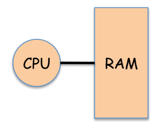
15-853: Algorithms in the Real World

Locality I: Cache-aware algorithms
- Introduction
- Sorting
- List ranking
- B-trees
- Buffer trees

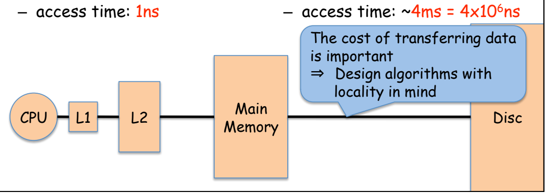
RAM Model

Standard theoretical model for analyzing algorithms:
- Infinite memory size
- Uniform access cost
- Evaluate an algorithm by the number of instructions executed



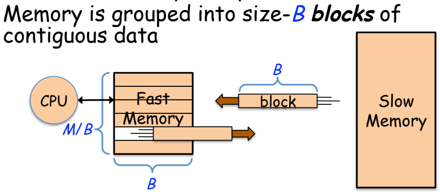
Real Machine Example

Pentium 4
- CPU
  - ~ 0.2ns / instruction
  - 8 Registers
- L1 cache
  - size: 8KB
  - line size: 64B
  - access time: 1ns
- L2 cache
  - size: 512KB
  - line size: 128B
  - access time: 8ns
- Memory
  - access time: 150ns
- Disc
  - access time: ~4ms = 4×10^6ns



I/O Model

Abstracts a single level of the memory hierarchy
- Fast memory (cache) of size *M*
- Accessing fast memory is free, but moving data from slow memory is expensive
- Memory is grouped into size-*B* ***blocks*** of contiguous data



- Cost: the number of ***block transfers*** (or **I/O**s) from slow memory to fast memory.

Notation Clarification

- *M*: the number of objects that fit in memory, and
- *B*: the number of objects that fit in a block
- So for word-size objects, and memory size 512KB, $M = 128{,}000$

Why a 2-Level Hierarchy?

- It's simpler than considering the multilevel hierarchy
- A single level may dominate the runtime of the application, so designing an algorithm for that level may be sufficient
- Considering a single level exposes the algorithmic difficulties — generalizing to a multilevel is often straightforward
- We'll see cache-oblivious algorithms later as a way of designing for multi-level hierarchies

What Improvement Do We Get?

Examples

- Adding all the elements in a size-*N* array (scanning)
- Sorting a size-*N* array
- Searching a size-*N* data set in a good data structure

Problem	RAM Algorithm	I/O Algorithm
Scanning	$\Theta(N)$	$\Theta(N/B)$
Sorting	$\Theta(N \log_2 N)$	$\Theta((N/B)\log_{M/B}(N/B))$
Searching	$\Theta(\log_2 N)$	$\Theta(\log_B N)$
Permuting	$\Theta(N)$	$\Theta(\min(N,\text{sort}(N)))$

- For 4-byte words on example Pentium 4
  - $B \approx 32$ in L2-cache, $B \approx 2000$ on disc
  - $\log_2 B \approx 5$ in L2-cache, $\log_2 B \approx 11$ on disc

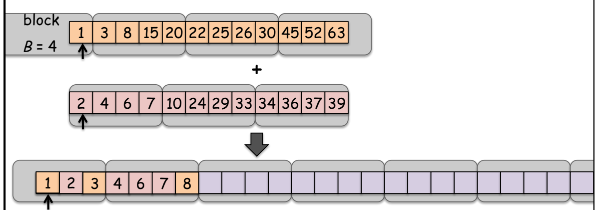
Sorting

Standard MergeSort algorithm:

- Split the array in half
- MergeSort each subarray
- Merge the sorted subarrays
- Number of computations is $O(N \log N)$ on an *N*-element array

How does the standard algorithm behave in the I/O model?

