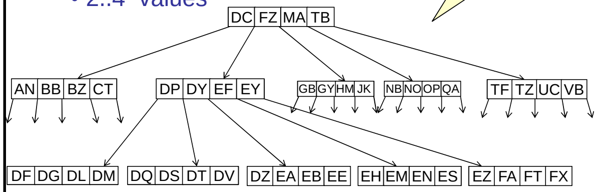
# B+ Trees

## B Trees

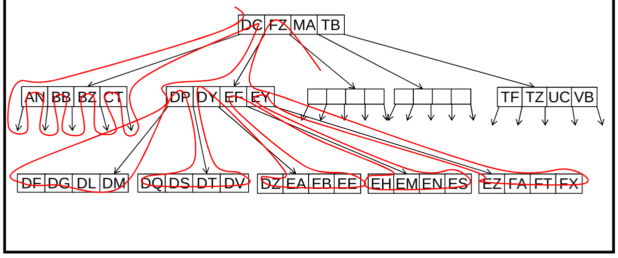
- Example of a B tree with m = 5
  - nodes contain
    - 3..5 children
    - 2..4 values

Maximally full depth 3 tree
(Can't fit all the nodes on slide)



## Traversing a B Tree

- Listing all the items in order is a bit messy:
  - Have to constantly return to higher nodes.



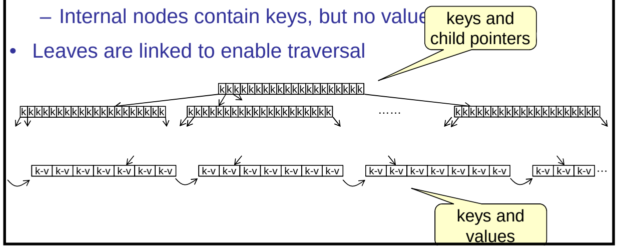
# Sets versus Key-Value pairs

What are the items we are storing?

- Set:
  - B-Tree contains a set of values
  - Only need to store the values you are searching on.
- Map / Dictionary / database table
  - B-Tree contains a set of key-value pairs
  - Search down the tree governed only by the keys
  - Need to store each value with its key
    - ⇒ B-tree nodes can't have as many keys
    - ⇒ lower branching factor
    - ⇒ deeper trees
  - If node is of fixed capacity and value is large (eg, a whole record from a database table) then may only fit a very few key-value pairs in a node.

# B+ Trees

- The most commonly used variant of B Trees.
- Intended for storing key–value (or key–record) pairs.
- Leaves contain key-value pairs,
- Keys are repeated in the internal nodes.
  - Internal nodes contain keys, but no value
- Leaves are linked to enable traversal



# B+ Trees: Leaves

- Each leaf node contains
  - between $\lceil max_L /2 \rceil$ and $max_L$ key-value pairs,
  - a link to the next leaf in the tree

$K_0$-$V_0$, $K_1$-$V_1$, $K_2$-$V_2$, …, $K_{leaf.size-1}$-$V_{leaf.size-1}$

  - For each key $K_i$ in the leaf :

    $K_i < K_{i+1}$

  - The value might be either
    - the actual associated value (if it is small)
    - the index of a data block where value can be found (maybe in another file)

## B+ Trees:  Internal Nodes

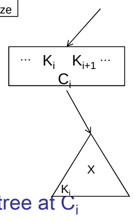
Except root may have fewer

- Each internal node contains
  - between $\lfloor max_N /2 \rfloor$ and $max_N$  keys,  and
  - up to $max_N$ +1  child node indexes

$C_0$, $K_1$ $C_1$, $K_2$ $C_2$, … $C_{node.size-1}$ $K_{node.size}$ $C_{node.size}$

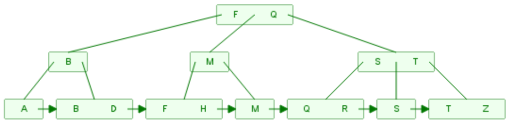
  - Branching factor = $max_N$ +1
  - For each key  X  in the subtree at $C_i$ :

    $K_i \leq X < K_{i+1}$
  - $K_i$  will the be the leftmost key in the subtree at $C_i$ ie, the first key in leftmost leaf of $C_i$



## B+ example

- 3-degree, Add in order: M H T S R Q B A F D Z

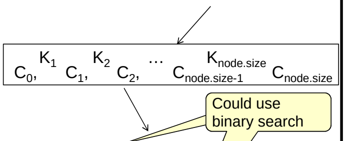


## B+ Tree:  Find

To find value associated with a key:

```
Find(key):
    if root is empty    return null
    else return Find(key, root)

Find(key, node):
    if node is a leaf
        for i from 0 to node.size-1
            if  key = node.keys[ i ]   return node.values[ i ]
        return null
    if node is an internal node
        for  i from 1 to node.size
            if  key < node.keys[ i ] return Find(key, getNode(node.child[i-1]))
        return Find(key, getNode(child[node.size] ))
```



Could use binary search

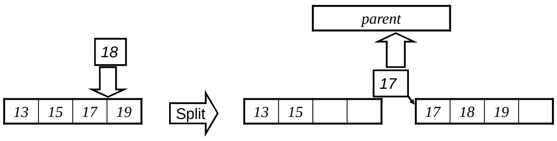
$K_0$-$V_0$,  $K_1$-$V_1$,  $K_2$-$V_2$,  …,  $K_{leaf.size-1}$-$V_{leaf.size-1}$

