CMSC 420 - 0201 - Fall 2019
Lecture 09

B Trees
B-Trees
A Search Structure for External Memory

- Binary trees are the method of choice for ordered dictionaries stored in main memory
- On external memory systems (disk), entire blocks (pages) are accessed at once
- We would like each node of our tree to fill a block of external memory
- Multiway search tree - Fan-out depends on block size (e.g., 100)
B-Trees

- **B-Tree of order** $m$:
  - The root is either a leaf or has between 2 and $m$ children
  - Each non-root node has between $\lceil m/2 \rceil$ to $m$ children (and one fewer keys)
  - All leaves are at the same level of the tree

- **Example**: B-tree of order 5
B Trees

Height

**Theorem:** A B-tree of order \( m \) with \( n \) nodes has height at most \( (\log n)/\gamma \), where \( \gamma = \log \frac{m}{2} \).

**Proof:**
- With each level, fan-out is at least \( m/2 \).
- Number of nodes in a tree of height \( h \) is roughly \( n = \left(\frac{m}{2}\right)^h \).
- Solving for \( h \) as function of \( n \), implies \( h = \log n / \log \frac{m}{2} \).

- If \( m = 100 \), height \( \leq \log n / 5.6 \)
B-Trees

Node structure

- Because number of keys/children may vary, we allocate the maximum allowed storage for each node:

```cpp
const int M = ... // order of the B-tree

class BTreeNode {
    int nChildren; // number of children (from M/2 to M)
    BTreeNode child[M]; // children pointers
    Key key[M-1]; // keys
    Value value[M-1]; // values
}
```

- Setting M=3 yields a 2-3 tree, M=4 yields a 2-3-4 tree
B-Trees - Rebalancing operations

Key Rotation (Adoption)

- If node overflows (underflows), and sibling can take (give) a key, rotate the key through the parent out of (into) this node
- Example ($M = 5$): Node $p$ needs a key and sibling $q$ can give one
B-Trees - Rebalancing operations

Node splitting

- If a node has too many children \((m + 1)\), split the node in half and promote extra key to parent
- New nodes have \(m' = \left\lfloor \frac{m}{2} \right\rfloor\) and \(m'' = (m + 1) - m'\) children, respectively


**B-Trees – Rebalancing operations**

Node splitting

- Need to prove that new node sizes are valid
- **Lemma 1**: For all $m \geq 2$, $\left\lfloor \frac{m}{2} \right\rfloor \leq m', m'' \leq m$
- **Proof**: This is clearly true for $m'$. Suffices to consider just $m''$.
  - **Case 1 (m is even)**:
    - $\Rightarrow \left\lfloor \frac{m}{2} \right\rfloor = \frac{m}{2} \Rightarrow m'' = m + 1 - \frac{m}{2} = \frac{m}{2} + 1$
    - The lemma reduces to proving that $\frac{m}{2} \leq \frac{m}{2} + 1 \leq m$, which is clearly true for any $m \geq 2$.
  - **Case 2 (m is odd)**:
    - $\Rightarrow \left\lfloor \frac{m}{2} \right\rfloor = \frac{m+1}{2} \Rightarrow m'' = m + 1 - \frac{m+1}{2} = \frac{m+1}{2}$.
    - The lemma reduces to proving that $\frac{m+1}{2} \leq \frac{m+1}{2} \leq m$, which is clearly true for any $m \geq 1$. 


B-Trees - Rebalancing operations

Node merging

- If a node has too few children \(\left\lfloor \frac{m}{2} \right\rfloor - 1\), and both siblings have the minimum \(\left\lfloor \frac{m}{2} \right\rfloor\), merge node with sibling and demote one key from parent.

