B-trees

- The B-tree guarantees a storage utilization of at least 50% (at least half of each allocated page actually stores index entries)
- In a B-tree, the cost of search is, even in the worst case, logarithmic in the index cardinality (that is, the number of search keys)
- In a B-tree, each node at *m* children, where *m* is the only parameter depending on the specific characteristics of storage, that is, the block dimension

B-trees

- A B-tree of order *m* ($m \geq 3$) is a balanced tree that verifies the following properties:
  - each node contains at most m - 1 elements
  - each node contains at least $\lceil m/2 \rceil - 1$ elements,
  - The root may contain a single element
- Each non leaf node containing j elements has j +1 children
- each node has a structure of the form:

  $p_0(k_1, r_1)p_1(k_2, r_2)p_2 \ldots p_{j-1}(k_j, r_j)p_j$

  where j is the number of the elements in the node

B-trees

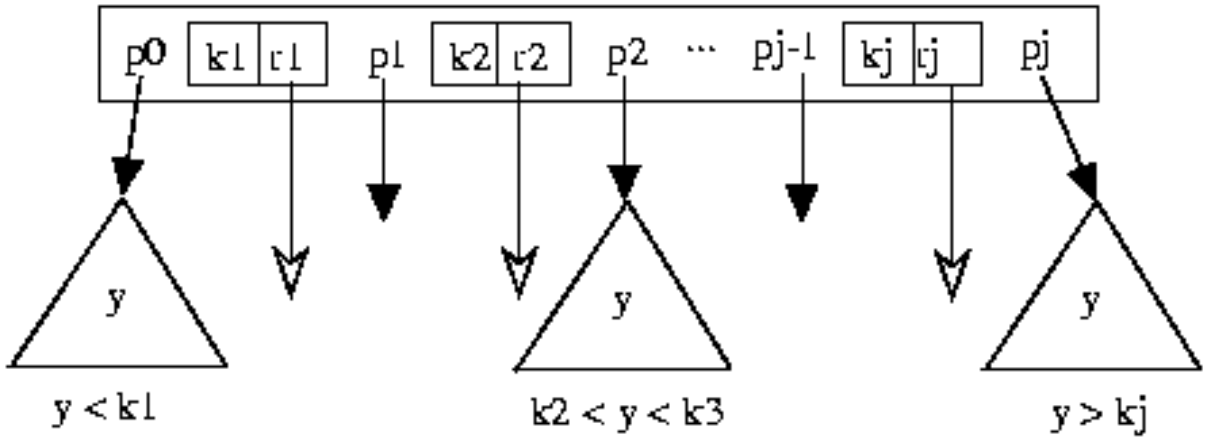
In each node

$$p_0(k_1, r_1)p_1\ (k_2, r_2)p_2\ \ldots\ p_{j-1}(k_j, r_j)p_j$$

- $k_1, \ldots, k_j$ are ordered key values, that is, $k_1 < k_2 < \ldots < k_j$
- There are $j + 1$ references to children nodes $p_0, \ldots, p_j$ and j references to the data file $r_1, \ldots, r_j$
- let $K(p_i)$ $(i = 1, \ldots, j)$ denote the set of key values stored in the subtree with root p_i, for each non-leaf node we have that:
  - $\forall y \in K(p_0),\ y < k_1$
  - $\forall y \in K(p_i),\ k_i < y < k_{i+1},\ i = 1, \ldots, j - 1$
  - $\forall y \in K(p_j),\ y > k_j$