## B+ Tree  Add

- Find leaf node where item belongs
- Insert in leaf .
- If node too full,
  split, and promote middle key up to parent, middle key also go to the right
- If root split, create new root containing promoted key

## Splitting a B$^+$-Tree Leaf

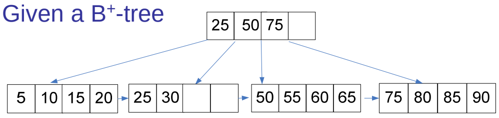
- If a leaf overflows:
  - The left most *m* keys are left in the node,
  - The right most *m + 1* keys are moved to a new node,
  - The (*m* + *1*)-st key is propagated to the parent node



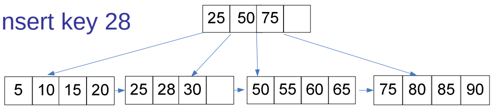
- A non leaf node splits as in a common B-tree
- The right sub-tree of each non leaf node contains greater or equal key values

## B$^+$-Tree Insertion Example

- Given a B$^+$-tree

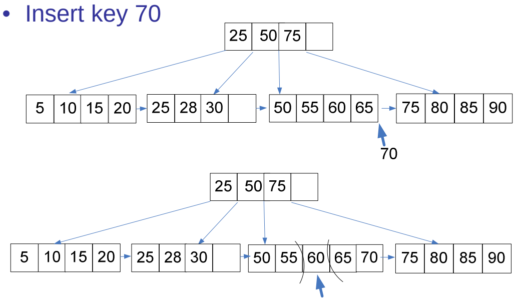


- Insert key 28



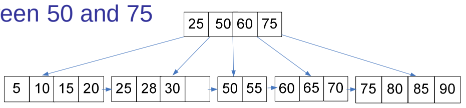
## $B^+$-Tree Insertion Example (cont.)

- Insert key 70

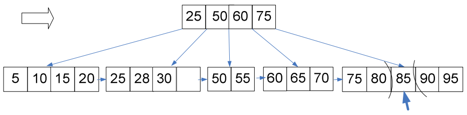


## $B^+$-Tree Insertion Example (cont.)

- The middle key of 60 is placed in the node between 50 and 75

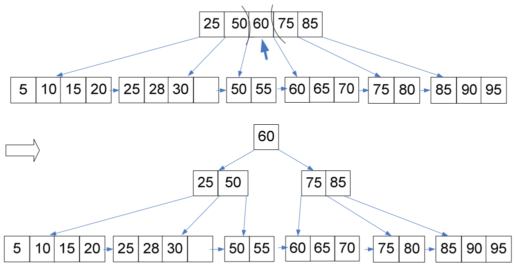


- Insert 95



## $B^+$-Tree Insertion Example (cont.)

- Split the leaf and promote the middle key to the parent node

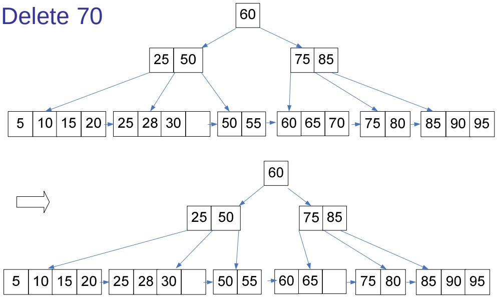


## $B^+$-Tree Deletion

- When a record is deleted from a $B^+$-tree it is always removed from the leaf level
- If the deletion of the key does not cause the leaf underflow
  - Delete the key from the leaf
  - If the key of the deleted record appears in an index node, use the next key to replace it
- If deletion causes the leaf and the corresponding index node underflow
  - Redistribute, if there is a sibling with more than *m* keys
  - Merge, if there is no sibling with more than *m* keys
  - Adjust the index node to reflect the change

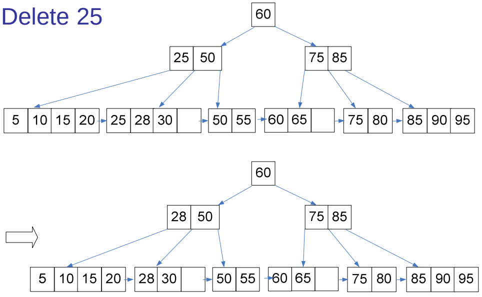
## $B^+$-Tree Deletion Example

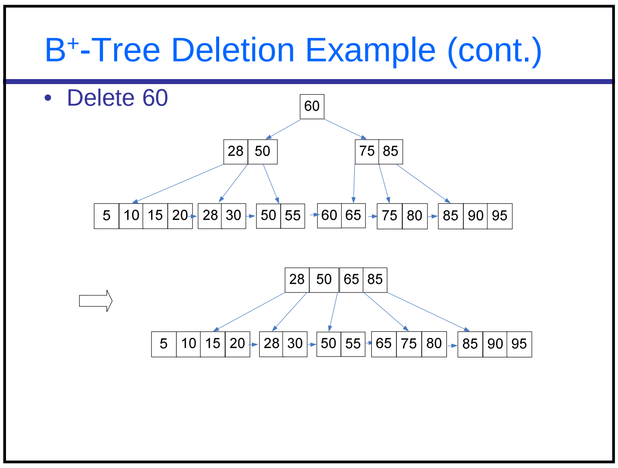
- Delete 70



## $B^+$-Tree Deletion Example (cont.)

- Delete 25





## B+-Tree Deletion Example (cont.)

- Delete 60