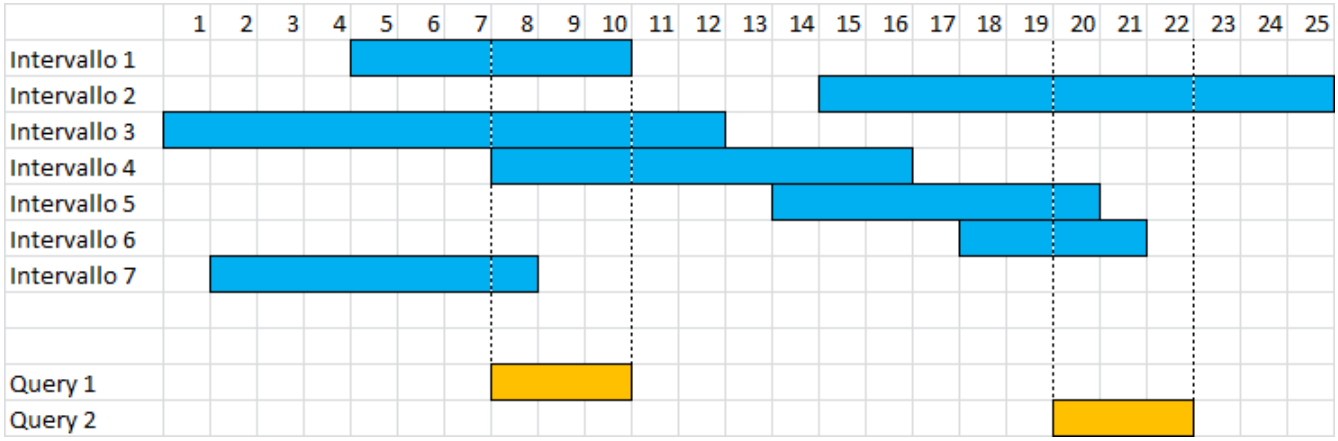
# Data Structures: Augmented Interval Tree to search for intervals overlapping

