PRINCETON UNIV. F'25 COS 521: ADVANCED ALGORITHM DESIGN

Lecture 22: External Memory Algorithms & Data Structures

Lecturer: *Huacheng Yu*

1 The I/O Model

Today we will describe a model called the "I/O" model, which attempts to capture complexity when the input is so large it cannot be loaded into fast random access memory, but has to be stored in slow disk. For this lecture, we will define the I/O model (External Memory Model) as follows.

1. There is a "main memory" of size M.
2. There is an "external memory" (disk) of unbounded size that is partitioned into blocks of size B.
3. A block can be read from external memory to main memory or written from main memory to external memory (an I/O) at unit cost.
4. All computation can only be done directly in the main memory.
5. **Goal:** Design an algorithm whose cost (number of I/Os it does) is minimized.

We should think of the regime where the input size $N \gg M \gg B \gg O(1)$.

Example 1. *Scanning an array of size N takes $O(N/B)$ I/Os.*

Really, we should think of a "linear-time" algorithm as linear in N/B.

1.1 Sorting in External Memory

Theorem 1. *There exists a sorting algorithm sorting N numbers stored in $\frac{N}{B}$ blocks that requires at most $O\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$ I/Os.*

One way to achieve this complexity is using a K-way merge sort where $K = \frac{M}{4B}$.

`K-WAY MERGE SORT(A)`

1. **Base Case:** we have K blocks of unsorted numbers. Then, we can
   - Read the K blocks into main memory.
   - Sort them.
   - Write them back to external memory.
2. **Recursive Case:**

- We split all of the blocks recursively into K arrays of $T = \frac{N}{KB}$ blocks each. We run the algorithm recursively on each array.
- We try to `MERGE` K sorted arrays of T blocks each into a sorted array of KT blocks.

Lemma 2. *The `MERGE` of K-`WAY MERGE SORT` can be done in $O(KT)$ I/Os.*

Proof. The procedure is as follows.
`MERGE`$(A_1, A_2, \ldots, A_K)$

1. Read the first block of each array into the main memory.
2. Sort these $KB = M/4$ numbers. Call this sorted array A. Set an output block C to empty (in main memory).
3. Repeatedly move the smallest number from A to C.
   - If C has B numbers, write C to external memory, empty C
   - If one of the K arrays has all its numbers moved from A to C, read its next block, and insert into A.

Notice that every block is read only once and every block is written once so the number of I/Os is at most $2KT$. □

Proof of Theorem 1. Call a level of merge sort the calls with a given size of array being merged. As in the proof of guarantees of standard merge sort, each level uses $O(N/B)$ I/Os to perform all merges and there are $O\left(\log_{M/B} \frac{N}{B}\right)$ merges, so we match the guarantee. □

1.2 External Memory Priority Queues

Recall what a priority queue is.

Definition 1. *A **priority queue** is a data structure that maintains a set S subject to the following operations.*

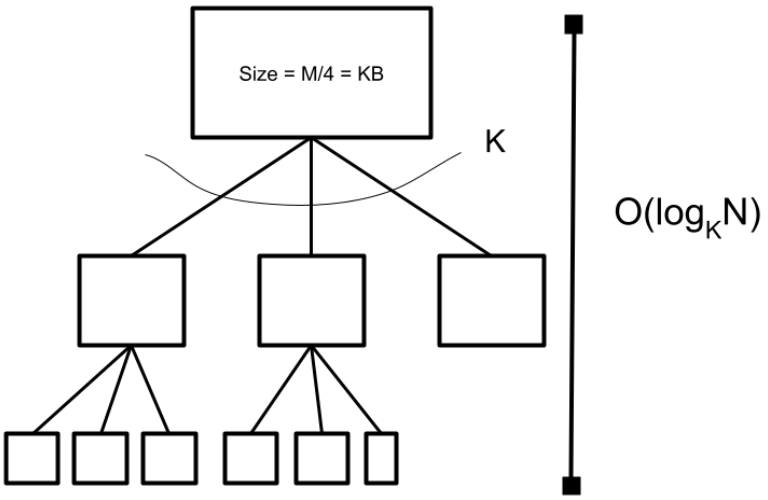
- *`INS`(x): insert x to S.*
- *`EXTRACT-MIN()`: delete and return $\min S$.*

In some settings, we have a decrease key operation that takes a key and changes its value.

