Problem 1: In the dynamic predecessor problem we have a database of key/value pairs 
\((k_1, v_1), \ldots, (k_n, v_n)\), where all the \(k, v\)'s are integers, and we wish to support the following 
operations:

- **insert**\((k, v)\): Insert key \(k\) into the database with value \(v\). If an item with key \(k\) already 
  exists, overwrite its old value with \(v\).
- **pred**\((k)\): For the largest \(k' \leq k\) which exists as a key in the database, output \(k'\) and 
  its associated value. If no \(k' \leq k\) exists in the database, output **null**.

(a) (5 points) Show that for any \(\varepsilon \in [1/\log_2 B, 1]\) (the continuous interval), there ex-
ists a DAM data structure for the dynamic predecessor problem supporting **pred** in 
\(O(\varepsilon \log_B n)\) worst-case I/O’s and **insert** in \(O(\frac{1}{\varepsilon B} \log_B n)\) amortized I/O’s. Recall 
what an amortized bound means: it means that any sequence of \(T\) insertions costs at 
most \(O(T \frac{1}{\varepsilon B} \log_B n + 1)\) I/O’s.

**Hint:** Try a buffered repository tree but with a different internal branching factor on 
internal nodes.

(b) (5 points) Show how to maintain the same performance as (a), but where in addi-
tion to **insert** requiring \(O(\frac{1}{\varepsilon B} \log_B n)\) amortized I/O’s, it never requires more than 
\(O(\frac{1}{\varepsilon} \log_B n)\) I/O’s in the worst-case.

Problem 2: Consider the following Array Insertion Problem (AIP). We need to maintain 
\(n\) items in an array of size \(O(n)\) in a given order. The array is allowed to have some empty 
gaps in it with no items. That is, the only operations we want to suport are:

- **insert**\((x, y, z)\): Given pointers to \(x\) and \(y\) already in the AIP data structure, insert \(z\) 
  somewhere in the array interval between them. We are allowed to move items around.
- **delete**\((x)\): Given a pointer to \(x\) in the array, delete \(x\).

Every item in the database must be stored in one array slot, and we are only allowed to move 
items around in the array, with some empty cells allowed. We must obey the constraint that 
the relative order of items in the array respects the order specified by insertions.

A standard solution to the AIP problem, in RAM (not external memory!), is shown in 
Figure 1. The perfect binary search tree is conceptual and is not actually implemented in 
the data structure. We imagine the array is of length \(N = \Theta(n)\). We conceptually break
Figure 1: A conceptual perfect binary search of height $\log(N/\log N)$ tree built over $N/\log N$ leaves, each leaf being a contiguous chunk of $\log N$ cells in an array of length $N$.

up the array elements into contiguous chunks of size $\log N$ each. Each internal node $v$ in the conceptual binary search tree has an upper density $u(v)$ and lower density $\ell(v)$. For a node at depth $d$ (where the root has depth 0 and the bottom-most internal nodes have depth $h = \log(N/\log N)$) we define $u(v) = \frac{3}{4} + \frac{1}{4}d$ and $\ell(v) = \frac{1}{2} - \frac{1}{4}d$. The invariant we maintain is that for every $v$, the number of non-empty array cells in the subtree rooted at $v$ is at least a $d(v)$ fraction of the total cells in that subtree and at most a $u(v)$ fraction. Note these constraints become stricter when moving upward in the tree (toward the root).

Now, how do we perform an insertion of the item $z$ between two given elements $x$ and $y$? (Note: we may also be asked to insert before the first element or after the last element in the array.) Well, we just try to insert it somewhere in the same chunk as either $x$ or $y$, after $x$ and before $y$. If that violates the upper density constraint at that leaf chunk (i.e. the leaf is totally full), we conceptually traverse up the tree (in actuality we just scan the array to the left and to the right) until we reach a node $v$ such that inserting $z$ into the subtree rooted at that node does not violate the density constraints of that node. We then evenly redistribute all elements in the subtree rooted at $v$ into the array cells rooted at $v$. It is important to stress: the tree is only conceptual; it does not actually need to be coded in an implementation of this data structure. Deletion is done similarly. Note it may become impossible to satisfy the upper or lower density at the root, if we inserted or deleted too many items so that $n < N/2$ or $n > 3N/4$. In this case we just rebuild the entire structure from scratch of new size $N = 5n/8$ and redistribute all items evenly in the new array.

(a) (5 points) Show the amortized insertion and deletion costs in the data structure above are $O(\log^2 n)$. Note: this is in the RAM model and has nothing to do with the DAM.

(b) (5 points) Now for the DAM! Show that the AIP solution above can be combined with the static cache-oblivious B-tree from class to yield a dynamic cache-oblivious B-tree supporting both predecessor queries and also insertions/deletions, with amortized insertion and deletion cost $O(\log_B n + \frac{1}{B} (\log n)^2)$ I/O’s and queries in $O(\log_B n)$ I/O’s. The space consumption of this dynamic $B$-tree should be $O(n)$.