B-trees

- Format of a node of a B-tree

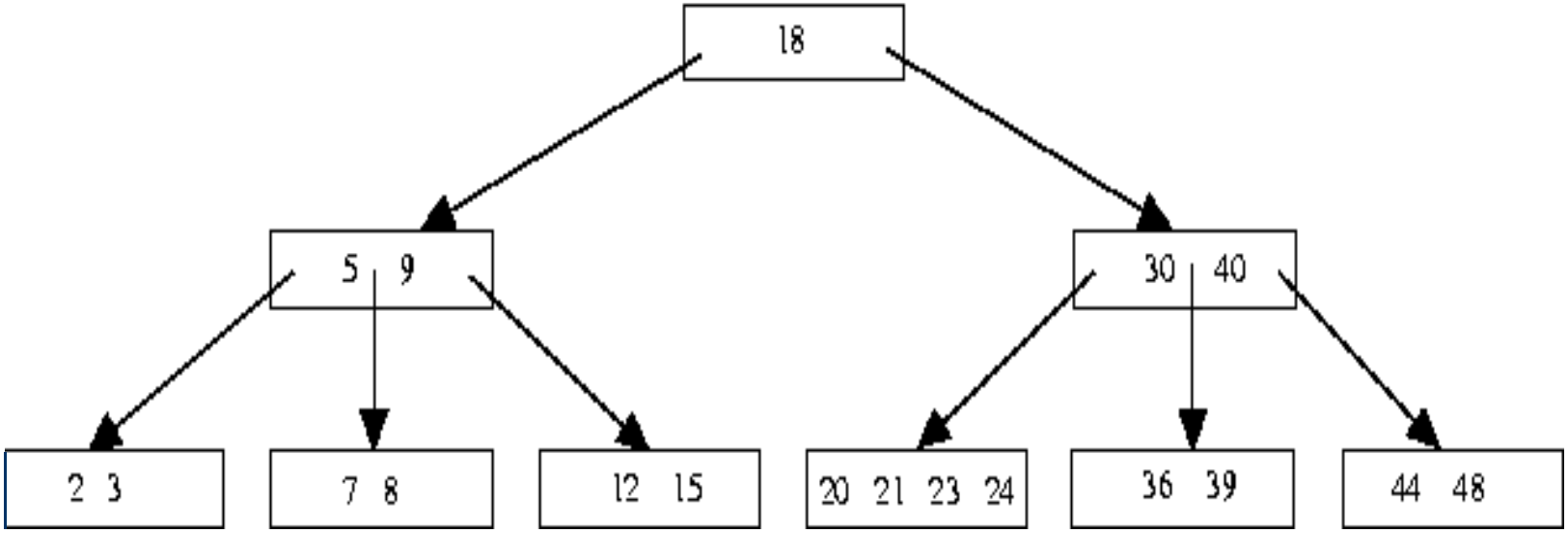


Pointer to a tree node

Pointer to the data file

B-trees

- Example of B-tree with m=5

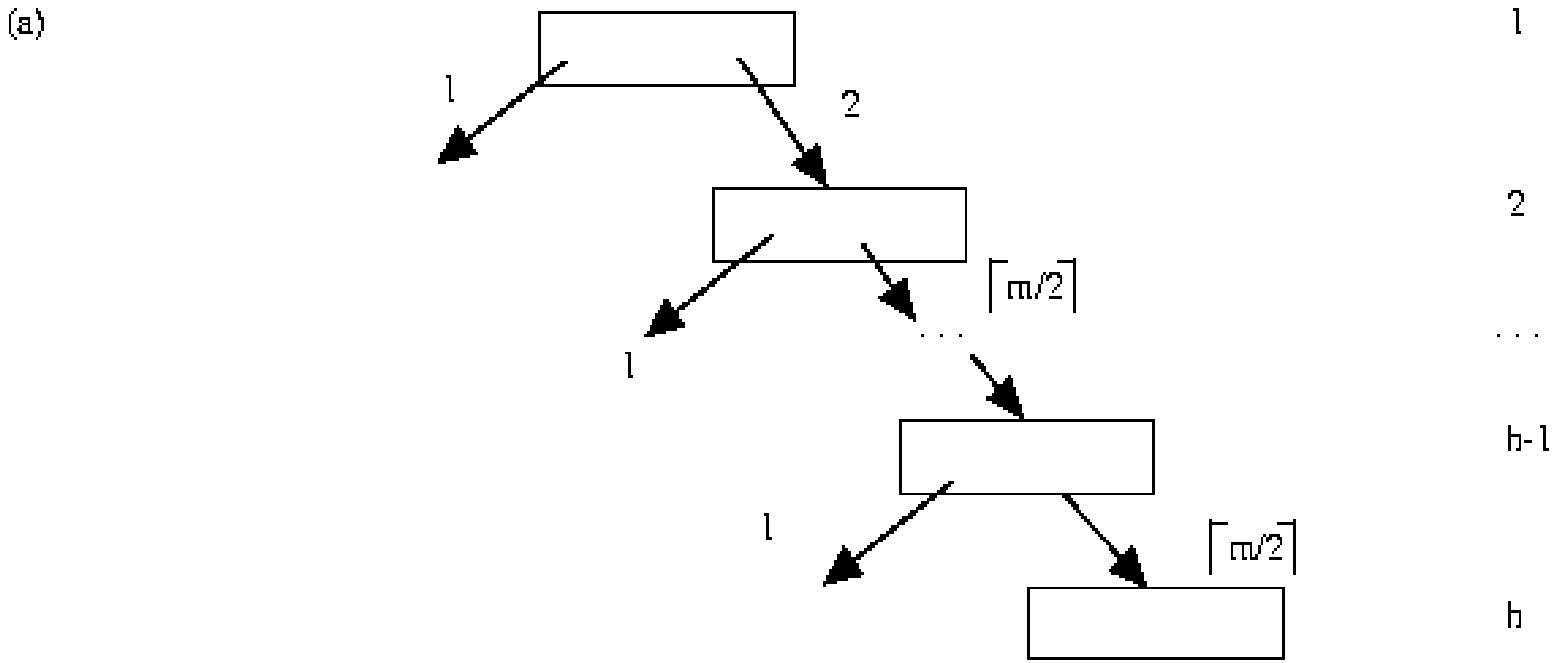


B-trees

- **Height** = number of nodes to be traversed in a path from the root to a leaf node
- the B-trees allow one to estimate with good approximation with average height of a tree based on the number of key values
  - Therefore it is easy to estimate the search costs
- $\mathbf{b_{min}}$ = minimum number of nodes that a B-tree of height *h* may store
- $\mathbf{b_{max}}$ = maximum number of nodes that a B-tree of height *h* may store

