Algorithms in the Real World (15-853), Fall 10
Assignment #4 Due: October 26

Complete all problems.
You are not permitted to look at solutions of previous year assignments. You can work together in groups, but all solutions have to be written up individually. Please submit a hard copy during class.

Problem 1: Cover Tree Insertion (15pt)

Consider inserting elements into a Cover Tree.

(a) "Algorithm 2: Insert" from the reading on cover trees is written recursively. Give an example of a point set and insertion in which the algorithm will return multiple levels up the recursion before executing step 3(b).

(b) The paper on cover trees says that a batch construction is empirically superior to a sequence of single point insertions and references a longer version of the paper that describes it. Without looking at that paper (we'll have to trust you....and be suspicious if your algorithm is too close) please describe a batch algorithm—i.e., an algorithm that takes all the points at the start and processes them together to create a cover three. Please use pseudocode, with associated description if needed. The cleaner and more concise, but understandable, the better.

Problem 2: Collision Detection (10pt)
Write pseudocode for detecting collisions in a BVH. You can assume the BVH has binary forking, a function `overlap(N1,N2)` that returns true if the bounding volumes for two nodes of the tree (leaves or internal nodes) overlap and `collide(L1,L2)` that returns true if two leaves collide. This code should very short and familiar (hint).

Problem 3: Cache-aware list ranking (20pt)
The following asks you to fill in details for list ranking as described in the first locality lecture. The answers should be on the order of a few sentences, reducing the problem to scans and sorts.

(a) Complete the details for bridging out (slide 24). Specifically, suppose you have a total of N_R red and N_B blue list nodes stored contiguously in $N = N_R+N_B$ memory locations. The blue nodes contain pointers forming a single linked list. The red nodes contain arbitrary pointers. Describe how to copy the list of blue nodes to N_B contiguous memory locations (i.e., removing the red nodes). The solution should use $O(sort(N)) = O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ block transfers. (*The difficulty here is adjusting the pointers efficiently as the list is compressed.*) If convenient, you may assume pointers are relative to the start of the list in memory, i.e., 0 is the value of the pointer directed towards the node that appears first in memory. You may assume that nodes already contain a constant amount of extra space you may use to record any intermediate data. (*Hint: associate with each node x a value $BlueCount(x)$ counting the number of blue nodes appearing before x in memory.*)

(b) Describe how to perform the merge-in step of list ranking (slide 25). Specifically, suppose you have a total of N_R red and N_B blue list nodes stored contiguously in $N = N_R + N_B$ memory locations. All nodes contain integer-weighted pointers forming a single linked list. The successor of any red node is a blue node. The blue nodes already contain their rank in the list, where the rank is the total weight of the path from the start of the list to that node. Describe how to compute the rank for all red nodes in $O(sort(N))$ block transfers. You may assume that the first and last node in the linked list are blue and hence already have their ranks computed.

Problem 4: Buffered B-tree (20pt)

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) What is the height of a tree containing N objects?

(b) 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) 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) 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) What is the cost of a search?

(f) 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) When should a buffered B-tree be used instead of a regular B-tree?