ASSORTED MINUTIAE

• Project 2 is due tonight
  • Make canvas group submissions
  • Load factor: total number of elements / current table size
  • Can select any load factor (but since we don’t measure memory consumption, lower may be better)
TODAY’S LECTURE

• Review of relevant info from Monday
• New, memory-conscious data structure
  • B-trees
HARDWARE CONSTRAINTS

- So far, we’ve taken for granted that memory access in the computer is constant and easily accessible
  - This isn’t always true!
  - At any given time, some memory might be cheaper and easier to access than others
  - Memory can’t always be accessed easily
  - Sometimes the OS lies, and says an object is “in memory” when it’s actually on the disk
HARDWARE CONSTRAINTS

- Back on 32-bit machines, each program had access to 4GB of memory
  - This isn’t feasible to provide!
  - Sometimes there isn’t enough available, and so memory that hasn’t been used in a while gets pushed to the disk
- Memory that is frequently accessed goes to the cache, which is even faster than RAM
The Memory Mountain

Pentium III Xeon
550 MHz
16 KB on-chip L1 d-cache
16 KB on-chip L1 i-cache
512 KB off-chip unified L2 cache

Ridges of Temporal Locality

Slopes of Spatial Locality

read throughput (MB/s)

stride (words)

working set size (bytes)
LOCALITY AND PAGES

• Secondly, the OS uses temporal locality,
  • Memory recently accessed is likely to be accessed again
  • Bring recently used data into faster memory
• Types of memory (by speed)
  • Register
  • L1, L2, L3
  • Memory
  • Disk
  • The interwebs (the cloud)
LOCALITY AND PAGES

• The OS is always processing this information and deciding which is the best
  • This is why arrays are faster in practice, they are always next to each other in memory
  • Each new node in a tree may not even be in the same page in memory!!

• Important to consider when designing and explaining design problems.
COST OF MEMORY ACCESSES

- Registers (128B): Instantaneous access
- L2 Cache (128KB): 0.5 nanoseconds
- L3 Cache (2MB): 7 nanoseconds
- Main Memory (32 GB): 100 nanoseconds
- Disk (TBs): 8,000,000 nanoseconds
LARGE AVL

• Suppose we are storing terabytes of data in an AVL tree
LARGE AVL

• Suppose we are storing terabytes of data in an AVL tree
  • Height is about 50
LARGE AVL

- Suppose we are storing terabytes of data in an AVL tree
  - Height is about 50
  - How many disk accesses will a find take?
LARGE AVL

• Suppose we are storing terabytes of data in an AVL tree
  • Height is about 50
  • How many disk accesses will a find take?
  • Between 0 and 50!
LARGE AVL

• Suppose we are storing terabytes of data in an AVL tree
  • Height is about 50
  • How many disk accesses will a find take?
  • Between 0 and 50!
  • This is the difference between nanoseconds and almost half a second!
Suppose we are storing terabytes of data in an AVL tree

- Height is about 50
- How many disk accesses will a find take?
- Between 0 and 50!
- This is the difference between nanoseconds and almost half a second!
- If lots data is stored on the disk, $O(\log n)$ finds don’t happen in practice
PROBLEMS

• Why is AVL so bad on disk?
PROBLEMS

• Why is AVL so bad on disk?
  • Each piece of data is its own node
PROBLEMS

• Why is AVL so bad on disk?
  • Each piece of data is its own node
  • Each call of `new` may not place objects next to each other
PROBLEMS

• Why is AVL so bad on disk?
  • Each piece of data is its own node
  • Each call of new may not place objects next to each other
  • Has large height, for the number of elements?
SOLUTIONS

• What changes might we want to make to an AVL to make it better for disk?
  • Still want to keep log n height
  • Allocate more objects closer together
  • Have a higher branching factor so that data you want is at a lower depth
  • Take advantage of page sizes
B-TREE

- Noded data structure
B-TREE

- Noded data structure
  - As an aside, what we will discuss in this course is called a B+ tree, which has slight differences if you go and look for resources online
B-TREE

• Noded data structure
  • Two types of nodes:
    • internal “signpost” nodes
    • leaf “data” nodes
  • Each node has a capacity
    • M for “signpost” nodes
    • L for “leaf/data” nodes
B-TREE