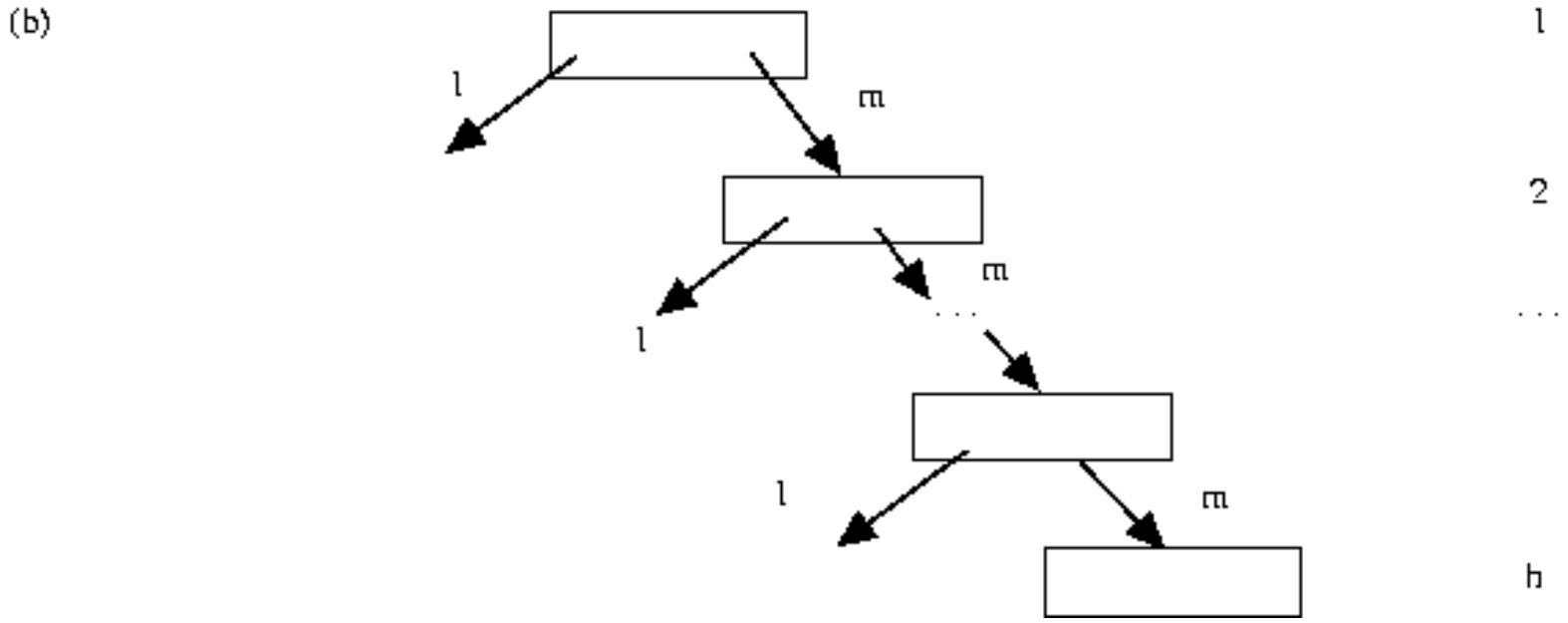
B-trees

- Minimum number of nodes in a tree with height *h*



B-trees

- Maximum number of nodes in a tree with height *h*



B-trees

- Minimum number of nodes

  $\mathbf{b_{min}} = 1+2+2\lceil m/2 \rceil+2\lceil m/2 \rceil^{2}+\ldots+2\lceil m/2 \rceil^{h-2}=$

$$= \quad 1+2\frac{\lceil m/2 \rceil^{h-1}-1}{\lceil m/2 \rceil-1}$$

we recall that $\sum_{i=0}^{k-1} x^i = \frac{x^k-1}{x-1}$

B-trees

- Maximum number of nodes

  $\mathbf{b_{max}} = 1+m+m^2+\ldots+m^{h-1} = \frac{m^h-1}{m-1}$

- $\mathbf{N_{min}}$ = minimum number of key values in a tree of height *h*
- $\mathbf{N_{max}}$ = maximum number of key values in a tree of height *h*

