Algorithms in the Real World (15-853), Spring 2018
Assignment #6 Due: Tuesday, April 10, 2018

Complete all problems.

You are not permitted to look at solutions of previous years' assignments. You can work together in groups, but all solutions must be written up individually, and please list your collaborators. If you get information from sources other than the course notes and slides, please cite the information, even if from Wikipedia or a textbook.

For algorithm design problems, you should try to provide pseudocodes or pseudocode-like well organized verbal description. Then you should analyze the complexity (work and span) of your algorithms based on the pseudocodes. Please make your solutions as clear as possible, and they will be graded by the correctness and clearness. If only some "high-level" concepts are provided, you will get no more than half of the credits even if they are correct.

Problem 1: Buffered B-tree [30pts]

Consider the following mix of a B-tree and buffer tree supporting insertions and searches. Each internal node has $\Theta(B^\epsilon)$ children, for some constant parameter $0 < \epsilon < 1$. Moreover, each node has a buffer containing $\Theta(B)$ objects. A node (the buffer, child pointers, and pivots partitioning the children) is stored contiguously in 1 block.

To insert an object in the tree, first insert it into the root buffer. Partition that buffer according to the pivots. While any partition contains at least $\Omega(B^{1-\epsilon})$ objects, recursively insert $\Theta(B^{1-\epsilon})$ objects into the appropriate child. When a leaf node becomes full, split it, inserting a new child pointer into its parent. If an internal node has the maximum number of children, split it into two internal nodes, inserting a new child pointer into the parent. Each split is implemented as in a buffer tree and costs $\Theta(\text{nodesize}) = \Theta(1)$ block transfers.

To search for an object, follow the appropriate root-to-leaf path according to the pivots, and also search the entire buffer within each node along the path.

For parts (a)–(e), justify your answer with a *brief* proof sketch (at most a few sentences).

(a) [5 pts] What is the height of a tree containing N objects?

(b) [5 pts] Assuming the leaf node is already in memory, what is the amortized cost of inserting an object into a leaf (i.e., the cost of performing splits up the tree)? *More specifically, by "amortized cost" here, we mean suppose you start with an empty tree and perform a sequence of N insertions directly into the appropriate leaves (without moving down the tree through buffers). What is the average cost of each insertion, assuming that the relevant leaf is already in memory?*

(c) [5 pts] What is the amortized cost of moving an object down the tree to a leaf buffer? Assume the root node is always in memory. *Specifically, suppose you start with a tree containing N objects in which all buffers are empty; then perform a sequence of N insertions into the tree. What is the average cost of each insertion, assuming that node splitting happens for free?*

(d) [5 pts] What is the amortized cost of an insert starting from an empty tree? That is, when performing a sequence of N insertions starting from an empty tree, what is the average insertion cost?

(e) [4 pts] What is the cost of a search?

(f) [3 pts] The insertion cost in (d) is usually worse than the $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ amortized insertion cost for a buffer tree. What is the advantage of the buffered B-tree?

(g) [3 pts] When should a buffered B-tree be used instead of a regular B-tree?

Problem 2: Row-to-column Update on Matrices [30 pts]
Consider the following computation:

```
for i=1 to n do
  for j=1 to n do
    for k=1 to n do
      a[k,i] = SomeFunctionOf(a[i,j], a[k,i])
```

The high-level idea of this computation is to use every row to update every column in the increasing order for the row/column indices. This algorithm is widely used in many computational biology algorithms, using $O(n^3)$ operations.

Here we assume $n^2 > M$ (i.e., the matrix A does not fit in the cache directly). In this problem, you will design a cache-oblivious algorithm to implement such computation with cache complexity $O(n^2/B + n^3/BM)$.

Although most cache-oblivious algorithms can be highly-parallelized, you don't need to consider parallelism in this specific problem since otherwise the algorithm can be much more complicated.

(a) [10 pts] As a first step, figure out the cache-oblivious algorithm for each row-to-column update with cache complexity $O(n/B + n^2/BM)$. This may require the change of data layout.

(b) [20 pts] Give a cache-oblivious algorithm to this problem with cache complexity $O(n^2/B + n^3/BM)$.

Hint: since each row-to-column update are based on all previous updates, you need to guarantee that all values in the i-th row are up-to-date when they are used in the i-th row-to-column update. Similar algorithms for this step are discussed in the class, but you need to cooperate it into this algorithm correctly.

Problem 3: A Bonus problem [5pts]

Show that the cache-oblivious matrix-multiplication algorithm with cache complexity $O(n^3/B\sqrt{M})$ is optimal.