External Memory

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
- now can traverse $B/2$ 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
  - now both blocks are at least half full
  - 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 $(B+1)$-ary tree; block has B splitters
- Require all leaves at same depth
- ensures balanced
- So depth $O(\log_B N)$, much better
- Need to keep balanced while insert/delete:

  - require every block to be at least half full (so degree $\geq B/2$)
  - also ensures
  - 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.

2011 Lecture 24 end. 2012 Lecture 26 end

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

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 overfull buffer down to children
  - invariant: arriving items are sorted
  - buffer had M
  - sort M items
  - merge with sorted incoming X in time $O(M/B + X/B)$
  - write sorted contiguous elements to proper children
  - check for child overflow, recurse

  - cost is at most 1 partial 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 fills, 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 leaves?
  - no problem: they are empty on root-lead path 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

"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 $(N/B)/(M/B)$ (N/B leaf blocks divided among M/B leaf-blocks per parent)
- so total cost N/B
- so can sort in $N/B \log_{M/B} N/B$

Extensions

- search: "insert" query, look for closest match as flushes down
- delete: insert "hole" that triggers when it catches up to target item
- delete-min: extra buffer of M minimum elements in memory. when empties, flush left path and extract leftmost items to refill
- range search: insert range, push to *all* matching children on flush
- all ops take $1/B \log_{M/B} N/B$ (plus range output)

Cache Oblivious Algorithms

In practice, don't want to consider B and M

- hard-coding them makes your algorithm non-portable
- and, with multi-level memory hierarchies, unclear where bottleneck is, i.e. which B and M you use
- without knowing B and M, how can we tune for them?
- surprisingly often, you can
- key: divide and conquer algorithms
- these tend to be sequentially optimal
- and rapidly shrink subproblems to where they fit on one page
- regardless of the size of that page

Assumption: optimal memory management

- so we can assume whatever page eviction policy works for us
- because we know e.g. LRU with double memory is 2-competitive against optimal management
- and we are ignoring constant factors

Example: scanning contiguous array

- N/B
- without knowing B!

Matrix Multiply

Adding matrices is easy

- $N \times N$ arrays
- treat as vectors and scan to add
- time N^2/B

Standard sequential algorithm

- $N \times N$ arrays
- N^3 time
- row major order

- so scanning a row is very cheap
- but scanning a column is super expensive
- unless $N \times N < M$, evict rows on every column
- so N^3 external memory accesses!
- can we do better?

store second matrix column major?

- One output now requires N/B reads
- So N^3/B, an improvement
- problem: what if next have to multiply in other order?
- need fast transpose operation!
- and turns out still not optimal—not leveraging large M

Divide and conquer

- mentally partition A and B into four blocks
- solve 8 matrix-multiply subproblems
- add up the pieces
- sequentially, $T(N) = 8T(N/2) + N^2 = O(N^3)$ (same as standard)
- external memory: $T(N) = 8T(N/2)+N^2/B = O(N^3/B)$ (same as column major)
- because at level i of recurrence do $2^i N^2/B$ fetches
- wait, at size $N = \sqrt{M}$, whole matrix fits in memory
- no more recursions. $T(c\sqrt{M}) = O(M/B)$
- this is at level i such that $N/2^i = \sqrt{M}$, ie $i = \log N/\sqrt{M}$
- so runtime $2^i N^2/B = N^3/B\sqrt{M}$

Binary Search Trees

B-trees are optimal, but you need to know B
Divide and conquer

- We generally search an array by binary search
- We query one value and cut problem in half
- bad in external memory, because only halve on one fetch
- B-tree is better, divides by B on one fetch
- but we don't know B

Van Emde Boas insight

- use new degree parameter d to avoid confusion
- problem of finding right child among d in page is same problem
- before, we set $d = B$ and ignored it because "in memory"
- now we don't know B but can still set a d
- recurrence: $T(N) = T(d) + T(n/d)$
- Balance: set $d = \sqrt{N}$
- $T(N) = 2T(\sqrt{N}) = 2^{\log \log N} = \log N$ (still sequentially optimal)
- Wait, when $N = B$ we get the whole array in memory and finish in $O(1)$
- so, stop at i such that $N^{(1/2)^i} = B$, ie $B^{2^i} = N$ so $2^i = (\log N)/\log B = \log_B N$
- so runtime $\log_B N$
- draw a picture
- see how you get a chain of $\log B$ items all on same page.
- yields speedup

Generalizations:

- Dynamic
- Buffer Trees

Linked Lists

Want $O(1)$ insert/delete but $O(1/B$ scan

- Cache aware, we used dynamic splitting/merging of contiguous "runs" to limit fragmentation
- How without B?
- Need to combat fragmentation without knowing whether it exists
- idea: defragment while scanning
- all scanned items become unfragmented
- use potential function (amount of fragmentation) to pay for defrag

Solution:

- on delete, just leave a hole
- on insert, append to end of external memory
- on scan, append all scanned items in order to end of memory

Analysis

- potential function: number of contiguous "runs" of elements in sequence on block
- equals number of "jumps" of pointers from one block to another
- suppose traverse K items in r runs
- the cost is at most $K/B + r$
- scanned items written to end
- eliminates all but first and last run:
- reduces runs by $r - 2$
- so amortized cost $O(K/B)$

Controlling size

- all operations including traversal increase size of data structure
- solution: after N/B work, scan whole list
- amortized cost N/B
- but now list is sorted/contiguous
- all previous blocks become free/irrelevant