COMP261 Lecture 24

B and B+ Trees

Victoria
UNIVERSITY OF WELLINGTON
Te Whare Wānanga
o te Ūpoko o te Ika a Māui
CAPITAL CITY UNIVERSITY

B Trees are great – can we do better?

- B-Trees a great when contains a set of values
  - Only need to store the values you are searching on.

- In file/database setting:
  - Need to store key-value pairs
  - Search down the tree governed only by the keys
  - Values usually much larger than keys (eg, a whole record from a database table)
  - Storing values with keys reduces number of keys in each node (assuming fixed size nodes).

    ⇒ lower branching factor

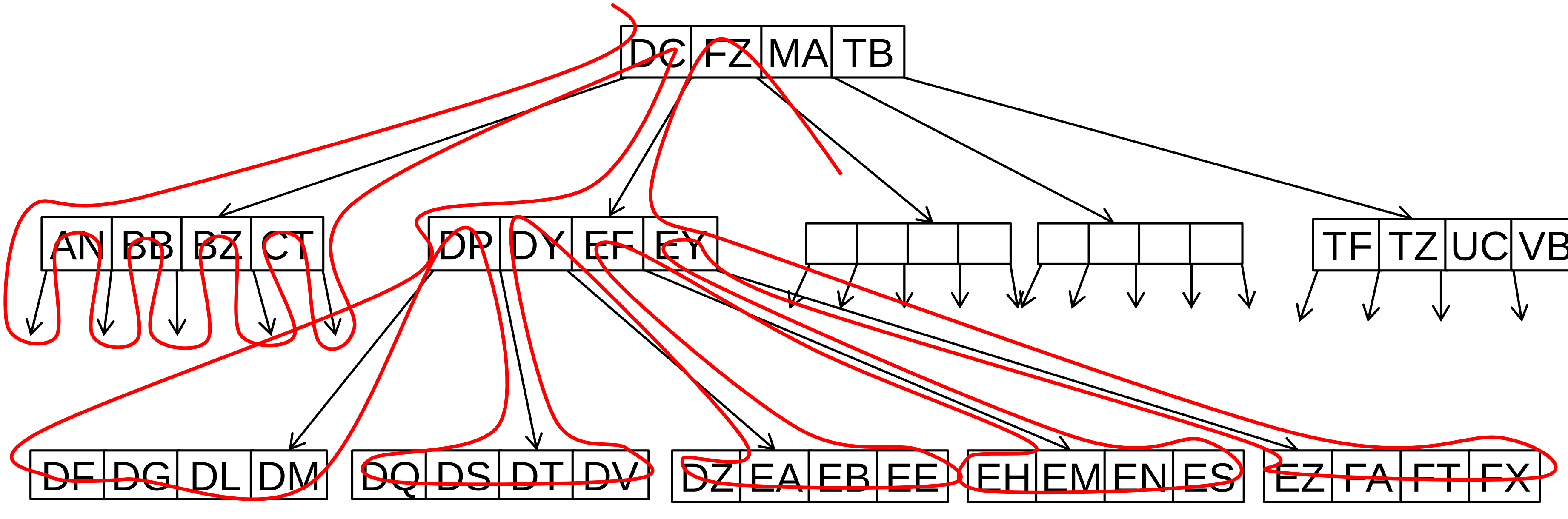
    ⇒ deeper trees

    ⇒ slower access times

- What if we want to index on more than one key?

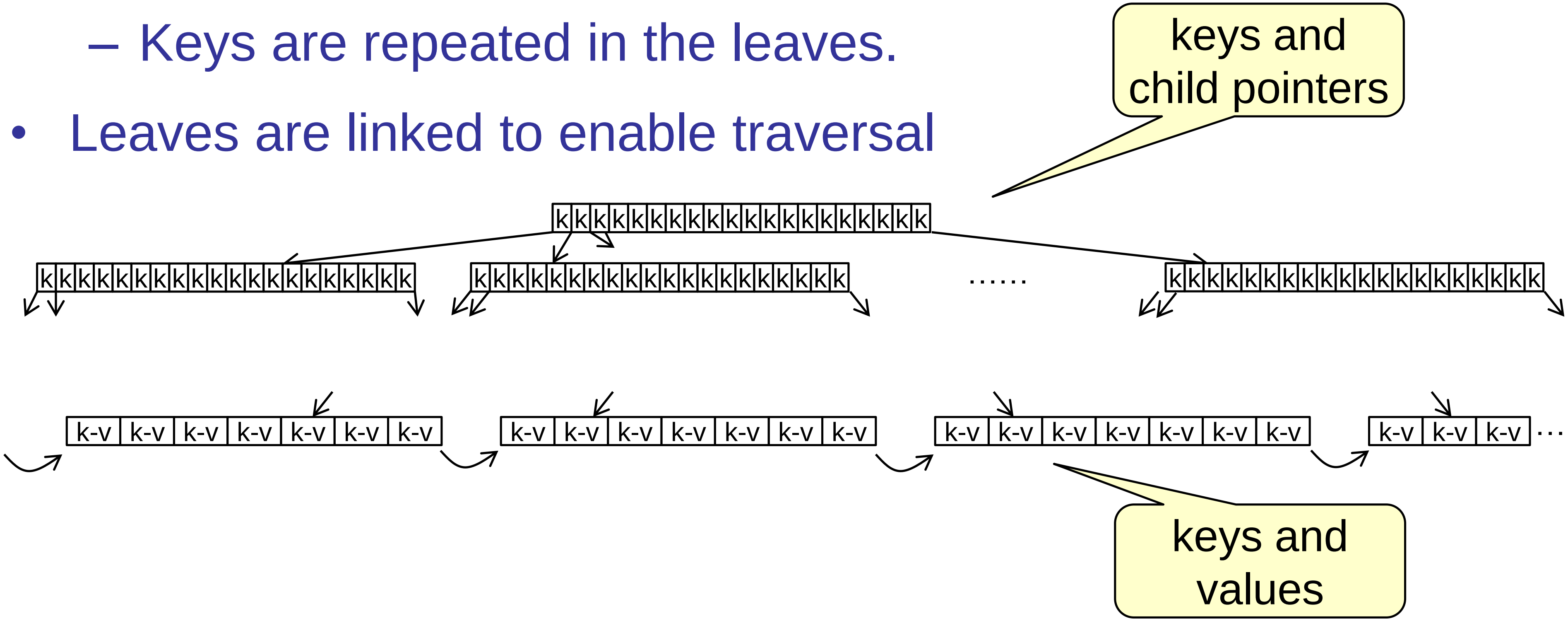
Traversing a B Tree

- Listing items in order is expensive:
  - Moving up and down the tree means lots of record accesses.



