

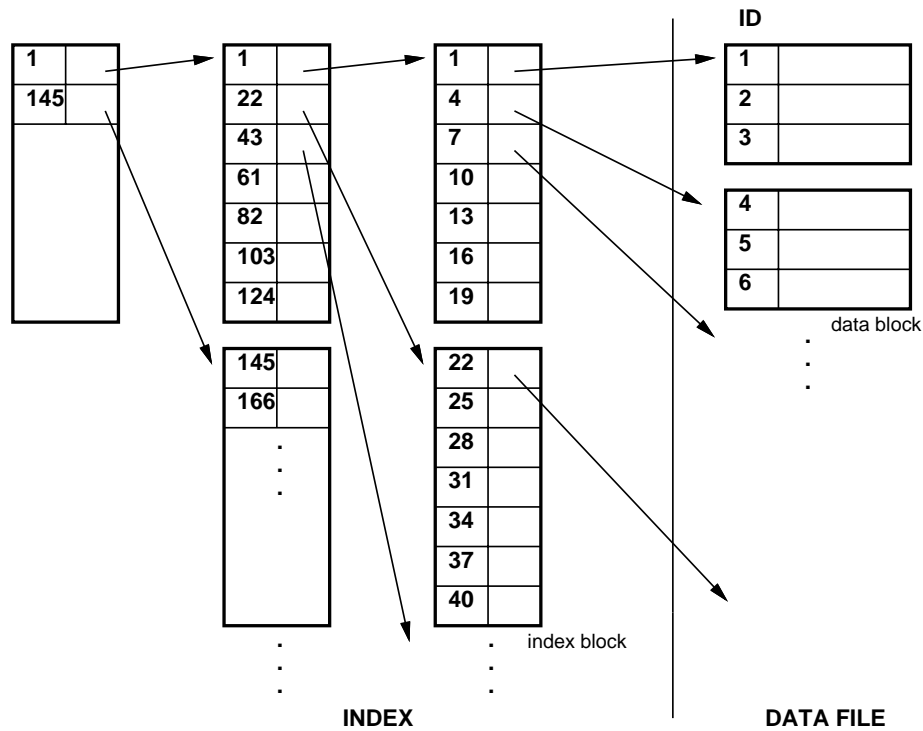
# CS 178: Database Systems

**B. NARAHARI**

Department of Computer Science  
GWU

# B-Trees: An Introduction

- Recall multilevel indices. For example:



- Observe that values like 1 and 145 occur in several places in the index  $\Rightarrow$  deleting ‘1’ from the index requires several record deletions.
- If deletions are left in place (no packing), some index blocks could contain many deleted values  $\Rightarrow$  even if data file becomes small, index remains large.
- If re-packing is done for each insertion,  $O(b_i)$  blocks may be processed.
- If overflow blocks are used, then many insertions cause the overflow blocks to be processed frequently  $\Rightarrow$  slower search.

- Using Tree Structures for Multilevel Indexing
  - natural correspondence between multi-level indexing and tree structures
  - each node in multi-level index can have  $d$  key values and  $d$  pointers
  - index field values (key value) in each node guides us to next node until we reach data file block that contains required node
  - by following pointers we restrict search at each level to subtree of search tree, thus ignoring all nodes not in search tree
  - Suppose we use Binary search trees ?

- What if we use Binary search trees ?
  - each node has at most two children
  - each node entry has one index entry search key and pointer to data record
  - leaves have no children
  - searching for key value equivalent to inorder search time is  $O(h)$  where  $h$  is height of tree
  - ideal case: height is  $O(\log n)$  where  $n$  is number of nodes
  - worst case: height is  $O(n)$ 
    - $\Rightarrow$  same as heap file!
  - need to construct tree such that nodes in subtrees at each node are “balanced”
    - $\Rightarrow$  need concept of Balanced Trees or B-Tree

- A B-tree is a type of multilevel index which:
  - is dense;
  - implements insertion and deletion by dynamic reconstruction of the index at the time of each insertion/deletion;
  - limits the cost of dynamic reconstruction to approximately  $O(\log_f b_i)$  block accesses;
  - Incurs a search cost of approximately  $O(\log_f b_i)$  block accesses;
  - limits the amount of wasted space;
  - does not allow duplicate values in the index.

- A B-tree is a collection of blocks (called B-tree nodes) that contain:
  - search-key (index) values
  - data pointers (block numbers and tuple numbers of data records)
  - tree pointers (block numbers of other B-tree nodes)
  - some information local to each block (such as the number of key values in it)
- Each B-tree node fits into one disk block and the entire B-tree resides on disk  $\Rightarrow$  when a B-tree is accessed, relevant nodes are brought into main memory.

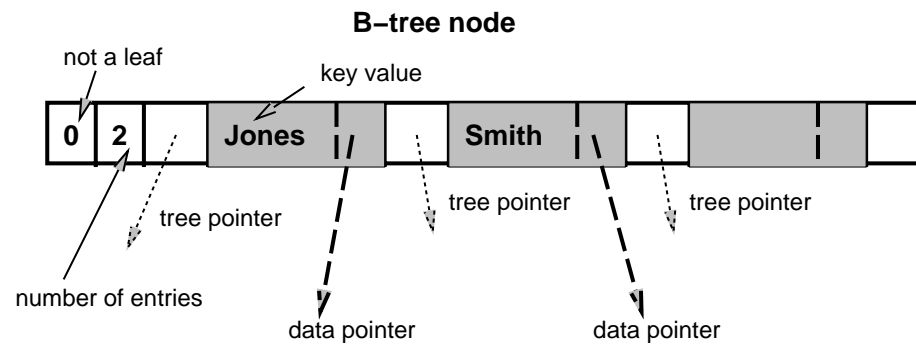
- Every B-tree node of a B-tree of *degree*  $m$  ( $m > 1$ ):
  - (except the root) must have at least  $m - 1$  key values, sorted inside the node;
  - cannot have more than  $2m - 1$  key values;
  - has as many data pointers as key values;
  - has one more tree pointer than the number of key values;
  - is either a *leaf* or an *internal* node.

- NOTE: Alternate (but equivalent) definition:
- Every B-tree node of a B-tree of *order*  $d$  ( $d > 1$ ):
  - (except the root) must have at least  $d$  key values, sorted inside the node;
  - cannot have more than  $2d$  key values;
  - has as many data pointers as key values;
  - has one more tree pointer than the number of key values, i.e., between  $d + 1$  and  $2d + 1$ ;
  - is either a *leaf* or an *internal* node.

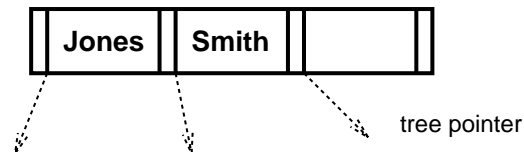
- Each B-tree node fits into one disk block and the entire B-tree resides on disk  $\Rightarrow$  when a B-tree is accessed, relevant nodes are brought into main memory.

Additionally:

- Internal nodes contain tree pointers whereas leaves do not;
- All leaves are at the same depth from the root.



**Conceptual view of a B-tree node**



## Picking Degree of B-Tree

For example, suppose

- block size  $B$  is 512
- 2 bytes are used for block-specific information
- an index value, search key, requires  $K = 20$  bytes
- a data pointer  $P_d$  requires 6 bytes
- a tree pointer  $P_t$  requires 4 bytes

Then, we must have

$$2 + 20(2m - 1) + 6(2m - 1) + 4(2m) \leq 512.$$

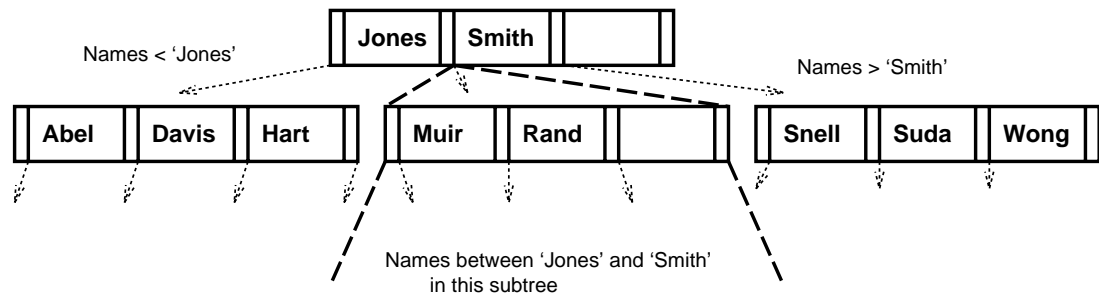
Or

$$m \leq 8.93.$$

Thus, pick  $m = 8$ . In general terms, we must have

$$2 + K(2m - 1) + P_d(2m - 1) + P_t(2m) \leq B.$$

- The nodes in a B-tree are arranged in *in-order*:



- NOTE:

- The root of the B-tree can have fewer than  $m - 1$  values.
- Since  $2m - 1$  is odd, the maximum number of values is either 3, 5, 7, ... etc.
- Since  $m > 1$ , smallest value possible for  $2m - 1$  is 3.
- In a full node, the  $m$ -th value is the called *median* or *middle* value.
- It is possible to generalize the definition to allow for an even number of values. If  $e$  values are allowed (e.g.  $e = 8$ ), the middle value is defined to be the  $(e + 1)/2$ -th value.

- Operations supported by a B-tree:
  - **Insertion**: insert a key-value into the tree.
  - **Deletion**: delete a key-value from the tree.
  - **Search**: search for a particular key-value.
- NOTE:
  - A B-tree is used on top of a data file (usually, a heap file).
  - Often, the combination of the tree and data file is called a B-tree file (or, confusingly, B-tree for short).  $\Rightarrow$  the B-tree's **insert** function also inserts into the data file: `btree_insert(key, tuple)`
  - *Search* in a B-tree typically means *equality search*.
  - Range searches are inefficient in a B-tree.
- Remember: the *tree pointer* in a B-tree node is *not* a memory location  $\Rightarrow$  it is a block number.

- Worst-case height of a B-Tree.

It is possible to compute the worst possible height of a B-tree containing  $n$  nodes without additional detail.

In the worst case, all nodes have only  $m - 1$  values and the root has only 1 value. Thus, 1 level 0 (root) has 1 node

level 1 has 2 nodes

level 2 has  $2m$  nodes

level 3 has  $2m^2$  nodes

⋮

level  $h$  has  $2m^{h-1}$  nodes Therefore,

$$\begin{aligned} n &= 1 + 2 + 2m + 2m^2 + \dots + 2m^{h-1} \\ &= 1 + 2(1 + m + m^2 + \dots + m^{h-1}) \\ &= 1 + 2\frac{m^h - 1}{m - 1} \end{aligned}$$

## Height of B-tree of degree $m$

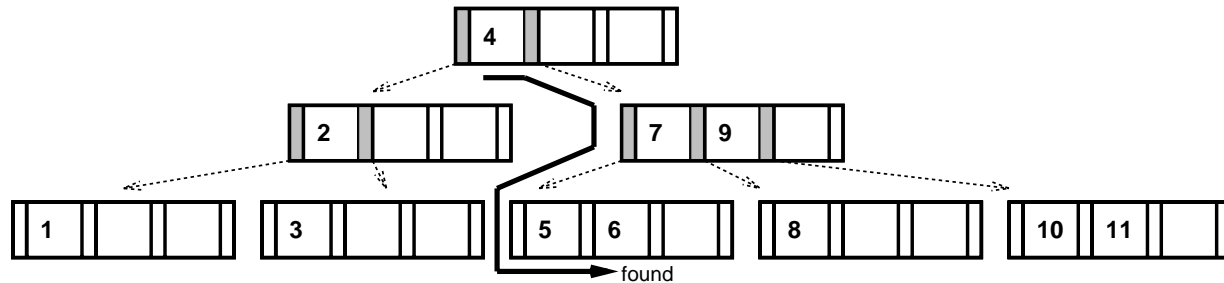
Or,  $h = O(\log_m n)$ .

In fact,  $h \leq 1 + \log_m n$ .

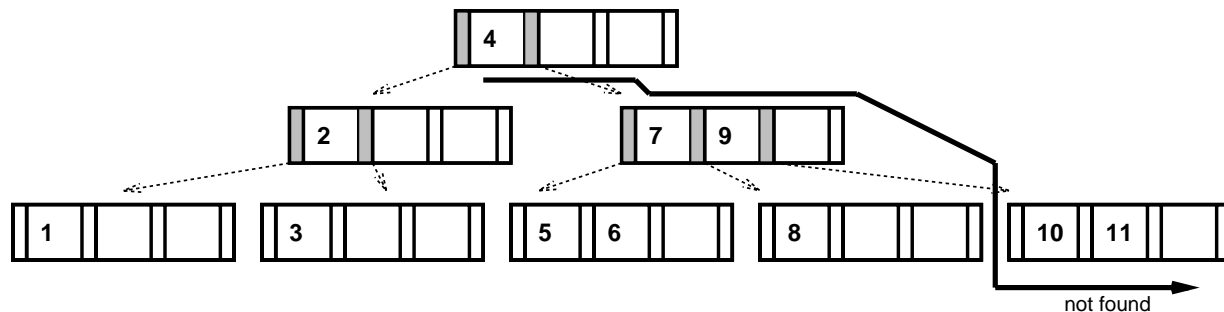
- NOTE:
  - The  $m - 1$  lower bound translates to “at least 50% full”.
  - Any lower bound less than  $m - 1$  is easy to support  $\Rightarrow$  a restriction like “at least 70% full” is possible but difficult.
- in practice, steady state analysis shows nodes are 69% “full”

## Search in a B-Tree

- Searching in a B-tree follows in-order traversal.
  - Start at root block.
  - For each block, search sequentially through elements in block until element is either found or the correct child block is identified.
  - If child block pointer is NULL  $\Rightarrow$  item not in tree.
  - If item is found  $\Rightarrow$  extract data pointer and retrieve tuple from heap file.
  - Otherwise, follow childpointer.
- Example: search for '6' in this tree:



- Example: search for '14':



- What is the complexity of search?  $\Rightarrow$  as many blocks as height of tree (worst case)  $\Rightarrow O(\log_m n)$  for search ( $n$  blocks of data).

