CMU SCS

15-826: Multimedia Databases and Data Mining

Primary key indexing – B-trees

Christos Faloutsos - CMU

www.cs.cmu.edu/~christos

CMU SCS

Problem

Given a large collection of (multimedia) records, find similar/interesting things, ie:

- Allow fast, approximate queries, and
- Find rules/patterns

CMU SCS

Outline

Goal: 'Find similar / interesting things'

- Intro to DB
- Indexing - similarity search
- Data Mining

CMU SCS

Indexing - Detailed outline

- primary key indexing
  - B-trees and variants
  - (static) hashing
  - extendible hashing
- secondary key indexing
- spatial access methods
- text
- ...

CMU SCS

Primary key indexing

- find employee with ssn=123

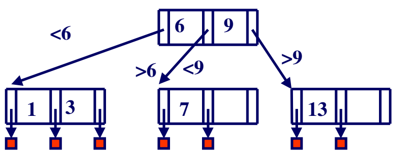
CMU SCS

B-trees

- the **most successful** family of index schemes (B-trees, B^{+}-trees, B^{*}-trees)
- Can be used for primary/secondary, clustering/non-clustering index.
- balanced "n-way" search trees

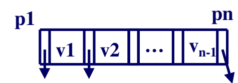
B-trees

Eg., B-tree of order 3:



B - tree properties:

- each node, in a B-tree of order *n*:
  - Key order
  - at most *n* pointers
  - at least *n/2* pointers (except root)
  - all leaves at the same level
  - if number of pointers is *k*, then node has exactly k-1 keys
  - (leaves are empty)

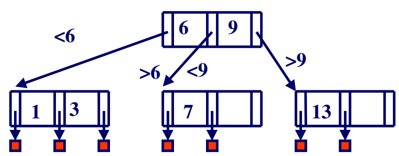


Properties

- “block aware” nodes: each node -> disk page
- O(log (N)) for everything! (ins/del/search)
- typically, if m = 50 - 100, then 2 - 3 levels
- utilization >= 50%, guaranteed; on average 69%

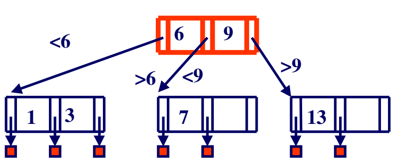
Queries

- Algo for exact match query? (eg., ssn=8?)



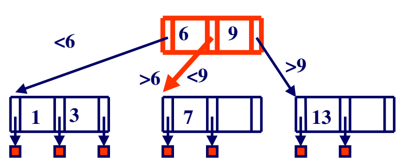
Queries

- Algo for exact match query? (eg., ssn=8?)



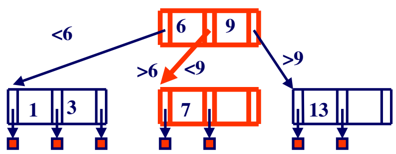
Queries

- Algo for exact match query? (eg., ssn=8?)



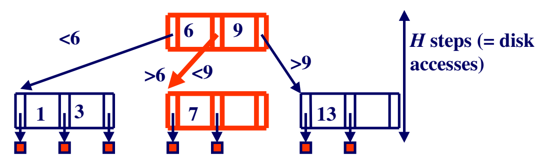
Queries

- Algo for exact match query? (eg., ssn=8?)



Queries

- Algo for exact match query? (eg., ssn=8?)

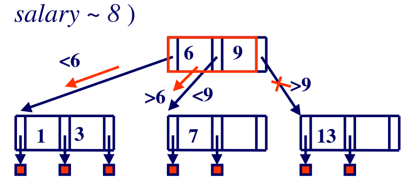


Queries

- what about range queries? (eg., 5<*salary*<8)
- Proximity/ nearest neighbor searches? (eg., *salary* ~ *8*)

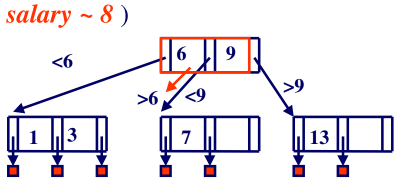
Queries

- what about range queries? (eg., ***5<salary<8***)
- Proximity/ nearest neighbor searches? (eg., *salary* ~ *8*)



Queries

- what about range queries? (eg., 5<*salary*<8)
- Proximity/ nearest neighbor searches? (eg., ***salary ~ 8***)

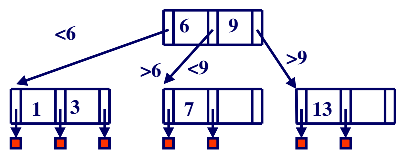


B-trees: Insertion

- Insert in leaf; on overflow, push middle up (recursively)
- split: preserves B - tree properties

