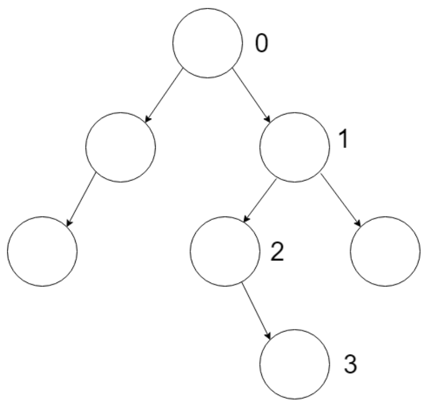
# Assignment 9

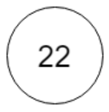
## Exercise 1: AVL-Tree

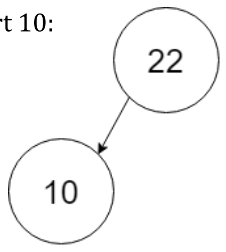
1. What is the maximum height for an AVL-Tree with 7 nodes?
   The maximum height can be computed with the following formula: n(h) = 1+ n(h-1) + n(h-2) which represents the minimum nodes at each level. Furthermore, the height difference of external nodes can only be one.
   n(0) = 1
   n(1) = n(1-1) + n(1-2) + 1 = 1 + 0 + 1 = 2
   n(2) = n(2-1) + n(2-2) + 1= 2 + 1 + 1 = 4
   n(3) = n(3-1) + n(3-2) + 1= 4 + 2 + 1 = **7** → the maximum height for an AVL-Tree with 7 nodes is 3

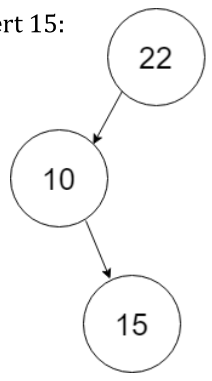
   Example

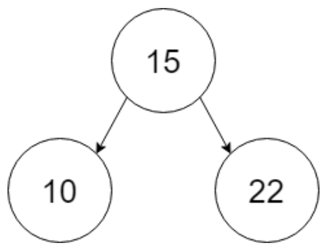


2. Build an AVL-Tree by inserting the following numbers starting with 22: [22,10,15,9,3,45,4,31,12,5]

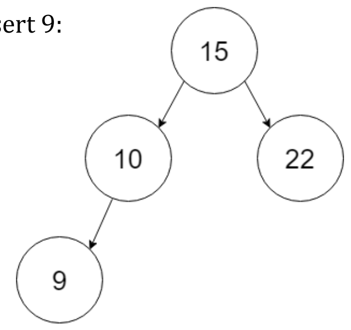
Insert 22:


Insert 10:


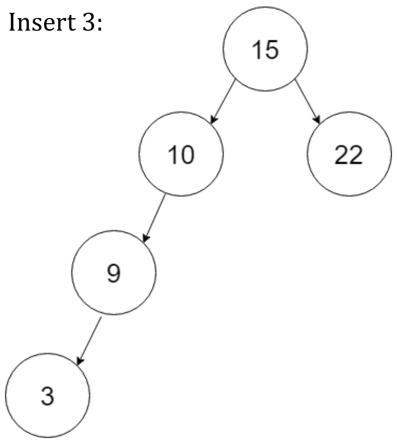
Insert 15:


Restructure:


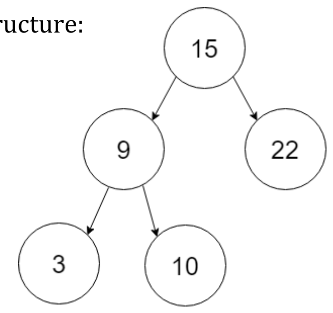
Insert 9:



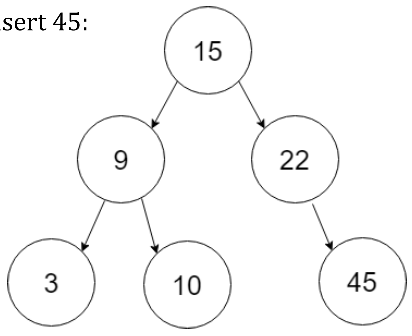
Insert 3:



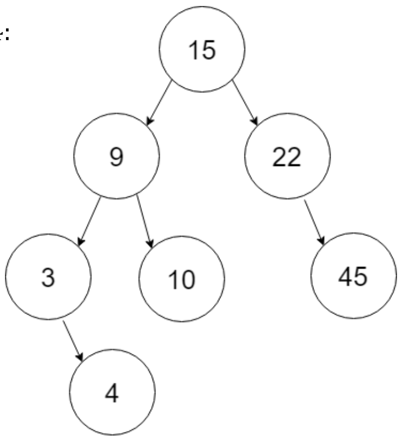
Restructure:



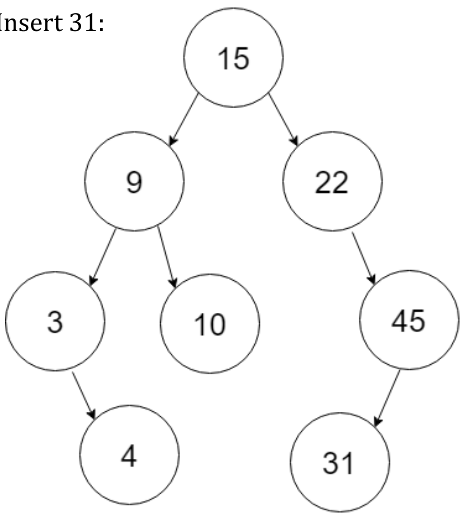
Insert 45:



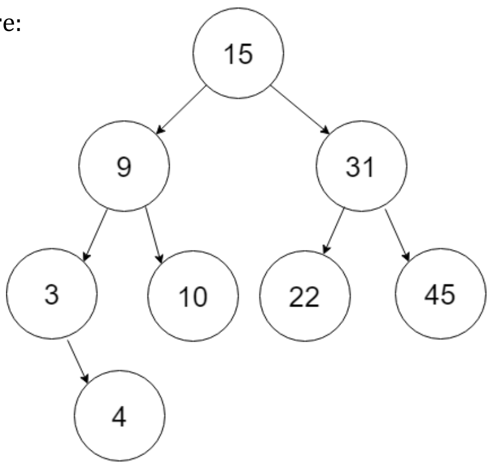
Insert 4:



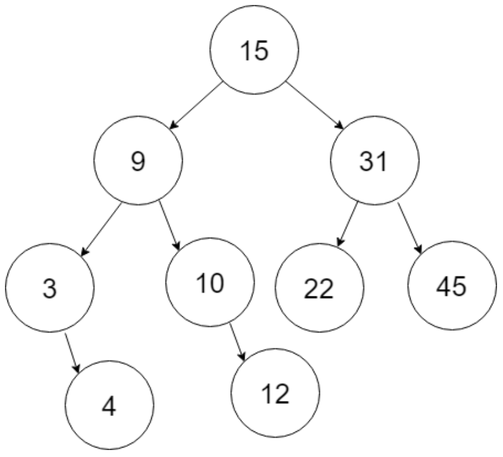
Insert 31:



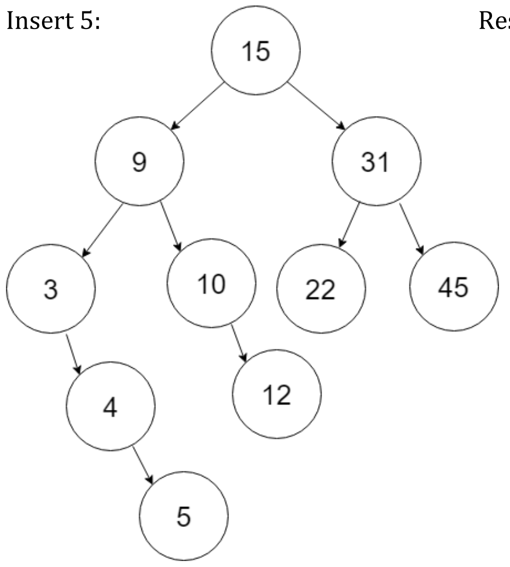
Restructure:



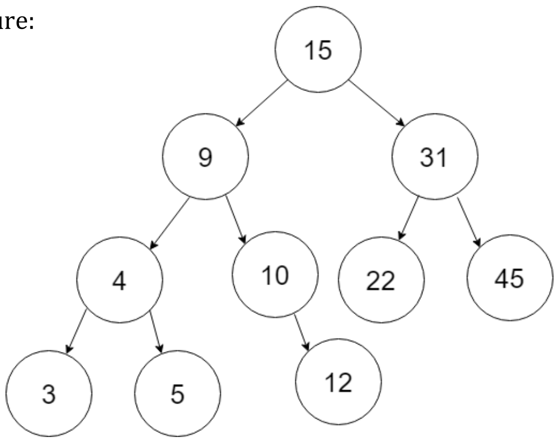
Insert 12:



Insert 5:



Restructure:



## Exercise 2: Greedy Algorithms

```
public static LinkedList<Integer> greedyAlgorithm(LinkedList<Task> tasks, int slots){
        LinkedList<Integer> timeSlot = new LinkedList<>();
        Task highestProfitTask = new Task();

        for(int i = 0; i < slots; i++){
            for(Task x : tasks){
                if(x.getDeadline() > i && x.getProfit() > highestProfitTask.getProfit()){
                    highestProfitTask = x;
                }
            }
            tasks.remove(highestProfitTask);
            timeSlot.add(highestProfitTask.getProfit());
            highestProfitTask.setProfit(0);
        }
        return timeSlot;
}
```

m … number of timeslots
n … number of tasks
The loop for the timeslots takes O(m).
The loop which iterates through the tasks takes at most O(n).
Removing, adding and setting takes O(1).
Therefore, the complexity is O(m · n).

This algorithm returns the optimal solution (maximized profit) because it searches after the task with the maximum profit every timeslot. It also checks if the deadline isn't over. Therefore, only tasks with the highest profit and a deadline in the future can be added to the output list.
The optimal output for this example is: [200, 150, 100]

The algorithm will not yield an optimal solution if the deadline of fifth job is set to 1, because at first the algorithm takes 200, then 100 and then there is no element with the needed deadline left. Therefore, the output is: [200, 100, 0] which is not the optimal solution.

## Exercise 3: Complexity of merge-sort

Proof that the recursion depth of merge-sort is O(log (n)) for a sequence of n comparable elements.
Merge-sort is a recursive algorithm which divides its input into two halves, calls itself for the halves and then merges the two halves together. So, if the input is divided into two halves every function call, then there are log n recursive calls. This can be easily shown with the master theorem:

$$T(n) = a \cdot T(\tfrac{n}{b}) + f(n)$$

where $a \geq 1 \, and \, b \geq 1$
a … number of subproblems
n/b … size of each subproblem
f(n) … time to create subproblems and combine their result

Therefore, we can substitute a by 2 because of the number of subproblems gets multiplied by 2 every level, n/b can be substituted by n/2 because the algorithm divides the number of elements into two parts every level.

$$T(n) = 2 \cdot T(\tfrac{n}{2}) + f(n)$$

This is the second case of the theorem because $f(n) \in \theta(n^{log_b a}) = f(n) \in \theta(n^1)$.
It follows that $T(n) \in \theta(n \, log(n))$ where log(n) is the recursion depth → $log(n) \in O(log(n))$.

## Exercise 4: Parallel Sorting

***Merge-Sort***

Merges-Sort is a divide and conquer algorithm, because the input is divided into two halves and then the algorithm calls itself for both subproblems until the level which has only one element per subproblem is reached. After that the sorted halves get merged together. For parallelization multiple threads that execute the subproblems could be used.

***Quick-Sort***

Is also a divide and conquer algorithm. It picks pivot element which divides the input into two halves. The numbers which are smaller than or equal to the pivot come to the left sides and the bigger ones come to the right. For this algorithm multiple threads which execute the subproblems can be used too.

***Bucket-Sort***

This algorithm puts the elements into buckets and then applies some sorting algorithm to sort the elements in each bucket, for example merge-sort or quick-sort. Afterwards the elements must be concatenated. Therefore, bucket sort may be good for parallelization.

***Heap-Sort***

Heap can’t be divided into multiple heaps because the whole heap is needed for sorting.

***Selection-Sort/Insertion-Sort***

Parallel sorting would be possible but not be very efficient for these algorithms, because the input can be divided into subproblems and those subproblems can be sorted independently but concatenating would take as long as the normal selection-sort/insertion-sort.
Another way is to use a pipeline where every process holds one number. When the process gets a new number, it keeps the smaller one and passes along the larger to another process.