Lec 16: Indexed Sequential Files

Last Day: Static Indexed Files.
- Flat indexes
- Multi-level indexes
- Index reorganization

Today: Dynamic Indexed Files.
- Binary Trees
- Multi-Way Trees
- B-trees
- Tree insertions

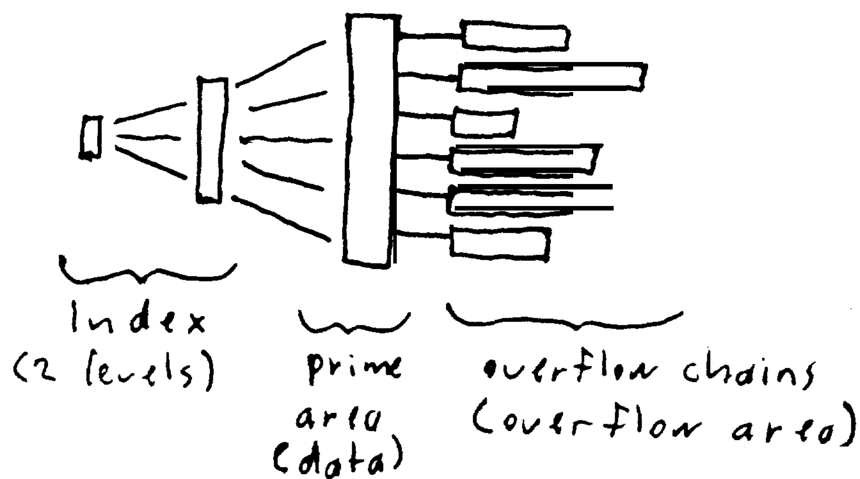
Folk + Zoellick, ch. 8.

Static Indexed Files (Review)

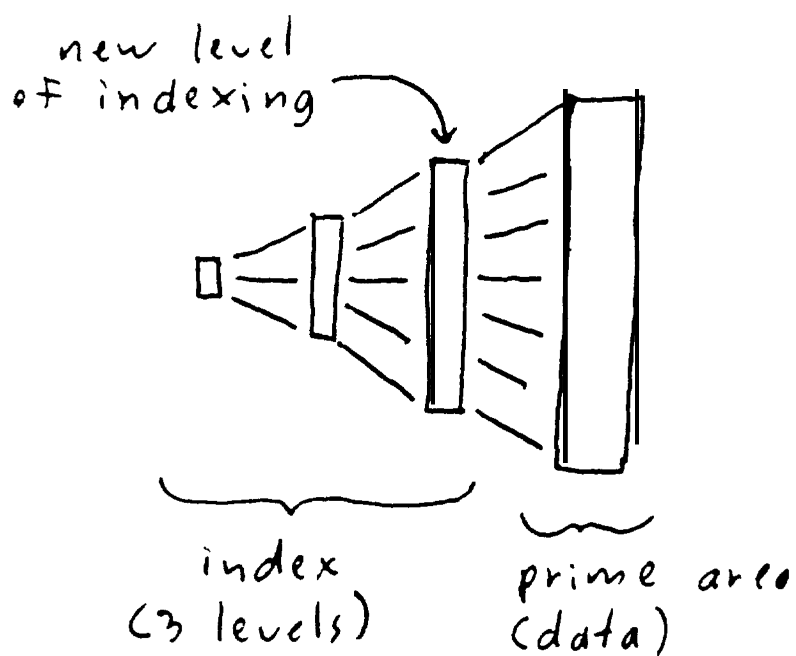
- When the overflow area becomes large, access time increases.
- From time to time, the file and index must be reorganized (expanded).
- ie, The overflow area is "merged" with the prime area, & a new level of indexing is created.
- Note: Reorganizations are infrequent but massive (since the entire data file and index file are affected).

Reorganizing Static Indexed Files

Before Reorganization:



After Reorganization:



Disadvantages of Static Files

- File reorganization is time-consuming.
- Users cannot access the data during reorganization.
- ∴ Must be done after business hours (eg, at night or on weekends)
- ∴ Not suitable for 7×24 operations (7 days a week, 24 hours a day), like airline reservations, ATM banking, etc.

<u>Solution:</u> Dynamic Indexed Files

Dynamic Indexed Files

- Usually implemented as a tree.

