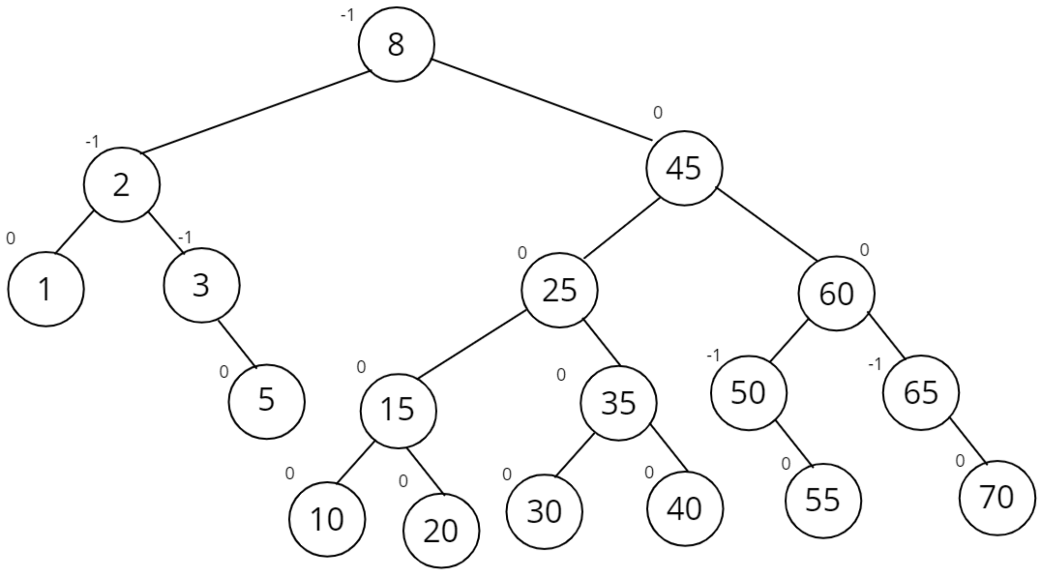


14 step



LR Rotation

Answer



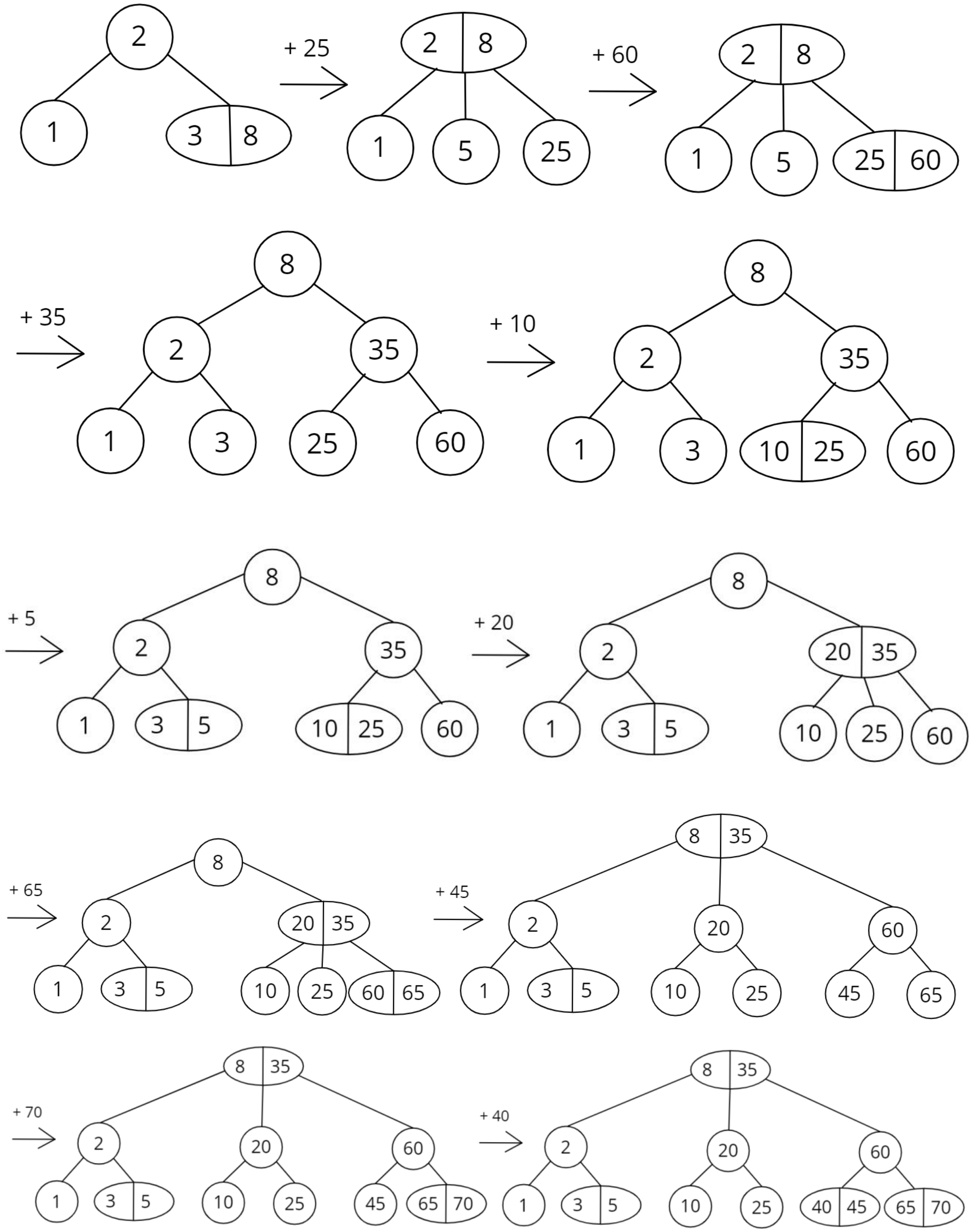
# 2.3.4

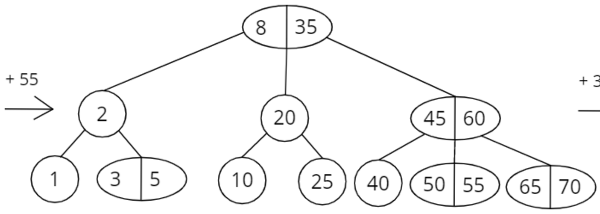
In-Order traversal of the tree:

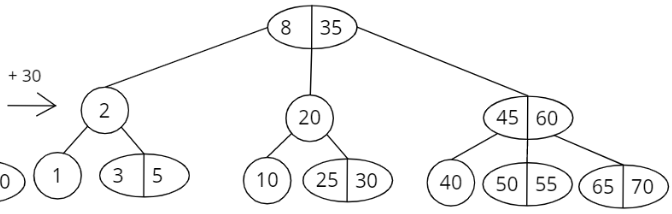
