Performance Guarantees for B-trees with Different-Size Atomic Keys

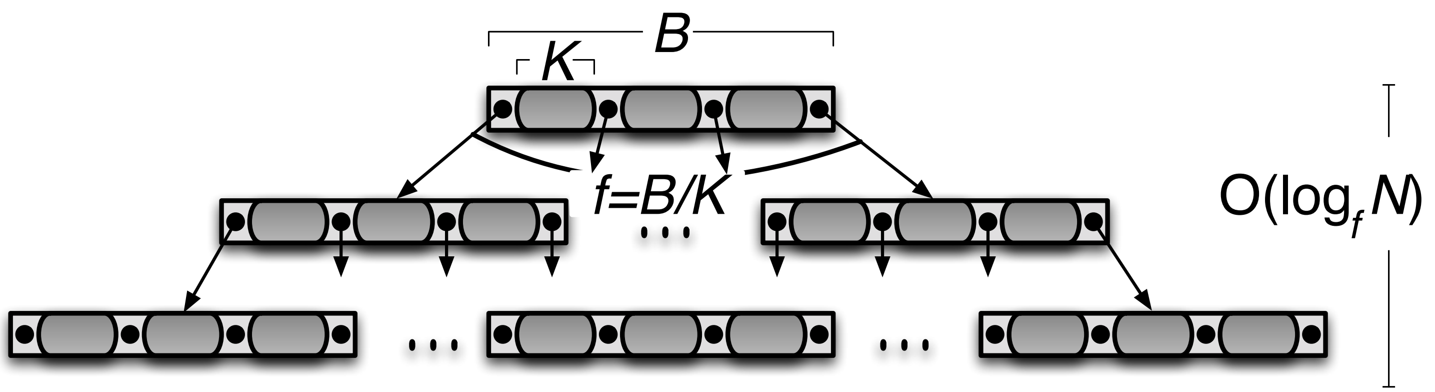
Michael A. Bender
Stony Brook
Tokutek, Inc.

Haodong Hu
Microsoft

Bradley C. Kuszmaul
MIT
Tokutek, Inc

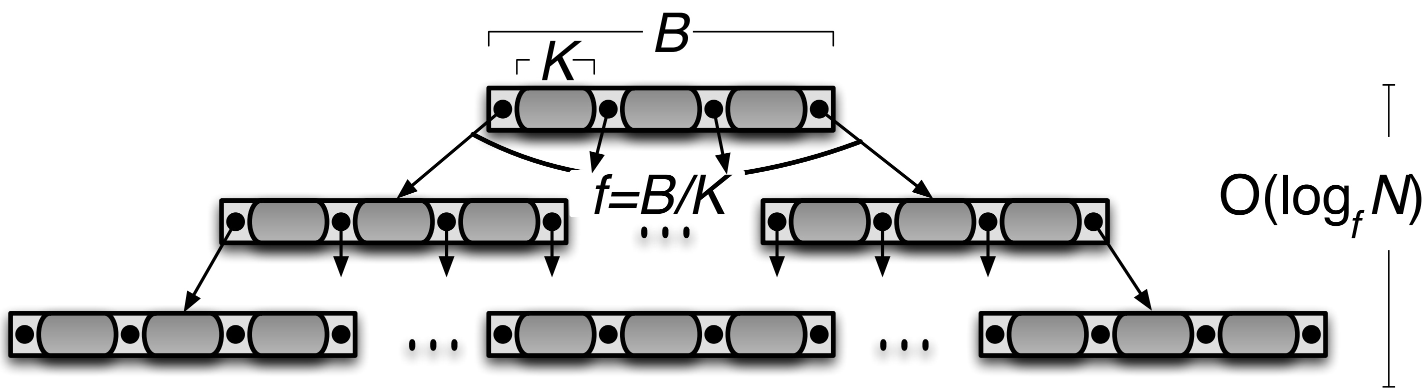
This talk

In B-trees in textbooks, all keys have the same size.



This talk

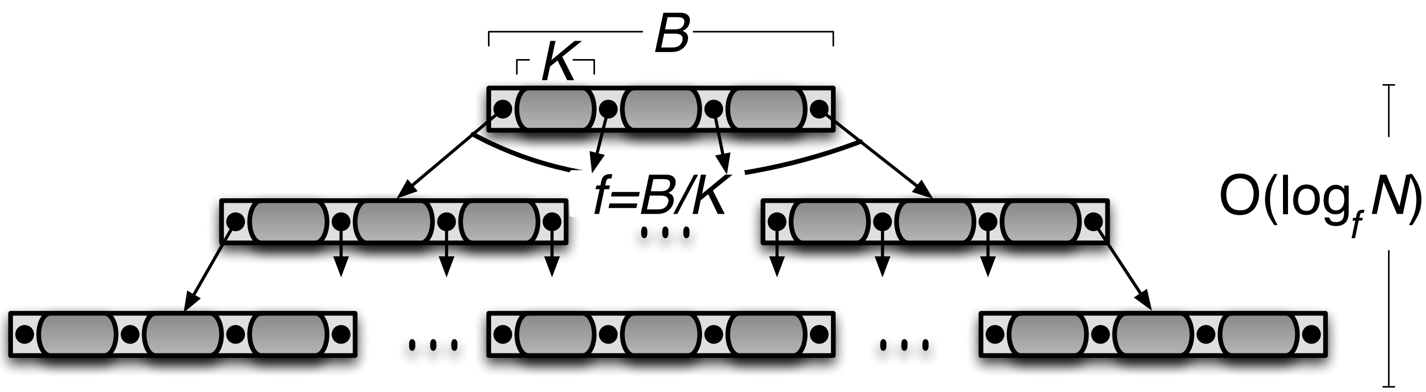
In B-trees in textbooks, all keys have the same size.

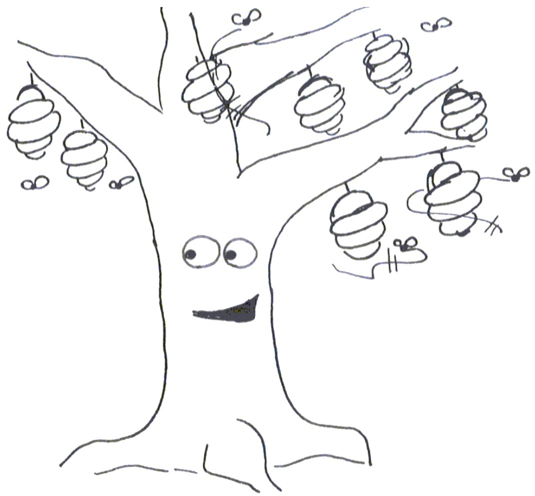


Production B-trees support different-size keys, but with no nontrivial performance guarantees.

This talk

In B-trees in textbooks, all keys have the same size.





Production B-trees support different-size keys, but with no nontrivial performance guarantees.

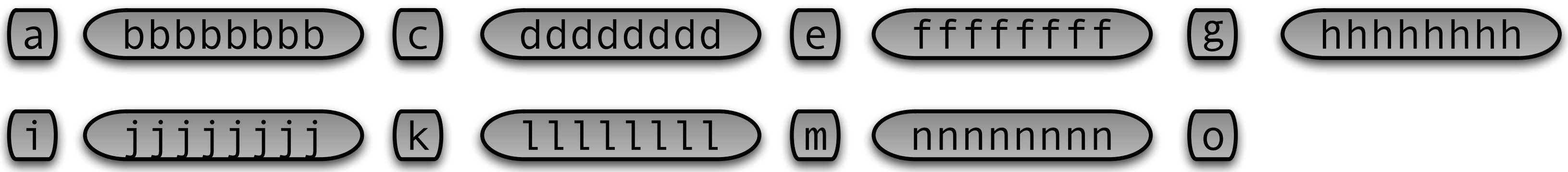
This talk: give provably good guarantees in an only slightly modified B-tree.

Example Showing Problem

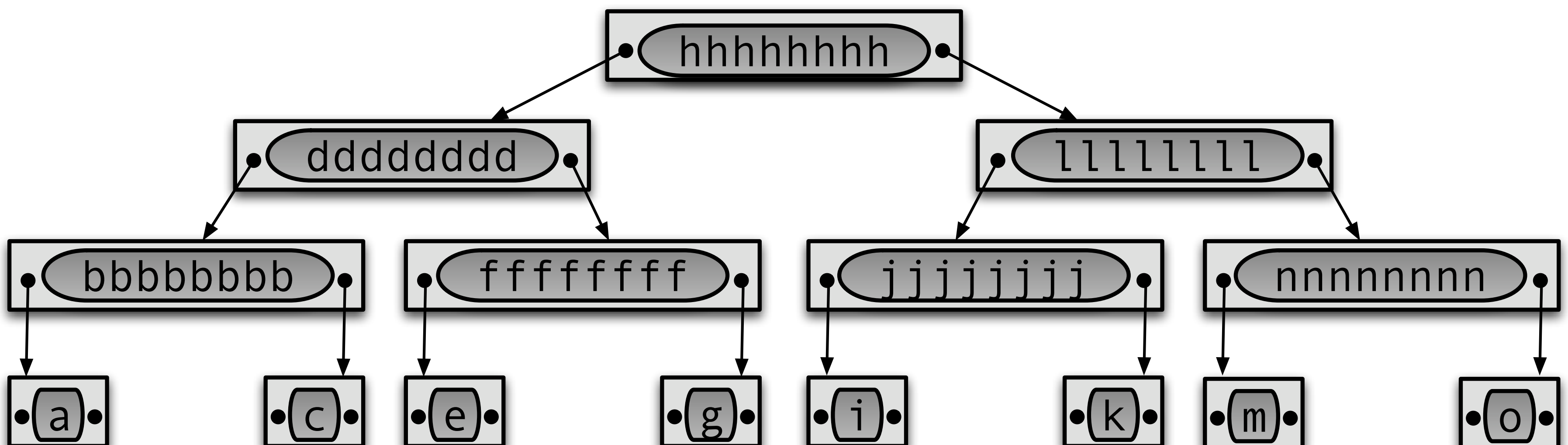
a bbbbbbbb c dddddddd e ffffffff g hhhhhhhh

