CSE 100:
B-TREE
Memory accesses

• Suppose you are accessing elements of an array:
  \[
  \text{if ( a[i] < a[j] ) \{ }\]
  \[
  \text{... or suppose you are dereferencing pointers:}
  \text{temp->next->next = elem->prev->prev;}
  \[
  \text{... or in general reading or writing the values of variables:}
  \text{disc = x*x / (4 * a * c);}
  \[
  \text{In simple algorithmic analysis, each of these variable accesses is assumed to have the same, constant, time cost}
  \[
  \text{However, in reality this assumption may not hold}
  \[
  \text{Accessing a variable may in fact have very different time costs, depending where in the memory hierarchy that variable happens to be stored}
The memory hierarchy

- In a typical computer, there is a lot of memory
- This memory is of different types: CPU registers, level 1 and level 2 cache, main memory (RAM), hard disk, etc.
- This memory is organized in a hierarchy
- As you move down the hierarchy, memory is
  - cheaper,
  - slower,
  - and there is more of it

- Differences in memory speeds can be very dramatic, and so it can be very important for algorithmic analysis to take memory speed into account
Typical memory hierarchy: a picture

AMOUNT OF STORAGE (approx!)

- hundreds of bytes
- hundreds of kilobytes
- hundreds of megabytes
- hundreds of gigabytes

TIME TO ACCESS (approx!)

- CPU registers: 1 nanosecond
- cache: 10 nanoseconds
- main memory: 100 nanoseconds
- disk: 10 milliseconds

1 sec = 1,000 millisec = 1,000,000 microsec = 1,000,000,000 nanosec
Consequences of the memory hierarchy

- Accessing a variable can be fast or slow, depending on various factors

- If a variable is in slow memory, accessing it will be slow

- However, when it is accessed, the operating system will typically move that variable to faster memory (“cache” or “buffer” it), along with some nearby variables
  - The idea is: if a variable is accessed once in a program, it (and nearby variables) is likely to be accessed again

- So it is possible for one access of a variable to be slow, and the next access to be faster; possibly orders of magnitude faster

  ```
  x = z[i]; // if z[i] is on disk this takes a long time
  z[i] = 3; // now z[i] is in cache, so this is very fast!
  z[i+1] = 9; // nearby variables also moved, so this is fast
  ```

- The biggest speed difference is between disk access and semiconductor memory access, so that’s what we will pay most attention to
Accessing data on disk

- Because disk accesses are many (thousands!) of times slower than semiconductor memory accesses, if a data structure is going to reside on disk, it is important that it can be used with very few disk accesses.

- The most commonly used data structure for large disk databases is a B-tree, which can be designed to use disk accesses very efficiently.

- Operations that we are interested in:
  - Insert
  - Delete
  - Find

All of them should be done with fewest disk accesses as possible.
B-trees and storage

- Each node in a B-tree fits into a block (i.e., if you get part of the node, you get it all)
- Search tree property
- Keys in each node are sorted
The goal of B-Trees

- Always at least half full
- Perfectly Balanced
- Few levels
Properties of an **m-order B trees**

1. The root has at least 2 sub-trees, unless it is a leaf.
2. All leaves are at the same level.
3. Each node (leaf as well as non leaf) holds k-1 keys where \( \lceil m/2 \rceil \leq k \leq m \) (non-root).
4. Each non-leaf node additionally holds k pointers to subtrees where \( \lceil m/2 \rceil \leq k \leq m \).
Order of B-trees

What order is this B-tree?

A. 2
B. 3
C. 4
D. 5
E. 6
What is the minimum number of keys each non-root node in this B-tree is allowed to store?

A. 0
B. 1
C. 2
D. 3
E. 4

How can we guarantee this?
Insert 21 into this B-tree. Then insert 50.
Insert 15 into this B-tree
Insertion and properties of B-trees

Insert 22 and 23
Insertion and properties of B-trees

Insert 16
Insertion and properties of B-trees

Insert 16, after
Insertion and properties of B-trees

Insert 62

Which key will be promoted up?
A. 61  B. 62  C. 68  D. 75  E. 80
Insert 62

Which key will be promoted up?
A. 13   B. 25   C. 60   D. 85   E. 68
Insertion and properties of B-trees

Insert 62
Insertion and properties of B-trees

B-trees grow up! (Which is why all their leaves are always at the same level)
B-Tree performance

- The time savings in a B-Tree comes from *efficiently reading lots of data from disk*
- When B-Trees are stored in memory they are typically comparable to other search trees
- When they have to access disk they are a big win
B-tree operations: find

• The find or lookup operation for a key in a B-tree is a straightforward generalization from binary search tree find
  • Read in to main memory from disk the key/data pairs in the root node
  • Do a search among these keys to find which child to branch to if desired key is not present (note the keys in an internal node are sorted, so you can use binary search there)
  • When you visit any other node, read in to main memory from disk the key/data pairs stored there and do a search among them for the desired key (binary search can work here also)
• So, the number of disk accesses required for a find is equal to the height of the tree, assuming the entire tree is stored on disk. But semiconductor memory accesses are required too... how do they compare to each other?
Analyzing find in B-trees