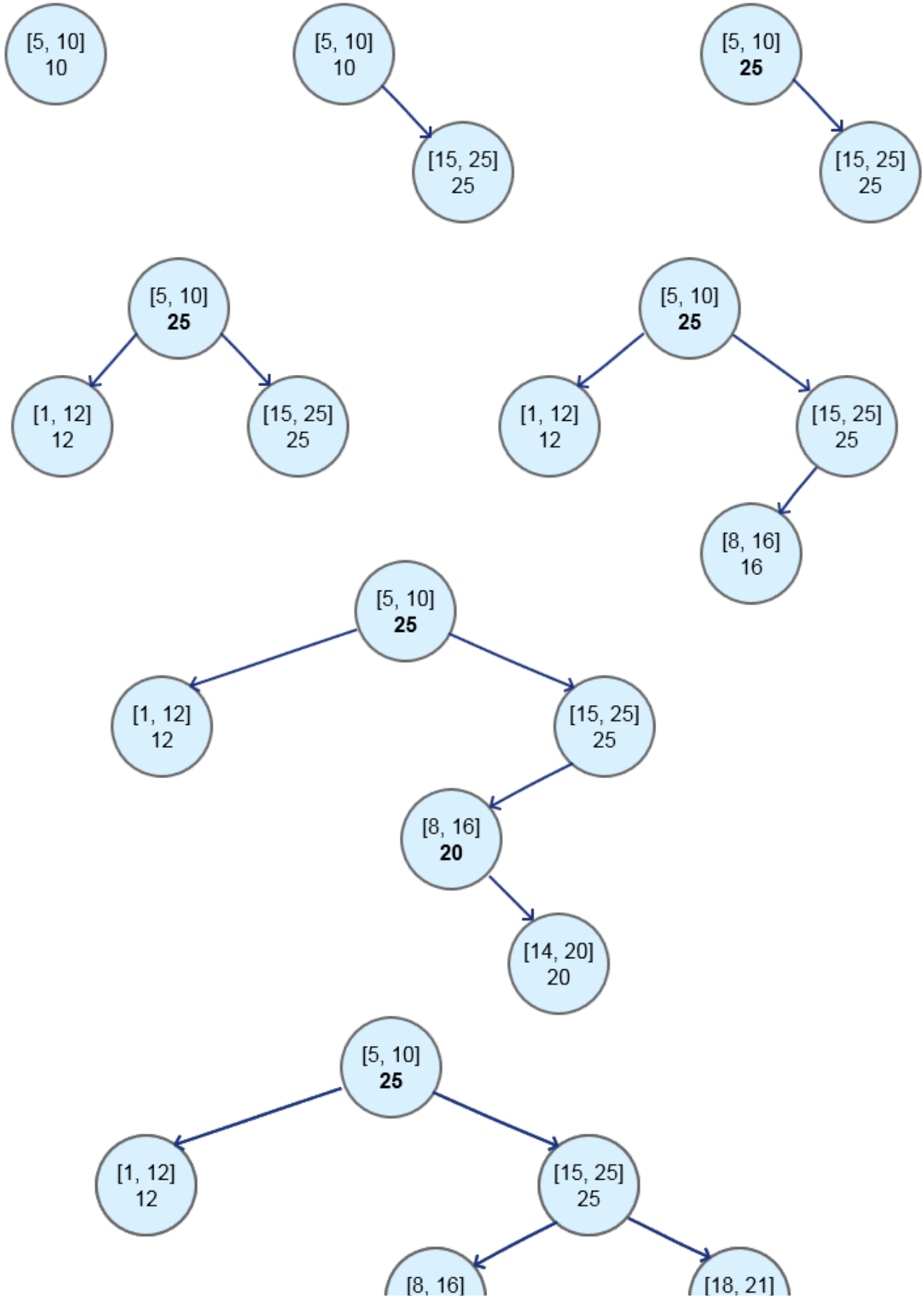


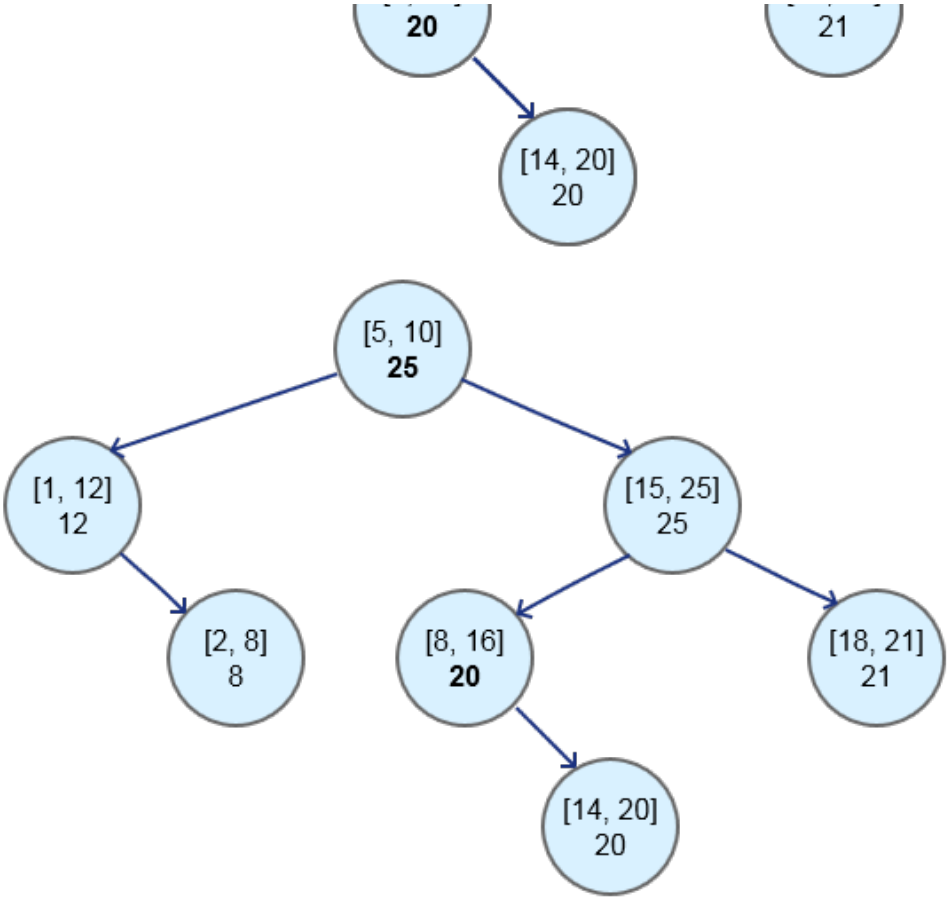
An **Interval Tree** is an ordered data structure whose nodes represent the intervals and are therefore characterized by a start value and an end value. A typical application example is when we have a number of available intervals and another set of query intervals, for which we want to verify the overlap with the given intervals.
The following picture shows an example where in blue are represented a series of intervals and in yellow are defined two query intervals for which we want to verify the overlap with the previous ones:
Picture 1:




The Interval Tree structure comes into play to provide a more efficient solution than the naive approach of applying a brutal force strategy and compare each query range with all the others and check if, according to the values of the relative bounds, there is an overlap (total or partial) between them.
In particular, to make more efficient this type of search we can use an **Augmented Interval Tree**, a structure in which **the information contained in each node is increased** by adding, in addition to the bounds of the range, also the information related to **the maximum value of the subtree of the node that we are analyzing**. This value must be calculated during the insertion phase of each node and it need to be updated, backward to the top, in case of deletion of a node.
Let's see an example in which we create the Augmented Interval Tree

related to the previously described nodes; each node contains information of its bounds, in the format [5,10] and, below, the information on the maximum value of its sub-tree.
Picture 2:






Let us analyze the insertion sequence: starting from the root node [5,0] is then added the second node whose bounds are the values 15 and 25. In this case we must immediately proceed with the update of the additional information about the maximum value of the subtree for the first node (third picture in the top right). The second node has, in fact, as the maximum extreme 25 and then this value must be reported in the first node (being the second node in its the subtree). Proceeding, nodes with the intervals [1,12] and [8,16] are inserted and both do not require updates to the maximum value of the other nodes. 16 in fact, is not greater than the maximum value of any of the nodes we went through (25 for both).

At this point we insert the node for the interval [14,20]. In this case, the node ends in the subtree of [8,16] and that node has exactly 16 as maximum value. Since the upper bound 20 is larger than 16, we have to update in the node [8,16] the information about the maximum value of the subtree, passing just from 16 to 20. Finally we insert nodes for the last two intervals, [18,21] and [2,8] which however do not require the update of anyone of the nodes that pass through, because their upper

values do not exceed the maximum value of the subtree in which they end up.
Now let’s start to implement the class that represents the nodes, which will then contain the bounds of the ranges, the maximum upper limit of the subtree and the references to the child nodes, the right and the left.
In this class, that we call **Interval**, we implement the **Comparable** interface and then redefine the **compareTo()** method, as we have to establish a criterion to compare the nodes and then determine the insertion position in the tree. In addition we also provide a override of the toString() method that provide us a readable representation of our nodes, showing the extremes of the ranges and the additional information of the maximum value of the node subtree.
In the light of the above, our Interval class will be as follows:

```
class Interval implements Comparable<Interval> {

   private long     start;
   private long     end;
   private long     max;
   private Interval left;
   private Interval right;

   public Interval(long start, long end) {

      this.start = start;
      this.end = end;
      this.max = end;
   }

   @Override
   public String toString() {
      return "[" + this.getStart() + ", " + this.getEnd() + ", "
            + this.getMax() + "]";
   }

   @Override
   public int compareTo(Interval i) {
      if (this.start < i.start) {
         return -1;
      }
      else if (this.start == i.start) {
         return this.end <= i.end ? -1 : 1;
      }
      else {
         return 1;
      }
   }

 // GETTERS E SETTERS OMITTED FOR SIMPLICITY

}
```

Once described the single node structure, we can move on to define our Augmented Interval Tree through a class that:

- contains an Interval reference that represents the root of the tree
- provides a method for inserting new nodes
- provides a method for printing “orderly” the nodes of the tree

For the insertion of the nodes we create a recursive method that runs through the tree until it finds the correct point in which the node has to be inserted and it also updates, when needed, the additional information about the maximum value of the subtree of the various nodes. For the tree printing, as we are interested in checking if the nodes have been inserted correctly according to the value of the range they contain and the maximum value of their subtree, we use the classical recursive algorithm for ordered traversal of a binary tree: **In-Order Tree Traversal**.

