CMU SCS

Carnegie Mellon Univ.
Dept. of Computer Science
15-415/615 – DB Applications

C. Faloutsos & A. Pavlo

Lecture#9 (R&G ch. 10)

Indexing

CMU SCS

Outline

- **Motivation**
- ISAM
- B-trees (not in book)
- B+ trees
- duplicates
- B+ trees in practice

CMU SCS

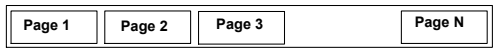
Introduction

- How to support range searches?
- equality searches?

CMU SCS

Range Searches

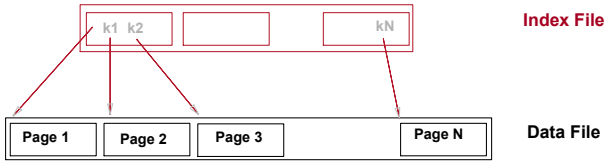
- ``*Find all students with gpa > 3.0*''
- may be slow, even on sorted file
- What to do?



CMU SCS

Range Searches

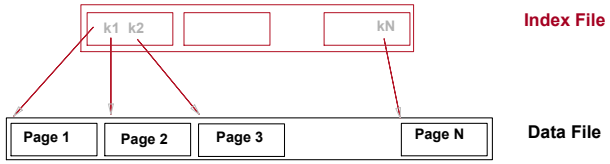
- ``*Find all students with gpa > 3.0*''
- may be slow, even on sorted file
- Solution: Create an `index' file.



CMU SCS

Range Searches

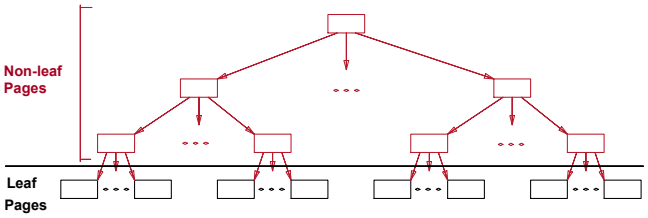
- More details:
- if index file is small, do binary search there
- Otherwise??



CMU SCS

ISAM

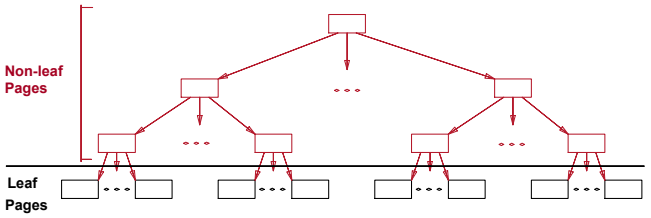
- Repeat recursively!



CMU SCS

ISAM

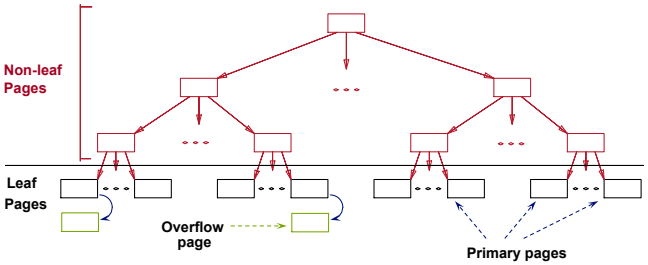
- OK - what if there are insertions and overflows?



CMU SCS

ISAM

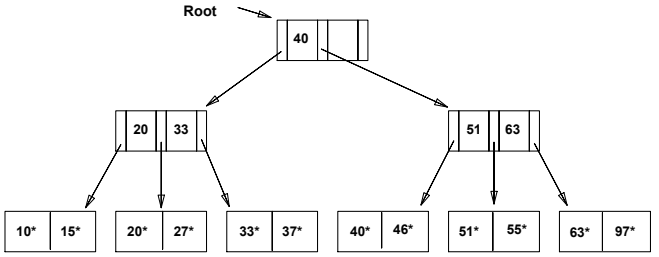
- Overflow pages, linked to the primary page



CMU SCS

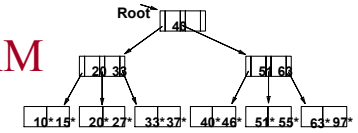
Example ISAM Tree

- **2 entries per page**



CMU SCS

ISAM

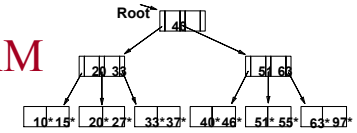


Details

- format of an index page?
- how full should a newly created ISAM be?

CMU SCS

ISAM



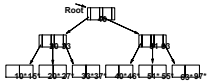
Details

- format of an index page?
- how full should a newly created ISAM be?
  - ~80-90% (**not** 100%)

CMU SCS

ISAM is a STATIC Structure

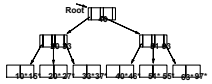
- that is, index pages don't change
- *File creation*: Leaf (data) pages allocated sequentially, sorted by search key; then index pages allocated, then overflow pgs.



CMU SCS

ISAM is a STATIC Structure

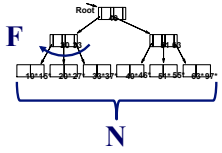
- *Search*: Start at root; use key comparisons to go to leaf.
- Cost = ??



CMU SCS

ISAM is a STATIC Structure

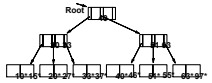
- *Search*: Start at root; use key comparisons to go to leaf.
- Cost = ??



CMU SCS

ISAM is a STATIC Structure

- *Search*: Start at root; use key comparisons to go to leaf.
- Cost = $\log_F N$;
- F = # entries/pg (i.e., fanout),
- N = # leaf pgs

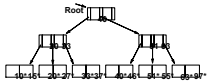


CMU SCS

ISAM is a STATIC Structure

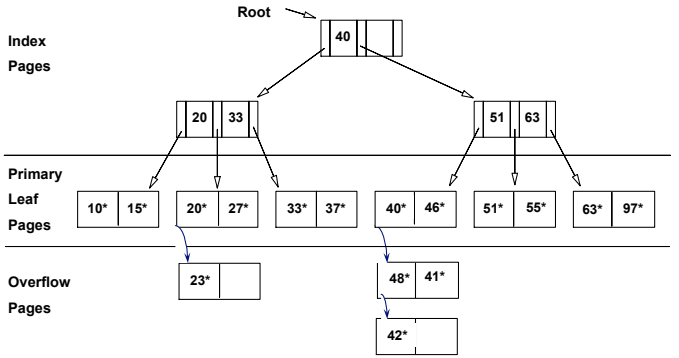
Insert: Find leaf that data entry belongs to, and put it there. Overflow page if necessary.

Delete: Find and remove from leaf; if empty page, de-allocate.



