CSE 530A

B+ Trees

Washington University
Fall 2013
B Trees

• A B tree is an ordered (non-binary) tree where the internal nodes can have a varying number of child nodes (within some range)
B Trees

- When a key is inserted into or removed from a node, the number of child nodes changes
- To maintain the proper range of number of keys, internal nodes are split or joined
- The lower and upper bounds on number of keys per node are usually fixed
  - The upper bound is most often twice the lower bound
    - At most half the space is wasted in the nodes
- Balance is kept by requiring that all leaf nodes have the same depth
B Trees

• B tree of order n (as defined by Knuth):
  1. Every node has at most n children
  2. Every node (except the root) has at least n/2 children
  3. The root has at least two children (if it is not a leaf)
  4. A non-leaf node with k children contains k – 1 keys
  5. All leaves are at the same level and have data
B Trees

- Each internal node's elements (a.k.a., the keys) divide up the subtrees
  - E.g., if an internal node has 3 children then it must have 2 keys, \( k_1 \) and \( k_2 \), where \( k_1 < k_2 \)
    - All elements in the left subtree must be < \( k_1 \)
    - All elements in the middle subtree must be between \( k_1 \) and \( k_2 \)
    - All elements in the right subtree must be > \( k_2 \)
B Trees

• Internal nodes
  – Non-leaf nodes and not the root node
  – Each has between the maximum and minimum number of children
  – If the minimum number of children is defined as half of the maximum number of children then
    • Each internal node is at least half full
    • Two half-full nodes can be joined to make a legal (full) node
    • A full node can be split into two legal nodes, if there is room to push the splitting key up into the parent

• Root node
  – Same upper limit but no lower limit

• Leaf nodes
  – Same restriction (lower and upper) on number of elements, but no child nodes
B+ Trees

- PostgreSQL (and other databases) actually use B+ Trees
  - B+ Trees are a variant of B Trees
    - All of the data are stored at the leaves
    - All of the keys appear at the leaves (in sorted order)
    - Some keys are duplicated in internal nodes
    - Leaf nodes are linked together (in order) to create a linked list
B+ Trees

1 2 3 4

3 4 3 <= x < 5

x < 3

3 5

x >= 5

5 6 7
Searching

• To find key $x$ in the tree
  – Start at the root node
  – Find the keys in the node where $m \leq x < n$
  – Follow pointer to child node
  – Repeat until leaf is reached
  – Find key $x$ in leaf

• Note that search always goes all the way to the leaves
Inserting

- Find leaf where new key $x$ belongs
- If leaf is not full then add $x$ to leaf
- If leaf is full then split leaf and push splitting key up to parent
  - If parent is full then split parent and push splitting key up
  - If we need to add a key to the root and the root is full then split the root and create a new parent as root
• Insert 28
  – Leaf is not full so we can just add it
• Insert 28
  – Leaf is not full so we can just add it
• Insert 70
  – Should go in leaf with (50, 55, 60, 65), but leaf is full
    • Need to split the leaf and then insert
Inserting

- Insert 70
  - Should go in leaf with (50, 55, 60, 65), but leaf is full
    - Need to split the leaf and then insert
    - 60 is pushed up to the parent
• Insert 95
  – Should go in leaf with (75, 80, 85, 90), but leaf is full
    • Need to split the leaf and then insert
    • 85 is pushed up to the parent, but parent is full
    • Need to split parent node then insert 85
• Insert 95
  – Should go in leaf with \((75, 80, 85, 90)\), but leaf is full
    • Need to split the leaf and then insert
    • 85 is pushed up to the parent, but parent is full
    • Need to split parent node then insert 85
Inserting

- Insert 95
  - Should go in leaf with (75, 80, 85, 90), but leaf is full
    - Need to split the leaf and then insert
    - 85 is pushed up to the parent, but parent is full
    - Need to split parent node then insert 85
    - 60 is then pushed up to the new root
• If inserting into a full node with a non-full sibling, we could shift keys from the full node to the non-full sibling
  – Requires modifying parent keys
• Insert 95
  – Should go in leaf with (75, 80, 85, 90), but leaf is full
    • Shift 75 to left sibling
      – Modify key in parent node
    • Insert 95 in newly opened node
Deleting

• Find leaf containing key \( x \)
• Delete \( x \) from leaf
  – If \( x \) was leftmost key then replace \( x \) in inner nodes with new leftmost key
• If leaf is not below lower limit
  – If \( x \) was a key in the parent then fix parent
  – Propagate change up to root if necessary
• If leaf is below lower limit then …
  – If sibling is above lower limit then shift key from sibling
    • Adjust keys in ancestors
  – If sibling is at lower limit then merge nodes
    • This removes a key from parent
    • Merge parent with sibling if below limit and propagate up
      – Potentially will remove the current root
• Delete 70
  – 70 is removed from leaf (60, 65, 60, 65)
  • No other change needed
Deleting

- Delete 70
  - 70 is removed from leaf (60, 65, 60, 65)
  - No other change needed
Deleting

- Delete 25
  - 25 is removed from leaf (25, 28, 30)
    - 25 is used in inner nodes so the inner nodes must be fixed
Deleting

- Delete 25
  - 25 is removed from leaf (25, 28, 30)
  - 25 is used in inner nodes so the inner nodes must be fixed
Deleting

- Delete 60
  - Remove from leaf (60, 65)
    - And replace 60 in inner nodes with new leftmost key
Deleting

- **Delete 60**
  - Leaf is now below lower limit
    - Can't shift key from sibling as sibling is at lower limit
    - Combine with leaf (75, 80)
• **Delete 60**
  - Extra key now needs to be removed from parent
Deleting

- Delete 60
  - Inner node is now below lower limit
  - Combine with sibling
    - This will eliminate root
Deleting

- Delete 60
  - Inner node is now below lower limit
  - Combine with sibling
    - This will eliminate root
B+ Trees

• For a $b$ order B+ tree (max of $b$ children per node)
  – Find, insert, and delete are all $O(\log_b n)$
  – Space is $O(n)$
  – Range queries can be done in $O(\log_b n + k)$ for a range of $k$
    • Range queries are queries that ask for all elements between two values
  – Elements in a range are already in order
Database Index

- An inner node with $n$ keys needs $n + 1$ pointers.
- A leaf node with $n$ keys also needs $n + 1$ pointers ($n$ pointers to the actual data and 1 pointer to its sibling).
- For a key size $k$ and a pointer size $p$:
  - A node holding $n$ keys needs $kn + p(n + 1) = (k + p)n + p$ bytes.
Database Index

- Key size varies depending on type
  - Assume 8 bytes per key (long int)
- Pointer size varies depending on architecture
  - Assume 8 bytes (64 bits)
- A node holding $n$ keys needs a minimum of $16n + 8$ bytes
Database Index

• In practice, disk fetches are much, much more expensive than RAM reads
  – Want to minimize disk fetches
  – No point in reading less than a page at a time from disk

• If we make our B+ tree nodes the size of a page then
  – Page size is typically 4 or 8 KB
  – For 4 KB and 8-byte keys and pointers, a node can hold a maximum of 255 keys
Database Index

- A tree just 3 levels deep can hold more than 16 million keys at the leaves.
- A tree just 4 levels deep can hold more than 4 billion keys at the leaves.
- Searching for a key in a 4 billion record table takes just 4 page fetches using an index.
  - A sequential scan of the table would take at least 16 million page fetches.