B-trees

- **Minimum number of key values**: the number of nodes is b_{min} and thus each node contains $\lceil m/2 \rceil - 1$ key values and the root contains a single key values

$$N_{\min} = 1 + (\lceil m/2 \rceil - 1)(b_{\min} - 1) =$$

$$= 1 + (\lceil m/2 \rceil - 1) 2 \frac{\lceil m/2 \rceil^{h-1} - 1}{\lceil m/2 \rceil - 1} = 2 \lceil m/2 \rceil^{h-1} - 1$$

B-trees

- **Maximum number of key values**: the number of nodes is b_{max} and thus each node, included the root, contains m - 1 keys

$$N_{\max} = (m-1)b_{\max} = (m-1)\frac{m^h - 1}{m-1} = m^h - 1$$

B-trees

- let N denote the number of key values in a B-tree, we have that:

$$2\lceil m/2 \rceil^{h-1} - 1 \leq N \leq m^h - 1$$

$$\log_m (N+1) \leq h \leq 1 + \log_{\lceil m/2 \rceil} \frac{N+1}{2}$$

B-trees

- height of a B-tree in terms of the number of nodes and the order; assume that key values are 10 bytes long and the pointers are 4 bytes long

Page size	m	$N = 1000$		$N = 10000$		$N = 100000$		$N = 1000000$	
		h_{min}	h_{max}	h_{min}	h_{max}	h_{min}	h_{max}	h_{min}	h_{max}
512	36	1,9	3,2	2,6	3,9	3,2	4,7	3,9	5,5
1024	73	1,6	2,7	2,1	3,4	2,7	4,0	3,2	4,6
2048	146	1,4	2,4	1,8	3,0	2,3	3,5	2,8	4,1
4096	292	1,2	2,2	1,6	2,7	2,0	3,2	2,4	3,6

B-trees: search operation

- Once the root is transferred in main memory, a search is performed on the key values in the root, until the presence or absence from the node of the key value is determined
  - If the key value is not found, the search is continued in the only subtree of the current node that may contain the key value
  - If the node is a leaf node, then the key value is not present in the tree
- The cost of the search is given by the number of nodes that are read, that is:

$$C_{search}^{\min} = 1$$

$$C_{search}^{\max} = h$$

B-trees: search operation

INPUT: y = the key value to be searched
root = the pointer to the root of the tree
OUTPUT: found = True is the key value is present in the tree
found = False otherwise
METHOD:
P(p) = the node pointed by p
k1, . . . , kj = the keys in P(p)
p0, . . . , pj = the pointers to the children in P(p)
p := root;
found := False;
While (p ≠ nil) and (not found) Do:
If y < k1 Then p := p0
Else If ∃i y = ki Then found := True
Else If ∃ i ki < y < ki+1 Then p := pi
Else p := pj
endif
endif
endif
endwhile

B-trees: insertion

- Main idea for insertion and deletion operations:
  - The updates always start from the leaf nodes and the tree grows or shrinks from the bottom of the tree
- Example: insertion
  - No new children nodes of the leaf nodes are created; rather an new leaf node, that the same level of the existing ones, is created and a key value (separator) is propagated to the next level (toward the root)
  - The nodes at the next level are not necessarily full and then can store information propagated from the leaf nodes
  - The propagation of the update effects until the root may result in an increase of the height of the tree

B-trees: insertion

- The insertion requires first of all a search operation to verify whether the key value is already present in the tree
- The insertion is always performed on a leaf node – two cases can arise:
  - If the leaf node is not full, the key value is inserted and the updated leaf node is re-written
  - If the leaf node is full, a **splitting** process starts that can propagate to next level and, in the worst case, can reach the root

B-trees: insertion - splitting

- Let P be a full node in which a key value must be inserted
- Ordered sequence of m entries that would be created with the new key value

  $p_0 k_1 p_1 k_2 p_2 \ldots k_g p_g k_{g+1} p_{g+1} k_{g+2} p_{g+2} \ldots k_m p_m$

  with g = $\lceil m / 2 \rceil - 1$
- The keys are partitioned as follows:
  - node P stores the elements:

    $p_0 k_1 p_1 k_2 p_2 \ldots k_g p_g$
  - a new node P' stores the elements:

    $p_{g+1} k_{g+2} p_{g+2} \ldots k_m p_m$

B-trees: insertion - splitting

- In the node *Q*, father of P, the following entry is inserted

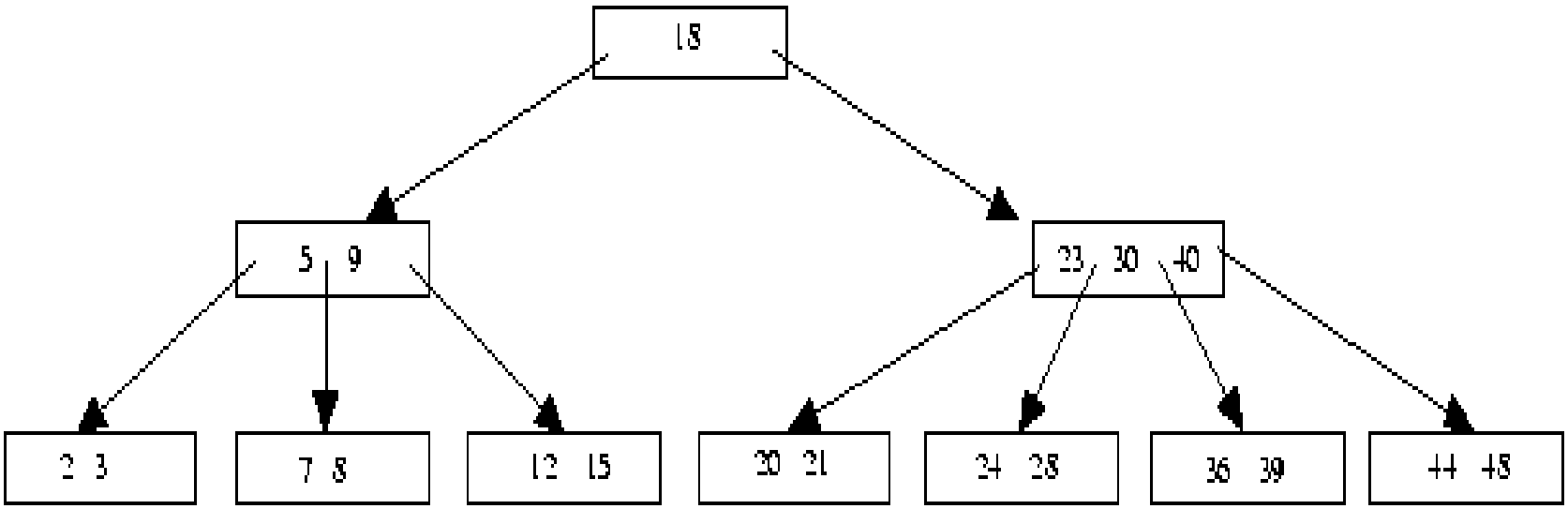
  $k_{g+1}p'$

  where p' is the pointer to the new node P'
- If also Q is full, the splitting process must be repeated
- If the root must be split, the new root becomes

  $pk_{g+1}p'$

  where p is the pointer to node P (the old root node)

B-trees: insertion - example



B-trees: insertion - costs

- Best case (no splitting):
  - *h* nodes are read and a leaf node is re-written, thus

$$C_{insertion}^{\min} = h + 1$$

- Worst case (the splitting propagates up to the root):
  - h nodes are read and 2h+1 are re-written, thus

$$C_{insertion}^{\max} = 3h + 1$$

B-trees: deletion

- Also in this case the process starts from a deletion from a lead node
- First of all a search is performed to determine whether the node storing the element to be deleted
  - If the element is not stored in a leaf node, then it is replaced with an element such that its key value is the smallest value in the subtree pointed by the pointer to the right of the element to be deleted

B-trees: deletion

- Deletion of an element from a leaf node – there are two cases:
  - If the leaf node is not in underflow (that is, it has at least $\lceil m/2 \rceil - 1$ elements after the deletion), the key is deleted and the updated leaf node is re-written
  - If the leaf node is in underflow, a **concatenation** process or **redistribution** process is started

B-trees: deletion *concatenation*

- The concatenation of two adjacent nodes P and P' is possible if the two nodes contain on the overall less than m - 1 keys
- A node with less than $\lceil m/2 \rceil - 1$ elements, that is, in **underflow**, is combined with an adjacent node with at most $\lfloor m/2 \rfloor$ keys
- the concatenation process is the opposite of the splitting process

B-trees: deletion *concatenation*

- Initial situation:
  - node P in underflow with elements:

    $p_0 k_1 p_1 k_2 p_2 \ldots k_e p_e$ ($e = \lceil m/2 \rceil - 2$)
  - node P' adjacent to the right of P with elements:

    $p'_0 k_{e+1} p_{e+1} \ldots$
  - node Q, father of P and P', with elements

    $\ldots k_{t-1} p_{t-1} k_t p_t k_{t+1} p_{t+1} \ldots$

    where p_{t-1} is a pointer to P and p_t is a pointer to P'

B-trees: deletion *concatenation*

- the concatenation of two adjacent nodes results in the following situation:
  - node P with elements:

    $p_0 k_1 p_1 k_2 p_2 \ldots k_e p_e k_t p'_0 k_{e+1} p_{e+1} \ldots$
  - node Q with elements:

    $\ldots k_{t-1} p_{t-1} k_{t+1} p_{t+1} \ldots$

    where p_{t-1} is a pointer to P

B-trees: deletion *concatenation*

- The deletion of the key value k_t from the father node may in turn trigger a concatenation (or a redistribution)
- the concatenation may propagate to the root, resulting in the deletion of the root node and thus in a decrease of the tree height