CMU SCS

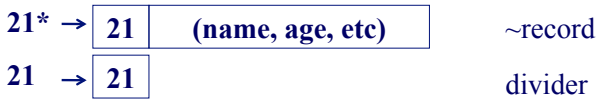
Example: Insert 23*, 48*, 41*, 42*



CMU SCS

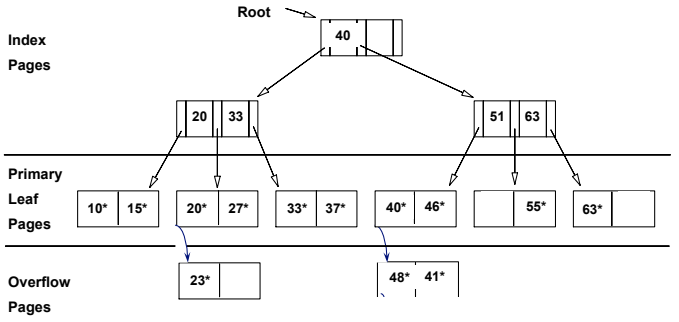
21* means

- <21> + rest of record
- (it's a bit more complicated – but we stay with that, for the moment).
- '21' plain means just 4 bytes, to store integer 21



CMU SCS

... then delete 42*, 51*, 97*



☛ *Note that 51 appears in index levels, but not in leaf!*

CMU SCS

ISAM ---- Issues?

- Pros
  - ????
- Cons
  - ????

CMU SCS

Outline

- Motivation
- ISAM
- **B-trees (not in book)**
- B+ trees
- duplicates
- B+ trees in practice

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

CMU SCS

B-trees

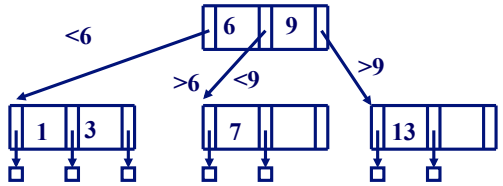
[Rudolf Bayer and McCreight, E. M. Organization and Maintenance of Large Ordered Indexes. Acta Informatica 1, 173-189, 1972.]



CMU SCS

B-trees

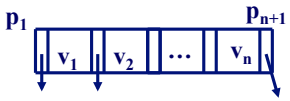
Eg., B-tree of order d=1:



CMU SCS

B - tree properties:

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



CMU SCS

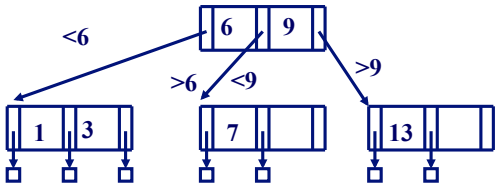
Properties

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

CMU SCS

Queries

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



CMU SCS

JAVA animation!

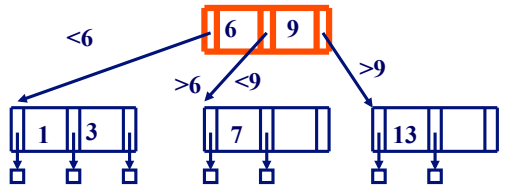
http://slady.net/java/bt/

strongly recommended! (with all usual pre-cautions – VM etc)

CMU SCS

Queries

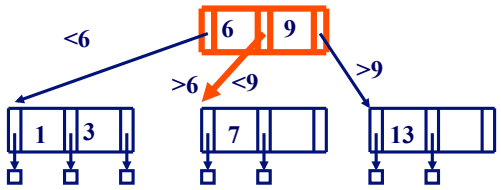
- Algo for exact match query? (eg., ssn=8?)



CMU SCS

Queries

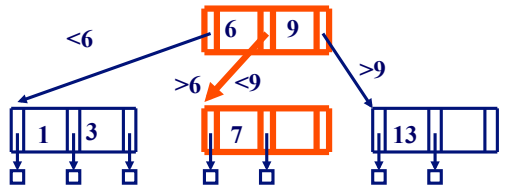
- Algo for exact match query? (eg., ssn=8?)



CMU SCS

Queries

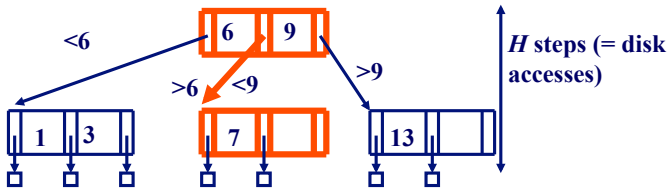
- Algo for exact match query? (eg., ssn=8?)



CMU SCS

Queries

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



CMU SCS

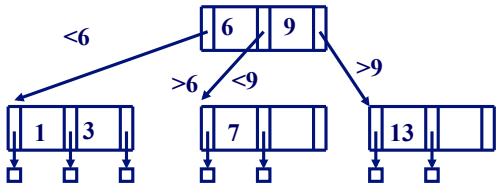
Queries

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

CMU SCS

Queries

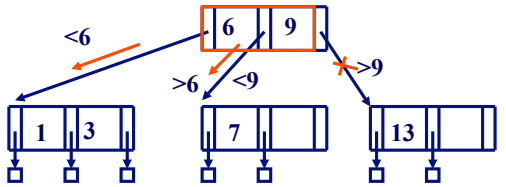
- what about range queries? (eg., ***5<salary<8***)
- Proximity/ nearest neighbor searches? (eg., *salary ~ 8*)



CMU SCS

Queries

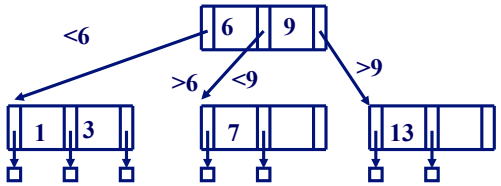
- what about range queries? (eg., ***5<salary<8***)
- Proximity/ nearest neighbor searches? (eg., *salary ~ 8*)



CMU SCS

Queries

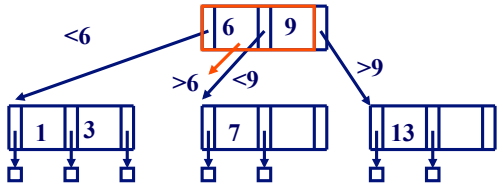
- what about range queries? (eg., *5<salary<8*)
- Proximity/ nearest neighbor searches? (eg., ***salary ~ 8***)



CMU SCS

Queries

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



CMU SCS

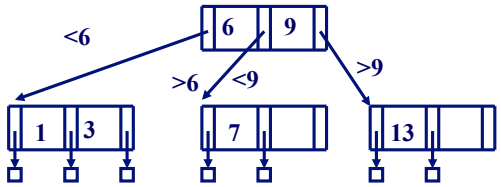
B-trees: Insertion

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

CMU SCS

B-trees

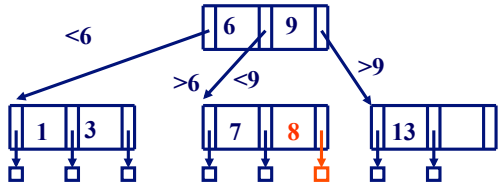
Easy case: Tree T0; insert '8'



