CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

B+ Trees
Announcements and Feedback

- Course evaluations are now open!
  - All feedback welcome, please fill them out
  - If 85% of the class has filled it out by the start of lecture on the last day of classes, I will release one of the final exam questions
Improving on Fence Pointers

- Fence pointer array (in memory)
- Page of actual data
Improving on Fence Pointers

1. Binary Search FP Table to find page
Improving on Fence Pointers

1. Binary Search FP Table to find page
2. Load page and binary search for record
Improving on Fence Pointers

- Fence pointer array (in memory)
- Fence pointer array (in a page on disk)
- Page of actual data
Improving on Fence Pointers

1. Binary Search @ Level 0 to find Level 1 page
Improving on Fence Pointers

1. Binary Search @ Level 0 to find Level 1 page

2. Load and search Level 1 page to find data page
Improving on Fence Pointers

1. Binary Search @ Level 0 to find Level 1 page

2. Load and search Level 1 page to find data page

3. Load and search data page for the record
Improving on Fence Pointers

- Fence pointer array (in memory)
- Fence pointer array (in a page on disk)
- Page of actual data

Diagram showing a tree-like structure with nodes representing fence pointer arrays and leaves representing pages of actual data.
Improving on Fence Pointers

1. Binary Search @ Level 0 to find Level 1 page

- Fence pointer array (in memory)
- Fence pointer array (in a page on disk)
- Page of actual data
Improving on Fence Pointers

1. Binary Search @ Level 0 to find Level 1 page

2. Load and search Level 1 page to find Level 2 page

Page of actual data

Fence pointer array (in memory)

Fence pointer array (in a page on disk)
Improving on Fence Pointers

1. Binary Search @ Level 0 to find Level 1 page
2. Load and search Level 1 page to find Level 2 page
3. Load and search Level 2 page to find data page
Improving on Fence Pointers

1. Binary Search @ Level 0 to find Level 1 page
2. Load and search Level 1 page to find Level 2 page
3. Load and search Level 2 page to find data page
4. Load and search data page to find the record
Improving on Fence Pointers ISAM Index

1. Binary Search @ Level 0 to find Level 1 page
2. Load and search Level 1 page to find Level 2 page
3. Load and search Level 2 page to find data page
4. Load and search data page to find the record
ISAM Index

IO Complexity:
- 1 read at L0 (or assume already in memory)
- 1 read at L1
- 1 read at L2
- ...
- 1 read at $L_{\text{max}}$
- 1 read at data level
How many levels will there be (this isn't a binary tree...)
ISAM Index

How many levels will there be (this isn't a binary tree...)

- Level 0: 1 page with $C_{\text{key}}$ keys
How many levels will there be (this isn't a binary tree...)

- Level 0: 1 page w/$C_{key}$ keys
- Level 1: Up to $C_{key}$ pages w/$C_{key}^2$ keys
ISAM Index

How many levels will there be (this isn't a binary tree...)

- Level 0: 1 page with $C_{key}$ keys
- Level 1: Up to $C_{key}$ pages with $C_{key}^2$ keys
- Level 2: Up to $C_{key}^2$ pages with $C_{key}^3$ keys
- ...
How many levels will there be (this isn't a binary tree...)

- Level 0: 1 page with $C_{key}$ keys
- Level 1: Up to $C_{key}$ pages with $C_{key}^2$ keys
- Level 2: Up to $C_{key}^2$ pages with $C_{key}^3$ keys
- ...  
- Level max: Up to $C_{key}^{max}$ pages with $C_{key}^{max+1}$ keys
How many levels will there be (this isn't a binary tree...)

- Level 0: 1 page w/ $C_{key}^{0}$ keys
- Level 1: Up to $C_{key}^{1}$ pages w/ $C_{key}^{2}$ keys
- Level 2: Up to $C_{key}^{2}$ pages w/ $C_{key}^{3}$ keys
- ...
- Level max: Up to $C_{key}^{max}$ pages w/ $C_{key}^{max+1}$ keys
- Data Level: Up to $C_{key}^{max+1}$ pages w/ $C_{data}C_{key}^{max+1}$ records
ISAM Index

\[ n = C_{\text{data}} C_{\text{key}}^{\max + 1} \]
ISAM Index

\[ n = C_{\text{data}} C_{\text{key}}^{\text{max} + 1} \]

\[ \frac{n}{C_{\text{data}}} = C_{\text{key}}^{\text{max} + 1} \]
ISAM Index

\[ n = C_{\text{data}} C_{\text{key}}^{\max+1} \]

\[ \frac{n}{C_{\text{data}}} = C_{\text{key}}^{\max+1} \]

\[ \log_{C_{\text{key}}} \left( \frac{n}{C_{\text{data}}} \right) = \max + 1 \]
ISAM Index

\[ n = C_{data} C_{key}^{\text{max}+1} \]

\[ \frac{n}{C_{data}} = C_{key}^{\text{max}+1} \]

\[ \log_{C_{key}} \left( \frac{n}{C_{data}} \right) = \text{max} + 1 \]

\[ \log_{C_{key}} (n) - \log_{C_{key}} (C_{data}) = \text{max} + 1 \]
ISAM Index

\[ n = C_{\text{data}} C_{\text{key}}^{max+1} \]

\[ \frac{n}{C_{\text{data}}} = C_{\text{key}}^{max+1} \]

\[ \log_{C_{\text{key}}} \left( \frac{n}{C_{\text{data}}} \right) = max + 1 \]

\[ \log_{C_{\text{key}}}(n) - \log_{C_{\text{key}}}(C_{\text{data}}) = max + 1 \]

