Extra: B+ Trees

CS1: Java Programming
Colorado State University

Slides by Wim Bohm and Russ Wakefield

Motivations

Many times you want to minimize the disk accesses while doing a search. A binary search tree allows two keys per node – a B+ tree allows as many values that will fit on a page.

2

Differences between BST and B+

- B+ is a balanced tree
  - Same distance for every path through the tree
  - Not true for BST
- B+ tree keeps data only in the leaf nodes
  - The index nodes are simply for traffic control
  - BST has data with the keys
- B+ tree has many keys per node
  - BST has 1

3

Objectives

- To understand the basic structure of a B+ Tree

- Understanding how the way keys are stored allows for efficient searching

- To understand the insertion algorithm

- To understand the deletion algorithm

- To see how rotation minimizes the cost of insertion and deletion algorithms

4

What is a B+ tree?

- B+ tree – a dynamic structure that adjusts to changes in the file gracefully. It is the the most widely used structure because it adjusts well to changes and supports both equality and range queries.
- It is a balanced tree in which the internal nodes direct the search and the leaf nodes contain the data entries. The leaf nodes are organized into a doubly linked list allowing us to easily traverse the leaf pages in either direction.

5

Example of a B+ tree

6

2
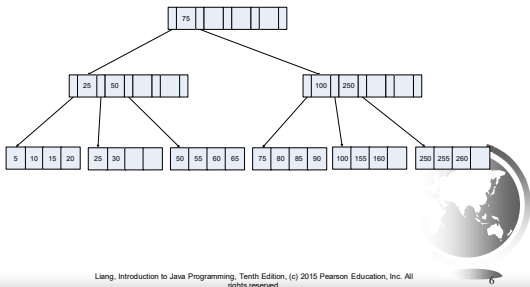
Advantages / Disadvanges

- Advantage of B[+]-tree index files:
  - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B[+]-trees:
  - extra insertion and deletion overhead, space overhead.
- Advantages of B[+]-trees outweigh disadvantages
  - B[+]-trees are used extensively

B[+]-Tree Index Files (Cont.)

A B[+]-tree is a rooted tree satisfying the following properties:
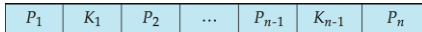
- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n-1)$ values.

B[+]-Tree Node Structure

- Typical node

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n

  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered
$$K_1 < K_2 < K_3 < . . . < K_{n-1}$$
(Initially assume no duplicate keys, address duplicates later)

Non-Leaf Nodes in B⁺-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \le i \le n-1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
  - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n

Example of B⁺-tree

B⁺-tree for *instructor* file ($n = 5$)
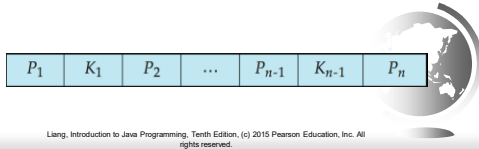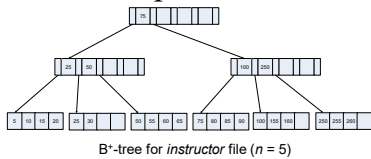
- Leaf nodes must have between 2 and 4 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n = 5$).
- Non-leaf nodes other than root must have between 3 and 5 children ($\lceil n/2 \rceil$ and n with $n = 5$).
- Root must have at least 2 children.

Observations about B⁺-trees

- Operations (insert, delete) on the tree keep it balanced. Log_fN cost where f=fanout, N = # of leaf pages.
- Minimum occupancy of 50% is guaranteed for each node except the root node if the deletion algorithm we will present is used. (in practice, deletes just delete the data entry because files usually grow, not shrink). Each node contains m entries where d <= m <= 2d entries. d is referred to as the **order** of the tree.
- Search for a record is just a traversal from the root to the appropriate leaf. This is the height of the tree – because it is balanced is consistent Because of the high fan-out, the height of a B+ tree is rarely more than 3 or 4.

Queries on B$^+$-Trees

☐ Find record with search-key value V.
1. $C=root$
2. While C is not a leaf node {
   1. Let i be least value such that $V \leq K_i$.
   2. If no such exists, set $C = $ *last non-null pointer in C*
   3. Else { if ($V= K_i$) Set $C = P_{i+1}$ else set $C = P_i$}
   } //Now C is in leaf node containing K_i
3. Let i be least value such that $K_i = V$
4. If there is such a value i, follow pointer P_i to the desired record.
5. Else no record with search-key value k exists.