So we create the class **AugmentedIntervalTree** that at the time is as follows:

```
public class AugmentedIntervalTree {

    private Interval root;

    public static Interval insertNode(Interval tmp, Interval newNode) {
        if (tmp == null) {
            tmp = newNode;
            return tmp;
        }

        // Update of the maximum extreme of the subtree
        // during insertion
        if (newNode.getEnd() > tmp.getMax()) {
            tmp.setMax(newNode.getEnd());
        }

        if (tmp.compareTo(newNode) <= 0) {

            if (tmp.getRight() == null) {
                tmp.setRight(newNode);
            }
            else {
                insertNode(tmp.getRight(), newNode);
            }
        }
        else {
            if (tmp.getLeft() == null) {
                tmp.setLeft(newNode);
            }
```

```
            else {
                insertNode(tmp.getLeft(), newNode);
            }
        }
        return tmp;
    }

    // In-Order Tree Traversal
    public static void printTree(Interval tmp) {
        if (tmp == null) {
            return;
        }

        if (tmp.getLeft() != null) {
            printTree(tmp.getLeft());
        }

        System.out.(tmp);

        if (tmp.getRight() != null) {
            printTree(tmp.getRight());
        }
    }
}
```

At this point we add to the AugmentedIntervalTree class a main method where we configure as a test the scenario used so far, as an example. To do so, we instantiate an AugmentedIntervalTree object, we insert all nodes and then we print them in output as a ordered sequence. The tree structure is the one previously illustrated graphically in Picture 2, for which the ordered sequence of nodes that we expect as output will be:

```
[1, 12, 12][2, 8, 8][5, 10, 25][8, 16, 20][14, 20, 20][15, 25, 25][18, 21, 21]
```

So, we can change the AugmentedIntervalTree class adding the main method:

```
public class AugmentedIntervalTree {

    private Interval root;

    public static Interval insertNode(Interval tmp, Interval newNode) {
        if (tmp == null) {
            tmp = newNode;
```

```
        return tmp;
    }

    if (newNode.getEnd() > tmp.getMax()) {
        tmp.setMax(newNode.getEnd());
    }

    if (tmp.compareTo(newNode) <= 0) {

        if (tmp.getRight() == null) {
            tmp.setRight(newNode);
        }
        else {
            insertNode(tmp.getRight(), newNode);
        }
    }
    else {
        if (tmp.getLeft() == null) {
            tmp.setLeft(newNode);
        }
        else {
            insertNode(tmp.getLeft(), newNode);
        }
    }
    return tmp;
}

public static void printTree(Interval tmp) {
    if (tmp == null) {
        return;
    }

    if (tmp.getLeft() != null) {
        printTree(tmp.getLeft());
    }

    System.out.(tmp);

    if (tmp.getRight() != null) {
        printTree(tmp.getRight());
    }
}

public static void main(String[] args) {
```

```
        AugmentedIntervalTree ait = new AugmentedIntervalTree();

        ait.root = insertNode(ait.root, new Interval(5, 10));
        ait.root = insertNode(ait.root, new Interval(15, 25));
        ait.root = insertNode(ait.root, new Interval(1, 12));
        ait.root = insertNode(ait.root, new Interval(8, 16));
        ait.root = insertNode(ait.root, new Interval(14, 20));
        ait.root = insertNode(ait.root, new Interval(18, 21));
        ait.root = insertNode(ait.root, new Interval(2, 8));

        printTree(ait.root);
    }
}
```

By running the program we get the following result:

```
[1, 12, 12][2, 8, 8][5, 10, 25][8, 16, 20][14, 20, 20][15, 25, 25][18, 21, 21]
```

As we can see, this result coincides with what we expected.

At this point we can proceed with the implementation of the functionality for which we turned in this type of data structure, that is the verification of the **overlap between our given intervals and a list of query intervals**. For this purpose, we define a new method that, for each query interval, it traverses the tree and adds the intervals with whom it overlaps to a list.

In making the search for nodes within the Augmented Interval Tree we will use the information we have added, relative to the maximum value of the upper bound found in the subtree of each node, to maximize the search performance and exclude unnecessary comparisons. The way we use it is the following: **if the maximum value of the subtree of the left child of a node is lower than the starting point of the query range, then we can exclude from the comparison all the nodes in the entire subtree and go directly to the right subtree of the node.**
We will modify our class AugmentedIntervalTree, adding:

- a recursive method **intersectInterval** which takes two Intervals as parameters, representing the node of the tree we are analyzing and the query node, and a list of intervals in which are added those that

intersect the query node
- a list of query intervals in the test main for which we verify the overlapping with the intersectInterval method and we print in output the results

Again we use the intervals from the example seen so far, so we insert in the list of query intervals to check: [8,10] e [20,22]

As previously highlighted in Picture 1:

- the query interval [8,10] overlaps with intervals: [5, 10], [1, 12], [2, 8], [8, 16]
- the query interval [20, 22] overlaps with intervals: [15, 25, 25], [14, 20, 20], [18, 21, 21]

