Multiway Search Trees

Prof. Ki-Hoon Lee
School of Computer and Information Engineering
Kwangwoon University

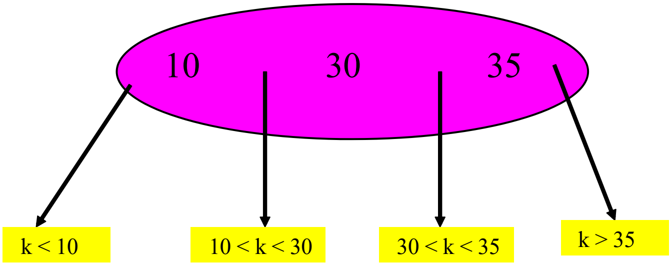
AVL Trees

- n = 1,000,000
- height = 28 = $\lfloor 1.44 \log_2(n + 2) \rfloor$
- When the AVL tree resides on a disk, up to 28 disk access are made for a search.
- Not acceptable.

→ We must reduce the tree height.

m-Way Search Trees

- Each node has up to m – 1 elements and m children.
- m = 2 ➔ binary search tree.



Maximum # of Elements

- Happens when all internal nodes are m-nodes.
- Full degree m tree.
- # of nodes $= 1 + m + m^2 + m^3 + \ldots + m^{h-1}$

  $= (m^h - 1)/(m - 1)$.
- Each node has $m - 1$ elements.
- So, # of elements $= m^h - 1$.

Capacity of m-Way Search Tree

Definition of *m*-Way Search Trees

An *m*-way search tree is either empty or satisfies the following properties:

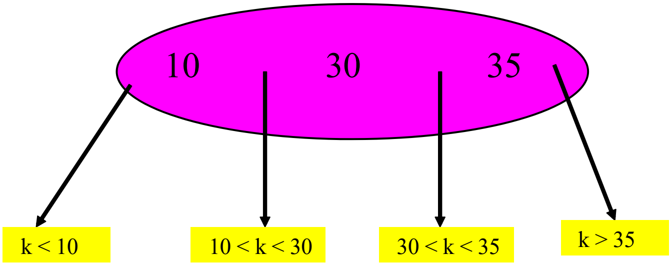
1. The root has at most *m* subtrees and has the following structure:

   $$n, A_0, (E_1, A_1), (E_2, A_2), \ldots, (E_n, A_n)$$

   where the A_i, $0 \le i \le n < m$, are pointers to subtrees, and the E_i, $0 \le i \le n < m$, are elements. Each element E_i has a key $E_i.K$
2. $E_i.K < E_{i+1}.K$, $1 \le i < n$
3. Let $E_0.K = -\infty$ and $E_{n+1}.K = \infty$. All keys in the subtree A_i are greater than $E_i.K$ and less than $E_{i+1}.K$, $0 \le i \le n$
4. The subtrees A_i, $0 \le i \le n$, are also *m*-way search trees

m-Way Search Trees

$3, A_0, (10, A_1), (30, A_2), (35, A_3)$

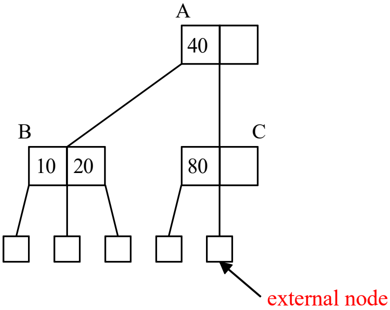


Searching an m-Way Search Trees

// Search an m-way search tree for an element with key x.
// Return the element if found. Return **NULL** otherwise.
$E_0.K = -$MAXKEY;
for ($p = root$; p != NULL; $p = A_i$)
{
    Let p have the format n, A_0, (E_1, A_1), … (E_n, A_n);
    $E_{n+1}.K$ = MAXKEY;
    Determine i such that $E_i.K \leq x < E_{i+1}.K$;
    if ($x == E_i.K$) **return** E_i;
}
// x is not in the tree
return NULL;