Queries on B$^+$-Trees (Cont.)

☐ If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{n/2}(K) \rceil$. (n = number of indices / node)

☐ A node is generally the same size as a disk block, typically 4 kilobytes
   – and n is typically around 100 (40 bytes per index entry).

☐ With 1 million search key values and $n = 100$
   – at most $log_{50}(1,000,000) = 4$ nodes are accessed in a lookup.

☐ Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
   – above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

Insertion Algorithm

| Leaf page full?	Index page full?	Action
No	No	Place the record in sorted position in the appropriate leaf page
Yes	No	1. Split the leaf page including the inserted record. 2. Place Middle Key in the index page in sorted order. 3. Left leaf page contains records with keys below the middle key. 4. Right leaf page contains records with keys equal to or greater than the middle key.
Yes	Yes	1. Split the leaf page including the inserted record. 2. Records with keys < middle key go to the left leaf page. 3. Records with keys >= middle key go to the right leaf page. 4. Split the index page. 5. Keys < middle key go to the left index page. 6. Keys > middle key go to the right index page. 7. The middle key goes to the next (higher level) index. IF the next level index page is full, continue splitting the index pages.

15

Insertion

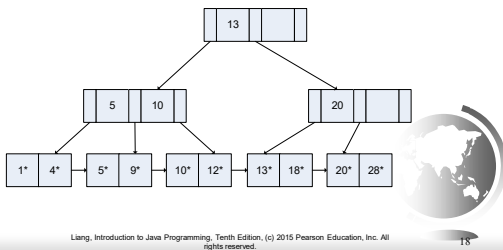Examples of insertion with B+ tree with order = 1. Starting with a tree looking like this:

16
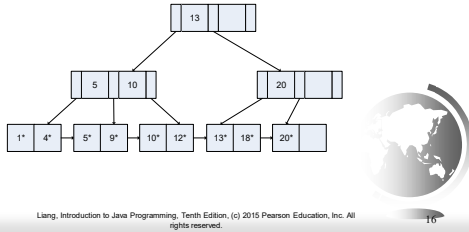
Insert 28
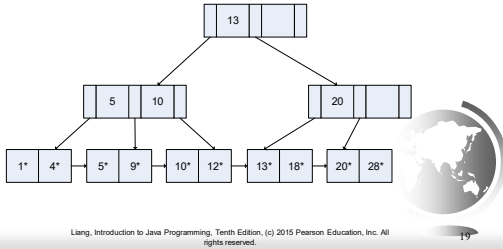
Insert 28 into the below tree:

17
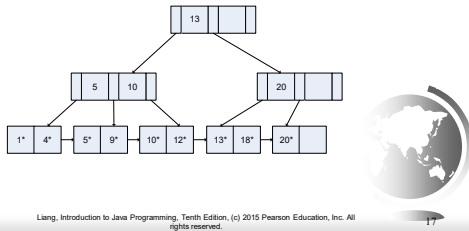
Insert 28

Our first insertion has an index of 28. We look at the leaf node to see if there is room. Finding an empty slot, we place the index in node in sorted order.
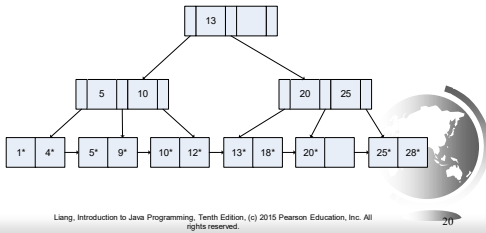
18

Insert 25

Insert 25 in the below tree:

19

Insert 25

Our next insertion is at 25. We look at the leaf node it would go in and find there is no room. We split the node, and roll the middle value to the index mode above it.