B+ Trees

- Commonly variant of B Trees – used in many DB/file systems.
- Intended for storing key–value (or key–record) pairs in files.
- Leaves contain key-value/record/index pairs.
- Internal nodes only contain (some) keys – only for searching.
  - Keys are repeated in the leaves.
- 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, …, $K_{leaf.size-1}$-$V_{leaf.size-1}$

- Keys are ordered: for each key K_i in the leaf :

$$K_i < K_{i+1}$$

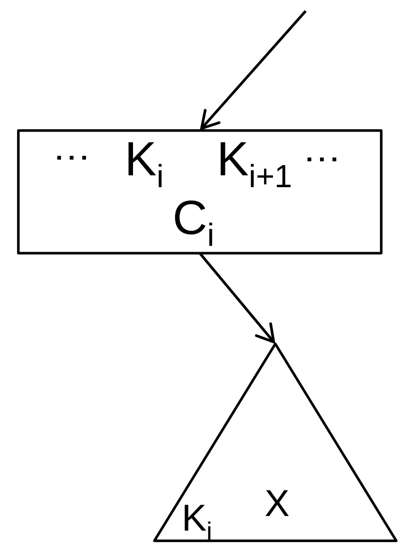
- The value might be either
  - the actual associated value/record (if small enough)
  - the index of a data block where value can be found (maybe in another file)

B+ Trees: Internal Nodes

Except root may have fewer

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

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



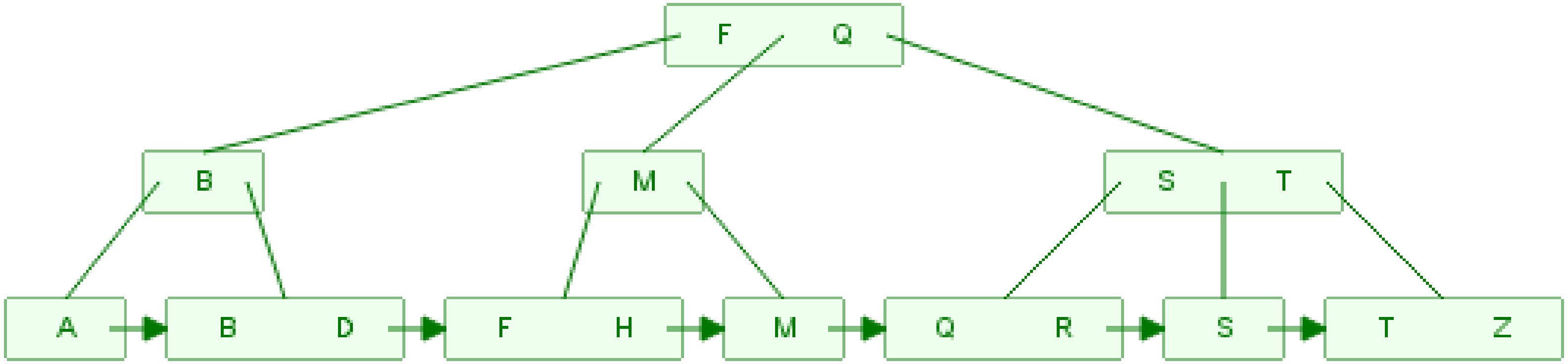
- Branching factor = $max_N +1$
- Keys act as separators for subtrees.
  - For each key X in the subtree at C_i :

$$K_i \leq X < K_{i+1}$$

  - K_i is 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)

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

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

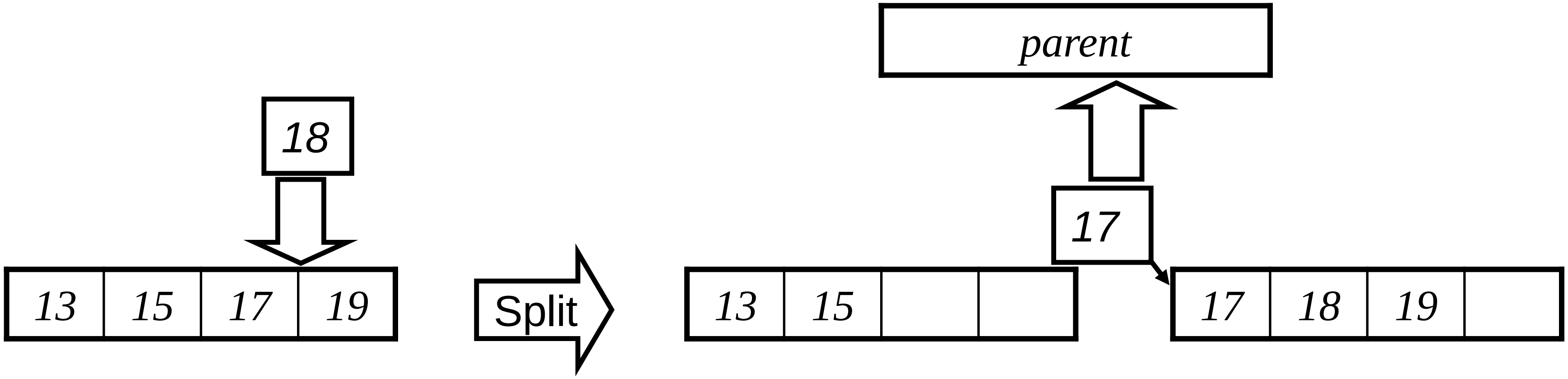
$K_0\text{-}V_0, \; K_1\text{-}V_1, \; K_2\text{-}V_2, \; \ldots, \; K_{leaf.size-1}\text{-}V_{leaf.size-1}$

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 goes to the right
- If root split, create new root containing promoted key

