1. Write down the level-order representation of the following two binary tries.

SOLUTION: For level-order representation, for each child write down 1 if the node has that child, and 0 if it doesn’t. Therefore for first tree the level-order representation is: 1 1 1 1 0 0 1 1 0 0 0 0. And for second tree it is: 1 1 1 1 0 0 1 0 1 1 0 0 0 0 0 0 1 0 0.

2. In Exercise Set 4 you augmented the RLZ parsing of a string so that later an arbitrary character $i$ in the original string could be accessed in $O(\log z)$ time, where $z$ is the number of phrases produced by the parsing. Show how to improve the character access time to $O(1)$. How has your modification affected the space usage?

SOLUTION: Use a bitvector for constant time rank and select operations. At the start of each phrase there is a 1. Then for position $i$ you get the phrase $j$ for it with $j = \text{rank}_{1}(i)$. The position within phrase is $i - \text{select}_{1}(j)$, because $\text{select}_{1}(j)$ is the start of phrase $j$. It adds $o(n)$ space usage.

3. Back in Lecture 3 we looked at the Simple-9 word-aligned integer code. Last lecture we saw how we could get random access to arrays of integers encoded with Elias $\gamma$ and vByte codes by first rearranging the bits in the codes appropriately, and then adding rank and select data structures.

Using the techniques you have learnt about in the last two weeks, devise a way to provide random access to arrays encoded with the Simple-9 scheme. What is the access time of your data structure? What space overhead does it add to the normal Simple-9 encoding?

SOLUTION: Split the code into selector (4 bits) and offset (28) parts. Create a bitvector for the offset bits with 1 at the start of every code word (use selectors to know where the starts are). Then to find $i$th code word, do $k = \text{select}_{1}(i)$ for the start. Alternately code the lengths of the code words with unary, and adjust select accordingly, since 1 is at the last bit of each code instead of first.
You can have a bitvector for selectors (in which case you have to copy each selector as many times as there are code words). But you don’t necessarily need it. As the offset parts are 28 bits and selectors are 4 bits, you can calculate that \( i^{th} \) code word is in word \( \lfloor k / 28 \rfloor \). Then the bits in the selector array are \([\lfloor k / 28 \rfloor \times 4, \ldots, (\lfloor k / 28 \rfloor + 1) \times 4 - 1]\). When you know the selector bits, you know how many bits to read in the offset array.

Creating the new selector and offset arrays requires \( O(n) \) space, where \( n \) is the length of the original array (but in practice you don’t then need the original array). Then the bitvector for \( o(n) \) space. The access time is constant.

4. Recall that the basic version of the Relative-10 word-aligned integer code uses four different selectors: same as last, one bigger, one smaller, and biggest. Use bitvectors to provide constant-time access to the \( i^{th} \) selector in an array encoded with the Relative-10 scheme.

SOLUTION: Here the wording is ambiguous. It’s not clear whether \( i^{th} \) selector means the selector for \( i^{th} \) code or \( i^{th} \) word. In the case of \( i^{th} \) code, we can use the solution in the previous problem. In the case of \( i^{th} \) word, the solution is even easier. Just set a 1 bit at the start of each word.

You know the selector is two bits, so read two bits and done.

5. (Challenge) Design a data structure that provides random access to interpolative binary codes. (Hint: group code words by length).

SOLUTION: This was a bonus question. One way to do this is to put the codes into a balanced binary tree so that each node holds one code, starting from the middle, and then always taking the middle code in the interval for the next node. Then left child corresponds to the left half interval and right side to right half interval.

When each node holds the current interval and the encoding for the middle number, you can get access to any code in \( O(\log n) \) time, where \( n \) is the number of codes, by traversing the tree.