i jjjjjjjj k llllllll m nnnnnnnn o

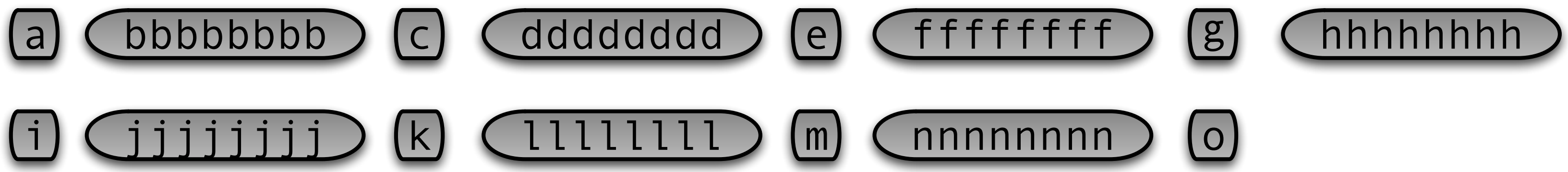
Example Showing Problem



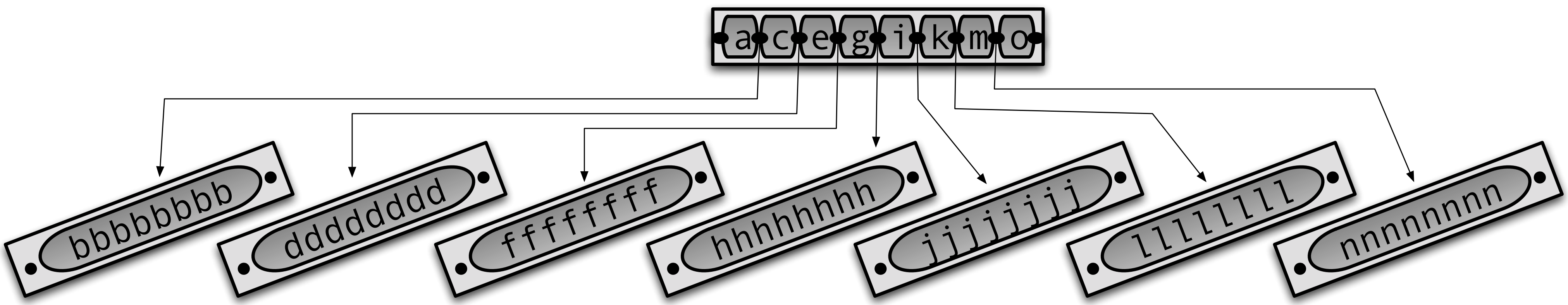
When the length-8 keys are pivots (and the block size is 8), the tree height is 4:



Example Showing Problem



When the length-1 keys are pivots (and the block size is 8), the tree height is 2:



Choice of pivot affects the B-tree performance.

Keys are Atomic, Not Strings

Cannot compare first byte of bbbbbbbb **with** c

Keys are Atomic, Not Strings

Cannot compare first byte of bbbbbbbb with c.

Only the comparison function understands the keys.

Keys are opaque. Need to send entire key to comparison function and store entire key in node.

Keys are Atomic, Not Strings

Cannot compare first byte of bbbbbbbb **with** c**.**

Only the comparison function understands the keys.

Keys are opaque. Need to send entire key to comparison function and store entire key in node.

Keys are Atomic, Not Strings

Cannot compare first byte of bbbbbbbb with c.

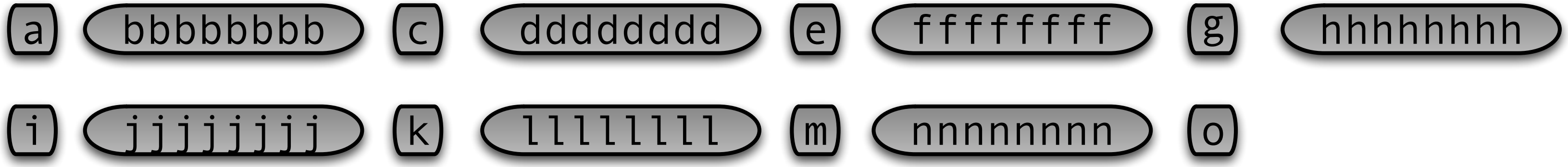
Only the comparison function understands the keys.

Keys are opaque. Need to send entire key to comparison function and store entire key in node.

String B-tree techniques don't work [Ferragina, Grossi 98].

Front compression (prefix compression) doesn't work [Bayer, Unterauer 77] [Wagner 73].

Terminology

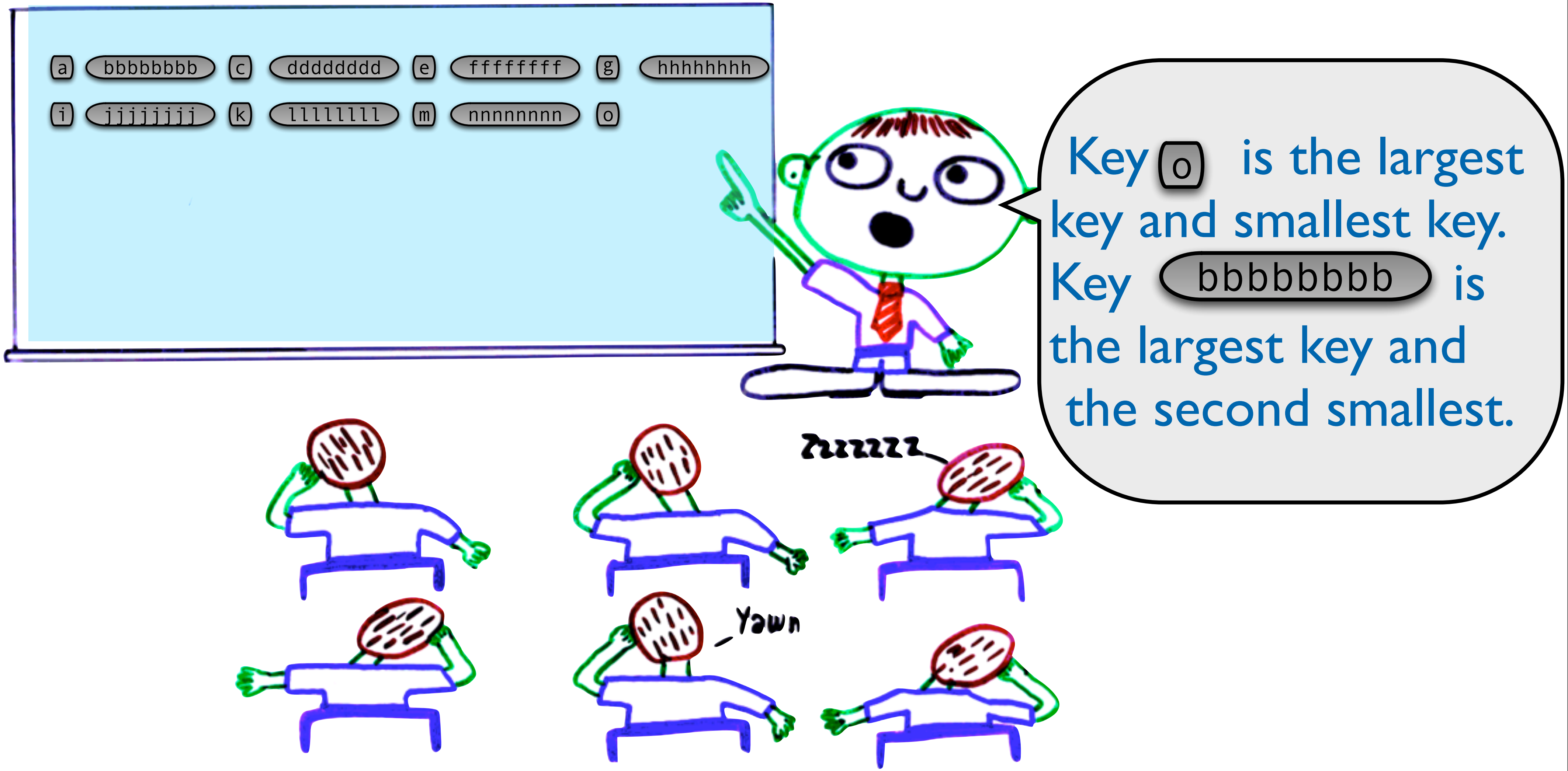


earliest vs. *latest*: order determined by comparison function.