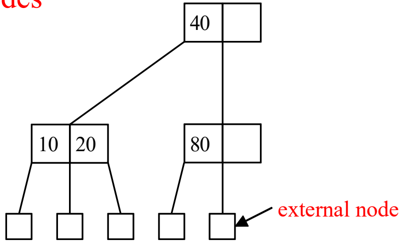
B-Trees

- A balanced *m*-way search tree
- In defining a B-tree, it is convenient to extend *m*-way search trees by the addition of external nodes



B-Trees (cont.)

- An external node represents a node that can be reached during a search only if the element being sought is not in the tree
  - The corresponding child pointer of the parent of each external node is set to NULL
- Nodes that are not external nodes are called internal nodes



Definition of B-Tree

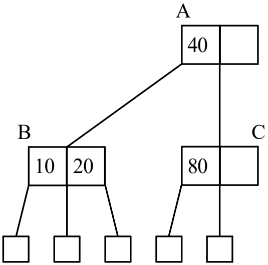
A *B-tree of order m* is an m-way search tree that either is empty or satisfies the following properties:

1. The root node has at least two children.
2. All nodes other than the root node and external nodes have at least $\lceil m/2 \rceil$ children.
3. All external nodes are at the same level.

2-3 and 2-3-4 Trees

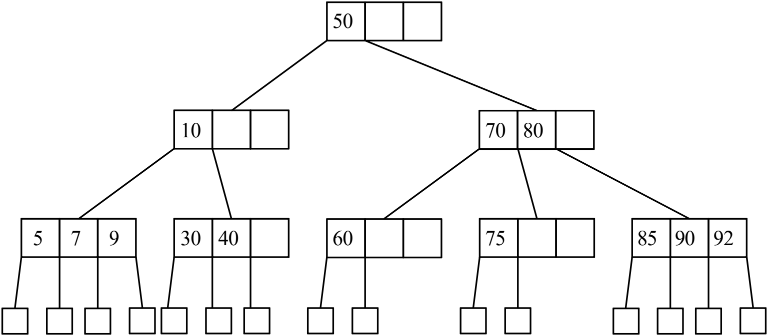
- When $m = 3$, all internal nodes of a B-tree have a degree that is either 2 or 3 (because $\lceil 3/2 \rceil = 2$)
- For this reason, a B-tree of order 3 is known as a 2-3 tree
- A B-tree of order 4 is known as a 2-3-4 tree ($\lceil 4/2 \rceil = 2$)
- A B-tree of order 5 is not a 2-3-4-5 tree ($\lceil 5/2 \rceil = 3$)
  - Root may be 2-node though

2-3 Tree
(= B-tree of order 3)



< 2-3 tree >

2-3-4 Trees
(= B-tree of order 4)

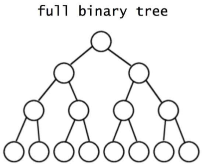


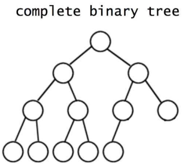
< 2-3-4 tree >

B-Trees of Order 2

- All B-trees of order 2 are full binary trees
  1. The root node has at least two children.
  2. All nodes other than the root node and external nodes have at least one child.
  3. All external nodes are at the same level.

➔ By 1 and 2, the tree is a binary tree, and by 3, the tree is a full binary tree.