Splitting a B$^+$-Tree Leaf

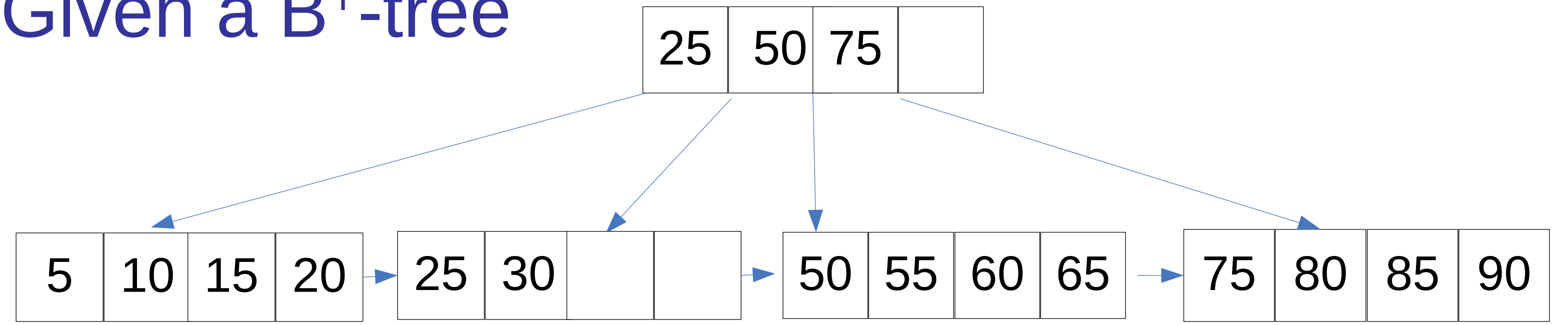
- If a leaf overflows:
  - Leave the left most *m* keys in the node,
  - Move the right most *m + 1* keys to a new node,
  - Propagate the (*m + 1*)-st key to the parent node



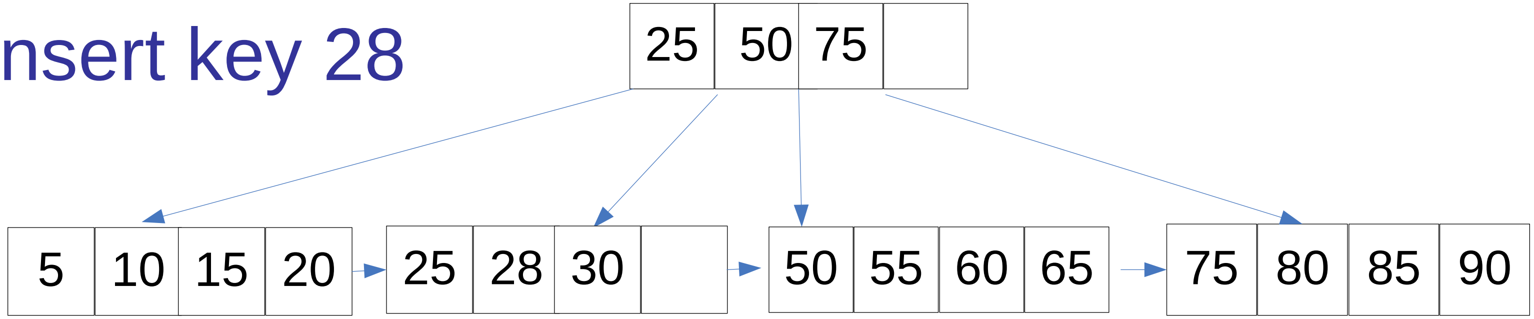
- A non leaf node splits as in an ordinary B-tree
- The right sub-tree of each non leaf node contains greater or equal key values