20

Insert 8

Insert 8 in the below tree:
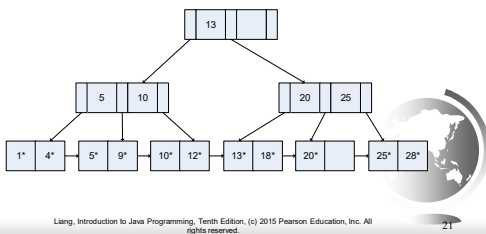
21

Insert 8

Our next case occurs when we want to add 8. The leaf node is full, so we split it and attempt to roll the index to the index node. It is full, so we must split it as well.
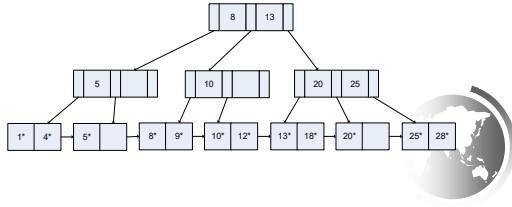
22

Insert 15

Insert 15 into the below tree:
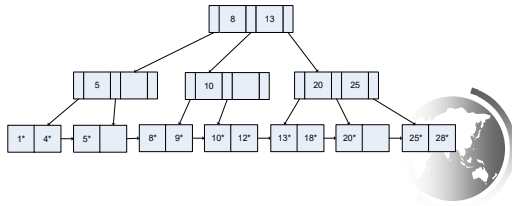
23

Insert 15

Our last case occurs when we want to add 15. This is going to result in the root node being split. The leaf node is full, as are the two index nodes above it. This gives us:
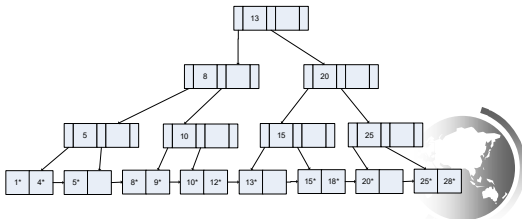
24

Delete Algorithm

Leaf page less than 1/2?	Index page less than 1/2?	Action
No	No	Delete the record from the leaf page. Arrange keys in ascending order to fill void. If the key of the deleted record appears in the index pages, use the next key to replace it.
Yes	No	Combine the leaf page and its sibling. Change the index page to reflect the change.
Yes	Yes	1. Combine the leaf page and its sibling. 2. Adjust the index page to reflect the change. 3. Combine the index page with its sibling 4. Delete entry from parent Continue combining index pages until you reach a page with thecorrect fill factor or you reach the root page.

25

Delete

Let's take our tree from the insert example with a minor modification (we have added 30 to give us an index node with 2 indexes in it:

26
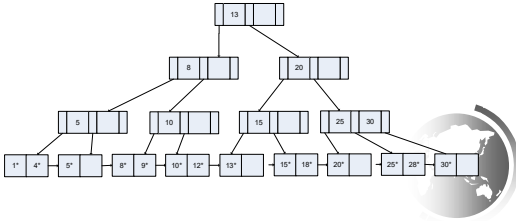
Delete 18

Delete 18 from the below tree:

27

9
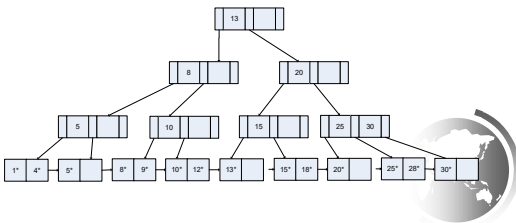
Delete 18

Our first delete is of 18. Simplest case is that it is not an index and in a leaf node that deleting it will not take you below 1/2.
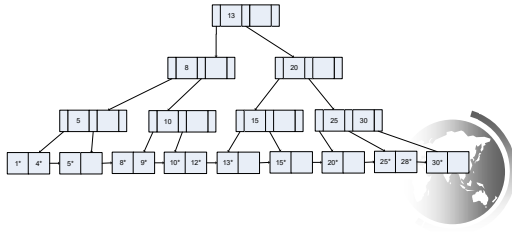
Delete 25
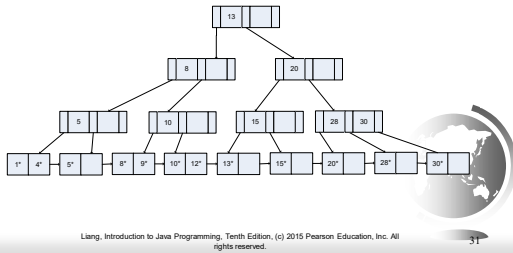
Delete 25 from the below tree:
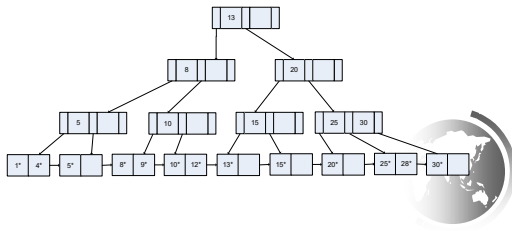
Delete 25