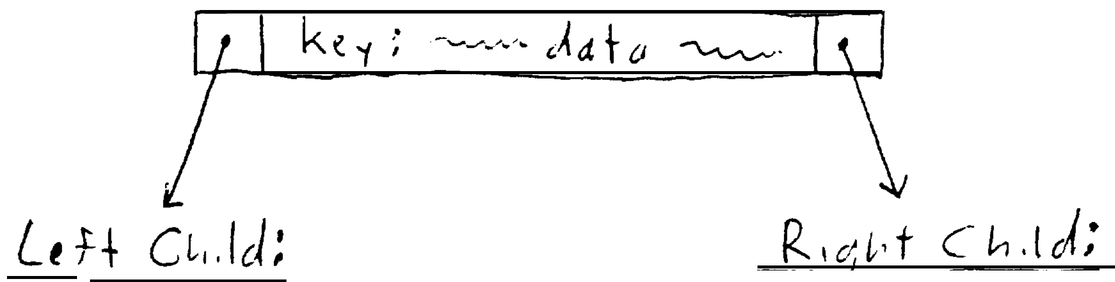
- The tree shape & size adapts incrementally as records are inserted & deleted.

- ∴ Data does not have to be taken off-line during reorganization.

- Typical tree structures:
    - Binary trees
    - AVL trees
    - Multiway trees
    - B trees & variations (B* trees, B+ trees) } very popular

Binary Trees

Each node of the tree contains a record and two pointers, as follows:



Left Child: Leads to all nodes with smaller keys.

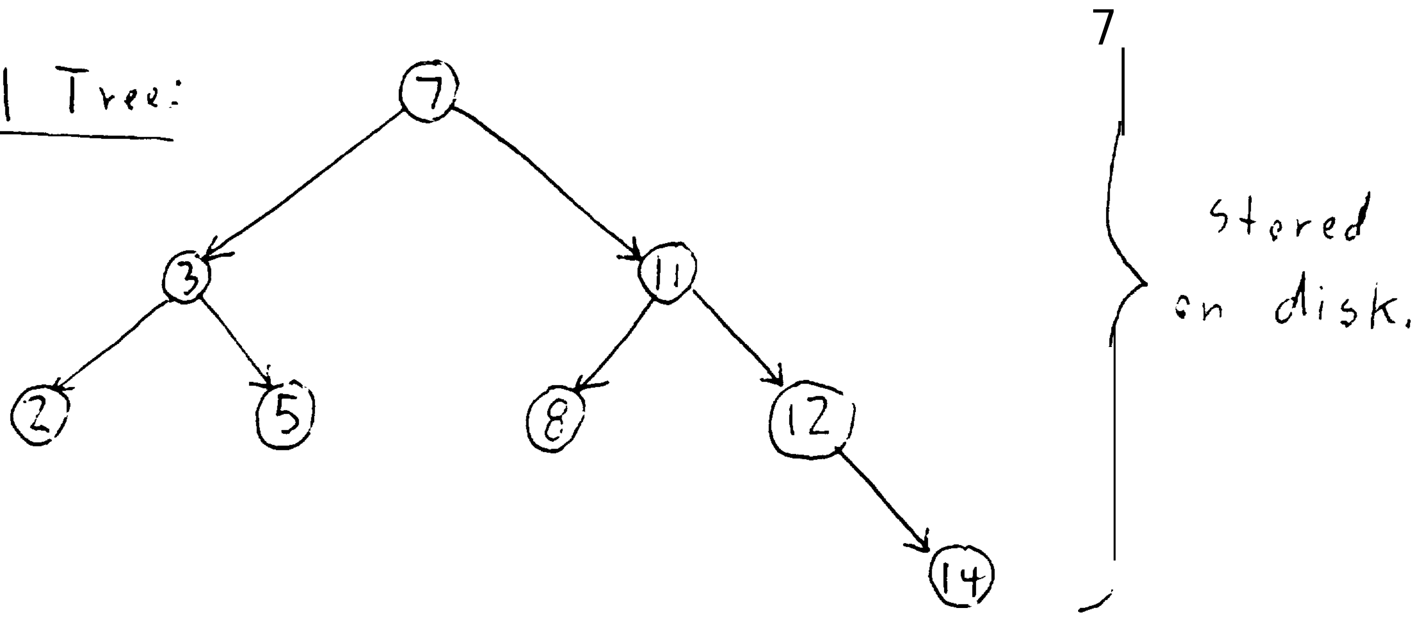
Right Child: Leads to all nodes with larger keys.

Note: Each pointer is just the address of another node.

Growing a Binary Tree: Example

Insert: 7, 3, 2, 11, 5, 8, 12, 14.

Final Tree:



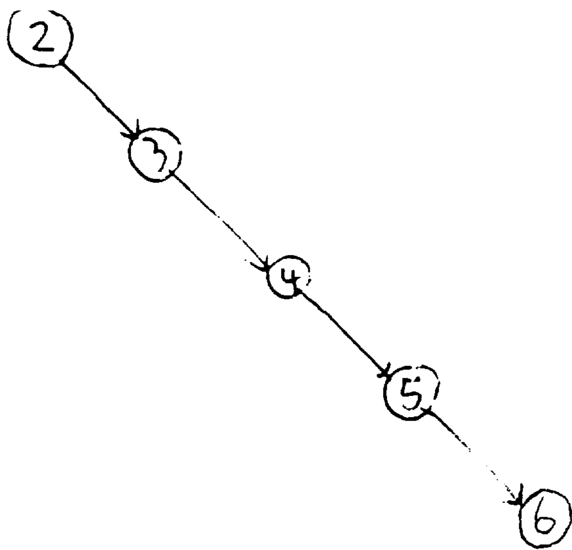
Note: Each node access requires 1 file access

e.g. Retrieve 8: Cost = 3 file accesses.