CMU SCS

B-trees

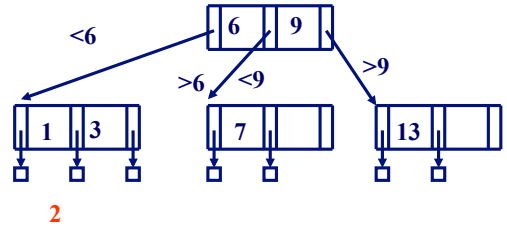
Tree T0; insert '8'



CMU SCS

B-trees

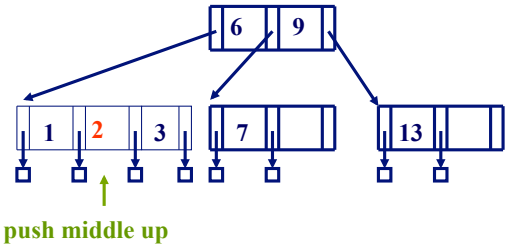
Hardest case: Tree T0; insert '2'



CMU SCS

B-trees

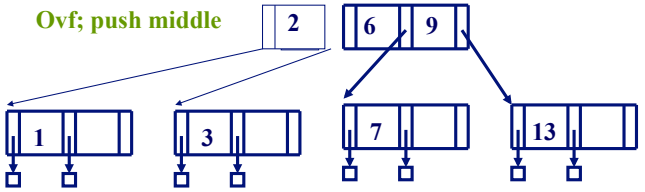
Hardest case: Tree T0; insert '2'



CMU SCS

B-trees

Hardest case: Tree T0; insert '2'

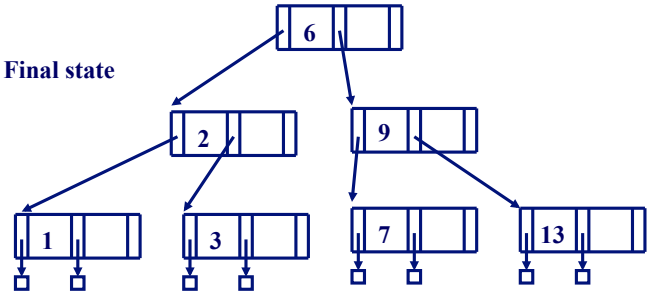


CMU SCS

B-trees

Hardest case: Tree T0; insert '2'

Final state



CMU SCS

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 (contrast with ISAM!)

CMU SCS

Pseudo-code

```
INSERTION OF KEY 'K'
    find the correct leaf node 'L';
    if ( 'L' overflows ){
        split 'L', and push middle key to parent node 'P';
        if ('P' overflows){
            repeat the split recursively; }
    else{
        add the key 'K' in node 'L';
        /* maintaining the key order in 'L' */ }
```


CMU SCS

Overview

- ...
- B – trees
  - Dfn, Search, insertion, **deletion**
- ...

CMU SCS

Deletion

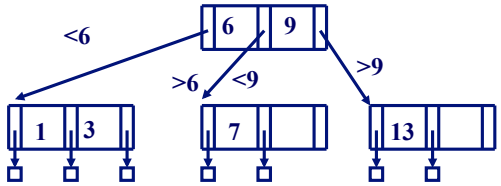
Rough outline of algo:
- Delete key;
- on underflow, may need to merge

In practice, some implementors just allow underflows to happen…

CMU SCS

B-trees – Deletion

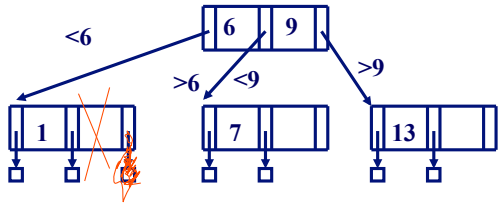
Easiest case: Tree T0; delete '3'



CMU SCS

B-trees – Deletion

Easiest case: Tree T0; delete '3'



CMU SCS

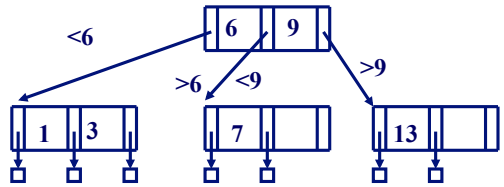
B-trees – Deletion

- 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'

CMU SCS

B-trees – Deletion

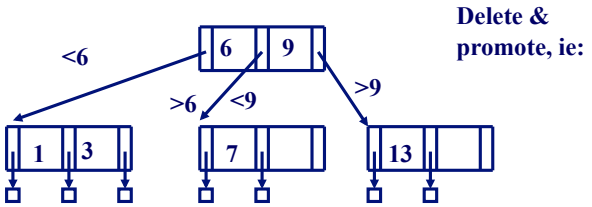
- Case1: delete a key at a leaf – no underflow (delete 3 from T0)



CMU SCS

B-trees – Deletion

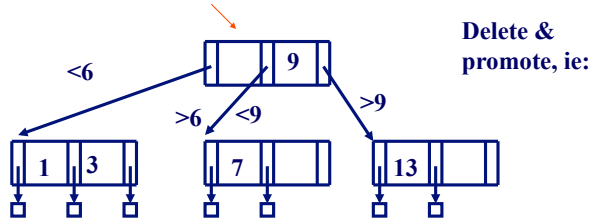
- Case2: delete a key at a non-leaf – no underflow (eg., delete 6 from T0)



CMU SCS

B-trees – Deletion

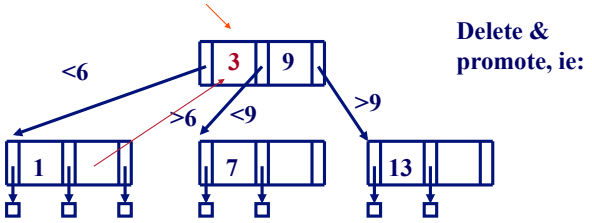
- Case2: delete a key at a non-leaf – no underflow (eg., delete 6 from T0)



CMU SCS

B-trees – Deletion

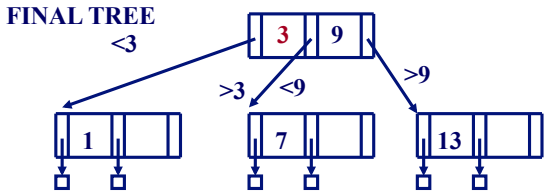
- Case2: delete a key at a non-leaf – no underflow (eg., delete 6 from T0)



CMU SCS

B-trees – Deletion

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



CMU SCS

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

CMU SCS

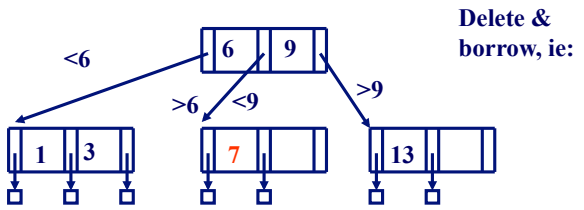
B-trees – Deletion

- 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'

CMU SCS

B-trees – Deletion

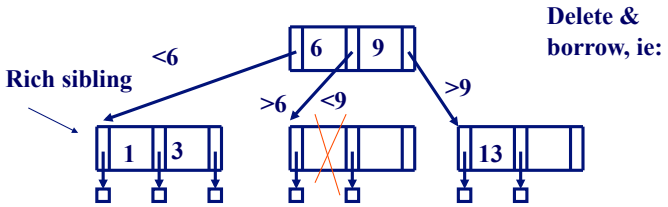
- Case3: underflow & 'rich sibling' (eg., delete 7 from T0)



CMU SCS

B-trees – Deletion

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



CMU SCS

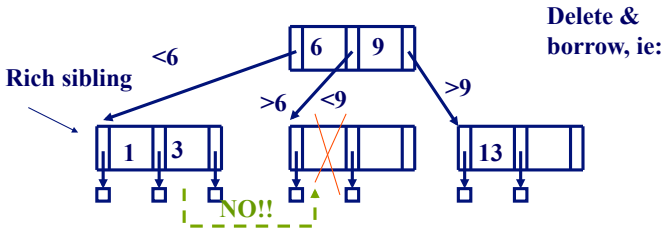
B-trees – Deletion

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

CMU SCS

B-trees – Deletion

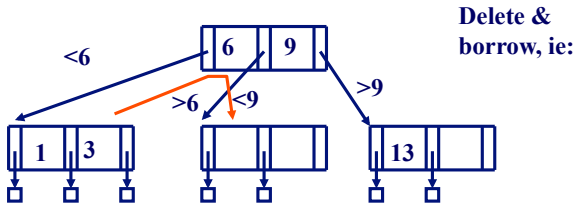
- Case3: underflow & 'rich sibling' (eg., delete 7 from T0)



CMU SCS

B-trees – Deletion

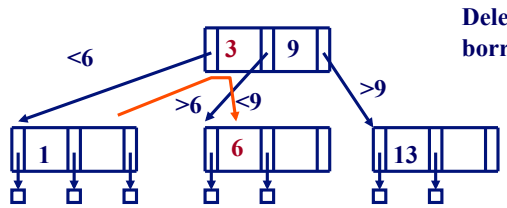
- Case3: underflow & 'rich sibling' (eg., delete 7 from T0)



CMU SCS

B-trees – Deletion

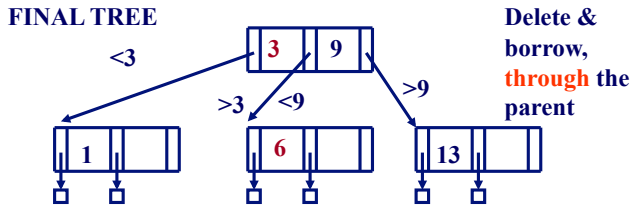
- Case3: underflow & 'rich sibling' (eg., delete 7 from T0)



CMU SCS

B-trees – Deletion

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



CMU SCS

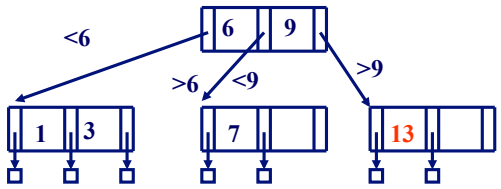
B-trees – Deletion

- 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'

CMU SCS

B-trees – Deletion

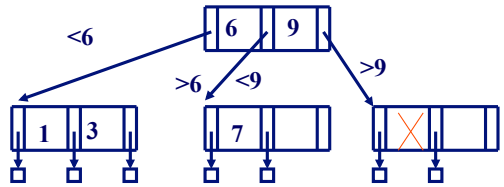
- Case4: underflow & 'poor sibling' (eg., delete 13 from T0)



CMU SCS

B-trees – Deletion

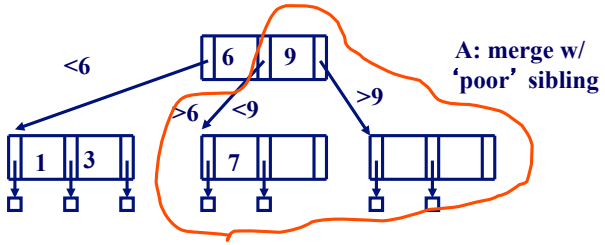
- Case4: underflow & 'poor sibling' (eg., delete 13 from T0)



CMU SCS

B-trees – Deletion

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



CMU SCS

B-trees – Deletion

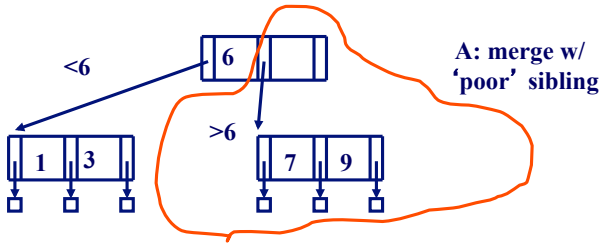
- Case4: underflow & 'poor sibling' (eg., delete 13 from T0)

- Merge, by pulling a key from the **parent**
- exact reversal from insertion: 'split and push up', vs. 'merge and pull down'
- Ie.:

CMU SCS

B-trees – Deletion

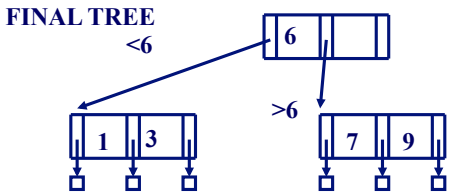
- Case4: underflow & 'poor sibling' (eg., delete 13 from T0)



CMU SCS

B-trees – Deletion

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



CMU SCS

B-trees – Deletion

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

CMU SCS

B-tree deletion - pseudocode

```
DELETION OF KEY 'K'
  locate key 'K', in node 'N'
  if( 'N' is a non-leaf node) {
    delete 'K' from 'N';
    find the immediately largest key 'K1';
      /* which is guaranteed  to be on a leaf node 'L' */
    copy 'K1' in the old position of 'K';
    invoke this DELETION routine on 'K1' from the leaf node 'L';
  else {
/* 'N' is a leaf node */
... (next slide..)
```


CMU SCS

B-tree deletion - pseudocode

```
/* 'N' is a leaf node */
    if( 'N' underflows ){
      let 'N1' be the sibling of 'N';
      if( 'N1' is "rich"){  /* ie., N1 can lend us a key */
        borrow a key from 'N1' THROUGH the parent node;
      }else{   /* N1 is 1 key away from underflowing */
        MERGE: pull the key from the parent 'P',
          and merge it with the keys of 'N' and 'N1' into a new
  node;
        if( 'P' underflows){ repeat recursively }
      }
  }
```

CMU SCS

Outline

- Motivation
- ISAM
- B-trees (not in book)
  - algorithms
  - **extensions**
- B+ trees
- duplicates
- B+ trees in practice

CMU SCS

Variations

- How could we do even better than the B-trees above?

