
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

```

1 currnode  $\leftarrow$  root;
2 direction  $\leftarrow$  down;
3 repeat
4   if direction = down then
5     nextnode = FirstChild(currnode);
6     if nextnode = nil then
7       direction  $\leftarrow$  up;
8     else
9       currnode = nextnode;
10    end
11  else
12    nextnode = NextSibling(currnode);
13    if nextnode = nil then
14      currnode = Parent(currnode);
15    else
16      currnode = nextnode;
17      direction  $\leftarrow$  down;
18    end
19  end
20 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 $\text{BWT}[1..n]$ 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.

```

1  $B[1..n] = 0^n$ ;
2  $\text{currnode} \leftarrow \text{ST.root}$ ;
3  $\text{direction} \leftarrow \text{down}$ ;
4 repeat
5   if  $\text{direction} = \text{down}$  then
6      $[i..j] \leftarrow \text{Interval}(\text{currnode})$ ;
7      $c \leftarrow \text{BWT}[j]$ ;
8      $\text{count} = \text{rank}_{\text{BWT}}(c, j) - \text{rank}_{\text{BWT}}(c, i - 1)$ ;
9     if  $\text{count} \neq j - i + 1$  then
10       $B[i] \leftarrow 1$ ;
11       $B[j] \leftarrow 1$ ;
12    end
13     $\text{nextnode} = \text{FirstChild}(\text{currnode})$ ;
14    if  $\text{nextnode} = \text{nil}$  then
15       $\text{direction} \leftarrow \text{up}$ ;
16    else
17       $\text{currnode} = \text{nextnode}$ ;
18    end
19  else
20     $\text{nextnode} = \text{NextSibling}(\text{currnode})$ ;
21    if  $\text{nextnode} = \text{nil}$  then
22       $\text{nextnode} = \text{Parent}(\text{currnode})$ ;
23    else
24       $\text{currnode} = \text{nextnode}$ ;
25       $\text{direction} \leftarrow \text{down}$ ;
26    end
27  end
28 until  $\text{currnode} = \text{root}$ ;

```

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 $\text{BWT}[1..n]$ 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 bwtintervals of maximal repeats.

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

```

1 if ( $B[i] = 1$  AND  $B[j] = 1$ ) then
2    $i' \leftarrow C[c] + \text{rank}_{\text{BWT}}(c, i - 1) + 1;$ 
3    $j' \leftarrow C[c] + \text{rank}_{\text{BWT}}(c, j);$ 
4   if  $j' < i'$  then
5     return nil;
6   end
7 else
8   if  $\text{BWT}[j] = c$  then
9      $i' \leftarrow C[c] + \text{rank}_{\text{BWT}}(c, i - 1) + 1;$ 
10     $j' = i' + j - i;$ 
11  else
12    return nil;
13  end
14 end
15 return  $[i', j'];$ 

```