1, 2, 3, 5, 8, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70

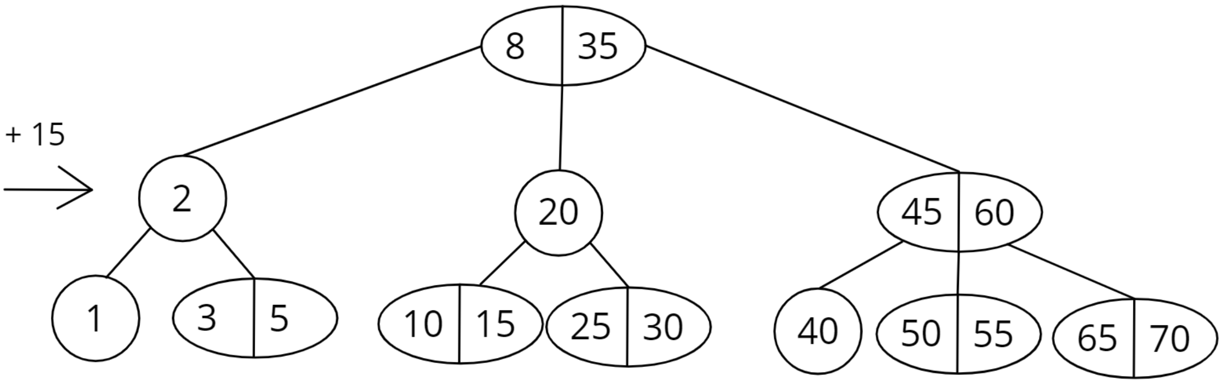
Numbers ordered in increasing order. But if we have equivalent nodes in our tree, order will be not-decreasing. Therefore it is not always the case.