Merging

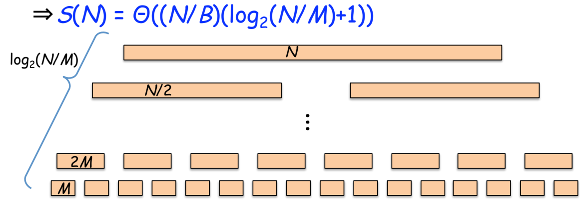


- A size-N array occupies at most $\lceil N/B \rceil+1$ blocks
- Each block is loaded once during merge, assuming memory size $M \geq 3B$

MergeSort Analysis

- Sorting in memory is free, so the base case is $S(M) = \Theta(M/B)$ to load a size-M array
- $S(N) = 2S(N/2) + \Theta(N/B)$

$\Rightarrow S(N) = \Theta((N/B)(\log_2(N/M)+1))$



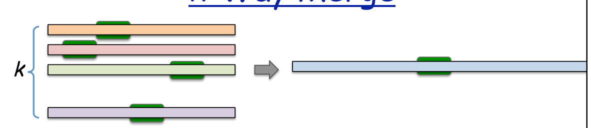
I/O Efficient MergeSort

- Instead of doing a 2-way merge, do a $\Theta(M/B)$-way merge

IOMergeSort:

- Split the array into $\Theta(M/B)$ subarrays
- IOMergeSort each subarray
- Perform a $\Theta(M/B)$-way merge to combine the subarrays

k-Way Merge

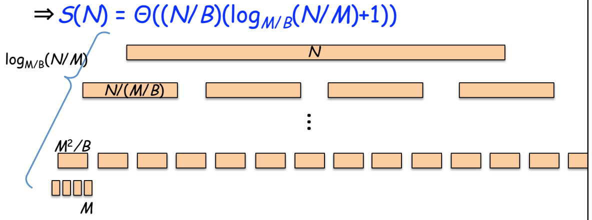


- Assuming $M/B \geq k+1$, one block from each array fits in memory
- Therefore, only load each block once
- Total cost is thus $\Theta(N/B)$

IOMergeSort Analysis

- Sorting in memory is free, so the base case is $S(M) = \Theta(M/B)$ to load a size-M array
- $S(N) = (M/B)\ S(NB/M) + \Theta(N/B)$

$\Rightarrow S(N) = \Theta((N/B)(\log_{M/B}(N/M)+1))$



MergeSort Comparison

Traditional MergeSort costs $\Theta((N/B)\log_2(N/B))$ I/Os on size-N array

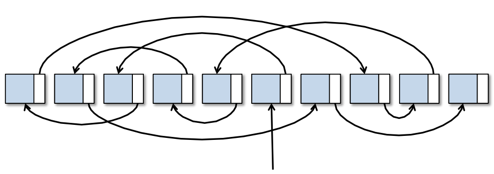
- IOMergeSort is I/O efficient, costing only $\Theta((N/B)\log_{M/B}(N/B))$
- The new algorithm saves $\Theta(\log_2(M/B))$ fraction of I/Os.

How significant is this savings?

- Consider L2 cache to main memory on Pentium 4
  - M = 128,000, B = 32, so $\log_2(M/B) \approx 12$x savings
- Main memory to disc has even bigger savings.

List Ranking

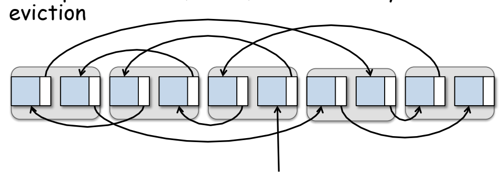
- Given a linked list, calculate the ***rank*** of (number of elements before) each element



- Trivial algorithm is $O(N)$ computation steps

List ranking in I/O model

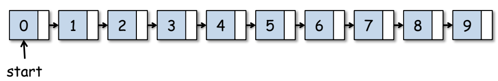
- Assume list is stored in ~N/B blocks!
- May jump in memory a lot.
- Example: $M/B = 3$, $B = 2$, least-recently-used eviction



- In general, each pointer can result in a new block transfer, for $O(N)$ I/Os

Why list ranking?