CMU SCS

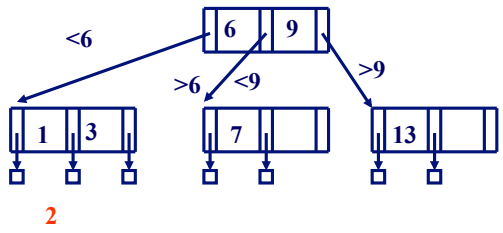
B*-tree

- In B-trees, worst case util. = 50%, if we have just split all the pages
- how to increase the utilization of B - trees?
- ..with B* - trees!

CMU SCS

B-trees and B*-trees

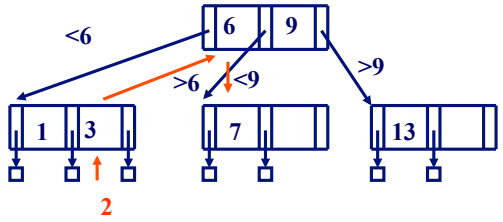
Eg., Tree T0; insert '2'



CMU SCS

B*-trees: deferred split!

- Instead of splitting, LEND keys to sibling! (through PARENT, of course!)

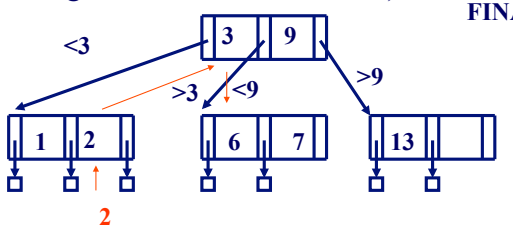


CMU SCS

B*-trees: deferred split!

- Instead of splitting, LEND keys to sibling! (through PARENT, of course!)

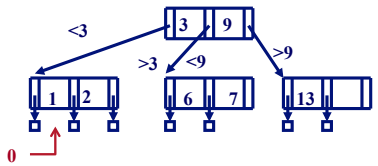
FINAL TREE



CMU SCS

B*-trees: deferred split!

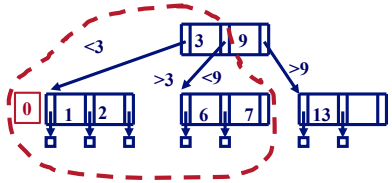
- Notice: shorter, more packed, faster tree
- It's a rare case, where space utilization and speed improve together
- BUT: What if the sibling has no room for our 'lending'?



CMU SCS

B*-trees: deferred split!

- 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.
- Could we extend the idea to 3-to-4 split, 4-to-5 etc?



CMU SCS

B*-trees: deferred split!

- 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.
- Could we extend the idea to 3-to-4 split, 4-to-5 etc?
- Yes, but: diminishing returns

CMU SCS

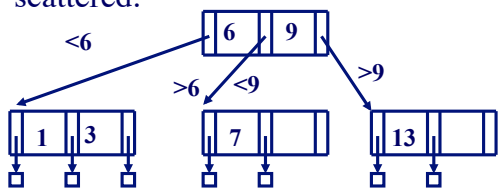
Outline

- Motivation
- ISAM
- B-trees (not in book)
- **B+ trees**
- duplicates
- B+ trees in practice

CMU SCS

B+ trees - Motivation

For clustering index, data records are scattered:



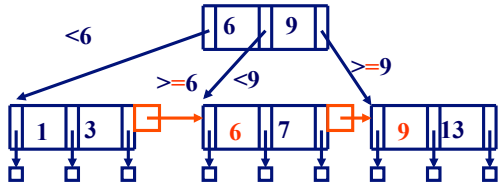
CMU SCS

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
- (vital, for clustering index!)

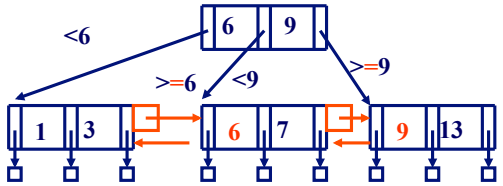
CMU SCS

B+ trees



CMU SCS

B+ trees



CMU SCS

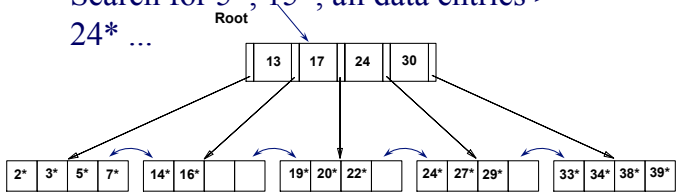
B+trees

- More details: next (and textbook)
- In short: on split
  - at leaf level: **COPY** middle key upstairs
  - at non-leaf level: push middle key upstairs (as in plain B-tree)

CMU SCS

Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...



Based on the search for 15, we know it is not in the tree!*

CMU SCS

B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 2*100*0.67 = 134
- Typical capacities:
  - Height 4: 133^4 = 312,900,721 entries
  - Height 3: 133^3 = 2,406,104 entries

CMU SCS

B+ Trees in Practice

- Can often keep top levels in buffer pool:
  - Level 1 = 1 page = 8 KB
  - Level 2 = 134 pages = 1 MB
  - Level 3 = 17,956 pages = 140 MB

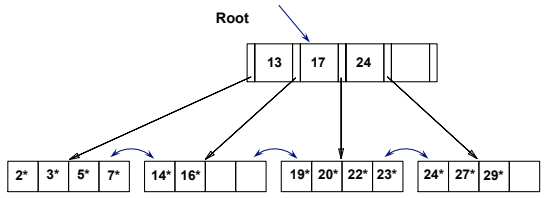
CMU SCS

Inserting a Data Entry into a B+ Tree

- Find correct leaf *L*.
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
- parent node may overflow
  - but then: **push up** middle key. Splits "grow" tree; root split increases height.

