B/ B+ Trees

Kumkum Saxena

Motivation for B-Trees

- Index structures for large datasets cannot be stored in main memory
- Storing it on disk requires different approach to efficiency
- Assuming that a disk spins at 3600 RPM, one revolution occurs in 1/60 of a second, or 16.7ms
- Crudely speaking, one disk access takes about the same time as 200,000 instructions

Motivation (cont.)

- Assume that we use an AVL tree to store about 20 million records
- We end up with a **very** deep binary tree with lots of different disk accesses; $\log_2 20{,}000{,}000$ is about 24, so this takes about 0.2 seconds
- We know we can't improve on the log *n* lower bound on search for a binary tree
- But, the solution is to use more branches and thus reduce the height of the tree!
  - As branching increases, depth decreases

Definition of a B-tree

- A B-tree of order *m* is an *m*-way tree (i.e., a tree where each node may have up to *m* children) in which:
  1. the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
  2. all leaves are on the same level
  3. all non-leaf nodes except the root have at least $\lceil m / 2 \rceil$ children
  4. the root is either a leaf node, or it has from two to *m* children
  5. a leaf node contains no more than $m - 1$ keys
- The number *m* should always be odd

Structure of binary search tree node

POINTER TO LEFT SUB TREE	VALUE OR KEY OF THE NODE	POINTER TO RIGHT SUB TREE

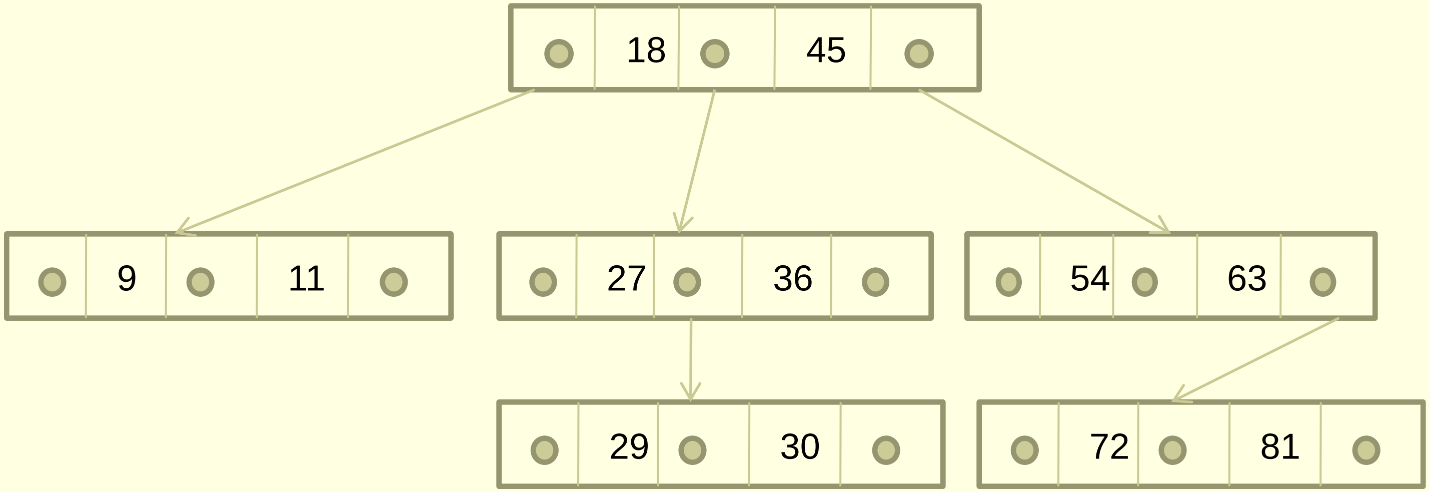
B tree of order **m** can have a maximum **m-1** keys and **m** pointers to its sub-tree. Storing a large number of keys in the single node keeps the height of tree relatively small. In addition it has the following properties:

- Every node in the B-tree has atmost **m** children
- Every node in B-tree except the root node and leaf node has atleast **m/2** children. This condition helps to keep the tree bushy so that path from root node to leaf node is very short, even in a tree that stores lot of data.
- All leaf nodes are at the same level.

Structure of M-way search tree

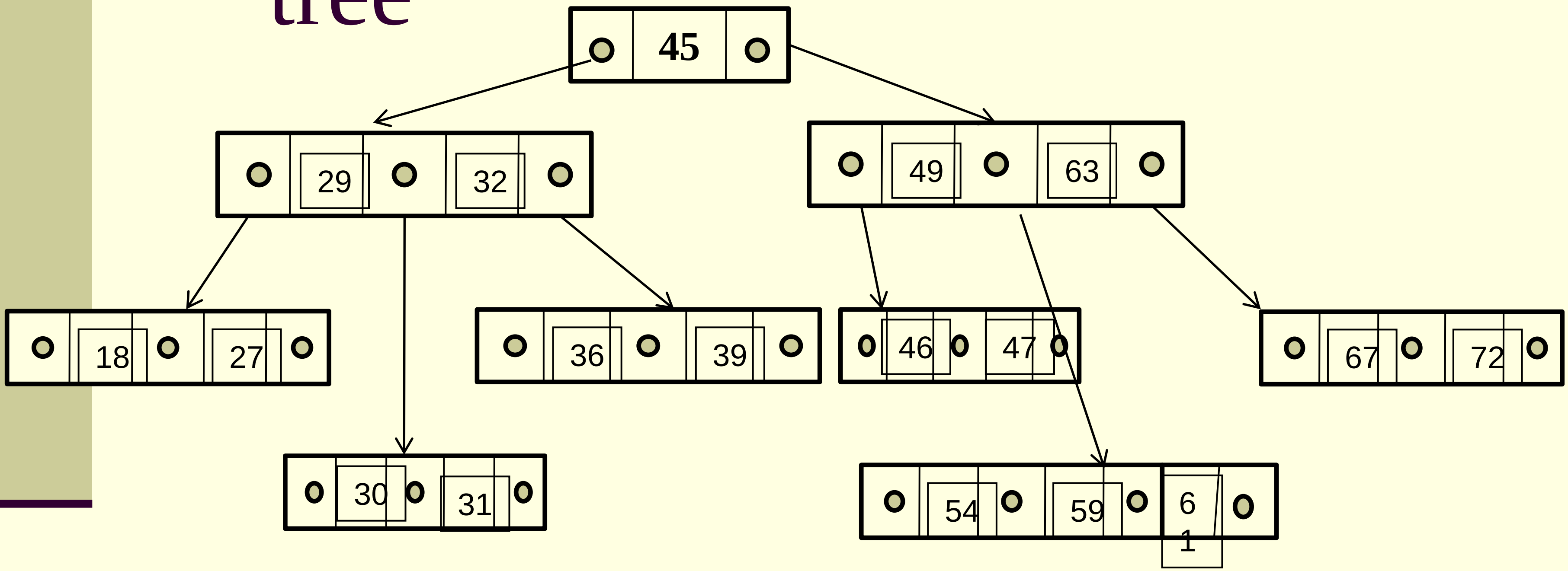
- In the structure shown P0,P1,….PN are the pointers to the node subtree and k0,k1,k2…k_{n-1} are key values of the node. All key values are stored in ascending order. i.e. $k_i < k_{i+1}$ for $0 <= i <= n-2$

P0	K0	P1	K1	P2	K2	……	P_{N-1}	K_{N-1}	P_N



M-way search tree of order 3

Searching for element in B-tree



Search value 59 and 9

Searching for element in B-tree

- Searching for element in B-tree is similar to that in BST.
- To search for 59, we begin at the root node.
- Root has value 45 which is less than 59. so we traverse to right sub-tree. Right sub tree of root node has two key values 49 and 63.
- Since 49<=59<=63. now we traverse the right sub tree of 49 that is the left sub-tree of 63.
- This sub-tree has three values 54 59 61. on finding value 59 search is successful

Inserting new node in B-tree

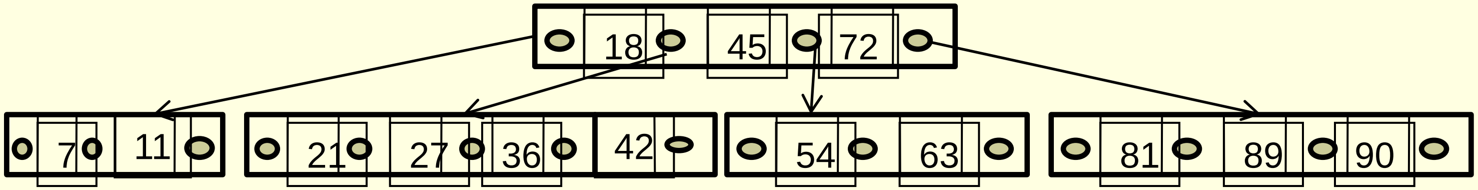
- In a B-tree, all insertions are done at leaf level node. A new value is inserted in B-tree using algorithm given below.
- Search the B-tree to find the leaf node where new key value should be inserted.
- If leaf node is not full that contains less than m-1 key values, then insert the new element in the node keeping the node's element ordered
- If node is full then

  A) insert new values in order into existing set of values.

  B)split the node at its median into two nodes(note that split nodes are half full and

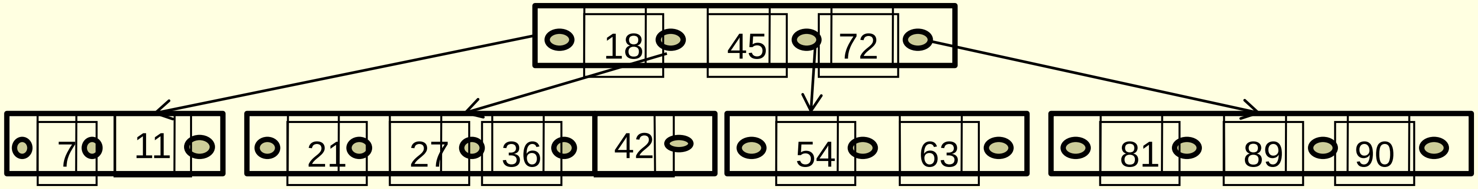
  C)push median at its parent node. If the parent node is already full then split parent node by following same steps

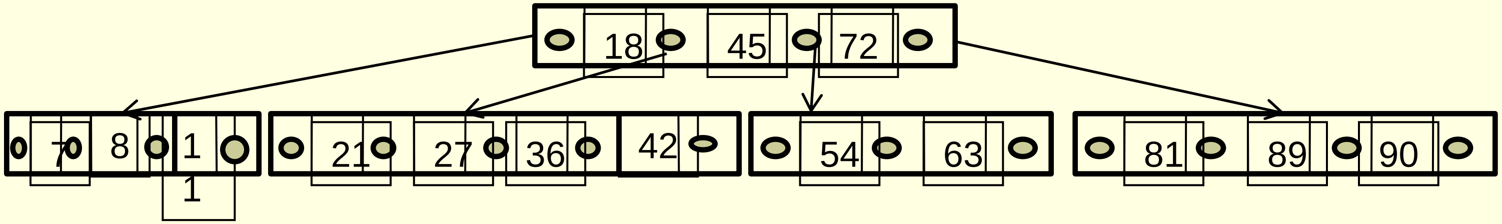
Look at the B tree of order 5 given below insert 8 ,9, 39 and 4 in it



B- tree with order 5

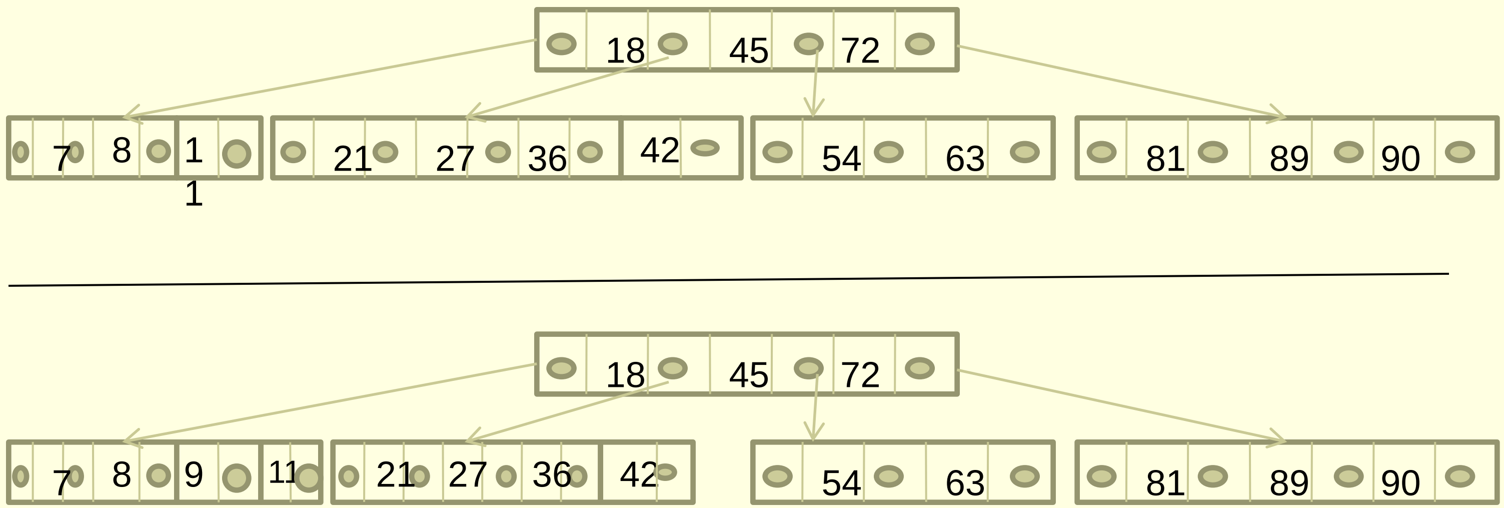
Step 1
Insert 8





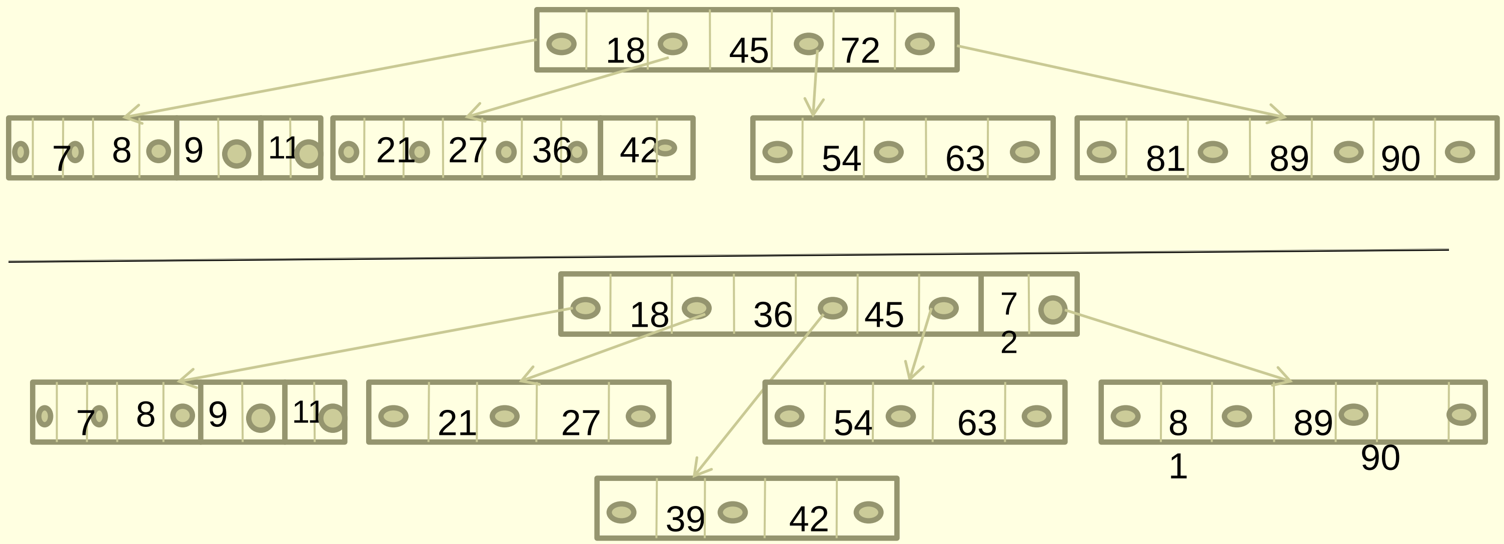
After inserting new value 8

Step 2
Insert 9



After inserting new value 9

Step 3
Insert 39



After inserting new value 39

Step 4
Insert 4