External Memory II

- Access Path Traversal
- Searching with Fast Updates

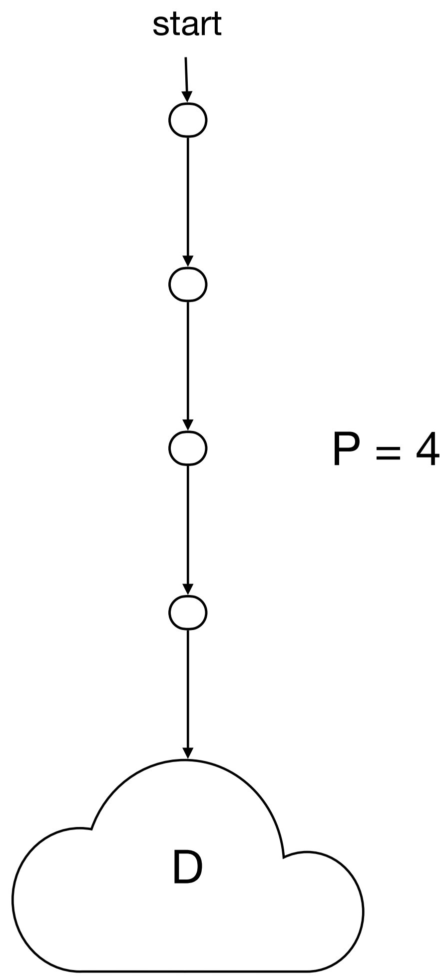
Philip Bille

External Memory II

- Access Path Traversal
- Searching with Fast Updates

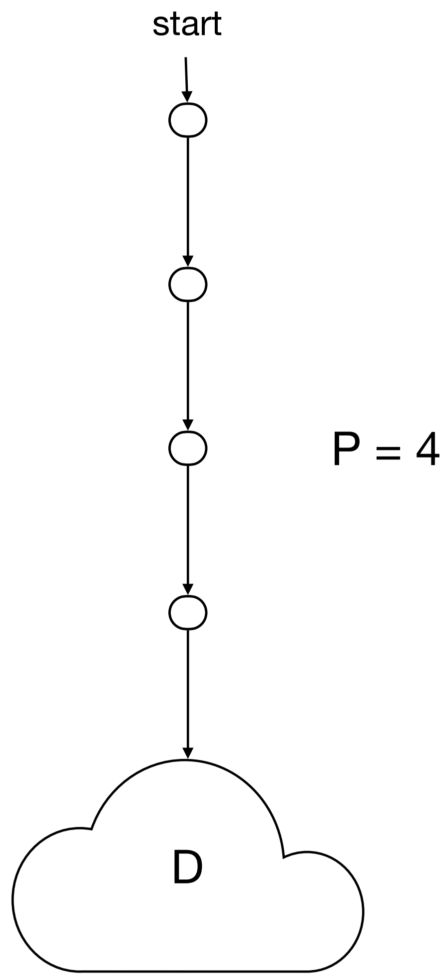
Access Path Traversal

- Access Path Traversal.
  - Data structure D stores a dynamic set of items.
  - Can only access D by following an access path of length $P \geq B$.
  - We want to support the following operations.
    - search(x): lookup x in D.
    - insert(x): insert x into D.
    - delete(x): remove x from D.
- Twist.
  - Each operation must start at the top of the access path.
  - How many I/Os for each operation? Ignore I/Os on D.



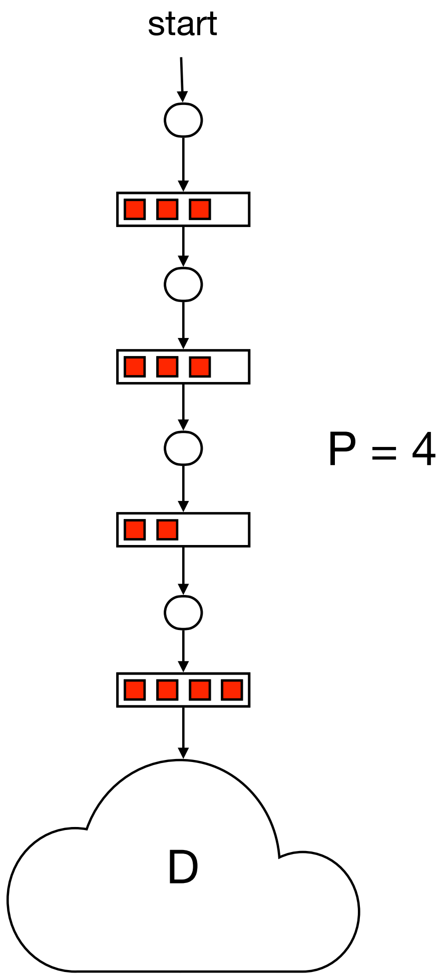
Access Path Traversal

- Solution 1. Direct traversal.
  - search(x): traverse path and lookup x in D.
  - insert(x): traverse path and insert x into D.
  - delete(x) traverse path and delete x from D.
- I/Os. O(P)