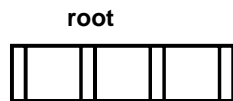
## Insertion in a B-Tree

- We will learn insertion via an example:

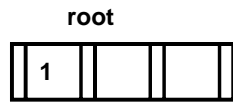
- Consider the case  $m = 2$ .
- The key values are integers.
- Insert the following key values in order: 1,7,8,10,9,3,2,5,4,6,11

NOTE:

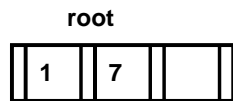
- $m = 2 \Rightarrow$  at least  $m - 1 = 1$  value per node.
  - $m = 2 \Rightarrow$  at most  $2m - 1 = 3$  values per node.
  - Median value is 2nd value.
  - We will not describe insertion of tuple into heap file.
- Initially: Create (empty) root node:



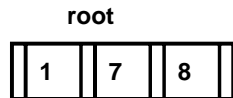
- Insert '1':
  - Space available in root block.



- Insert '7':
  - Fits into root block
  - Insert to right of '1' (in sort-order)



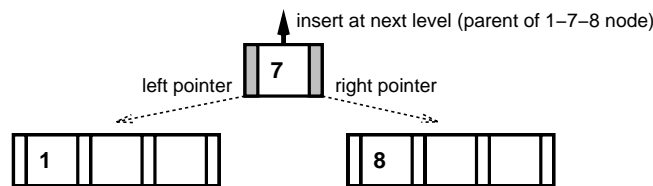
- Insert '8':
  - Fits into root block
  - Insert to right of '7'



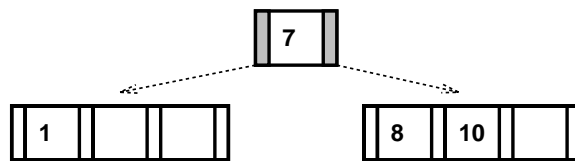
- Insert '10':

- Root node is full  $\Rightarrow$  must split root.
- Median element ('7') is *bumped up* one level (to new root).
- Split nodes are children (left and right) of median element.
- New element ('10') is added in appropriate split node.

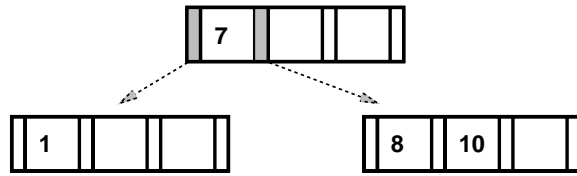
Step 1: Split node:



Step 2: Add new value ('10'):



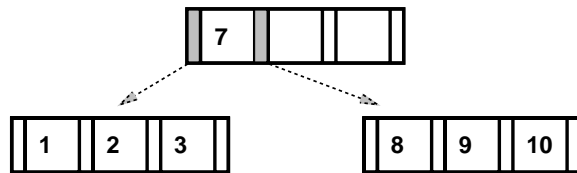
Step 3: Insert median element (with its left and right pointers) at level above.  $\Rightarrow$  in this case, create new root.



This illustrates the general principle behind insertion:

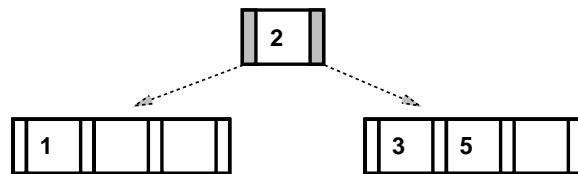
- First, find the correct leaf for insertion.  $\Rightarrow$  use in-order to navigate to correct leaf.
- Note: check equality within interior nodes.  $\Rightarrow$  if found then insertion is a duplicate.
- If space is available, insert into leaf (maintaining sort order).
- Else, split leaf and ‘push up’ median element  $\Rightarrow$  insert median element into old leaf’s parent.
- In pushing up median element, *also move up left and right pointers.*
- Add new item in left or right child (according to sort order).
- If parent is full, split parent etc, recursively.

- After insertion of '9', '3' and '2':

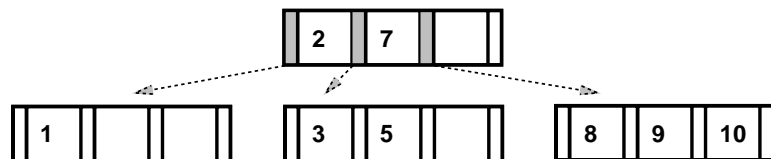


- Insert '5':

- Search for correct leaf  $\Rightarrow$  the '1-2-3' block.
- Block is full  $\Rightarrow$  split (median element is '2').
- Add new element into correct child  $\Rightarrow$  the '3' block:

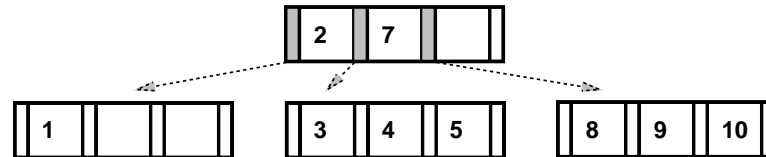


- Insert '2' into parent (the root, in this case):



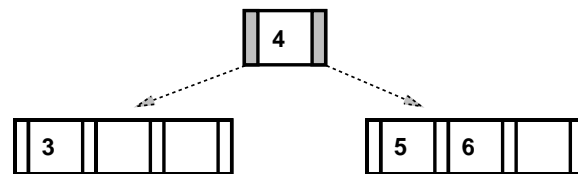
- Insert '4':

- Find correct leaf  $\Rightarrow$  the ‘3-5’ node.
- Space available  $\Rightarrow$  insert.

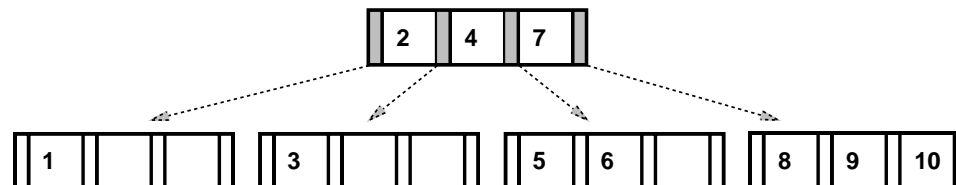


● Insert ‘6’:

- Find correct leaf  $\Rightarrow$  the ‘3-4-5’ node.
- Node full  $\Rightarrow$  split (median element is ‘4’)
- Add new element to correct child (the ‘5’ child, here):

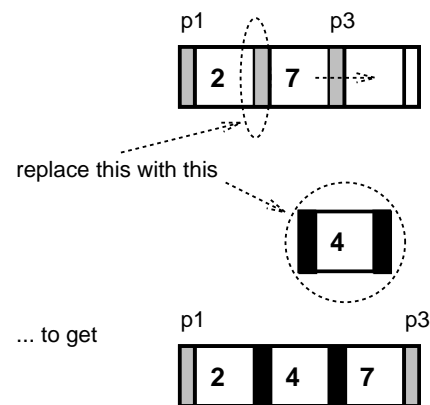


- Insert ‘4’ into parent:



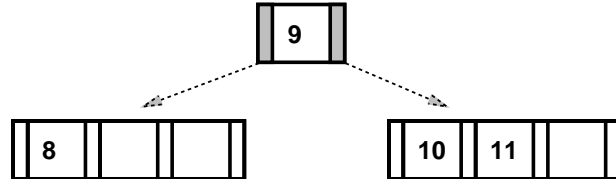
## NOTE:

- The element '7' and everything to the right of it (including pointers) are shifted to the right.
- The (tree) pointer between '2' and '7' is *overwritten* by '4' and its two pointers.

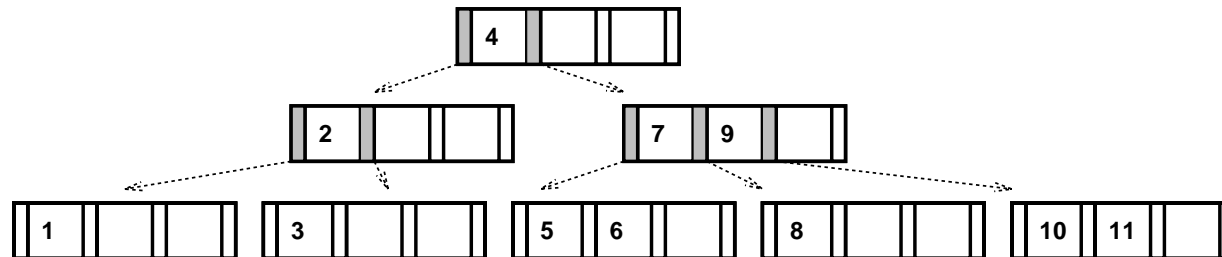


- Insert '11':

- Search for correct leaf  $\Rightarrow$  '8-9-10' block.
- Block is full  $\Rightarrow$  split needed (median element is '9').
- Add new element in correct child  $\Rightarrow$  the '10' block:



- Insert '9' into parent node (the '2-4-7' block)  $\Rightarrow$  '2-4-7' needs to be split (with median '4')  $\Rightarrow$  create new root with '4'.
- Final tree:



## Summary of Insertion Algorithm

- First search to find node where value is to be inserted
- insert into node
- IF node overflows (i.e., more than  $2m - 1$  elements) then SPLIT
  - push median element into parent
  - split node into two nodes with  $m - 1$  values each
  - note that this may incur recursive splitting if parent is full

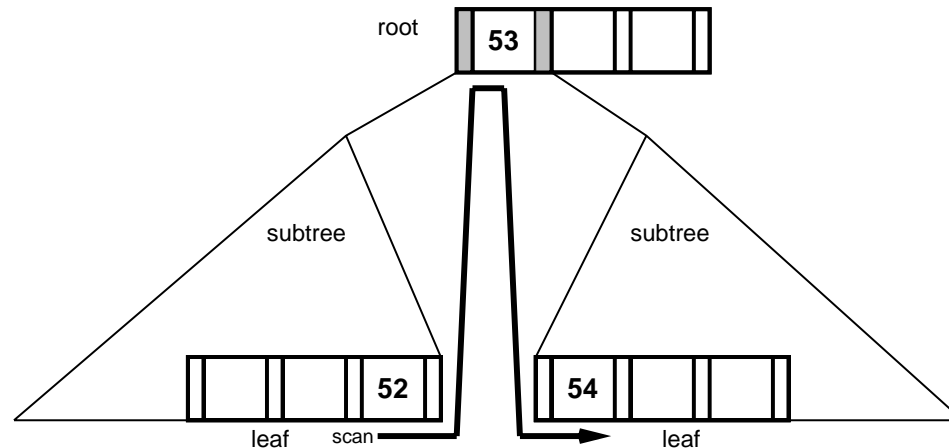
- What is the complexity of insertion?
  - Suppose height of tree is  $h$ .
  - Need to search for correct leaf (in-order traversal)  $\Rightarrow$  at most  $h$  blocks accessed for search.
  - In the worst case, a split propagates to root  $\Rightarrow h$  blocks accessed going back up.
  - Thus,  $2h$  old blocks read,  $h$  old blocks written,  $h$  new blocks written.  $\Rightarrow 4h$  disk accesses (worst case)  $\Rightarrow O(\log_m n)$

NOTE:

- In practice, some blocks are likely to remain in memory.
- Typically,  $m$  is large  $\Rightarrow$  height is small.
- For example, suppose  $m = 100 \Rightarrow$  if  $h = 4$ ,  $2m^{h-1} = 2,000,000 \Rightarrow$  2 million blocks is a lot of data!
- Access times are  $O(h)$ .

## B+-Trees: Introduction

- Recall the following about a B-Tree:  
Index values occur all over the tree  $\Rightarrow$  a **range search** or sorted **scan** can be quite expensive.

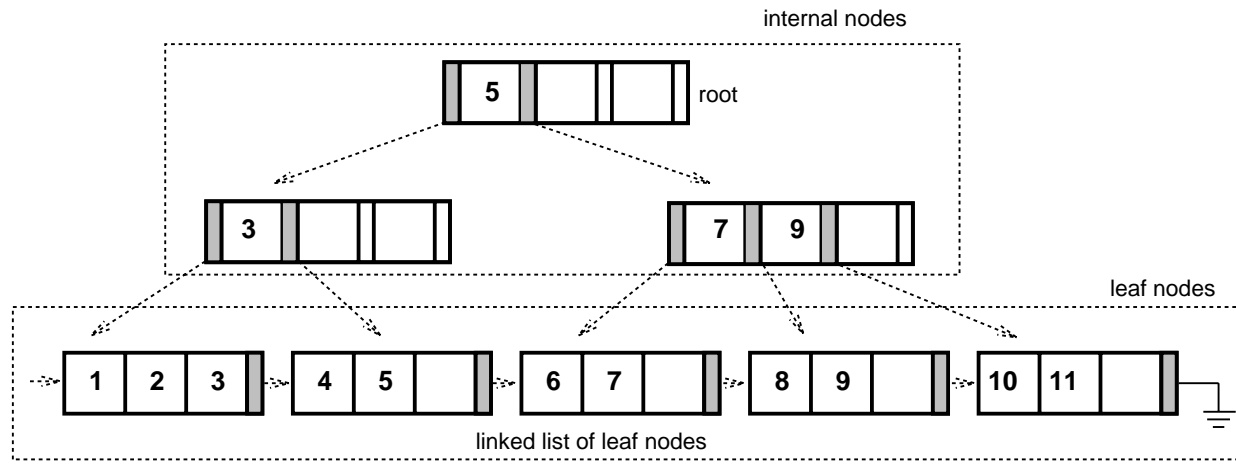


- Recall definition of multi-level index  $\Rightarrow$  data file is only at bottom level
- A B+-tree overcomes this problem (at the cost of allowing duplicate entries).