- Recovers locality in the list (can sort based on the ranking)

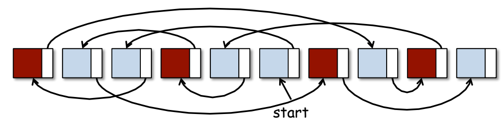


Generalizes to trees via Euler tours

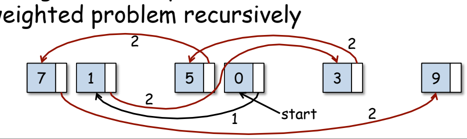
- Useful for various forest/tree algorithms like least common ancestors and tree contraction
- Also used in graph algorithms like minimum spanning tree and connected components

List ranking outline

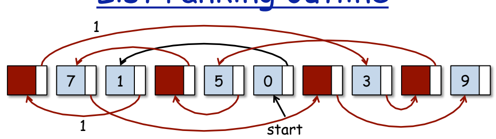
1. Produce an ***independent set*** of $\Theta(N)$ nodes (if a node is in the set, its successor is not)



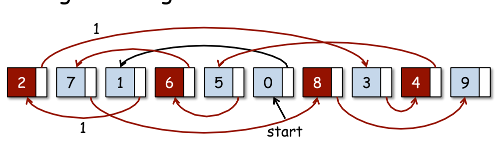
2. "Bridge out" independent set and solve weighted problem recursively



List ranking outline

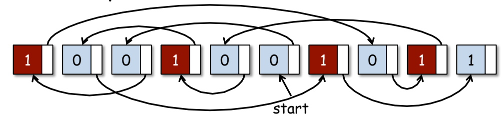


3. Merge in bridged-out nodes



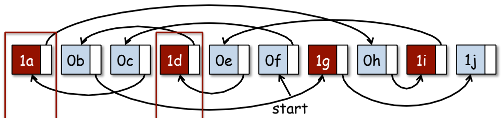
List ranking: 1) independent set

- Each node flips a coin {0,1}
- A node is in the independent set if it chooses 1 and its predecessor chooses 0



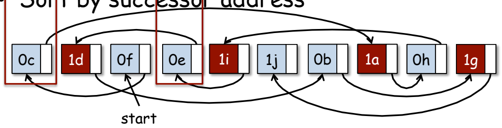
- Each node enters independent set with prob $\frac{1}{4}$, so expected set size is $\Theta(N)$.

List ranking: 1) independent set



Identifying independent-set nodes efficiently:

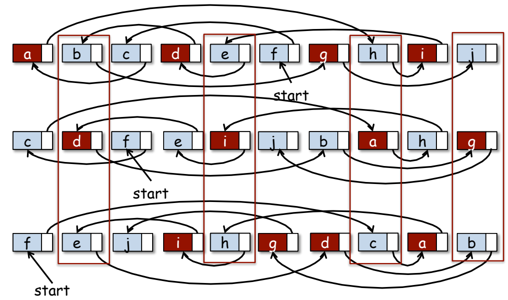
- Sort by successor address



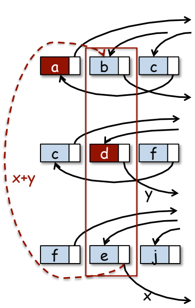
- After sort, requires O(scan(N))=O(N/B) block transfers

List ranking: 2) bridging out

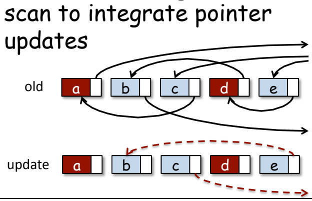
- Sort by successor address twice



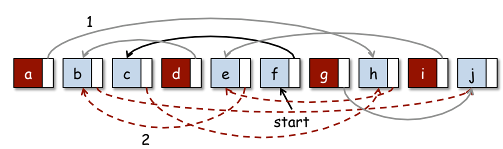
List ranking: 2) bridging out



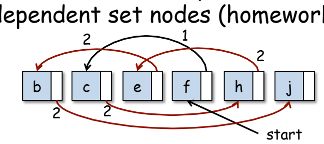
- If middle node is in independent set, "splice" it out
- Gives a list of new pointers
- Sort back to original order and scan to integrate pointer updates