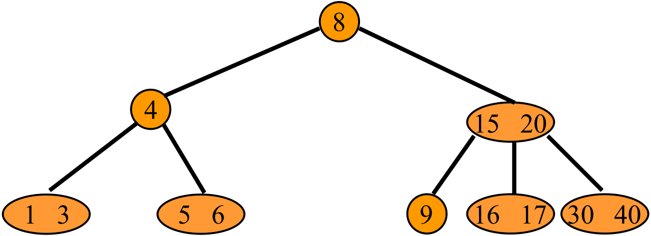




Insertion into a B-Tree

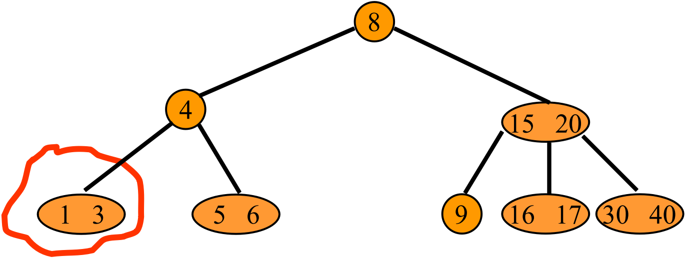
- Performing a search to determine the leaf node, *p*, into which the new key is to be inserted
- If the insertion of the new key into *p* results in *p* having *m* keys, the node *p* is split.
  - This splitting process can propagate all the way up to the root
  - When the root splits, a new root with a single element is created, and the height of the B-tree increases by one
- Otherwise, the new *p* is written to the disk, and the insertion is complete.

Insert (m = 3)



Insertion into a full leaf triggers bottom-up node splitting pass.

Insert (m = 3)

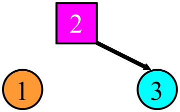


- Insert an element with key = 2.
- New element goes into a 3-node.

Insert into a Leaf 3-node

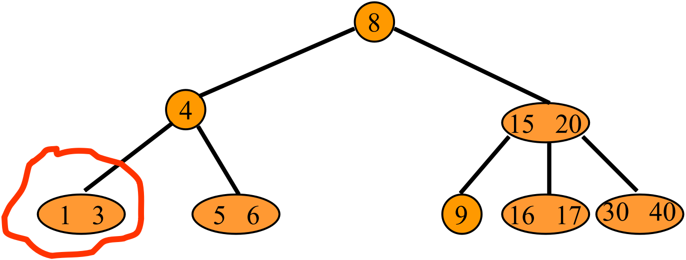
- Insert the new key so that the 3 keys are in ascending order.

- Split the overflowed node around the middle key.



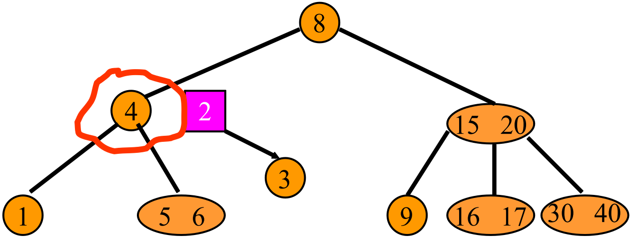
- Insert the middle key and a pointer to the new node into the parent.

Insert (m = 3)



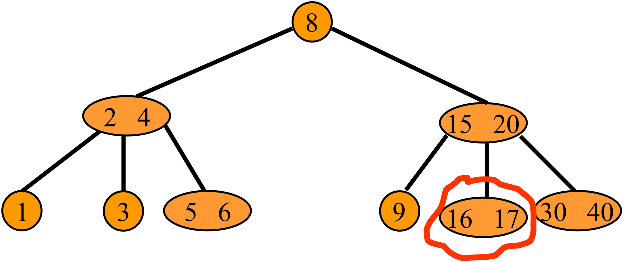
- Insert an element with key = 2.

Insert (m = 3)



- Insert an element with key = 2 plus a pointer into parent.

Insert (m = 3)

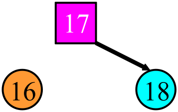


- Now, insert an element with key = 18.

