Algorithm 1: Traversal (in preorder) of a tree representation that supports only firstchild, nextsibling and parent operations. The space is constant ($O(\log n)$ bits) and time is $O(n)$.

Input: Tree representation that supports `FirstChild`, `NextSibling` and `Parent` in constant time

```
currnode ← root;
direction ← down;
repeat
    if direction = down then
        nextnode = FirstChild(currnode);
        if nextnode = nil then
            direction ← up;
        else
            currnode = nextnode;
        end
    else
        nextnode = NextSibling(currnode);
        if nextnode = nil then
            currnode = Parent(currnode);
        else
            currnode = nextnode;
            direction ← down;
        end
    end
until currnode = root;
```

Algorithm 2: Marking of intervals in the `BWT` that correspond to maximal repeats. The space is constant ($O(\log n)$ bits) and time is $O(n \log \sigma)$.

Input: `ST` representation that supports `FirstChild`, `NextSibling`, `Parent` and `Interval` in constant time and `BWT`$[1..n]$ array that supports `rank` queries in $O(\log \sigma)$ time.

Output: A bitvector $B[1..n]$ in which $B[i] = 1$ (resp. $B[j] = 1$) iff i is leftmost (resp. rightmost) position in the bwt interval of a maximal repeat.

```
B[1..n] = 0^n;
currnode ← ST.root;
direction ← down;
repeat
    if direction = down then
        [i..j] ← Interval(currnode);
        c ← BWT[j];
        count = rank_BWT(c, j) − rank_BWT(c, i − 1);
        if count ≠ j − i + 1 then
            B[i] ← 1;
            B[j] ← 1;
        end
        nextnode = FirstChild(currnode);
        if nextnode = nil then
            direction ← up;
        else
            currnode = nextnode;
        end
    else
        nextnode = NextSibling(currnode);
        if nextnode = nil then
            nextnode = Parent(currnode);
        else
            currnode = nextnode;
            direction ← down;
        end
    end
until currnode = root;
```

Algorithm 3: Weiner link algorithm that exploits the array $B[1..n]$ that marks all the intervals of maximal repeats. The input consists in a bwt-interval corresponding to substring p and in a character c. the output is the interval corresponding to substring cp. The running time is $O(\log \sigma)$ (running time of rank query).

Input: A bwt-interval $[i..j]$ corresponding to some rightmaximal substring p, a character c, the `BWT`$[1..n]$ array that supports `rank` queries in $O(\log \sigma)$ time, the $C[1..\sigma]$ array and the bitvector $B[1..n]$ that marks starting and ending of bwtntervals of maximal repeats.

Output: The interval $[i', j']$ correponding to the substring cp

```
if (B[i] = 1 AND B[j] = 1) then
    i' ← C[c] + rank_BWT(c, i − 1) + 1;
    j' ← C[c] + rank_BWT(c, j);
    if j' < i' then
        return nil;
    end
else
    if BWT[j] = c then
        i' ← C[c] + rank_BWT(c, i − 1) + 1;
        j' = i' + j − i;
    else
        return nil;
    end
end
return [i', j'];
```