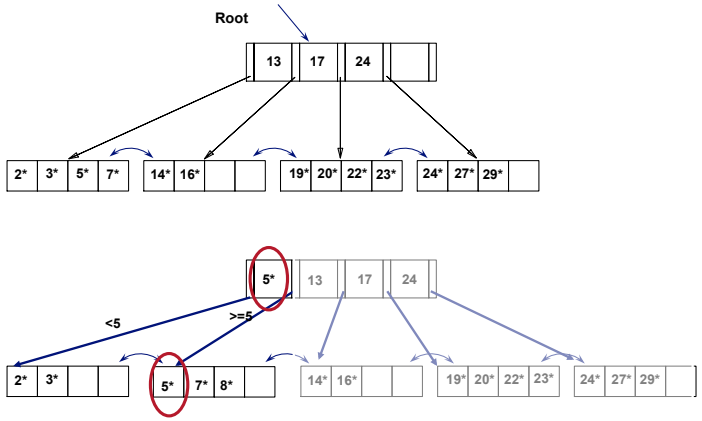
CMU SCS

Example B+ Tree - Inserting 8*



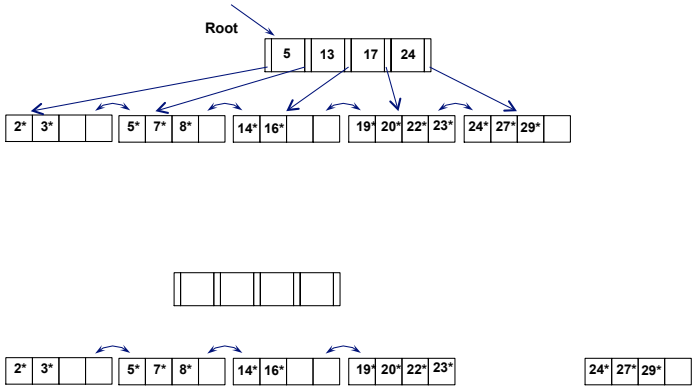
CMU SCS

Example B+ Tree - Inserting 8*



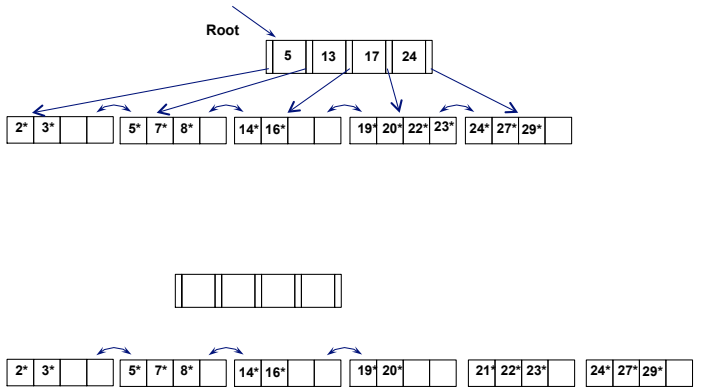
CMU SCS

Example B+ Tree - Inserting 21*



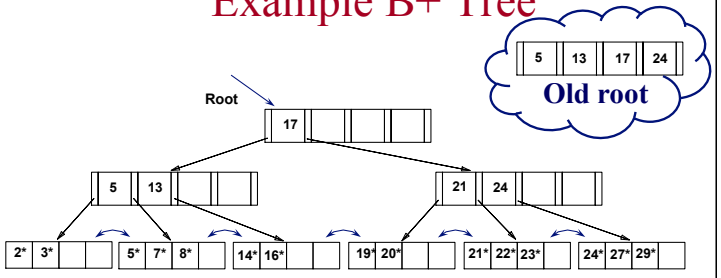
CMU SCS

Example B+ Tree - Inserting 21*



CMU SCS

Example B+ Tree

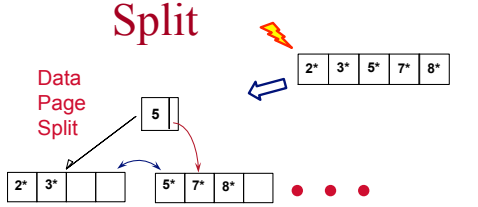


- Notice that root was split, increasing height.
- Could use defer-split here. (Pros/Cons?)

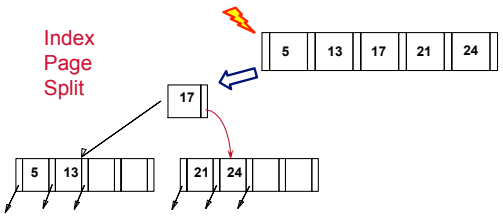
CMU SCS

Example: Data vs. Index Page Split

- leaf: 'copy'
- non-leaf: 'push'

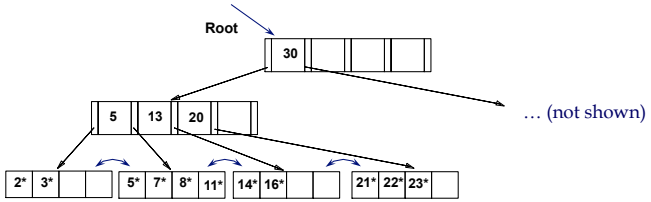


- why not 'copy' @ non-leaves?



CMU SCS

Now you try…

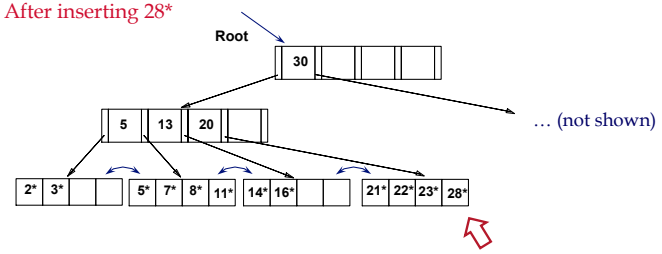


Insert the following data entries (in order): 28*, 6*, 25*

CMU SCS

Now you try…

After inserting 28*

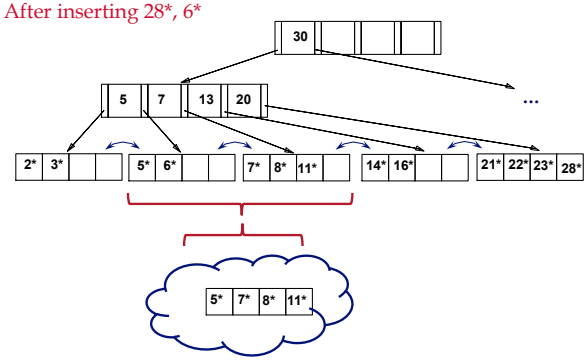


Insert the following data entries (in order): 28*, 6*, 25*

CMU SCS

Answer…

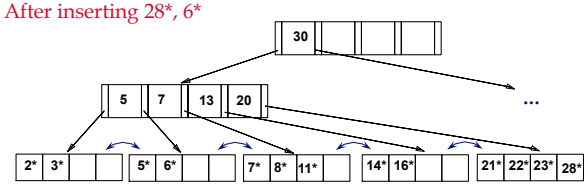
After inserting 28*, 6*



CMU SCS

Answer…

After inserting 28*, 6*

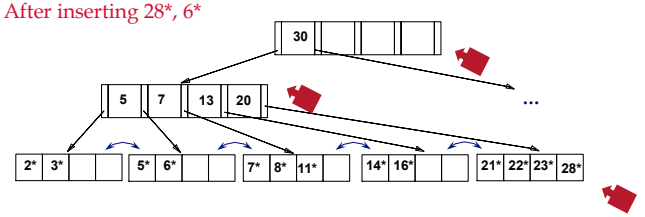


insert 25*:
Q1: which pages will be affected:
Q2: how will the root look like after that?

CMU SCS

Answer…

After inserting 28*, 6*



insert 25*:
Q1: which pages will be affected:
Q2: how will the root look like after that?

