1) Let us consider the following variants of join/split operations.

a. Define a join($T_1$,$T_2$) operation, where all elements of $T_1$ are smaller than any element of $T_2$. Note that we do not give any value i for the root.

We may assume $T_1$ to be nonempty. We compute the maximum element i from $T_1$, and then we remove i from $T_1$. Finally, we call join(i,$T_1$,$T_2$). For a self-balanced binary search tree implementation, the runtime is in O(log(n)).

b. Define a split(x,T) operation, with x possibly not in T.

We may assume x not in T (otherwise, this is the classical split operation on binary search trees). There are two cases:
<u>Case y > min(T)</u>. Then, y=floor(x,T) exists. Let $T_1$,$T_2$ be the output of split(y,T). Finally, let $T_1$' be obtained from $T_1$ by inserting y. We output $T_1$',$T_2$.
<u>Case y < min(T)</u>. Then, y=ceil(x,T) exists. Let $T_1$,$T_2$ be the output of split(y,T). Finally, let $T_2$' be obtained from $T_2$ by inserting y. We output $T_1$,$T_2$'.
For a self-balanced binary search tree implementation, the runtime is in O(log(n)).

2) Propose an efficient data structure which dynamically maintains a rooted tree subject to the following operations:
- insert(u,v): add a new leaf with father v
- remove(u): deletes leaf u (undefined if u is not a leaf)
- lca(u,v): returns the lca of nodes u,v.

If only insertions are allowed, then the following approach works (already discussed in a previous seminar): stores at each node u its level and pointers p(u,j) to its $2^j$-closest ancestors. In this situation insert and lca can be done in O(log(n)). To remove a leaf u is in O(1) because there is no x,j such that p(x,j) = u.

The following variant for computing the lca was discussed in class (another variant can be found in the correction of a previous seminar).
i) Set x := u, y := v. Without loss of generality, x and y are on the same level (if not, then we can use the shortcuts in order find a new pair u,v so that it is the case).
ii) Set j := largest exponent so that p(x,j) and p(y,j) are defined.
  While p(x,j) $\neq$ p(y,j) do:
    x := p(x,j), y = p(y,j)
    j := largest exponent so that p(x,j) and p(y,j) are defined
  done
  From now on, j is defined such that lca(x,y) is at most at distance $2^j$ from x,y.
iii) Consider the following procedure lca(x,y,j):
    If j = 0 then either x=y or x,y have the same parent node
    Else if p(x,j-1) = p(y,j-1) then call lca(x,y,j-1)
    Else call lca(p(x,j-1),p(y,j-1),j-1)
Since each step runs in O(1) and j decreases at each step, the runtime is also in O(log(n)).

3) Propose an efficient data structure which dynamically maintains a rooted tree subject to the following operations:

- insert(u,v): add a new leaf with father v
- remove(u): deletes leaf u (undefined if u is not a leaf)
- size(u): returns the size of the subtree rooted at u.

We use an Euler tour tree (augmented Euler tour stored in a self-balanced binary search tree), where each node of the BST (edge or loop) further retains the size of its rooted subtree *in the BST* (not in the tree T represented). In doing so, insertion and removal of a leaf can be done in O(log(n)). We can also compute size(u) in O(log(n)), as follows:
- if u is the root, then we output n
- else, let $e_1$ = (v,u) and $e_2$ = (u,v), with v the father of u. We may map both edges in some Hash-table to their respective positions in the BST. In doing so (going backward until the root), we may simulate searching for $e_1$ and $e_2$. By searching for both edges in the BST, we can compute the respective numbers $n_1$,$n_2$ of elements (edges and loops) smaller than $e_1$,$e_2$ in the augmented Euler tour. In particular, there are $k = n_2 - n_1 - 1$ elements between $e_1$ and $e_2$. These k elements represent an augmented Euler tour of the subtree rooted at u. If size(u) = s, then there are s loops and s-1 edges, where each edge is repeated twice. As a result, k = 3*s-2.

