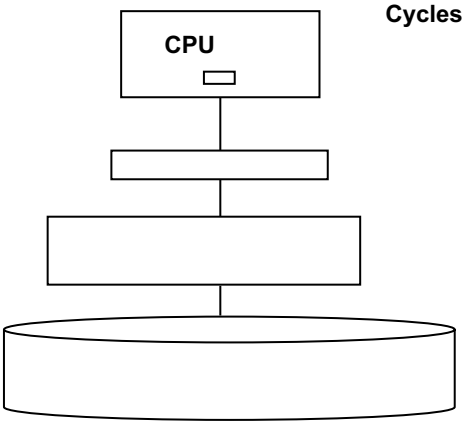
Class 19

B-TREES

Typical Memory Access Costs



	Cycles to access:
Registers	1
Cache	tens
Main memory	hundreds
Disk or SSD	millions

Problem

Implement a huge database or filesystem with key-based search, sequential access, insertions, deletions

Only fits on disk/SSD.

Idea:

- Tree structure acts as an index to data.
- But, each node access is a random disk access.
  - Binary tree, e.g. AVL or red-black, may require a disk read for every node. Wasteful!
  - On the other hand, a disk read fetches a whole block into fast memory. Can we make better use of the whole block?
- How could we lower the number of disk accesses (nodes in a branch)?

B-Trees

History

- Bayer and McCreight, Boeing Research Labs, 1971.
- They did not explain what, if anything, the *B* stands for.

Each node has at most *M* children, and at most *M-1* keys

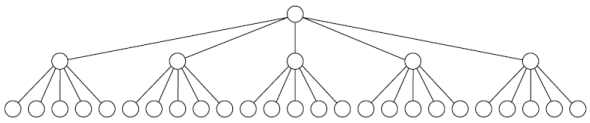


Figure 4.62 5-ary tree of 31 nodes has only three levels Source

Many variants/conventions.

- I am specifically talking today about B+ trees, call them B-trees for simplicity

B-Tree Structure Properties

1. The data items are stored at leaves
2. The root is either a leaf or has between 2 and *M* children
3. The non-leaf nodes store up to *M – 1* keys to guide traversal
   - Key *i* represents the smallest key in subtree *i+1*
4. All non-leaf nodes (except the root) have between ⌈ *M/2* ⌉ and *M* children.
   - # of children = # keys + 1
5. All leaves are at the same depth and have between ⌈ *L/2* ⌉ and *L* data items.

B-Tree Example

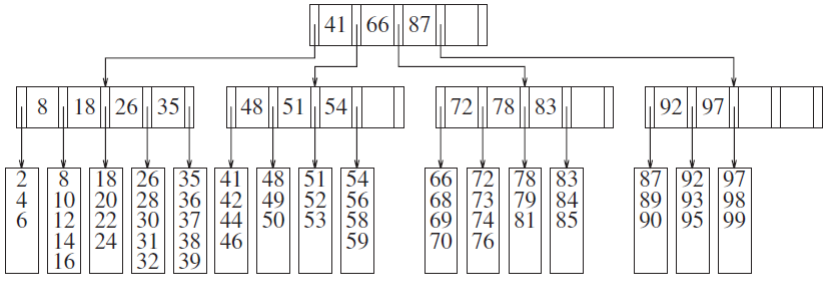


Figure 4.63 B-tree of order 5

Disk Friendliness

What makes B-trees disk-friendly?

1. The tree is "wide and shallow": for 1 billion records,

order *m*	16	32	64	128	256	512	1024
upper bound on height	10	8	6	5	5	4	4

2. Many keys stored at a node
   - Each node is one disk page/block
   - Entire block brought to cache in one disk access.

Insertion and Removal

INSERT(x)

- find correct leaf
- if space in leaf
  - add x to leaf
- else
  - split leaf into two…
  - median element recursively inserted into parent node as splitting key. Base case: at root, create new root.

Other frequently-used techniques:

- Rotations (rebalancing overflowing keys over to siblings)
- Top-down splitting to avoid recursion on the way back

REMOVE(x)

- if leaf containing x > min full
  - delete directly from leaf
- else
  - if sibling > min full
    - adopt from sibling
  - else
    - merge leaf with sibling
    - if parent underflows, percolate up. Base case: replace root with its lone child