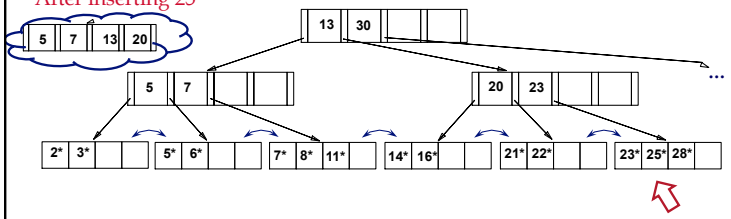
A1: red arrows

A2: (13; 30; _ ; _)

CMU SCS

Answer…

After inserting 25*



25* causes propagated split!

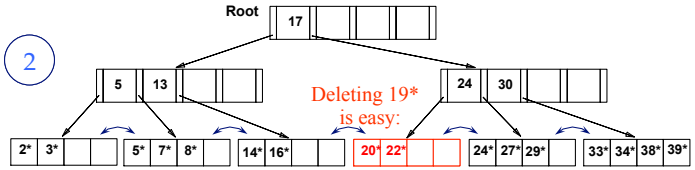
CMU SCS

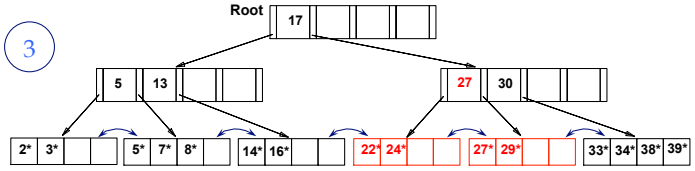
Deleting a Data Entry from a B+ Tree

- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L underflows
    - Try to re-distribute, borrowing from *sibling (adjacent node with same parent as L)*.
    - If re-distribution fails, *merge* *L* and sibling.
      - update parent
      - and possibly merge, recursively

CMU SCS

Example: Delete 19* & 20*

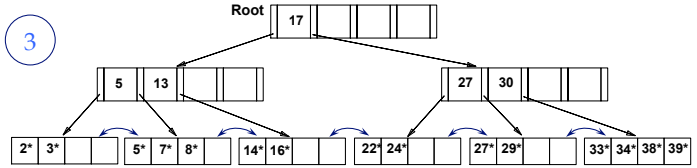


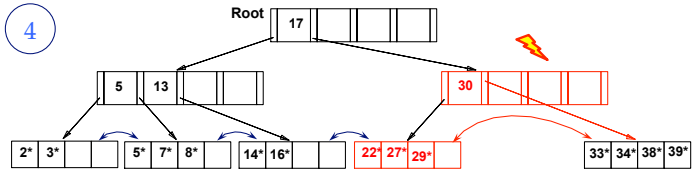


- Deleting 20* -> re-distribution (notice: 27 copied up)

CMU SCS

... And Then Deleting 24*

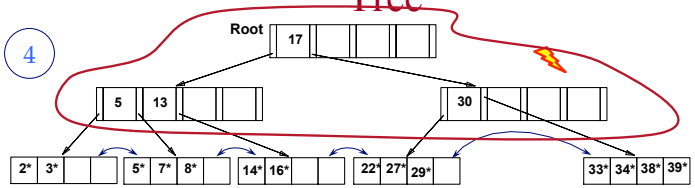


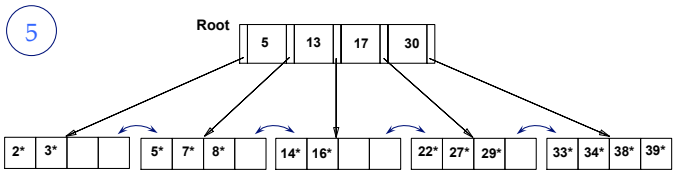


- Must merge leaves … but are we done??

CMU SCS

... Merge Non-Leaf Nodes, Shrink Tree

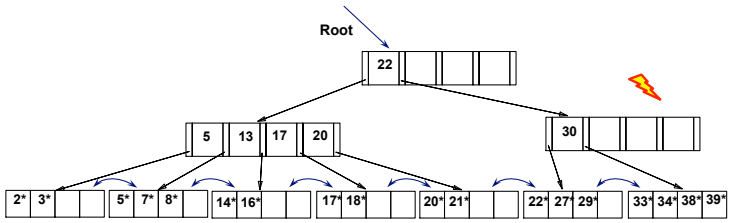




CMU SCS

Example of Non-leaf Re-distribution

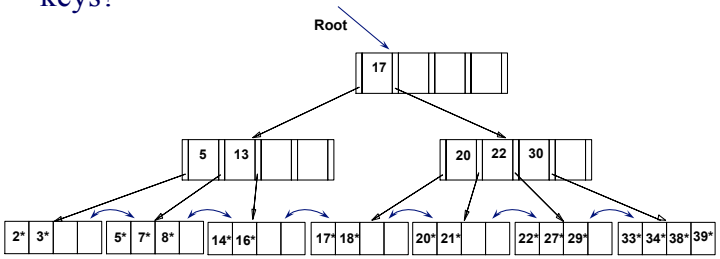
- Tree is shown below *during deletion* of 24*.
- Now, we **can** (and **must**) re-distribute keys



CMU SCS

After Re-distribution

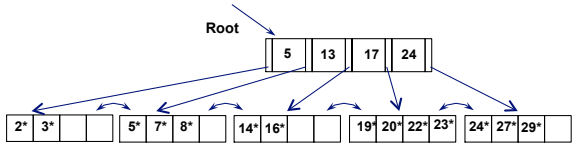
- need only re-distribute '20'; did '17', too
- why would we want to re-distributed more keys?



CMU SCS

Main observations for deletion

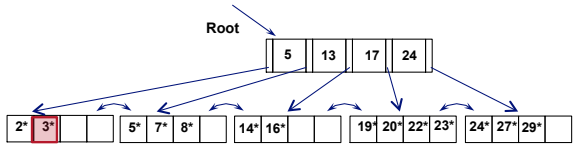
- If a key value appears twice (leaf + nonleaf), the above algorithms delete it from the leaf, only
- why not non-leaf, too?



CMU SCS

Main observations for deletion

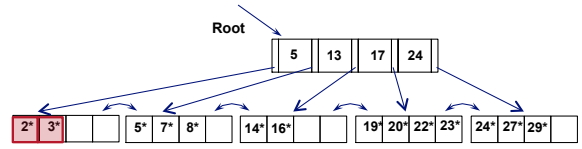
- 'lazy deletions' - in fact, some vendors just mark entries as deleted (~ underflow),
  - and reorganize/compact later



CMU SCS

Main observations for deletion

- 'lazy deletions' - in fact, some vendors just mark entries as deleted (~ underflow),
  - and reorganize/compact later

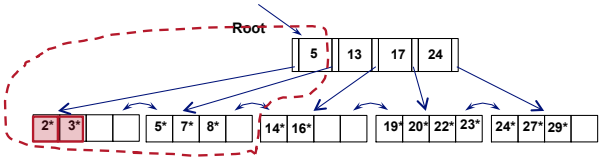


Q: Now, what?