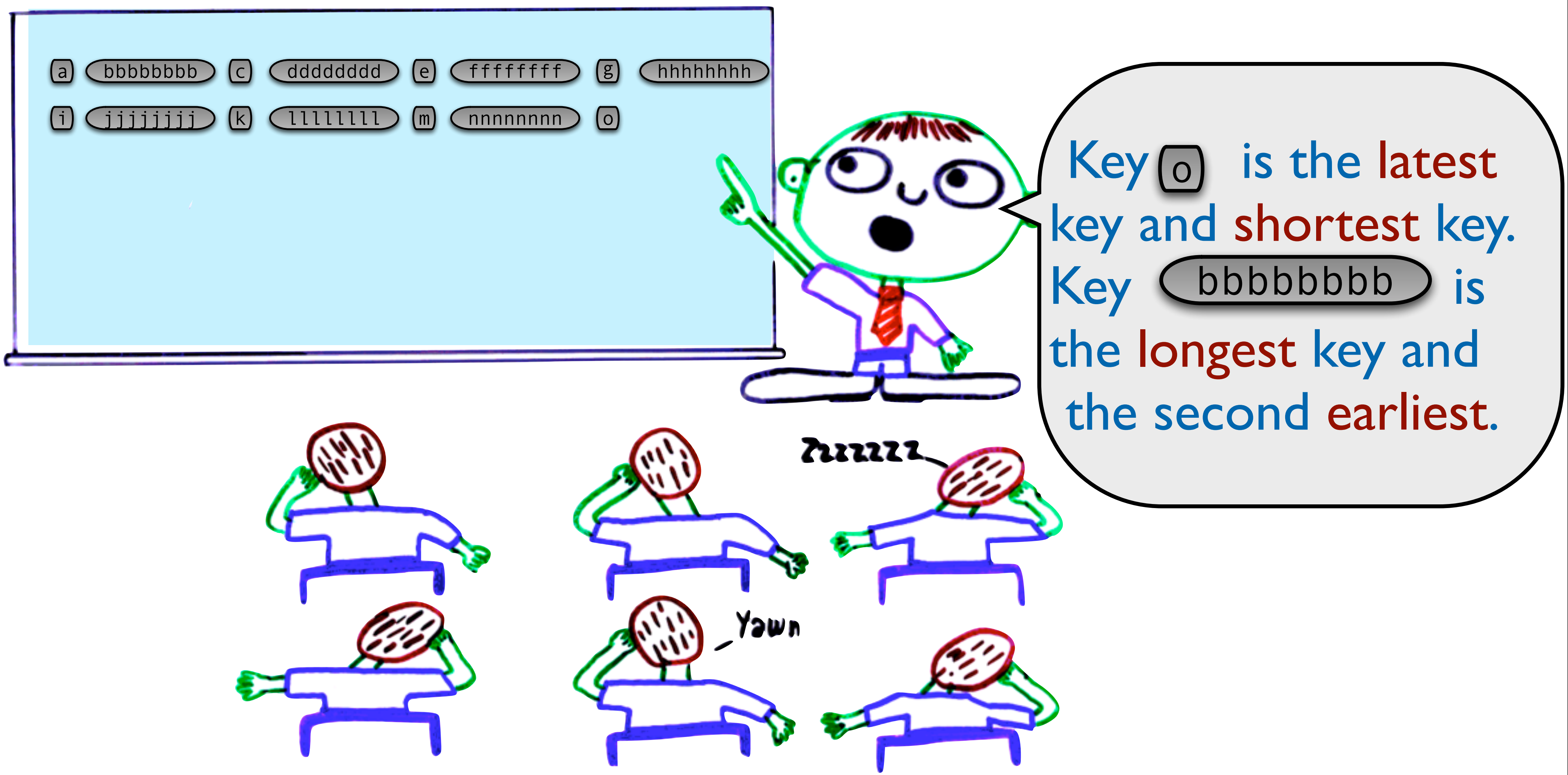
shortest versus *longest*: how many bytes to store key

(Words to avoid: *small*, *large*, *minimum*.)

So if I say....


a bbbbbbbb c dddddddd e ffffffff g hhhhhhhh
i jjjjjjjj k llllllll m nnnnnnnn o
Key o is the largest key and smallest key. Key bbbbbbbb is the largest key and the second smallest.
Zzzzzzz
Yawn

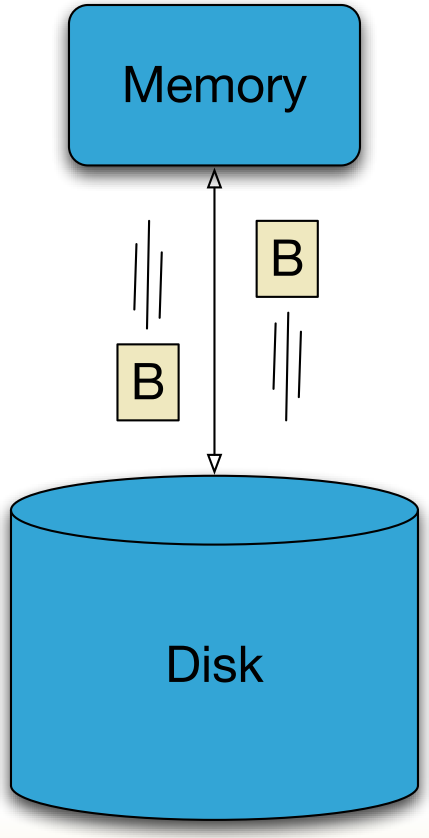
... I mean...


a bbbbbbbb c dddddddd e ffffffff g hhhhhhhh
i jjjjjjjj k llllllll m nnnnnnnn o
Key o is the latest key and shortest key.
Key bbbbbbbb is the longest key and the second earliest.
Zzzzzzz
Yawn

Algorithmic Performance Model

Disk-Access Machine (DAM) [Aggrawal, Vitter 88]

- Two-levels of memory.
- Two parameters: block-size ***B***, memory-size ***M***.



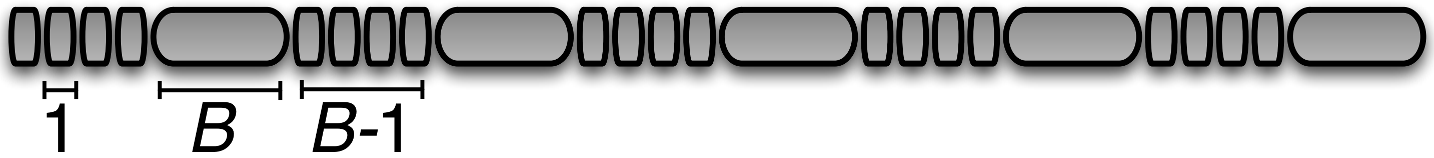
Performance metric:

- Minimize # of block transfers

Choice of Pivot Matters For Variable-Size Keys

Example: N **keys with average size** <2**.**

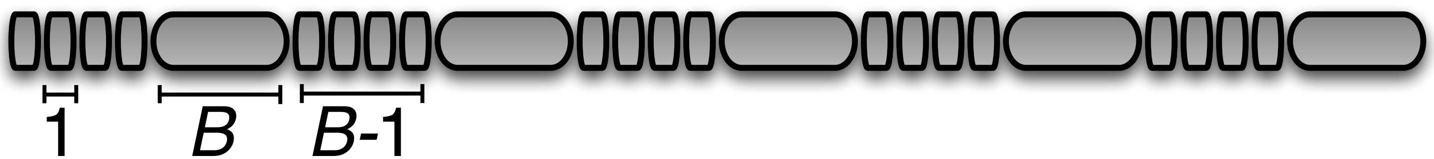
- N/B keys with size B and N-N/B keys with size 1.



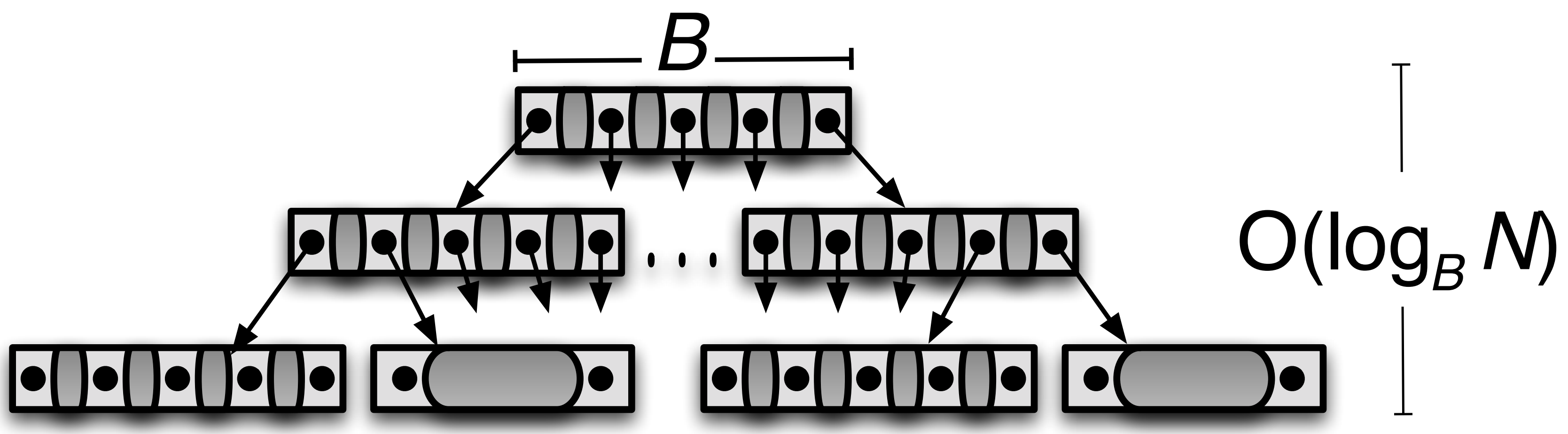
Choice of Pivot Matters For Variable-Size Keys

Example: ***N*** **keys with average size <2.**

- N/B keys with size B and $N-N/B$ keys with size 1.



