Politecnico di Milano

Progetto di Ingegneria Informatica

Professor Alessandro Barenghi

Academic Year 2025/2026

PRNG Tree Library:

Binary Tree Initialization, Navigation, and Leaf Access

Students:

Enrico Chen (enrico.chen@mail.polimi.it)

Emanuele Brotti (emanuele.brotti@mail.polimi.it)

Contents

1 Introduction

This documentation describes the PRNG Tree library, which provides a PRNG-based binary tree data structure designed for experimentation, educational purposes, and testing. The library provides utilities for building, manipulating, and traversing pseudo-randomly generated trees, enabling deterministic reproducibility through fixed seeds. It is intended to be lightweight, portable, and easily integrable into other C projects.

The project was developed as part of the "Progetto di Ingegneria Informatica" course (Academic Year 2025/2026) at Politecnico di Milano. It was designed and built by Emanuele Brotti and Enrico Chen, under the academic supervision of Professor Alessandro Barenghi.

2 Overview

Each node is represented by an `unsigned char seed`. The tree is not stored in memory; instead, it is generated at runtime using a pseudo-random number generator (PRNG). The `shake256` function, provided by CROSS-implementation, takes a seed as input and outputs a new seed with twice the length. The left and right children are extracted from this concatenated result using the `leftSeed` and `rightSeed` functions from `utilityFunctions.h/c`. The library works with unbalanced trees, as long as each left subtree is balanced.

The tree is initialized by calling the `INIT` function defined in `PRNGTree.h/c`. During initialization, the user provides the root `seed`, `bitmask`, and the array `leftTreeLevels`. The `bitmask` is an array of 0s (hidden) and 1s (visible) indicating which leaves should be printed, where index 0 corresponds to the leftmost leaf and the last index corresponds to the rightmost leaf. Similarly, `leftTreeLevels` specifies the heights of each left subtree along the rightmost path of the tree, where index 0 corresponds to the root and the last index corresponds to the rightmost node.

The function `nextLeaf` in `PRNGTree.h/c` prints the next leaf in sequence with each call, skipping any leaves hidden by the `bitmask`: the first call prints the first visible leaf, the second call prints the next visible leaf, and so on.

To access the seed of a specific leaf, the user calls the function `findLeaf` in `PRNGTree.h/c` with an integer index as a parameter. The function prints the seed if the corresponding `bitmask` value is 1, or an `"X"` if the leaf is hidden. The leftmost leaf has index 0, while the rightmost leaf has index `"number of leaves - 1"`.

The library is divided into two main components:

- `utilityFunctions.h/c`: seed manipulation functions, PRNG functions, stack operations, and conversion utilities.
- `PRNGTree.h/c`: functions to initialize the PRNG tree and print the next seed leaf or a specific one.

3 Algorithm `nextLeaf`

`nextLeaf`, together with `nextLeafAux`, is used to traverse all leaves and print only the ones visible in the `bitmask`. `tree->stack` keeps track of the nodes visited during each call. `iterations` indicates the number of times `nextLeaf` was called, to keep track of the current leaf. `leftSubTreeLevels` is used to determine how far to descend on the left branch before reaching a leaf.

3.1 `nextLeaf`

3.1.1 Check if the stack is empty:

1. If the stack is empty, start from `root`.
2. If the root has left children, call `nextLeafAux 3.2.1`, using the `root` as parameter, and return.
3. If there are no left children:
    - If other right nodes exist, push the root onto the stack.
    - Otherwise, the root is a leaf: print it and return.

3.1.2 Return to upper nodes:

1. Pop the last visited node from the stack.
2. Generate the right child (`rightSeed`) of the popped node.
3. If the stack becomes empty, the removed node was part of the rightmost path. Additional checks are made on its right child:
   - If it has left children, continue as normal.
   - Otherwise, move to the next right node.
   - If none are present, print it as a leaf and return.
4. Call `nextLeafAux` 3.2.1, using the right child as parameter, and return.

3.2 `nextLeafAux`

3.2.1 Descend along the left branch

1. Save the current seed in the stack (if it is not a leaf).
2. Generate children using `shake256` and take the left child (`leftSeed`).
3. Increment the level and repeat until reaching `"subTreeLevel - 1"`, and hence the leaf.

3.2.2 Check bitmask and print leaf

1. If `bitmask[iterations] = 1` $\Rightarrow$ print the leaf.
2. Otherwise, call `nextLeaf` 3.1.2 again, until a valid leaf is found.