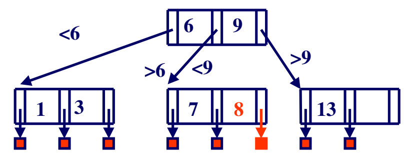
B-trees

Easy case: Tree T0; insert ‘8’



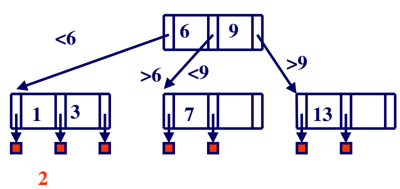
B-trees

Tree T0; insert ‘8’



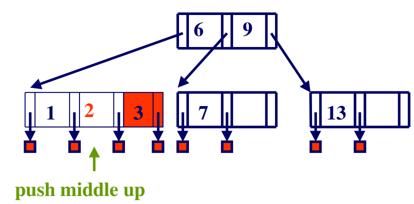
B-trees

Hardest case: Tree T0; insert ‘2’



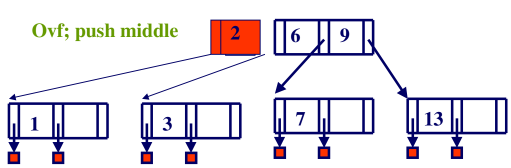
B-trees

Hardest case: Tree T0; insert ‘2’



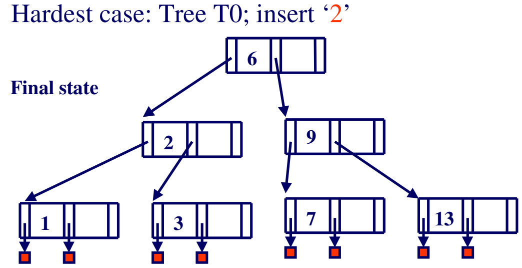
B-trees

Hardest case: Tree T0; insert ‘2’



B-trees

Hardest case: Tree T0; insert ‘2’



B-trees: Insertion

- Q: What if there are two middles? (eg, order 4)
- A: either one is fine

B-trees: Insertion

- Insert in leaf; on overflow, push middle up (recursively – 'propagate split')
- split: preserves all B - tree properties (!!)
- notice how it grows: height increases when root overflows & splits
- Automatic, incremental re-organization

Overview

- B – trees
  - Dfn, Search, insertion, **deletion**
- B+ - trees
- hashing

Deletion

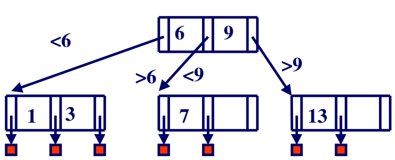
Rough outline of algo:

- Delete key;
- on underflow, may need to merge

In practice, some implementors just allow underflows to happen…

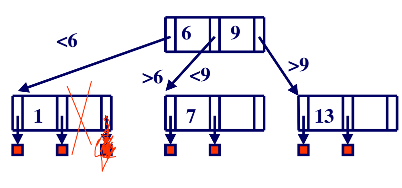
B-trees – Deletion

Easiest case: Tree T0; delete '3'



B-trees – Deletion

Easiest case: Tree T0; delete '3'

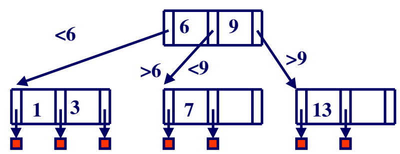


B-trees – Deletion

- Case1: delete a key at a leaf – no underflow
- Case2: delete non-leaf key – no underflow
- Case3: delete leaf-key; underflow, and 'rich sibling'
- Case4: delete leaf-key; underflow, and 'poor sibling'

B-trees – Deletion

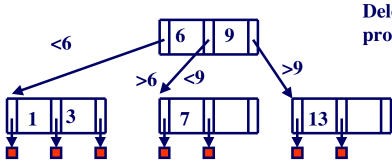
- Case1: delete a key at a leaf – no underflow (delete 3 from T0)



B-trees – Deletion

- Case2: delete a key at a non-leaf – no underflow (eg., delete 6 from T0)

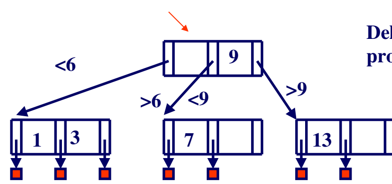
Delete & promote, ie:



B-trees – Deletion

- Case2: delete a key at a non-leaf – no underflow (eg., delete 6 from T0)

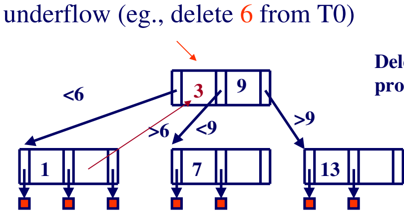
Delete & promote, ie:



B-trees – Deletion

- Case2: delete a key at a non-leaf – no underflow (eg., delete 6 from T0)

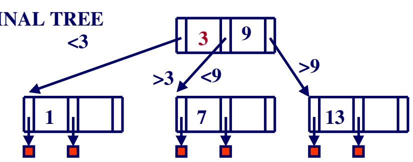
Delete & promote, ie:



B-trees – Deletion

- Case2: delete a key at a non-leaf – no underflow (eg., delete 6 from T0)

FINAL TREE



B-trees – Deletion

- Case2: delete a key at a non-leaf – no underflow (eg., delete 6 from T0)
- Q: How to promote?
- A: pick the largest key from the left sub-tree (or the smallest from the right sub-tree)
- Observation: every deletion eventually becomes a deletion of a leaf key

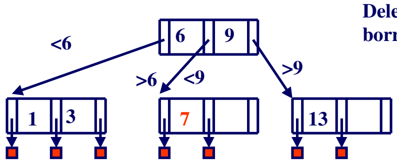
B-trees – Deletion

- Case1: delete a key at a leaf – no underflow
- Case2: delete non-leaf key – no underflow
- ⇨ Case3: delete leaf-key; underflow, and ‘rich sibling’
- Case4: delete leaf-key; underflow, and ‘poor sibling’

B-trees – Deletion

- Case3: underflow & ‘rich sibling’ (eg., delete 7 from T0)

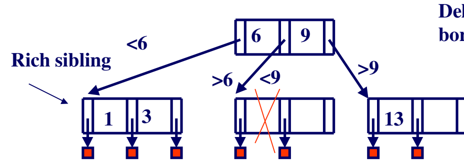
Delete & borrow, ie:



B-trees – Deletion

- Case3: underflow & ‘rich sibling’ (eg., delete 7 from T0)

Delete & borrow, ie:



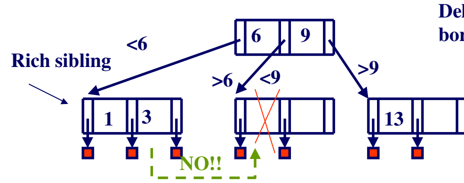
B-trees – Deletion

- Case3: underflow & ‘rich sibling’
- ‘rich’ = can give a key, without underflowing
- ‘borrowing’ a key: THROUGH the PARENT!

B-trees – Deletion

- Case3: underflow & ‘rich sibling’ (eg., delete 7 from T0)

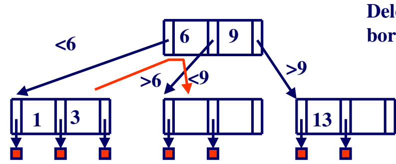
Delete & borrow, ie:



B-trees – Deletion

- Case3: underflow & ‘rich sibling’ (eg., delete 7 from T0)

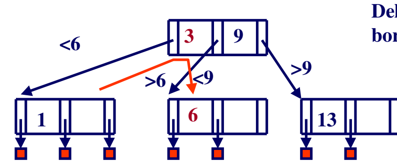
Delete & borrow, ie:



B-trees – Deletion

- Case3: underflow & ‘rich sibling’ (eg., delete 7 from T0)

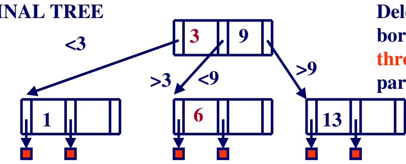
Delete & borrow, ie:



B-trees – Deletion

- Case3: underflow & ‘rich sibling’ (eg., delete 7 from T0)

FINAL TREE



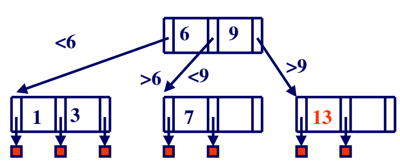
Delete & borrow, through the parent

B-trees – Deletion

- Case1: delete a key at a leaf – no underflow
- Case2: delete non-leaf key – no underflow
- Case3: delete leaf-key; underflow, and ‘rich sibling’
- ➡ Case4: delete leaf-key; underflow, and ‘poor sibling’

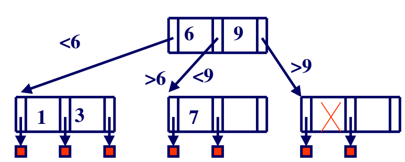
B-trees – Deletion

- Case4: underflow & ‘poor sibling’ (eg., delete 13 from T0)



