

Due by the beginning of class, March 29.

1. Counting Sort can sort n integers between 1 and n in $O(n)$ time. Develop an external-memory equivalent to Counting Sort, that is, an algorithm that can sort n integers between 1 and n in $O(n/B)$ I/Os, or argue that this is impossible.
2. Given a set of n intervals $[a_1, b_1], [a_2, b_2], \dots, [a_n, b_n]$, the interval overlap problem is to report all pairs of intervals $[a_i, b_i]$ and $[a_j, b_j]$ such that $[a_i, b_i] \cap [a_j, b_j] \neq \emptyset$. Design an algorithm that solves the interval overlap problem in $O(\text{sort}(n) + t/B)$ I/Os, where t is the number of reported pairs. Can you make it Cache-Oblivious?
3. In class, we showed that doubling the space of a cache brought us to within a factor of 2 of the optimum for the paging problem. What is the general tradeoff between space increase and factor over optimal?
4. Bernard Chazelle described the following almost-trivial data structure, called a *window list*, to store a set I of real intervals. A window list is a partition of the real line into disjoint windows; each window w has a pointer to a linked list of the intervals $I_w \subseteq I$ that intersect w . (Any interval in I that intersects more than one window is stored multiple times.) Given a real value x , we can determine which intervals in S contain x by finding the window w containing x and then scanning through the corresponding list I_w . Such a query is called a *stabbing query*. For any window w , we can partition the intervals I_w into ending intervals E_w , which have at least one endpoint in w , and crossing intervals C_w , which don't. A window list is *efficient* if $|\#C_w - \#E_w| \leq 1$ for every window w .
 - (a) Prove that for any set of intervals, an efficient window list exists.
 - (b) Analyze window lists in internal memory. Given a set of N intervals, show that an efficient window list using $O(N)$ space can be constructed in $O(N \lg N)$ CPU time, so that any stabbing query can be answered in $O(\lg N + K)$ CPU time, where K is the number of output intervals.
 - (c) Now generalize window lists to external memory. Given a set of N intervals, show that an efficient window list can be constructed in $O(N/B \log_{M/B} N/B)$ I/Os and stored in $O(N/B)$ blocks, so that any stabbing query can be answered in $O(\log_B N/B + K/B)$ I/Os, where K is the number of output intervals. [Hint: Beware of nearly-empty blocks.]
 - (d) Show that the same efficient window list can also be used to answer intersection queries (Which intervals intersect this query interval?), containment queries (Which intervals contain this query interval?), and reverse containment queries (Which intervals are contained in this query interval?) in $O(\log_B N/B + K/B)$ I/Os.
5. In the early 1980s, Jon Bentley discovered the following technique, called the *logarithmic method* to modify internal-memory data structures to support efficient insertions.

The main idea is to partition the set of N elements into several subsets of different sizes and build a data structure for each subset. In Bentley's original scheme, the subset sizes are determined by writing N in binary: if the i th bit of N is 1, exactly one of the subsets has size 2^i , so there are at most $\lg N$ subsets altogether. Suppose the original structure uses $O(N)$ space to store N objects, that it can be built in $O(N \lg N)$ time, and that it supports queries in $O(\lg N)$ time.

- The new structure uses $\sum_{i=0}^{\lg N} O(2^i) = O(N)$ space.
- To answer a query, we simply query each of the component data structures in turn and combine the answers. The total query time is $\sum_{i=0}^{\lg N} O(\lg 2^i) = \sum_{i=0}^{\lg N} O(i) = O(\lg^2 n)$.
- Inserting a new element mirrors incrementing a binary counter – we find the first i such that there is no subset of size 2^i , and then combine the new element and the elements in all subsets smaller than 2^i into a new structure of size $1 + \sum_{j < i} 2^j = 2^i$. This insertion requires $O(i2^i)$ time. Since we need 2^i insertions to create a new structure of size 2^i , the amortized cost of a single insertion is $\sum_{i=0}^{\lg N} O(i) = O(\lg^2 N)$.

Generalize the logarithmic method to the external memory setting. Given an external data structure that uses $O(N)$ blocks, can be built in $O(N/B \log_{M/B} N/B)$ I/Os, and supports queries in $O(\log_B N/B)$ I/Os – for example, external window lists! – modify it to support efficient insertions. What are the amortized query and insertion times for your modified structure? [Hint: For each i , you should have at most one subset whose size is between B^{i-1} and B^i .]

6. The *packed memory array* 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 array maintains the invariant that the density of any node at depth d lies between α_d and β_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 α_d and β_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 α_h must be a positive constant to get constant-size gaps.]

7. 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 $O(N/B \log_{M/B} N/B)$ memory operations and $O(N \lg N)$ time. (These are the same time bounds as for sorting the array.)
8. Describe an efficient external topological sorting algorithm. Given a directed acyclic graph as input, your algorithm should label the vertices with integers from 1 to V so that for any edge $u \rightarrow w$, the label of u is smaller than the label of w . [Hint: Topological sorting is usually done in internal memory via depth-first search. Full credit will be given for an algorithm that runs in the same time as the external DFS algorithm of Buchsbaum et al., but this may not be the most efficient approach.]