Exercice 1

Application of the Prefix, even Prefix Sum.

Consider this array of integers, where 0 values are non-significant, and ideally, should be suppressed.

(7,0,0,9,0,1,0,0,0,3).
Compute the new position of each non nul element, using the prefix sum algorithm. Show on this example how it works.

Assume we associate an array of same length as the original array you can name Original, which can be named Flags, where each entry is = to 1 if the element at the same indice is <> of 0, and = to 0, if not.

Flags=(1,0,0,1,0,1,0,0,0,1)

Then, compute sum prefix of Flags, and store it in another array you can name Index

Index=Prefix Sum(Flags)=(1,1,1,2,2,3,3,3,3,4)

Use Index to move non 0 elements of the original collection Original, to the position indicated by Index in the final collection, named Final.

7 goes to position 1; 9 should go to position 2, etc, so to have Final=(7,9,1,3,x,x,x,x,x,x). We do not care about what values are behind the x. As we can easily know that the maximum of the values in Index is equal to 4, we can deduce that only the 4 first values of Final are significative.

```
// algorithm to move elements, thanks to the Index calculated array

For all (i=1 to i=n) do in parallel

        If Flags[i]==1 { Final [Index[i]] = Original[i];}
```

The overall time complexity of this algorithm, is bounded by the one of the prefix sum (O(log n)), using O(n) processors on an EREW PRAM.
This prefix sum operation can be transformed as a work optimal algorithm (compared to the sequential work), by using the principle seen in the class (inspired by the Brent principle), that was illustrated on the maximum parallel computation. So, the prefix sum can become a O(log(n/log n)) parallel time operation, using only O(n/log n) processors of the EREW PRAM.

The other operations, like the one above to move non nul elements, being embarrassingly parallel, can easily be transformed into O(log n) parallel time operation, by asking each of the processors to execute log n iterations of the loop. So, in this work-optimal version, the complexity of the complete algorithm would be bounded by O(log n) parallel time, using O(n/log n) processors.

Exercice 2 (taken from Legrand/Robert book)

Yet another application of Prefix parallel operation.
Assume we have a linked tree, where each node holds three pointers:
father[i], left[i], right[i]. By starting from the root of the tree, if one follows these pointers, it is possible to traverse the tree and visit each node/leaf, applying what is known as the Euler tour. The traversal will be a prefix visit, i.e, a depth-first traversal: once at each node, first visit the left children, and once coming back at that node, follow the pointer to visit the right children, and so on.

The problem is to compute the distance of each element of the tree, from the root, **or the distance to the root**. By following a depth first traversal, one can build a list, this list length will be equal 3*total number of elements in the tree. If you associate to each element of this list some integer values, like -1, 0, or 1, and you apply a prefix sum, you should solve the problem easily. Of course, to each of the 3n elements of the list, you can consider you associate a parallel processor.
Depict the algorithm you propose. Evaluate its complexity
The solution is to consider to associate to each node of the tree 3 processors, named A,B,C. The head of the linked list is the processor A of the root node, and the tail is the processor C of the root node. For a given node:

- The processor A points to the processor A of its left son, if it exists, and else towards its own B processor
- The processor B points to the processor A of its right son, if it exists, and else towards its own C processor
- The processor C points to the processor B of its father if the node is a left node, and to the processor C of its father if the node is a right node. The processor C of the root node of the tree points to nil.