Retrieve 11: Cost = 2 file accesses.

In general, average retrieval cost = $O(\log_2 N)$

where N = # nodes in tree = # records in file.



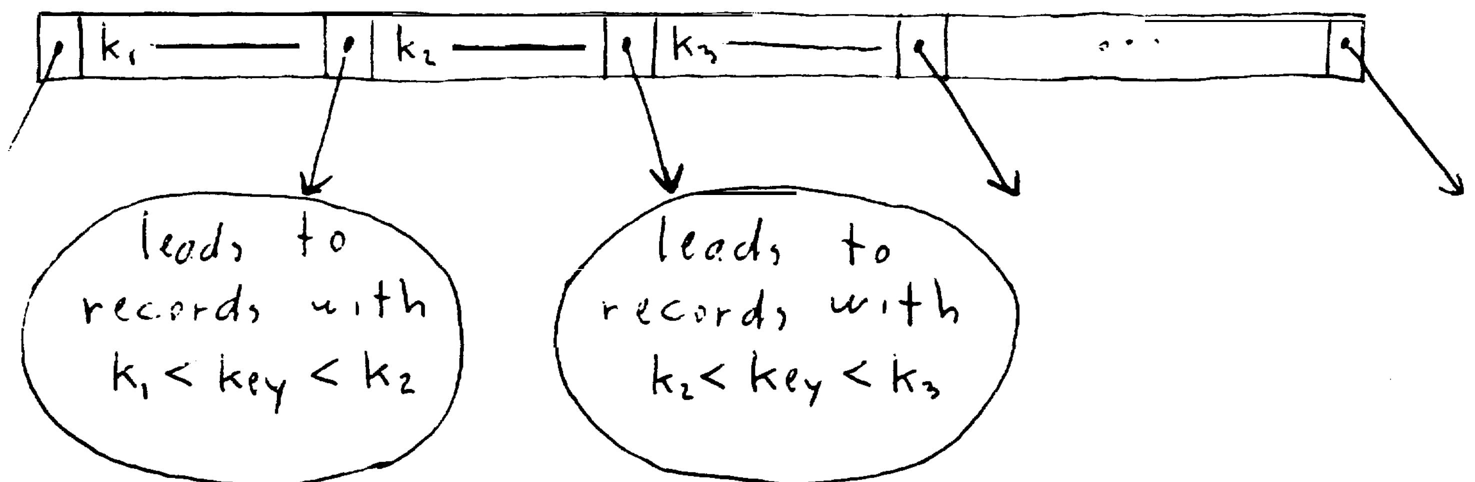
∴ Worst-case retrieval time $= O(N)$.

ie, As bad as sequential files.

(We will return to this problem later.)

Multiway Trees

- A generalization of binary trees.
- Now a node can have up to <u>m</u> pointers and <u>m-1</u> records (i.e., as many as will fit on a disk block).
- Each node has the following form:



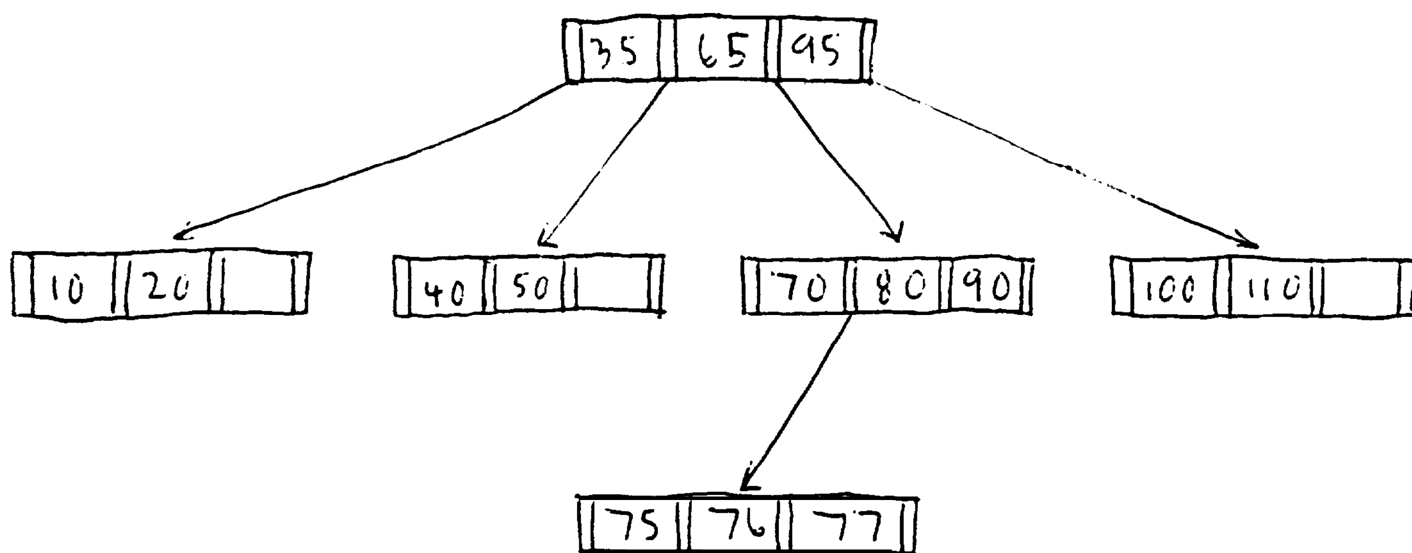
- **Note:** A node with n children (where $n \leq m$) stores n-1 records.