B-trees: deletion *redistribution*

- If two adjacent nodes cannot be concatenated, then the elements of the two nodes can be redistributed
- The redistribution operation involves also the father node because one of its elements must be modified; however the number of its elements is not modified and therefore there is no propagation to the next level

B-trees: deletion *redistribution*

- Initial situation:
  - node P' in underflow with elements:

    $p'_0 k'_1 p'_1 k'_2 p'_2 \ldots k'_j p'_j$ ($j = \lceil m/2 \rceil - 2$)
  - node P adjacent to P' on the right with elements:

    $p_0 k_1 p_1 k_2 p_2 \ldots k_e p_e$
  - node Q, father of P and P', with elements :

    $\ldots k_{t-1} p_{t-1} k_t p_t k_{t+1} p_{t+1} \ldots$

    where p_{t-1} is a pointer to P' and p_t a pointer to P

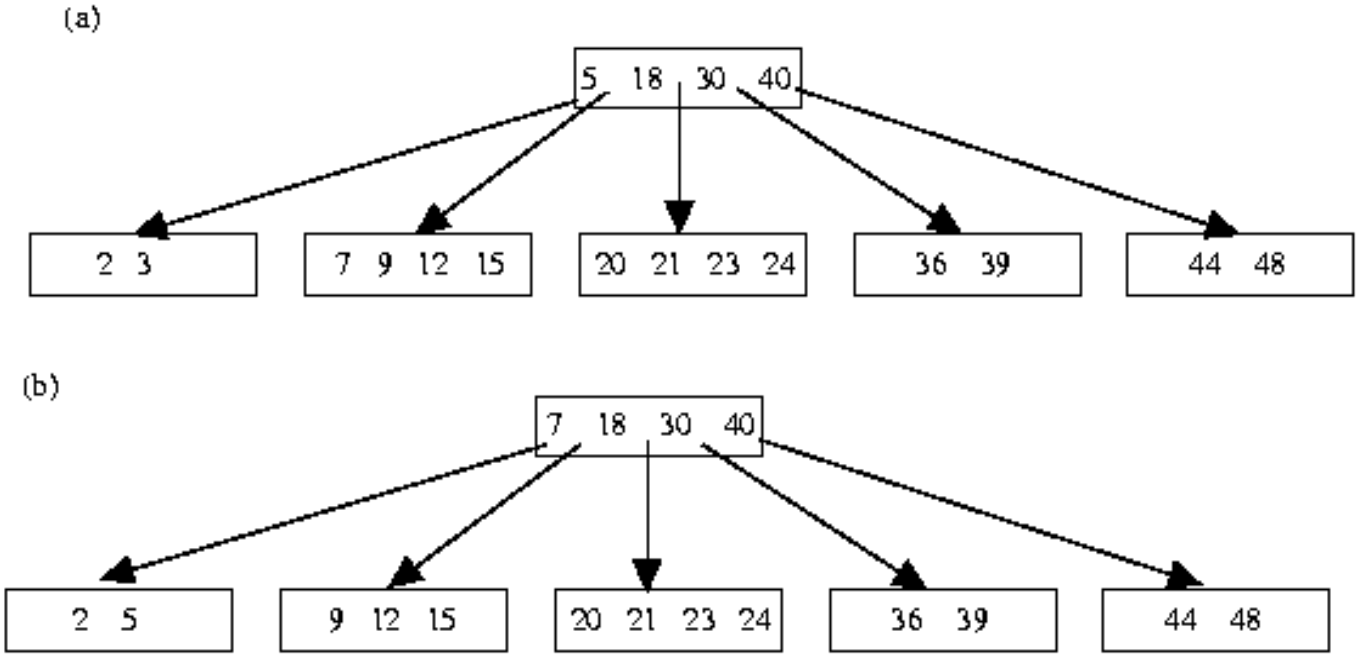
B-trees: deletion *redistribution*

- To redistribute the elements in the two nodes:
  - Consider the list of key values

    $k'_1\ k'_2 \ldots k'_j\ k_t\ k_1\ k_2 \ldots k_e$
  - The first $\lfloor (e+j)/2 \rfloor$ key values are left in P'
  - In the father node the key k_t is replaced by the key value of position $\lfloor (e+j)/2 \rfloor + 1$
  - The remaining $\lceil (e+j)/2 \rceil$ key values are inserted in P

B-trees: deletion

- (a) Concatenation [deletion of 8]
- (b) Redistribution [deletion of 3]



B-trees: deletion - costs

- **Best case** (the deletion is performed in a leaf node and no concatenation nor redistribution are needed):
  - h nodes are read and a leaf node is re-written, thus

$$C_{deletion}^{\min} = h + 1$$

- **Worst case** (all nodes along the search path must be concatenated, with the exception of the first two; therefore for the child of the root a redistribution operation must be performed and the root has to be modified):
  - 2h - 1 nodes are read and h+1 nodes are re-written, thus

$$C_{deletion}^{\max} = 3h$$

B-trees: deletion
details of the worst case

- R: C(search) + C(excluding the root) + C(root)
- W: C(concationation) + C(redistribution) + C(root)
- hp1: each read operation requires a I/O operation
  - R: h + 3(h-1) + 1 = 4h -2
  - W: (h - 2) + 2 + 1 = h+1
- hp2: if I assume to concatenate always to right or left:
  - R: h + 2(h-1) + 1 = 3h - 1
  - W: (h-2) + 2 + 1 = h+1
- hp3: the accessed nodes remain in main memory + hp2
  - R: h + (h-1) + 0 = 2h -1
  - W: (h-2) + 2 + 1 = h+1

B-tree: redistribution in data insertion

- The structure of a B-tree depends from the order according to which the data are inserted
- If the key values were inserted according to their relative order, we would have a B-tree with all leaf nodes filled at a half, possibly except for the last node
- To improve storage utilization:
  - The redistribution during the insertion is used: instead of splitting a full node, a redistribution is executed with an adjacent node, until such node is full
  - Such an approach generates trees with nodes storing more keys, but it increases the insertion costs

B+-trees

- A B-tree is very efficient with respect to search and modification operations that involve a single record
  - For example, let m = 100 and N = 1000000; in such case the search of a key requires at most 4 disk accesses
- A B-tree however is not particularly suited for sequential operations nor for range searches
  - The retrieval of the next key value may requires accessing a large number of nodes
- To address such problem a variation to the B-tree structure has been proposed, known as B+-tree

B+-trees

- **Main idea**: in a B-tree, the key values have two functions:
  - **separators**: to determine the path to follow during the search
  - **Key values**: to allow accessing the information associated with them (that is, the pointers to the data)
- In a B+-tree such functions are kept distinct:
  - The leaf nodes contain all the key values (and the associated information)
  - The internal nodes (organized as a B-tree) store some separators which have the only function of determining the path to follow when searching for a key value

B+-trees

- In addition the leaf nodes are linked in a list, in order to efficiently support range searches or sequential searches (there is also a pointer to the first element of such list to support fast accesses to the minimum key value)

- partial duplication of the keys
  - The index entries (keys and data references) are only stored in the leaf nodes
  - A search for a given key value must always determine a leaf node

B+-trees

- The subtree on the left side of a separator contains key values that are lower than the separator; the subtree on the right side of a separator contains key values which are greater or equal than the separator

- In the case of alphanumeric keys, one can reduce the space requirements by using separators that have reduced lengths

B+-trees

- A B+-tree of order m ($m \geq 3$) is a balanced tree that verifies the following properties:
  - each node contains at most m - 1 elements
  - each node, except the root, contains at least $\lceil m/2 \rceil - 1$ elements; the root may contain a single element
  - Each non leaf node containing j elements has j +1 children

B+-trees

- Each leaf node has the following structure

$$(k_1, r_1)(k_2, r_2) \ldots (k_j, r_j)$$

where:

  - j is the number of the elements of the node
  - $k_1, \ldots, k_j$ are ordered key values, that is, $k_1 < k_2 < \ldots < k_j$
  - $r_1, \ldots, r_j$ are j riferences to the data file

- Each leaf node has a pointer to the next leaf node and to the previous leaf node

B+-trees

- Each non leaf node has the following structure:

$$p_0 k_1 p_1 k_2 p_2 \ldots p_{j-1} k_j p_j$$

where:

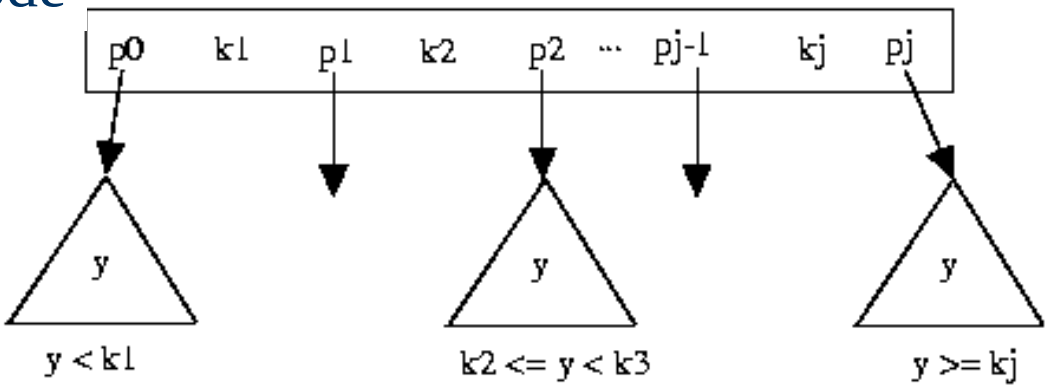
- j is the number of the elements of the node
- $k_1, \ldots, k_j$ are ordered key values, that is, $k_1 < k_2 < \ldots < k_j$
- $p_0, \ldots, p_j$ are j+1 references to the children nodes