In either case, increment `Iterations`. If it equals the number of leaves, reset to 0.

3.3 Example 1: Balanced Tree

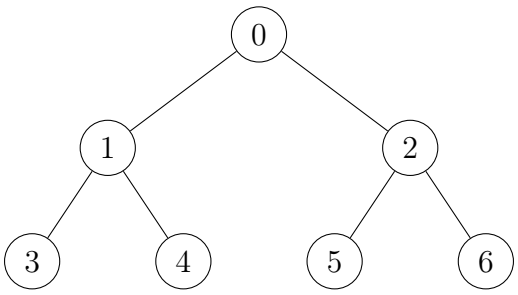

Figure 1: A Balanced Tree with 4 leaves

- `leftSubTreeLevels = [2, 1, 0]`
- `bitmask = [1, 0, 1, 0]`
- Leaves: 3, 4, 5, 6

Stack and Traversal Steps:

1. `nextLeaf` 3.1.2 is called, the stack is empty. The root has left children → Call `nextLeafAux` 3.2.1, using the root as parameter, then return.
2. `nextLeafAux` starts at root, pushing 0, 1 onto the stack. Stops after reaching leaf 3.
3. bitmask[0] = 1 → Print 3.
4. `nextLeaf` is called, the stack contains [0,1]. Pop 1, call `nextLeafAux`, using its right child (4) as parameter, then return.
5. bitmask[1] = 0 → Call `nextLeaf` again.
6. `nextLeaf` is called, the stack contains [0]. Pop 0, the stack is now empty: the next right node (2) has left children → continue as normal. Call `nextLeafAux`, using its right child (2) as parameter, and return.
7. `nextLeafAux` starts at 2, pushing 2 onto the stack. Stops after reaching leaf 5.
8. bitmask[2] = 1 → Print 5.

9. `nextLeaf` is called, the stack contains [2]. Pop 2, the stack is now empty: the next right node (6) does not have left children and it's the rightmost node → It's a leaf.

10. bitmask[3] = 0 → `Iterations` reset to 0, call `nextLeaf` again (return to step 1).

Output: 3, 5, 3

3.4 Example 2: Unbalanced Tree

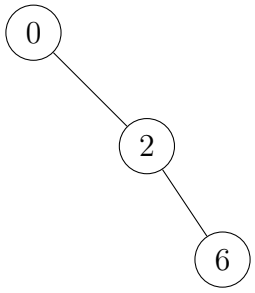

Figure 2: An Unbalanced Tree with 1 leaf

- `leftSubTreeLevels = [0, 0, 0]`
- `bitmask = [1]`
- Leaf: 6

Stack and Traversal Steps:

1. `nextLeaf` 3.1.2 is called, the stack is empty. The root does not have left children → Push root onto stack.

2. the stack contains [0]. Pop 0, the stack is now empty: the next right node (2) does not have left children → move to next right node.

3. 6 does not have left children and is the rightmost node → It's a leaf.

4. bitmask[0] = 1 → Print 6. `Iterations` reset to 0.

Output: 6

4 Algorithm `findLeaf`

The `position` array represents the target leaf's index in binary form. Each element in the array is either 0 or 1, corresponding to a direction to traverse the tree:

1. Starting from the root (level 0), check the value at `position[0]`:
   - If it is 0, proceed to the left child.
   - If it is 1, proceed to the right child.
2. Move to the next level and repeat the process using the next value.
3. Continue until reaching the last level of the tree, and hence the target.

4.1 findLeaf

Given a target position X:

1. Verify that X is within bounds.
2. Call `selectSubtree` 4.2

4.2 selectSubTree: Select the Correct Subtree

When the data structure is created, the user inputs the height of each left subtree. Therefore, the number of leaves can be calculated as:

$$2^{(\text{height}-1)}.$$

1. Start from the root's left subtree and calculate its number of leaves.
2. If $X <$ numLeaves, the leaf is contained in that subtree:
   - Apply the perfect-tree search algorithm 4.3 to that subtree.
3. Otherwise:
   $$X \leftarrow X - \text{numLeaves}$$
   and repeat step 1 to the next left subtree.

- If the final rightmost node is reached and it has no left child, that node is a leaf and must be printed.