Theorem 3. *There exists an external memory priority queue (heap) such that the amortized cost of operations is $O\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$.*

Note that this is $\ll 1$ when $B \gg \log N$. That means that only $\sim 1/B$ of the time do we incur an I/O charge on a query, which means that we get nearly the full batching parallelism of our model!

Proof. We will define a special heap structure is defined as follows. It is a complete tree (we fully fill in every level except the bottom, where the children at the left-most possible positions) with branching factor $K = \frac{M}{4B}$, as below.



Each node will be of size $M/4 = KB$. Note that the depth of the tree is $O(\log_K N)$. Finally, it should have the heap property.

Definition 2. *A tree has the heap property if all numbers stored in the parent node must be smaller than or equal to all unmbers in the children nodes.*

Then, as the root node has the smallest $M/4$ numbers, we will maintain the root node in the main memory. The rest of the tree is maintained in the external memory. Each node always stores between $\frac{KB}{2}$ and KB numbers in sorted order, except maybe the last (lowest and rightmost) leaf.

The heap will be associated with a buffer of size $M/4$ in the main memory, which stores "unprocessed" insertions. It makes sense to insert numbers in a huge batch, this saves us I/Os when considering amortization.

Insertion. First, we will implement insertion, which will be similar to a standard heap. `INS`(x)

1. Put x into the buffer.
    - If the buffer is full,
        (a) Sort ther numbers, creat a new leaf in the last level, move them to new leaf.
        (b) Restore the heap property bottom-up.

The way to restore the heap property as follows. First, load in the leaf and the parent into main memory. Sort them and and split into two even pieces (except perhaps at the lowest level, so that the parent has large enough size). Then, the heap property is restored with the leaf and parent, but we need to fix between parent and parent's parent. You can recursively apply this.

Then, if the buffer was full, this costs $O(K)$ I/Os per node x for $O(\log_{M/B} N)$ levels. This yields $O\left(\frac{M}{B} \log_{M/B} N\right)$ I/Os. Since each batch processes $O(M)$ insertions, the amortized time per insertion is as claimed.

For the other operation, notice that the minimum must be either be in the root node, or in the buffer. The issue is maintaining the invariants of the data structure after removing this minimum element.

`EXTRACT-MIN`()

1. Find the min across the buffer and the root and delete it.
   - If the root node has $< \frac{KB}{2}$ numbers
     (a) "refill" by moving up the smallest $KB/2$ numbers in its children
     (b) Recursively fix any children with $< \frac{KB}{2}$ numbers.
     (c) For all leaves with $< \frac{KB}{2}$ numbers, merge the last leaf into it and restore the heap property bottom up.

Note that since each child node is in sorted order, to refill an internal node only have to read one block from each node, which is $O(K)$ I/Os, or $O\left(\frac{1}{B}\right)$ I/Os per node. Again, we have $O(\log_{M/B} N)$ levels, so this is the maximum times a single element moves up. Hence, charging each time an element moves up to that element, we see that amortized cost per element in the internal nodes is $O\left(\frac{1}{B} \log_{M/B} N\right)$. For leaf nodes, one can similarly see that it costs $O\left(\frac{M}{B} \log_{M/B} N\right)$ I/Os to merge the last leaf into a leaf. Each last leaf is merged at most $2 = O(1)$ times before it is emptied out; we can charge these costs to the members of the last leaf that are being moved. Hence, we still get a cost per number in the leaf nodes of $O\left(\frac{1}{B} \log_{M/B} N\right)$ as well. □

However, the arguments presented only work if we assume the query sequences are long sequences of `INS` following by long sequences of `EXTRACT-MIN`. To give an argument that works for any interleaving, we have to use a **potential function argument**, which is a more sophisticated way to charge costs. The idea is that each operation does some "work" and increases some function which correlates with how much work needs to be done in a given operation. Using the right function, you can argue that either an operation is fast or it increases this potential function a lot. We will leave this as an exercise for an interested reader.