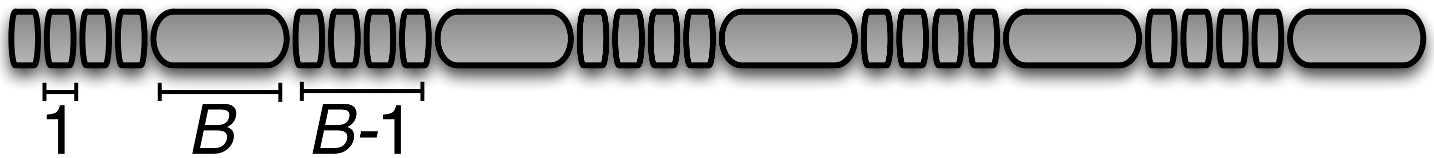
Size 1 keys as pivots: optimal.



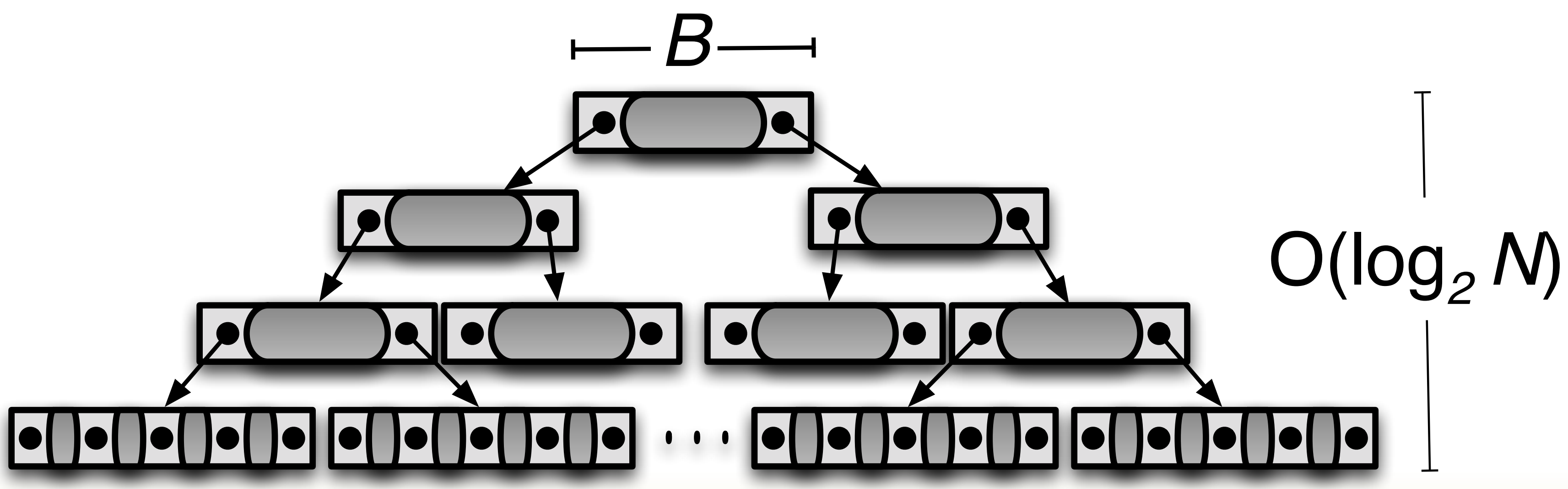
Choice of Pivot Matters For Variable-Size Keys

Example: ***N*** **keys with average size <2.**

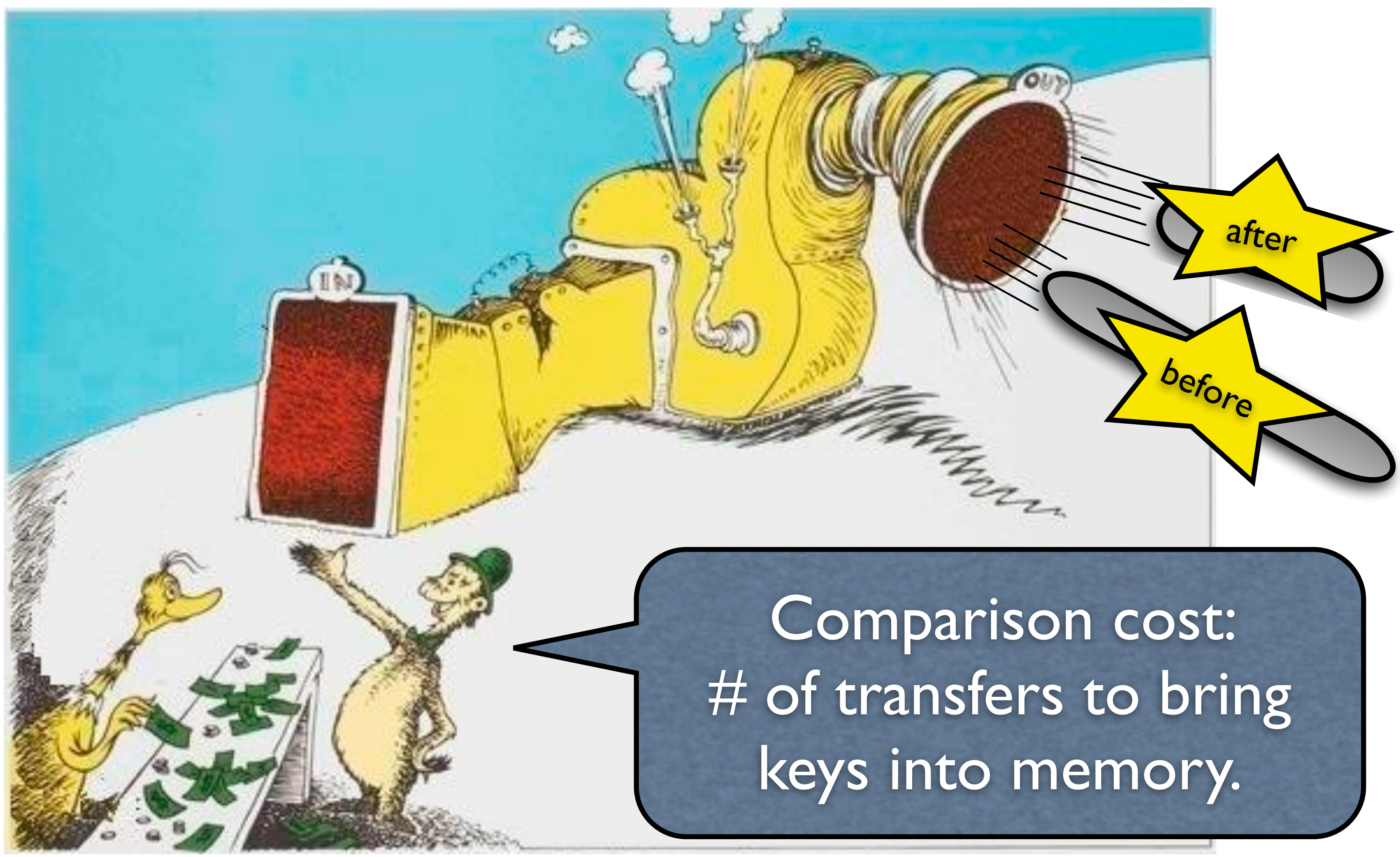
- N/B keys with size B and N-N/B keys with size 1.



Size *B* keys as pivots: $O(\log B)$ factor worse.



Comparison Cost



Desired Guarantee

Let *K* be the *average* key size.

Goal: $O(\log_{B/K} N)$ memory transfers per operation.

- Generalizes what happens if keys all have the same size K.

Desired Guarantee

Let *K* be the *average* key size.

Goal: $O(\log_{B/K} N)$ memory transfers per operation.

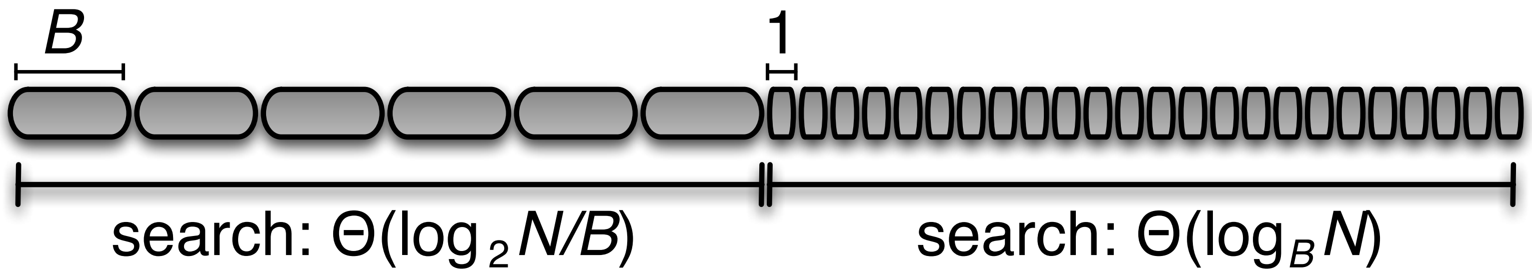
- Generalizes what happens if keys all have the same size *K*.