4.3 findLeafAux: Search the Leaf in a Perfect Subtree

1. Verify that the provided seed is not `NULL`.
2. If the current visited level is the last, the node is a leaf:
   - Output the current seed using `printSeed`, then return.
3. Generate children using `shake256`.
4. Check the `position` array at the index corresponding to the current visited level:
   - If the value is 0, proceed with the left child:
   - If the value is 1, proceed with the right child:
5. Move to the next level by recursively calling `findLeafAux` with the selected child seed.

4.4 Example

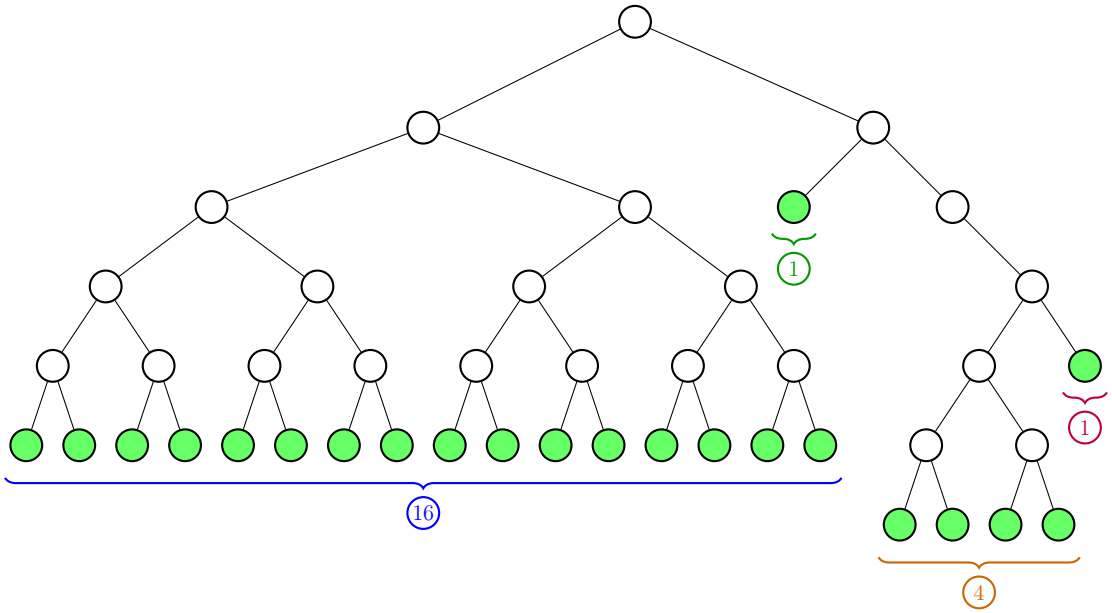

Figure 3: An Unbalanced Tree with 22 leaves

- `leftSubTreeLevels = [5, 1, 0, 3, 0]`
- Valid positions range from 0 to 21.

Case $X = 21$

1. Root subtree: height $5 \Rightarrow 2^4 = 16$ leaves. $21 > 16 \Rightarrow X = 5$.
2. Next subtree: height $1 \Rightarrow 2^0 = 1$ leaf. $5 > 1 \Rightarrow X = 4$.
3. Next subtree: height $0 \Rightarrow 0$ leaves. $4 > 0 \Rightarrow X = 4$.
4. Next subtree: height $3 \Rightarrow 2^2 = 4$ leaves. $4 = 4 \Rightarrow X = 0$.
5. The last rightmost node has no left child, hence it is a leaf.

$$\Rightarrow \text{Print the 22nd leaf.}$$

Case $X = 19$

1. Root subtree: $2^4 = 16$ leaves. $19 > 16 \Rightarrow X = 3$.
2. Next subtree: $2^0 = 1$ leaf. $3 > 1 \Rightarrow X = 2$.
3. Next subtree: height $0 \Rightarrow 0$ leaves. $2 > 0 \Rightarrow X = 2$.
4. Next subtree: height $3 \Rightarrow 2^2 = 4$ leaves. $2 < 4 \Rightarrow$ the leaf is inside this subtree.
   - Convert $X = 2$ to binary: 10.
   - Apply the perfect-tree search algorithm.

$$\Rightarrow \text{Print the 20th leaf.}$$

5 Time Complexity Analysis of `nextLeaf` and `findLeaf`

5.1 Balanced Tree

In a *perfect binary tree*, the best-case, average-case, and worst-case time complexities are the same. This happens because every path from the root to any leaf has the same length.
Let n be the total number of leaves, if the tree has height h, then the number of leaves is 2^h, and therefore:

$$h = \log_2(\text{n}).$$

Consequently, the three complexity cases coincide:

$$T_{\text{best}}(n) = T_{\text{avg}}(n) = T_{\text{worst}}(n) = \Theta(h) = \Theta(\log_2(\text{n})).$$

5.2 Unbalanced Tree

Worst-Case Scenario

Let n be the total number of leaves:

$$T_{\text{worst}}(n) = O(h) \tag{1}$$

where h is the height of the longest path from the root to a leaf.

