1 External Memory

Used Erik Demaine's 2003 notes.
Memory M, block size B, problem size N
basic operations:

- Scanning: $O(N/B)$
- reversing an array: $O(N/B)$

Linked list

- operations insert, delete, traverse
- insert and delete cost $O(1)$
- must traversing k items cost $O(k)$?
- keep segments of list in blocks
- keep each block half full
- no can travese $B/2$ items on one read
- so $O(K/B)$ to traverse K items
- on insert: if block overflows, split into two half-full blocks
- on delete:
  - if block less than half full, check next block
  - if it is more than half full, redistribute items
  - otherwise, merge items from both blocks, drop empty block
- Note: can also insert and delete while traversing at cost $1/B$ per operation
- so, e.g., can hold $O(1)$ "fingers" in list and insert/delete at fingers.

Search trees:

- binary tree cost $O(\log n)$ memory transfers
- wastes effort because only getting one useful item per block read
- Instead use $B + 1$=array tree; block has B splitters
- Now $O(\log_B N)$, much better
- Need to keep balanced while insert/delete:
  - require every block to be at least half full (so degree $\geq B/2$)
  - on insert, if block is full, split into two blocks and pass up a splitter to insert in parent

  - may overflow parent, forcing recursive splits
  - on delete, if block half empty, merge as for linked lists
  - may empty parent, force recursive merges
- optimal in comparison model:
  - Reading a block reveals position of query among B splitters
  - i.e. $\log B$ bits of information
  - Need $\log N$ bits.
  - so $\log N/\log B$ queries needed.

Sorting:

- Standard merge sort based on linear scans: $T(N) = 2T(N/2)+O(N/B) = O((N/B)\log N)$
- Can do better by using more memory
- M/B-way merge
- keep head block of each of M/B lists in memory
- keep emitting smallest item
- when emit B items, write a block
- when block empties, read next block of that list
- $T(M) = O(N/B) + (M/B)T(N/(M/B)) = O((N/B)\log_{M/B} N/B)$

Optimal for comparison sort:

- Assume each block starts and is kept sorted (only strengthens lower bound)
- loading a block reveals placement of B sorted items among M in memory
- $\binom{M+B}{B} \approx (eM/B)^B$ possibilities
- so $\log() \approx B \log M/B$ bits of info
- need $N \log N$ bits of info about sort,
- except in each of N/B blocks $B \log B$ bits are redundant (sorted blocks)
- total $N \log B$ redundant i.e. $N \log N/B$ needed.
- now divide by bits per block load

1.1 Buffer Trees

Mismatch:

- In memory binary search tree can sort in $O(n \log n)$ (optimal) using inserts and deletes
- But sorting with B-tree costs $N \log_B N$
- So inserts are individually optimal, but not "batch optimal"
- Basic problem: writing one item costs 1, but writing B items together only costs 1, i.e. $1/B$ per item
- Is there a data structure that gives "batch optimality"?
- Yes, but if inserts/queries are to happen in batches, sometimes you will have to wait for an answer until your batch is big enough

Basic idea:

- Focus on supporting N inserts and then doing inorder traversal
- Idea: keep buffer in memory, push into B-tree when we have a block's worth
- Problem: different items push into different children, no longer a block's worth going down
- Solution: keep a buffer at each internal node
- Still a problem: writing one item into the child buffer costs 1 instead of $1/B$
- Solution: make buffers **huge**, so most children get whole blocks written

Details:

- make buffer "as big as possible": size M
- increase tree degree to M/B
- basic operation: pushing full buffer down to children
  - bring buffer into memory, sort: cost M/B
  - 2-way merge with items arriving *sorted* from above: cost $1/B$ times number of items
  - write sorted contiguous elements to proper children
  - cost is at most 1 block per child (M/B) plus 1 block per B items.
  - since at least M items, account as $1/B$ per item
- on insert, put item in root buffer (in memory, so free)

- when a buffer is full, flush
- may fill child buffers. flush recursively
- when flush reaches leaves, store items using standard B-tree ops (split leaf nodes, possibly recursing splits)
  - cost of splits dominated by buffer flushing
  - how handle buffers when split? no problem: they are empty because we have just flushed to leaves.
- cost of flushes:
  - buffer flush costs $1/B$ per item
  - but each flushed item descends a level
  - total levels $\log_{M/B} N/B$
  - So cost per item is $(1/B) \log_{M/B} N/B$
  - So cost to insert N is optimal sort cost

Extensions

- Can add delete, range search by storing ops in buffers
- delete "triggers" when it catches up to target item
- range search outputs items when query reaches leaves

"Flush" operation

- empties *all* buffers so can directly query structure
- full buffers already accounted
- unfull buffers cost M/B per internal node
- but number of internal nodes is $N/(M/B)$
- so total cost N