B$^+$-Tree Insertion Example

- Given a B$^+$-tree

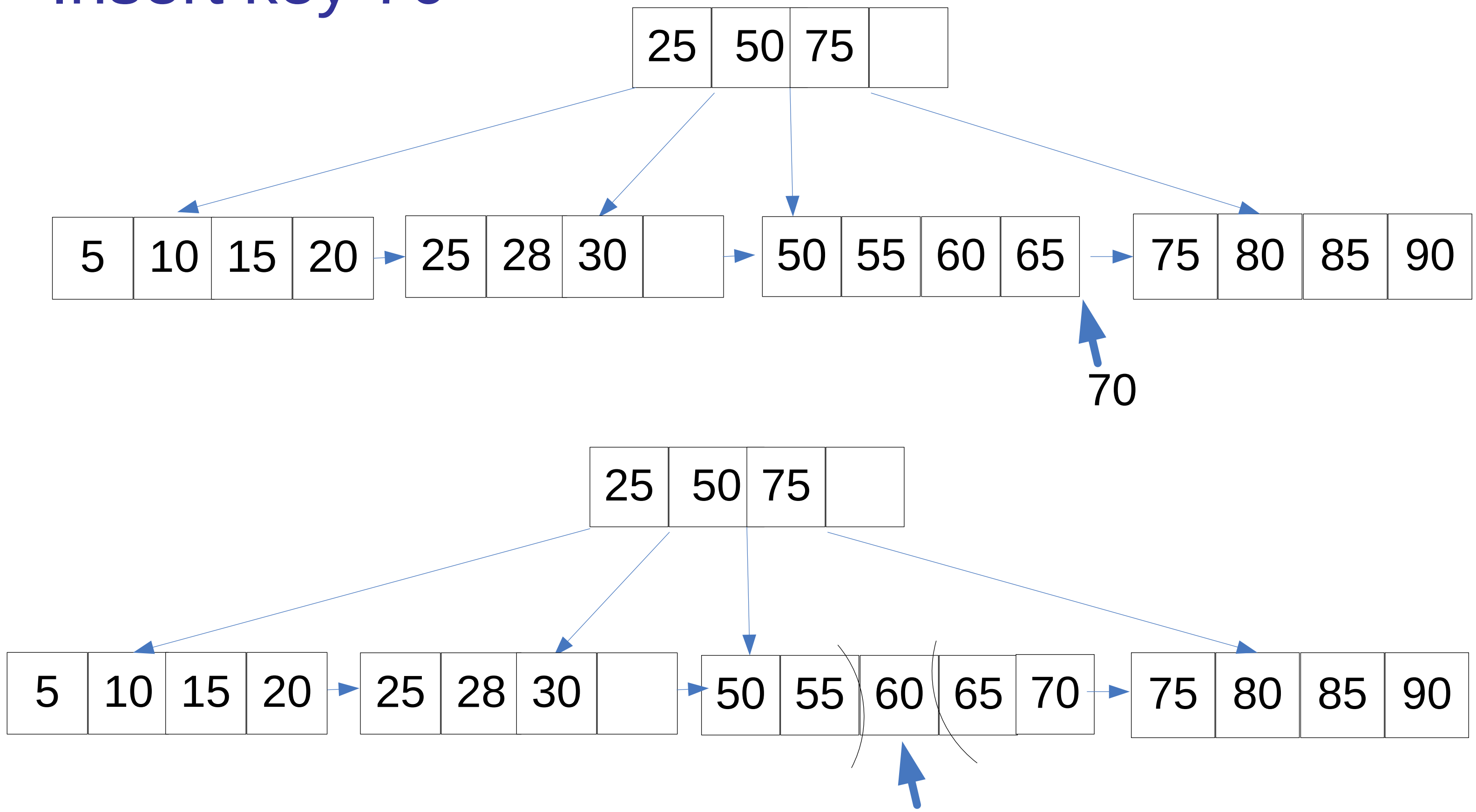


- Insert key 28



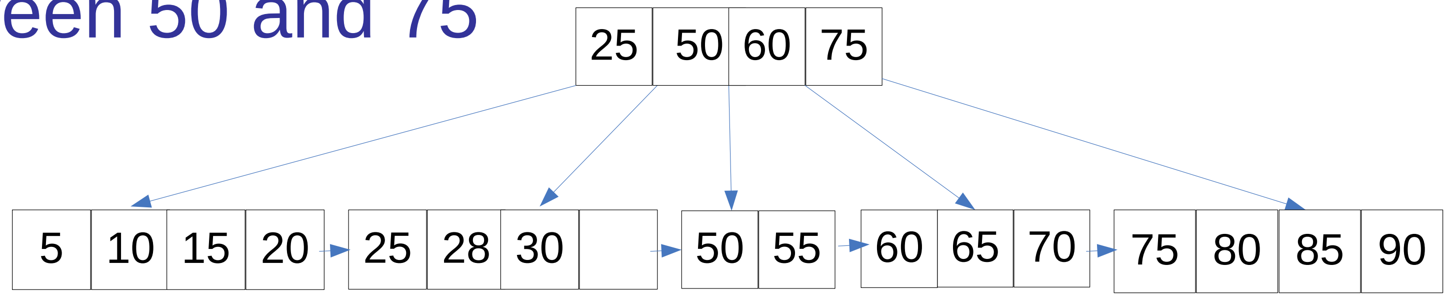
B^{+}-Tree Insertion Example (cont.)

- Insert key 70

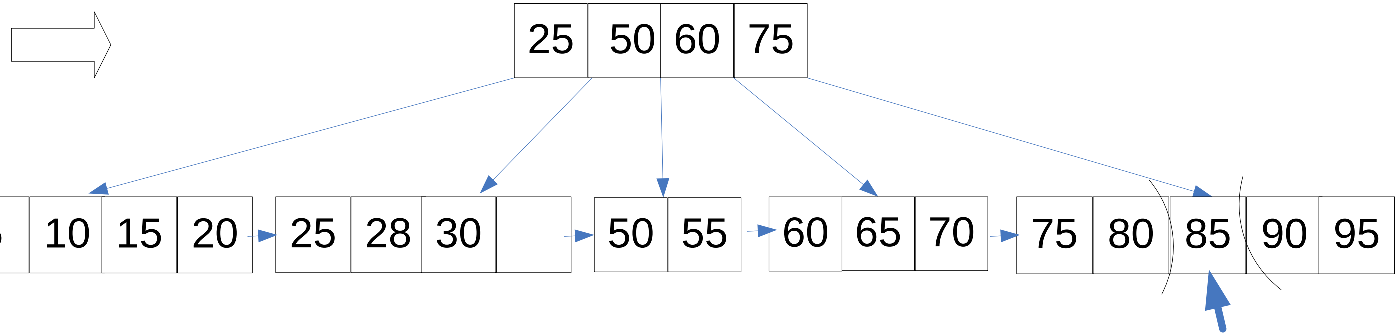


B$^+$-Tree Insertion Example (cont.)