• Rules
  • Other than the root, internal nodes have between $M/2$ and $M$ children and leaves have between $L/2$ and $L$ data
B-TREE

• Rules
  
  • Other than the root, internal nodes have between M/2 and M children and leaves have between L/2 and L data
  
  • Elements in the leaves are stored in sorted order
B-TREE

• Rules
  • Other than the root, internal nodes have between M/2 and M children and leaves have between L/2 and L data
  • Elements in the leaves are stored in sorted order
  • The number of subtrees for a signpost is one more than the number of elements in the signpost
B-TREE

• Rules
  • Other than the root, internal nodes have between M/2 and M children and leaves have between L/2 and L data
  • Elements in the leaves are stored in sorted order
  • The number of subtrees for a signpost is one more than the number of elements in the signpost
  • The signpost has the smallest piece of data to the right of it – *all data is in a leaf*
B-TREE

- Example
B-TREE

• Find
B-TREE

• Find
  • Find the correct subnode at every signpost
    • $O(\log_2 M)$
B-TREE

• Find
  • Find the correct subnode at every signpost
    • $O(\log_2 M)$
  • Go through the depth of the tree
    • $O(\log_M N)$
B-TREE

• Find
  • Find the correct subnode at every signpost
    • $O(\log_2 M)$
  • Go through the depth of the tree
    • $O(\log_M N)$
  • Find the object in the leaf
    • $O(\log_2 L)$
B-TREE

- **Find**
  - Find the correct subnode at every signpost
    - $O(\log_2 M)$
  - Go through the depth of the tree
    - $O(\log_M N)$
  - Find the object in the leaf
    - $O(\log_2 L)$
  - Total find = $O(\log_2 L + \log_2 M \times \log_M N)$
B-TREE

• Insertion
  • Insert into the correct leaf (in sorted order)
B-TREE

• Insertion
  • Insert into the correct leaf (in sorted order)
  • If the leaf overflows
    • split into two
B-TREE

• Insertion
  • Insert into the correct leaf (in sorted order)
  • If the leaf overflows
    • split into two
    • attach new child to parent
    • add new key to parent
B-TREE

• Insertion
  • Insert into the correct leaf (in sorted order)
  • If the leaf overflows
    • split into two
    • attach new child to parent
    • add new key to parent
  • Recursively overflow as necessary
B-TREE

• Insertion
  • Insert into the correct leaf (in sorted order)
  • If the leaf overflows
    • split into two
    • attach new child to parent
    • add new key to parent
  • Recursively overflow as necessary
  • If the root overflows, make a new root
B-TREE

• Insertion
  • Find the correct leaf $O(\log_2 L + \log_2 M \cdot \log_M N)$
B-TREE

• Insertion
  • Find the correct leaf $O(\log_2 L + \log_2 M \times \log_M N)$
  • Insert in the leaf
B-TREE

• Insertion
  • Find the correct leaf $O(\log_2 L + \log_2 M \cdot \log_M N)$
  • Insert in the leaf $O(L)$
B-TREE

• Insertion
  • Find the correct leaf $O(\log_2 L + \log_2 M \cdot \log_M N)$
  • Insert in the leaf $O(L)$
  • Split the leaf $O(L)$
B-TREE

• Insertion
  • Find the correct leaf $O(\log_2 L + \log_2 M^*\log_M N)$
  • Insert in the leaf $O(L)$
  • Split the leaf $O(L)$
  • Split parents back to the root:
B-TREE

• Insertion
  • Find the correct leaf $O(\log_2 L + \log_2 M \cdot \log_M N)$
  • Insert in the leaf $O(L)$
  • Split the leaf $O(L)$
  • Split parents back to the root: $O(M \log_M n)$
B-TREE

• Insertion
  • Find the correct leaf $O(\log_2 L + \log_2 M^* \log_M N)$
  • Insert in the leaf $O(L)$
  • Split the leaf $O(L)$
  • Split parents back to the root: $O(M \log_M n)$
  • Total runtime = $O(L + M \log_M n)$
B-TREE