Growing a Multiway Tree: Example

- Suppose m = 4 (ie, at most m children per node)

Insert: 35, 65, 95, 70, 80, 40, 10, 20, 50, 100, 110, 90, 75, 76, 77

- **Final Tree:**



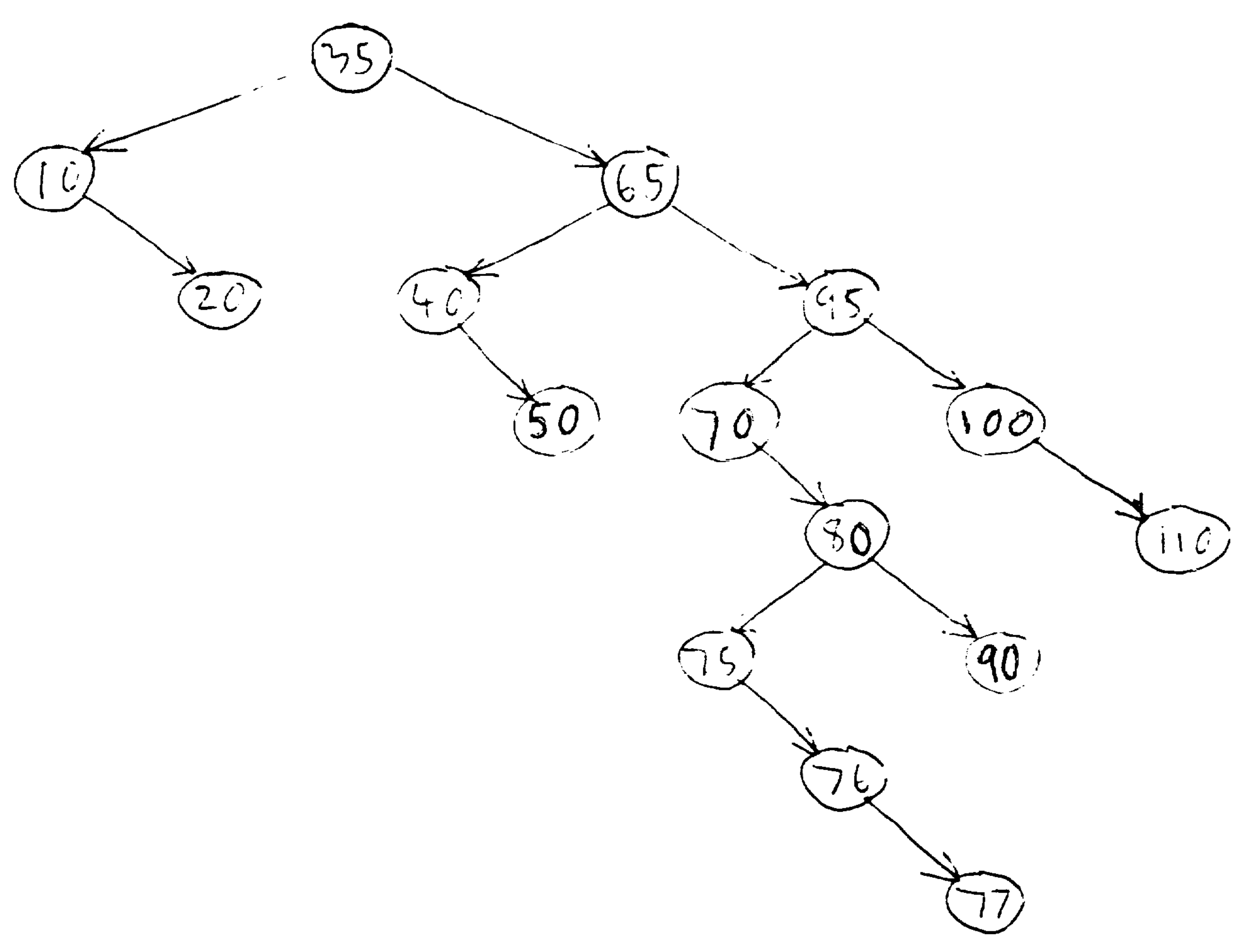
- **Note:** Each node represents 1 file access.

eg. Retrieve 77: Cost = 3 file accesses.