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

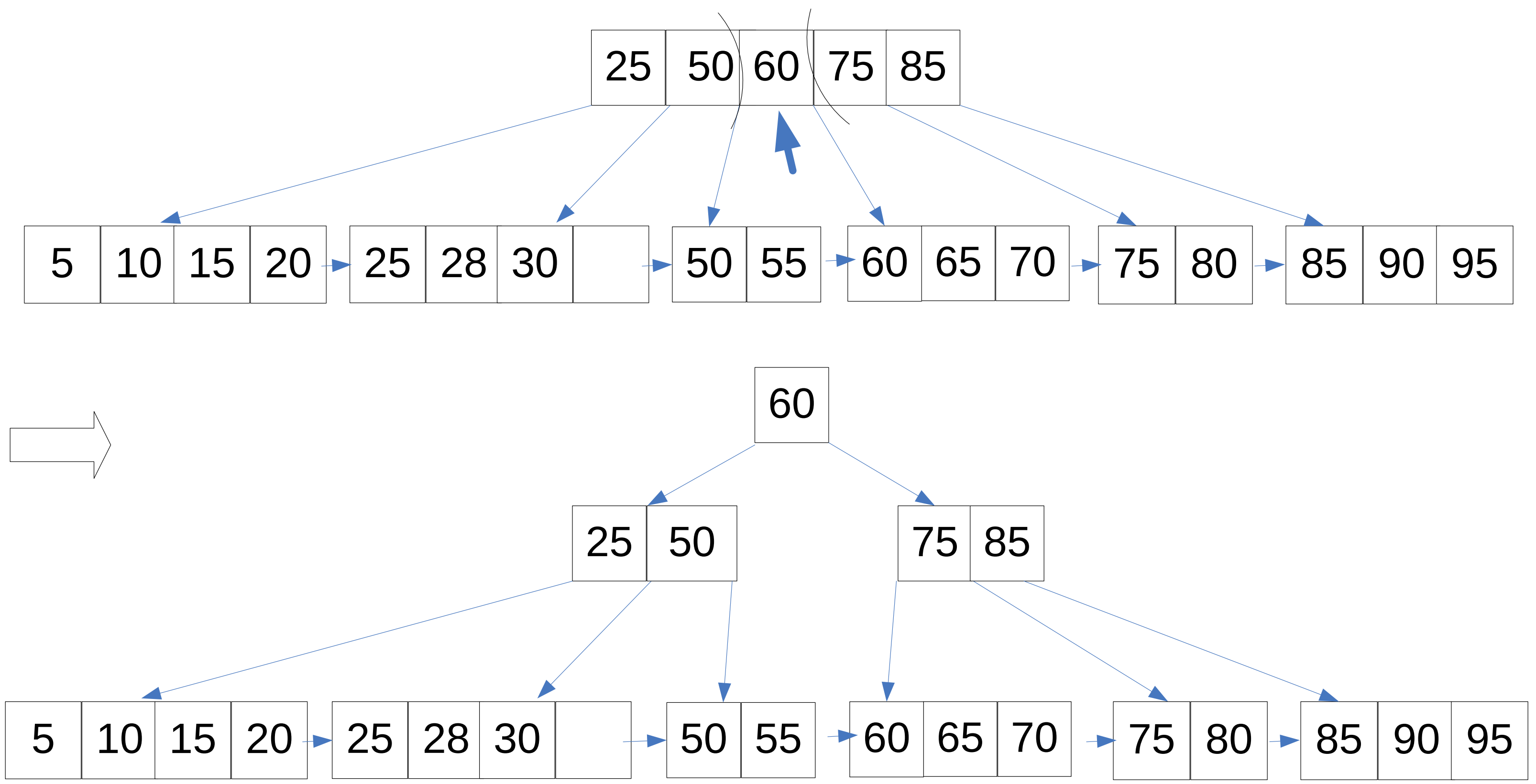


- Insert 95



B⁺-Tree Insertion Example (cont.)

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



B+ Tree Add (1)

<u>Add(key, value)</u>:

if root is empty
  create new leaf, add key-value,
  root ← leaf

If root was full:
returns new key
and new leaf node,

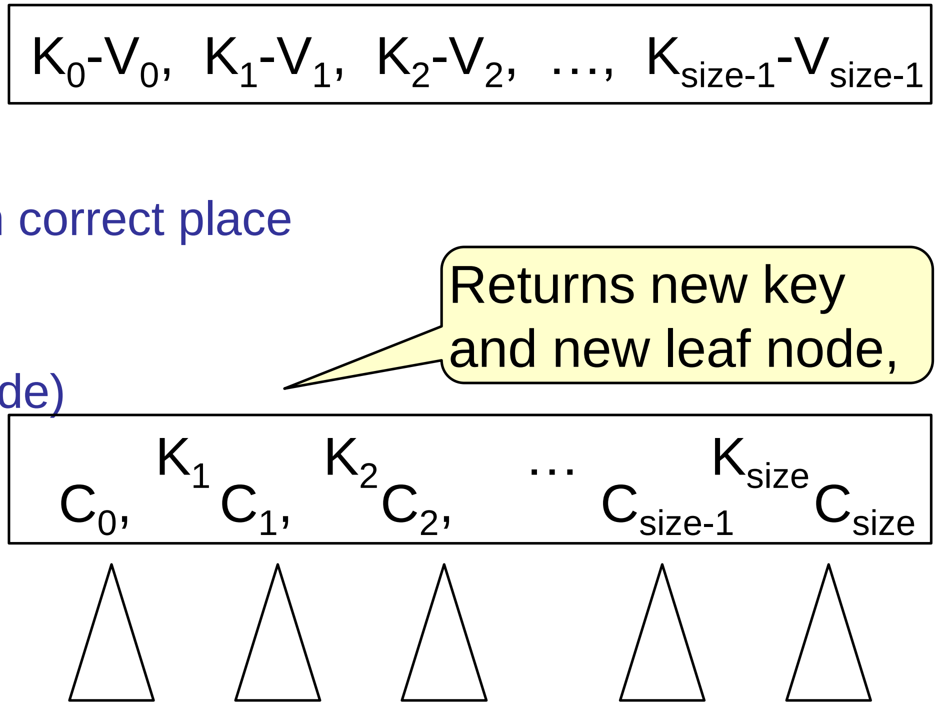
else
  (newKey, rightChild) ← <u>Add</u>(key, value, root)
  if (newKey, rightChild) ≠ null
    node ← create new internal node
    node.size ← 1
    node.child[0] ← root
    node.keys[1] ← newKey
    node.child[1] ← rightChild
    root ← node

Make a new
root node

B+ Tree Add (2)

```
Add(key, value, node):
    if node is a leaf
        if  node.size < maxLeafKeys
            insert  key and value into leaf in correct place
            return null
        else
            return  SplitLeaf(key, value, node)
    ________________________________________

    if node is an internal node
        for  i from 1 to node.size
            if  key < node.keys[i]
                (k, rc) ← Add(key, value, node.child[i-1])
                if (k, rc)=null   return null
                else   return dealWithPromote(k,rc,node)
        (k, rc) ← Add(key, value, node.child[node.size])
        if (k, rc)= null   return null
        else return  dealWithPromote( k, rc, node)
```



Inserts new key and child into node,

B+ Tree Add (3)

Could make the array one larger than necessary to give room for this.

SplitLeaf(key, value, node):

insert key and value into leaf in correct place (spilling over end)