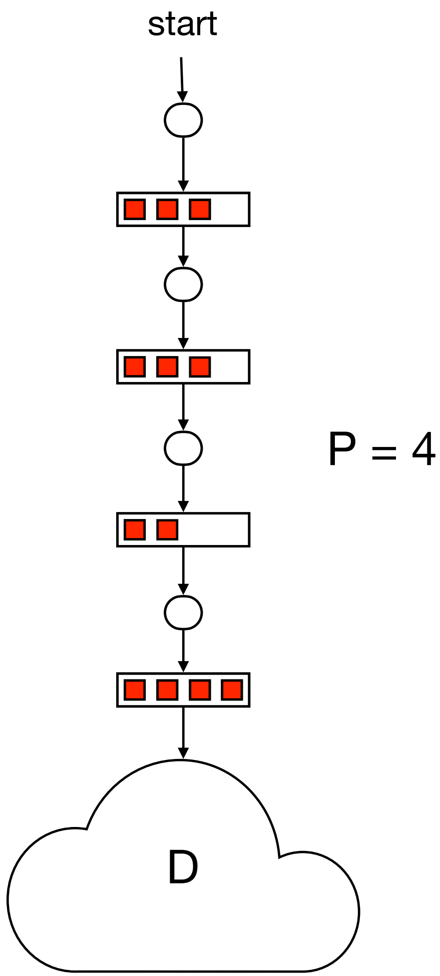
Access Path Traversal

- Solution 2. Buffered updates.
  - Add buffers of size Θ(B) to each edge stored in O(1) blocks.
  - Buffers store delayed updates to D. A delayed update is a message to insert or delete an item.



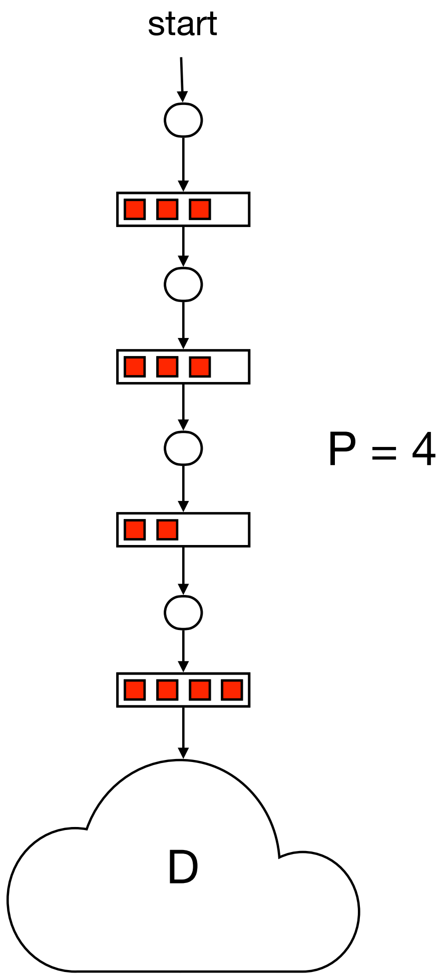
Access Path Traversal

- search(x).
  - Traverse path and check buffers for delayed updates on x (remove duplicate delayed updates on x).
  - Return x if we find a delayed insert on x on the path.
  - Otherwise, search x in D and return the result.
- I/Os. O(P)



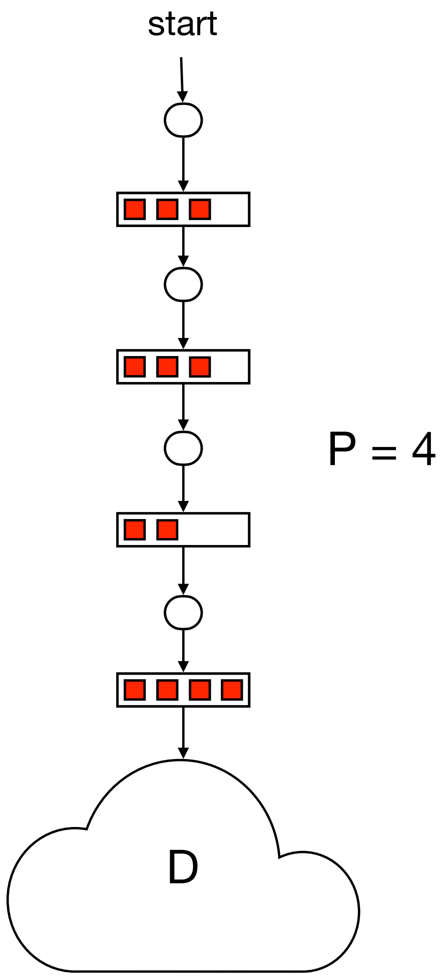
Access Path Traversal

- insert(x) or delete(x).
  - Insert delayed insert/delete into the first buffer on the path. If full, flush and recurse on the next node in the path.
  - If we flush the last buffer on the path, insert/delete items in D.
- I/O intuition.
  - Flush moves Θ(B) message together in O(1) I/Os.
  - A message moves at most P times.
  - ⇒ $O(P/B + 1) = O(P/B)$ amortized I/Os.



Access Path Traversal

- insert(x) or delete(x).
  - Insert delayed insert/delete into the first buffer on the path. If full, flush and recurse on the next node in the path.
  - If we flush the last buffer on the path, insert/delete items in D.
- I/Os. Amortized analysis via accounting method. Assign extra credits to items to pay for future operations. Credits must always be non-negative.
- Amortized cost is ≤ credits + actual cost of operation.
- Assign cP/B credits to each delayed update for appropriate constant c>1.
  - When a delayed update enters a buffer, we leave Θ(1/B) of the credits with the buffer.
  - When we flush a buffer, we use the Θ(B · 1/B) = Θ(1) credits to pay for the flush.
  - ⇒ We can pay for all flushes.
  - ⇒ Amortized I/Os is credits + actual cost = O(P/B + 1) = O(P/B).

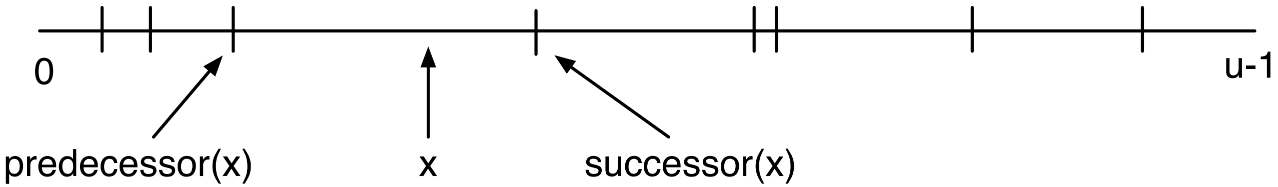


External Memory II

- Access Path Traversal
- Searching with Fast Updates

Searching

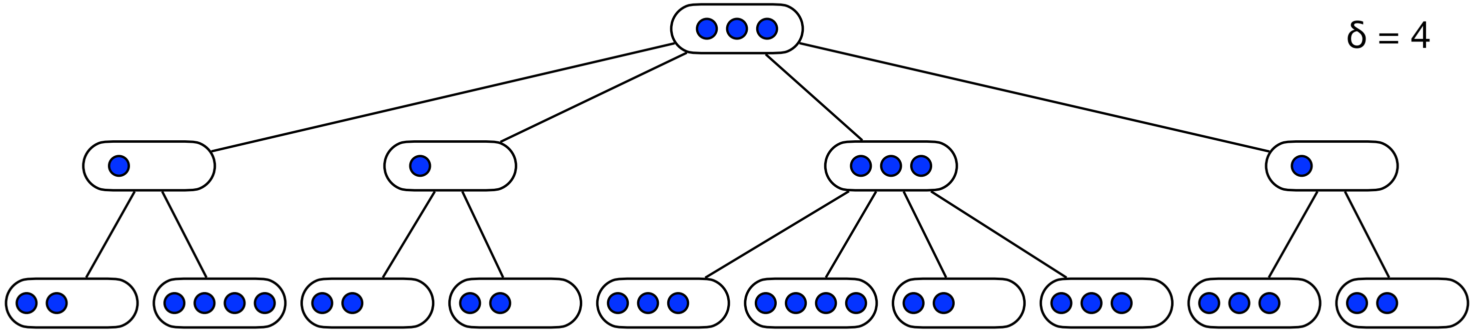
- Searching. Maintain a set $S \subseteq U = \{0, ..., u-1\}$ supporting
  - search(x): determine if $x \in S$
  - predecessor(x): return largest element in $S \leq x$.
  - successor(x): return smallest element in $S \geq x$.
  - insert(x): set $S = S \cup \{x\}$
  - delete(x): set $S = S - \{x\}$



Searching

- Applications.
  - Relational data bases.
  - File systems.