**Number of Levels:** \( O \left( \log_{C_{\text{key}}}(n) \right) \)
ISAM Index

\[ n = C_{\text{data}} C_{\text{key}}^{\max + 1} \]

\[ \frac{n}{C_{\text{data}}} = C_{\text{key}}^{\max + 1} \]

\[ \log_{C_{\text{key}}} \left( \frac{n}{C_{\text{data}}} \right) = \max + 1 \]

\[ \log_{C_{\text{key}}} (n) - \log_{C_{\text{key}}} (C_{\text{data}}) = \max + 1 \]

Note this isn't base 2!

Number of Levels: \[ O \left( \log_{C_{\text{key}}} (n) \right) \]
ISAM Index

Like Binary Search, but "Cache-Friendly"

- Still takes $O(\log(n))$ steps
- Still requires $O(1)$ memory (1 page at a time)
- Now requires $\log_{c_{\text{key}}}(n)$ loads from disk ($\log_{c_{\text{key}}}(n) \ll \log_2(n)$)
What if the data changes?
Inserting New Records

Disk:

- keys 0 - 178
- keys 192 - 273
- keys 274 - 412
- keys 412 - 611
- ...  

Page 0

Page 1

Page 2

Page 3

Insert record with key 181
Inserting New Records

**Idea:** Keep "free" space on each page for new records

... what happens when it fills up?
Inserting New Records

Idea: Linked lists to store overflow

...but now our I/O complexity is $O(n)$ again...
Inserting New Records

Idea: We'll have to rearrange the tree
Dynamic Page Allocation

Treat the disk as an ADT:

allocate(): PageID
  ● Allocates a page in the data file and returns its position

load[T](page: PageID): T
  ● Reads in a 4k chunk of data

write[T](page: PageID, data: T)
  ● Writes a 4k chunk of data to the page
Pointers to Pages

Our pages are now dynamic, need "pointers" instead of indices

PageID (pointers)

Keys
Pointers to Pages

Our pages are now dynamic, need "pointers" instead of indices
Free Space Revisited

Diagram showing free space allocation with numbers 6, 12, 21, and 19.
Free Space Revisited

Add 9

1 3 6

8 12

19 21
Free Space Revisited

Add 9
Free Space Revisited

Add 14

1 3 6
8 9 12
19 21
Free Space Revisited

Add 14

1 3 6

8 9 12

14 19 21
Free Space Revisited

Add 10

1 3 6
8 9 12
14 19 21

6 12 21 - -
Free Space Revisited

Add 10

1 3 6
8 9 10 12
14 19 21
Free Space Revisited

Add 11? Where does it go?
Free Space Revisited

Add 11? Where does it go? Split the page!
Free Space Revisited

Add 11? Where does it go? Split the page!
Free Space Revisited

Add 11? Where does it go? Split the page!
Free Space Revisited

Add 11? Where does it go? Split the page!
Free Space Revisited

Add 22, 27?
Free Space Revisited

Add 22, 27?
Free Space Revisited

Add 22, 27? Split the page of pointers!

1 3 6
8 9 10
11 12
Free Space Revisited

Add 22, 27? Split the page of pointers!
B+ Tree (Almost)

Insert

1. Find the page the record belongs on
2. Insert record there
3. If full, "split" the page
   a. Insert additional separator in the parent directory
   b. If full, split the parent directory and repeat
      i. If root is split, create a new root
Observation: Don't need the largest key
B+ Trees

Observation: Don't need the largest key
**Question:** What if separators are mispositioned? What if we insert 13?
**B+ Trees**

**Question:** What if separators are mispositioned? What if we insert 13?

**Idea:** Steal space from neighbor (and update separator)
Question: What if we delete records?
B+ Trees

Delete 22, 27
Delete 22,27
B+ Trees

Delete 22,27
B+ Trees

Delete 8-12

1 3 6
8 9 10
11 12
14 19 21
B+ Trees

Delete 8-12
Problem: We have $O(\log(n))$ reads per search for the biggest $n$ in the tree's history.
B+ Trees Minimum Fill

Enforce that each directory and data node must have \( \geq \frac{c}{2} \) records

- **Exception**: the root

What does this do to tree depth?

- \( O(\log_{c/2}(n)) \) (as compared to \( O(\log_c(n)) \) when the tree is static)
B+ Trees Minimum Fill
B+ Trees Minimum Fill

This node is underfull
B+ Trees Minimum Fill

This node is underfull.
B+ Trees Minimum Fill
B+ Trees Minimum Fill

This node is underfull
B+ Trees Minimum Fill
B+ Trees

Delete

1. Find the page the record is on
2. Delete the record (if present)
3. If underfull, "merge" the page with a neighbor
   a. If either neighbor has > c/2 entries then steal instead
   b. If parent underfull, repeat
      i. If root, then drop the lowest layer