sibling ← create new leaf

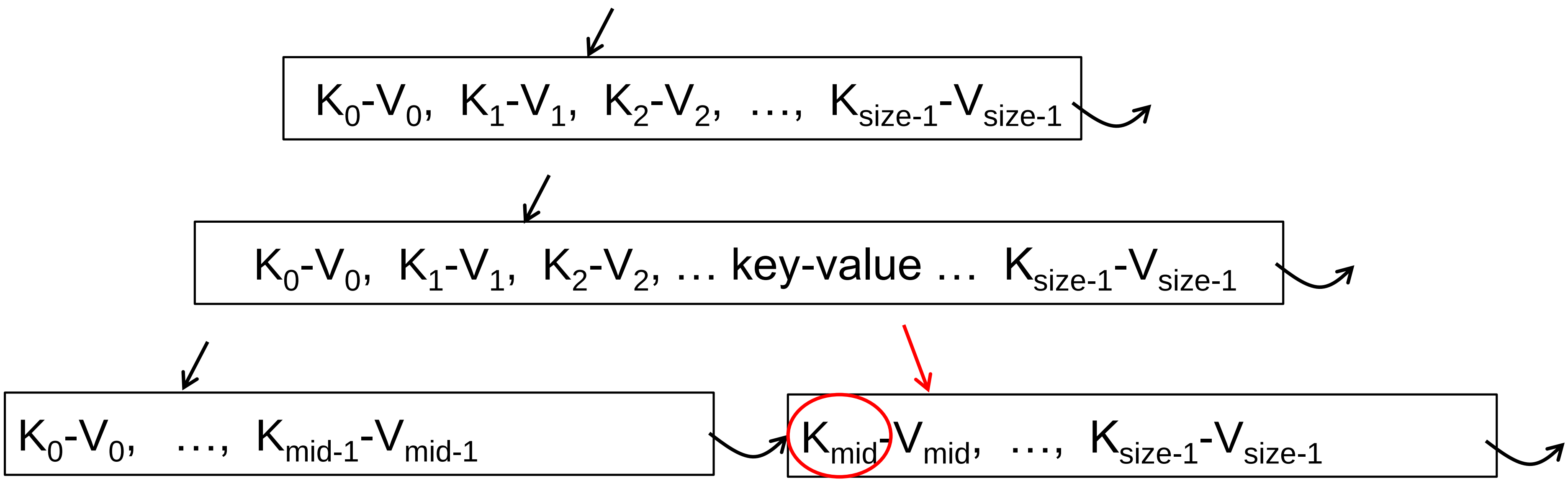
mid ← $\lfloor$(node.size+1)/2$\rfloor$

= $\lfloor$(max$_L$+2)/2$\rfloor$ since size is now max$_L$+1

move keys and values from mid … size out of node into sibling.

sibling.next ← node.next   node.next ← sibling

return (sibling.keys[0] , sibling)



B+ Tree Add (4)

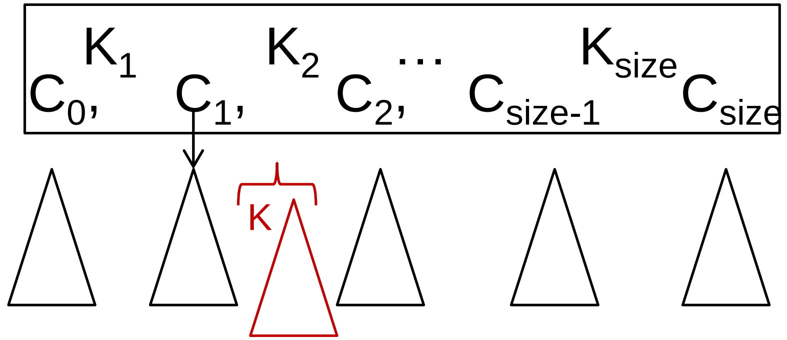
Nothing was promoted

DealWithPromote(newKey, rightChild, node):

if (newKey, rightChild) = null **return** null

if newKey > node.keys[node.size]
  insert newKey at node.keys[node.size+1]
  insert rightChild at node.child[node.size+1]

else for i from 1 to node.size
  if newKey < node.keys[i]
    insert newKey at node.keys[i]
    insert rightChild at node.child[i]



if size ≤ maxNodeKeys **return** null

No need to promote further

sibling ← create new node
mid ← ⌊size/2⌋ +1
move node.keys[mid+1… node.size] to sibling.node [1… node.size-mid]
move node.child[mid … node.size] to sibling.child [0 … node.size-mid]
promoteKey ←node.keys[mid],
remove node.keys[mid]
return (promoteKey, sibling)

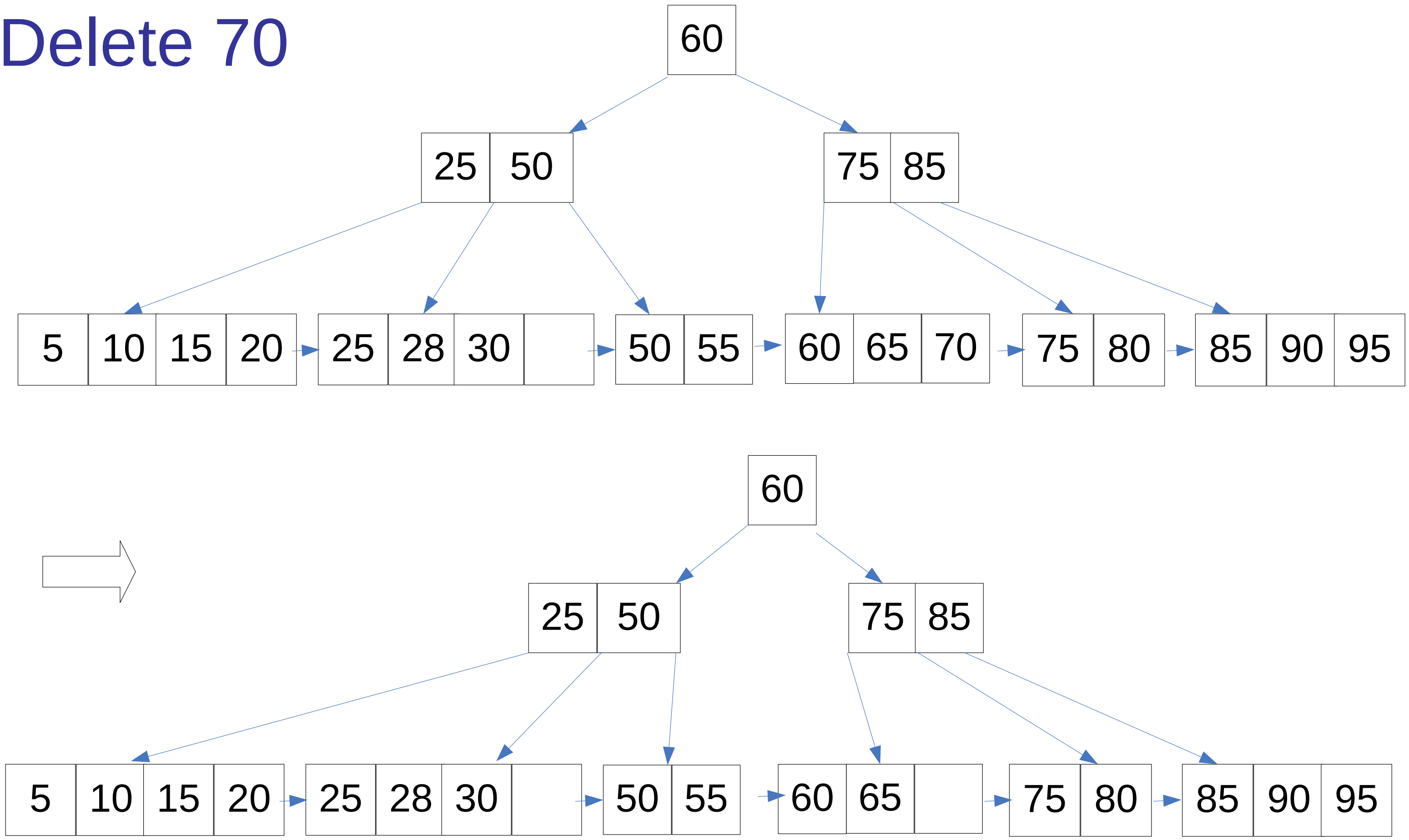
Node is overfull:
Have to split and promote

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 record does not cause the leaf underflow
  - 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

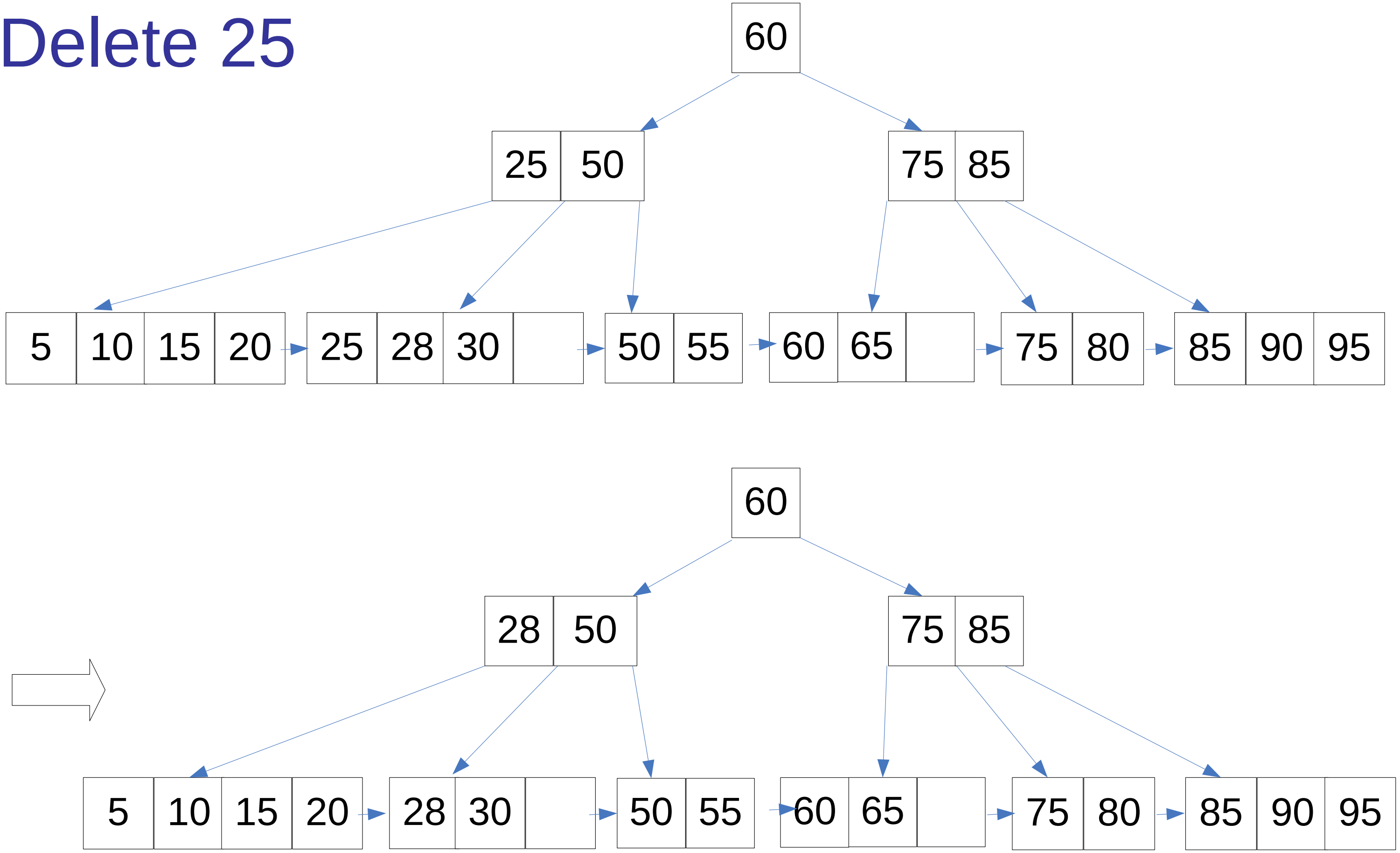
B+-Tree Deletion Example

- Delete 70



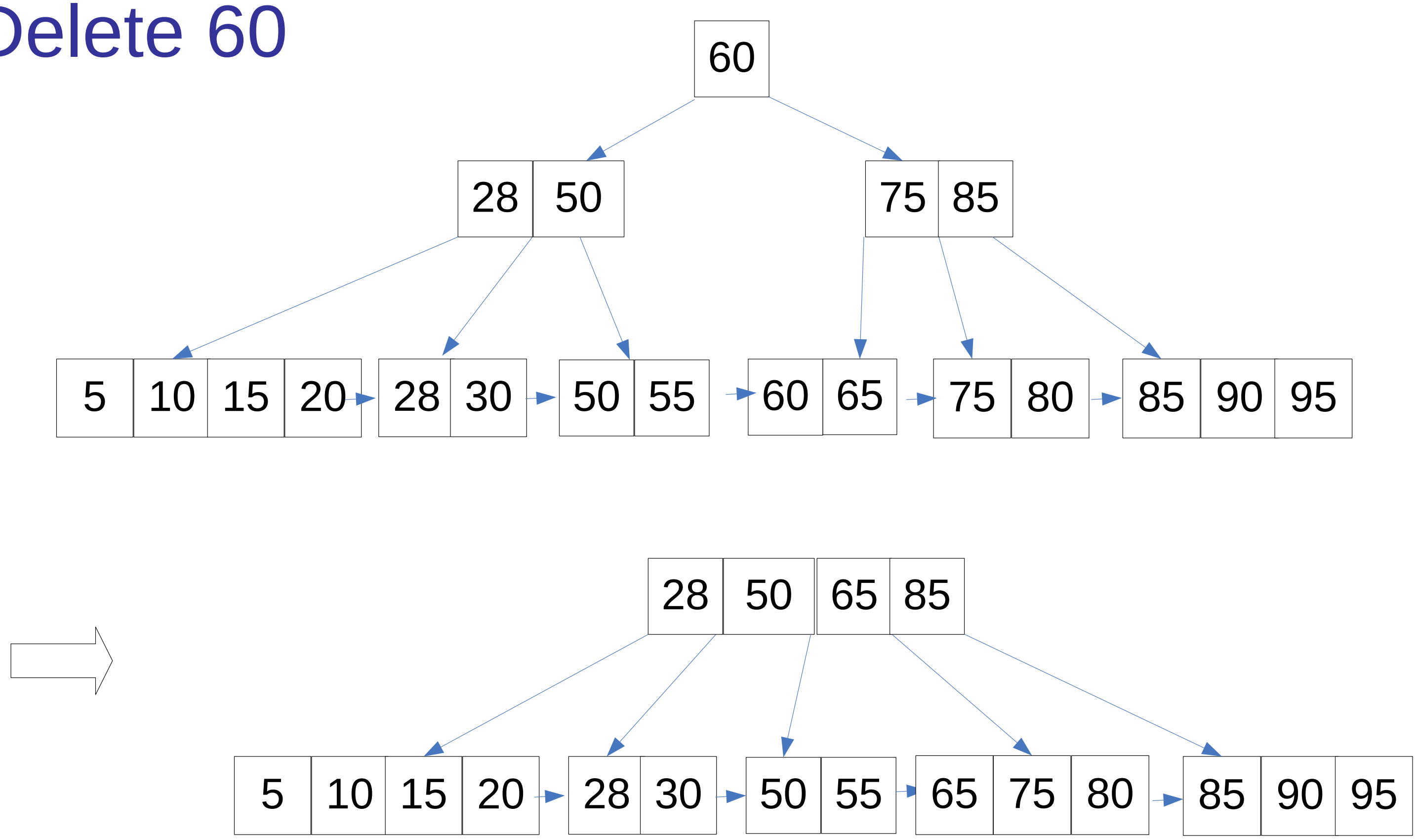
B+-Tree Deletion Example (cont.)

- Delete 25



B+-Tree Deletion Example (cont.)

- Delete 60



Using B+ trees for organising Data

- B+ tree is an efficient index for very large data sets
- The B+ tree must be stored on disk (ie, in a file)

  ⇒ costly actions are accessing data from disk
- Retrieval action in the B+ tree is accessing a node

  ⇒ want to make one node only require one block access

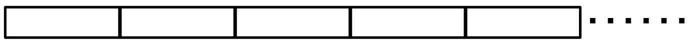
  ⇒ want each node to be as big as possible ⇒ fill the block

B+ tree in a file:

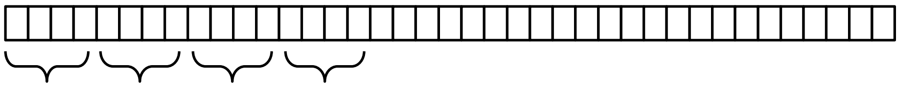
- one node (internal or leaf) per block.
- links to other nodes = index of block in file.
- need some header info.
- need to store keys and values as bytes.

Implementing B+ Tree in a File

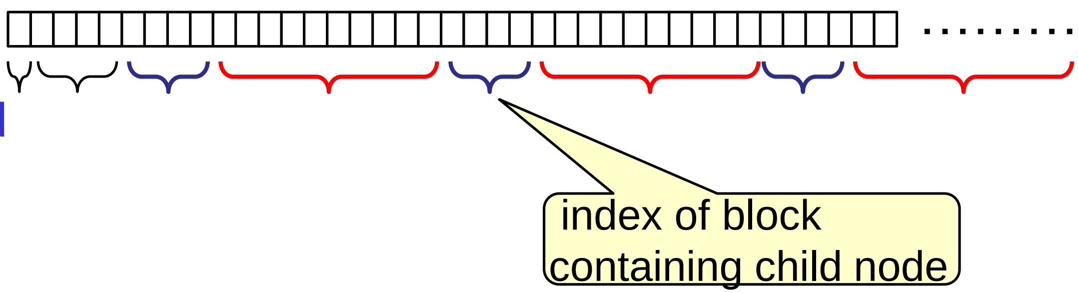
- Use a block for each node of the tree
  - Can refer to blocks by their index in the file.



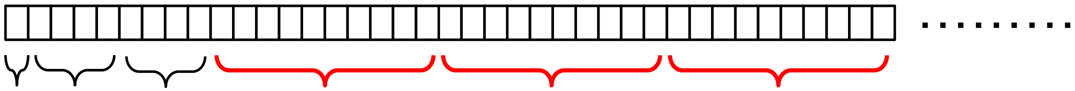
- Need an initial block (first block in file) with meta data:
  - index of the root block