• Since the B-tree is perfectly height-balanced, the worst case time cost for find is $O(\log N)$

• Best case: If every internal node is completely filled
  • $N \leq m^{H-1}$
  • $H \geq \log_m(N+1)$

• Worst case ($d = \text{ceil}(m/2)$)
  • $N \geq 2 \cdot d^{H-1} - 1$
  • $H \leq \log_d((N+1)/2)$
B+-tree with parameter M, L

- There are several variants of the basic B tree idea
  - B+ trees is one of them
- In a B+ tree, data records are stored only in the leaves
  - A leaf always holds between \( \lceil L/2 \rceil \) and \( L \) data records (inclusive)
  - All the leaves are at the same level
- Internal nodes store key values which guide searching in the tree
  - An internal node always has between \( \lceil M/2 \rceil \) and \( M \) children (inclusive)
    - (except the root, which, if it is not a leaf, can have between 2 and \( M \) children)
  - An internal node holds one fewer keys than it has children:
    - the leftmost child has no key stored for it
    - every other child has a key stored which is equal to the smallest key in the subtree rooted at that child
A B+-tree with $M=4$, $L=3$

- Every internal node must have 2, 3, or 4 children.
- Every leaf is at the same level (level 2 in the tree shown), and every leaf must have 2 or 3 data records (keys only shown).
Designing a B+ tree

Suppose each key takes $K$ bytes, and each pointer to a child node takes $P$ bytes
- then each internal node must be able to hold $M^*P + (M-1)^*K$ bytes

Suppose each data record takes $R$ bytes
- then each leaf node must be able to hold $R^*L$ bytes

If a disk block contains $B$ bytes, then you should have $M$ and $L$ as large as possible and still satisfy these inequalities (assume $R<=B$; if not, you can use multiple blocks for each leaf):
- $B >= M^*P + (M-1)^*K$ and so: $M <= (B+K)/(P+K)$
- $B >= R^*L$, and so: $L <= B/R$
Designing a B+-tree, cont’d

- The branching factor in a B+-tree is at least \( \frac{M}{2} \) (except possibly at the root), and there are at least \( \frac{L}{2} \) records in each leaf node.
- So, if you are storing \( N \) data records, there will be at most \( \frac{2N}{L} \) leaf nodes.
- There are at most \( \frac{2N}{L} \) / \( \frac{M}{2} \) nodes at the level above the leaves; etc.
- Therefore: the height of the B+-tree with \( N \) data records is at most
  \[
  \log_{\frac{M}{2}} \left( \frac{2N}{L} \right) = \log_{\frac{M}{2}} N - \log_{\frac{M}{2}} L + \log_{\frac{M}{2}} 2
  \]
- In a typical application, with 32-bit int keys, 32-bit disk block pointers, 1024-byte disk blocks, and 256-byte data records, we would have:
  - \( M = 128 \), \( L = 4 \)
  - And the height of the B+-tree storing \( N \) data records would be at most \( \log_{64} N \).
- How does all this relate to the performance of operations on the B+-tree?
Analyzing find in B+-trees: more detail

- Assume a two-level memory structure: reading a disk block takes $T_d$ time units; accessing a variable in memory takes $T_m$ time units.

- The height of the tree is at most $\log_{M/2} N$; this is the maximum number of disk blocks that need to be accessed when doing a find.
  - so the time taken with disk accesses during a find is at most $T_d \cdot \log_{M/2} N$.

- The number of items in an internal node is at most $M$; assuming they are sorted, it takes at most $\log_2 M$ steps to search within a node, using binary search.
  - so the time taken doing search within internal nodes during a find is no more than $T_m \cdot \log_2 M \cdot \log_{M/2} N$.
  - Which about the same as $T_m \cdot \log_2 N$ since $\log_{M/2} N = \log_2 N / (\log_2 M/2)$.

- The number of items in a leaf node is at most $L$; assuming they are sorted, it takes at most $\log_2 L$ steps to search within a node, using binary search.
  - so the time taken searching within a leaf node during a find is at most $T_m \cdot \log_2 L$. 
Analyzing find in B+-trees: still more detail

• Putting all that together, the time to find a key in a B-tree with parameters M, L, and memory and disk access times Tm and Td is at most:
  \[ T_d \cdot \log_{M/2} N + T_m \cdot \log_2 M \cdot \log_{M/2} N + T_m \cdot \log_2 L \]

• This formula shows the breakdown of find time cost, distinguishing between time due to disk access (the first term) and time due to memory access (the last 2 terms).

• This more complete analysis can help understand issues of designing large database systems using B+-tree design.

• For example, from the formula we can see that disk accesses (the first term) will dominate the total runtime cost for large N if:
  \[ \frac{T_d}{T_m} \gg \log_2 M \]

• This will almost certainly be true, for typical inexpensive disks and memory and reasonable choices of M.

• And so the biggest performance improvements will come from faster disk systems, or by storing as many top levels of the tree as possible in memory instead of on disk.