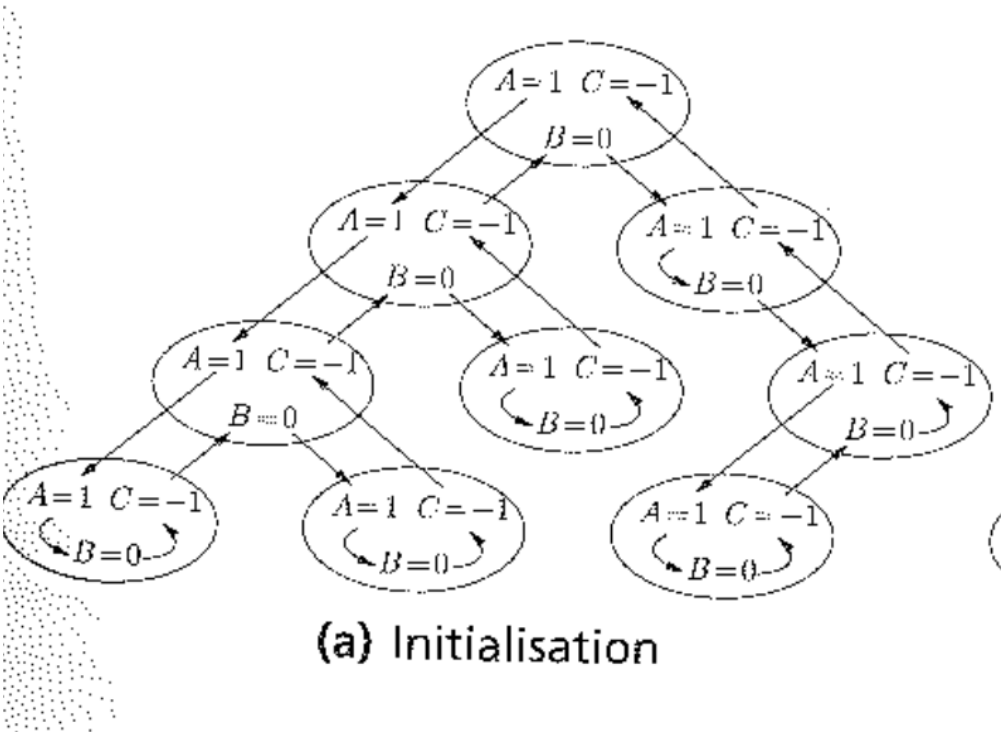
Each of the $3n$ processors i is responsible of a value value[i] initialized to 1 for all A type processors, to 0 for all B type processors, to -1 for all C type processors.

Then, compute the prefix sum of value[i] of the linked list. The final result (i.e., the depth of a node in the tree) is stored in the C processor associated to each node. Indeed, visiting a sub tree rooted at a given node does not modify the depth of that node, as it adds nothing to the current sum:

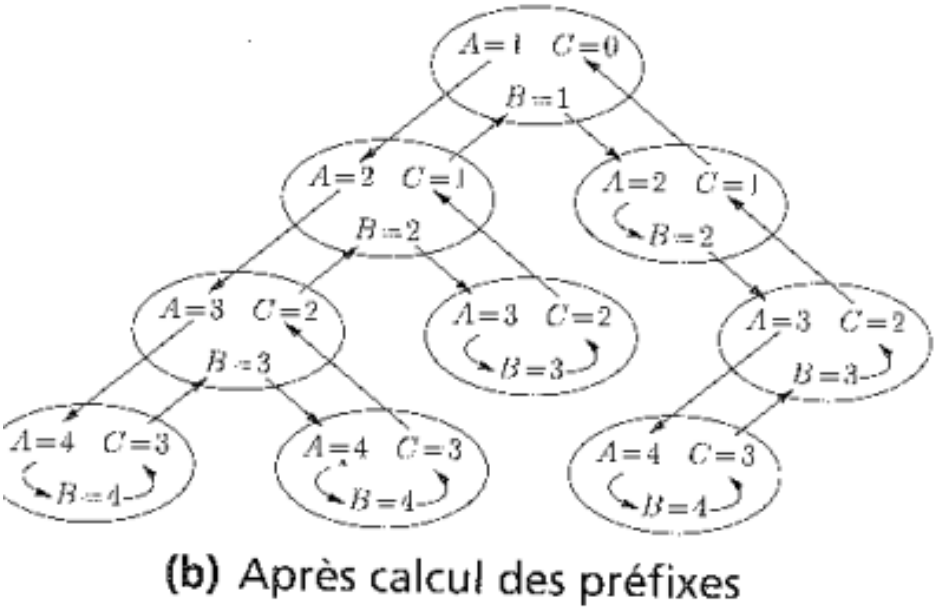
- The processor A of a node i contributes with +1 to the sum of the sub tree rooted in its left son, reflecting the fact that d(left[i]) = d[i] + 1; (d[i] is the depth of the node i).
- The processor B of a node i contributes with 0, because the depth of its left son is the same as the depth of its right son.
- The processor C of a node i contributes with -1.

The complexity in time is O(log $3n$) = O(log n) if the tree has n nodes. It is simply a prefix sum computation, which can run on an EREW PRAM, using O($3n$) =O(n) processors. As for

the Version 2 of Prefix studied in the class, there is not an easy way to make this algorithm becomes work optimal, because it is based upon a linked list.



(a) Initialisation



(b) Après calcul des préfixes