Unfortunately, we cannot get this for worst-case searches, but we'll get it in expectation.

Why We Cannot Attain Good Worst-Case Bounds

Example: *N* keys with average size *K*<2.

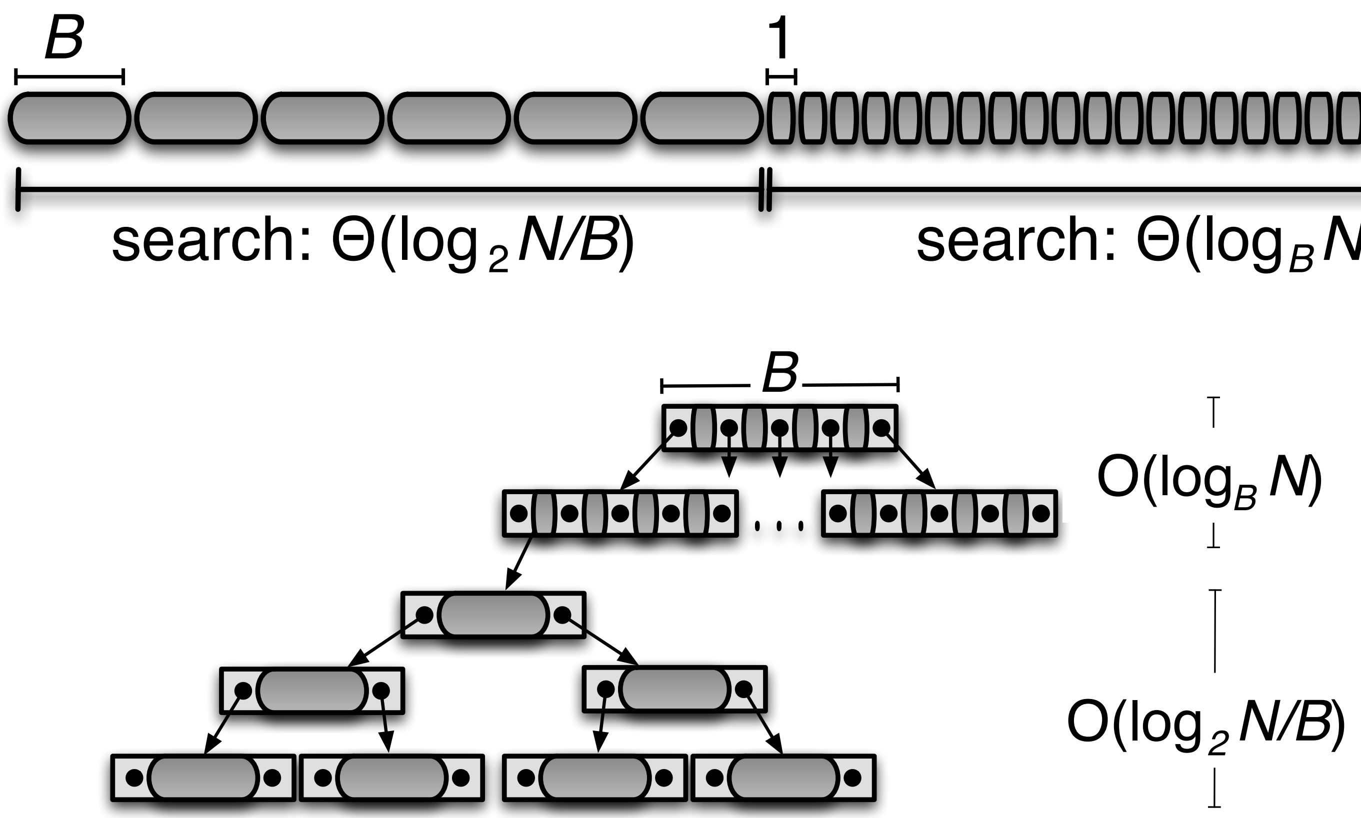
- *N*/*B* keys with size *B* and *N*-*N*/*B* keys with size 1.



Why We Cannot Attain Good Worst-Case Bounds

Example: ***N*** **keys with average size** ***K*<2.**

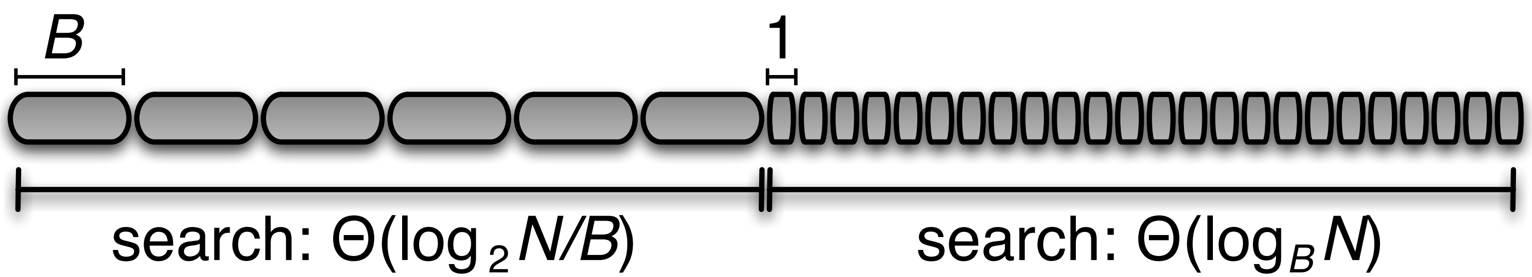
- *N*/*B* keys with size *B* and *N*-*N*/*B* keys with size 1.

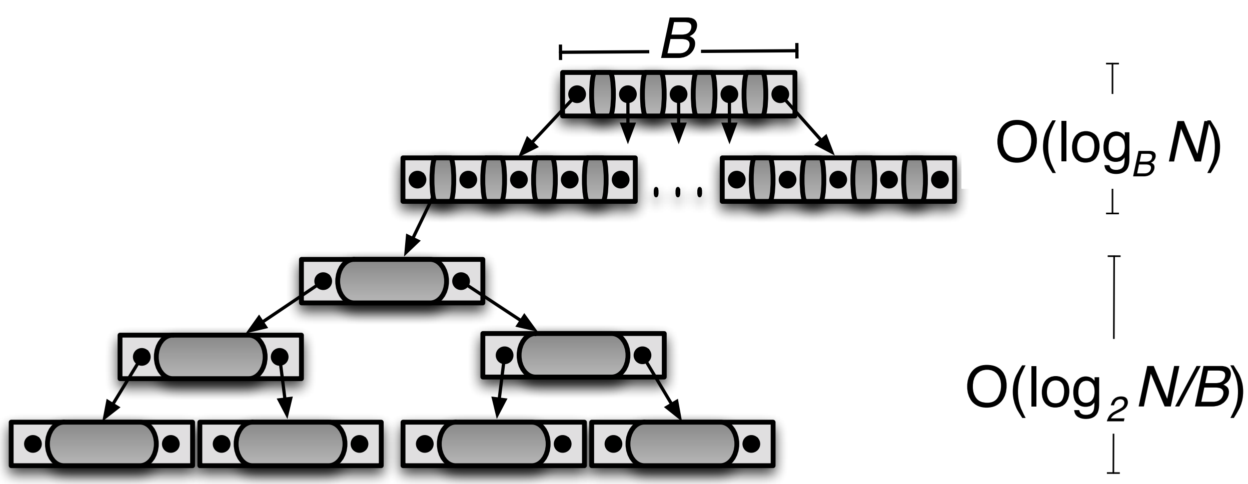


Why We Cannot Attain Good Worst-Case Bounds

Example: N keys with average size K<2.

- N/B keys with size B and N-N/B keys with size 1.



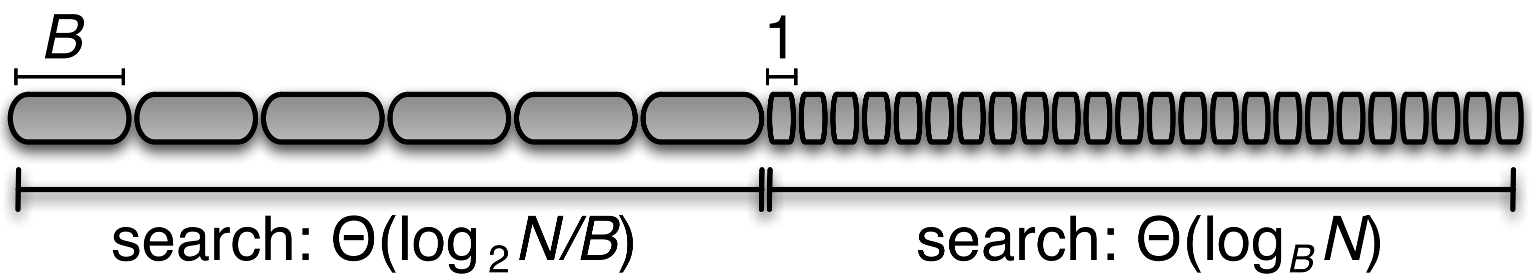


But we are ok in expectation: $(1-1/B)\log_B N + (1/B)\log_2 N = O(\log_B N)$.