6 Pseudocodes

6.1 Function Prototypes

`TreeData INIT(root, bitmask, bitmaskSize, leftTreeLevels, leftTreeLevelsSize)`

Input:

- `root`: the seed stored at the root of the tree;
- `bitmask`: an array of $0s$ and $1s$ specifying which leaves are visible;
- `bitmaskSize`: the total number of leaves;
- `leftTreeLevels`: an array giving the height of each left subtree along the rightmost path;
- `leftTreeLevelsSize`: the length of the rightmost path.

Output: A `TreeData` structure initialized for leaf traversal.

`void nextLeaf(tree)`

Input:

- `tree`: a pointer to the current `TreeData` structure.

Effect: Advances the tree traversal and prints the next visible leaf.

`void findLeaf(tree, index)`

Input:

- `tree`: a pointer to the current `TreeData` structure;
- `index`: the position of the target leaf.

Effect: Locates and prints the leaf at the given index, if it is visible.

6.2 nextLeftAux

```
nextLeafAux(T, s, h)
 1   n = T.bitmaskSize
 2   curr = s
 3   while h < T.subTreeLevel − 1
 4       push(T.stack, curr, h)
 5       h = h + 1
 6       children = RNG(curr)
 7       curr = leftSeed(children)
 8   if T.bitmask[T.iter] == 1
 9       printSeed(curr, T.iter, T)
10       T.iter = (T.iter + 1) mod n
11   else
12       T.iter = (T.iter + 1) mod n
13       nextLeaf(T)
```

The procedure receives:

- T: the `TreeData` structure representing the current traversal state;
- s: the seed of the root of the current subtree;
- h: the level of s within the subtree.

The procedure saves the traversal state needed to reach the next available leaf during each `NextLeaf` invocation. Leaves are visited from left to right.
Starting from a given node, lines **3**–**7** push its leftmost children onto the stack. When `NextLeaf` is invoked again, the node in the lowest level will be extracted from the stack, and `NextLeafAux` will continue the traversal from its right child. The loop ends when a leaf is found. At that point, lines **8**–**13** compare the leaf's position with the bitmask, to determine its visibility. If it is visible, the leaf is printed. Else, `NextLeaf` is invoked again to continue the traversal. After reaching the last leaf, `iter` is set to 0 and the procedure returns to the first available leaf.

6.3 nextLeaf

```
nextLeaf(T)
 1  if stackEmpty(T.stack)
 2      T.currentRightMostNode = 0
 3      if T.leftSubTreeLevels[0] ≠ 0
 4          T.subTreeLevel = T.leftSubTreeLevels[0] + 1
 5          nextLeafAux(T, T.root, 0)
 6          return
 7      if T.leftSubTreeLevelsSize == 1
 8          printSeed(T.root, 0, T)
 9          return
10      push(T.stack, T.root, 0)
11  h = T.stack.levels[T.stack.size − 1]
12  seed = pop(T.stack)
13  if stackEmpty(T.stack)
14      if T.currentRightMostNode == T.leftSubTreeLevelsSize − 1
15          if T.leftSubTreeLevels[T.currentRightMostNode] == 0
16              if T.bitmask[T.iter] == 1
17                  printSeed(seed, T.iter, T)
18              T.iter = (T.iter + 1) mod T.bitmaskSize
19              return
20          nextLeaf(T)
21          return
22      T.currentRightMostNode = T.currentRightMostNode + 1
23      if T.leftSubTreeLevels[T.currentRightMostNode] == 0
24          children = RNG(seed)
25          right = rightSeed(children)
26          push(T.stack, right, T.currentRightMostNode)
27          nextLeaf(T)
28          return
29      T.subTreeLevel = T.leftSubTreeLevels[T.currentRightMostNode] +
          1 + T.currentRightMostNode
30  children = RNG(seed)
31  right = rightSeed(children)
32  nextLeafAux(T, right, h + 1)
```

The procedure continues the traversal to the next visible leaf.
The stack stores the current state of the traversal:

- `currentRightMostNode` identifies the current node along the rightmost path of the tree;
- the array `leftSubTreeLevels` specifies the heights of each left subtree, from the root to the rightmost node.