Insert into a Leaf 3-node

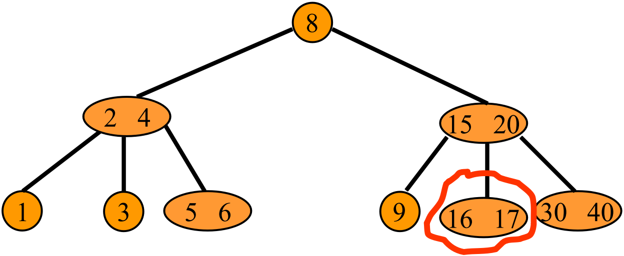
- Insert the new key so that the 3 keys are in ascending order.

- Split the overflowed node.



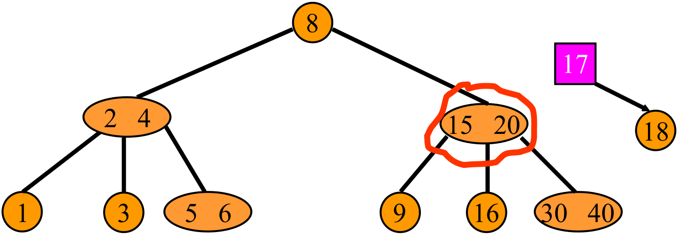
- Insert the middle key and a pointer to the new node into the parent.

Insert (m = 3)

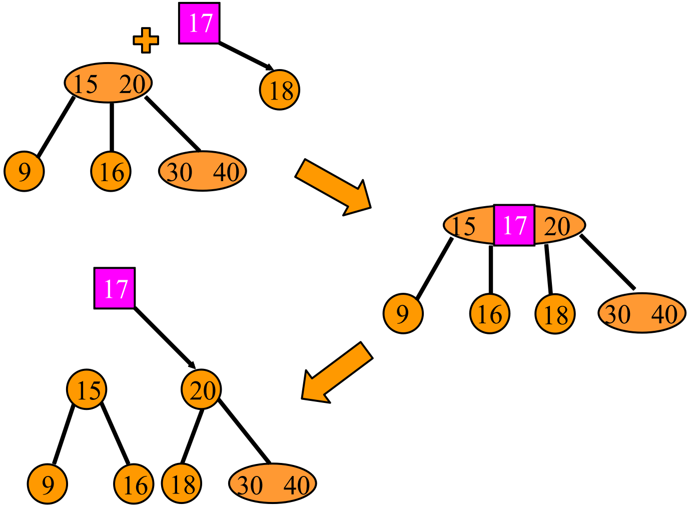


- Insert an element with key = 18.

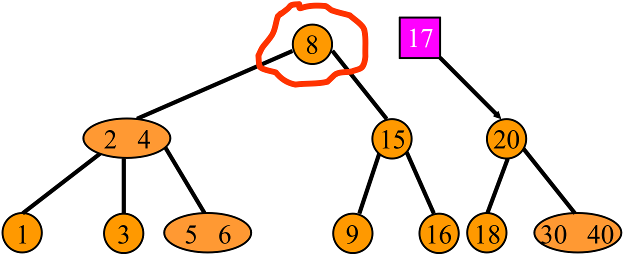
Insert (m = 3)



- Insert an element with key = 17 plus a pointer into parent.

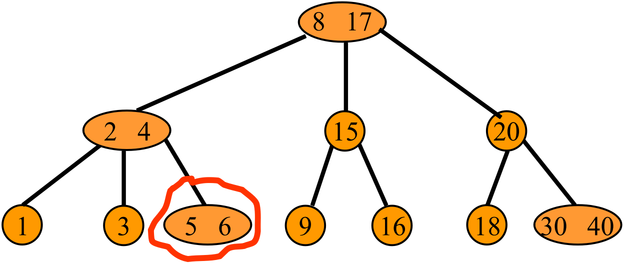

17
15 20
18
9
16
30 40
15 17 20
9
16
18
30 40
17
15
20
9
16
18
30 40

Insert (m = 3)



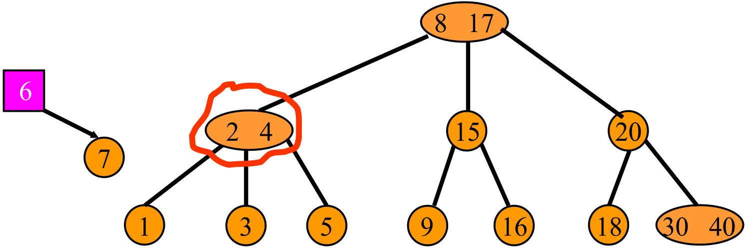
- Insert an element with key = 17 plus a pointer into parent.