B-tree

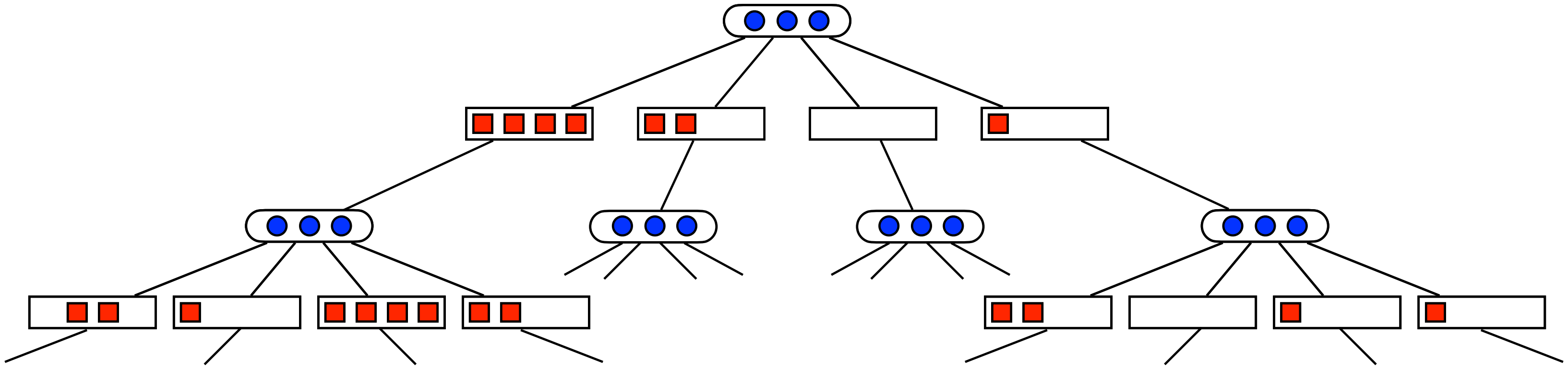


- B-tree of order $\delta = \Theta(B)$ with N keys.
  - Keys in leaves. Routing elements in internal nodes.
  - Degree between $\delta/2$ and δ.
  - Root degree between 2 and δ.
  - Leaves store between $\delta/2$ and δ keys.
  - All leaves have the same depth.
- Height. $\Theta(\log_\delta (N/B)) = \Theta(\log_B N)$
- Search and update. $O(\log_B N)$ I/Os.

B^{ε}-tree

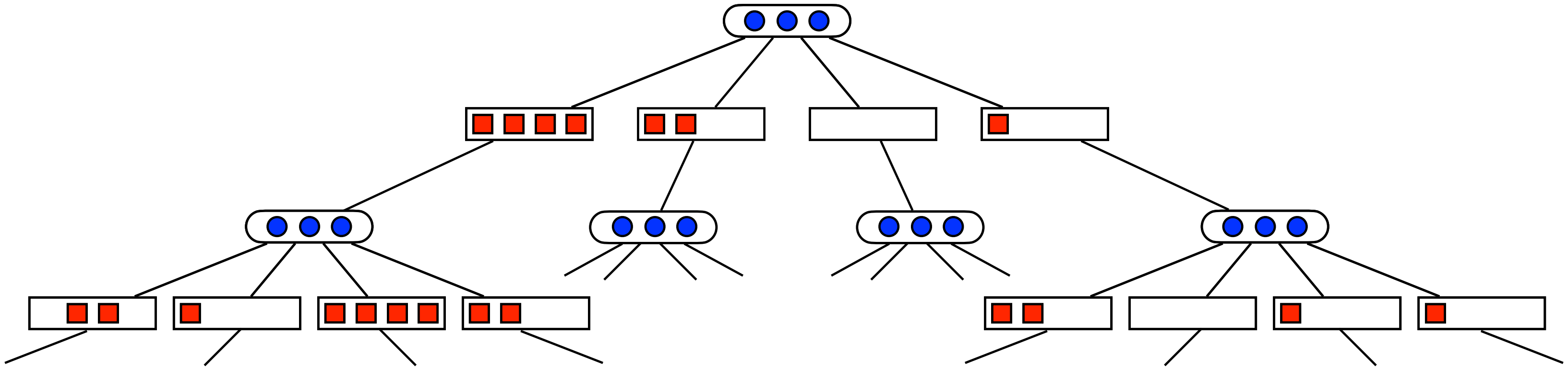
- Idea.
  - Speed up updates by buffering them at each node along the path to a leaf.
  - Move many updates together in each I/O.
  - Search (almost) as before.
  - $\varepsilon \in (0, 1]$ is a parameter.
- Solution in 2 steps.
  - Focus on $\sqrt{B}$-tree (ε = 1/2).
    - Searching in $O(\log_B N)$ I/Os.
    - Updates in $O((\log_B N)/\sqrt{B})$ amortized.
  - Generalize to any ε.