Retrieve 110: Cost = 2 file accesses

Contrast

The same data inserted into a Binary Tree:



Note: The tree is much deeper.

∴ Average access time is greater.

eg, Retrieve 77: Cost = 8 file accesses

Retrieve 110: Cost = 5 file accesses

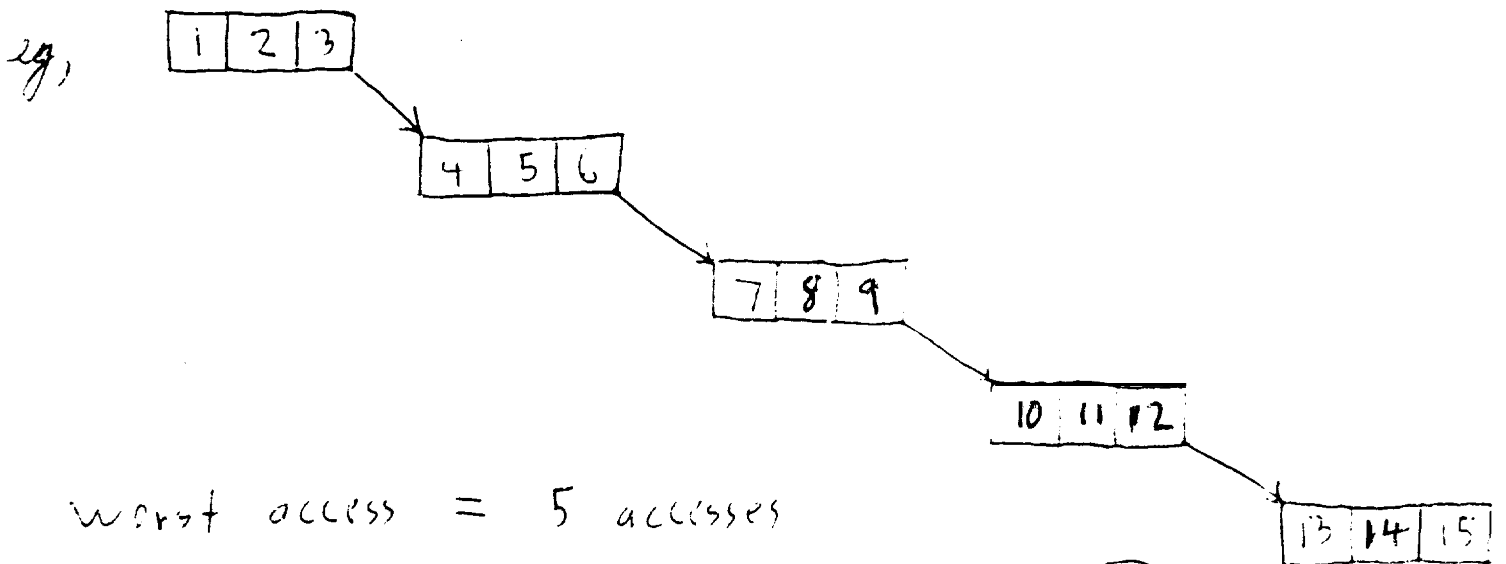
The Moral of the Tale ...

- Binary trees are deeper (and thus slower) than multiway trees.
- ∴ Multiway trees are better for file access.
- <u>Note:</u> Reading a node takes 1 file access whether it is 2-way, 3-way, 4-way ... k-way as long as the node fits on <u>one disk block</u>

Problem: Unbalanced Trees

Multi-way trees may be deep & unbalanced.

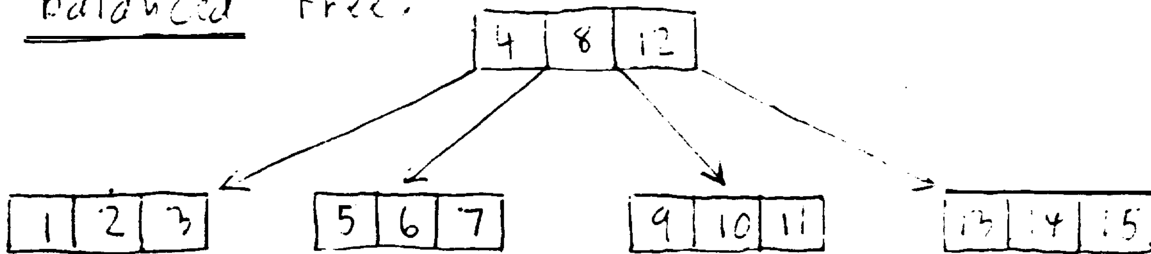
eg,



worst access = 5 accesses

= $O(N)$ in general.