Insert (m = 3)



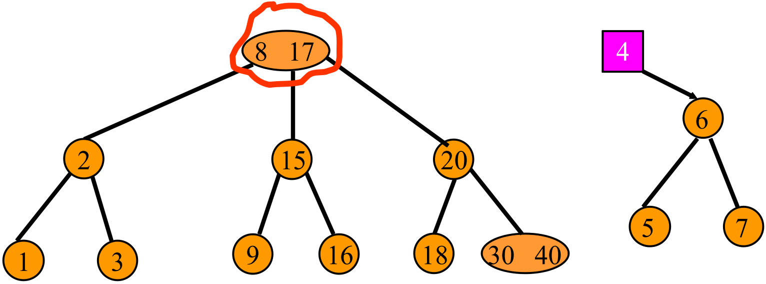
- Now, insert an element with key = 7.

Insert (m = 3)



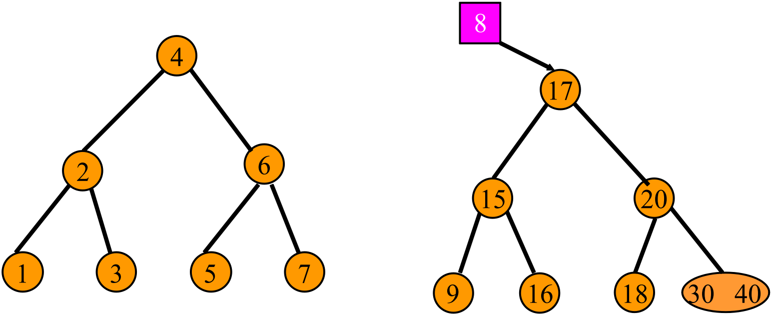
- Insert an element with key = 6 plus a pointer into parent.

Insert (m = 3)



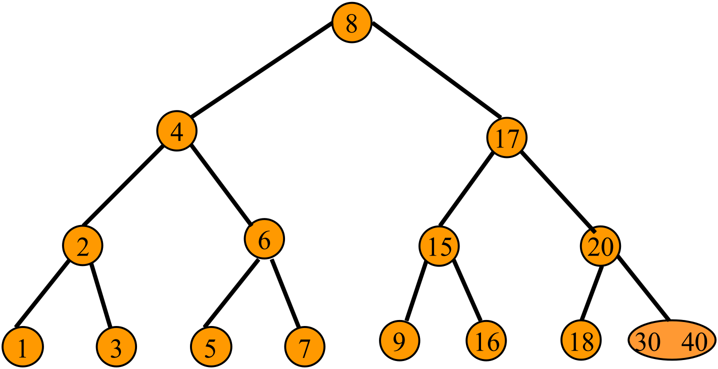
- Insert an element with key = 4 plus a pointer into parent.

Insert (m = 3)



- Insert an element with key = 8 plus a pointer into parent.
- There is no parent. So, create a new root.

Insert (m = 3)



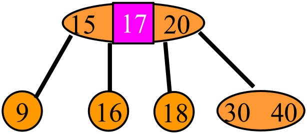
- Height increases by 1.