$\sqrt{B}$-tree



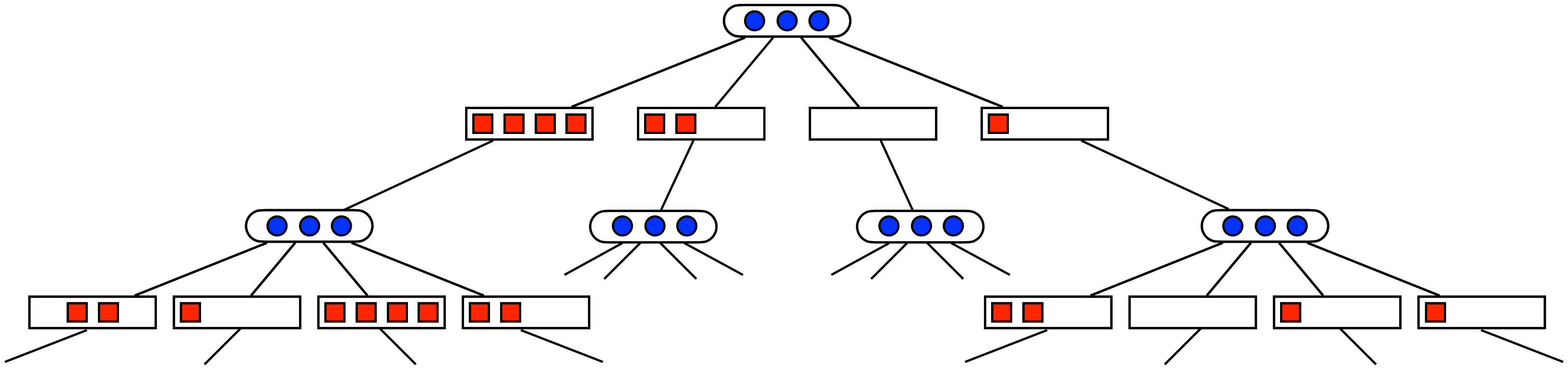
- $\sqrt{B}$-tree with N keys.
  - B-tree of degree $\Theta(\sqrt{B})$ with buffers of size $\Theta(\sqrt{B})$ at each edge.
  - Buffer stores delayed updates in subtree.
  - Nodes and child buffers stored together in O(1) blocks.
- Height. $\Theta(\log_{\sqrt{B}} N) = \Theta(\log_B N)$