- We would prefer to store the data as a **balanced** tree:



worst access = 2 file accesses

= $O(\log N)$ in general

→

Problem: How to keep trees balanced?

Solution: B-trees (Balanced Trees)

A B-tree of order m is a multiway tree with the following properties:

- Each node has at most m children.

- Each node (except the root) is at least half full, ie, has at least $\lceil m/2 \rceil$ children (This ensures that B-trees are wide & "bushy")

- All leaf nodes are the same distance from the root. (This ensures that B-trees are balanced.)

Note: m = maximum # children per node.

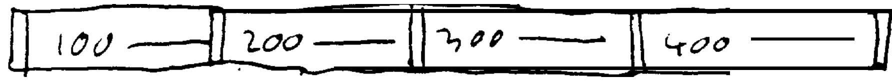
Basic B-tree Operations

- When a node becomes too large, split it into two nodes.
- When a node becomes too small, borrow records from a sibling node (if possible)
- When two sibling nodes become too small, merge them into a single node.

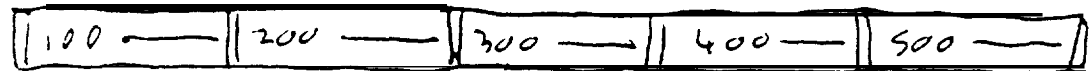
Example

Building a B-tree of order 5.

Insert: 100, 200, 300, 400

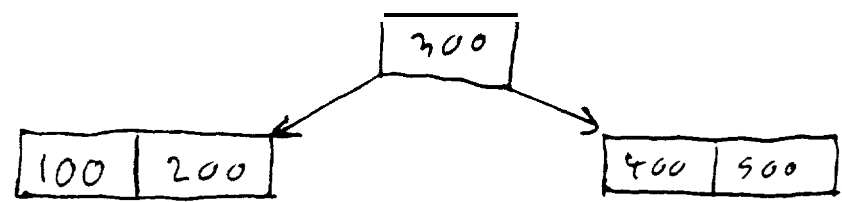


insert: 500

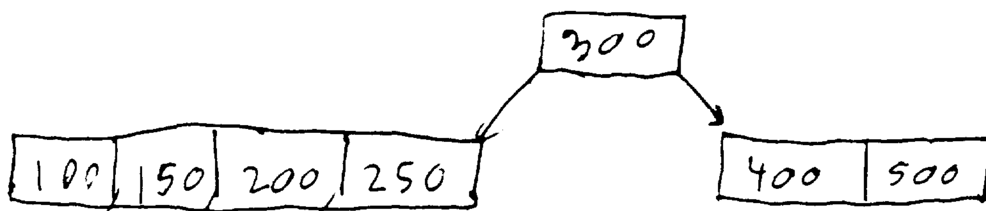


Node is too big! Only 4 records are allowed.

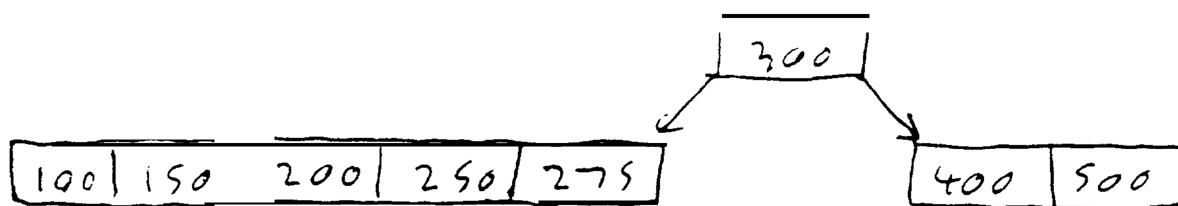
Split the node, creating a new root.