CMU SCS

Main observations for deletion

- 'lazy deletions' - in fact, some vendors just mark entries as deleted (~ underflow),
  - and reorganize/compact later



Q: Now, what? A: 'Merge'

CMU SCS

Recap: main ideas

- on overflow, split (and 'push', or 'copy')
  - or consider deferred split
- on underflow, borrow keys; or merge
  - or let it underflow...

CMU SCS

Outline

- Motivation
- ISAM
- B-trees (not in book)
- B+ trees
- **duplicates**
- B+ trees in practice
  - prefix compression; bulk-loading; 'order'

CMU SCS

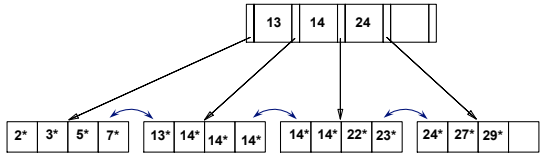
B+ trees with duplicates

- Everything so far: assumed unique key values
- How to extend B+-trees for duplicates?
  - Alt. 2: <key, rid>
  - Alt. 3: <key, {rid list}>
- 2 approaches, roughly equivalent

CMU SCS

B+ trees with duplicates

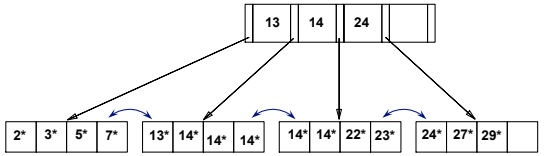
- approach#1: repeat the key values, and extend B+ tree algo's appropriately - eg. many '14's



CMU SCS

B+ trees with duplicates

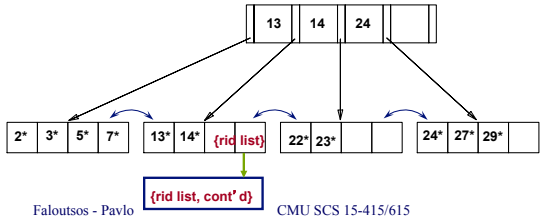
- approach#1: subtle problem with deletion:
- treat *rid* as part of the key, thus making it unique



CMU SCS

B+ trees with duplicates

- approach#2: store each key value: once
- but store the {rid list} as variable-length field (and use overflow pages, if needed)

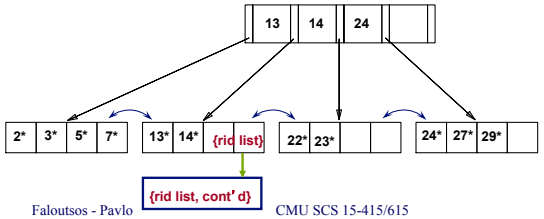


CMU SCS

B+ trees with duplicates

- approach#2: store each key value: once
- but store the {rid list} as variable-length field (and use overflow pages, if needed)

Used in HW3



CMU SCS

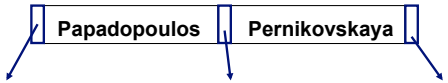
Outline

- Motivation
- ISAM
- B-trees (not in book)
- B+ trees
- duplicates
- B+ trees in practice
  - **prefix compression**; bulk-loading; 'order'

CMU SCS

Prefix Key Compression

- Important to increase fan-out. (Why?)
- Key values in index entries only `direct traffic'; can often compress them.



CMU SCS

Prefix Key Compression

- Important to increase fan-out. (Why?)
- Key values in index entries only `direct traffic'; can often compress them.



CMU SCS

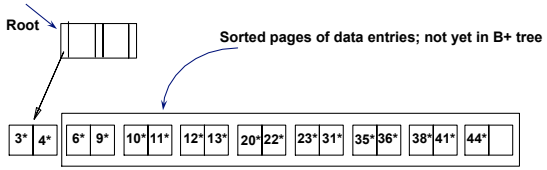
Bulk Loading of a B+ Tree

- In an empty tree, insert many keys
- Why not one-at-a-time?

CMU SCS

Bulk Loading of a B+ Tree

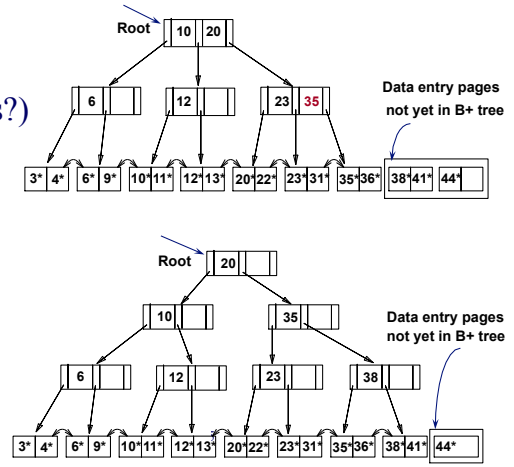
- *Initialization*: Sort all data entries
- scan list; whenever enough for a page, pack
- <repeat for upper level - even faster than book's algo>



CMU SCS

Bulk Loading (Contd.)

- Book's algo
- (any problems?)



CMU SCS

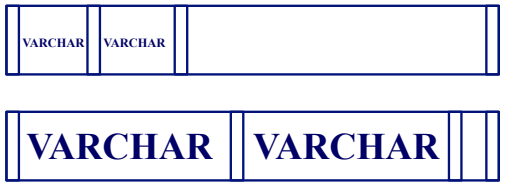
Outline

- Motivation
- ISAM
- B-trees (not in book)
- B+ trees
- duplicates
- B+ trees in practice
  - prefix compression; bulk-loading; **'order'**

CMU SCS

A Note on `Order'

- *Order* (**d**) concept replaced by physical space criterion in practice (`*at least half-full*').
- Why do we need the distinction?



CMU SCS

A Note on `Order'

- *Order* (**d**) concept replaced by physical space criterion in practice (`*at least half-full*').
- Why do we need it?
  - Index pages can typically hold many more entries than leaf pages.
  - Variable sized records and search keys mean different nodes will contain different numbers of entries.
  - Even with fixed length fields, multiple records with the same search key value (*duplicates*) can lead to variable-sized data entries (if we use Alternative (3)).

CMU SCS

A Note on `Order'

- Many real systems are even sloppier than this: they allow **underflow**, and only reclaim space when a page is **completely** empty.
- (what are the benefits of such 'slopiness'?)

CMU SCS

Conclusions

- B+tree is the **prevailing** indexing method
- **Excellent**, O(logN) worst-case performance for ins/del/search; (~3-4 disk accesses in practice)
- guaranteed 50% space utilization; avg 69%

CMU SCS

Conclusions

- Can be used for **any** type of index: primary/ secondary, sparse (clustering), or dense (non-clustering)
- Several fine-tuning extensions on the basic algorithm
  - deferred split; prefix compression; (underflows)
  - bulk-loading
  - duplicate handling