- Key ideas in a B+-tree:
  - Similar structure to that of a B-tree.
  - Distinguish between *internal* and *leaf* nodes.
  - Leaf nodes contain *all* search keys inserted into tree.
  - Leaf nodes connected by a linked list in *sorted order*.
  - Each *leaf* node:
    - \* has no tree pointers
    - \* has only search keys and data pointers
    - \* has linked list pointer
  - Each *internal* node:
    - \* has search keys
    - \* has tree pointers
    - \* has no data pointers
  - Search keys in internal nodes are used only for *navigation*.

- Search keys in internal nodes are repeated in the leaves.
- Every search key does not occur as an internal value.
- Which search keys get to be in internal nodes as well?  
⇒ depends on what's wrought about by insertions/deletions.

For example:



- Constraints on node contents for a B+-tree of degree  $m$ :
  - Each internal or leaf node must contain at least  $m - 1$  keys.
  - Each internal or leaf node can contain at most  $2m - 1$  keys.
  - Each internal node can contain at most  $2m$  pointers.
  - Each internal node must contain one more pointer than the number of keys.
  - Each leaf node must contain as many data pointers as there

are keys in the node.

#### NOTE:

- In a full node, the  $m$ -th value is the *median* or *middle* value.
- In practice, tree pointers can be smaller than data pointers  
⇒ internal and leaf nodes can have different *degree* ⇒ usually internal nodes can pack more keys ⇒ possibly fewer levels than a corresponding B-tree.
- For simplicity, this presentation will use the same degree for both.
- As with B-trees, the above definitions can be extended to trees with even numbers of keys – use **order** of tree.

Definitions using **order** of tree:

Constraints on node contents for a B+-tree of order  $d$ :

- Each internal or leaf node must contain at least  $d$  keys.
- Each internal or leaf node can contain at most  $2d$  keys.
- Each internal node can contain at most  $2d + 1$  pointers.
- Each internal node must contain one more pointer than the number of keys.
- Each leaf node must contain as many data pointers as there are keys in the node.

NOTE:

- In a full node, the  $d$ -th value is the *median* or *middle* value.
- In practice, tree pointers can be smaller than data pointers  $\Rightarrow$  internal and leaf nodes can have different *order*  $\Rightarrow$  usually

internal nodes can pack more keys  $\Rightarrow$  possibly fewer levels than a corresponding B-tree.

- For simplicity, this presentation will use the same order for both.

## Picking Degree of B+-Tree

For example, suppose

- block size  $B$  is 512
- 2 bytes are used for block-specific information
- an index value, search key, requires  $K = 20$  bytes
- a tree pointer  $P_t$  requires 4 bytes

Then, we must have

$$2 + 20(2m - 1) + 4(2m) \leq 512.$$

$$m \leq 11.1.$$

Thus, pick  $m = 11$ ...Note that this is higher than B-tree degree

In general terms,

$$2 + K(2m - 1) + P_t(2m) \leq B.$$

## Picking Order of B+-Tree

For example, suppose

- block size  $B$  is 512
- 2 bytes are used for block-specific information
- an index value, search key, requires  $K = 20$  bytes
- a tree pointer  $P_t$  requires 4 bytes

Then, we must have

$$2 + 20(2d) + 4(2d + 1) \leq 512.$$

$$d \leq 10.5$$

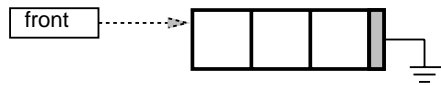
Thus, pick  $d = 10$ ...Note that this is higher than B-tree degree In general terms,

$$2 + K(2d) + P_t(2d + 1) \leq B.$$

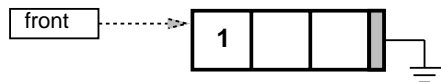
## Insertion in a B+-Tree

- Key ideas similar to a B-tree.
- Search:
  - Search for correct leaf using *in-order* search.
- Insert
  - Must insert in appropriate leaf.
  - If leaf is full, split leaf: push up *copy* of median element to next level and link the two new leaf nodes
    - \* If this leads to a overflow in internal node then Split internal node
    - \* split non-leaf node is same as B-tree: push up median element

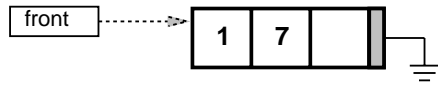
- We will learn insertion via an example:
  - Consider the case  $m = 2$ .
  - The key values are integers.
  - Insert the following key values in order: 1,7,8,10,9,3,2,5,4,6,11
- Initially: Create (empty) leaf node:



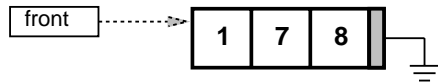
- Insert '1':
  - Space available  $\Rightarrow$  insert.



- Insert '7':
  - Space available  $\Rightarrow$  insert (in sort-order).

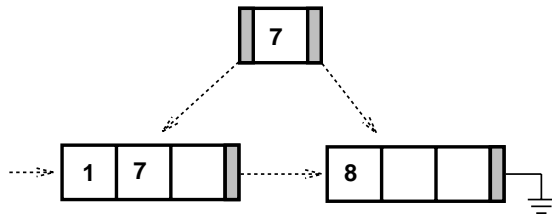


- Insert '8':
  - Space available  $\Rightarrow$  insert (in sort-order).



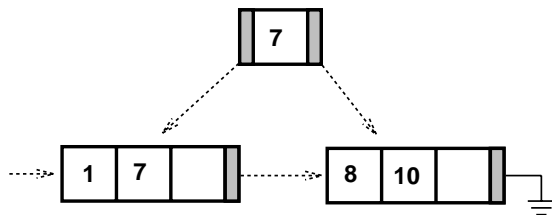
- Insert '10':
  - Node is full  $\Rightarrow$  must split.
  - Median element is '7'.
  - Bump up a *copy* of median ('7') to next level.
  - Split nodes are children (left and right) of median element.
  - New element ('10') is added in appropriate split node.

Step 1: Split node:

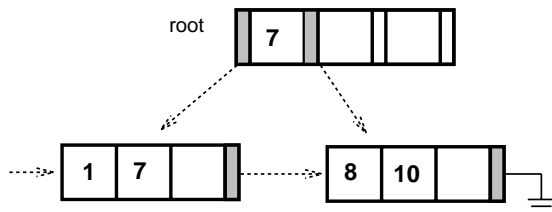


NOTE: '7' remains in left child  $\Rightarrow$  search must use " $\leq$ " (to reach '7' in leaf).

Step 2: Add new value ('10'):



Step 3: Insert median element (with its left and right pointers) at level above.  $\Rightarrow$  in this case, create new root.



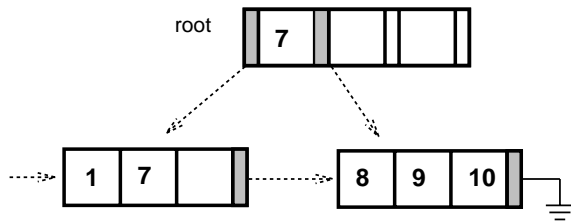
As in a B-tree, the general principle behind insertion is now apparent:

- First, find the correct leaf for insertion.  $\Rightarrow$  use in-order to navigate to correct leaf.
- Check if already present in leaf  $\Rightarrow$  if so, then insertion is a duplicate  $\Rightarrow$  halt.
- If space is available, insert into leaf (maintaining sort order).
- Else, split leaf and ‘push up’ a *copy* of the median element  $\Rightarrow$  insert median element into old leaf’s parent.
- Add new item in left or right child (according to sort order).
- In pushing up median element from a leaf, *create* left and right pointers.
- In pushing up median element from an internal node, also move up left and right pointers.
- If parent is full, split parent, ..., etc, recursively.

- Insert '9':

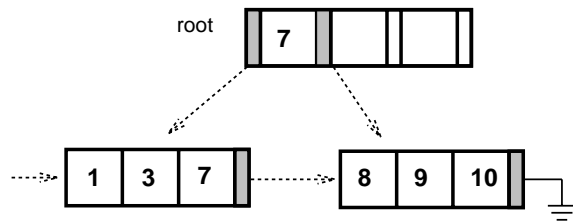
- Search for correct leaf  $\Rightarrow$  the '8-10' node.

- Space available  $\Rightarrow$  insert.



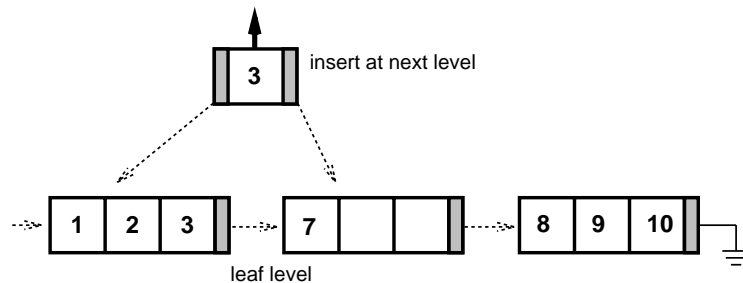
- Insert '3':

- Search for correct leaf  $\Rightarrow$  the '1-7' node.
- Space available  $\Rightarrow$  insert.

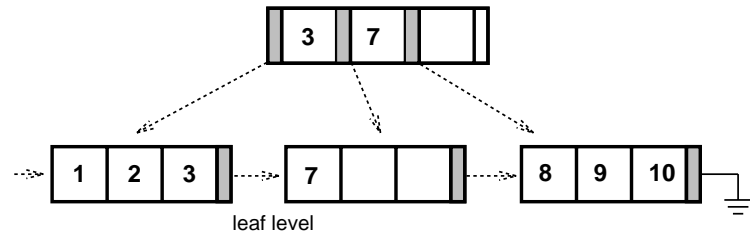


- Insert '2':

- Search for correct leaf  $\Rightarrow$  '1-3-7' block.
- Node is full  $\Rightarrow$  split required (median is '3'):

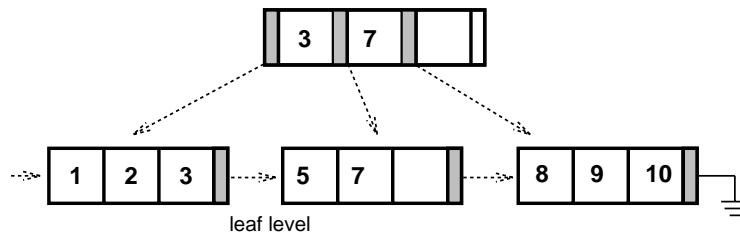


- Insert '3' into parent (the root, in this case):



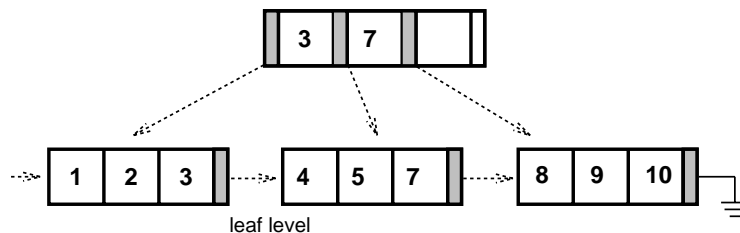
- Insert '5':

- Search for correct leaf  $\Rightarrow$  the '7' node.
- Space available  $\Rightarrow$  insert.



- Insert '4':

- Search for correct leaf  $\Rightarrow$  the '5-7' node.
- Space available  $\Rightarrow$  insert.

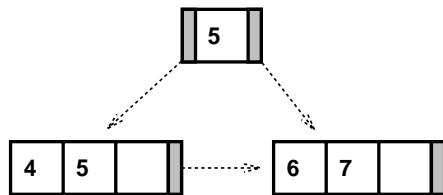


- Insert '6':

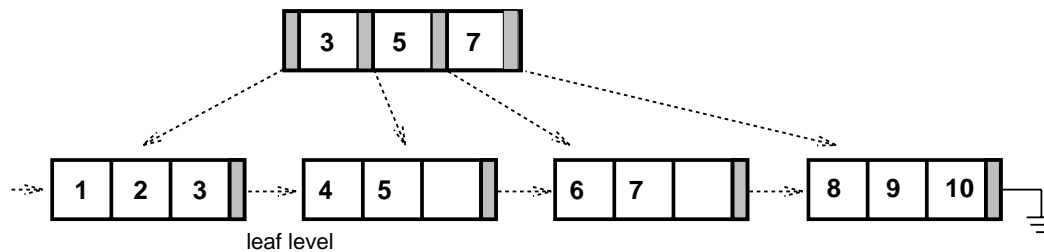
- Search for correct leaf  $\Rightarrow$  the '4-5-7' node.