Insertion Examples

Insert 57 into this B-tree

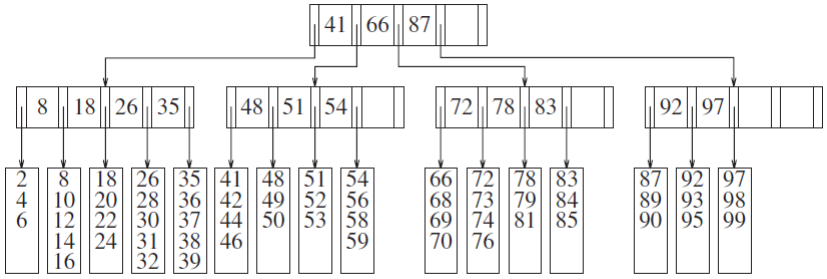


Figure 4.63 B-tree of order 5

Insertion Examples

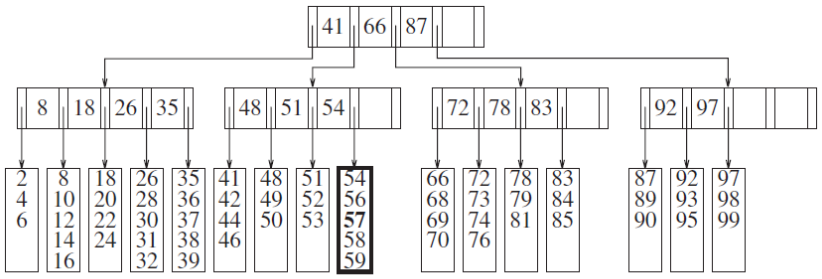


Figure 4.64 B-tree after insertion of 57 into the tree in Figure 4.63

Insertion Examples

Insert 55 into this tree

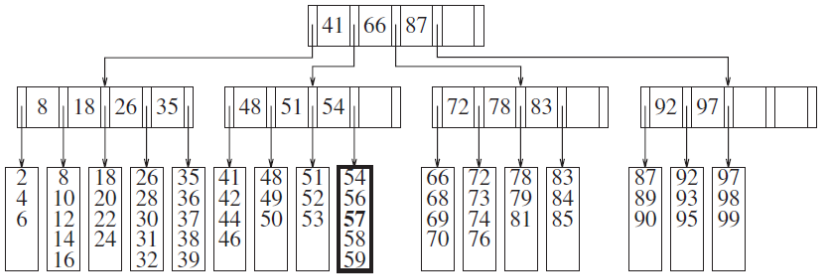


Figure 4.64 B-tree after insertion of 57 into the tree in Figure 4.63

Insertion Examples

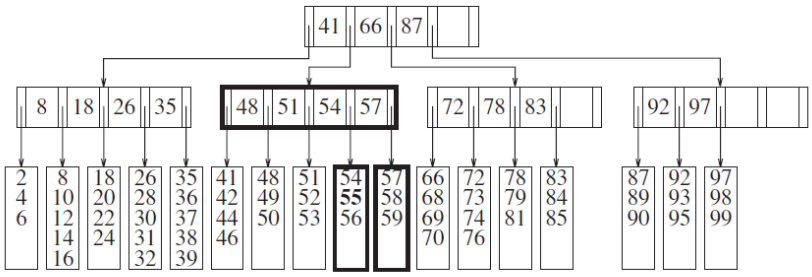


Figure 4.65 Insertion of 55 into the B-tree in Figure 4.64 causes a split into two leaves

Insertion Examples

Insert 40

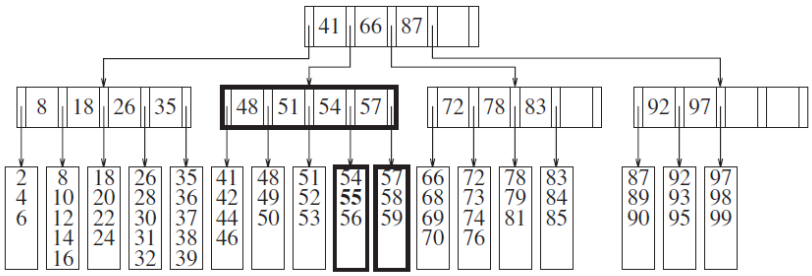


Figure 4.65 Insertion of 55 into the B-tree in Figure 4.64 causes a split into two leaves

Insertion Examples

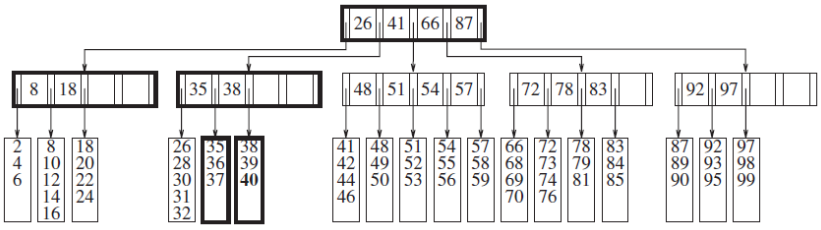


Figure 4.66 Insertion of 40 into the B-tree in Figure 4.65 causes a split into two leaves and then a split of the parent node

Deletion Example

Delete 99

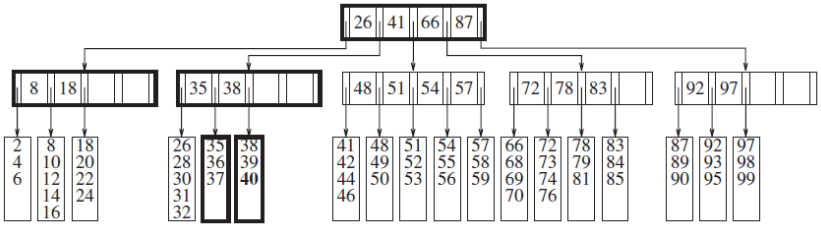


Figure 4.66 Insertion of 40 into the B-tree in Figure 4.65 causes a split into two leaves and then a split of the parent node

Deletion Example

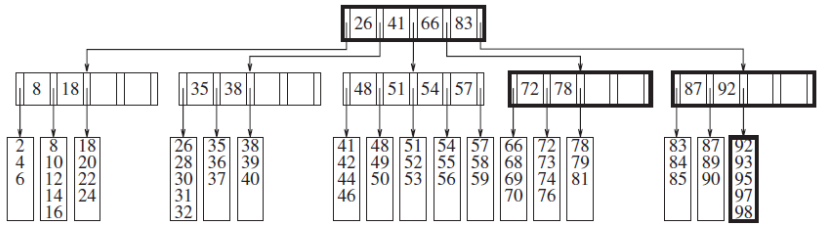


Figure 4.67 B-tree after the deletion of 99 from the B-tree in Figure 4.66

Question

Why is B-tree a space-for-time tradeoff?