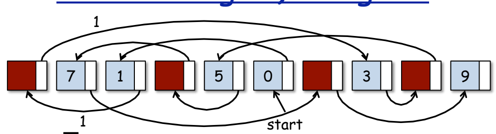
List ranking: 2) bridging out



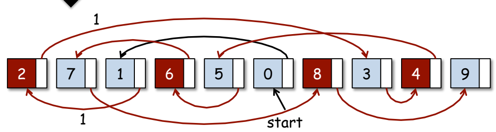
Scans and sorts to compress and remove independent set nodes (homework)



List ranking: 3) merge in



Uses sorts and scans (homework)



List ranking analysis

1. Produce an ***independent set*** of $\Theta(N)$ nodes (keep retrying until random set is good enough)
2. "Bridge out" and solve recursively
3. Merge-in bridged-out nodes

All steps use a constant number of sorts and scans, so expected cost is $O(\text{sort}(N)) = O((N/B) \log_{M/B} (N/B))$ I/Os at this level of recursion

Gives recurrence $R(N) = R(N/c) + O(\text{sort}(N))$
$= O(\text{sort}(N))$

B-Trees

A B-tree is a type of search tree ((a,b)-tree) designed for good memory performance

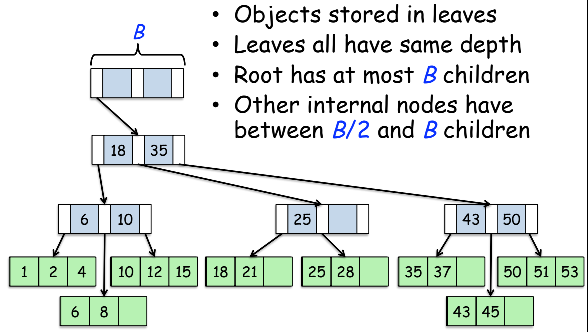
- Common approach for storing searchable, ordered data, e.g., databases, filesystems.

Operations

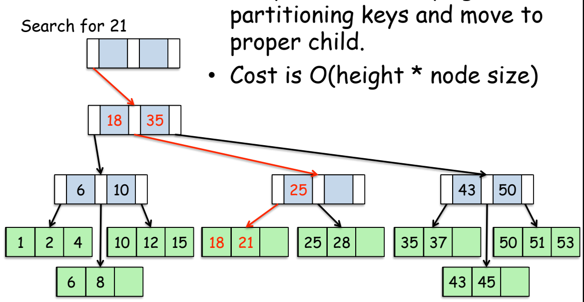
- Updates: Insert/Delete
- Queries
  - Search: is the element there
  - Successor/Predecessor: find the nearest key
  - Range query: return all objects with keys within a range
  - …

B-tree/(2,3)-tree

- Objects stored in leaves
- Leaves all have same depth
- Root has at most B children
- Other internal nodes have between $B/2$ and B children

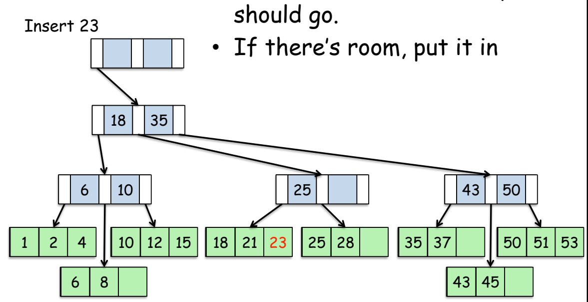


B-tree search



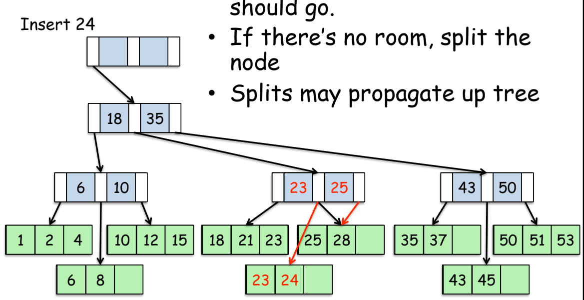
- Compare search key against partitioning keys and move to proper child.
- Cost is O(height * node size)

B-tree insert



- Search for where the key should go.
- If there's room, put it in

B-tree insert



- Search for where the key should go.
- If there's no room, split the node
- Splits may propagate up tree

B-tree inserts

- Splits divide the objects / child pointers as evenly as possible.
- If the root splits, add a new parent (this is when the height of the tree increases)
- Deletes are a bit more complicated, but similar — if a node drops below *B*/2 children, it is merged or rebalanced with some neighbors.