insert: 150, 250

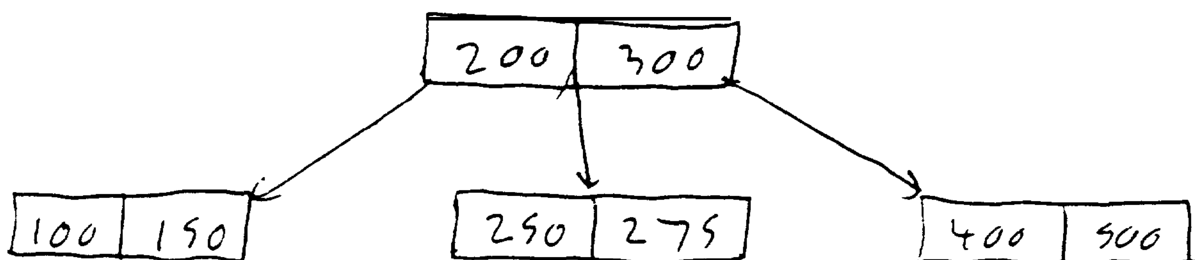


insert: 275

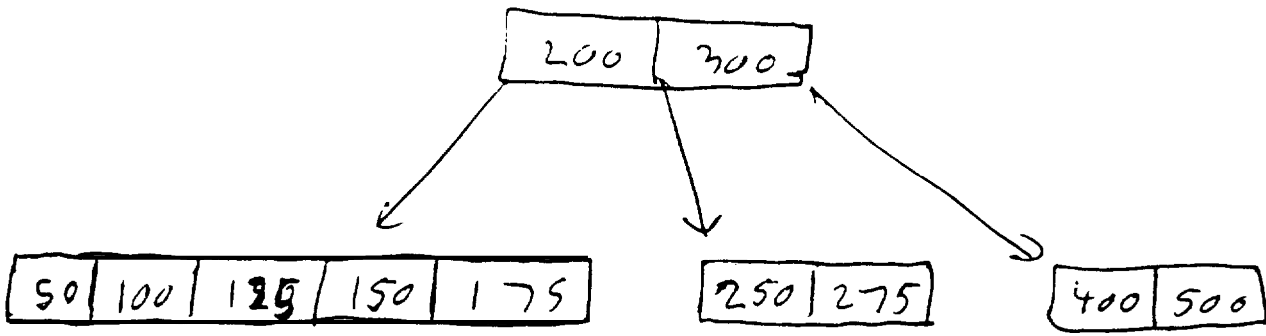


Node is too big! Only 4 records allowed.

Split the node, pushing middle record up, into the root.

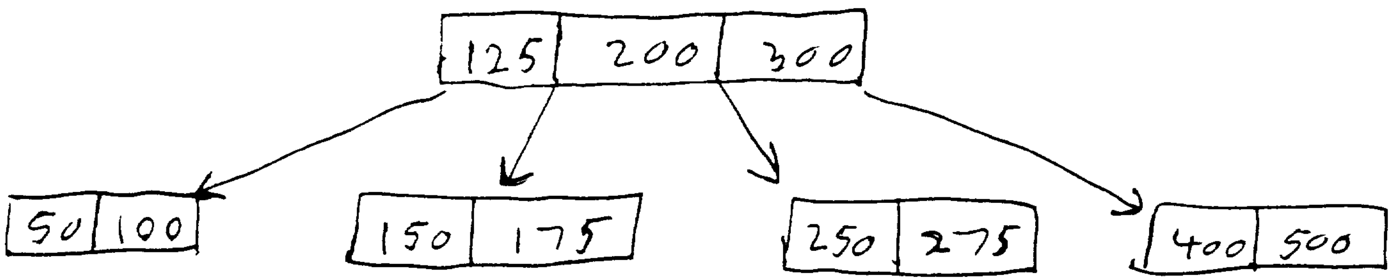


Insert: 50, 125, 175

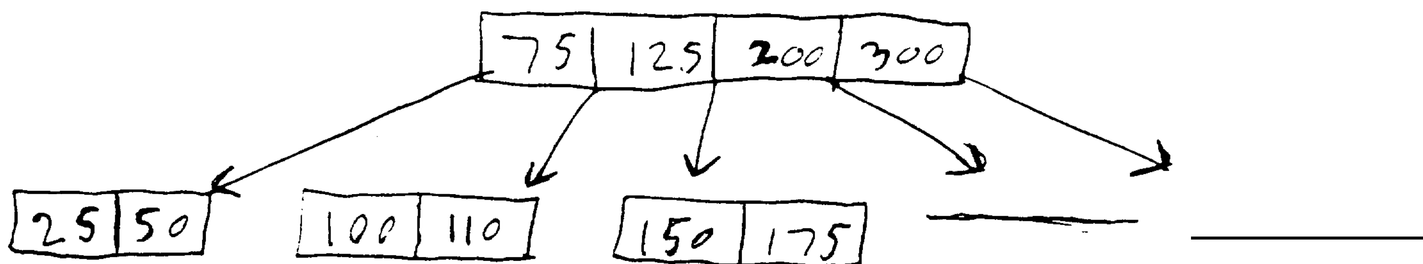


Node is too big!

