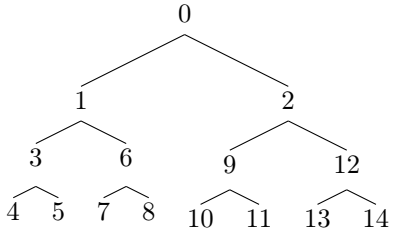
Implicit navigation in vEB layout

Consider $N = 2h - 1$ keys where h is a power of 2, and the implicit cache-oblivious vEB layout of their corresponding complete binary tree, where the keys are suitably permuted and stored in an array of length N without using pointers (as it happens in the classical implicit binary heap but the rule here is different). The root is in the first position of the array. Find a rule that, given the position of the current node, it is possible to locate in the array the positions of its left and right children. Discuss how to apply this layout to obtain (a) a static binary search tree and (b) a heap data structure, discussing the cache complexity

Assumptions

Suppose you have a binary vEB tree T stored in an array `T[]` in external memory, as seen in class, ordered as in the tree's induced "fractal" numbering, starting from 0:



This is the same order in which elements are stored in `T[]` Let n be the element whose children needs to be calculated.

Solution

First of all, we can check whether $n\%3 = 0$ in order to know whether n is the root of a minimal sub-tree (ie. a sub-tree with 3 nodes). If that's the case, then the left and right children of n are respectively $n + 1$ and $n + 2$, and we can return. Otherwise, we go on with the search:

1. Since h is a power of two, we can split T into subtrees of height $h/2$, in particular we obtain one top-tree and $2^{h/2}$ bottom-trees. Each of them contains $size = 2^{h/2} - 1$ elements. In order to know which sub-tree our element is part of, we calculate $tree\_id = \lfloor \frac{n}{size} \rfloor$. if $tree\_id = 0$ then the element is contained in the top tree, otherwise if $tree\_id > 0$ the element is contained in one of the bottom trees.
2. Once we identified the sub-tree, we have two possibilities:
   - *The element is in the top-tree*, then we do the following:

     `offset_sx = 0, jump = size, current_root = T.root`

     ($current\_root$ shall identify the root of the currently selected sub-tree, $offset\_sx$ shall count the number of "uncles and cousins" of n at the left of n)
   - *The element is in one of the bottom-trees*, then we have to transpose the root with

     `current_root += tree_id * size`

     and then adjust n with respect to the new root

     `n = n % size`

     Finally, we increment

     `offset_sx += (tree_id - 1) * size.`
3. We recursively repeat step 1 and 2, until we reach a minimal sub-tree (ie. $h = 2$), which is our base case. If the node ends up being the right child of the minimal subtree we need to further increment $offset\_sx+ = 1$, otherwise we are ready.

4. Finally, we can find the children of n with the following:

```
left_child   = current_root + jump + 2 * offset_sx * jump
right_child = current_root + jump + 2 * offset_sx * jump + jump
```

Discussion

It is possible to use a vEB tree as a binary search tree. In the worst case, during search h/k blocks are traversed in order to reach a leaf starting from the root (worst case, $k = logB$) The cost is then $O(logN/logB) = O(log_B N)$

If we instead use the vEB tree as a heap, during each deletion we have to replace the root with the right-most leaf. Then, we need to rebalance the new tree until the heap-ordering is re-established. This operation is done only on one path root-leaf, therefore the complexity stays the same, $O(log_B N)$.