- Node is full  $\Rightarrow$  split (median is '5').

- Add new element to correct child ('6-7' block):

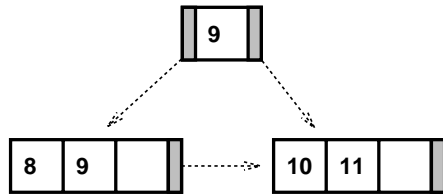


- Insert '5' into parent:

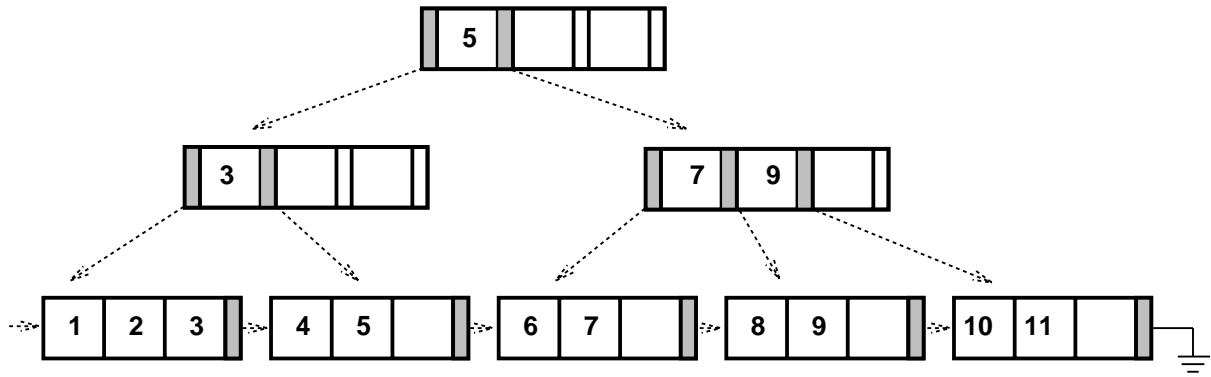


- Insert '11':

- Search for correct leaf  $\Rightarrow$  the '8-9-10' block.
- Block full  $\Rightarrow$  split (median is '9').
- Add new element in correct child (the '10' block):



- Insert '9' into parent.
- Node full  $\Rightarrow$  split (Median is '5').
- Insert '9' in correct child  $\Rightarrow$  the '7' block.
- Create new root ('5').
- Final tree:



- NOTE:

- When a leaf gets split, a *copy* of the median gets pushed up to the next level.
- The median itself stays in the *left* child (by convention).  
⇒ searching must use “ $\leq$ ”.
- When an internal node gets split, the median itself gets pushed up.

- *Searching* a B+-tree is almost identical to searching a B-Tree, except:
  - Use “ $\leq$ ” when searching.
  - Must traverse down to leaf to obtain data pointer.
- What was the whole point about a B+-tree?  $\Rightarrow$  range search is fast!

To search for search keys in the range 4-7:

- Locate the lower limit (‘4’) via search.
- Traverse linked list until you reach (‘7’).  $\Rightarrow$  *optimal* after lower limit is found.

- Complexity of B+-tree operations:
  - Identical to B-tree: depends on tree-height.
  - Tree height is  $O(\log_m n)$ .
  - Search and insertion (and deletion) are  $O(\log_m n)$ .
  - Range search costs  $O(k + \log_m n)$  where  $k$  is the number of blocks containing the desired range.
- Pseudocode: analogous to the pseudocode for the B-tree, the following functions are defined:
  - B+TREE-CREATE (tuplesize, keysize, keyoffset).
  - B+TREE-SEARCH (key).
  - B+TREE-RECURSIVE-SEARCH ( $b$ , key).
  - B+TREE-INSERT (tuple).
  - B+TREE-RECURSIVE-INSERT ( $b$ , key,  $T$ , leftblock, rightblock).

- **B+TREE-CREATE-NEW-ROOT** (key,  $T$ , leftblock, rightblock).
- refer to notes/book

Note that leaf-nodes will have **next**-pointers to link them.

## Deletion in a B+-Tree

- Since some values appear twice (leaf and internal nodes), is deletion in a B+-tree complex? Yes and no.
- Three types of deletion:

### **Lazy deletion:**

- Delete only at leaf level.
- Mark corresponding internal node value as deleted (if one exists).
- Do not remove internal value  $\Rightarrow$  it can still be used for navigation.

- Periodically rebuild tree and remove deleted internal values.

### **Very lazy deletion:**

- Mark both internal and leaf values as deleted.
- Periodically rebuild tree and remove both deleted internal and leaf values.

### **Full deletion:**

- Remove both leaf and internal copies.

### Lazy deletion methods:

- Easy to implement.
- Tree can become large.
- Have to be careful about re-insertions of deleted values (otherwise duplicates might exist in internal nodes).
- Need to keep track for periodic re-building.

Full deletion is harder to implement, but is space efficient. Each delete takes longer.

- more details refer to notes...

## B-Trees and B+-Trees: A Summary

- B-trees and B+-trees:
  - are versatile indexing methods, usually used instead of plain single or multilevel indices;
  - support search in  $O(\log_m n)$  time where  $m =$  degree of node  
 $n =$  number of index blocks
  - support dynamic insertion and deletion in time  $O(\log_m n)$ ;
  - are in-order trees with multiple key values in each tree node;
- B+-trees are usually favored over B-trees in practice.  $\Rightarrow$  range search is faster in a B+-tree.
- Disadvantages?
  - Worst-case, each node will be only 50% full  $\Rightarrow$  twice the blocks needed for an ISAM index.

- However, it can be shown that under uniform accesses, block occupancy is more than 65%.
- B\*-tree: a variation of a B+-tree that forces a high occupancy (67%).

- Summary of Node Splitting

- split node when number of entries exceeds maximum,  $2m - 1$
- find median element
- split node into two “siblings”
  - \* values less than median in left subtree
  - \* values greater than median in right subtree
- For B-Tree push up median element into parent node
- For B+ trees, push up **copy** of median into parent node, keep data entry at leaf node.

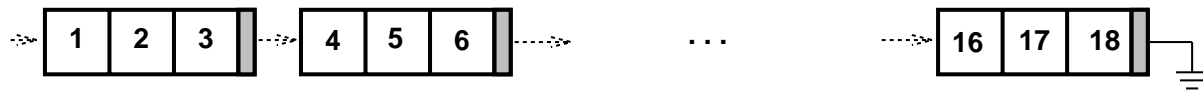
- Some remaining practical issues:
  - *Duplicate values*:
  - *Key compression*:
    - \* Sometimes, keys can be very long  $\Rightarrow$  few records in each index node  $\Rightarrow$  small  $m$   $\Rightarrow$  not very efficient.
    - \* In many cases, it makes sense to use only a *substring* in the key (e.g., a small prefix)  $\Rightarrow$  compression of key increases  $m$   $\Rightarrow$  more efficient.
    - \* Need to be careful with duplicate substrings from different keys.
  - *Bulk loading*:
    - \* Often, a B+-tree is created on top of an existing heapfile.
    - \* Naive approach to building B+-tree: scan heapfile and insert items successively.

- \* Successive insertion can cause many nodes to have low occupancy.
- \* Better to create initial B+-tree using a *bulk loading* algorithm.
- \* Some bulk loading algorithms build the tree bottom-up.

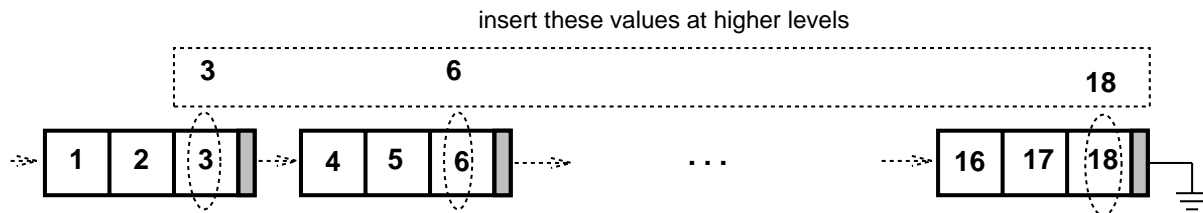
## Summary of Indexing

- Organizing the file of records leads to lower access times
- indexing a file can speed up access
- Hashing is a simple but effective method
- multi-level indices implemented using B-tree concept
- external sorting helps create a sorted data file
- File organization methods help us implement the relational operators  $\Rightarrow$  NEXT....

- Example of bulk-loading ( $m = 2$ ):  
Create a B+-tree from the keys 1,2,3,...,18.
- First, create the leaf-level nodes in order:

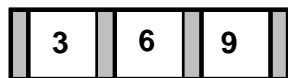


- Next, create copies of last values in each block: leaf block: 3,...,18.

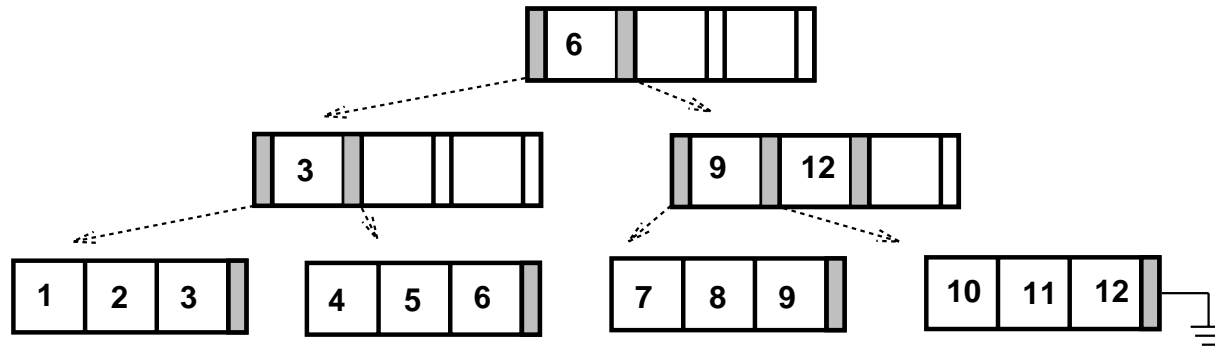


These values are to be copied into higher levels.

- Values 3,6,9 form first block at next level



- Inserting 12 causes a split:



- Other items are added similarly.

## Deletion in a B+-Tree

- Since some values appear twice (leaf and internal nodes), is deletion in a B+-tree complex? Yes and no.
- Three types of deletion:

### **Lazy deletion:**

- Delete only at leaf level.
- Mark corresponding internal node value as deleted (if one exists).
- Do not remove internal value  $\Rightarrow$  it can still be used for navigation.
- Periodically rebuild tree and remove deleted internal values.

### **Very lazy deletion:**

- Mark both internal and leaf values as deleted.

- Periodically rebuild tree and remove both deleted internal and leaf values.

### **Full deletion:**

- Remove both leaf and internal copies.

### Lazy deletion methods:

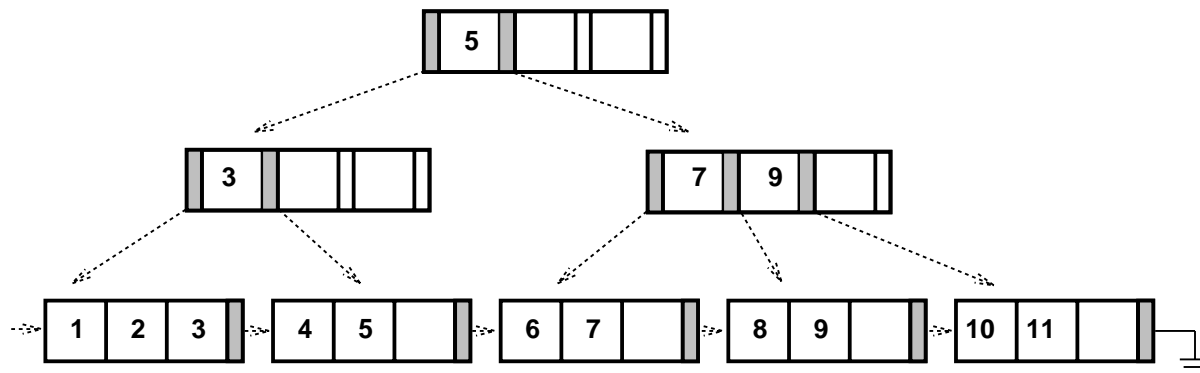
- Easy to implement.
- Tree can become large.
- Have to be careful about re-insertions of deleted values (otherwise duplicates might exist in internal nodes).
- Need to keep track for periodic re-building.

Full deletion is harder to implement, but is space efficient. Each delete takes longer.

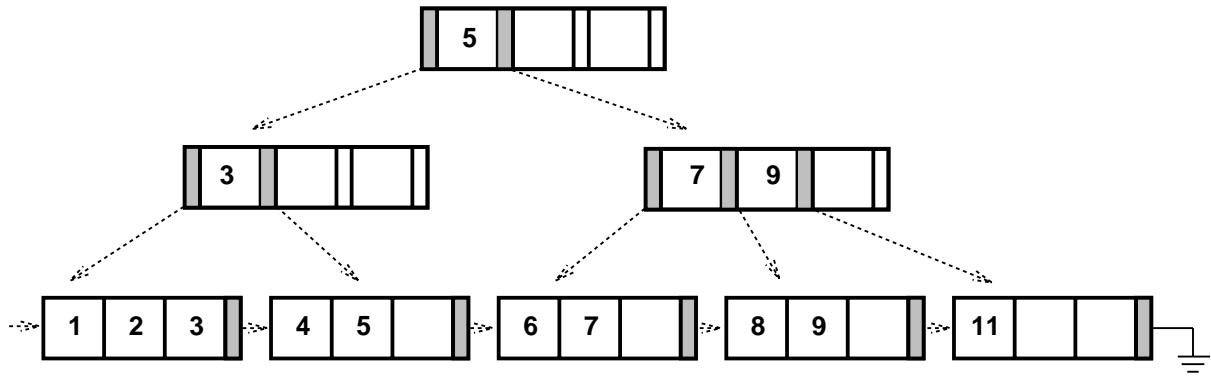
## Lazy Deletion in a B+-Tree

- Consider this example (degree  $m = 2$ ):

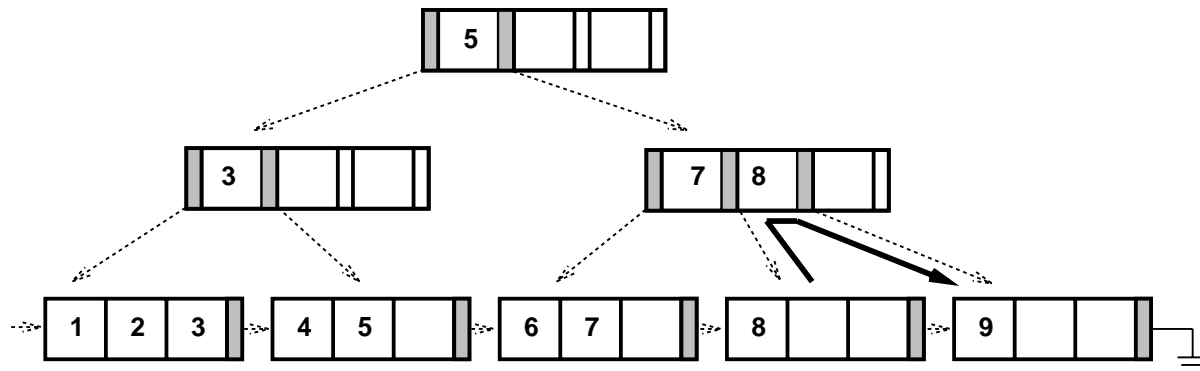
Delete 10, 11, 5, 4, 2, 3, 8 from:



- Delete '10':
  - Search for '10'.
  - '10' is not found in an internal node  $\Rightarrow$  no marking needed.
  - Direct delete does not cause underflow  $\Rightarrow$  delete.

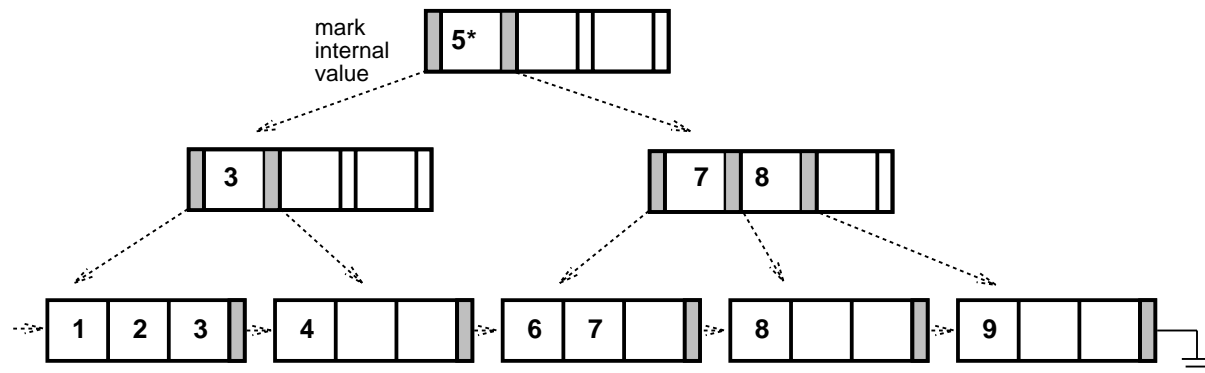


- Delete '11':
  - Search for '11'.
  - '11' does not occur as an internal value.
  - Deletion from leaf causes underflow  $\Rightarrow$  try to borrow from sibling.
  - To borrow, rotate from sibling  $\Rightarrow$  when moving '9' over, copy of '8' is placed into parent:

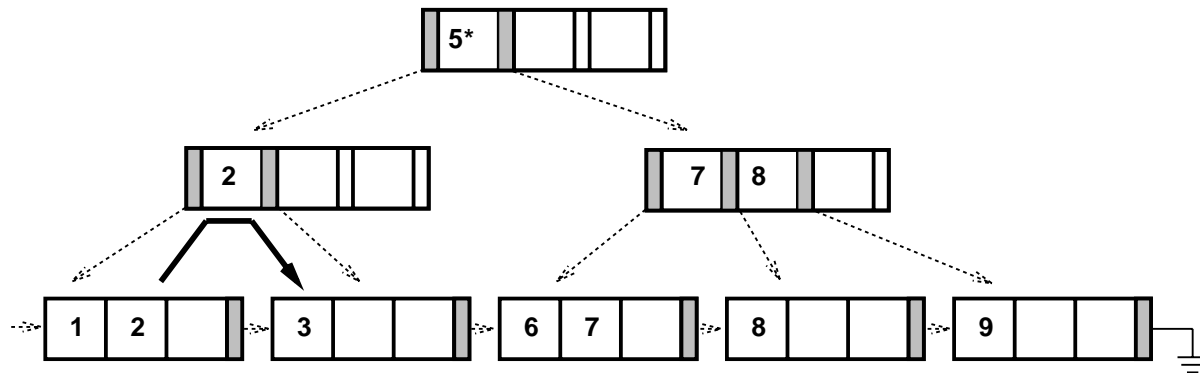


- Delete '5':
  - Search for '5'.

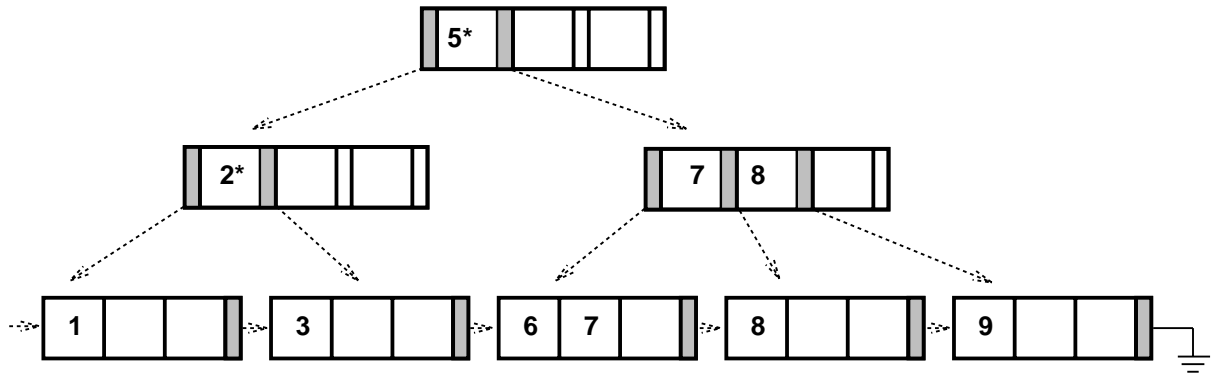
- '5' occurs as internal value  $\Rightarrow$  mark internal value.
- Deleting '5' from leaf does not cause underflow.



- Delete '4':
  - Search for '4'.
  - '4' does not occur as an internal value.
  - Deletion from leaf causes underflow  $\Rightarrow$  borrow from sibling.
  - Rotate from sibling:

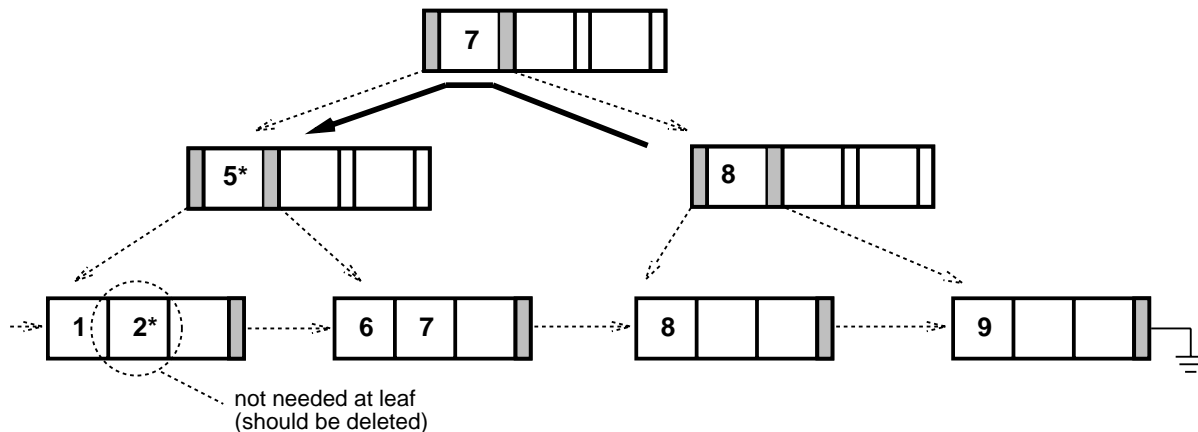


- Delete '2':
  - Search for '2'.
  - '2' occurs as an internal value  $\Rightarrow$  mark '2'.
  - Deletion from leaf does not cause overflow.

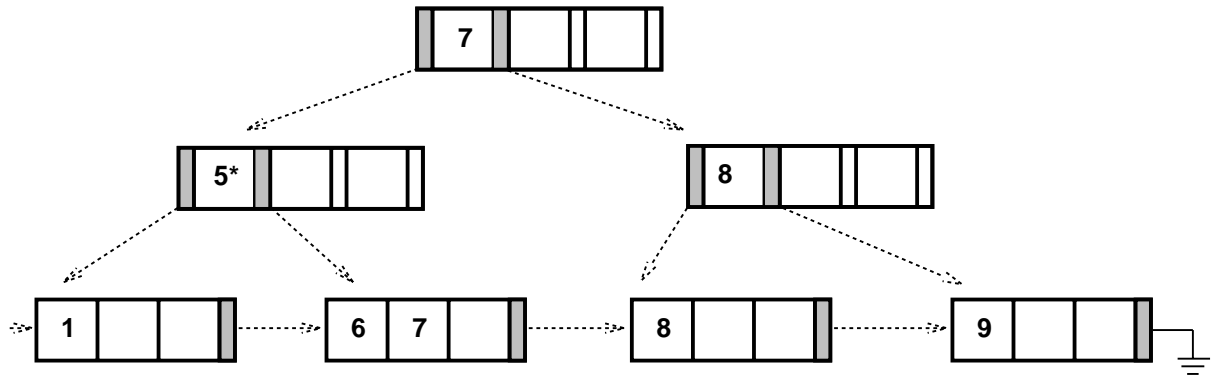


- Delete '3':

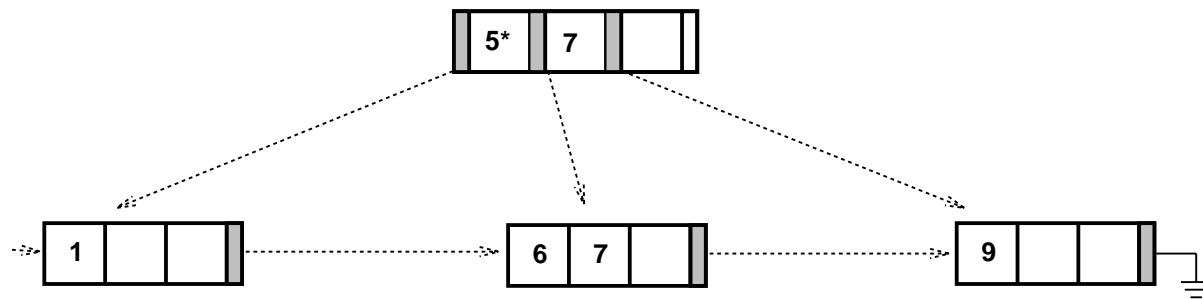
- Search for '3'.
- '3' does not occur as internal value.
- Deletion from leaf causes underflow  $\Rightarrow$  borrow from sibling
- Borrow from sibling not possible  $\Rightarrow$  Merge needed.
- Pull-down from parent causes underflow at parent  $\Rightarrow$  borrow from parent's sibling:



- Marked values are not needed at leaf level (they have no data pointers)  $\Rightarrow$  they should be deleted.