Lines **1**–**10** handle the case of an empty stack, which occurs when the procedure is invoked for the first time:

- If the tree consists of a single node, the root is printed as the only visible leaf;
- If the root's left child is missing, the procedure jumps to the next node in the rightmost path (its right child). To do so, the root is pushed onto the stack, to indicate that all its leftmost nodes were already visited;
- Otherwise, the procedure continues as normal.

Then, the node in the lowest level is extracted from the stack.
Lines **13**–**29** handle the case in which the popped node lies on the rightmost path.

- If the current node's left subtree is empty, the procedure jumps to the next node in the rightmost path (its right child).
- If the current node is the rightmost, and its left subtree is empty, the rightmost node is the last leaf to visit.
- Otherwise, the procedure continues as normal.

Finally, lines **30**–**32** invoke **nextLeafAux** by passing the current's node right child, and the procedure continues the traversal until the next leaf is found.

6.4 findLeafAux

```
findLeafAux(T, s, pos, i, ℓ, H)
 1   if s == NIL
 2       return
 3   if ℓ == H − 1
 4       printSeed(s, i, T)
 5       return
 6   children = RNG(s)
 7   if pos[ℓ] == 0
 8       next = leftSeed(children)
 9   else
10       next = rightSeed(children)
11   findLeafAux(T, next, pos, i, ℓ + 1, H)
```

The procedure navigates a perfect subtree in order to find the leaf identified by a given binary path. The subtree is assumed to be perfect.
Its parameters are:

- T: the `TreeData` structure representing the current traversal state;
- s: the seed identifying the root of the current subtree;
- pos: the binary path representing the traversal path of the target leaf;
- i: the leaf index to print;
- ℓ: the current level;
- H: the height of the subtree.

The array `pos` is assumed to be a valid binary path of length $H - 1$.
In line **3**, when $\ell = H - 1$, the traversal has reached the selected leaf.
In lines **7**–**10**, the value `pos[`ℓ`]` determines whether the traversal should proceed to the left or to the right child.
in line **11**, the procedure is invoked again, using the selected path on the next tree level.

6.5 selectSubtree

```
selectSubtree(T, s, j, i, g)
 1   k = i
 2   if T.leftSubTreeLevels[j] ≠ 0
 3       L = 2^(T.leftSubTreeLevels[j]−1)
 4       if k < L
 5           pos = indexToPath(T.leftSubTreeLevels[j], k)
 6           children = RNG(s)
 7           next = leftSeed(children)
 8           findLeafAux(T, next, pos, g, 0, T.leftSubTreeLevels[j])
 9           return
10       k = k − L
11   if j == T.leftSubTreeLevelsSize − 1
12       if k == 0
13           printSeed(s, T.bitmaskSize − 1, T)
14       else
15           return
16       return
17   children = RNG(s)
18   next = rightSeed(children)
19   selectSubtree(T, next, j + 1, k, g)
```

The procedure determines which subtree contains the leaf with global index i.
Its parameters are:

- T: the `TreeData` structure representing the current traversal state;
- s: the seed of the root of the current subtree;
- j: the position along the rightmost path;
- i: the target leaf index in the tree;
- g: the original leaf index to be printed.

The tree is divided into contiguous blocks, each corresponding to a left subtree. Since each left subtree is assumed to be perfect, the $j-th$ left subtree contains $L = 2^{T.\text{leftSubTreeLevels}[j]-1}$ leaves.
The leaf enumeration for each block goes from 0 to $L - 1$.
Lines **4**–**9** compare i to the number of leaves of the current subtree. If the index is lower, the target leaf lies in this block, and `FindLeafAux` is invoked using the selected subtree.

Otherwise, in line **10** the algorithm subtracts L from k, skipping the entire block. The updated index will be compared to the number of leaves of the next subtree. If j is the last position on the rightmost path, and the updated index equals 0, the rightmost node is the selected leaf. Otherwise, the procedure recurses on the right child, advancing to the next subtree along the path.

6.6 findLeaf

```
findLeaf(T, i)
 1   if i < 0 or i ≥ T.bitmaskSize
 2       return
 3   selectSubtree(T, T.root, 0, i, i)
```

The procedure finds the target leaf by a given global index i.
Lines **1–2** check that the index is within the valid range of the global enumeration. If the check succeeds, line **3** calls `SelectSubTree` starting from the root and finds the target leaf.