4) Propose an efficient data structure which dynamically maintains a rooted tree *and* a preordering of its nodes subject to the following operations:
 - insert(u,v): add a new leaf with father v
 - remove(u): deletes leaf u (undefined if u is not a leaf)
 - preorder(u): returns the preorder of node u

 This is the same approach as for the previous question (encoding by an Euler tour tree). The main difference is that now, each node x in the BST stores the number p(x) of edges of the form (u->father,u) in its rooted subtree (i.e., it does not include the other edges and loops in the size of its rooted subtree). In doing so, we can compute preorder(u) in O(log(n)), as follows:
 - if u is the root, then we output 0.
 - else, let e = (v,u), with v the father of u. By searching for edge e in the BST, we can compute the number k of edges (y->father,y) that appear before e in the tour. Then, preorder(u) = k+1.

5) Consider an n-size vector v. Show that after a pre-processing in O(n*log(n)), we can answer in O(log(n)) to the following type of queries q(i,j): ``compute the number of *distinct* values e such that v[k] = e for some k between i and j.''
 – Hint: use 2-range trees.

 We scan the vector v and, each time we read an element v[i]=e, we maintain in a Hash-table H the last position i where e was found (by convention, before we see an element for the first time, its associated value in H equals -1). Furthermore, upon reading v[i], we also create a point (H[v[i]],i). Note that we only create n points. We put all these points in a 2-range tree. It takes O(n*log(n)). Now, to answer a query q(i,j), it suffices to compute the number of such points (x,y) such that: $x < i$ and $i \leq y \leq j$. Indeed, such a point witnesses a value between positions i and j whose latest appearance (if any) was before index i.

 a. (Discussed in class). By using Mo's trick, solve the above query problem in $O(n*\sqrt{n})$ pre-processing time and $O(\sqrt{n})$ query time.

i) We partition the vector in $O(\sqrt{n})$ blocks of size $O(\sqrt{n})$. Let $B_0, B_1, ..., B_q$ denote the blocks.
ii) Let a[i] denote the number of distinct elements in $B_i$. To compute this number, it suffices to scan the $i^{th}$ block. Each time we read a new value, we check whether this value was already inserted earlier in some auxiliary Hash table. The runtime is in $O(\sqrt{n})$ per block, and therefore it is in O(n) in total.
iii) Then, let b[i,j] be a list that contains all distinct elements in $B_i$ that *are not in any of* $B_{i+1} \cup B_{i+2} \cup ... \cup B_{j-1} \cup B_j$. For i=j, we put all distinct elements of block $B_i$ in b[i,i] (these elements were computed at the previous step ii). Then, in order to construct b[i,j+1] from b[i,j], it suffices to check for each element of b[i,j] whether it is contained in $B_{j+1}$. Since we put all elements of block $B_{j+1}$ in an auxiliary Hash-table (cf. step ii), this takes O(1) per element in b[i,j], and so at most $O(\sqrt{n})$. Overall, since there are n entries to compute, and that each entry can be computed in $O(\sqrt{n})$, the runtime is in $O(n*\sqrt{n})$.
iv) Finally, let c[i,j] be a Boolean value, equal to 1 if and only if v[i] is in $B_j$. We apply the ``partial sum trick'' in order to compute $d[i,j] = \sum c[i,j']$, j' = 0...j.

Now, to answer a query q(i,j), we proceed as follows.
* Let $B_p, B_{p+1}, ..., B_{p+t}$ denote all blocks that are fully between i and j.
* Now, consider all values v[k], $k \geq i$, that appear *before* the first block $B_p$. In the same way, consider all values v[k], $k \leq j$, that appear *after* the last block $B_{p+t}$. We put all these values in an $O(\sqrt{n})$-size auxiliary u. By using a Hash-table, we remove from u any repetition of an element (all values are now unique).
* We further remove from u any element that also appears in a block. For that, for each element v[k] in u, it suffices to check whether $d[k,p+t]-d[k,p-1] > 0$.
* We output $u.size() + \sum b[p+k,p+t].size()$, k=0...t.