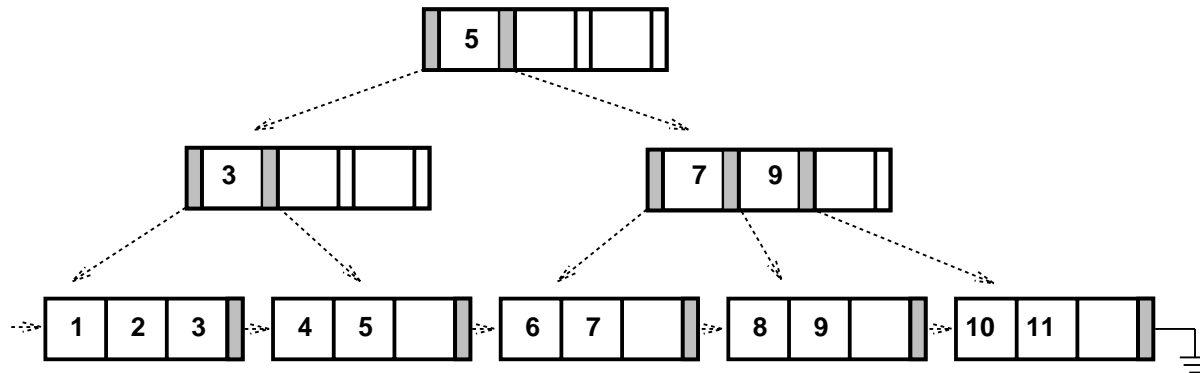
- Delete '8':
  - Search for '8'.
  - '8' occurs as an internal value  $\Rightarrow$  mark '8'.
  - Deletion from leaf causes underflow  $\Rightarrow$  try borrowing.
  - Borrow causes sibling underflow  $\Rightarrow$  pull down '8\*' from parent.
  - Underflow at parent's level  $\Rightarrow$  try borrowing at parent's level.
  - Borrow not possible at parent's level  $\Rightarrow$  pull down '7'.
  - Final tree:



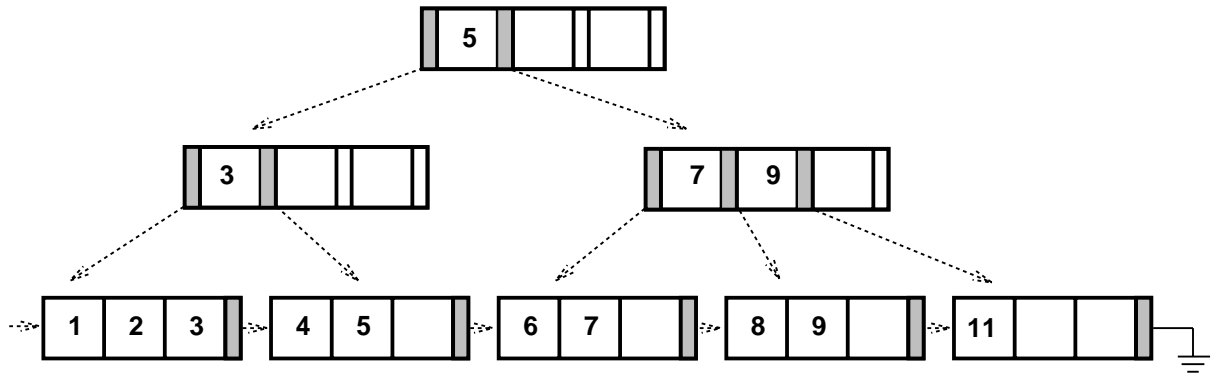
## Full Deletion in a B+-Tree

- Consider this example (degree  $m = 2$ ):

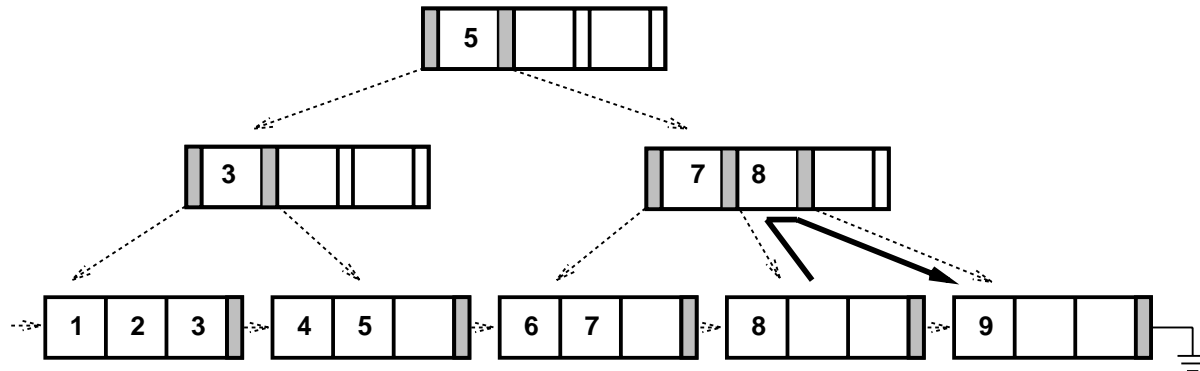
Delete 10, 11, 5, 4, 2, 3, 8 from:



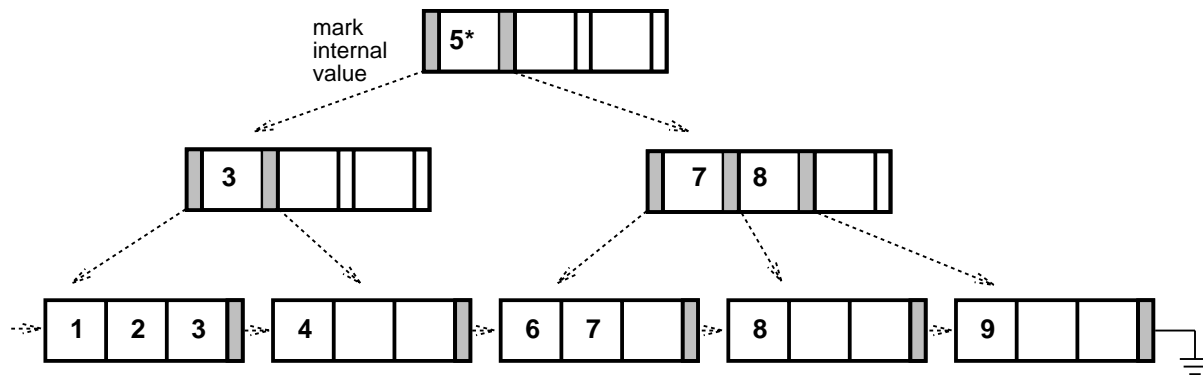
- Delete '10':
  - Search for '10'.
  - '10' does not occur as an internal value.
  - Deletion from leaf does not cause underflow.



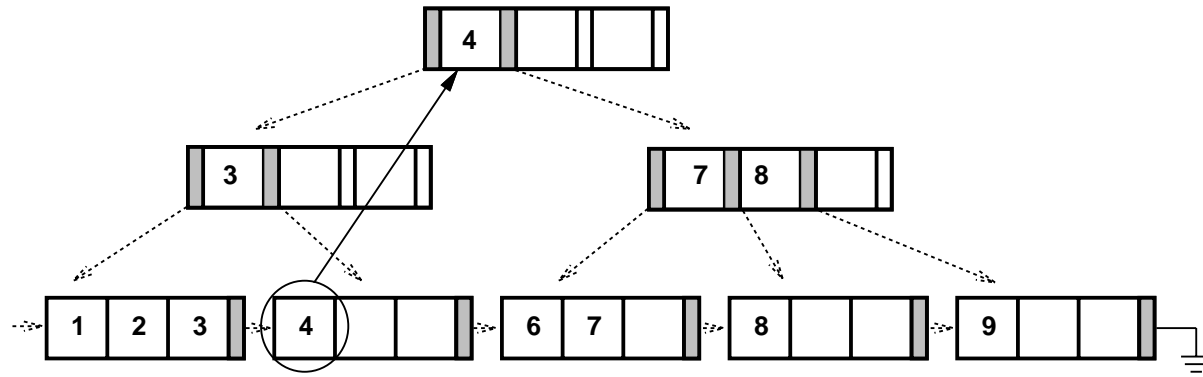
- Delete '11':
  - Search for '11'.
  - '11' does not occur as an internal value.
  - Deletion from leaf causes underflow  $\Rightarrow$  try to borrow from sibling.
  - To borrow, rotate from sibling  $\Rightarrow$  when moving '9' over, copy of '8' is placed into parent:



- Delete '5':
  - Search for '5'.
  - '5' occurs as internal value  $\Rightarrow$  mark internal value.
  - Deleting '5' from leaf does not cause underflow.

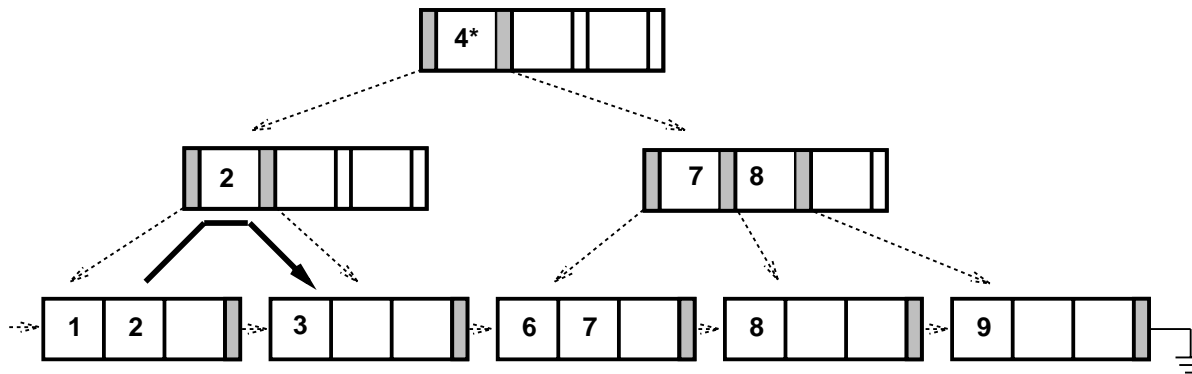


- Replace internal value with copy of predecessor (rightmost leaf value in left subtree).

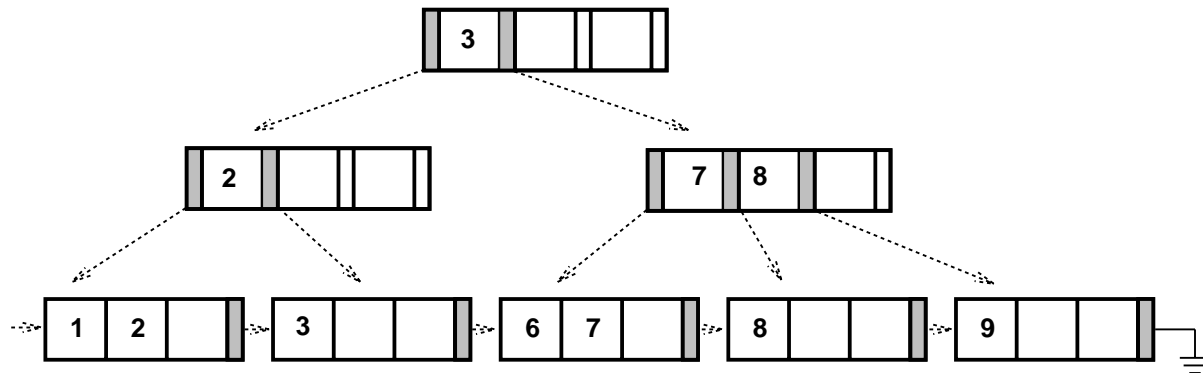


- Delete '4':

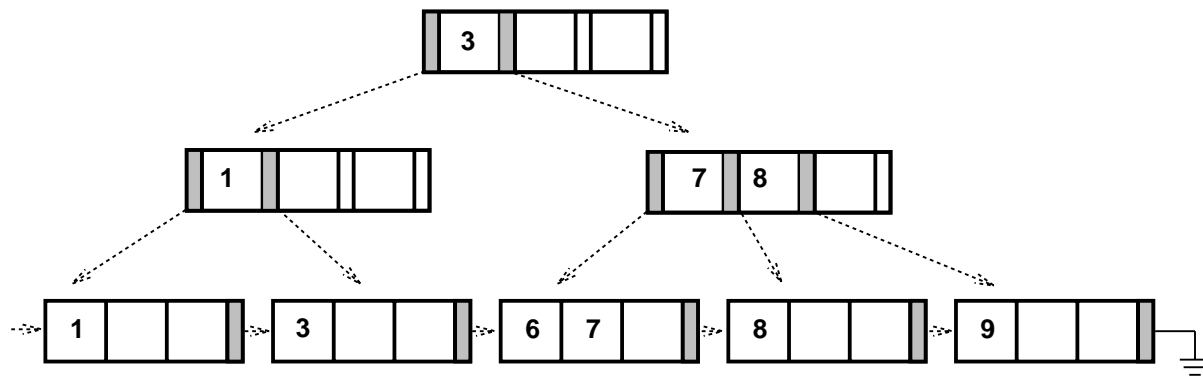
- Search for '4'.
- '4' occurs as an internal value  $\Rightarrow$  mark it.
- Delete '4' from leaf  $\Rightarrow$  need to rotate from sibling:



– Replace internal value by predecessor:

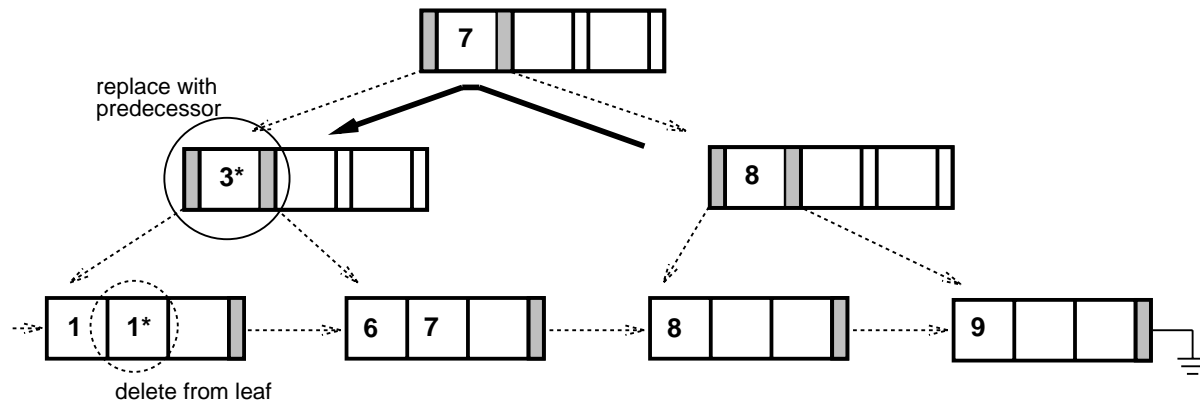


- Delete '2':
  - Search for '2'.
  - '2' occurs as an internal value  $\Rightarrow$  mark it.
  - Deletion from leaf does not cause underflow.
  - After leaf deletion, replace internal value by predecessor ('1'):

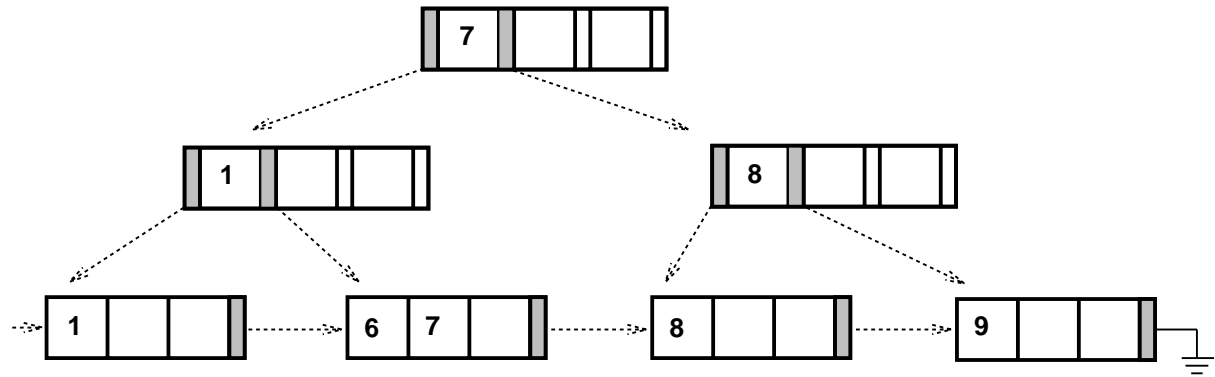


- Delete '3':

- Search for '3'.
- '3' occurs as an internal value  $\Rightarrow$  mark it.
- Deletion from leaf causes underflow  $\Rightarrow$  try borrowing.
- Borrowing causes underflow in sibling  $\Rightarrow$  pull down '1'.
- Pulling down '1' causes parental underflow  $\Rightarrow$  borrow at parent's level.

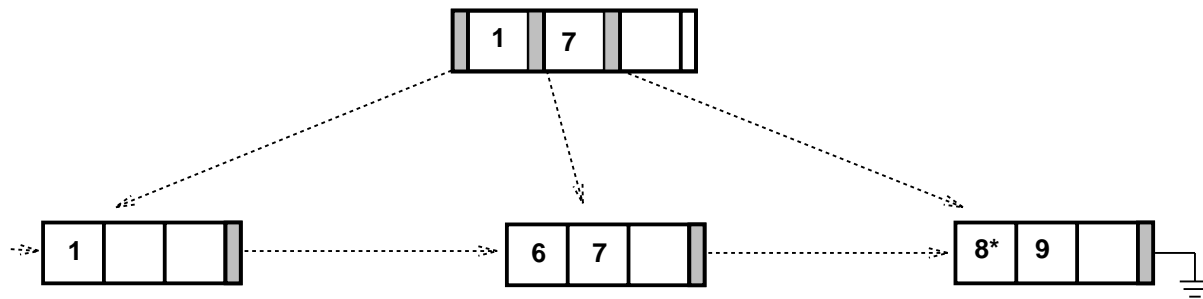


- Internal value of '1' does not belong in leaf  $\Rightarrow$  delete it.
- Replace internal value of '3' with predecessor ('1'):

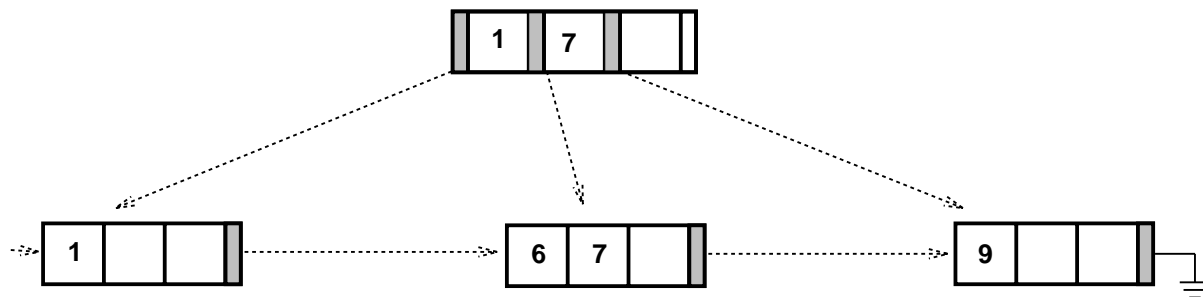


- Delete '8':

- Search for '8'.
- '8' occurs as an internal node  $\Rightarrow$  mark it.
- Deletion of '8' causes underflow  $\Rightarrow$  try to borrow from sibling
- Borrowing from sibling causes underflow  $\Rightarrow$  pull down parent.
- Borrowing at parent's level causes underflow  $\Rightarrow$  pull down '7'.

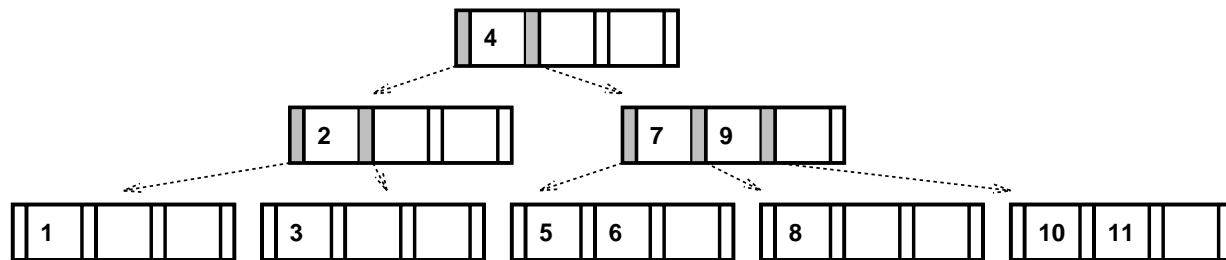


- Remove deleted internal value from leaf:

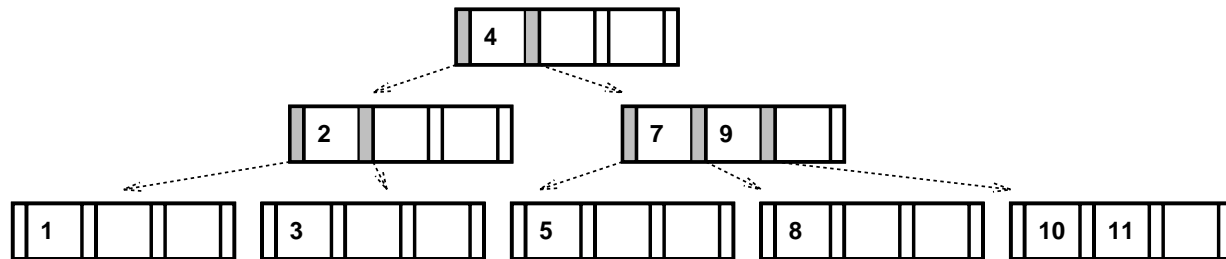


## Deletion in a B-Tree

- Deletion is somewhat more complicated than insertion. To see why, consider:

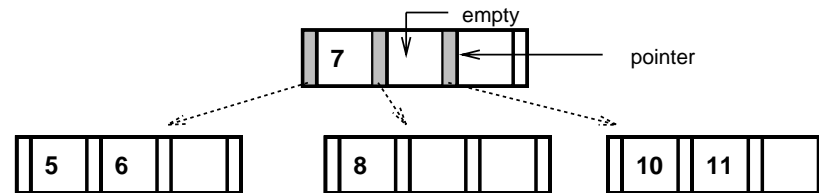


- Suppose we want to delete '6'  $\Rightarrow$  easy – simply delete '6' from '5-6' node:



- Suppose we want to delete '9':
  - \* Cannot delete '9' casually.

\* A search for '11' would cause problems  $\Rightarrow$  problem is worse for large  $m$ .



– Suppose we want to delete '2'  $\Rightarrow$  need to have at least  $m - 1$  elements  $\Rightarrow$  node will *underflow*.

- Key ideas used in deletion:

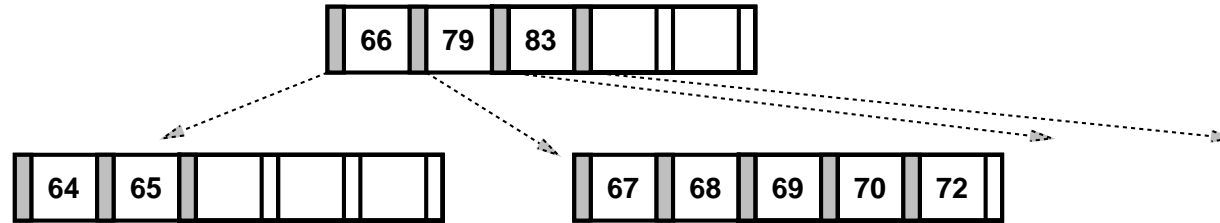
- *Rotation*: when underflow occurs, try to borrow key values from sibling via rotation.

- *Merging*: if rotation cannot be done, merge siblings.

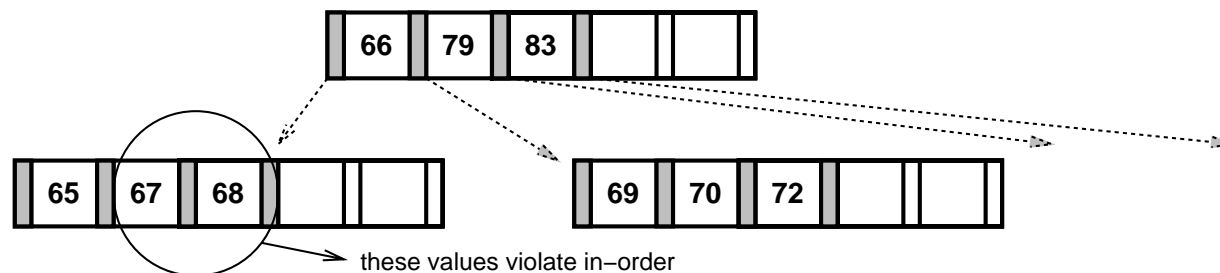
- *Replacement*: when deleting a value from an interior node, replace it with another key value.

- Rotation:

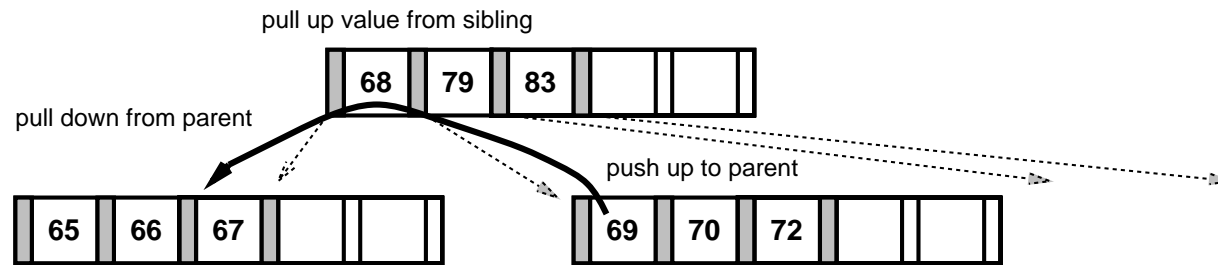
- Consider an example where  $m = 3 \Rightarrow$  must have at least  $m - 1 = 2$  values per node.



- Deleting '64' violates the lower limit.
- But, right sibling has enough values to re-distribute.
- Values are re-distributed via rotation (including the parent).
- Note: we cannot ignore parent value in rotation (since that would violate in-order).



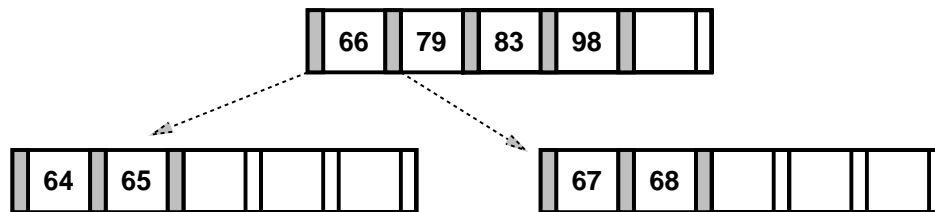
- Instead, parent value must be included in rotation.



– NOTE: both data pointers and tree pointers need to be moved in a rotation.