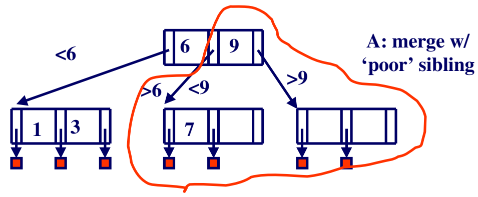
B-trees – Deletion

- Case4: underflow & ‘poor sibling’ (eg., delete 13 from T0)



B-trees – Deletion

- Case4: underflow & ‘poor sibling’ (eg., delete 13 from T0)

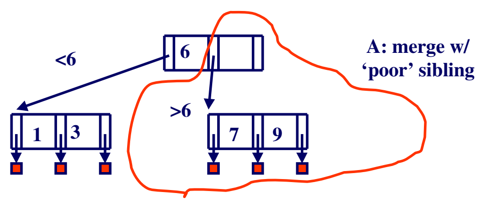


B-trees – Deletion

- Case4: underflow & ‘poor sibling’ (eg., delete 13 from T0)
- Merge, by pulling a key from the **parent**
- exact reversal from insertion: ‘split and push up’, vs. ‘merge and pull down’
- Ie.:

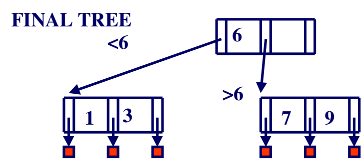
B-trees – Deletion

- Case4: underflow & ‘poor sibling’ (eg., delete 13 from T0)



B-trees – Deletion

- Case4: underflow & ‘poor sibling’ (eg., delete 13 from T0)



B-trees – Deletion

- Case4: underflow & ‘poor sibling’
- -> ‘pull key from parent, and merge’
- Q: What if the parent underflows?
- A: repeat recursively

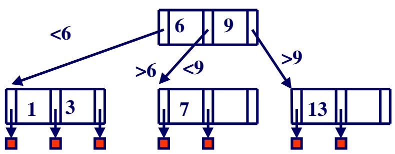
Overview

- B – trees
- **B+ - trees, B*-trees**
- hashing

CMU SCS

B+ trees - Motivation

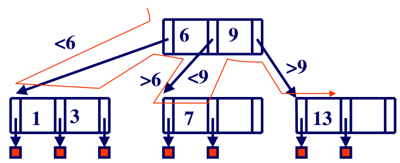
B-tree – print keys in sorted order:



CMU SCS

B+ trees - Motivation

B-tree needs back-tracking – how to avoid it?



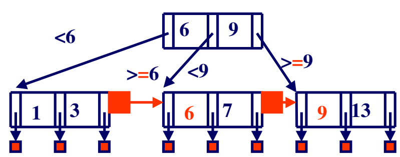
CMU SCS

Solution: B$^+$ - trees

- facilitate sequential ops
- They string all leaf nodes together
- AND
- replicate keys from non-leaf nodes, to make sure every key appears at the leaf level

CMU SCS

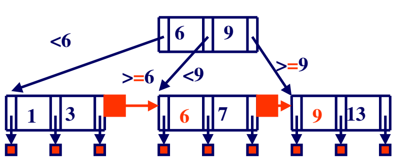
B+ trees



CMU SCS

B+ trees - insertion

Eg., insert '8'



CMU SCS

Overview

- B – trees
- **B+ - trees, B*-trees**
- hashing

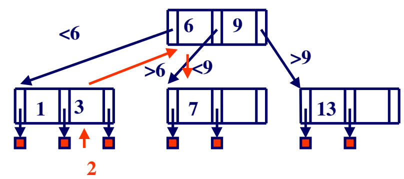
B*-trees

- splits drop util. to 50%, and maybe increase height
- How to avoid them?

B*-trees: deferred split!

- Instead of splitting, LEND keys to sibling!

(through PARENT, of course!)

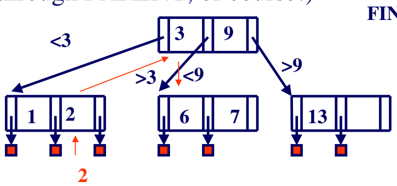


B*-trees: deferred split!

- Instead of splitting, LEND keys to sibling!

(through PARENT, of course!)

FINAL TREE



B*-trees: deferred split!

- Notice: shorter, more packed, faster tree
- It's a rare case, where space utilization and speed improve together
- BUT: What if the sibling has no room for our 'lending'?

B*-trees: deferred split!

- BUT: What if the sibling has no room for our 'lending'?
- A: 2-to-3 split: get the keys from the sibling, pool them with ours (and a key from the parent), and split in 3.
- Details: too messy (and even worse for deletion)

Conclusions

- Main ideas: recursive; block-aware; on overflow -> split; **defer** splits
- All B-tree variants have excellent, **O(logN) worst-case performance for ins/del/search**
- It's the prevailing indexing method
- More details: [Knuth vol 3.]