Our next delete is similar, except the index appears in a index node. In that case, the next index replaces the one in the index node. Let's delete 25 from the previous slide.
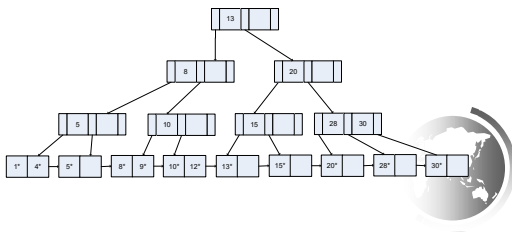
Delete 28

Delete 28 from the below tree:

Delete 28

Our next case takes the node below d. Let's delete 28. For this one we combine the leaf page (in our case it is empty) with its sibling and update the index appropriately. That gives us:
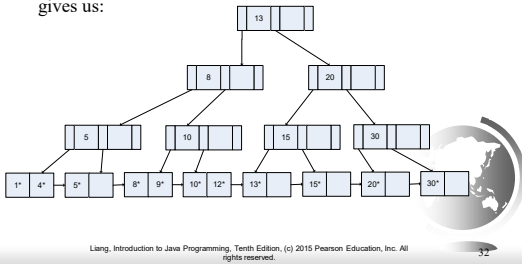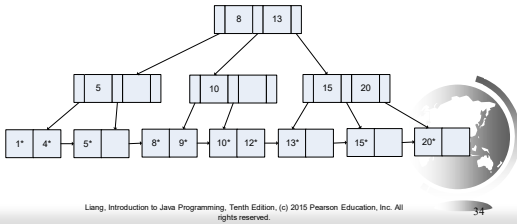
Delete 30

Delete 30 from the below tree:
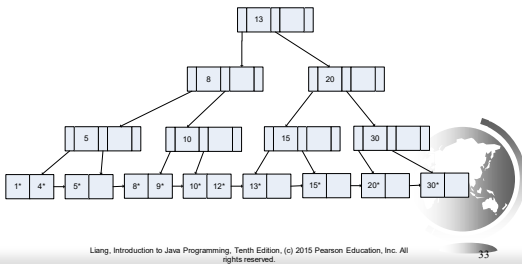
Delete 30

Next we delete 30. This takes us below d for the index. We combine the indexes, which has the effect of taking the index above below d. This continues to the root.
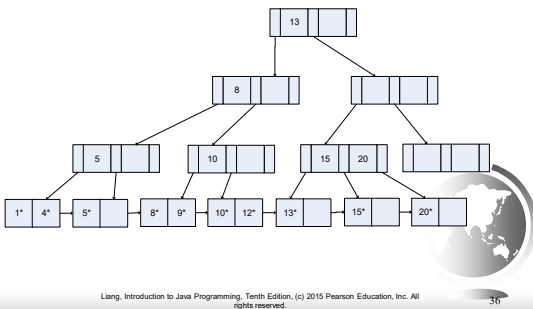
Delete 30

Woah. That seemed like magic. What process got us to that?

Ok – let's go through it. When we deleted 30, which took the data entry node that 30 was in below d. Now we have to merge with the sibling. When we merge – it's to the sibling on the left, which means pointer in the index above is no longer valid. We remove it, (which leaves it less than d), pull down the index from above and merge the index node with its sibling.
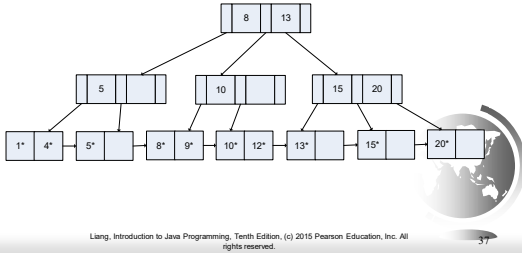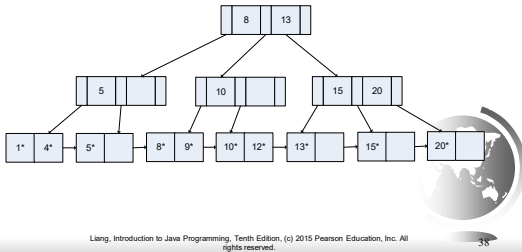
Delete 30