  - number of items
  - information about the item sizes and types?



- Use a block for each internal node
  - type
  - number of items
  - child key child key…. key child



- Use a block for each leaf node
  - type
  - number of items
  - link to next
  - key-value key-value …. key-value



Cost of B+ tree

- If the block size is 1024 bytes, how big can the nodes be?
- Node requires
  - some header information
    - leaf node or internal node
    - number of items in node,
  - internal node:
    - m_N x key size
    - m_N+1 x pointer size
  - leaf node
    - m_L x item size
    - pointer to next leaf

Must specify which node, ie which block of the file

Leaf nodes may hold more values than internal nodes!

- How big is an item?
- How big is a pointer?

Cost of B+ tree: Example

- Suppose:
  - a block has 1024 bytes
  - each node has header ⇒ 5 bytes
  - a key is a string of up to 10 characters ⇒ 10 bytes
  - a value is a string of up to 20 characters ⇒ 20 bytes
  - a child pointer is an int ⇒ 4 bytes

- Internal node (m_N keys, m_N+1 child pointer)
  - size = 5 + (10 + 4) m_N + 4

    ⇒ $m_N \le (1024 - 9)/14 = 72.5$ ⇒ 72 keys in internal nodes

- Leaf node (with pointer to next)
  - size = 5 + 4 + (10 + 20) m_L

    ⇒ $m_L \le (1024 - 9)/30 = = 33.8$ ⇒ 33 key-value pairs in leaves