B+-trees

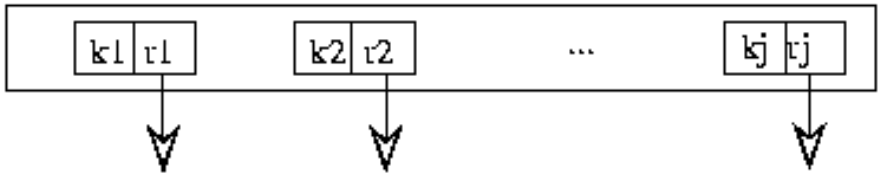
- For each non leaf node, let $K(p_i)$ (i = 0, ... , j) be the set of key values stored in the subtree with root p_i, we have:
  - $\forall\ y \in K(p_0)$, $y < k1$
  - $\forall\ y \in K(p_i)$, $k_i \leq y < k_{i+1}$, i = 1, . . . , j - 1
  - $\forall\ y \in K(p_j)$, $y \geq k_j$
  - each k_i, for i = 1, . . . , j, is the minimum element of $K(p_i)$

B+-trees

Non leaf node



Leaf node

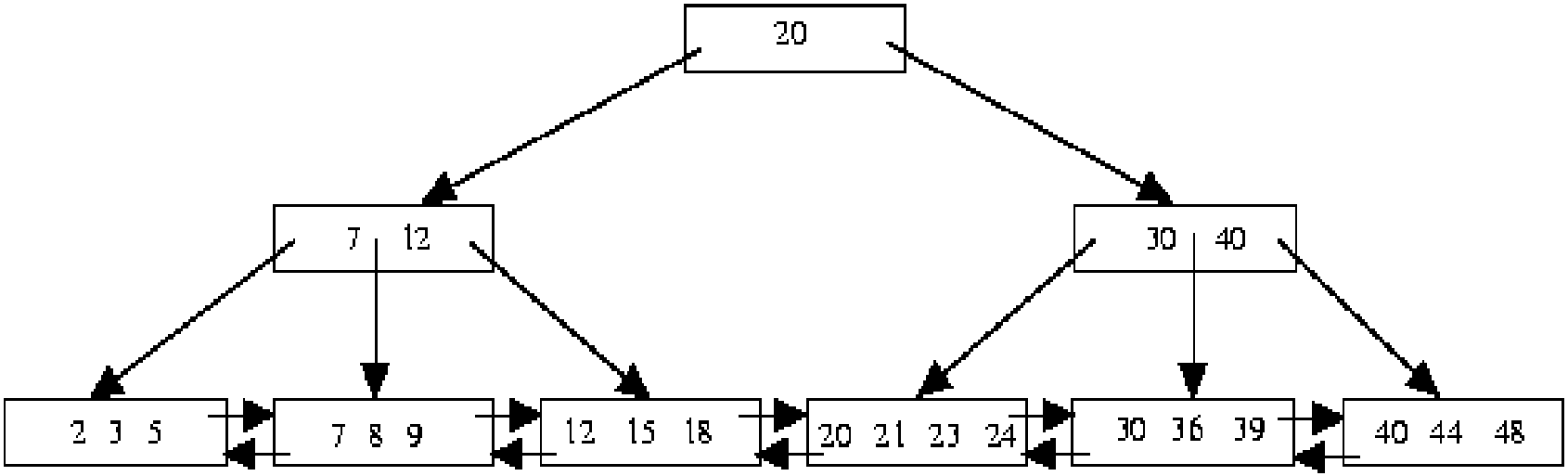


Pointer to a node of the tree

Pointer to the data file

B+-trees

- Example of B+-tree with order 5



B+-trees vs B-trees

- assumption: nodes have the same size in both organization
- The search of a single key value is in general more expensive in a B+-tree (we have always to reach a leaf node to fetch the pointer to the data file)
- for operations requiring an ordering of the retrieved records according to the search key values or for range queries, the B+- tree is to be preferred (the links among the leaf nodes eliminates the need of accessing the intermediate nodes)
- the B-tree requires less storage (the key values are stored only once)

B- and B+-trees: terminology

- **Order of a B-tree**
  - For us: maximum number of children of a node
  - variation: minimum number of keys that a node may store (used in the original definition)
- Some textbooks call B-trees what we have called B+-trees
- Other textbooks call B*-tree a variation of the B-tree where the storage utilization for nodes must be at least 66% instead of 50%

Hash organizations

- The hash organizations are mainly used primary data organizations
- The use of indexes has the disadvantage that a scan of an additional data structure has to be executed to retrieve the data; this is because the association

  (key, address)

  is explicitly maintained
- A hash organization use a hash function H that transforms each key value into an address
- To perform a search, given a key value k, one has simply to compute H(k)