Why We Cannot Attain Good Worst-Case Bounds

Example: N keys with average size K<2.

- N/B keys with size B and N-N/B keys with size 1.



Related work: how to optimize B-tree height

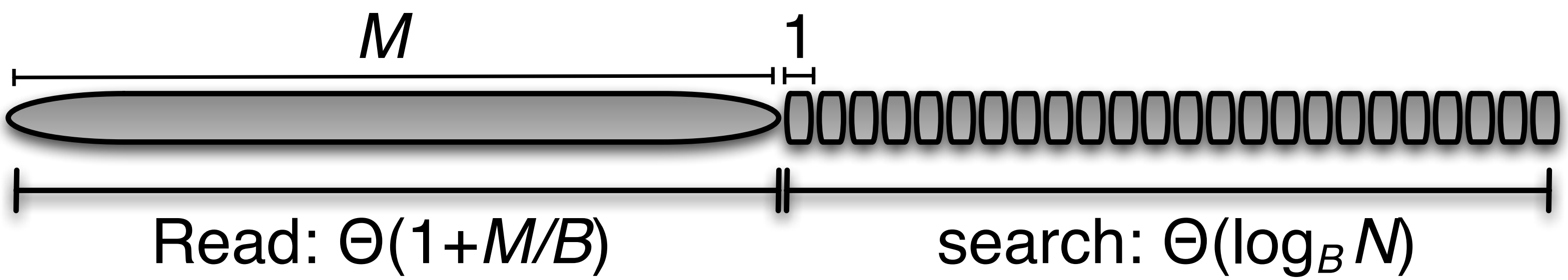
[Vaishnavi, Kriegel, Wood 80] [Gotleib 81] [Huang, Vishwanathan 90] [Becker 94]

- static (no inserts/deletes)
- DP-based
- (far from our target guarantee)

Why We Cannot Attain Good Worst-Case Bounds

Extreme example: N keys with average size K<2.

- 1 key with size M and N-1 keys with size 1.

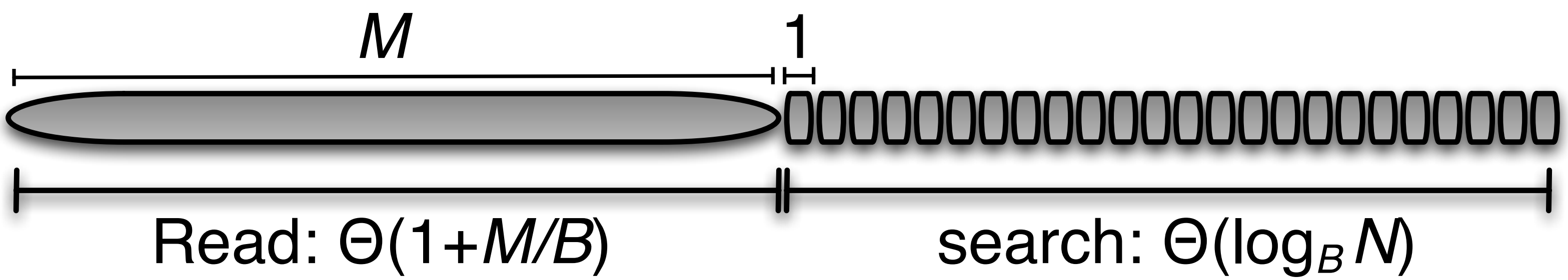


But we are ok in expectation: $(1-1/N)\log_B N + (1/N)\, M/B = O(\log_B N)$.

Why We Cannot Attain Good Worst-Case Bounds

Extreme example: *N* keys with average size *K*<2.

- 1 key with size M and N-1 keys with size 1.



But we are ok in expectation: $(1-1/N)\log_B N + (1/N)\, M/B = O(\log_B N)$.

These two examples have different flavors.

- LB for first example is based on tree structure.
- LB for for second example is based on reading the key.
- Both motivate why we consider expectation.

Static atomic-key B-tree (only searches)

- Expected leaf search cost: $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$
- Linear construction cost for sorted data: $O(NK/B)$

Results

Static atomic-key B-tree (only searches)

- Expected leaf search cost: $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$
- Linear construction cost for sorted data: $O(NK/B)$

Captures K=O(B) and $K \leq \Omega(B)$

Results

Static atomic-key B-tree (only searches)

- Expected leaf search cost: $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$ ← Captures K=O(B) and $K \leq \Omega(B)$
- Linear construction cost for sorted data: $O(NK/B)$ ← Scan bound since total length= NK

Results

Static atomic-key B-tree (only searches)

- Expected leaf search cost: $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$
- Linear construction cost for sorted data: $O(NK/B)$

Captures $K=O(B)$ and $K \leq \Omega(B)$

Scan bound since total length= NK

Dynamic atomic-key B-tree

- Expected leaf search cost : $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$
- Cost to insert/delete/search for key L of *random* rank (amort): $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N + |L|/B)$
- Cost to insert/delete/search for key of *arbitrary* rank: ***modification cost is dominated by search cost.***

Results

Static atomic-key B-tree (only searches)

- Expected leaf search cost: $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$ ← Captures K=O(B) and $K \leq \Omega(B)$
- Linear construction cost for sorted data: $O(NK/B)$ ← Scan bound since total length= NK

Dynamic atomic-key B-tree

- Expected leaf search cost : $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$
- Cost to insert/delete/search for key *L* of *random* rank (amort): $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N + |L|/B)$ ← O(|L|/B) is cost to read *L* into memory
- Cost to insert/delete/search for key of *arbitrary* rank: ***modification cost is dominated by search cost.***

Results

Static atomic-key B-tree (only searches)

- Expected leaf search cost: $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$ ← Captures K=O(B) and $K \leq \Omega(B)$
- Linear construction cost for sorted data: $O(NK/B)$ ← Scan bound since total length= NK

Dynamic atomic-key B-tree

- Expected leaf search cost : $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$
- Cost to insert/delete/search for key *L* of *random* rank (amort): $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N + |L|/B)$ ← O(|L|/B) is cost to read *L* into memory
- Cost to insert/delete/search for key of *arbitrary* rank: ***modification cost is dominated by search cost.*** ← important

Results

Static atomic-key B-tree (only searches)

- Expected leaf search cost: $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$ ← Captures K=O(B) and $K \leq \Omega(B)$
- Linear construction cost for sorted data: $O(NK/B)$ ← Scan bound since total length= NK

Dynamic atomic-key B-tree

- Expected leaf search cost : $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$
- Cost to insert/delete/search for key L of *random* rank (amort): $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N + |L|/B)$ ← O(|L|/B) is cost to read L into memory
- Cost to insert/delete/search for key of *arbitrary* rank: ***modification cost is dominated by search cost.*** ← important

Optimal static atomic-key B-tree:

- $O(BN^3)$ operations
- Applies even for nonuniform search probabilities

Results

Static atomic-key B-tree (only searches)

- Expected leaf search cost: $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$ ← Captures K=O(B) and $K \leq \Omega(B)$
- Linear construction cost for sorted data: $O(NK/B)$ ← Scan bound since total length= NK

Dynamic atomic-key B-tree

- Expected leaf search cost : $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$
- Cost to insert/delete/search for key *L* of *random* rank (amort): $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N + |L|/B)$ ← O(|L|/B) is cost to read *L* into memory
- Cost to insert/delete/search for key of *arbitrary* rank: ***modification cost is dominated by search cost.*** ← important

Optimal static atomic-key B-tree:

- $O(BN^3)$ operations ← RAM not external memory. (Won't discuss in talk.)
- Applies even for nonuniform search probabilities

Results

Static atomic-key B-tree (only searches) ← To discuss next

- Expected leaf search cost: $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$
- Linear construction cost for sorted data: $O(NK/B)$

Dynamic atomic-key B-tree

- Expected leaf search cost : $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$
- Cost to insert/delete/search for key *L* of *random* rank (amort): $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N + |L|/B)$
- Cost to insert/delete/search for key of *arbitrary* rank: ***modification cost is dominated by search cost.***

Optimal static atomic-key B-tree:

- $O(BN^3)$ operations
- Applies even for nonuniform search probabilities

Static Atomic-Key B-tree

Greedy construction algorithm

- Greedily select pivot elements for the root node
- Proceed recursively on all subtrees of the root.

Intuition

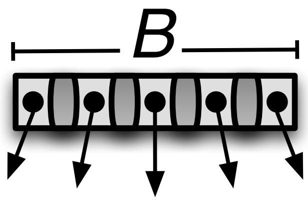
- Pick small keys in root to maximize fanout.
- Pick evenly distributed keys to reduce the search space.

To prove

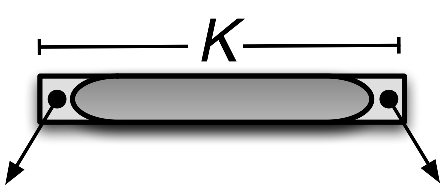
- Root has a good structure.
- Recursive substructures achieve good performance, even though subtrees may have different average key sizes.

Root Structure of Static Atomic Key B-tree

Case 1: $K=O(B)$. Root has size $O(B)$ and fanout $\Theta(B/K)$.