6) Consider an n-node tree T. For each node v, let q(v) denotes the number of pairs of nodes (x,y) such that: v is an ancestor of x and y, $d(v,x) < d(v,y)$, $x.val > y.val$.
   a. Show that in O(n) we can edge-partition T in trees $T_1, T_2$ of respective orders between n/3 and 2n/3.

   We compute a centroid x of T and consider the components $C_1, C_2, ..., C_k$ of $T \backslash x$. If the largest component, say it is $C_1$, has order $\geq n/3$, then we put $T_1 := C_1 + x$ and $T_2 := T \setminus C_1$. Otherwise, let $i_1$ be the smallest index such that there are $\geq 2n/3$ nodes in the union of $C_1, C_2, ..., C_{i1}$. Note that there are $< 2n/3$ nodes in the union of $C_{i1}, ..., C_k$ because otherwise the order of $C_{i1}$ would be $\geq n/3$. We put $T_1 := \cup\{ x, C_1, C_2, ..., C_{i1-1}\}$ and $T_2 := \cup\{x, C_{i1}, ..., C_k\}$.

   b. Show that in O(n*log(n)) we can *edge*-partition T in $O(\sqrt{n})$ subtrees of order $O(\sqrt{n})$ each.

   We apply the centroid decomposition (see question a) until all gotten subtrees have order between $\sqrt{n}/3$ and $\sqrt{n}$.

   Note that in general it is not possible to partition the *nodes*. For instance, if we are given a star, then one subtree should contain the root, and then any other subtree should be reduced to a leaf. Therefore, we would obtain O(n) subtrees in our partition, not $O(\sqrt{n})$.

Remark: It is possible to compute an edge-partition as above in O(n) by dynamic programming.

c. Deduce from the above that after an $O(n*\sqrt{n}*\log(n))$ preprocessing, we can answer to any query q(v) in $O(\sqrt{n})$.

0) We construct the edge-partition of question c. Let $T_0,T_1,....,T_q$ be the subtrees of this edge-partition.

Then, every node is associated to exactly *one* subtree to which it belongs. More precisely, each node v different than the root is associated to the subtree which contains the edge (v,father[v]).

The set of nodes associated to one subtree is called a block. In what follows, let $B_0,B_1,....,B_q$ denote the blocks. By construction, there are $O(\sqrt{n})$ blocks of size $O(\sqrt{n})$.

i) Let $M_0$ be the n x $\sqrt{n}$ matrix where $M_0[x,i]$ denotes the number of nodes y in block i such that: x.level < y.level, and x.val > y.val.

In the same way, let $M_1$ be the $\sqrt{n}$ x n matrix where $M_1[i,y]$ denotes the number of nodes x in block i such that: x.level < y.level, and x.val > y.val.

Both matrices can be constructed naively in $O(n^2)$. However, this can be improved as follows: If we represent all nodes in a block i as 2-dimensional points (value,level), and put these points in a 2-range tree (that costs us $O(\sqrt{n}*\log(n))$), then we can compute $M_0[x,i]$ and $M_1[i,y]$ in $O(\log(n))$. Overall, we can construct $M_0,M_1$ in $O(n*\sqrt{n}*\log(n))$.

ii) Let now $M_2$ be the $\sqrt{n}$ x n matrix where $M_2[i,v]$ denotes the number of pairs (x,y) where x is in block i, y is in the subtree rooted at v, x.level < y.level, and x.val > y.val. For any i, by using $M_1$ we compute all entries $M_2[i,v]$ by dynamic programming on T (partial sum trick). So, we can construct $M_2$ in $O(n*\sqrt{n})$.

iii) Let $M_3$ be the n x $\sqrt{n}$ matrix where $M_3[v,i]$ denotes the number of pairs (x,y) where x is *both* in the subtree rooted at v *and* in the same block as v, y is in block i, x.level < y.level, and x.val > y.val. For every i,j, by using $M_0$ we can compute all entries $M_3[v,i]$, v in block j, by dynamic programming on the jth subtree of the edge-partition. It takes $O(\sqrt{n})$ time for j fixed. Overall, we can construct $M_3$ in $O(n*\sqrt{n})$.

