1. The naïve implementation of the van Emde Boas layout requires storing pointers from each node to its children, but these pointers are not actually necessary. In an *implicit* van Emde Boas layout, only the values at the nodes are actually stored; the data structure consists entirely of a permuted array of values.

| H | D | L | B | A | C | F | E | G | J | I | K | N | M | O |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

   (a) Describe how to perform a binary search using the implicit van Emde Boas layout in $O(\log_B N)$ memory operations and $O(\lg N)$ time. For full credit, do not assume that the depth of the tree is a power of two.

   (b) Describe and analyze a cache-oblivious algorithm to permute an array of $N = 2^k - 1$ distinct values into an implicit van Emde Boas layout, in $O(n \log_m n)$ memory operations and $O(N \lg N)$ time. (These are the same time bounds as for sorting the array.)

2. The *packed memory structure* of Bender *et al.* maintains an ordered list of items under inserts and deletions, so that the items are stored in order in memory with constant-size gaps, which implies that $K$ contiguous items in the list can be scanned in optimal $O(K/B)$ memory operations in the cache-oblivious model. The list is subdivided into chunks of size $C = \Theta(\log N)$, which are maintained by brute force. The update algorithms rely on a (notional) balanced binary tree with height $h = \lg N - \lg C$ over the chunks. The density of any node in this tree is the ratio between the number of items in its subtree and the space used by those items. The packed memory structure maintains the invariant that the density of any node at depth $d$ lies between $\alpha_d$ and $\beta_d$, where

$$\alpha_d = \frac{1}{2} - \frac{d}{4h} \quad \text{and} \quad \beta_d = \frac{3}{4} + \frac{d}{4h}.$$

   Bender *et al.* show how to maintain these invariants using $O((\log^2 N)/B + 1)$ amortized memory operations per insertion or deletion.

   The description of this data structure relies on several seemingly arbitrary parameters—the chunk size $C$ and the density limits $\alpha_d$ and $\beta_d$. But in fact, these parameters are *not* arbitrary. Prove that modifying these parameters (but leaving the algorithm otherwise unchanged) cannot improve the amortized update cost by more than a constant factor without losing the optimal scanning cost. *[Hint: Start by showing that $\alpha_h$ must be a positive constant to get constant-size gaps.]*

3. Storing a complete binary search tree in the van Emde Boas layout allows us to perform any search in $O(\log_B N)$ memory operations in the cache-oblivious model. The goal of this problem is to generalize the van Emde Boas layout to arbitrary binary search trees.

   (a) Describe how to layout *any* binary search tree in memory so that *any* successful search requires $O(D/\lg B)$ memory operations, where $D$ is the depth of the target node.

   (b) What is the worst-case cost of a search if the tree consists of a single root-to-leaf path and is stored using your layout from part (a)?

   *(c) Describe a binary tree layout such that the cost of searching for a node at depth $f(N)$ is $O(f(N)/f(B))$. For example, if the length of the search path is $\sqrt{N}$, traversing that path should require only $O(\sqrt{N/B})$ memory operations; if the path length is $O(\log^2 N)$, the search cost should be $O(\log_B^2 N) = O(\log^2 N/\log^2 B)$. Alternately, prove that such a layout is impossible, and characterize the best possible "path-sensitive" layout.

4. Consider a complete binary tree with $N$ leaves, where each node has a distinct real *weight*, and the weight of any node is larger than the weight of its children (if any). In other words, the tree is a max-heap over the weights. The *frontier* of the tree is the set of nodes with negative weight whose parents have non-negative weight. There is an obvious depth-first-search algorithm to compute the frontier in $O(K)$ time, where $K$ is the number of frontier nodes. Your goal is to make this algorithm cache-oblivious by choosing an appropriate memory layout for the tree. The same layout should work for *any* assignment of weights (with the heap property).

   (a) Show that for the van Emde Boas layout and any value of $K$, the obvious search algorithm requires $\Omega(\min\{K, N/B\})$ memory operations in the worst case. *[Hint: The lower bound is obviously trivial when $K = 1$ or $K = N$. Prove it for $K \approx \sqrt{N}$ first.]*

   (b) Describe a randomized algorithm to lay out the tree in a contiguous block of memory, so that the obvious search algorithm uses only $O(K/\lg B)$ *expected* memory operations. The expectation is entirely over the internal bits of the layout algorithm; neither the search algorithm nor the (adversarial) assignment of weights is randomized.

   ⋆(c) Describe a deterministic layout that allows the frontier to be computed in $O(K/\lg B)$ memory operations in the worst case, or prove that there is no such layout.

   ⋆(d) Improve the $O(K/\lg B)$ expected I/O bound, ideally to $O(K/B)$, or prove that no such improvement is possible.

   This generic setup can be used to analyze algorithms that use *bounding volume hierarchies* for point location, ray shooting, range searching, collision detection, and other geometric problems.

∞. **Extra-credit open problem in case you're bored over winter break:**

   One method for specifying a memory layout for a binary search tree is through a series of *edge contractions*, just like in Borůvka's minimum-spanning tree algorithm. Contracting the edge from node $u$ to node $v$—replacing $u$ and $v$ with a single node $uv$—means that $v$ will be written into memory immediately after $u$. Note that $u$ and $v$ may already be the result of several contractions, in which case the subtree represented by $v$ will be (recursively) stored immediately after the subtree represented by $u$.

   For example, call an edge from a parent $u$ to a child $v$ *even* if the subtree rooted at $v$ has even depth. Contracting every even edge and recursing gives us the van Emde Boas layout!

   Consider the layout resulting from contracting the edges of a perfectly balanced binary search tree in random order.

   ⋆(a) Prove (or disprove) that the worst-case expected number of memory operations incurred by a binary search using this layout is $O(\log_B n)$.

   ⋆(b) What if the search tree isn't perfectly balanced? (See problem 3(c).)

   ★(c) Is this a good layout for problem 4(b)? (This is *not* the solution I had in mind.)

   ★(d) Can we efficiently maintain this (or some similar) layout under insertions and deletions?