$\sqrt{B}$-tree



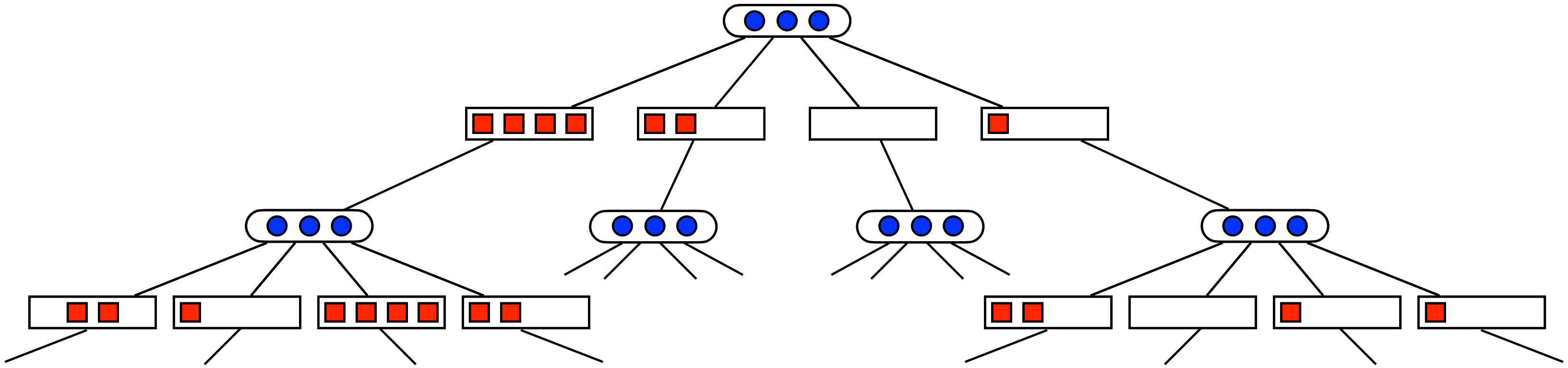
- search(x)
  - Find leaf using routing elements. Check buffers along path for delayed updates on x (remove duplicate delayed updates on x).
  - Return x if we find delayed insert on path.
  - Otherwise, return "yes" if x in leaf or "no" if not.
- I/Os. $O(\log_B N)$.

$\sqrt{B}$-tree



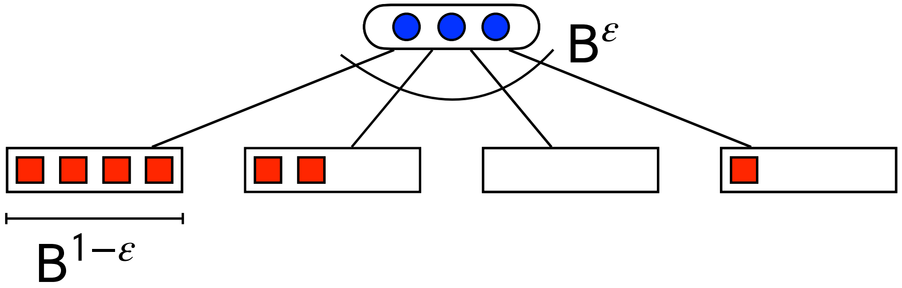
- insert(x) or delete(x)
  - Insert delayed insert/delete into first buffer on path. If full, flush and recurse on next node in path. If we fill leaf, rebalance tree as B-tree.
- I/O intuition.
  - A flush moves $\sqrt{B}$ messages together in O(1) I/Os.
  - A message moves at most $O(\log_B N)$ times.
  - $\Rightarrow O((\log_B N)/\sqrt{B})$ amortized I/Os.

$\sqrt{B}$-tree



- insert(x) or delete(x)
  - Insert delayed insert/delete into first buffer on path. If full, flush and recurse on next node in path. If we fill leaf, rebalance tree as B-tree.
- I/Os.
  - Assign $(c\log_B N)/\sqrt{B}$ credits to each update for appropriate constant $c > 1$.
  - Put $O(1/\sqrt{B})$ credits each node on the path to pay for flush and rebalancing.
  - When we flush a buffer, we use $O(\sqrt{B} \cdot 1/\sqrt{B}) = O(1)$ credits.
  - $\Rightarrow$ Amortized cost is $= O((\log_B N)/\sqrt{B})$

B^{ε}-tree



- B^{ε}-tree with N keys.
  - B-tree of degree $\Theta(B^{\varepsilon})$ with buffers of size $\Theta(B^{1-\varepsilon})$ at each edge.

- Searching. $O\left(\frac{\log_B N}{\varepsilon}\right)$ I/Os.
- Updates. $O\left(\frac{\log_B N}{\varepsilon B^{1-\varepsilon}}\right)$ I/Os.

B^{ε}-tree

	Search	Update
B-tree	$O(\log_B N)$	$O(\log_B N)$
$\sqrt{B}$-tree	$O(\log_B N)$	$O\left(\frac{\log_B N}{\sqrt{B}}\right)$
B^{ε}-tree	$O\left(\frac{\log_B N}{\varepsilon}\right)$	$O\left(\frac{\log_B N}{\varepsilon B^{1-\varepsilon}}\right)$

External Memory II

- Access Path Traversal
- Searching with Fast Updates