## 2.3.5





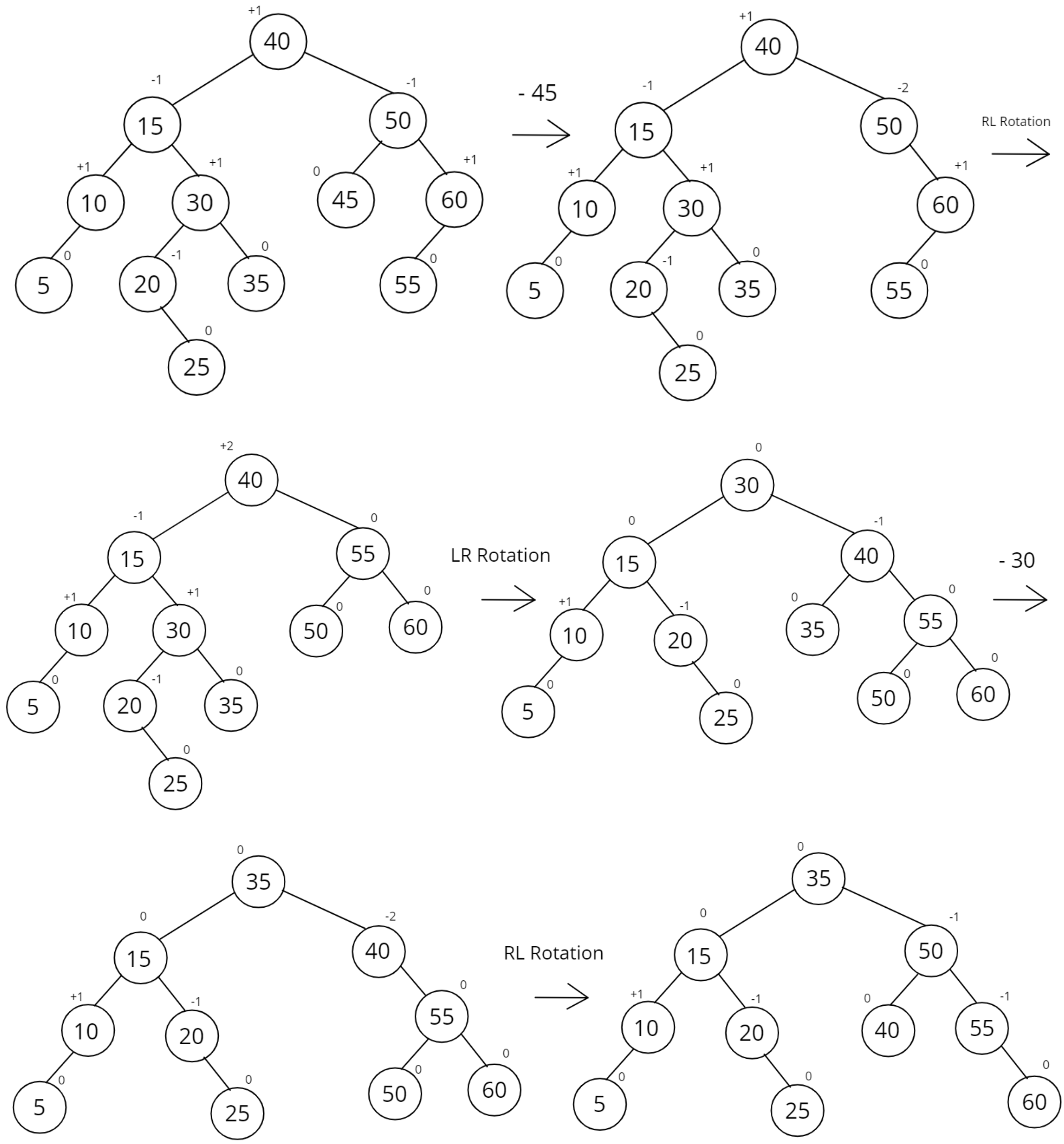




# 2.3.6

Delete operation for AVL trees:

1. Identify successor for given node, replace the node with successor, delete our new successor (given node – element N)
2. Count all differences between the height of left subtree and the height of right subtree, starting with the parent of successor (or with parent of N)
    a. Find the first unbalanced node
    b. Let E1 be the first unbalanced node. E2 be the larger height child of E1. E3 be the larger height child of E2.
    c. Rebalance tree(4 cases):
        i. Left Left Rotation (E2 is left child of E1 and E3 is left child of E2)
        ii. Left Right Rotation (E2 is left child of E1 and E3 is right child of E2)
        iii. Right Left Rotation (E2 is right child of E1 and E3 is left child of E2)
        iv. Right Right Rotation (E2 is right child of E1 and E3 is right child of E2)

## 2.3.7



# 2.4

```
class midElement
private:
   minHeap min;
   maxHeap max;

public:
   midElement()

   void insert(int val)
      min.insert(val);
      if (min.size() - 1 > max.size())
         max.insert(min.pop());
      if (max.size() and min.size() and min.peek() < max.peek())
         int t_Min = max.pop();
         int t_Max = min.pop();
         max.insert(t_Max);
         min.insert(t_Min);

   int remove_median()
      int res = min.pop();
      if (min.size() < max.size())
         min.insert(max.pop());
      return res;

   int size()
      return min.size() + max.size();

   bool isEmpty()
      return !size();
```

| **Wors-Case time complexity** | |
|---|---|
| insert() | $\theta(\log n)$ |
| remove_median() | $\theta(\log n)$ |
| size() | $\theta(1)$ |
| isEmpty() | $\theta(1)$ |