Now, to answer a query q(v), we sum all values $M_2[i,v]$ for blocks i in the subtree rooted at v (each node may retain the list of all blocks in its subtree). In doing so, we compute the number of pairs (x,y) with x in a block of the subtree, and y anywhere in this subtree. However, in doing so we miss all pairs (x,y) such that: x is in the subtree, but the block containing x is *not fully* in the subtree rooted at v. (The same situation occurs for vectors, where we may have a block starting/ending before/after the interval we consider)

Let B(v) be the block containing v. We claim that every block, except maybe B(v), is either fully in the subtree rooted at v, or disjoint from it. Indeed, let us consider any block B that intersects both the subtree rooted at v and its complementary subtree. This block must be corresponding to the subtree containing edge (v,father[v]). As a result, we computed all desired pairs (x,y) except maybe those with x in B(v). If B(v) is *not* fully in the subtree rooted at v, then we also sum $M_3$[v,B(v)] with all values $M_3$[v,i].

The runtime is in O(√n) since there are at most O(√n) blocks.

d. Remark: the above problem for trees is inspired by the problem of computing invertions in a sub-vector. However, it is simpler because:
* for vectors, we were given two inputs for a query: i and j, and we wanted to count the number of invertions *only* between i and j
* for trees, we only have one input for a query: node v, and we want to count ``invertions'' in the subtree rooted at v.

Hence, for vectors, the real equivalent of our problem for trees would be as follows: *We are given an n-size vector v, that we want to pre-process in order to answer as fast as possible to the following type of queries q(i): ``count the number of invertions in the sub-vector v[i...n-1'']*.
➔ This problem can be solved faster than by using Mo's trick. Indeed, in O(n*log(n)) time, we can compute for every index i the number inv[i] of indices j > i such that v[i] > v[j]. Then, we apply a classic ``partial sum trick'', namely: let u[i] = ∑ inv[k], k=0...i. Now to answer to a query q(i), it suffices to output u[n-1] – u[i] + inv[i]. The pre-processing time is in O(n*log(n)), while the query time is in O(1).

It is not clear whether our problem for trees can also be solved faster than by using Mo's trick. *If we only consider pairs (x,y) such that x is an ancestor of y*, then indeed we can do better:
* We perform a DFS traversal. In doing so, we also compute a preordering. For each node v, let [p(v);q(v)] be the interval such that the descendants of v are exactly the nodes whose preorder is between p(v) and q(v). (In particular, p(v) is the preorder of node v). All these operations can be done in O(n).
* Then, for each node u, we create a 2-dimensional point (p(u),u.val). We put all these points in a 2-range tree. This can be done in O(n*log(n)).
* Now, given any node x, we may compute in O(log(n)) the number inv(x) of its descendants y such that x.val > y.val. For that, it suffices to compute the number of points (a,b) in our 2-range tree such that: p(x) ≤ a ≤ q(x); b < x.val, that is just a classical range query.
* Finally, for any node v, let u[v] be the sum of all values inv[x], x a descendant of v. We can compute all values u[v] by dynamic programming. Indeed, if v is a leaf, then u[v] = inv[v] = 0. Otherwise, let $x_1, x_2, \ldots, x_d$ be the children of node v. We have that u[v] = inv[v] + ∑ u[$x_i$], i = 1…d. **To answer to a query q(v), it now suffices to output u[v].**

A better equivalent of the invertion problem for trees could be as follows:
q(u,v): compute the number of pairs (x,y) such that
* x,y are nodes of the unique path between u and v

* $d(u,x) < d(u,y)$, i.e. u is closer to x and v is closer to y
* x.val > y.val.

We can use Mo’s trick in order to solve this above problem. But this is a bit more complicated...