CHAPTER 28

B-TREES

MULTIWAY TREES **(Reading)**
B-TREES **(Reading)**
B-TREE INSERTION
ALGORITHM FOR B-TREE INSERTION
B+ TREES

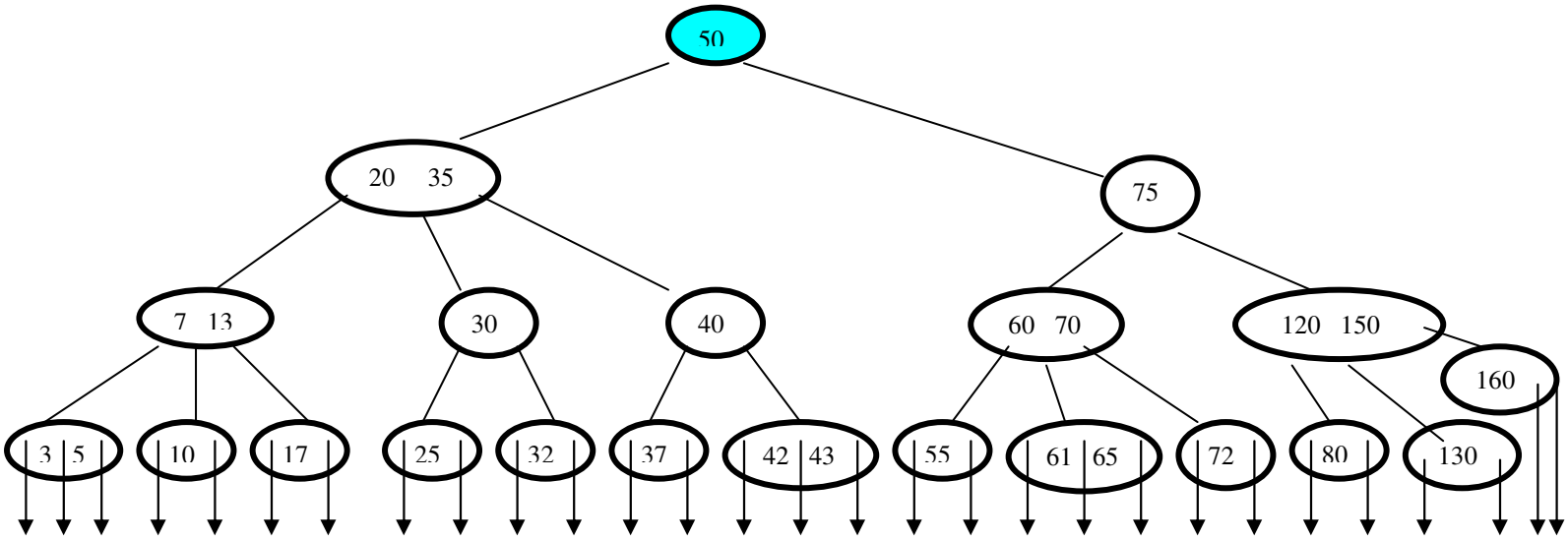
CHAPTER – 28

B-TREES

MULTIWAY TREES

A ***multi-way search tree*** of order *n* is a **general tree** in which each node has *n* or fewer sub-trees and contains one fewer key than it has sub-trees. In addition, *s0, s1, s2*, ……*sm-1* are the m sub-trees of a node containing keys *k0, k1, k2* …..*km-2* in ascending order, all keys in sub-tree *s0* are less than or equal to *k0*, all keys in the sub-trees *sj* are greater than *kj-1* and less than or equal to *kj*, and all keys in the sub-tree sm-1 are greater than *km-2*. The sub-tree *sj* is called the left sub-tree of key *kj* and its root the right child, of key *kj-1*. One or more of the sub-trees of a node may be empty. Nodes, which have the maximum number of keys, are called ***full nodes***.

A ***top-down multi-way search tree*** is characterized by the condition that any non-full node is a **leaf**. ***Semi-leaf*** is a node, which has at least one empty sub-tree. A ***balanced multi-way search tree*** has all its semi leaves at the same level, which are leaves.