B-tree analysis

Search

- All nodes (except root) have at least $\Omega(B)$ children ⇒ height of tree is $O(\log_B N)$
- Each node fits in a block
- Total search cost is $O(\log_B N)$ block transfers.

B-tree analysis

Insert (and delete):

- Every split of a leaf results in an insert into a height-1 node.
- In general, a height-h split causes a height-$(h+1)$ insert.
- There must be $\Omega(B)$ inserts in a node before it splits again.
- An insert therefore pays for $\sum(1/B)^h = O(1/B)$ splits, each costing $O(1)$ block transfers.
- Searching and updating the keys along the root-to-leaf path dominates for $O(\log_B N)$ block transfers

Sorting with a search tree?

Consider the following RAM sort algorithm:

1. Build a balanced search tree
2. Repeatedly delete the minimum element from the tree

Runtime is $O(N \log N)$

Does this same algorithm work in the I/O model?

- Just using a B-tree is $O(N \log_B N)$ which is much worse than $O((N/B) \log_{M/B}(N/B))$

Buffer tree

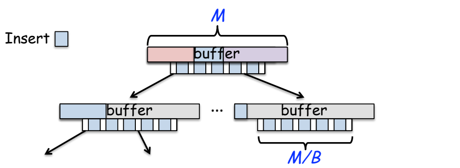
Somewhat like a B-tree:

- when nodes gain too many children, they split evenly, using a similar split method
- all leaves are at the same depth

Unlike a B-tree:

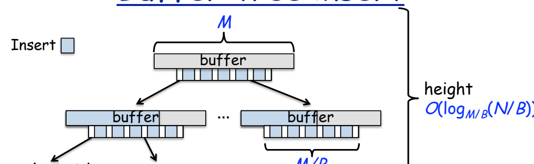
- queries are not answered online (they are reported in batches)
- internal nodes have $\Theta(M/B)$ children
- nodes have buffers of size $\Theta(M)$

Buffer-tree insert



- Start at root. Add item to end of buffer.
- If buffer is not full, done.
- Otherwise, partition the buffer and send elements down to children.

Buffer-tree insert



Analysis ideas:

- Inserting into a buffer costs $O(1+k/B)$ for k elements inserted
- On overflow, partitioning costs $O(M/B)$ to load entire buffer. Then M/B "scans" are performed, costing $O(\#\text{Arrays} + \text{total size}/B) = O(M/B + M/B)$
- Flushing buffer downwards therefore costs $O(M/B)$, moving $\Omega(M)$ elements, for a cost of $O(1/B)$ each per level
- An element may be moved down at each height, costing a total of $O((1/B)\ \text{height}) = O((1/B)\log_{M/B}(N/B))$ per element

I/O Priority Queue

Supporting Insert and Extract-Min (no Decrease-Key here)

- Keep a buffer of $\Theta(M)$ smallest elements in memory
- Use a buffer tree for remaining elements.
- While smallest-element buffer is too full, insert 1 (maximum) element into buffer tree
- If smallest-element buffer is empty, flush leftmost path in buffer tree and delete the leftmost leafs
- Total cost is $O((N/B)\log_{M/B}(N/B))$ for N ops.
- Yields optimal sort.

Buffer-tree variations

- To support deletions, updates, and other queries, insert records in the tree for each operation, associated with timestamps.
- As records with the same key collide, merge them as appropriate.

Examples of applications:

- DAG shortest paths
- Circuit evaluation
- Computational geometry applications