After considerations above, our AugmentedIntervalTree class becomes:

```
public class AugmentedIntervalTree {

   private Interval root;

   public static Interval insertNode(Interval tmp, Interval newNode) {
      if (tmp == null) {
         tmp = newNode;
         return tmp;
      }

      if (newNode.getEnd() > tmp.getMax()) {
         tmp.setMax(newNode.getEnd());
      }

      if (tmp.compareTo(newNode) <= 0) {

         if (tmp.getRight() == null) {
            tmp.setRight(newNode);
         }
         else {
            insertNode(tmp.getRight(), newNode);
         }
      }
      else {
         if (tmp.getLeft() == null) {
            tmp.setLeft(newNode);
```

```
        }
        else {
            insertNode(tmp.getLeft(), newNode);
        }
    }
    return tmp;
}

public static void printTree(Interval tmp) {
    if (tmp == null) {
        return;
    }

    if (tmp.getLeft() != null) {
        printTree(tmp.getLeft());
    }

    System.out.(tmp);

    if (tmp.getRight() != null) {
        printTree(tmp.getRight());
    }
}

public void intersectInterval(Interval tmp, Interval i, List<Interval> res) {

    if (tmp == null) {
        return;
    }

    if (!((tmp.getStart() > i.getEnd()) || (tmp.getEnd() < i.getStart()))) {
        if (res == null) {
            res = new ArrayList<Interval>();
        }
        res.add(tmp);
    }

    if ((tmp.getLeft() != null) && (tmp.getLeft().getMax() >= i.getStart())) {
        this.intersectInterval(tmp.getLeft(), i, res);
    }

    this.intersectInterval(tmp.getRight(), i, res);
}
```

```
    public static void main(String[] args) {

        AugmentedIntervalTree ait = new AugmentedIntervalTree();

        ait.root = insertNode(ait.root, new Interval(5, 10));
        ait.root = insertNode(ait.root, new Interval(15, 25));
        ait.root = insertNode(ait.root, new Interval(1, 12));
        ait.root = insertNode(ait.root, new Interval(8, 16));
        ait.root = insertNode(ait.root, new Interval(14, 20));
        ait.root = insertNode(ait.root, new Interval(18, 21));
        ait.root = insertNode(ait.root, new Interval(2, 8));

        printTree(ait.root);
        System.out.println("\n");

        List<Interval> queries = new ArrayList<>();
        queries.add(new Interval(8, 10));
        queries.add(new Interval(20, 22));

        List<Interval> over = new ArrayList<>();

        for (Interval i : queries) {
            over.clear();
            ait.intersectInterval(ait.root, i, over);
            System.out.println("Nodes overlapping [" + i.getStart() + ", " + i.getEnd() +
"]:");
            for (Interval ris : over) {
                System.out.println(ris);
            }
        }
    }
}
```

By running the program, we get the following result, in which as first the ordered tree is printed out and then are listed, for each query interval, the intervals that it overlaps with:

[1, 12, 12][2, 8, 8][5, 10, 25][8, 16, 20][14, 20, 20][15, 25, 25][18, 21, 21]

Nodes overlapping [8, 10]:
[5, 10, 25]
[1, 12, 12]
[2, 8, 8]
[8, 16, 20]
Nodes overlapping [20, 22]:
[15, 25, 25]
[14, 20, 20]
[18, 21, 21]

As we can see, once again, the result is exactly what we expected. Adding a debug printing in the intersectInterval method that indicates the node with which the query interval is compared, we can see where the algorithm uses the information of the subtree maximum value with which we augmented our Interval Tree. For example, for the query interval [20,22] we obtain the following result:

Comparing:[5, 10, 25] and [20, 22, 22]
Comparing: [15, 25, 25] and [20, 22, 22]
Comparing: [8, 16, 20] and [20, 22, 22]
Comparing: [14, 20, 20] and [20, 22, 22]
Comparing: [18, 21, 21] and [20, 22, 22]
Nodes overlapping [20, 22]:
[15, 25, 25]
[14, 20, 20]
[18, 21, 21]

Referring again to the tree structure shown in Picture 2, we can notice that after comparing the query interval with the root, the algorithm checks if the maximum upper value of the left child of the root itself (12) is greater than the lower bound of the query interval (20). In this case it is not, and this means that in the left subtree of the root there are no nodes representing intervals that may overlap with the query interval, so it excludes the entire left subtree from comparisons and it goes directly at analyzing the nodes of the right subtree of the root.

The complete code can be downloaded from here:

**Augmented Interval Tree data structure 1.22 KB**
**Download**