Split an Overfull Node

- To split the node, assume that following the insertion of the new element, p has the format

  $m, A_0, (E_1, A_1), \ldots, (E_{\lceil m/2 \rceil}, A_{\lceil m/2 \rceil}), \ldots, (E_m, A_m),$

  and $E_i < E_{i+1}$, $1 \leq i < m$

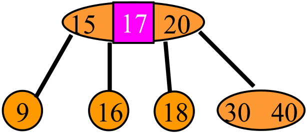


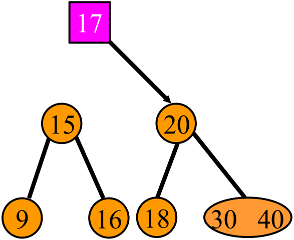
Split an Overfull Node (cont.)

- The node is split into two nodes, p and q, with the following formats: Eq. (11.5)

node p: $\lceil m/2 \rceil - 1$, A_0, (E_1, A_1), … , $(E_{\lceil m/2 \rceil - 1}, A_{\lceil m/2 \rceil - 1})$

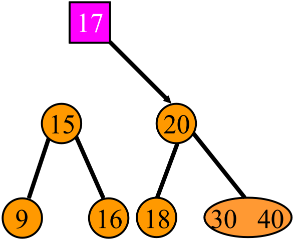
node q: $m - \lceil m/2 \rceil$, $A_{\lceil m/2 \rceil}$, $(E_{\lceil m/2 \rceil + 1}, A_{\lceil m/2 \rceil + 1})$, … , (E_m, A_m)





Split an Overfull Node (cont.)

- The remaining element, $E_{\lceil m/2 \rceil}$, and a pointer to the new node, q, form a tuple $(E_{\lceil m/2 \rceil}, q)$. This is to be inserted into the parent of p.



Split an Overfull Node (cont.)

- Let m = 5. $\lceil 5/2 \rceil = 3$

$5, A_0, (E_1, A_1), \ldots, (E_5, A_5)$

node p: $2, A_0, (E_1, A_1), (E_2, A_2)$

node q: $2, A_3, (E_4, A_4), (E_5, A_5)$

Insert (E_3, q) into the parent of p

- Let m = 4. $\lceil 4/2 \rceil = 2$

$4, A_0, (E_1, A_1), \ldots, (E_4, A_4)$

node p: $1, A_0, (E_1, A_1)$

node q: $2, A_2, (E_3, A_3), (E_4, A_4)$

Insert (E_2, q) into the parent of p

Insertion into a B-Tree

```
// Insert element x into a disk resident B-tree.
Search the B-tree for an element E with key x.K.
if such an E is found, replace E with x and return;
Otherwise, let p be the leaf into which x is to be inserted;
q = NULL;
for (e = x; p != NULL; p = p->parent())
{// (e, q) is to be inserted into p
        Insert (e, q) into appropriate position in node p;
        Let the resulting node have the form: n, A_0, (E_1, A_1), … ,(E_n, A_n);
        if (n <= m – 1) { // resulting node is not too big
                write node p to disk; return;
        }
        // node p has to be split
        Let p and q be defined as in Eq. (11.5);
        e = E_⌈m/2⌉;
        write nodes p and q to the disk;
}
// a new root is to be created
Create a new node r with format 1, root, (e, q);
root = r;
write root to disk;
```

B^+-Tree

- A B^+-tree is a close cousin of the B-tree. The essential differences are:

1. In a B^+-tree we have two types of nodes—index and data.
    - The index nodes of a B^+-tree correspond to the internal nodes of a B-tree while the data nodes correspond to external nodes.
    - The index nodes store keys (not elements) and pointers and the data nodes store elements (together with their keys but no pointers).
2. The data nodes are linked together to form a doubly linked list.

Definition of B$^+$-Tree

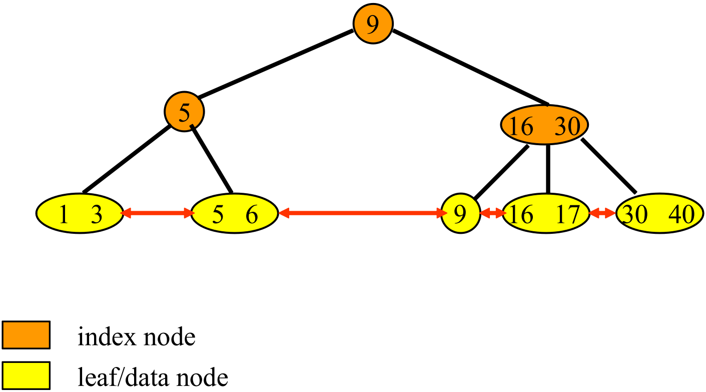
A *B^+-tree of order m* is a tree that either is empty or satisfies the following properties:

1. All data nodes are at the same level and are leaves. Data nodes contain elements only.
2. The index nodes define a B-tree of order m; each index node has keys but no elements.
3. Let

   $$n, A_0, (K_1, A_1), (K_2, A_2), \dots, (K_n, A_n)$$

   where the $A_i, 0 \le i \le n < m$, are pointers to subtrees, and the $K_i, 1 \le i \le n < m$, are keys be the format of some index node. Let $K_0 = -\infty$ and $K_{n+1} = \infty$. All elements in the subtree A_i have key greater than or equal to K_i and less than $K_{i+1}, 0 \le i \le n$.

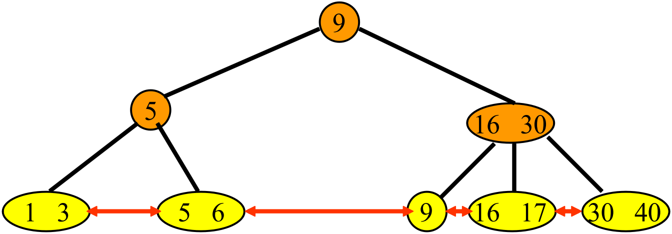
Example B+-tree



Searching a B^+-tree

- B^+-trees support two types of searches—exact match and range
- Range search
  - To search for all elements with keys in the range [A, B], we proceed as in an exact match search for the start, A, of the range
  - We march down (rightward) the doubly linked list of data nodes until we reach a data node that has an element whose key exceeds the end, B, of the search range (or until we reach the end of the list)

B^+-tree—Search



key = 5

6 ≤ key ≤ 20

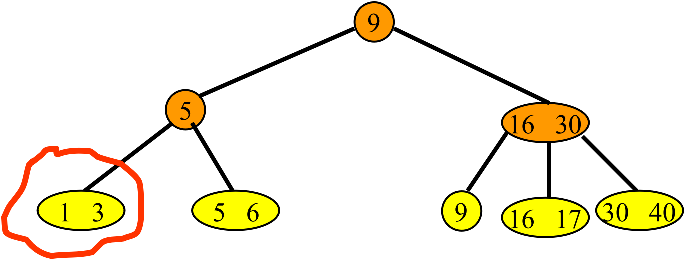
Searching a B^+-tree (cont.)

```
// Search a B+-tree for an element with key x.
// Return the element if found. Return NULL otherwise.
if the tree is empty return NULL;
K_0 = −MAXKEY;
for (p = root; p is an index node; p = A_i)
{
        Let p have the format n, A_0, (K_1, A_1), … , (K_n, A_n);
        K_{n+1} = MAXKEY;
        Determine i such that K_i ≤ x < K_{i+1};
}
// Search the data node p
Search p for an element E with key x;
if such an element is found return E
else return NULL;
```

Insertion into a B^+-tree

- An important difference between inserting into a B-tree and inserting into a B^+-tree is how we handle the splitting of a data node
- When a data node becomes overfull, take the *m* elements (including the one being inserted) in sorted order.
- Place the first half in the original node, and the rest in a new node.
- Let the new node be *q,* and let *k* be the least key value in *q*. Insert (*k, q*) into the parent index node (if any) using the insertion procedure for a B-tree
- The splitting of an index node is identical to the splitting of an internal node of a B-tree

B+-tree—Insert (m = 3)



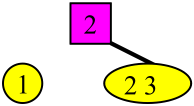
- Insert an element with key = 2.
- New element goes into a 3-node.

Insert into a 3-node

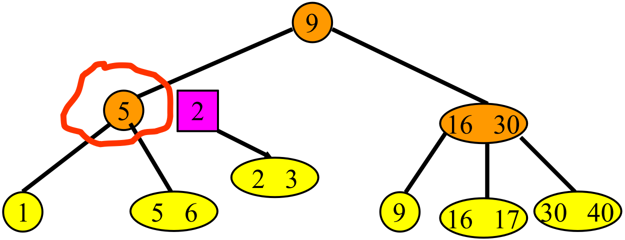
- Insert new key so that the keys are in ascending order.

- Split into two nodes.

- Insert smallest key in new node and pointer to this new node into parent.

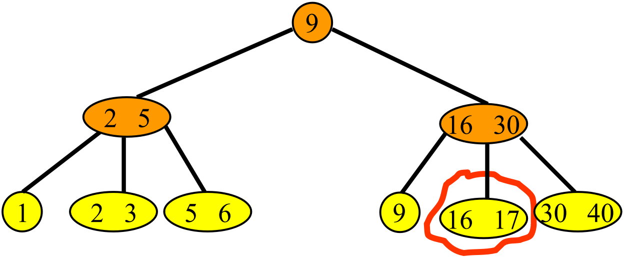


B^+-tree—Insert (m = 3)



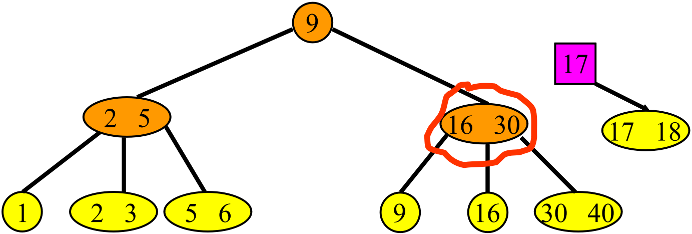
- Insert an index entry 2 plus a pointer into parent.

B^+-tree—Insert (m = 3)



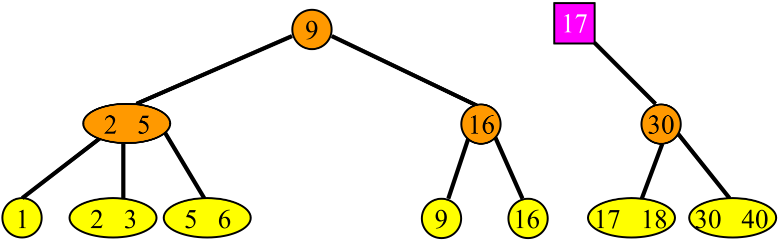
- Now, insert an element with key = 18.

B$^+$-tree—Insert (m = 3)



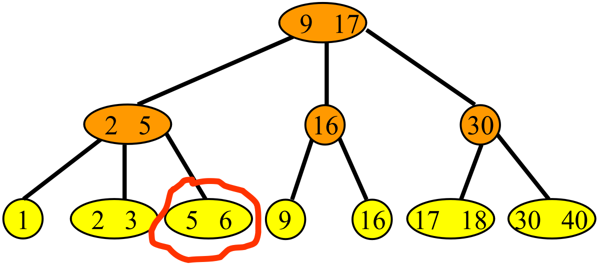
- Now, insert an element with key = 18.
- Insert an index entry 17 plus a pointer into parent.

B$^+$-tree—Insert (m = 3)



- Now, insert an element with key = 18.
- Insert an index entry 17 plus a pointer into parent.

B^+-tree—Insert (m = 3)



- Now, insert an element with key = 7.