Assignment 1
CSCI 6104: Algorithms for Massive Data Sets
Due Nov 30, 2011

Banner ID: ___________________ Name: ________________________

Question 1

Question 2

Question 3

Total
Question 1 (20 marks + 10 bonus marks) Recall the permutation problem: We are given an array $I$ of $N$ elements and an array $P$ storing each of the integers 1 through $N$ once. Our task is to store the elements of $I$ in an output array $O$, placing element $I[j]$ in position $O[P[j]]$, for all $1 \leq j \leq N$. The simplest linear-time algorithm that achieves this is

```
for j := 1 to N do 
    O[P[j]] := I[j]
```

In class, we argued that a well-crafted input instance for $B = 4$ and $M = 2B$ can force the LRU paging algorithm to perform one I/O per iteration of this algorithm, leading to an $\Omega(N)$ I/O bound. Now we want to be a bit more ambitious.

**Standard version (20 marks = 100%)**:
An online paging algorithm is an algorithm that has no knowledge of future memory accesses. Thus, when such an algorithm needs to evict a block from memory to make room for a block it needs to load into memory, it can use only its knowledge of the memory accesses it has seen so far to choose which block to evict. LRU is an online paging algorithm. Prove that for any online paging algorithm, any combination of block and memory sizes, and any $N$ sufficiently large, there exists an input instance of size $N$ that forces this paging algorithm to perform $\Omega(N)$ I/Os for the memory access sequence of the above permutation algorithm.

**Hint**: Argue that an adversary can choose the values in $P$ on the fly based on the blocks currently in memory so that the destination block of the current element $I[j]$ is not in cache, for $\Omega(N)$ values of $j$.

**Hard version (30 marks = 150%)**:
An offline paging algorithm is one that knows the entire memory access sequence from the beginning. Thus, when evicting a block from memory to make room for a block loaded into memory, it can choose to evict the block that is guaranteed to minimize the number of I/Os caused by all future memory accesses. Prove that for any offline paging algorithm, any combination of block and memory sizes, and any $N$ sufficiently large, there exists an input instance of size $N$ that forces this paging algorithm to perform $\Omega(N)$ I/Os for the memory accesses sequence of the above permutation algorithm.

**Hint**: Show that you can construct an input instance so that no element $I[j]$ is to be placed into the same block of $O$ as any of the preceding $M/B$ elements in $I$. Thus, if the blocks containing the previous $M/B$ elements are in memory, placing $I[j]$ into $O[P[j]]$ takes one I/O, and we need $\Omega(N)$ I/Os. Next argue that a paging strategy that results in a different set of blocks in memory by the time $I[j]$ is to be moved to $O[P[j]]$ cannot decrease the I/O bound of the algorithm.

Question 2 (20 marks) A basic and often useful data structure we did not discuss in class is a doubly linked list. In internal memory it uses $O(N)$ space and supports the following operations: Given a pointer to an element $x$ in the list, a new element can be inserted before or after $x$ in $O(1)$ time, and $x$ can be deleted in $O(1)$ time. Similarly, given a pointer to $x$, the $K$-element sublist starting with $x$ can be traversed in $O(K)$ time, for any $K \geq 1$. Here your goal is to construct an I/O-efficient doubly-linked list. The structure should occupy $O(N/B)$ blocks, should support insertions and deletions as above using $O(1)$ I/Os, and should allow the traversal of any $K$-element sublist using $O(1 + K/B)$ I/Os. Argue that the structure you develop achieves these space and I/O bounds.
Question 3 (20 marks) We discussed B-trees in class. An $N$-element B-tree occupies $O(N/B)$ disk blocks and supports insertions, deletions, and \textsc{Find} operations using $O(\log_B N)$ I/Os. Here your goal is to use the buffering idea discussed in class to obtain a structure with the same space requirements and the same cost for \textsc{Find} operations, but which uses only $O((1/\sqrt{B}) \log N)$ amortized I/Os per insertion or deletion. Argue that your structure achieves these space and I/O bounds. Then argue briefly how to extend your structure to reduce the cost of insertions and deletions further to $O((1/B^{1-\varepsilon}) \log N)$ amortized I/Os, for any $\varepsilon > 0$. 