• Insertion
  • Find the correct leaf $O(\log_2 L + \log_2 M^*\log_M N)$
  • Insert in the leaf $O(L)$
  • Split the leaf $O(L)$
  • Split parents back to the root: $O(M \log_M n)$
  • Total runtime = $O(L + M \log_M n)$
  • Splitting is actually fairly uncommon
B-TREE

- Insertion
  - Find the correct leaf $O(\log_2 L + \log_2 M \cdot \log_M n)$
  - Insert in the leaf $O(L)$
  - Split the leaf $O(L)$
  - Split parents back to the root: $O(M \log_M n)$
  - Total runtime = $O(L + M \log_M n)$
  - Splitting is actually fairly uncommon
  - Care most about # of disc accesses
    - $\log_M n$
B-TREE

• Deletion
B-TREE

• Deletion
  • Remove the data from the correct leaf
B-TREE

• Deletion
  • Remove the data from the correct leaf
  • If the leaf has too few elements,
**B-TREE**

- **Deletion**
  - Remove the data from the correct leaf
  - If the leaf has too few elements,
    - Adopt one from a neighbor (if it doesn’t result in an underflow)
B-TREE

• Deletion
  • Remove the data from the correct leaf
  • If the leaf has too few elements,
    • Adopt one from a neighbor (if it doesn’t result in an underflow)
    • Otherwise, merge with the neighbor
B-TREE

- Deletion
  - Remove the data from the correct leaf
  - If the leaf has too few elements,
    - Adopt one from a neighbor (if it doesn’t result in an underflow)
    - Otherwise, merge with the neighbor
  - Recursively underflow up to root if necessary
B-TREE

• Deletion

  • Find the correct element: $O(\log_2 L + \log_2 M \cdot \log_M N)$
  • Remove from the leaf: $O(L)$
B-TREE

• Deletion
  • Find the correct element: \(O(\log_2 L + \log_2 M^{\log_M N})\)
  • Remove from the leaf: \(O(L)\)
  • Adopt/merge with neighbor: \(O(L)\)
B-TREE

• Deletion
  • Find the correct element: $O(\log_2 L + \log_2 M \cdot \log_M N)$
  • Remove from the leaf: $O(L)$
  • Adopt/merge with neighbor: $O(L)$
  • Merge back up to root: $O(M \log_m n)$
B-TREE

• Deletion
  • Find the correct element: $O(\log_2 L + \log_2 M \cdot \log_M N)$
  • Remove from the leaf: $O(L)$
  • Adopt/merge with neighbor: $O(L)$
  • Merge back up to root: $O(M \log_m n)$
  • Total time: $O(L + M \log_m n)$
B-TREE

• Practice tool here:
  • https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html
B-TREE

• Why bother with the B-tree?
B-TREE

• Why bother with the B-tree?
  • Many keys stored in each signpost
**B-TREE**

- Why bother with the B-tree?
  - Many keys stored in each signpost
  - Each can be brought up in one disk access
B-TREE

• Why bother with the B-tree?
  • Many keys stored in each signpost
  • Each can be brought up in one disk access
    • Binary search is fast because it’s all in memory
B-TREE

• Why bother with the B-tree?
  • Many keys stored in each signpost
  • Each can be brought up in one disk access
    • Binary search is fast because it’s all in memory
  • Internal nodes have only the keys (values waste space)
B-TREE

• Why bother with the B-tree?
  • Many keys stored in each signpost
  • Each can be brought up in one disk access
    • Binary search is fast because it’s all in memory
  • Internal nodes have only the keys (values waste space)
  • What values of M and L do we want?
B-TREE

• Why bother with the B-tree?
  • Many keys stored in each signpost
  • Each can be brought up in one disk access
    • Binary search is fast because it’s all in memory
  • Internal nodes have only the keys (values waste space)
  • What values of M and L do we want?
    • Want each node to be one page
B-TREE

• Choosing M and L
**B-TREE**

- **Choosing M and L**
  - Let a page be $p$ bytes
  - Keys are $k$ bytes
  - Pointers are $t$ bytes
  - Values are $v$ bytes

- $p = M*p + M-1*k$; $M = p+k / t+k$
- $L = (p-t) / (k+v)$