Delete 30
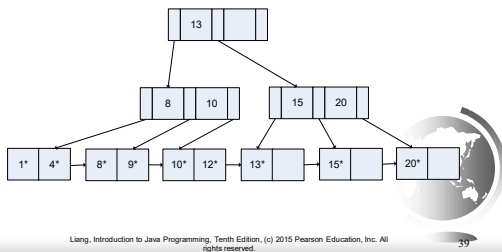
Repeating the process gets us back to

37

Delete 5

Delete 5 from the below tree:

38

Delete 5

Our last example deletes 5. This takes the node and the index above it below d. We remove the leaf node and combine the index with its neighbor.
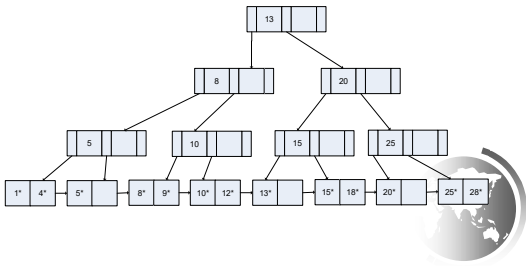
39

13

Rotation

It is also possible to rebalance a tree to reduce the number of splits – called rotation. If you are trying to insert, and a leaf page is full, but its sibling isn't – you can move an index to a sibling and avoid splitting. Let's go back to a tree from our insert example:

40

Add 3

41

Add 3 (Rotation)

We want to add 3 – but in this case we check the sibling to see if it has room. It does, so we move a record to it adjusting the index. Now we have :
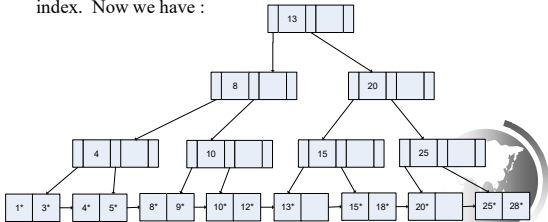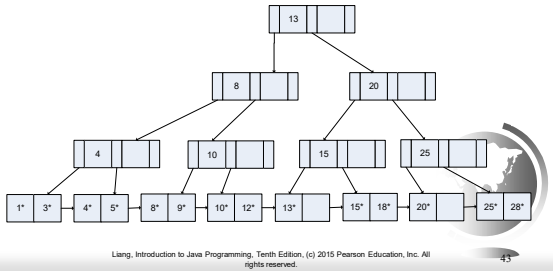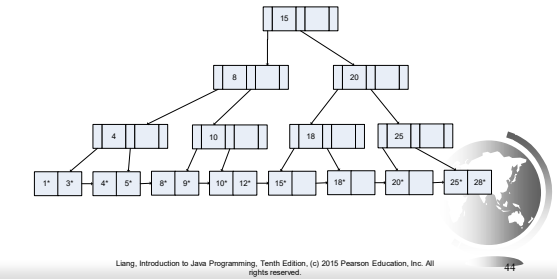
42

Delete 13 (Rotation)
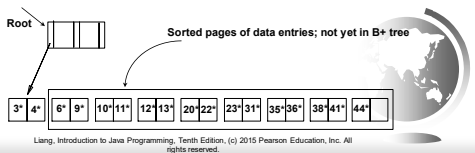
The same concept works with deletes.

Delete 13 (Rotation)

Bulk Loading of a B+ Tree

- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.
- *Bulk Loading* can be done much more efficiently.
- *Initialization*: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.

Bulk Loading (Contd.)

- Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up, it splits. (Split may go up right-most path to the root.)
- Much faster than repeated inserts.
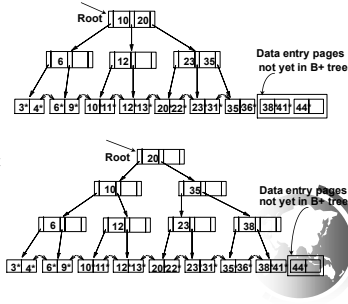
Summary of Bulk Loading

- Option 1: multiple inserts.
  - Slow.
  - Does not give sequential storage of leaves.
- Option 2: *Bulk Loading*
  - Has advantages for concurrency control.
  - Fewer I/Os during build.
  - Leaves will be stored sequentially (and linked, of course).
  - Can control "fill factor" on pages.