- The new node has size \(m''' = \left(\left\lfloor \frac{m}{2} \right\rfloor - 1\right) + \left\lfloor \frac{m}{2} \right\rfloor = 2 \left\lfloor \frac{m}{2} \right\rfloor - 1\).
B-Trees - Rebalancing operations

Node merging

- Need to prove that new node size is valid
- Lemma 2: For all \( m \geq 2 \), \( \left\lfloor \frac{m}{2} \right\rfloor \leq m''' \leq m 
- Proof:
  - Case 1 (\( m \) is even):
    - \( \Rightarrow \left\lfloor \frac{m}{2} \right\rfloor = \frac{m}{2} \Rightarrow m''' = 2 \left( \frac{m}{2} \right) - 1 = m - 1. \)
    - The lemma reduces to proving that \( \frac{m}{2} \leq m - 1 \leq m \), which is clearly true for any \( m \geq 2 \).
  - Case 2 (\( m \) is odd):
    - \( \Rightarrow \left\lfloor \frac{m}{2} \right\rfloor = \frac{m+1}{2} \Rightarrow m''' = 2 \left\lfloor \frac{m}{2} \right\rfloor - 1 = 2 \frac{m+1}{2} - 1 = m. \)
    - The lemma reduces to proving that \( \frac{m+1}{2} \leq m \leq m \), which is clearly true for any \( m \geq 1 \).
B-Trees - Dictionary operations

Find operation

- **Find(Key x):**
  - Finding a key is analogous to 2-3 trees
  - Descend the tree from the root
  - Let $a_1 < a_2 < \cdots < a_{j-1}$ be keys of current node (convention: $a_0 = -\infty, a_j = +\infty$)
  - Let $T_1, T_2, \ldots, T_j$ be children
  - Find $i$ such that $a_{i-1} < x \leq a_i$:
    - If $a_i = x$, found it
    - Else, if node is leaf, not found
    - Else, search $T_i$
B-Trees - Dictionary operations

Insertion operation

- **insert(Key x, Value v):**
  - Find the leaf node where x belongs
  - If x is already here - **Error**
  - Else, insert new (x, v) pair in this leaf
  - If node is **overfull**, attempt **key rotation** with siblings
  - If siblings are both full, **split** this node
  - One key is promoted to parent, rebalance the parent **recursively**

- **Note:** Key rotation and splitting are both options. Key rotation is preferred because it is **less costly** and **improves space utilization**
B-Trees - Dictionary operations

Insertion operation

- Example: insert(29) (M=5)
B-Trees - Dictionary operations

Deletion operation

- `delete(Key x, Value v)`:
  - Find the node containing `x`
  - If not found - Error
  - If not in leaf, find suitable replacement key from leaf level (largest in left subtree or smallest in right subtree), and copy it here
  - Delete the replacement key:
    - If node is underfull, attempt key rotation with siblings
    - If both siblings are minimal, merge this node with either sibling
    - One key is demoted from parent, rebalance the parent recursively
B-Trees - Dictionary operations

Deletion operations

- Example: delete(20) (M=5)
B-Trees - Dictionary operations

Deletion operations

- Example: delete(30) (M=5)
B-Trees - Variants

B+ Trees

- There are a number of variants of B-trees.
- **B+ trees**: A popular variant, used in disk storage
  - Key-value pairs are stored only at leaves
  - Internal nodes need only store keys, not values. (Saves space, bigger fan-out implies lower tree height, fewer disk accesses)
  - Leaf nodes do **not** need to waste space for child pointers
  - Each leaf node has a pointer to the next leaf node in the sequence. (Makes it easy to efficiently list all keys in a given range \([x_{min}, x_{max}]\). Find the leaf containing \(x_{min}\) and simply keep following next-leaf pointers until coming to \(x_{max}\).
Summary

- **B-Trees**
  - Multiway search trees - Very popular for disk storage
  - Fan-out m is controllable
  - Height is $O(\log n / \log m)$
  - Restructuring generalizes 2-3 tree:
    - Node rotation (adoption)
    - Split
    - Merge
  - Operations (insert, delete, find) run in time $O(\log n / \log m)$

- **B+ Trees** - A practical variant for disk storage