● Merging:

– Sometimes, you can't borrow from a sibling because the sibling doesn't have enough:

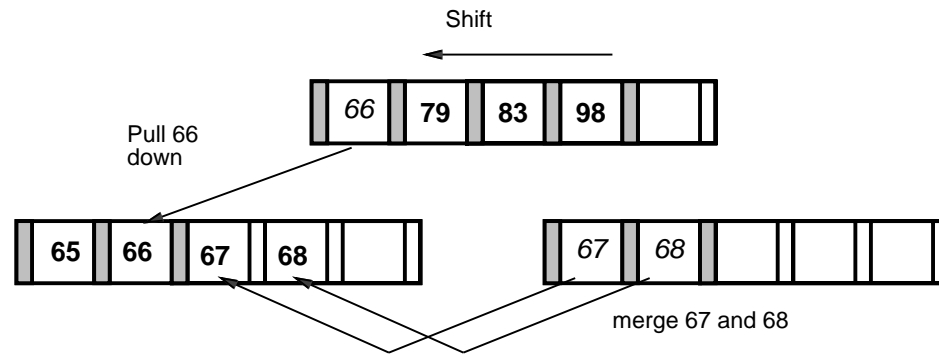


– Here, rotation from right sibling would cause right sibling to underflow.

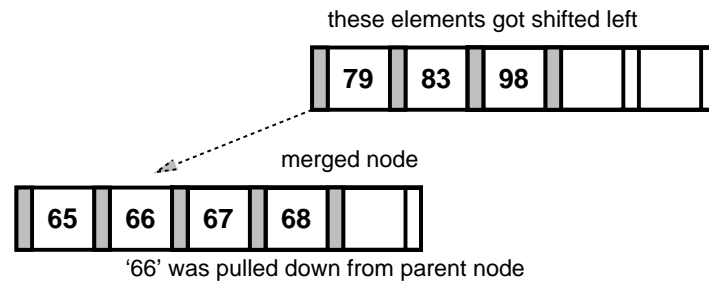
– Key idea: since both siblings are 'borderline', they both have

$m - 1$  values  $\Rightarrow$  together they have  $2m - 2$  values  $\Rightarrow$  together with parent value they have  $2m - 1$  values  $\Rightarrow$  can merge all of these into *a single node*.

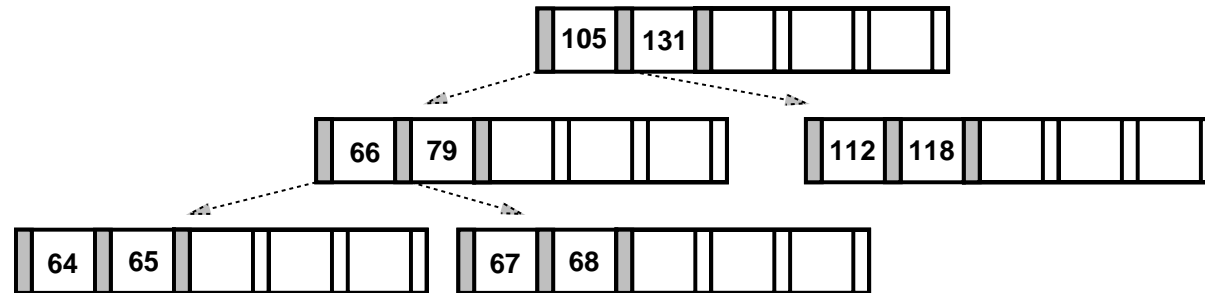
- Method: pull '66' down and merge siblings:



Result:



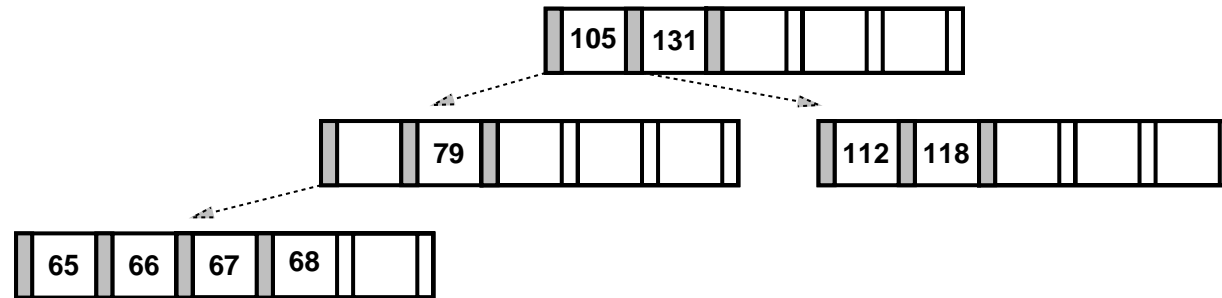
- But what if pulling down a value from the parent causes underflow at parent's level?  $\Rightarrow$  balance or merge at parent's level ... and so on, recursively.
- Example: delete '64' from this tree:



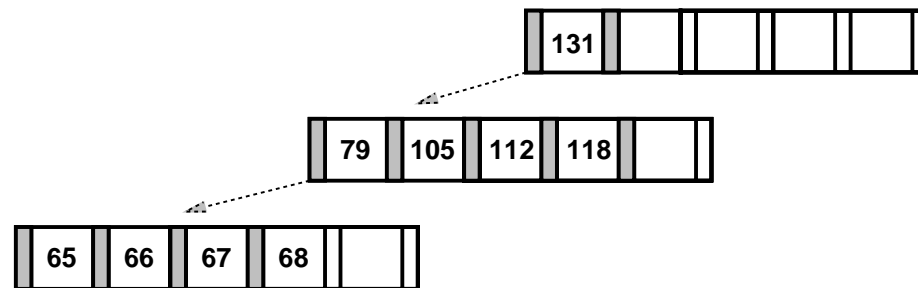
Here,

- \* Deleting '64' causes underflow in '64-65' node.
- \* Sibling is borderline  $\Rightarrow$  pull down '66' from parent  $\Rightarrow$  causes underflow in parent node  $\Rightarrow$  try to re-distribute among parent and parent's sibling.
- \* Parent's sibling is borderline  $\Rightarrow$  must merge parent and parent's sibling.

When we pull down '66', we get:



When we pull down '105' we get:

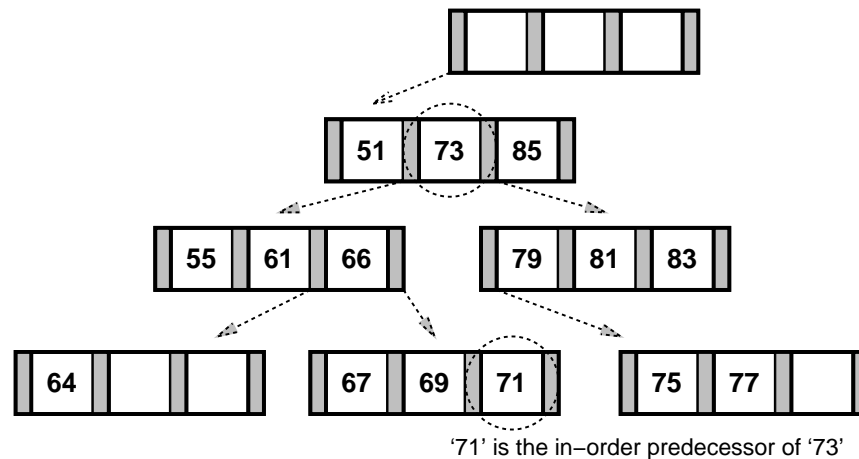


NOTE: if '105' was the only element in the root, we would have one less level.

NOTE: In the above example (and some that follow), the entire tree is not shown for lack of space.

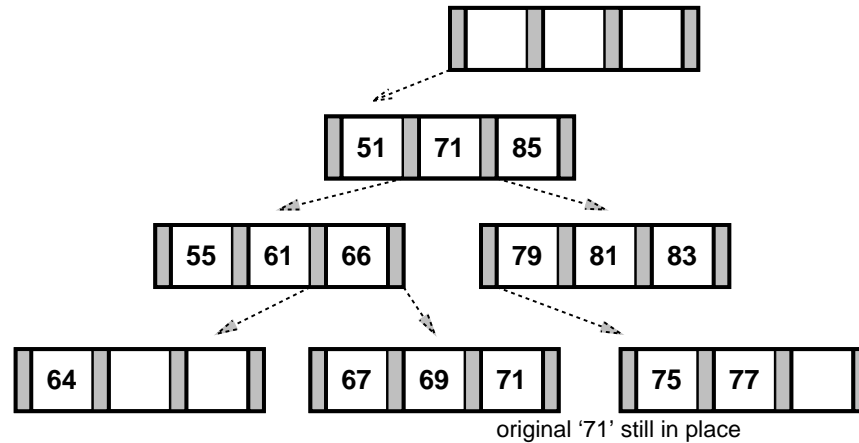
- Replacement:

- Deletion of an interior node value requires replacing the value.
- The value is replaced by the in-order *predecessor*.
- Finding the in-order predecessor requires a search all the way down to the leaf level.
- Example (with  $m = 2$ ): Delete '73' from this B-tree



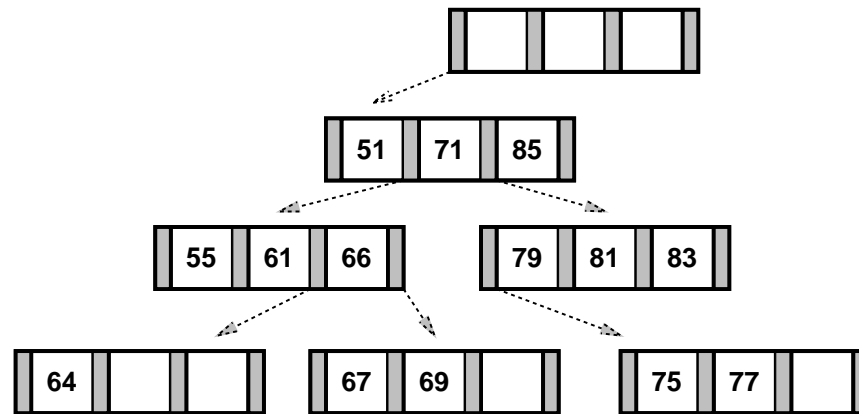
- The in-order predecessor of '73' is the largest element in the *left* subtree of '73'.  $\Rightarrow$  rightmost element in left subtree.

- Note: an interior value always has a left subtree.
- Replace '73' with '71'.



- Note: after replacement, in-order is maintained.

- Finally, do a regular delete of ‘71’:



- Note: in some cases, the final delete could recursively work its way up to the root. But it does not use *replacement* and so must finally end.
- At last we see why the underflow restriction is  $m - 1$  (as opposed to something higher)
  - ⇒ if the restriction were higher (such as 70% full), then how would we merge two nodes that are 70% full?
  - (It can be done, but it’s harder).

- We will not cover **delete** any further. For additional details, see the literature.

## Implementing a B-Tree

- Implementation issues:
  - How to store data and pointers in blocks?
  - Designing algorithms for insertion and search.

The following functions will be used to handle I/O in the pseudocode:

- **DISK-READBLOCK** (bnum)
  - This function reads block bnum into memory and returns the address.
- **DISK-NEWBLOCK** ()
  - Creates a new block on disk and returns a block number.

- A separate **DISK-READBLOCK** is needed to read the block into memory.
- **DISK-WRITEBLOCK** (bnum) – Writes block bnum to disk.

- Recall: B-tree blocks contain
  - key values (at most  $2m - 1$ )
  - data pointers (at most  $2m - 1$ )
  - tree pointers (at most  $2m$ )

It is useful to store additional information:

- A boolean indicating whether the block is a leaf.
- The number of entries (key-values) in a block.

Note: If a block has  $k$  key values, then it has  $k + 1$  tree pointers.

- If the degree  $m$  is known in advance, we can use a C-like **struct** for a B-tree block with fields, e.g.,

```

struct btree_node {
    int num_entries;
    int leaf;
} // M = degree
// Number of entries
// Leaf or interior node
```

```

    char value [KEYSIZE] [2*M-1];    // Values
    int blocknum [2*M-1];            // Together, blocknum
    int tuplenum [2*M-1];            // tuplenum constant
    int treep [2*M];                 // One additional
};

```

Now, typically, the system reads a block into memory and returns its address. In C, a *cast* may be used to extract fields:

```

    b = (btree_node *) Disk_Readblock (blocknum);
    if (b->leaf) {
        ... etc
    }

```

Casting will not be shown in the pseudocode that follows.

- Typical function specifications for creating, searching and in-

serting:

**Creating a B-tree:** `btree_create (tuplesize, keysize, keyoffset)`

- `tuplesize` is the length (in bytes) of the data tuple.
- `keysize` is the length of the key field.
- `keyoffset` is the offset (in bytes) of the key field.
- The function initializes the root block and stores (in private data structures) the `tuplesize` etc.
- Most often, a *file identifier* is returned:  
`fd = btree_create (tuplesize, keysize, keyoffset)`  
to be used later in searching or insertion: `btree_search (fd, key)`

**Searching a B-tree:** `tuple = btree_search (key)`

- `key` is a particular key value (e.g., the character string

‘Smith’ for a NAME field).

- The function typically returns the entire tuple (if found) or NULL.  $\Rightarrow$  the function retrieves the tuple from the data file.

### **Inserting a tuple: `btree_insert (tuple)`**

- `tuple` is the tuple to be inserted.
  - Often, such functions return the data block (of the heapfile) in which the tuple was inserted.
- Pseudocode for these operations are given in notes