Structure of node for mway tree of order n:

```
numtrees       /* number of sub trees under the node */
array of n child /* pointers to structure ( subtree) */
array of n-1  k  /* key fields in the node in ascending order */
array of n-1  r  /* record fields in the node associated with the key */
parent          /* pointer to the parent of the node  */
index           /* child position of this node in the parent node */
```

other variables used:

```
pos    /* position of the child subtree in the node   */
position  /* position of key in the node   */
```

algorithm for search:

```
/* recursive search (tree) returns a pointer to the node containing   */
/* key or –1 (representing null) if no such node exists in the tree. */
/* then it sets position of key in the node.                          */
```

```
p = tree;
if (p == null)
{
        position = -1;
        return (-1);
}       /* if (p == null)   */

i = nodesearch (p, key);

if ( i < numtrees(p) – 1 && key == k(p, i) )
{
        position = i;
        return (p, 0);
}       /* if ( i < numtrees(p) – 1 && key == k(p, i) ) */

return (search (child (p, i) ) );
```

function ***nodeseach*** (p, key) returns the smallest integer j such that *key* <= *k(p, j)*, or *numtrees(p) – 1* if key is greater than all the keys in *node*(*p*).

The function nodesearch is responsible for locating the smallest key in a node greater than or equal to the search argument. The simplest technique for doing this is a sequential search through the ordered set of keys in the node. If all keys are fixed equal length, a binary search can also be used to locate the appropriate key. The decision whether to use a sequential or binary search depends on the order of the tree, which determines how many keys must be searched.

A common operation on a data structure is traversal, accessing all the elements of the structure in a fixed sequence.

```
/* recursive algorithm for traverse(tree)   */

    if (tree != null)
    {
        nt = numtrees(tree)

        for (i = 0; i < nt – 1; i++)
        {
            traverse ( child (tree, i) );
            printf ("%d", k(tree, i) );
        }     /* end of for    */

        traverse ( child (tree, nt) );

    }     /* end of if   */
```

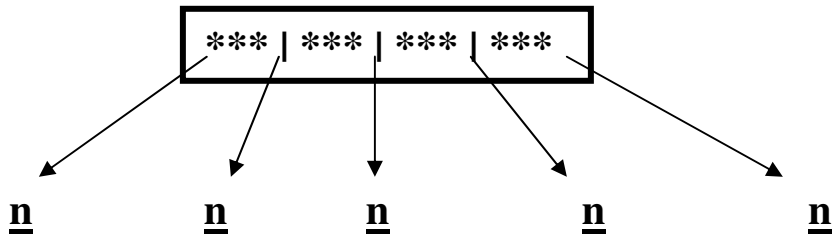
B –TREES

A balanced order-n **multi-way search tree** in which each non-root node contains at least *(n – 1) / 2* keys is called a ***B-tree of order n***. A B-tree of order n is also called *n-(n – 1)* **tree**. This reflects the fact that each node in the tree has a maximum of *n – 1* keys and n children. The ***order*** of a **B-tree** is defined as the minimum number of keys in a non-root node and the ***degree*** of a **B-tree** is to mean the maximum number children.

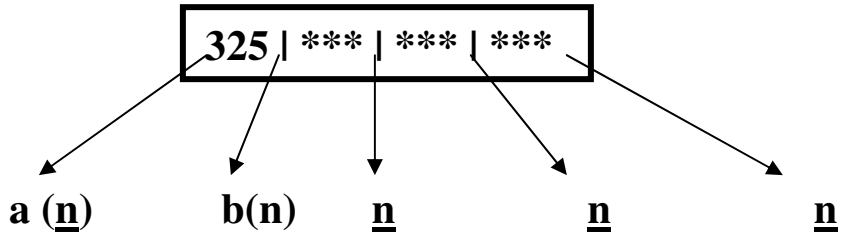
The insertion technique for **B-trees** uses ***find*** in the first step to locate the leaf into which the key should be inserted, and second, if the located leaf is not full, add the key using ***insleaf***. In the third step, when the located leaf is found to be full, instead of creating a new node with only one, split the full leaf into two. The *n* keys, consisting of *n – 1* keys in the full leaf and the new key to be inserted are divided into three groups. The lowest *n/2* keys are placed into the left leaf, the highest *n/2* keys are placed into the right leaf, and the middle key is placed into the parent node if possible. The two pointers on either side of the key inserted into the parent are set to the newly created left and right leaves, respectively.