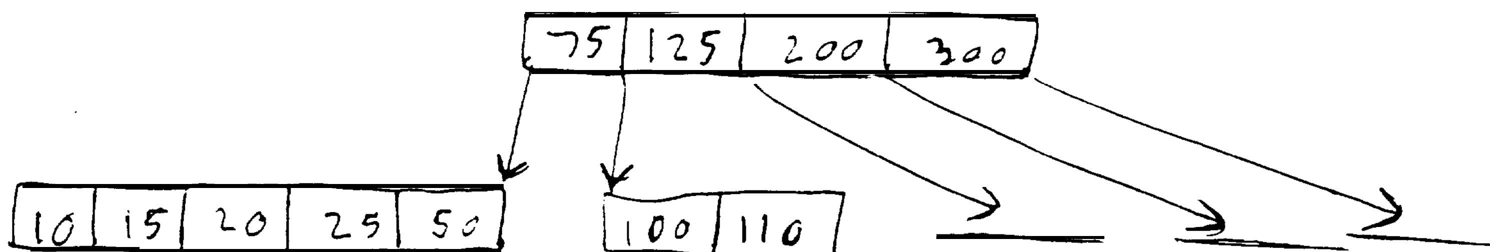
Split, pushing middle record up.



insert: 25, 75, 110.

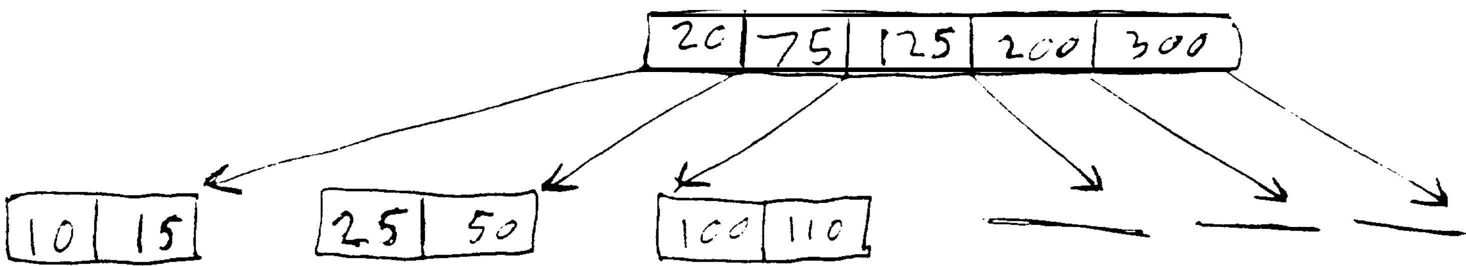


insert 10, 15, 20.



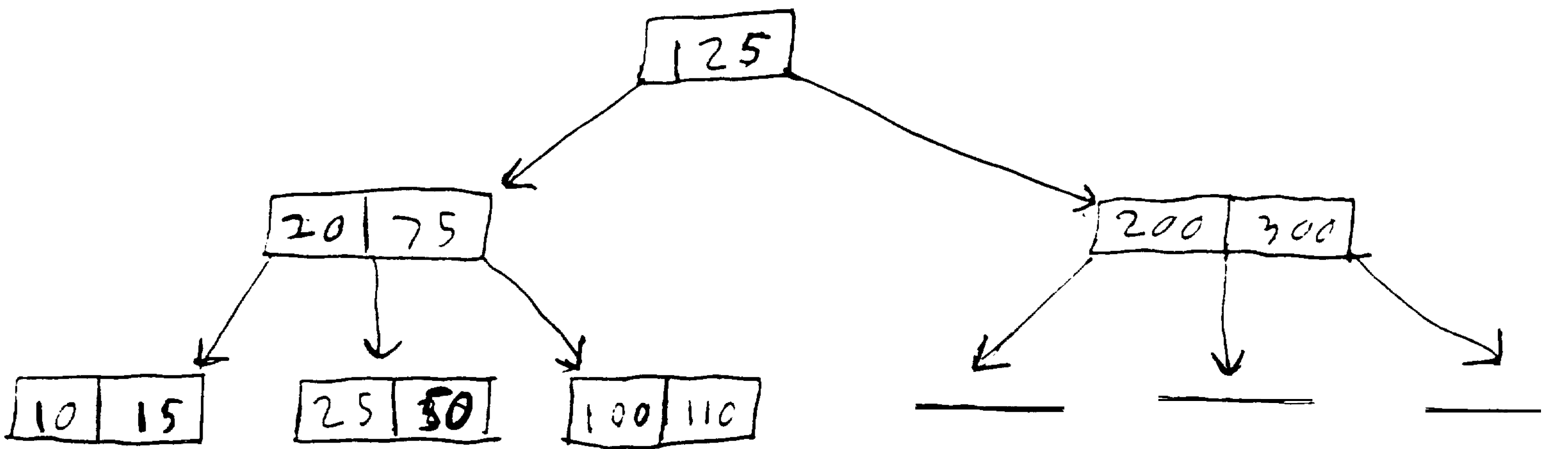
Node is too big!

Split, pushing middle record up.



Root node is **too big!**

Split, pushing middle record up, to create a new root.



Note: all leaves are at the same level (ie, at depth 3).

Note: All nodes are always at least half full (ie, have at least 2 records).