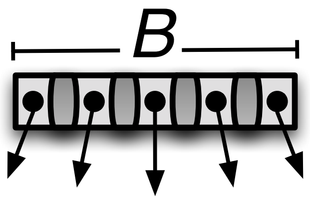


Case 2: $K=\Omega(B)$. Root has size $O(K)$ and fanout 2.



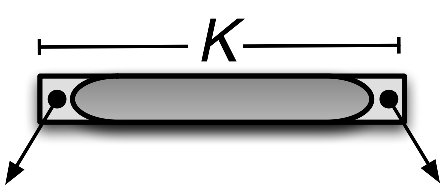
Root Structure of Static Atomic Key B-tree

Case 1: K=O(B). Root has size O(B) and fanout Θ(B/K).



Overall search cost: $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$

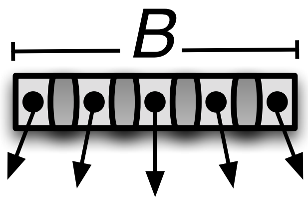
Case 2: K= Ω(B). Root has size O(K) and fanout 2.



Overall search cost: $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$

Root Structure of Static Atomic Key B-tree

Case 1: *K*=O(*B*). Root has size O(*B*) and fanout Θ(*B*/*K*).

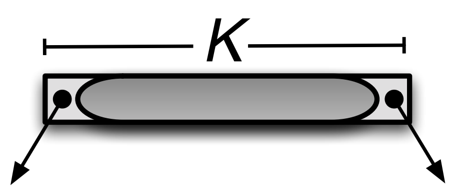


cost of reading root

ave fanout

Overall search cost: $O(\underbrace{\cancel{\lceil K/B \rceil}}_{=1} \log_{\underbrace{\cancel{1+\lceil B/K \rceil}}_{\approx\Theta(B/K)}} N)$

Case 2: *K*= Ω(*B*). Root has size O(*K*) and fanout 2.



cost of reading root

ave fanout

Overall search cost: $O(\underbrace{\cancel{\lceil K/B \rceil}}_{\approx K/B} \log_{\underbrace{\cancel{1+\lceil B/K \rceil}}_{=2}} N)$

Root Structure of Static Atomic Key B-tree

***Useful Lemma*: Consider N #s whose average is K. Divide into f groups of equal cardinality (each has N/f). Take the min in each group (say K_i is min of group i). Then average of these minima is at most the overall average K (i.e., $(K_1+K_2+\ldots+K_f)/f \le K$).**

Ex. 4 4 3 4 | 1 2 2 2 | 3 2 5 4 | 3 1 4 4

Ave K=3.

Root Structure of Static Atomic Key B-tree

***Useful Lemma*: Consider *N* #s whose average is *K*. Divide into *f* groups of equal cardinality (each has *N*/*f*). Take the min in each group (say K_i is min of group *i*). Then average of these minima is at most the overall average *K* (i.e., $(K_1+K_2+\ldots+K_f)/f \le K$).**

Ex. 4 4 3 4 | 1 2 2 2 | 3 2 5 4 | 3 1 4 4

Ave *K*=3. Ave of mins =1.75.

Root Structure of Static Atomic Key B-tree

***Useful Lemma*: Consider *N* #s whose average is *K*. Divide into *f* groups of equal cardinality (each has *N*/*f*). Take the min in each group (say K_i is min of group *i*). Then average of these minima is at most the overall average *K* (i.e., $(K_1+K_2+\ldots+K_f)/f \leq K$).**

Ex. 4 4 3 4 | 1 2 2 2 | 3 2 5 4 | 3 1 4 4

Ave *K*=3. Ave of mins =1.75.

Note: only true because groups have the same size.

Enough structure to bound the size and fanout of the root.

Root Construction

1. Divide keys into $f = \max\left\{3, \left\lfloor \frac{B}{K} \right\rfloor\right\}$ equal-size groups.
2. Pick shortest key in each group.
3. Store these keys in root (except 1st & last groups).

aaaa bbbb ccc dddd e ff gg hh iii jj kkkkk llll mmm n oooo pppp

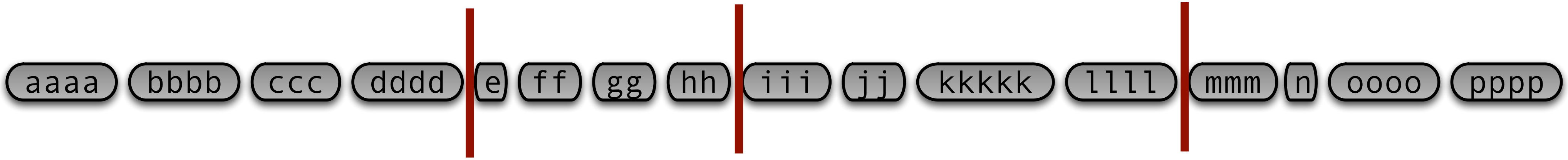
Ex: *B* = 12, *K* = 3. **So** *f*=4.

STONY BROOK

Tokutek™

Root Construction

1. **Divide keys into** $\mathrm{f} = \max\left\{3, \left\lfloor \frac{\mathrm{B}}{\mathrm{K}} \right\rfloor\right\}$ **equal-size groups.**
2. **Pick shortest key in each group.**
3. **Store these keys in root (except 1st & last groups).**



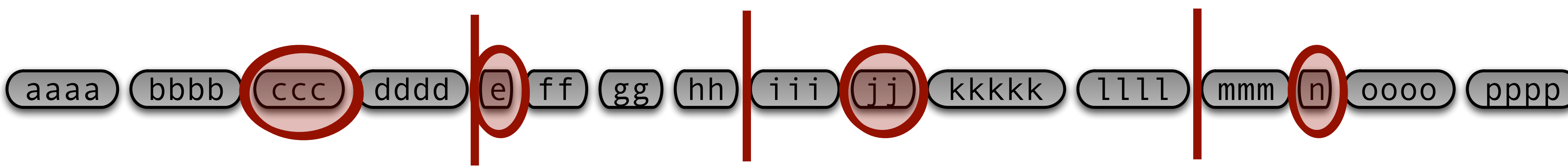
Ex: B = 12, K = 3. **So** f=4.

Root Construction

1. Divide keys into $f = \max\left\{3, \left\lfloor \frac{B}{K} \right\rfloor\right\}$ **equal-size groups.**

2. Pick shortest key in each group.

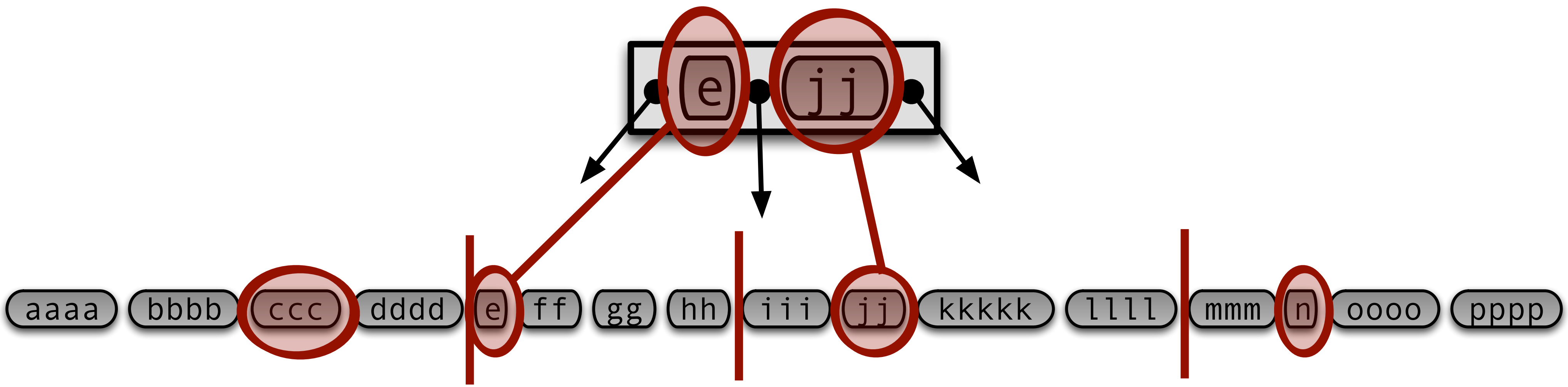
3. Store these keys in root (except 1st & last groups).



Ex: B = 12, K = 3. **So** f=4.

Root Construction

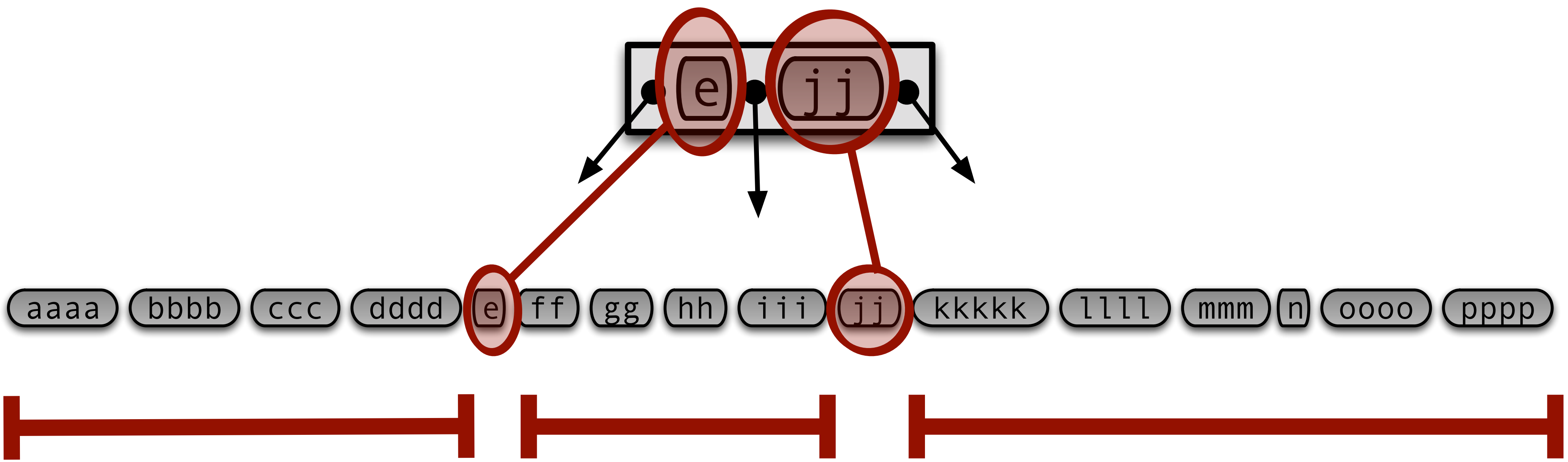
1. **Divide keys into** $f = \max\left\{3, \left\lfloor \frac{B}{K} \right\rfloor\right\}$ **equal-size groups.**
2. **Pick shortest key in each group.**
3. **Store these keys in root (except 1st & last groups).**



Ex: *B* = 12, *K* = 3. **So** *f*=4.

Construction of Rest of Tree

Proceed recursively in each subtree.
(Different value of *K* (and thus $f = \max\left\{3, \left\lfloor \frac{B}{K} \right\rfloor\right\}$)
in each subtree.



Results

Static atomic-key B-tree (only searches)

- Expected leaf search cost: $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$
- Linear construction cost for sorted data: $O(NK/B)$

Dynamic atomic-key B-tree

To discuss next

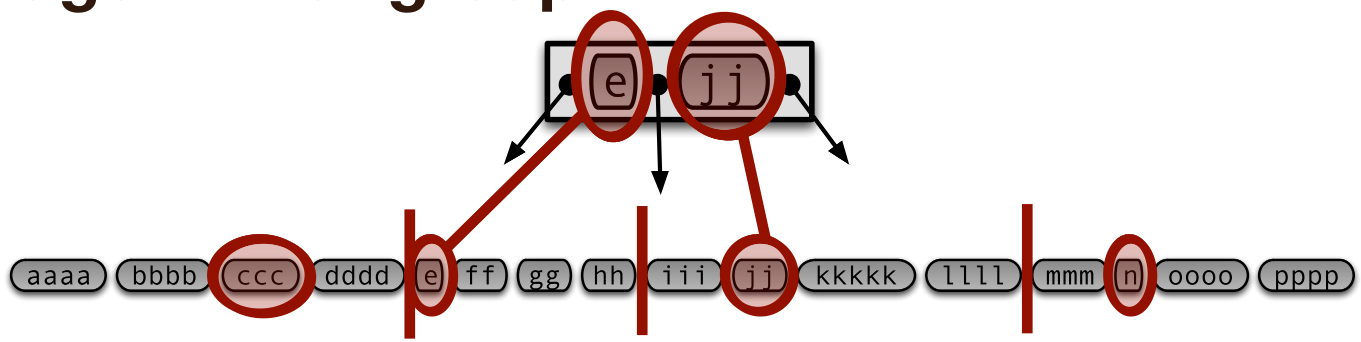
- Expected leaf search cost : $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$
- Cost to insert/delete/search for key *L* of *random* rank (amort): $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N + |L|/B)$
- Cost to insert/delete/search for key of *arbitrary* rank: ***modification cost is dominated by search cost.***

Optimal static atomic-key B-tree:

- $O(BN^3)$ operations
- Applies even for nonuniform search probabilities

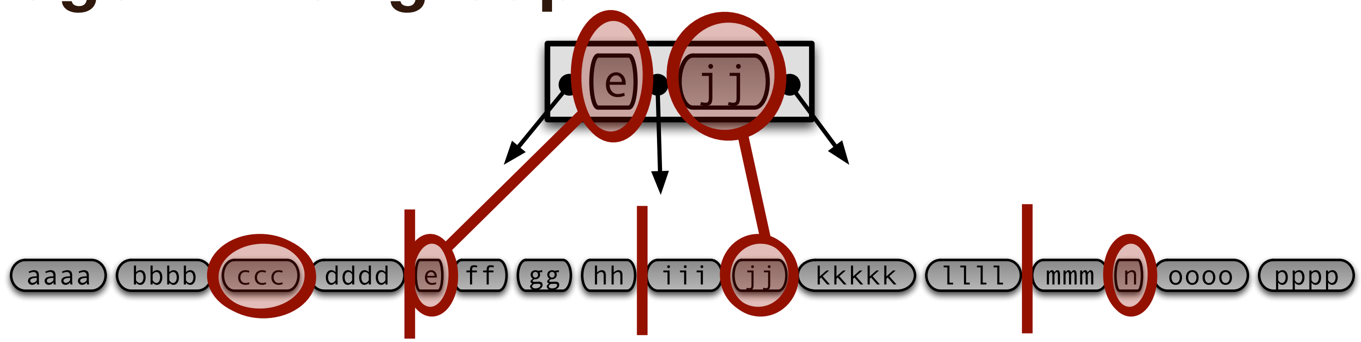
Dynamic Atomic-Key B-tree

One idea: **Groups need not be of equal cardinality.**
Within constant factors is good enough.
We don't need the shortest key as a pivot.
We can choose a key whose length is at most twice the average in that group.



Dynamic Atomic-Key B-tree

One idea: Groups need not be of equal cardinality.
Within constant factors is good enough.
We don't need the shortest key as a pivot.
We can choose a key whose length is at most twice the average in that group.



Thus, > half the keys in a group could be a pivot.

The shortest key can remain as pivot even if the group grows or shrinks by a constant factor.

Structure isn't brittle.

Dynamic Atomic-Key B-tree (Cont)

Second idea: **Insert elements directly into leaves. Rebuild entire subtrees whether there have been "too many" inserts/deletes.**
(Don't bother splitting and merging.)

$$\text{amortized update cost} = \frac{\text{rebuild cost}}{\text{\# updates between rebuilds}}$$

Problem: standard technique chokes because value of *K* changes over time. (Value of *K* during rebuild is different from value of *K* at actual insert/delete.)
Can be fixed. (Ask after talk.)

Waxing Philosophical

People want performance guarantees

- I have a startup Tokutek. One of the things customers like most about our product TokuDB is its predictability.

Waxing Philosophical

People want performance guarantees

- I have a startup Tokutek. One of the things customers like most about our product TokuDB is its predictability.

People want performance guarantees for B-trees

- The manual for Oracle Berkeley DB manual claims that BDB runs in $O(\log_{B/K} N)$ transfers.
- As our results show, this folk theorem is incorrect.
- But the claim helps motivate the guarantees we achieve.

Waxing Philosophical

People want performance guarantees

- I have a startup Tokutek. One of the things customers like most about our product TokuDB is its predictability.

People want performance guarantees for B-trees

- The manual for Oracle Berkeley DB manual claims that BDB runs in $O(\log_{B/K} N)$ transfers.
- As our results show, this folk theorem is incorrect.
- But the claim helps motivate the guarantees we achieve.

Will our theoretical guarantees have practical value?

- Maybe B-trees empirically perform predictably enough.

Waxing Philosophical

People want performance guarantees

- I have a startup Tokutek. One of the things customers like most about our product TokuDB is its predictability.

People want performance guarantees for B-trees

- The manual for Oracle Berkeley DB manual claims that BDB runs in $O(\log_{B/K} N)$ transfers.
- As our results show, this folk theorem is incorrect.
- But the claim helps motivate the guarantees we achieve.

Will our theoretical guarantees have practical value?

- Maybe B-trees empirically perform predictably enough.
- Maybe B-trees are so unpredictable already (e.g., because of memory cliffs) that our guarantees are second-order effects.

Waxing Philosophical

People want performance guarantees

- I have a startup Tokutek. One of the things customers like most about our product TokuDB is its predictability.

People want performance guarantees for B-trees

- The manual for Oracle Berkeley DB manual claims that BDB runs in $O(\log_{B/K} N)$ transfers.
- As our results show, this folk theorem is incorrect.
- But the claim helps motivate the guarantees we achieve.

Will our theoretical guarantees have practical value?

- Maybe B-trees empirically perform predictably enough.
- Maybe B-trees are so unpredictable already (e.g., because of memory cliffs) that our guarantees are second-order effects.
- Lots to explore.....

Waxing Philosophical

People want performance guarantees

- I have a startup Tokutek. One of the things customers like most about our product TokuDB is its predictability.

People want performance guarantees for B-trees

- The manual for Oracle Berkeley DB manual claims that BDB runs in $O(\log_{B/K} N)$ transfers.
- As our results show, this folk theorem is incorrect.
- But the claim helps motivate the guarantees we achieve.

Will our theoretical guarantees have practical value?

- Maybe B-trees empirically perform predictably enough.
- Maybe B-trees are so unpredictable already (e.g., because of memory cliffs) that our guarantees are second-order effects.
- Lots to explore.....