B-TREE INSERTION

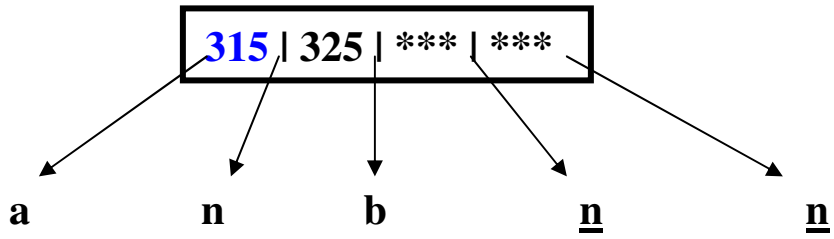
Ndiv2 = Max trees / 2; Get BTree:



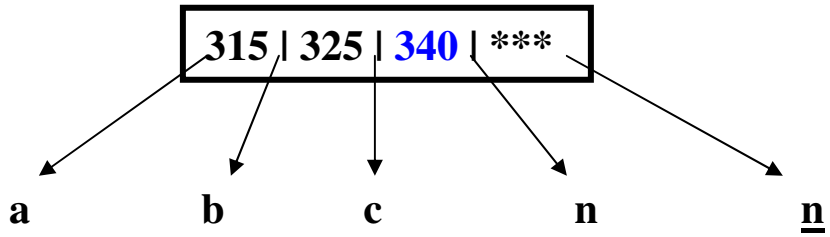
Input: 325



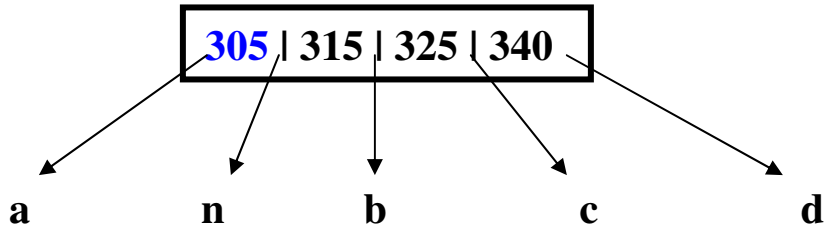
Input: 315



Input: 340

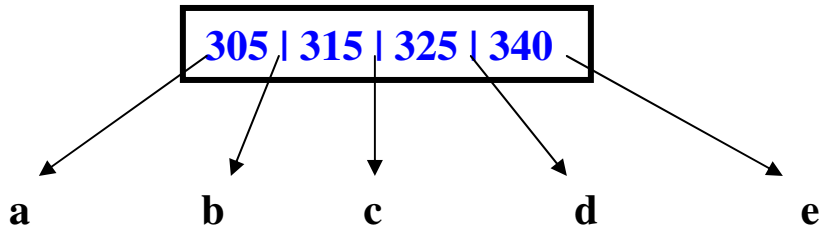


Input: 305

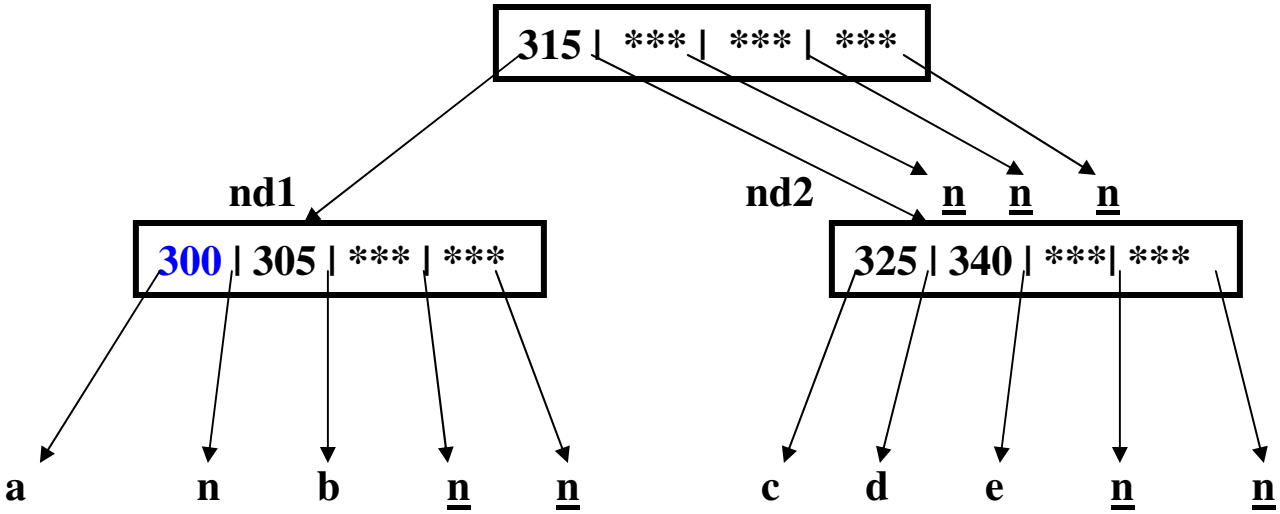


B-TREE split scenarios:

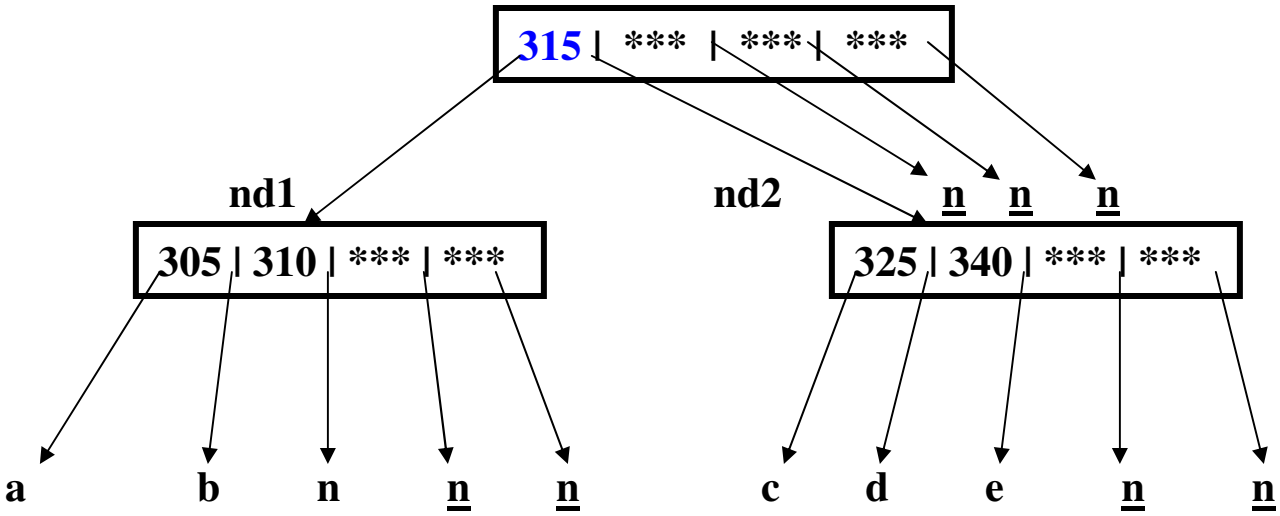
Input: 325, 315, 340, 305



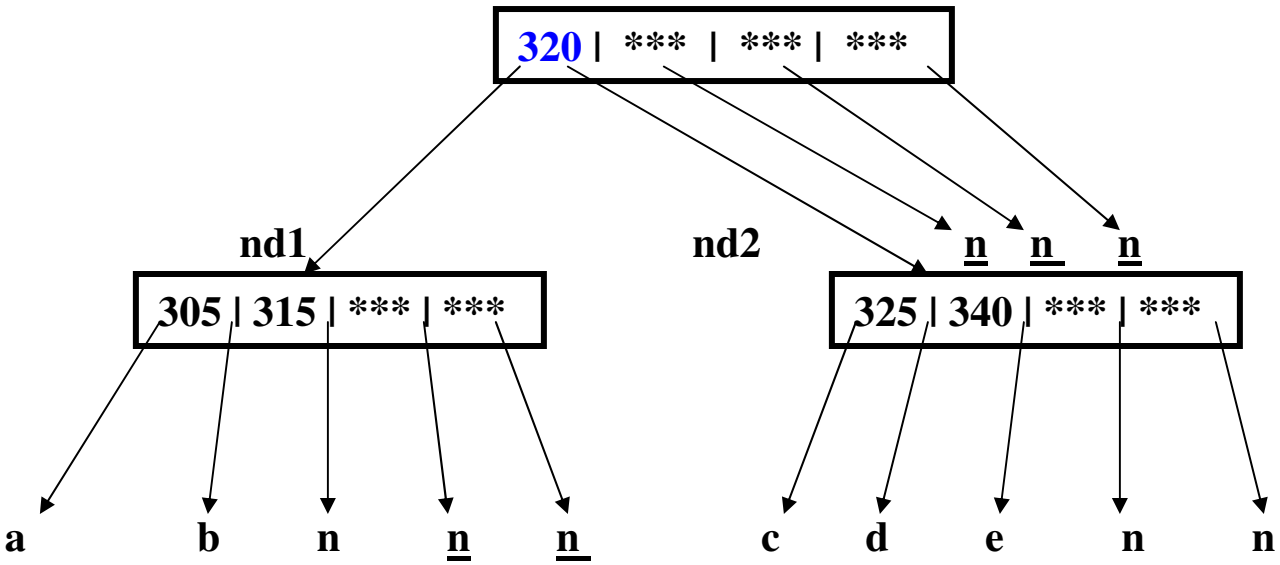
Case 1 - Input: 300, Pos: 0, midkey: 315, pos < ndiv2:



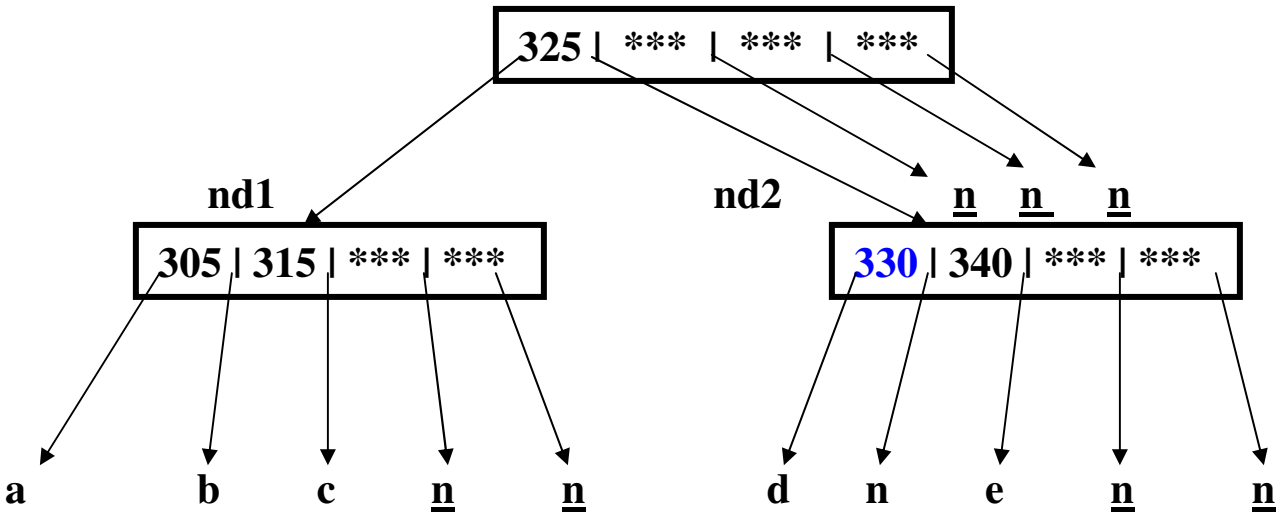
Case 2 - Input: 310, Pos: 1, midkey: 315:



Case 3 - Input: 320, Pos: 2, midkey: 320:



Case 4 - Input: 330, Pos: 3, midkey: 325:



Case 5 - Input: 350, Pos: 4, midkey: 325: