

Index Structures

Tecnologie delle Basi di Dati M

Cons of file organizations

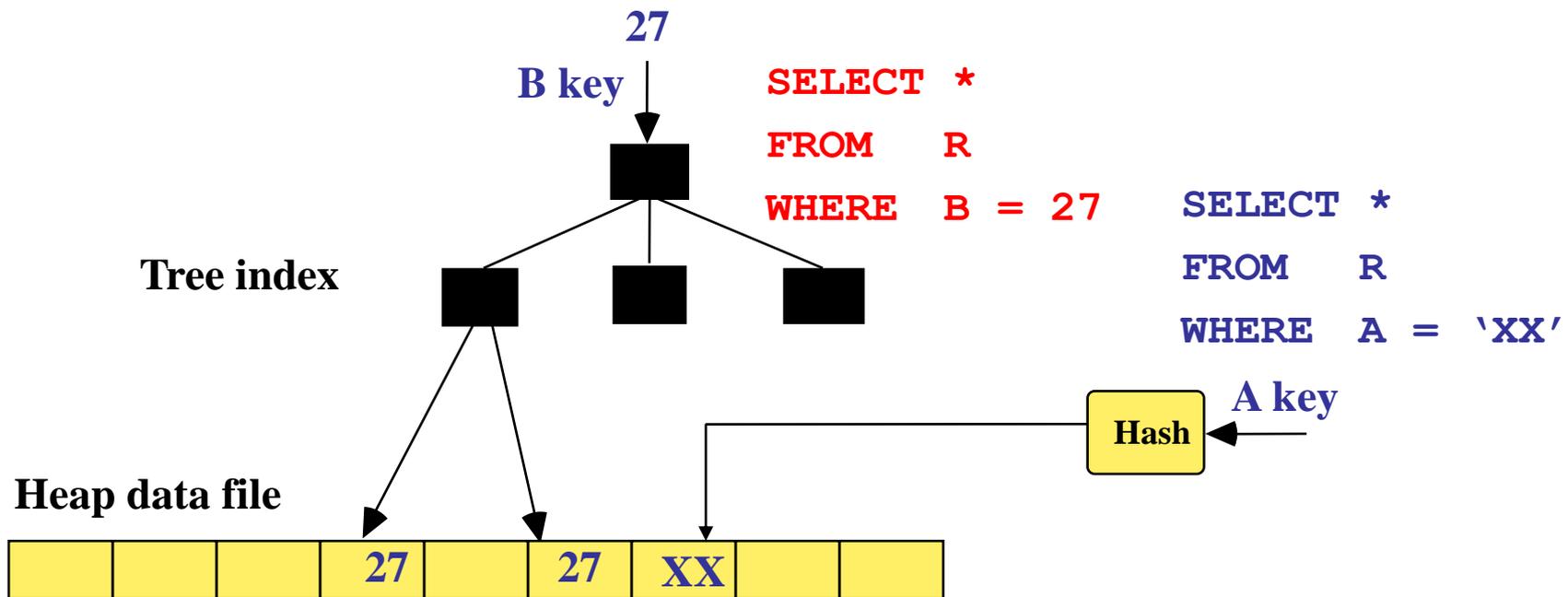
- Heap and sequential organizations have clear pros:
 - For heap, inserting is very quick
 - For sequential, all operations are quick enough
- Both have their cons, however:
 - For heap, searching is very slow
 - For sequential, searching is efficient (more or less) only if it is performed on the sort attribute (moreover, periodic re-organizations are a must)

Index structures

- They are **auxiliary** structures designed to speed up the search for records satisfying a given boolean predicate
 - Data are stored within heap or sequential files
 - Every index speeds up the search for a different predicate (**search key**)
 - They are (in practice) a collection of pairs $\langle \text{key}, \text{RID} \rangle$ (**entry**)
 - The goal of the index is to speed up the retrieval of those entries having a key value that satisfies the predicate
 - Pro: entries are (much) smaller than records

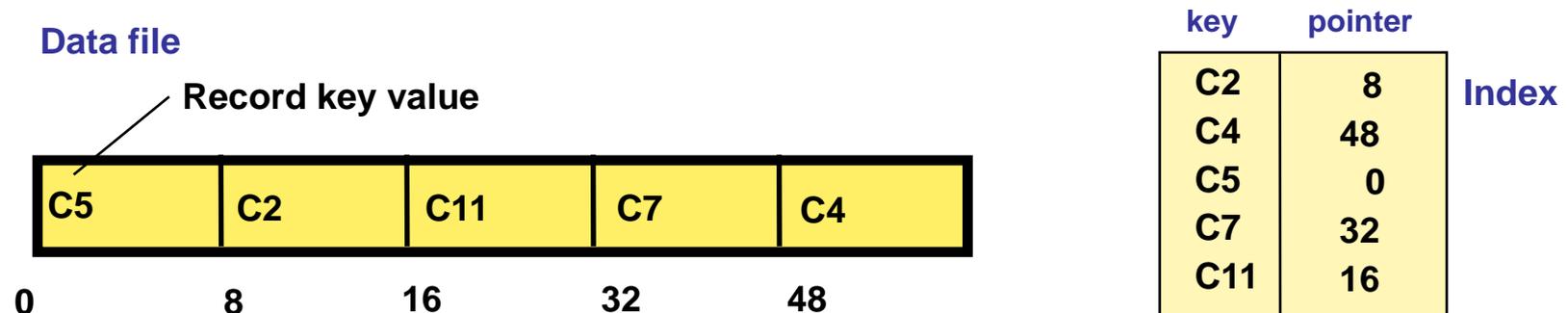
Access paths

- Building indices on a relation provides alternative ways (**access paths**) to quickly locate data of interest
- We commonly use the term (search) **key** (value) to indicate the value of an attribute used to select records (e.g., B is a key)



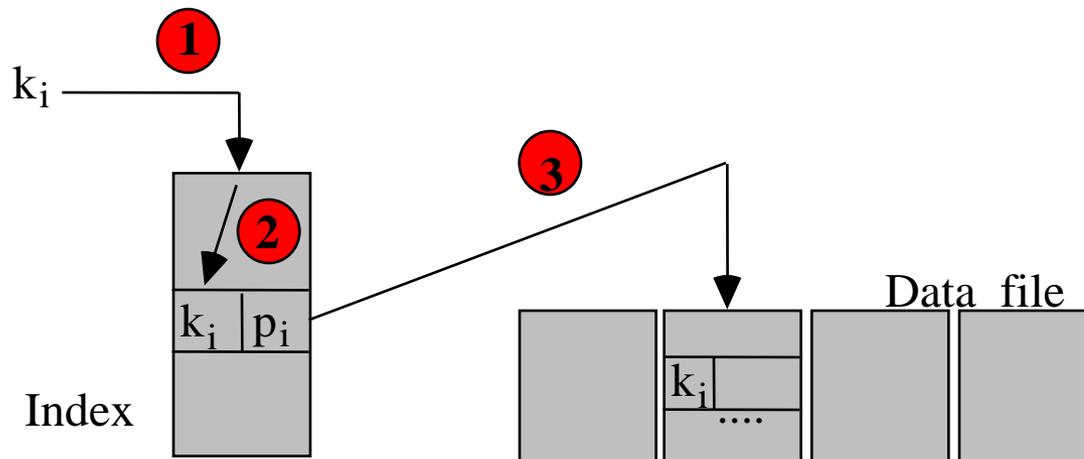
Indices: basic principle

- From the logical point of view, an index can be seen as a **set of pairs (entries) (k_i, p_i)** , where:
 - k_i is a **key value** of the attribute on which the index is built
 - p_i is a **pointer to the record(s)** with search key value k_i
In a DBMS, it is a **RID** or, at least, a **PID**
- The advantage of using an index comes from the fact that the key is just a (small) part of the information in a record
- Therefore, **the index is typically smaller than the data file**
- Indices differ in the way they organize the set of pairs (k_i, p_i)



Index access: general schema

- Let us consider an index built on a primary key used to search for the record having key k_i
- The steps to be performed are:
 1. Accessing the index
 2. Searching for the pair (k_i, p_i)
 3. Accessing the data page pointed by p_i



Index types

- Different types of indices exist

The first distinction is between

- **Sorted indices**: key values k_i are kept sorted, so that they can be accessed in a quicker way
- **Hash indices**: a hash function is used to find out the location of entries with key value k_i
 - Such indices are not well suited for range searches

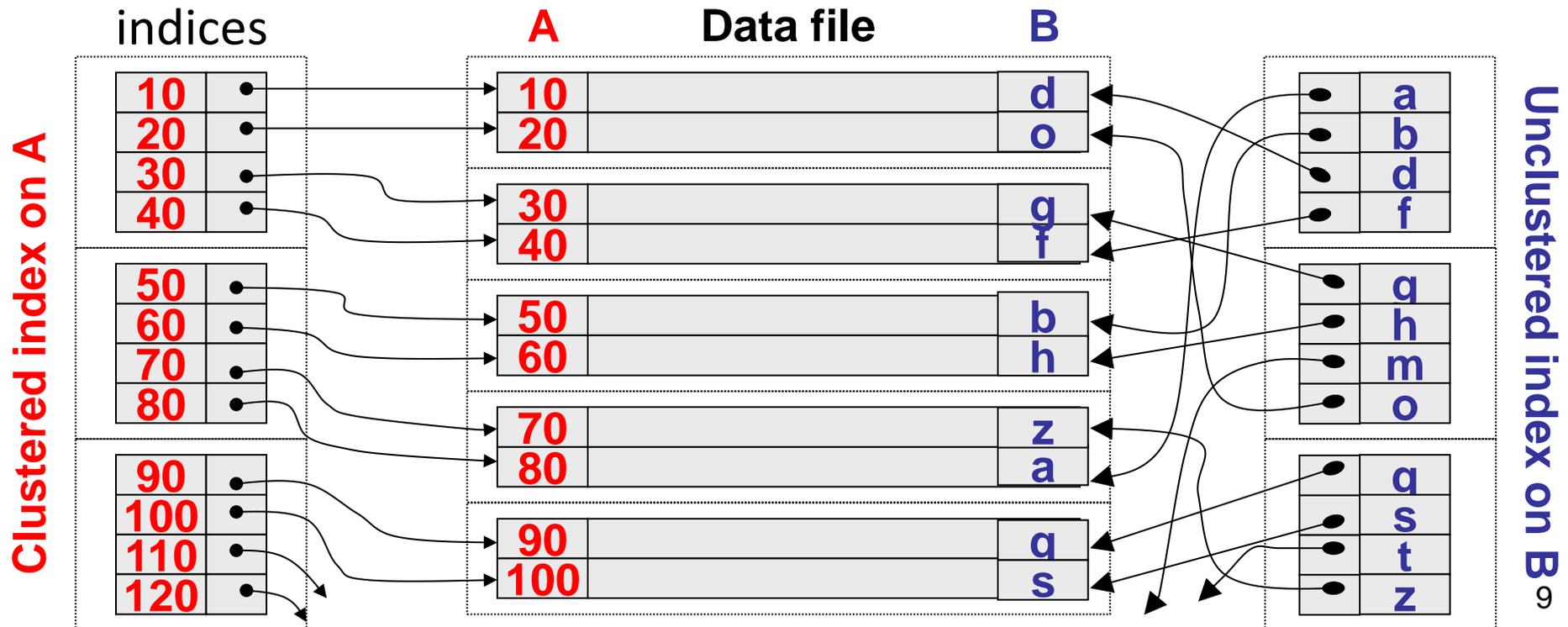
Indices: terminology

- Clustered vs. unclustered
- Primary vs. secondary
- Single-level vs. multi-level
- Dense vs. sparse

- **Such terminology is not standard,**
for example, someone calls primary what we call clustered
and secondary what we call unclustered

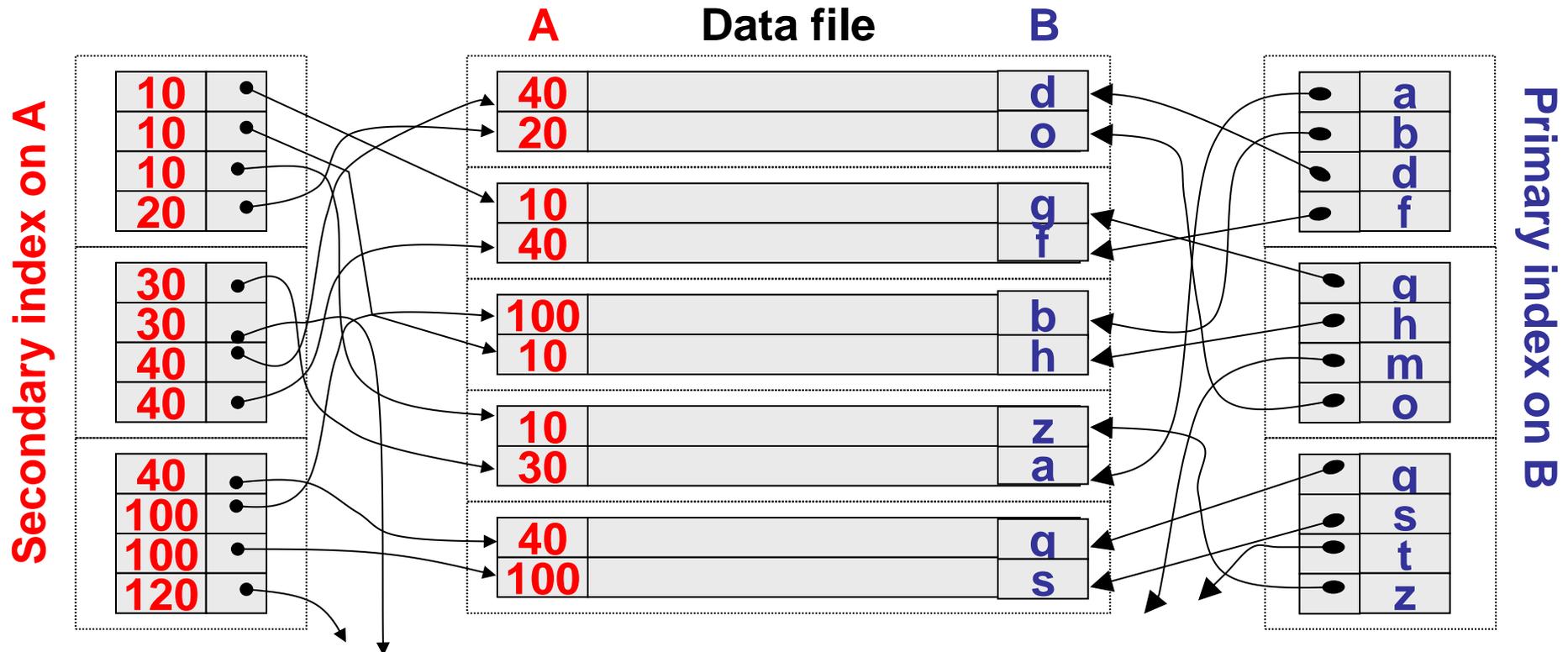
Clustered vs. unclustered indices

- We say that the index is **clustered** if it is built on the same attribute used to sort records in the data file, otherwise we say that is **unclustered**
- Clearly, we can build at most one clustered index for each relation, while we can build an arbitrary number of unclustered indices



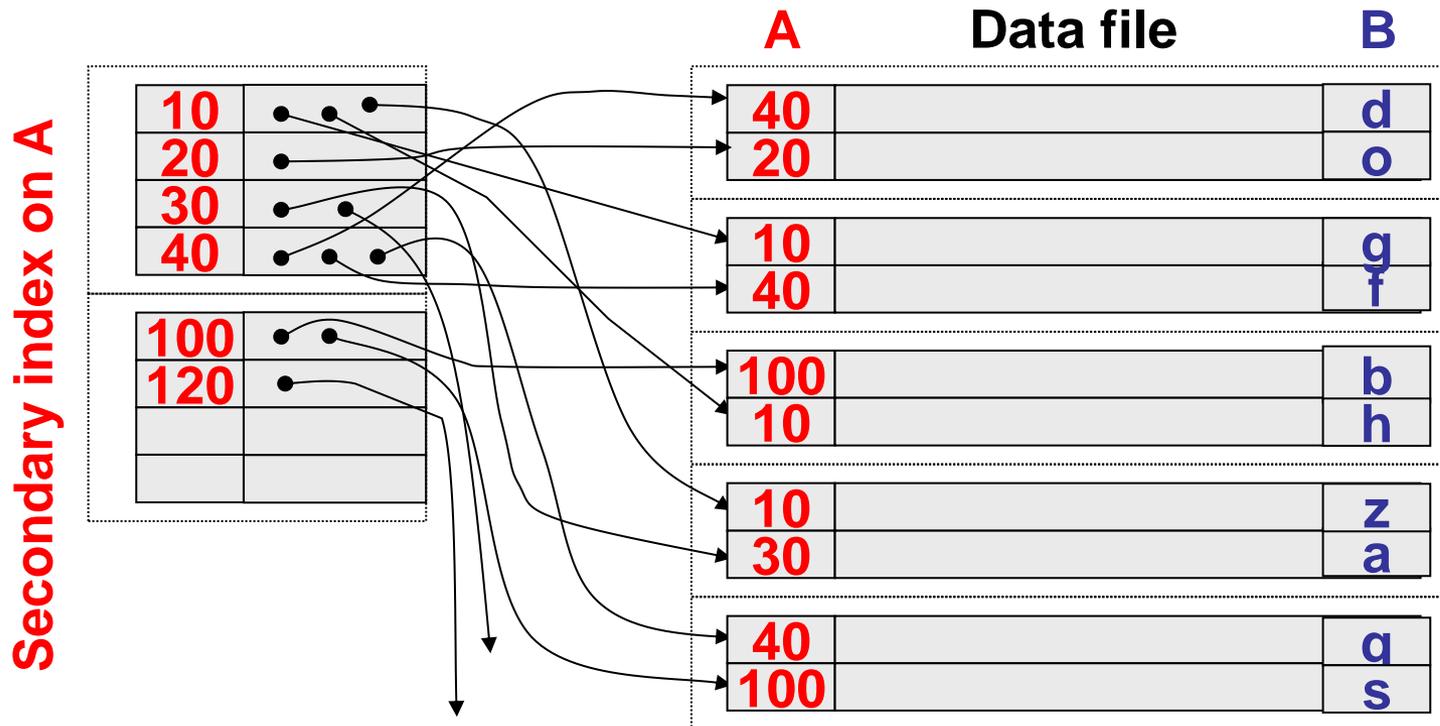
Primary vs. secondary indices

- We say that the index is **primary** if it is built on a **unique attribute** (candidate key), otherwise it is called **secondary**



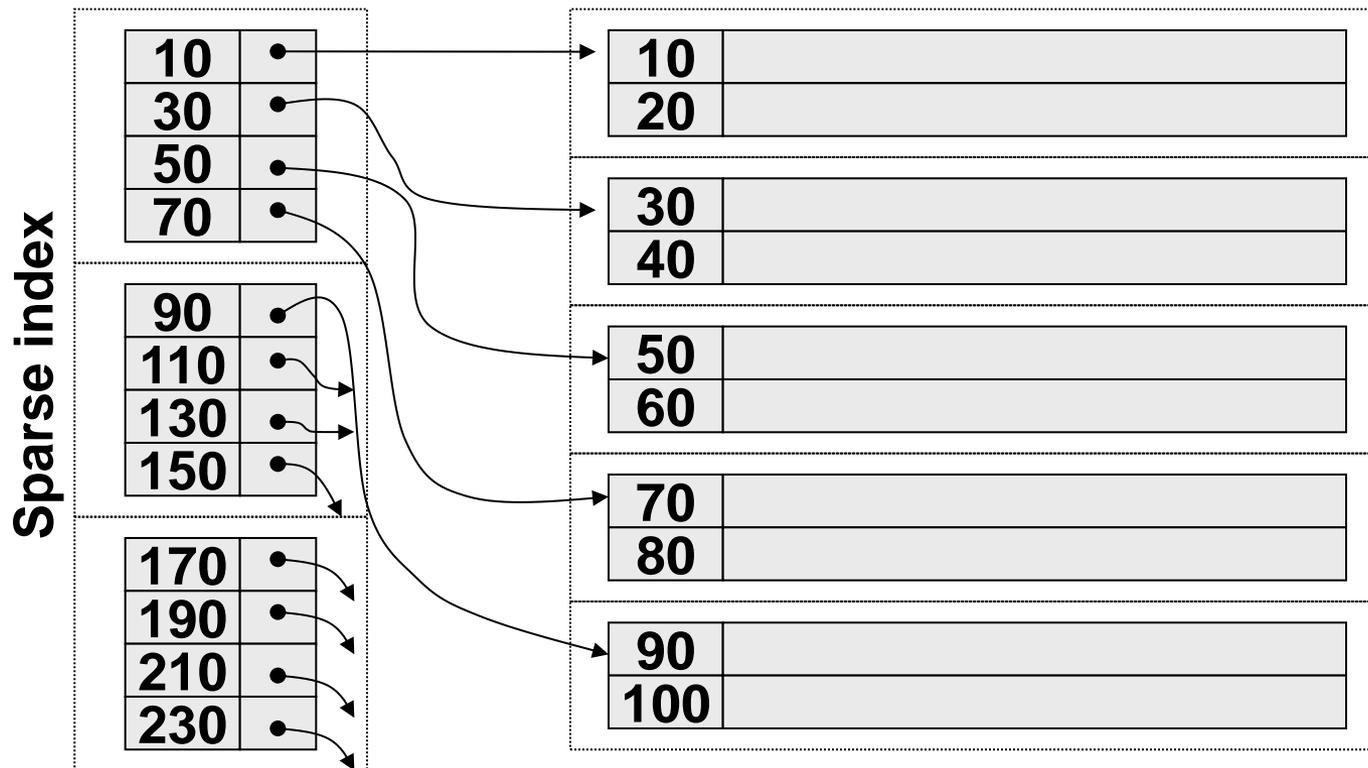
Secondary indices with pointers lists

- To avoid repeating key values, the most commonly used solution for secondary indices consists in **grouping all entries sharing the same key value in a pointers list**



Dense vs. sparse indices

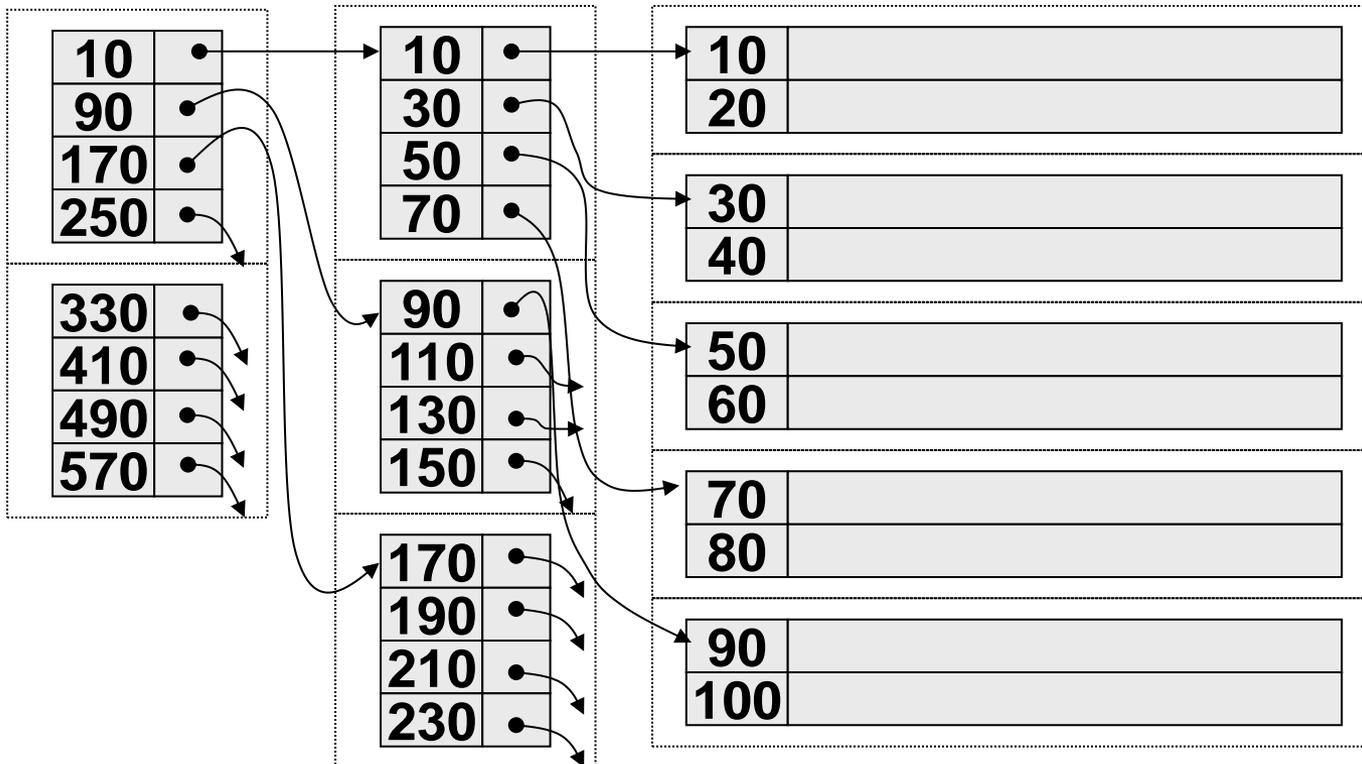
- In **dense** indices the number of pointers equals the number of records in the data file
- In **sparse** indices this is lower (usually, one for each data page)
 - This technique is **applicable only to clustered indices** (why?)



Single-level vs. multi-level indices

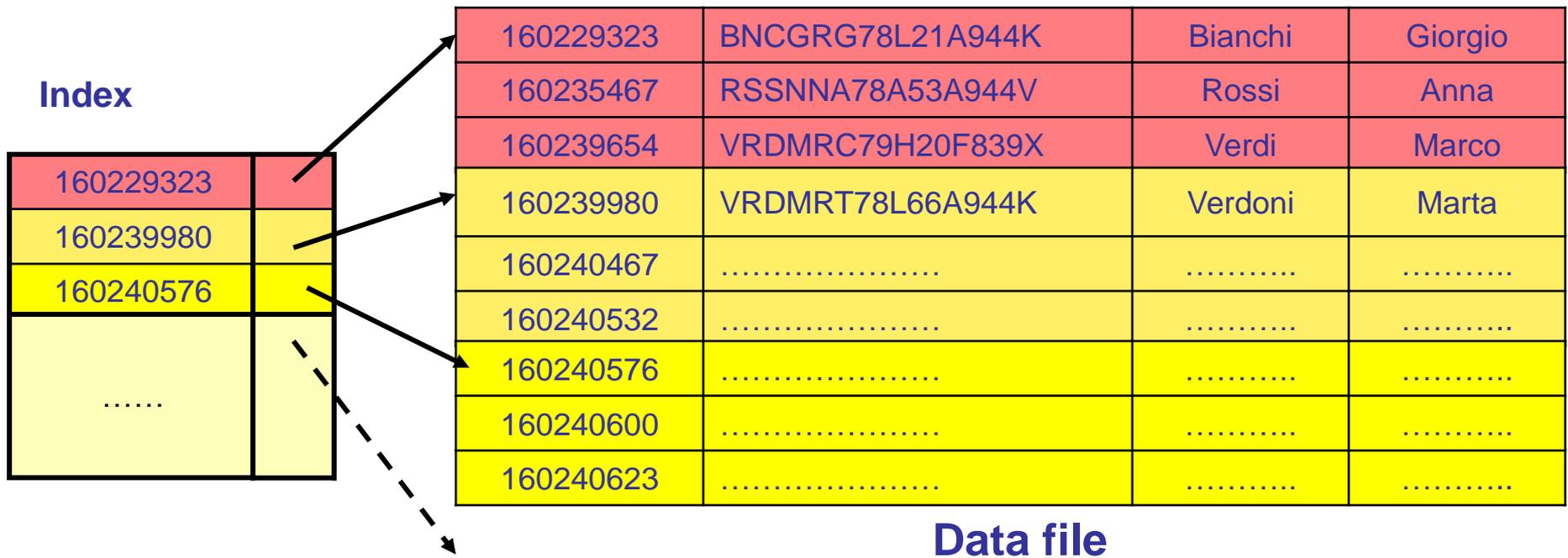
- All previous examples included **single-level** (or “flat”) indices
- However, we can “**index the index**” using a (sparse!) index, and so on (recursively), creating a **multi-level structure (tree)**

2-levels index



Example (1)

primary clustered sparse single-level index



Example (2)

secondary unclustered dense single-level index

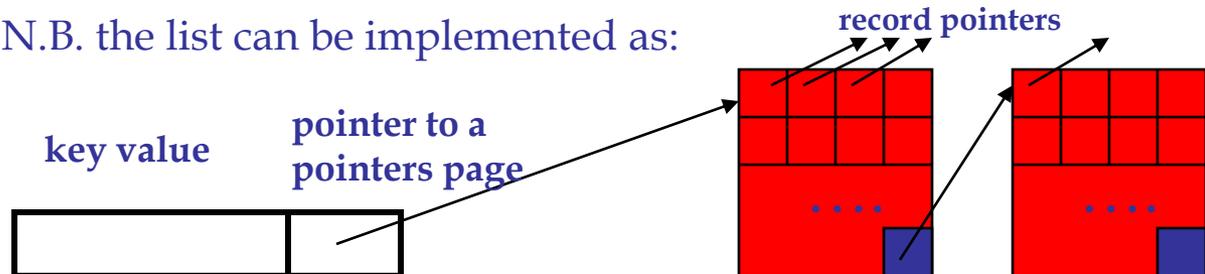
data pages

RID list

Alessandri
Bianchi
Bianchini
.....
.....
Rossi
.....
Verdi
Verdone
.....

160229323	BNCGRG78L21A944K	Bianchi	Giorgio
160235467	RSSNNA78A53A944V	Rossi	Anna
160239654	VRDMRC79H20F839X	Verdi	Marco
160239980	VRDMRT78L66A944K	Verdone	Marta
160240467	Alessandri	Maria
160240532	Bianchini	Carlo
.....			
160240576	Rossi	Demetrio
160240600	Bianchi	Arrigo
160240623	Verdi	Remo

N.B. the list can be implemented as:



Index specification in SQL

- In SQL, we can define indices by way of the CREATE INDEX statement (this is not standard!)
- In DB2:

```
CREATE INDEX VotoIDX          -- secondary unclustered index
ON Esami (Voto DESC)         -- ASC is the default
```

```
CREATE UNIQUE INDEX MatrIDX   -- primary unclustered index
ON Studenti (Matricola)
```

```
CREATE INDEX VotoIDX          -- clustered index
ON Esami (Voto DESC) CLUSTER
```

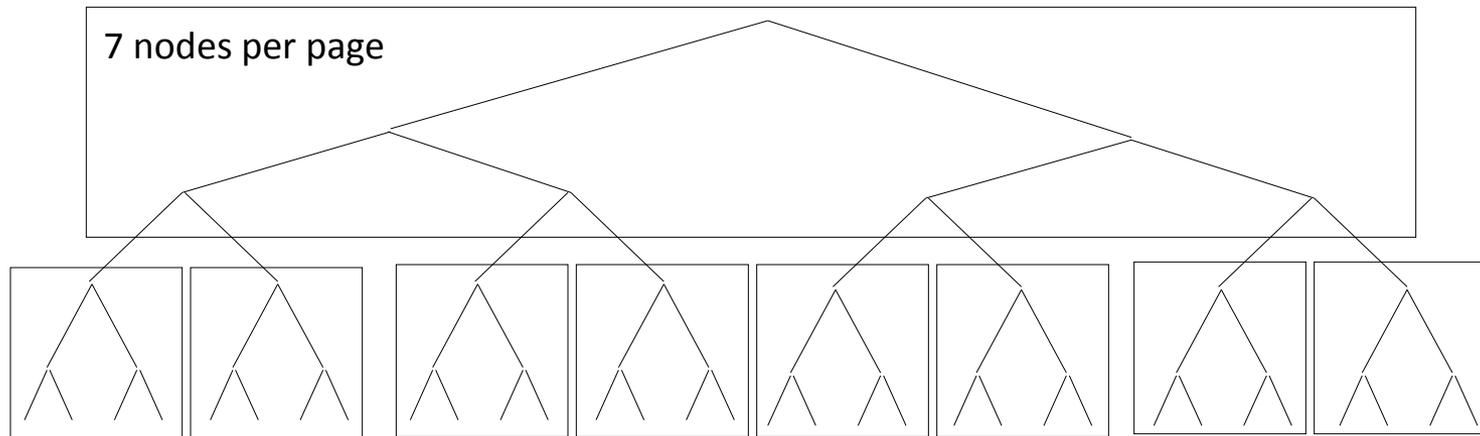
```
CREATE INDEX Anagrafica      -- multi-attribute index
ON Persone (Cognome, Nome)
```

Multi-level indices

- For efficiency reasons, usually indices are multi-level (**trees**)
- Can we “adapt” to secondary memory search trees meant for main memory?
- Requisites:
 - Balancing (worst case performance)
 - Pagination (we are using disk)
 - Minimal page utilization (size)
 - Updating efficiency

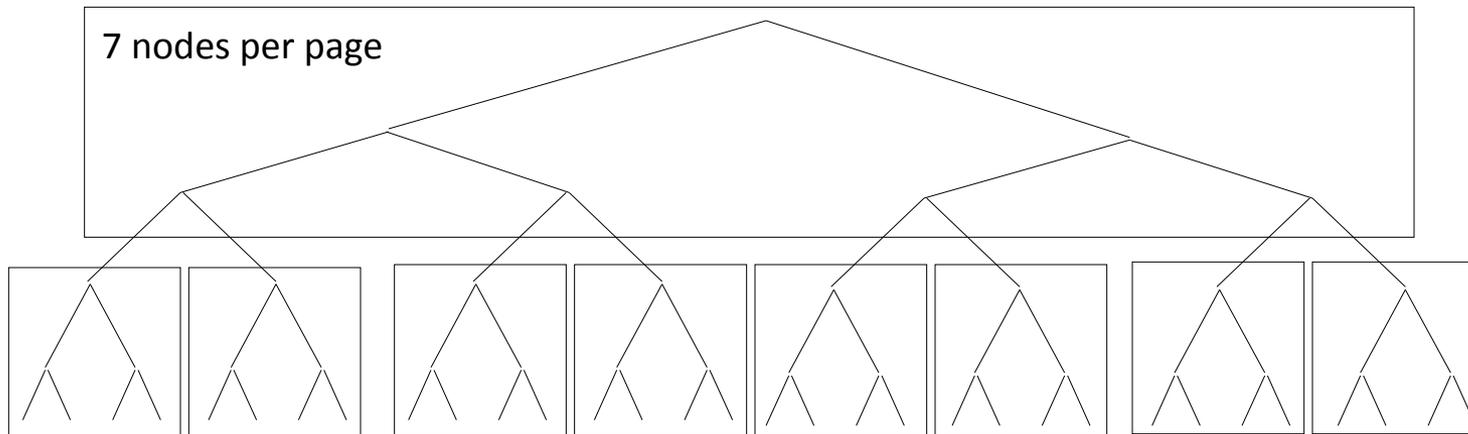
Paginating trees (i)

- Trees for main memory (AVL trees, red-black trees) are typically binary trees
- Very large number of nodes
 - Visiting the tree requires several accesses
- We should pack several nodes in a single page



Paginating trees(ii)

- Cons:
 - Complication of the balancing algorithm (inefficient during updates)
 - No guarantee on the minimum page utilization
- We should find a specific solution!



B-tree (R. Bayer & E. McCreight, 1972)

- Tree-shaped data structure that keeps **sorted data** and **balances nodes** allowing insert, deletion, and search operations in **logarithmically amortized time**
- Etymology (D. Comer):
 - Balanced?
 - Broad?
 - Bushy?
 - Boeing?
 - Bayer?
 - *“What really lives to say is: the more you think about what the B in B-trees means, the better you understand B-trees.”* (E. McCreight, 2013)

B-tree: terminology

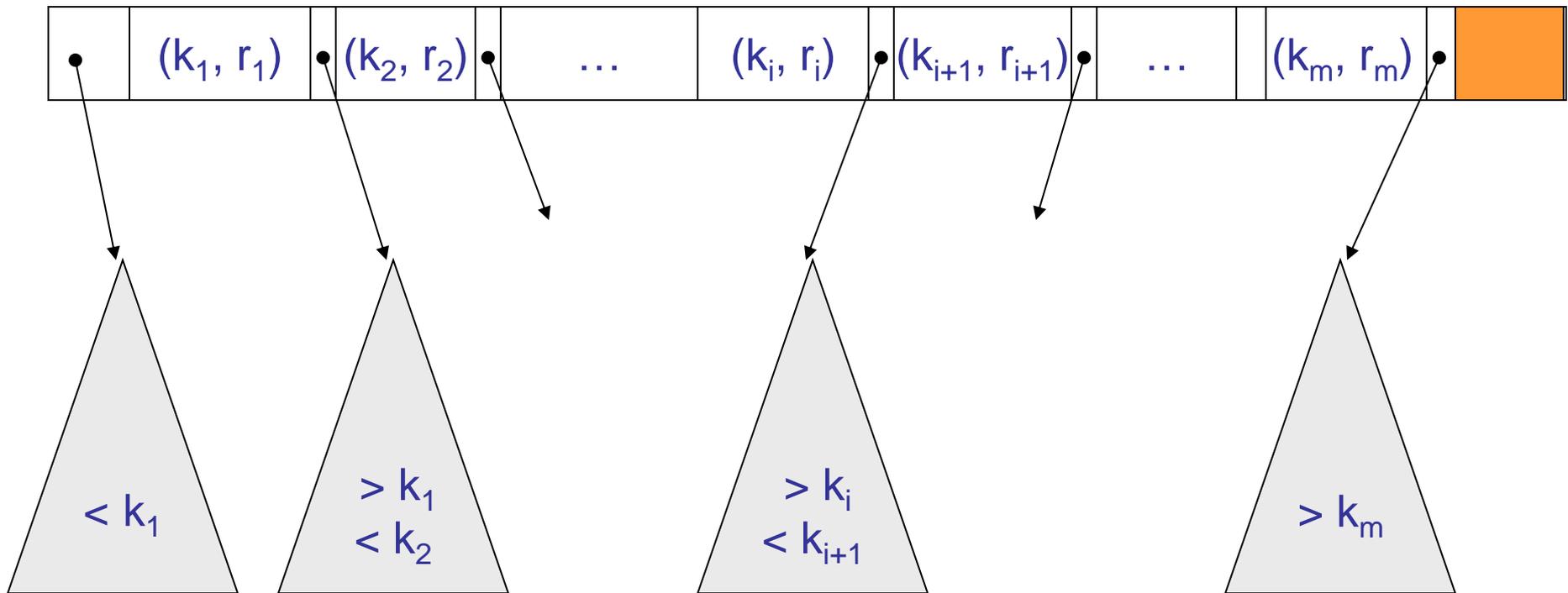
- A B-tree is a **multi-way perfectly balanced tree where nodes correspond to data pages**
- Every node contains a number **m** of entries that can vary in the range **d – 2d** (**d** = **order** of the tree)
- The number of children nodes of each node equals **m+1** (thus it can vary in the range **d+1 – 2d+1**)
 - High fan-out, thus limited height
 - (Very) low search cost
 - Limited size
- The root node is allowed to violate the minimum utilization constraint, thus having a single entry

B-tree: features

- Perfect balancing means that **every path from the root to any leaf has the same length** (height of the tree)
- The search algorithm **follows a single path from the root to a single leaf** (cost \leq height)
- Perfect balancing is guaranteed by algorithms for inserting and deleting records
 - In order to guarantee balancing, node-branching operations are performed towards the root (not towards the leaves)
- Minimum occupation of 50% (except for the root node)
- Typical occupation $> 66\%$

B-tree: format of internal nodes

- Internal nodes have the following format,
where: $k_1 < k_2 < \dots < k_m$



B-tree: format of leaf nodes

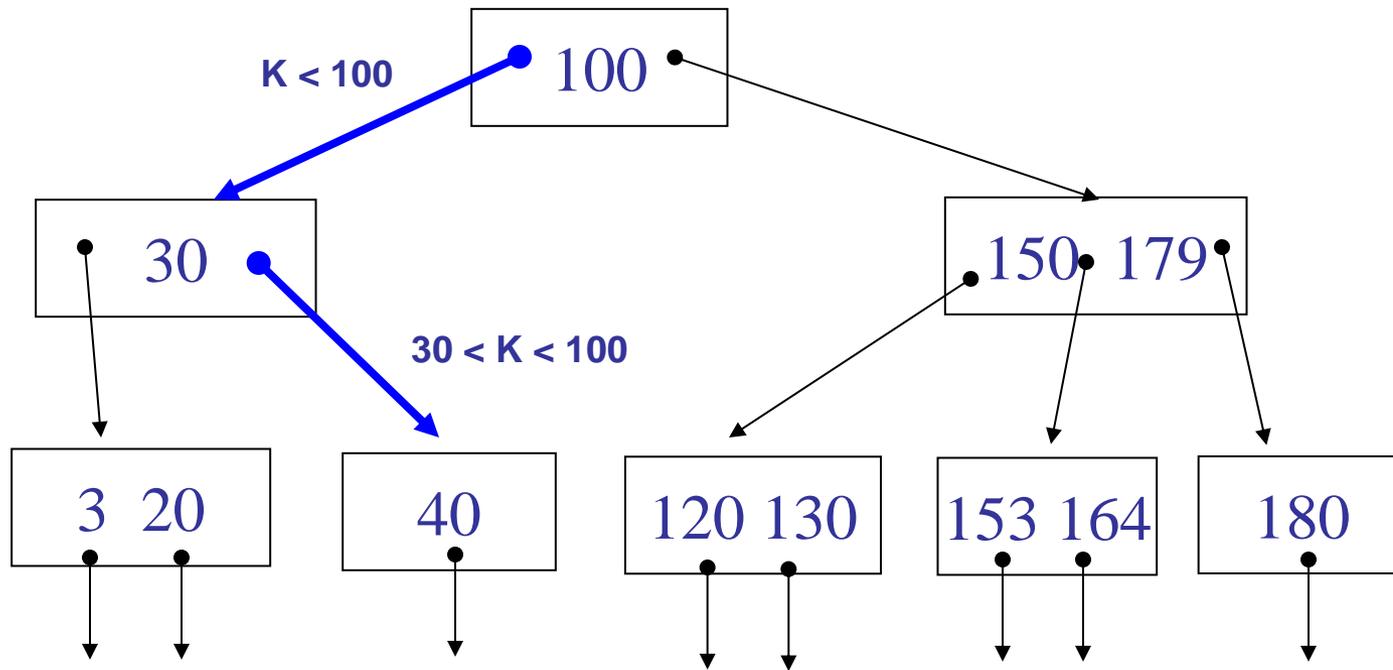
- Leaf nodes have the following format, where: $k_1 < k_2 < \dots < k_m$

(k_1, r_1)	(k_2, r_2)	...	(k_i, r_i)	(k_{i+1}, r_{i+1})	...	(k_m, r_m)	
--------------	--------------	-----	--------------	----------------------	-----	--------------	--

- Since no pointers (to sub-trees) exist, typically leaf nodes can store more entries than internal nodes

Example of B-tree

- This is an example of an order 1 B-tree:

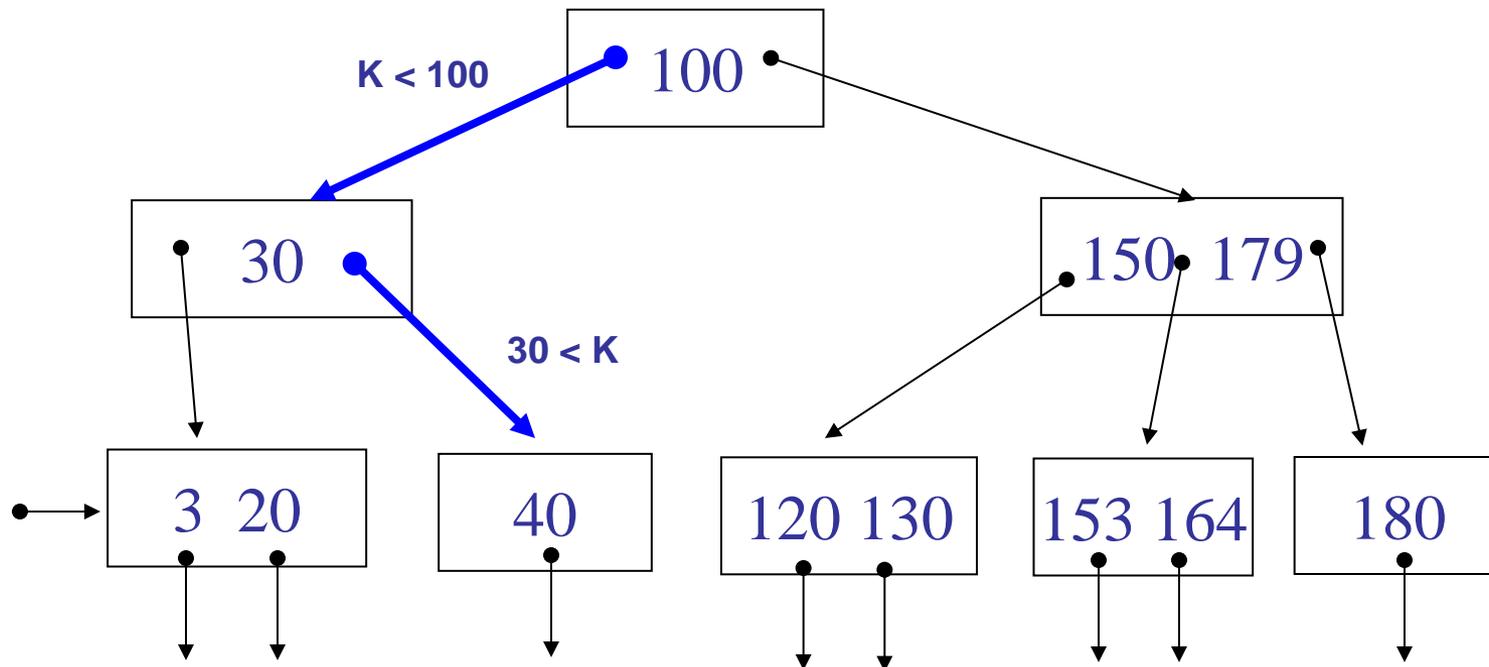


B-tree: search

- The algorithm starts by visiting the tree root
 - Usually, this is kept in main memory
- We look for the search key k within current node entries
 - If such an entry exist, **found**
 - If such entry does not exist and we are in a leaf, **not found**
 - If such entry does not exist and we are in an internal node, replace the current node with its i -th children node, where: $k_{i-1} < k < k_i$

B-tree: search example

- We search for the key 40
 - $40 < 100 \Rightarrow$ we follow the left child
 - $40 > 30 \Rightarrow$ we follow the right child



- What if we search for 30? And what for 90?

Search cost

- Every node is replaced by one of its children
- In the worst case we reach a leaf node
- $\text{Cost} \leq \text{tree height} - 1 + 1$
 - $- 1$ because the root node is in RAM
 - $+ 1$ for accessing the data file
 - Not needed if the index contains the data
- It follows that **we need to know how to compute the height of a B-tree with order d**

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

- The maximum number of nodes is reached when all nodes are full, that is they contain $2d$ entries
 - Every internal node thus has $2d+1$ child nodes

$$b_{\max} = \sum_{l=0}^{h-1} (2d+1)^l = \frac{(2d+1)^h - 1}{2d}$$

- The maximum number of entries is therefore:

$$N_{\max} = 2d \cdot b_{\max} = (2d+1)^h - 1$$

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

- The minimum number of nodes is reached when all nodes (except the root) are half full, that is they contain d entries, while the root only contains a single entry
 - Every internal node thus has $d+1$ child nodes

$$b_{\min} = 1 + 2 \sum_{l=0}^{h-2} (d+1)^l = 1 + 2 \frac{(d+1)^{h-1} - 1}{d}$$

- The minimum number of entries is therefore :

$$N_{\min} = 1 + d(b_{\min} - 1) = 2(d+1)^{h-1} - 1$$

Height of a B-tree

- Now, we know N , thus we can compute the tree height as:

$$N_{\min} \leq N \leq N_{\max}$$

$$2(d+1)^{h-1} - 1 \leq N \leq (2d+1)^h - 1$$

- Moving to logarithms, we obtain:

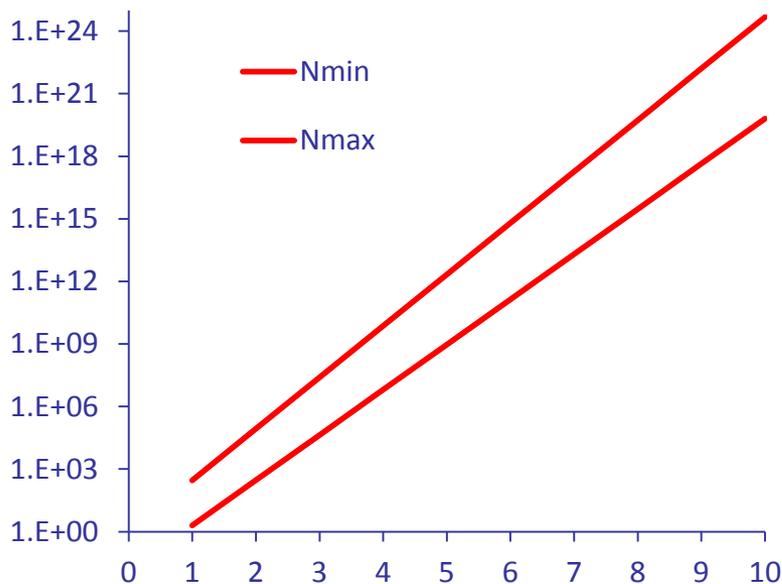
$$\lceil \log_{2d+1}(N+1) \rceil \leq h \leq \left\lfloor \log_{d+1} \left(\frac{N+1}{2} \right) \right\rfloor + 1$$

Height of a B-tree: example

- Let us consider the following values:
 - Key length: 8 bytes
 - RID length: 4 bytes
 - PID length: 2 bytes
 - Page size: 4096 bytes
- We obtain:
 - $(8+4)2d + 2(2d+1) = 4096$
 - $d = \lfloor (4096-2)/(24+4) \rfloor = 146$
- If $N = 10^9$, searching for a key value requires at most $\lfloor \log_{147}(10^9/2) \rfloor + 1 = 5$ I/O operations!
 - A binary search would require 22 accesses, supposing all pages are completely full

Height of a B-tree : considerations

- Variability of h is, with N and d fixed, **very limited** (maximum – minimum = 1)
- With the following values of d , with $h = 3$ we can store up to $(2*146+1)^3 - 1 =$ **about 25 millions keys**



P	d	N					
		1000		1000000		1000000000	
		h_{min}	h_{max}	h_{min}	h_{max}	h_{min}	h_{max}
512	18	3	3	5	5	7	8
1024	36	2	2	4	4	6	6
2048	73	2	2	4	4	5	5
4096	146	2	2	3	3	5	5
8192	292	2	2	3	3	4	4
16384	585	1	2	3	3	4	4

B-tree: range search

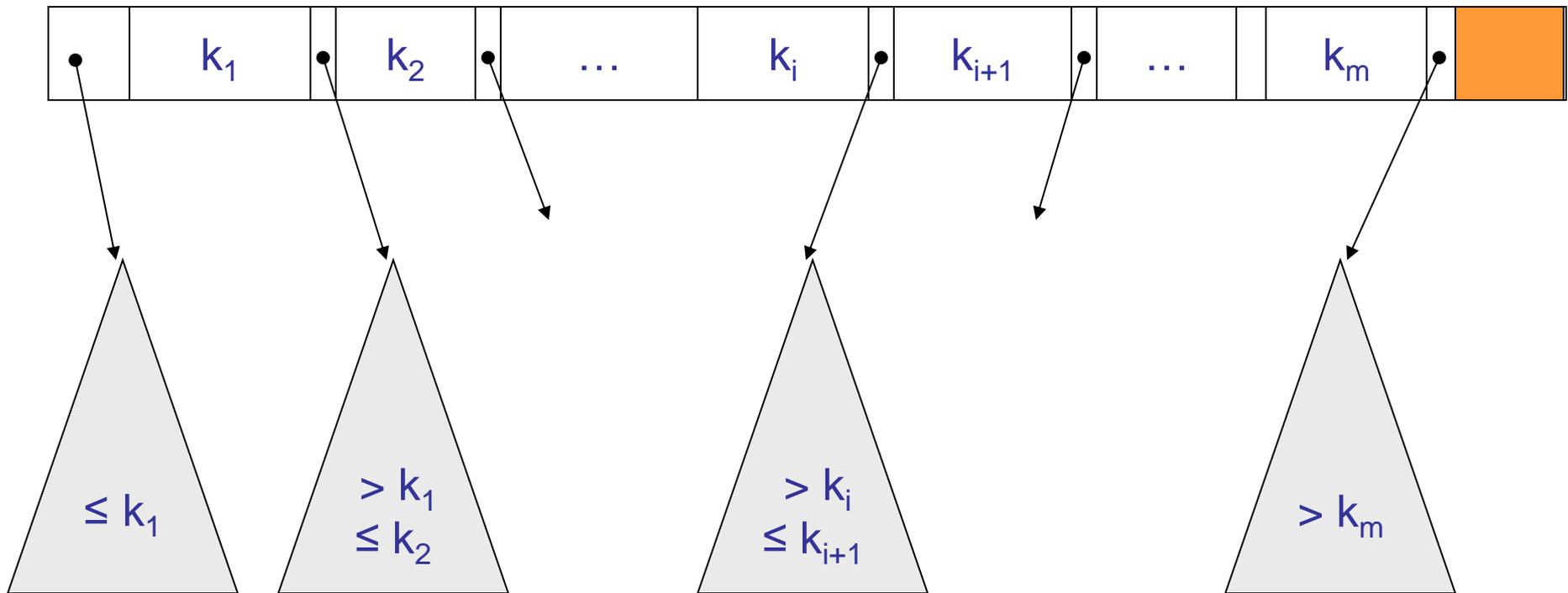
- The algorithm starts by visiting the tree root
- The tree is traversed **in-order**
- Supposing $d=0$:
 - Visit the left sub-tree
 - Visit the key
 - Visit the right sub-tree
- It is inefficient, since RIDs are stored also in internal nodes

B⁺-tree

- The main features of a B⁺-tree are:
 - The entries (k_i, r_i) are all **contained in leaf nodes**
 - The tree is higher (with respect to a B-tree)
 - Leaves are **linked as a list** (possibly, double-linked) using pointers (PIDs) to simplify range search
 - Internal nodes **contain only key values** (not necessarily corresponding to values existing in data records)
 - The order of internal nodes is higher
 - The tree is lower (with respect to a B-tree)

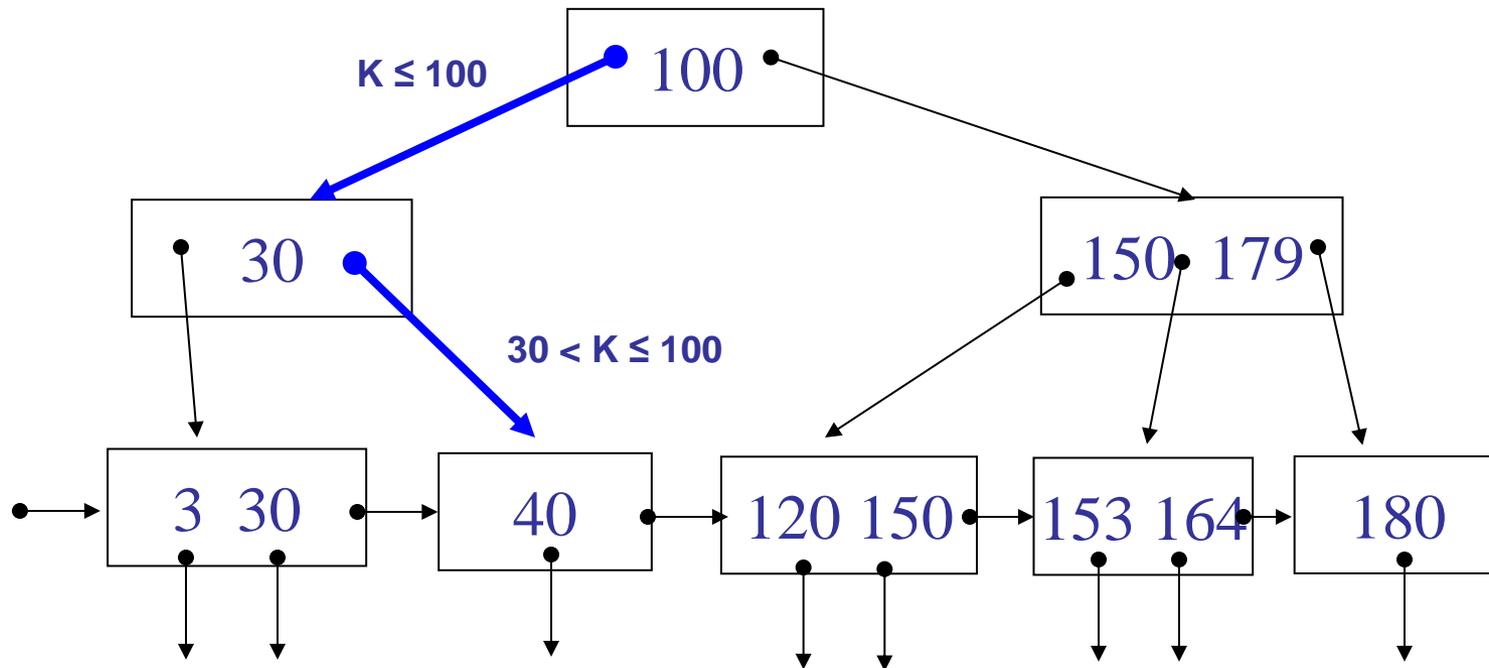
B⁺-tree: format of internal nodes

- Internal nodes have the following format,
where: $k_1 < k_2 < \dots < k_m$



Example of B⁺-tree

- This is an example of an order 1 B⁺-tree

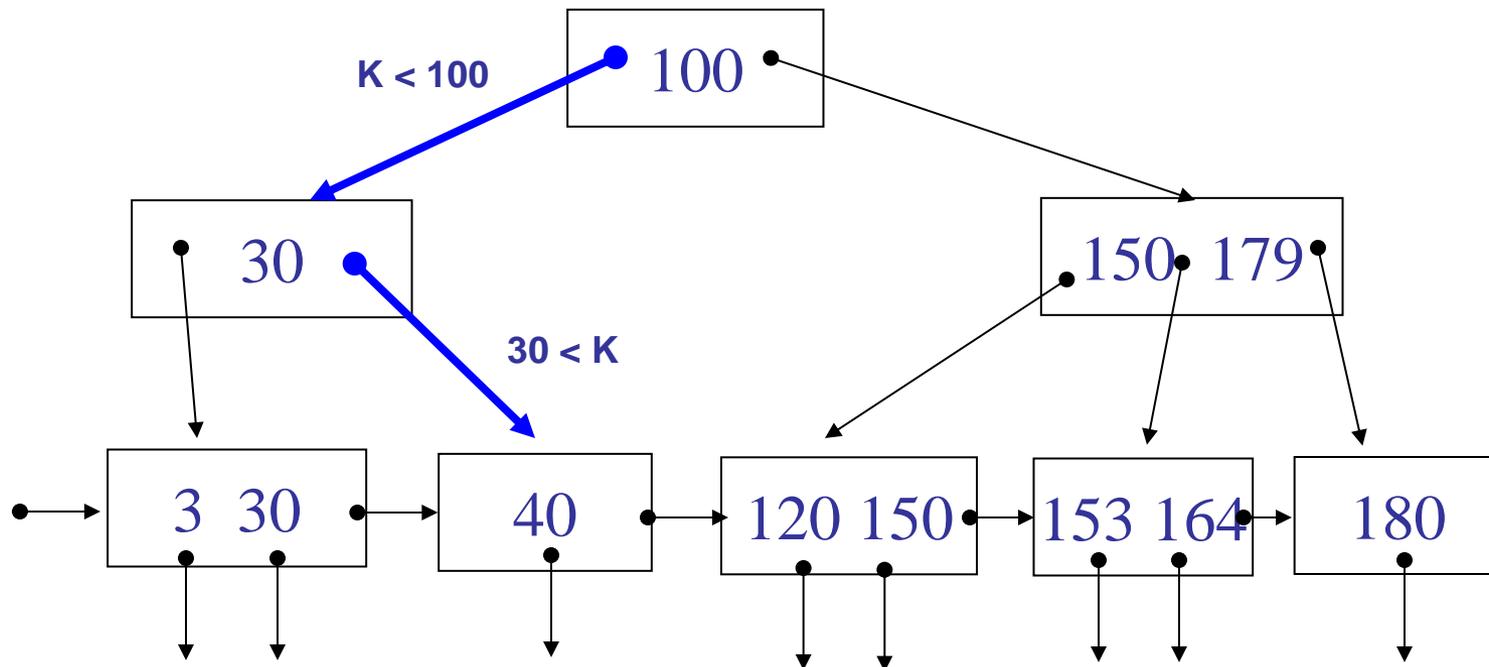


B⁺-tree: search

- The algorithm starts by visiting the tree root
 - Usually, this is kept in main memory
- We look for the search key k within current node entries
 - If we are in an internal node, replace the current node with its i -th children node, where: $k_{i-1} < k < k_i$
 - If we are in a leaf node and such an entry exist, **found**
 - If we are in a leaf node and such entry does not exist, **not found**
- Cost = tree height

B⁺-tree: search example

- We search for the key 40
 - $40 < 100 \Rightarrow$ we follow the left child
 - $40 > 30 \Rightarrow$ we follow the right child



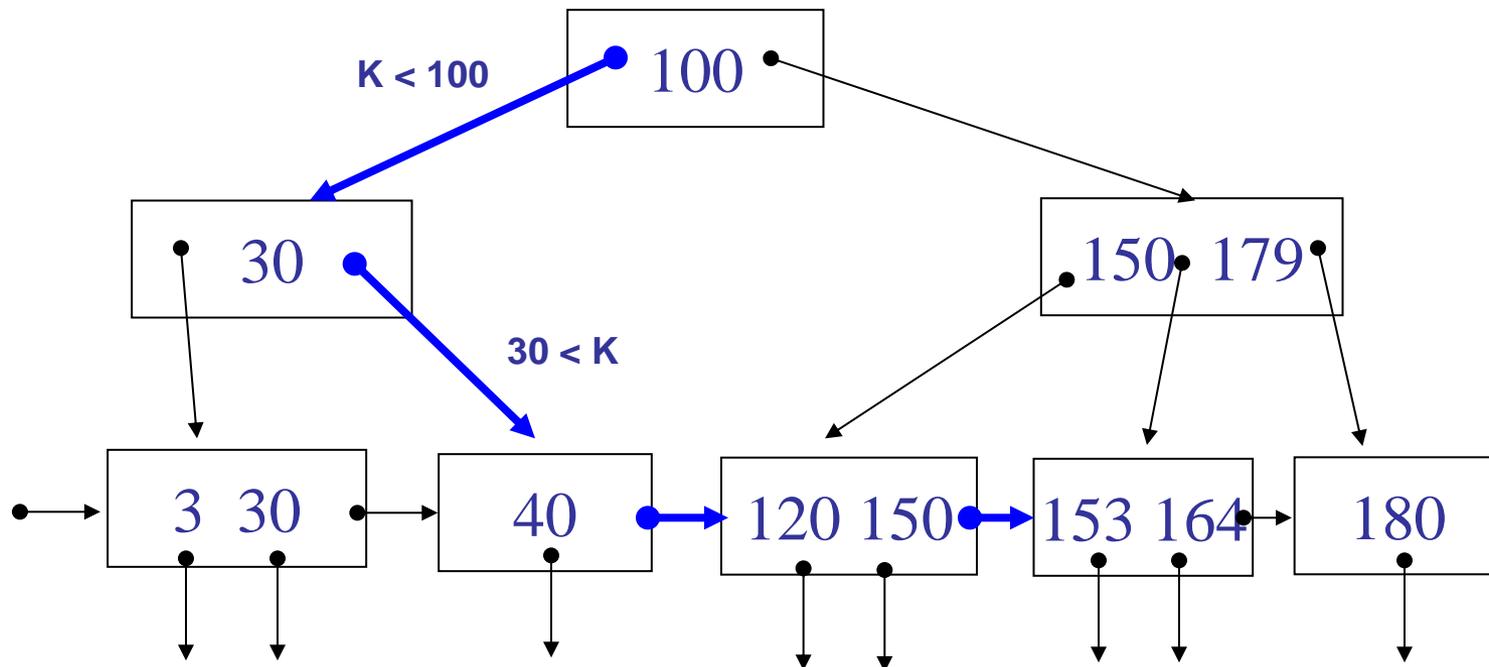
- What if we search for 30? And what for 90?

B⁺-tree: range search

- Let us suppose to search for the range $[k_{\text{low}}, k_{\text{high}}]$
- Search for the first key value $k \geq k_{\text{low}}$
- Since leaves are linked as a list, we can avoid traversing the tree and **sequentially scan leaf nodes** until we find a value $k > k_{\text{high}}$
- The RIDs we find in the list are the query result
 - If the index is unclustered, we might need to sort RIDs, in order to avoid accessing a page multiple times
 - And what if the index is sparse?

B⁺-tree: range search example

- We search for keys in the range [35,160]
 - $35 < 100 \Rightarrow$ we follow the left child
 - $35 > 30 \Rightarrow$ we follow the right child
 - From here, we scan leaves until we find $164 > 160$



B⁺-tree: insert

- Let us suppose we want to insert a new entry (k,r)
- The insert algorithm first looks for the leaf node where the new key value k should be inserted
- If there is enough space (the leaf contains less than $2d$ entries) the new pair (k,r) is inserted in the leaf and the algorithm ends
- What if there is not space enough (**overflowed** leaf)?

B⁺-tree: leaf split

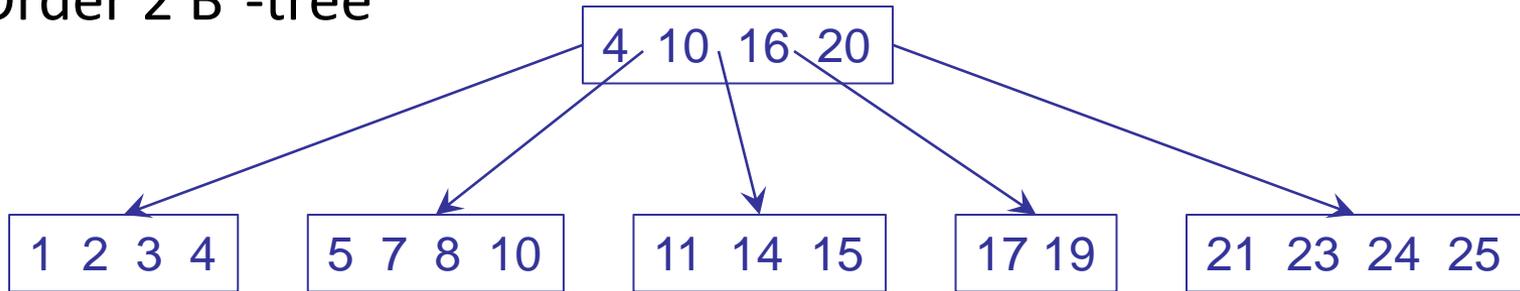
- The **overflowed** leaf F is split in two leaves (F_L e F_R)
- Each leaf will contain half of F entries
- We compute the **median** value k_c of F entries (usually $c=d$)
- We move to F_L all entries having key value $k \leq k_c$
- We move to F_R all entries having key value $k > k_c$
- In the parent node of F the pointer to F is replaced by the two pointers to F_L and F_R and by the value k_c

B⁺-tree: split propagation

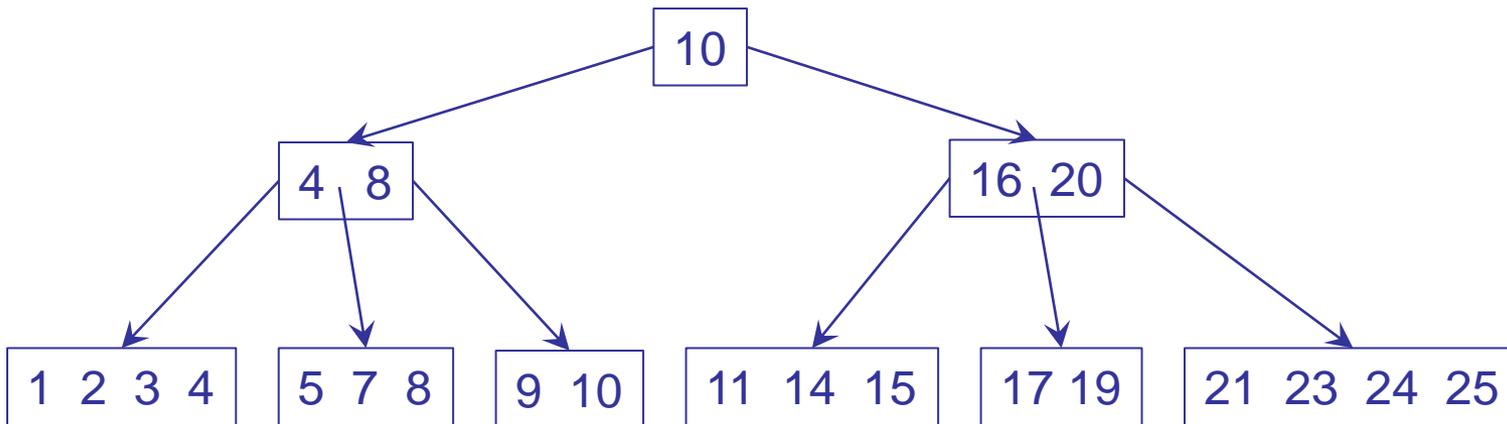
- What if the parent node of **F** is full?
- It is an overflowed node: we act as before, so we split it
- It follows that splitting propagates **recursively towards the root**
- In the worst case, the root node is also full, we split it in two and create a new root node
 - The tree **height increases**
 - This is why the root cannot have minimum occupation higher than 2

B⁺-tree: split example

- Order 2 B⁺-tree



- We insert the key value 9



B⁺-tree: insert cost

- Without split: h reads + 1 write
- With split:
 - In the worst case, we recursively split up to the root:
 h reads + $2h+1$ writes
 - For computing the average value, we should note that, for a tree with b nodes, we had $b-1$ splits
 - Since $N_{\min} = 1 + d(b-1) \leq N$
 - We obtain that the average number of splits is $(b-1)/N$, that is about $1/d$
 - Average cost: $\leq h$ reads + $1 + 2/d$ writes

B⁺-tree: alternative strategy for overflows

- As we just saw, splitting a node could be very inefficient
- An alternative strategy considers giving some entries of the overflowed node to a non-overflowed sibling node (with the same parent node) (**re-distribution**)
 - We reduce the number of splits
 - The cost is higher (we read and re-write more nodes)
 - On average, we obtain a “fuller” tree

B⁺-tree: delete

- Let us suppose we want to delete an entry (k,r)
- The delete algorithm first looks for the leaf node where the key value k should be stored
- We delete the entry from the leaf
- If the leaf contains at least d entries the algorithm ends
- What if the leaf contains $d-1$ entries (**underflow** leaf)?

B⁺-tree: underflow management

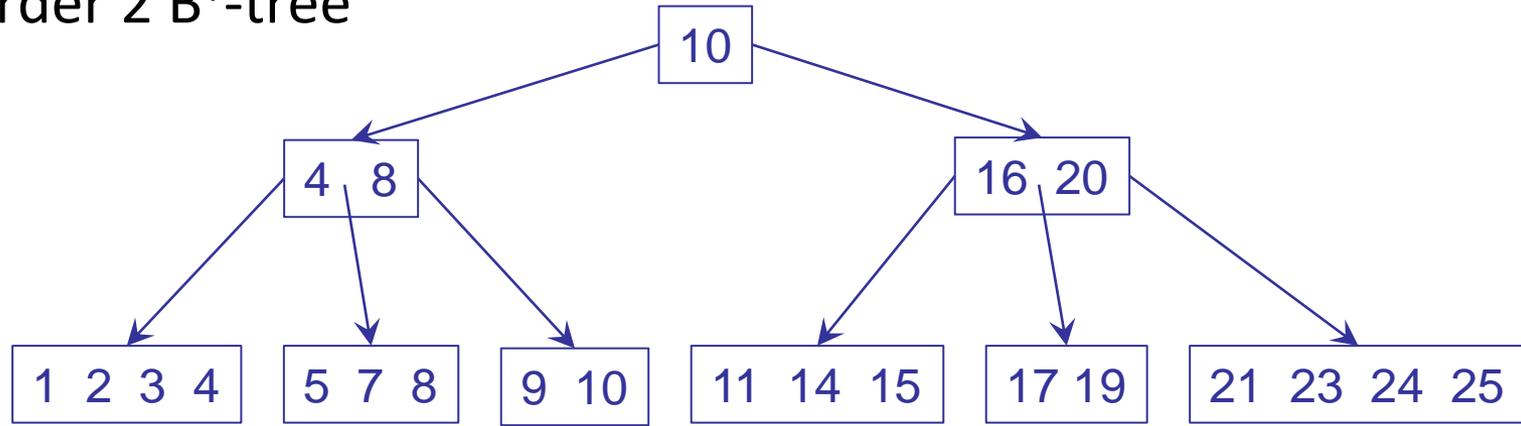
- We have two alternatives:
 - **Redistributing** entries of the underflown leaf with a sibling leaf (with the same parent node)
 - **Delete** the underflown leaf, inserting its entries in a sibling leaf
- The second alternative is possible only if the sibling leaf has d or $d+1$ entry (otherwise?)
- Deletion can be recursively propagated towards the root
 - In the worst case, the root underflows
 - The tree height decreases

B⁺-tree: redistribution

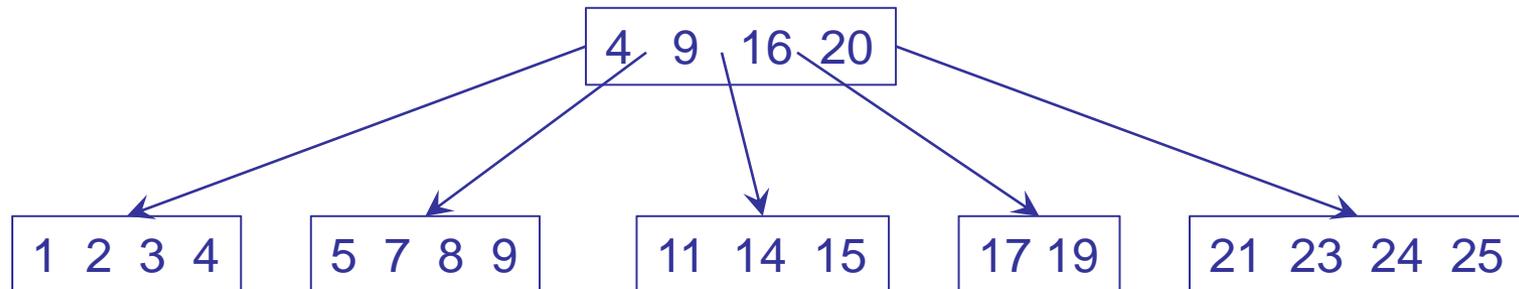
- If the sibling node contains more than $d+1$ entries we **have to redistribute**
- Entries are distributed in a balanced way among the two sibling nodes
- We should update the separating value in the parent node
 - The number of entries in the father node does not change
 - This phenomenon does not propagate towards the root

B⁺-tree: concatenation example

- Order 2 B⁺-tree

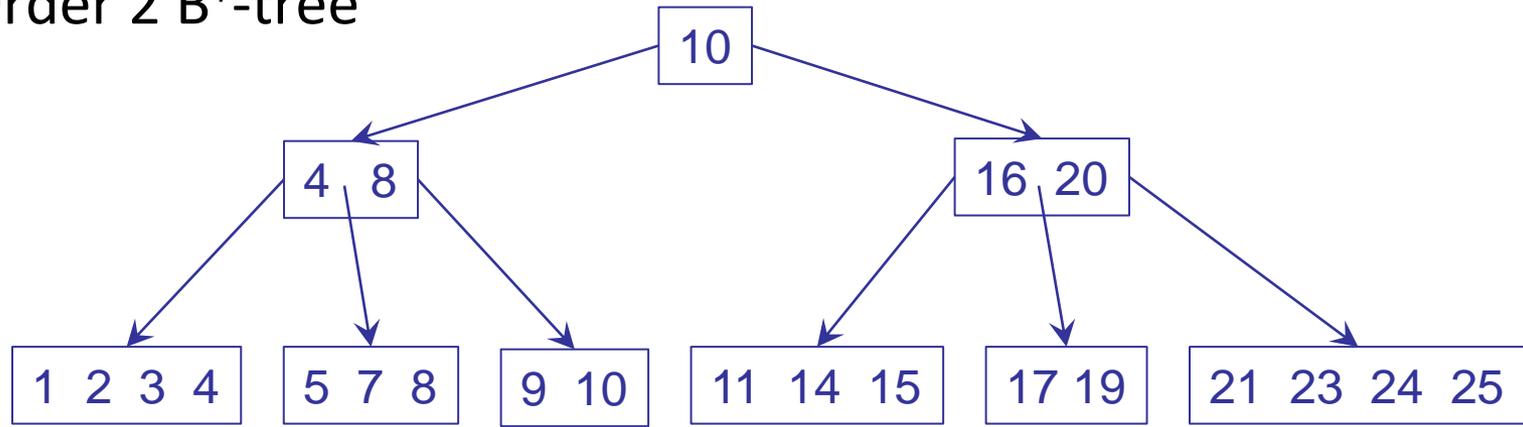


- Delete the key value 10

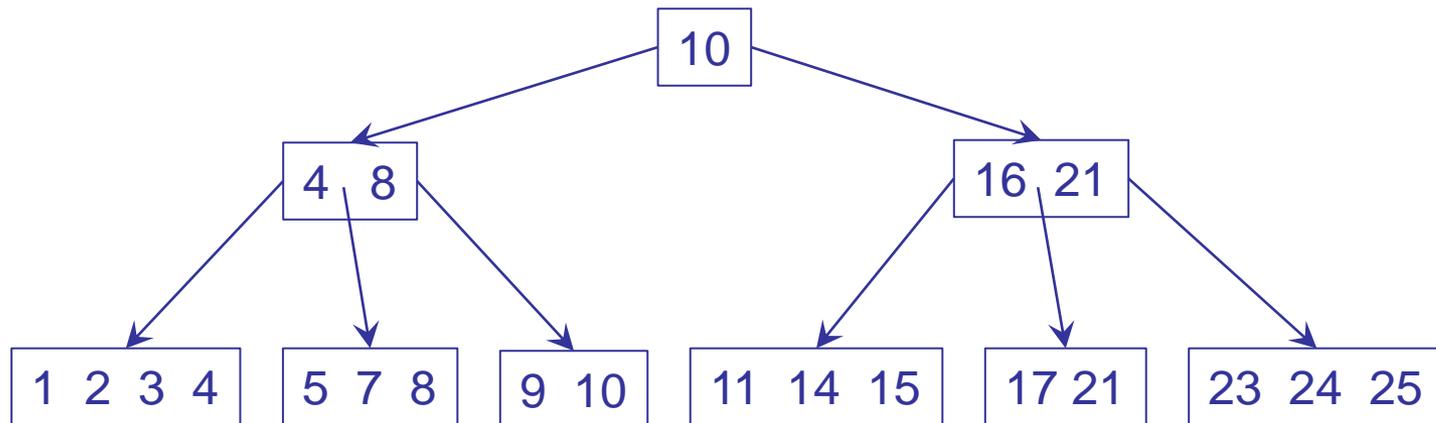


B⁺-tree: redistribution example

- Order 2 B⁺-tree



- Delete the key value 19



B⁺-tree: delete cost

- Without underflow: h reads + 1 write
- With underflow:
 - Every concatenation costs 1 read + 2 writes
 - In the worst case, we recursively concatenate up to the root
 - Concatenation for all levels, except the upper two
 - Redistribution in the root child
 - Maximum cost: $2h-1$ reads + $h+1$ writes
 - Average number of concatenations: $1/d$
 - Average cost: $\leq h+1+1/d$ reads + $1+2+2/d$ writes

B⁺-tree: memory occupation

- Every internal node contains at most $2d$ key values and $2d+1$ PIDs
- The order of a B⁺-tree is therefore computed as:

$$d = \left\lfloor \frac{\text{pagesize} - \text{PIDsize}}{2(\text{keysize} + \text{PIDsize})} \right\rfloor$$

- For trees, we need to know whether the index is primary or secondary
 - In some cases, the leaf level can coincide with the data file

B⁺-tree: number of leaves (primary index)

- In every leaf we find at most $2d$ entries (k,r) and 1 or 2 pointers to sibling nodes
- Therefore, the order of leaf nodes is:

$$d_{leaf} = \left\lfloor \frac{pagesize - 2PIDsize}{2(keysize + RIDsize)} \right\rfloor$$

- Thus, the number of leaf nodes is:

$$NL = \left\lceil \frac{N}{d_{leaf} \cdot u} \right\rceil$$

- The average fill factor u usually equals \log_2

Height of a B⁺-tree

- Since we know NL , we can now compute the B⁺-tree height:

$$NL_{\min} \leq NL \leq NL_{\max}$$
$$2(d+1)^{h-2} \leq NL \leq (2d+1)^{h-1}$$

- Moving to logarithms, we obtain:

$$\lceil \log_{2d+1}(NL) \rceil + 1 \leq h \leq \left\lfloor \log_{d+1}\left(\frac{NL}{2}\right) \right\rfloor + 2$$

B⁺-tree in practice

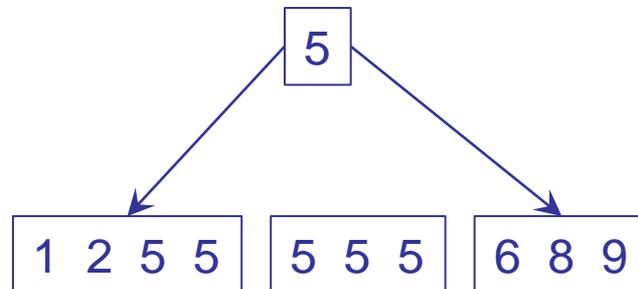
- Secondary index
- B⁺-tree as a data file
- Variable-length keys
- Key compression
- Multi-attribute B⁺-tree
- Bulk-loading
- Implementing B⁺-tree: GiST

B⁺-tree as a secondary index

- When duplicate values exist, every key value could not be paired with just a RID, but to a **RID list**
 - Typically, the list is kept sorted for PID values (why?)
- What if the RID list is (very) long?
 - It can be the case that the size of a single entry exceeds the page size
- Possible solutions:
 - Overflow pages for the leaf
 - Duplicating the keys in the index
 - Using PIDs instead of RIDs
 - Posting file

B⁺-tree: duplicating keys

- This means that several entries exist with the same key value (but different RID)
- We should slightly modify the search algorithm (in practice, this becomes a range search)
- Not every leaf is “addressed”
- Inefficient when deleting
 - We insert the RID as “part” of the key
 - The index is now “primary”



B⁺-tree: using PIDs

- Instead of keeping a list of RIDs, we keep a list of PIDs, including only those pages containing at least a record with that key value
 - $\text{PIDsize} \leq \text{RIDsize}$
 - number of PIDs \leq number of RID
- This is efficient if a page usually contains several records with the same key value
 - Clustered index

B⁺-tree: posting file

- The RID list is stored in a separate file
- Every entry in the posting file has the format (k,l) , where
 - k key value
 - l RID list having key value k
- B⁺-tree entries contain pointers to posting file entries
- We introduce an additional indirection level
 - The cost of each operation is increased by 1

B⁺-tree as a data file

- Tree leaves contain the data records
 - In its original version, the B-tree was introduced as a data file organization
- Pros:
 - The data file is automatically sorted
 - Sorting is kept also when insertion/deletions are present
- Cons:
 - Updates move records, thus changing their RIDs...

B⁺-tree: variable-length keys

- What we saw up to now is only valid for fixed-length entries
- This cannot hold in some cases:
 - Variable-length keys(e.g., varchar)
 - Secondary index
 - Index as a data file
- In such cases, the concept of order is no longer valid, and we should apply considerations on minimum node utilization

B⁺-tree: key compression

- Clearly, in order to minimize access costs, we should strive to have high values of d
- We can consider reducing the key length (**compress**) within nodes
 - B⁺-trees are not constrained to have existing key values within internal nodes
 - Their goal is to distinguish the content of sibling nodes
 - Example:

Semenzara ... Serbelloni Mazzanti Vien Dal Mare

Silvani...

“Ser” is enough...

B⁺-tree: multi-attribute searching

- Suppose we have a multi-attribute search predicate

```
SELECT * FROM persone  
WHERE cognome="Rossi"  
AND anno>1990
```

- How can we exploit an index to efficiently solve such query?

B⁺-tree: using multiple indices

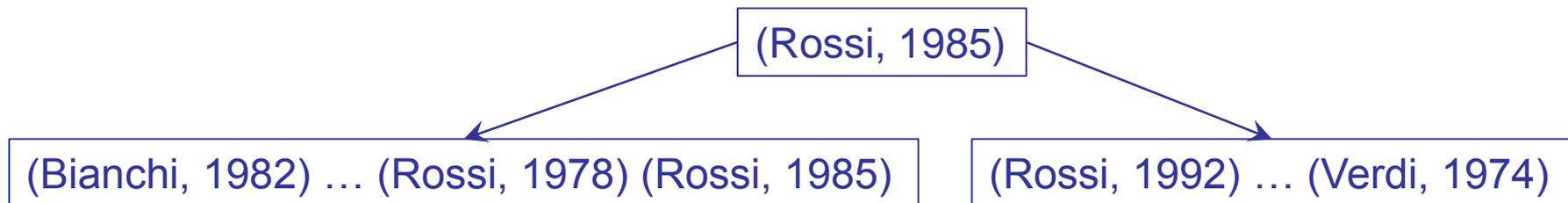
- A first technique exploits a single index:
 - We use only the first (or the second) predicate to retrieve all result records
 - Then, we verify the remaining predicate on such records
- A second technique exploits both indices:
 - We separately retrieve RIDs of records satisfying either predicate
 - We perform the intersection of the results
 - This is efficient if RIDs are sorted

Multi-attribute B⁺-tree

- In both cases, the additional work to be done can nullify the advantage of using the index
- We can build a multi-attribute index
- The key is composed by the concatenation of relevant attributes
- Sorting is performed on lexicographic order
 - We consider the following attribute only if the previous attribute is equal

Multi-attribute B+-tree: example

- Index on (cognome, anno)



- With this we can efficiently solve queries on **cognome** and **(cognome, anno)**
 - What about range searches?
- Not on **anno** alone (why?)
- With n attributes we can have $n!$ different indices

B⁺-tree: bulk-loading

- Come visto, l'inserimento di elementi in un B⁺-tree provoca uno split ogni d record inseriti
- Spesso la decisione di costruire un indice non avviene in fase di creazione del DB ma in seguito
 - Ad esempio, ci accorgiamo che una query è lenta
- È conveniente creare un indice su una tabella numerosa effettuando l'inserimento uno a uno?
 - Evidentemente no...

B⁺-tree: loading

- A list of entries (key,RID) is created and sorted for key values
- Such list (appropriately paged) corresponds to the leaf level
 - In case, we can use a fill factor lower than 100%
- Starting from the key values in every leaf node a list of entries (key,PID) is created
- Such list (appropriately paged) corresponds to the first level above the leaves
- ... and so on, recursively, until we reach the root (all entries can be contained in a single node)

Implementing B⁺-tree: GiST

- GiST (**G**eneralized **S**earch **T**ree) (Hellerstein, Naughton, Pfeffer, '95) is not a specific access method, but a general structure that, when appropriately instantiated, can behave like a B⁺-tree, a R-tree, etc.
- The main goal is not defining a new index type, rather simplifying the development of different access methods
 - **For example:** B⁺-tree in the Postgres system requires about 3000 lines of C code
 - The same B⁺-tree implemented as a GiST instance, requires about 500 lines of code

Basic GiST concepts

- Instead of considering specific queries, each query is seen as a generic predicate (q)
- Every GiST node contains a list of **entries** (p, ptr), where p is a **predicate** (key) and ptr is a **pointer**
- We only access the sub-tree referred by the pointer ptr associated to the key p if p is **consistent** with the q predicate, that is, only if p does not exclude the possibility that the sub-tree could contain some records satisfying q
- The only GiST constraint is the **monotonicity** of p , that should hold for every record contained in the sub-tree pointed by ptr

Predicate monotonicity

- We should check whether, given a query q , the sub-tree pointed by ptr should be accessed
- E.g.: $q = \text{italian}(X) \&\& \text{student}(X)$
- E.g.: $q = \text{mexican}(X) \&\& \text{freshman}(X)$
- E.g.: $q = \text{french}(X) \&\& \text{worker}(X)$

$p = \text{european}(X) \&\& \text{graduating}(X)$
 $ptr =$



GiST properties

- Independently of the specific instance, every GiST has the following properties:
 - A GiST is a **perfectly balanced paged tree**
 - An **entry** in an **internal node** is a pair (p, ptr) , where:
 - p predicate used as a search key
 - ptr pointer to another GiST node
 - An **entry** in a **leaf node** is a pair (p, ptr) , where:
 - p key value
 - ptr pointer to a tuple (DB record) satisfying p
 - Every node (except the root) contains at most M entries and at least $f M$ entries, with $2/M \leq f \leq 1/2$ (f = minimum fill factor)
 - With variable-length entries, we use values dependent on entries size
 - The root, if it is not a leaf, contains at least **two** entries
 - For each entry (p, ptr) in an internal node, p holds for all tuples reachable from ptr

The real GiST

- The base of GiST is the definition of a number of methods, used to manage:
 - key values (**Key methods**)
 - tree nodes (**Tree methods**)
- The GiST definition only specifies Tree methods
- Implementation of Key methods is given when GiST is instantiated to manage a specific type of keys
 - E.g.: real values (B⁺-tree)
 - E.g.: multi-dimensional ranges (R-tree)
- Since Key methods are called by Tree methods, we need to provide a standard interface for the former ones

Key methods: search

- **Consistent(E,q)**
 - **Input:** Entry $E=(p,ptr)$ and search predicate q
 - **Output:** **if** $p \ \& \ q == \text{false}$ **then** false **else** true
- The goal of **Consistent** is “pruning” the search space (that is, eliminating sub-trees)
- If the predicate of a sub-tree is not consistent with the query, then we avoid visiting the whole sub-tree

Consistent: comments

- If the test $p \ \& \ q$ is computationally demanding, we can still use a conservative approximation, that is, **Consistent** returns “true” even if $p \ \& \ q = \text{false}$
 - This only affects efficiency, not correctness of results (we access a sub-tree, although the records contained therein do not belong to the query result)
- **Consistent** (just like the other methods) is specified so as to work with arbitrarily complex predicates
 - In practice, predicates can be “restricted” in order to improve algorithms performance

Consistent in B⁺-tree

- Every predicate is a range $[x,y[$
- If the query is a single value v
 - **Consistent** returns true if and only if $x \leq v < y$
- If the query is a range $[v,w[$
 - **Consistent** returns false if and only if $x \geq w$ or $y \leq v$

Key methods: predicate creation

- **Union(P)**
 - **Input:** Set of entries $P = \{(p_1, ptr_1), \dots, (p_n, ptr_n)\}$
 - **Output:** A predicate r holding for all tuples reachable by way of one of the entry pointers
- The goal of **Union** is providing the information needed to characterize the predicate of a parent node, starting from predicates of children nodes
- In general, r can be logically derived as a predicate such as $(p_1 \mid \dots \mid p_n) \Rightarrow r$

Union in B⁺-tree

- Given $P = \{([v_1, w_1[, ptr_1), \dots, ([v_n, w_n[, ptr_n))\}$
- Returns $[\min\{v_1, \dots, v_n\}, \max\{w_1, \dots, w_n\}[$

Key methods: key compression

- **Compress(E)**
 - **Input:** Entry $E = (p, ptr)$
 - **Output:** Entry $E' = (p', ptr)$, with p' compressed representation of p
- The goal of **Compress** is providing a more efficient representation of the p predicate
 - E.g.: separators in place of totally ordered ranges
 - E.g.: prefixes from strings (prefix-B⁺-tree)

Key methods: key decomposition

- **Decompress(E)**
 - **Input:** Entry $E' = (p', ptr)$, with $p' = \text{Compress}(p)$
 - **Output:** Entry $E = (r, ptr)$, with $p \Rightarrow r$
- Compression is, in general, lossy
 - E.g.: prefix-B⁺-tree
- Condition $p \Rightarrow r$ requires that, if information loss occurs, what we obtain with **Decompress** is a predicate that holds if p holds
- The simplest case is when **Decompress** is the identity function

Key methods: insert

- $\text{Penalty}(E_1, E_2)$
 - **Input:** Entries $E_1 = (p_1, ptr_1)$ and $E_2 = (p_2, ptr_2)$
 - **Output:** A “penalty” value resulting from inserting E_2 in the sub-tree rooted in E_1
- **Penalty** is used by Tree methods **Insert** and **Split**, and is needed to compare different alternatives for updating the tree

Penalty in B⁺-tree

- $E_1 = ([x_1, y_1[, ptr_1)$ e $E_2 = ([x_2, y_2[, ptr_2)$
- If E_1 is the first entry in its node
 - Penalty returns $\max\{y_2 - y_1, 0\}$
- If E_1 is the last entry in its node
 - Penalty returns $\max\{x_1 - x_2, 0\}$
- Otherwise
 - Penalty returns $\max\{y_2 - y_1, 0\} + \max\{x_1 - x_2, 0\}$

Key methods: split

- **PickSplit(P)**
 - **Input:** Set of $M+1$ entries
 - **Output:** Two sets of entries, P_1 e P_2 , with cardinality $\geq f M$
- **PickSplit** implements the real split strategy, which is not detailed at this level
- Usually, we try to minimize some metric similar to **Penalty**

Picksplit in B⁺-tree

- P_1 contains the first $(M+1)/2$ entries
- P_2 contains the remaining entries
- With variable-length entries we use some criterion on entry size, not on number of entries
 - This could lead to violate the minimum utilization constraint

Tree methods

- **Tree methods** call other tree methods and use defined **Key methods**
- We implicitly assume that keys are compressed on write and de-compressed on read

Tree methods: architecture

- **Search**: calls **Consistent**
- **Insert**: calls **ChooseSubtree**, **Split** and **AdjustKeys**
- **ChooseSubtree**: calls **Penalty**
- **Split**: calls a **PickSplit** and **Union**
- **AdjustKeys**: calls **Union**
- **Delete**: calls **Search** and **CondenseTree**
- **CondenseTree**: calls **AdjustKeys** and **Insert**

Search

- The search algorithm recursively descend the tree, using **Consistent** to prune useless branches

Search(R,q)

Input: (sub-)tree rooted at **R** and query **q**

Output: all records satisfying **q**

if **R** is not a leaf

for each **E** in **R**

if **Consistent(E,q)** **Search(*(E.ptr),q)**

else for each **E** in **R**

if **Consistent(E,q)**

 add ***(E.ptr)** to result

Search in linear domains

- For (totally ordered) linear domains, as in the B+-tree case, GiST specifies a more efficient extension that exploits the contiguity of leaves to solve range queries
- In particular, **Search** reaches the first leaf which is “consistent” with query q
- After that, pointers in the linked list of leaves are used until a leaf is reached which is “inconsistent” with query q

Insert

- The insertion algorithm is used to both insert new entries and re-insert “orphaned” entries, resulting from underflows
- For this, the input also includes the tree level where the entry should be inserted, with the understanding that leaves are at level 0
- In case of overflow, we call the **Split** function and propagate updates towards the root

Insert

Insert(R,E,I)

Input: Tree rooted at R, entry E, level I

Output: Tree with E inserted at level I

N = **ChooseSubtree**(R,E,I)

if E can be inserted in N

 insert E in N

else **Split**(R,N,E)

AdjustKeys(R,N)

ChooseSubtree

- **ChooseSubtree** uses **Penalty** to recursively determine the sub-tree where **E** should be inserted

ChooseSubtree(R,E,I)

Input: Tree rooted at **R**, entry **E**, level **I**

Output: node **N** at level **I** where **E** should be inserted

if **R** is at level **I** **return** **R**

else choose among all entries $F = (p', ptr')$ in **R** the one minimizing **Penalty**(F,E)

return **ChooseSubtree**(***F.ptr'**),E,I)

Split

- **Split** uses **PickSplit** to divide entries of an overflown node
- The parent of the overflown node is still on the call stack

Split(R,N,E)

Input: Tree rooted in **R**, node **N**, entry **E**

Output: Tree with **N** split and **E** inserted

$P_1, P_2 = \text{PickSplit}(\{\text{entries of } N\} \cup \{E\})$

insert P_1 in **N** e P_2 in a new node **N'**

$p' = \text{Union}(P_2)$, $\text{ptr}' = \&N'$, $E' = (p', \text{ptr}')$

if E' can be inserted in **Parent(N)**

then insert E' in **Parent(N)**

else **Split(R,Parent(N),E')**

$F = \text{entry in Parent(N) with } F.\text{ptr} = \&N$

$F.p = \text{Union}(P_1)$

AdjustKeys

- **AdjustKeys** recomputes key values (predicates) following an update
- The algorithm recursively climbs the tree and it terminates when it reaches the root or an already accurate key value

AdjustKeys(R,N)

Input: Tree rooted at R , node N

Output: Tree with N ancestors having correct and accurate key values

if $N = R$ or for entry $E = (p, ptr)$, with $ptr = \&N$,
 $E.p = \text{Union}(\{\text{entry of } N\})$ already holds

return

else $E.p = \text{Union}(\{\text{entry of } N\})$

AdjustKeys(R,Parent(N))

Delete

- **Delete** keeps the tree balanced, decreasing its height if the root, when **CondenseTree** terminates, has a single child

Delete(R,E)

Input: Tree rooted at R, entry $E = (p,ptr)$

Output: Tree with E removed

Search(R,E.p)

if E not found **return**

L = node containing E, remove E from L

CondenseTree(R,L)

if R has a single entry

remove R

make the child of R the new GiST root node

CondenseTree

- **CondenseTree** manages the re-insertion at the original level of orphaned entries from underflowed nodes, kept in a set

CondenseTree(R,L)

Input: Tree rooted at **R** and leaf **L**

Output: New tree

$N = L, Q = \{\}$

if $N = R$ **goto end**

else $P = \text{Parent}(N), E = \text{entry in } P: E.\text{ptr} = \&N$

if $\#\{\text{entry of } N\} < k M$

$Q = Q \cup \{\text{entry of } N\}, \text{ remove } E \text{ from } P, \text{ AdjustKeys}(R,P)$

if E was not removed from P **AdjustKeys}(R,N)**

else $N = P, \text{ restart}$

for each E in Q **Insert}(R,E,\text{level}(E))**

Performance evaluation

- Up to now, we computed the search performance of a B⁺-tree as a primary index
- In the case of a secondary index we should know
 - How many leaves contain result entries
 - How many data pages contain result records
- We will suppose that:
 - RID lists in the leaves are sorted
 - We will not access a data page more than once (for each key value)
 - Attribute values are uniformly distributed in the data file
 - Every value is repeated (on average) N/K times
 - Records are uniformly distributed in pages of the data file

Estimating the number of result pages

- If we should retrieve R records in P pages
 - $1/P$ = probability that a record is inside a given page
 - $1-1/P$ = prob. that a page does not contain a record
 - $(1-1/P)^R$ = prob. that a page does not contain any record
 - $1-(1-1/P)^R$ = prob. that a page contains at least one record
- Multiplying this by the number of pages, we obtain the average number of pages to be accessed

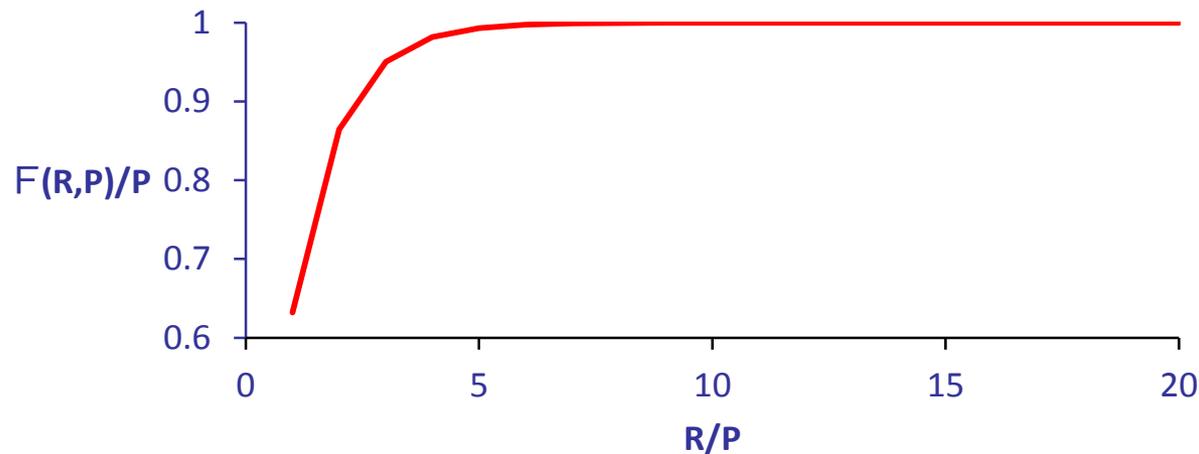
$$\bar{Q}(R,P) = P (1-(1-1/P)^R) \leq \min\{R,P\}$$

- Given a drawer with an infinite number of socks in P colors, how many different colors we have, in average, when R socks are taken?

Cardenas model

- The model we just saw (**Cardenas**) assumes pages with infinite size (note that N does not appear in the formula)
 - This leads to an appreciable underestimation of the correct value for pages with less than about 10 record
- If $R = N$, the formula returns a value lower than P

$$F(N,P) = P (1-(1-1/P)^N) \approx P (1-e^{-N/P})$$



Yao model

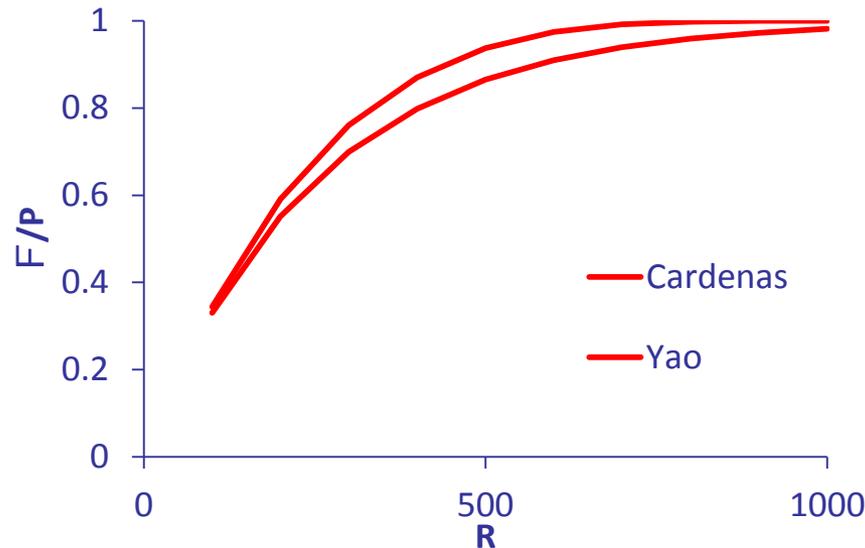
- The model by Yao takes into account the actual capacity $C = N/P$ of pages
- The model considers all possible ways for allocating the R result records in the P pages
 - $\binom{N}{R}$ = number of combinations
 - $\binom{N-C}{R}$ = number of combinations excluding a page
 - $\binom{N}{R} - \binom{N-C}{R}$ = combinations including a given page
 - $1 - \frac{\binom{N-C}{R}}{\binom{N}{R}}$ = probability that the page contains at least one record

Formula by Yao

- Multiplying by the number of pages, we obtain the average number of accessed pages

$$\Phi(R, N, C) = NP \times \left(1 - \frac{\binom{N-C}{R}}{\binom{N}{R}} \right)$$

- E.g.: $N=1000$, $P=250$



Comparing the models

- When pages contain a variable number of records, it can be proven that the formula by Yao overestimates costs
- If record allocation is not uniform, both models overestimate costs
- If R is large, computing the formula by Yao could be computationally expensive

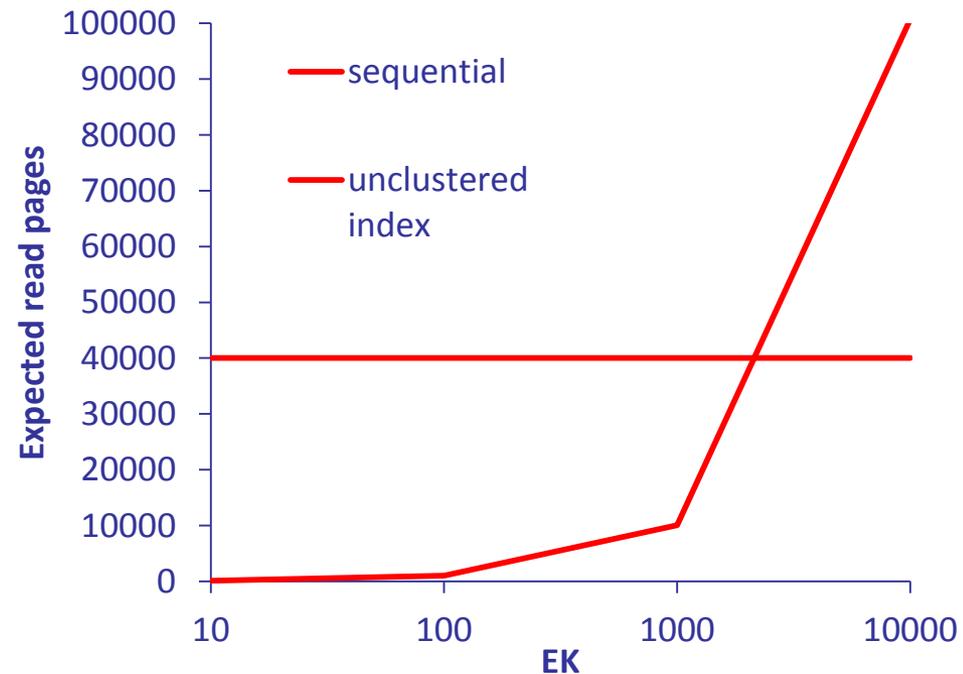
$$\Phi(R, N, C) = NP \times \left(1 - \prod_{i=1}^R \frac{N - C - i + 1}{N - i + 1} \right)$$

Cost of index access

- Total cost = index cost + data pages cost
- Index cost = cost for the first leaf + cost for reading all leaves
 - Cost for the first leaf = $h-1$
 - Number of leaves = $\lceil L * EK / K \rceil$
- Data pages cost: EK times the formula by Cardenas (or Yao)
 - = $EK \times F(N/K, P)$
- This should be compared with sequential cost
 - = P

Example

- File with $N = 10^6$ records in $P = 40000$ pages
- Unclustered index on an attribute with $K = 10^5$ values with $L = 7045$ leaves and height $h = 4$
- We see that the tree height contribution is entirely irrelevant for costs



Cost for range search $A \in [x,y]$

- Total cost = index cost + data pages cost
- Index cost = cost for the first leaf + cost for “sequentially” reading all leaves
 - Cost for the first leaf = $h-1$
 - Number of leaves = $\lceil fs * L \rceil$
 - fs = selectivity factor of predicate = $(y-x)/(maxA-minA)$
- Data pages cost: $\lceil fs * K \rceil$ times the formula by Cardenas (or Yao)
 - = $\lceil fs * K \rceil F(N/K, P)$ (sorting attribute)
 - = $\lceil fs * P \rceil$ (non-sorting attribute)

Hash indices

- Differently from table-based techniques, where the association $\langle \text{key}, \text{RID} \rangle$ is explicitly stored, a hash-based organization uses a hash function, H , transforming every key value into an address



Hash indices: collisions

- Except for particular cases, hash functions are non-injective:

$$k_1 \neq k_2 \not\Rightarrow H(k_1) \neq H(k_2)$$

thus **collisions** might occur

- If k_1 and $k_2 \neq k_1$ collide, $H(k_1) = H(k_2)$
- A hash function that does not generate collisions is called **perfect**
- Every address generated by the hash function identifies a logical page, or **bucket**
- The number of “elements” (key values for indices, records for data organizations) that are contained in a bucket defines the capacity, **C**, of buckets

Hash indices: overflow

- The memory composed by buckets addressable by the hash function is called **primary area**
- If a key is assigned to a bucket already containing C keys, the bucket **overflows**
- Managing overflows could require, depending on the specific technique, using a separate memory area, called **overflow area**

Static and dynamic hash indices

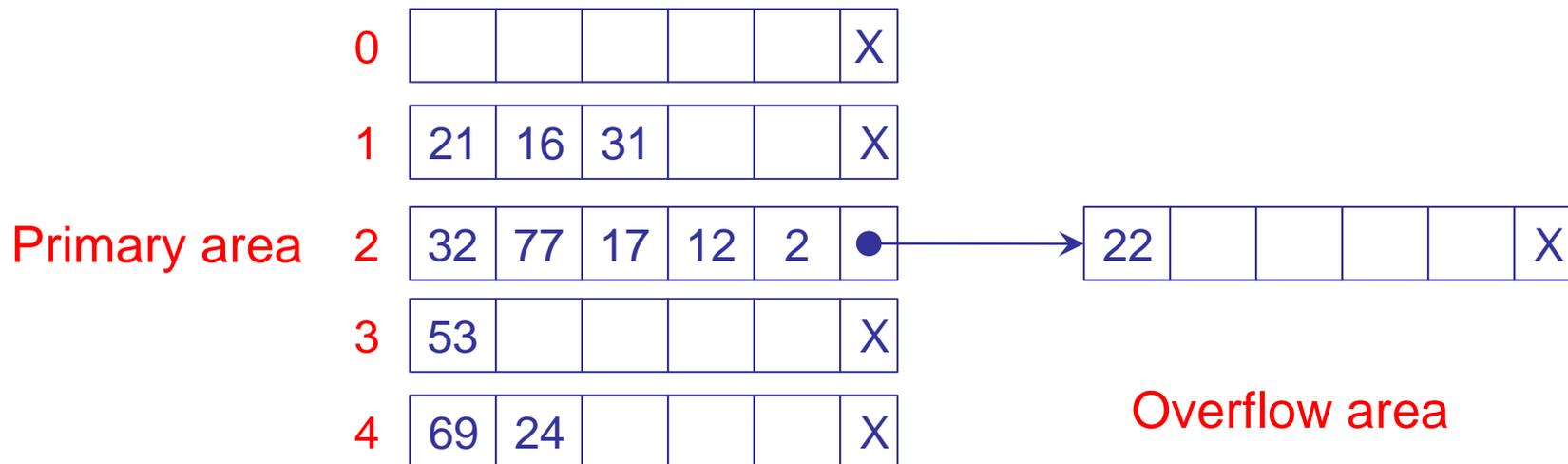
- A hash function should be surjective, thus able to generate P different addresses, as many as the primary area buckets
- If, for a specific hash technique, the value of P is constant, the hash technique is called **static**
 - In this case, the primary area size is part of the design of the hash index
- On the other hand, if the primary area can expand and contract, in order to adapt to the actual data volume, the hash technique is called **dynamic**
 - In this case, multiple hash functions are needed
- The first dynamic hash techniques were proposed around the end of '70s, while static techniques were first developed in the '50s

Common features of hash indices

- For both static and dynamic techniques, some common aspects are worth mentioning:
 - Choice of the actual hash function H
 - Overflow management policy
 - Capacity C of primary area buckets
 - Capacity C_{ov} (not necessarily equals to C) of buckets in the overflow area (if applicable)
 - Utilization of allocated memory
- Hash techniques are usually primary (but also secondary)
- Usually, hash functions do not preserve order, that is they are non-monotone
 - Using hash indices is not recommended in cases where range queries are possible (or frequent)

Static hashing

- The figure shows a simple example of a static hash organization, where:
 - Keys correspond to natural numbers
 - Primary area is made up of $P = 5$ buckets with size $C = 5$
 - The hash function is: $H(k_i) = k_i \% 5$
 - Overflows are managed by allocating, for each primary bucket, one or more overflow bucket, with size $C_{ov} = 5$, linked in a list



Static hashing: cost analysis

- For a hash file with N records in buckets of size $C = C_{ov}$, whose primary area contains P buckets, under the hypothesis of a perfect distribution of records on the P addresses, it is:
 - every address is generated N/P times
 - every list contains $N/(P*C)$ buckets
- The search cost for a record is thus proportional to $N/(P*C)$
- E.g.: $N = 10^6$, $C = 10$, $P = 25000$
 - A (successful) search accesses (on average) 2 buckets
 - This equals the number of I/O operations, supposing that each bucket requires a single I/O read

Hash functions

- A hash function is a (surjective) transformation from the key space, K , to the address space, $\{0, \dots, P - 1\}$
- The hypothesis that an arbitrary subset of K would be transformed to the P different addresses in a perfectly homogeneous way is pure abstraction, not very useful for analyzing performance obtainable from different hash organizations
- The ideal case that should be used to compare a specific hash function H is the one with uniform distribution on the address space, where, for every subset of K , each of the P addresses has the same probability, $1/P$, of being generated

Hash functions: uniform distribution

- In the ideal case, the number of keys, X_j , assigned to the j -th bucket follows a binomial distribution

$$\Pr\{X_j = x_j\} = \binom{N}{x_j} \left(\frac{1}{P}\right)^{x_j} \left(1 - \frac{1}{P}\right)^{N-x_j}$$

with average value μ and variance σ^2 given as:

$$\mu = \frac{N}{P} \quad \sigma^2 = \frac{N}{P} \left(1 - \frac{1}{P}\right)$$

where neither μ nor σ^2 depend on the specific bucket

- For $P \gg 1$, the ratio $\sigma/\sqrt{\mu}$ is almost equal to 1

Quality of a hash function

- For “real” hash functions, performance varies depending on the specific set of key values
- E.g.: The function $H(k_i) = k_i \% P$ is a “good” function, but for the set of key values $\{0, P, 2P, 3P, \dots, N \cdot P\}$ would allocate all keys in the bucket with address 0
- Every hash function, when it is chosen independently from the specific set of key values, could lead to very bad performance, in the worst case
- In the “average” case, however, when arbitrary subsets of K and real data files are considered, we observe that different hash functions actually behave differently

Degeneracy

- An appropriate criterion for evaluating an hash function H , with respect to a particular set of key values, is the analysis of its degeneracy $\sigma/\sqrt{\mu}$, where:

$$\mu = \sum_{j=0}^{P-1} \frac{x_j}{P} = \frac{N}{P} \quad \sigma^2 = \sum_{j=0}^{P-1} \frac{(x_j - \mu)^2}{P}$$

are computed over all the P buckets and x_j is the number of records within the j -th bucket

- The lower the degeneracy, the better the behavior of the hash function

Hash functions: mid square

- The key is multiplied by itself
- A number of central digits equal to those of $P - 1$ is extracted
- The so-obtained value is normalized by P

$$145142^2 = 21066200164$$

- For example, if $P = 8000$, by normalizing we obtain the address:
 $\lfloor 6620 \times 0.8 \rfloor = 5296$

Hash functions: shifting

- The key is divided into parts, each made up of a number of digits equal to those of $P - 1$
- Parts are then summed and the result normalized
- E.g.: $P = 800$, $k = 14514387$
$$387 + 514 + 14 = 915$$
- By normalizing we obtain $\lfloor 915 \times 0.8 \rfloor = 732$

Hash functions: folding

- The key is divided as for shifting
- Parts are “folded” and summed, then the result is normalized
- E.g.: $P = 800$, $k = 14514387$

$$783 + 514 + 41 = 1338$$

- By normalizing we obtain :

$$1338 \% 800 = 538$$

$$\lfloor 538 \times 0.8 \rfloor = 430$$

Hash functions: division

- The numeric key is divided by a number Q and the address is obtained as the rest:

$$H(k) = k \% Q$$

- For the value of Q we have the following (empirical) advices:
 - Q is the highest prime number not higher than P
 - Q is not prime, not higher than P , with no prime factor lower than 20
 - If $Q < P$, we should have $P = Q$ in order to ensure the surjectivity of the hash function

Hash functions : division (example)

□	$P = 6997$		
□	$k = 172146$	$H(k) =$	4218
□	172147		4219
□	172148		4220
□	172149		4221
□
□	174924		6996
□	174925		0

Alphanumeric keys

- The management of alphanumeric strings requires a preliminary conversion step
- Probably the most common technique is to define:
 - An alphabet A , where strings' characters are drawn
 - A bijective function $ord()$, associating to each alphabet element an integer value in the range $[1, |A|]$
 - A conversion radix b
- A string $S = s_{n-1}, \dots, s_i, \dots, s_0$ is then converted into a numeric key

$$k(S) = \sum_{i=0}^{n-1} ord(s_i) \times b^i$$

Alphanumeric keys: example

- Given $A = \{a,b,\dots,z\}$, $ord()$ with values in $[1,26]$, and $b = 32$, the string “indice” produces the value

$$\begin{aligned} & k(\text{“indice”}) \\ &= 9 \times 32^5 + 14 \times 32^4 + 4 \times 32^3 + 9 \times 32^2 + 3 \times 32^1 + 5 \times 32^0 \\ &= 316810341 \end{aligned}$$

- Simpler techniques, not using a radix, like, for example

$$k(S) = \sum_{i=0}^{n-1} ord(s_i)$$

are usually worse, since they can generate the same numeric key with different strings, obtained as anagrams

Choosing the “right” radix

- With the “division” technique we can have bad performance if the radix b has **prime factors in common** with P
- E.g.: given $A = \{a,b,\dots,z\}$, $b = 32$, and $P = 512$
- The value $H(k(S))$ is determined by the last two characters only:
 - “folder”: ordinal values are 6, 15, 12, 4, 5, 18,
 $k(\text{“folder”})=217,452,722$, $H(217452722)=\mathbf{178}$
 - “primer”: ordinal values are 16, 18, 9, 13, 5, 18,
 $k(\text{“primer”})=556,053,682$, $H(556053682)=\mathbf{178}$

Choosing the “right” radix (cont.)

- In order to understand the root causes of this problem, we need to comprehend the properties of the % operator
- The simplest case to consider is the one where P is a multiple of b , that is $P = \alpha \times b$
- A value y exists so that $b^{y \% (\alpha \times b)} = 0$
- For the % operator the following property holds (useful for computing $k(S)$):

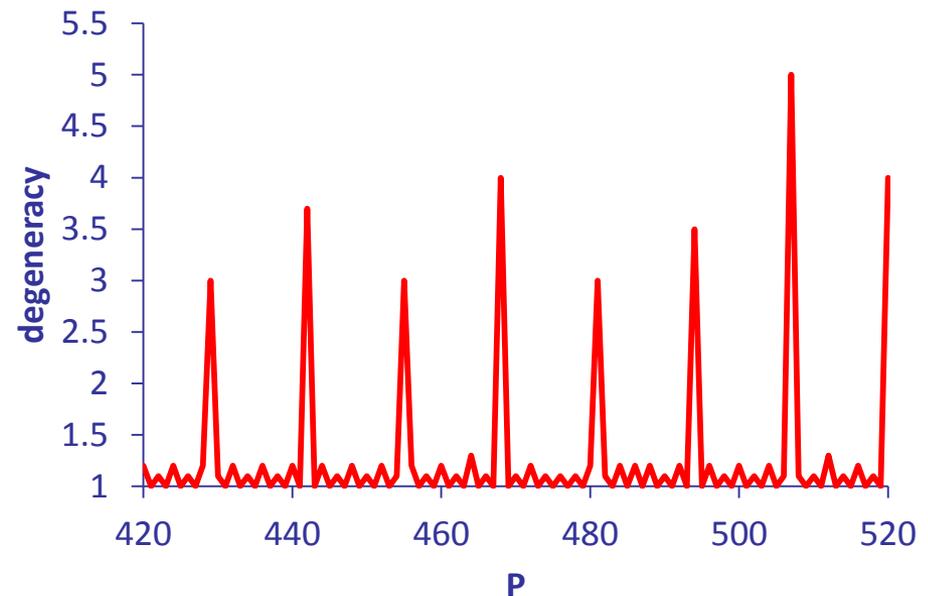
$$H(k(S)) = \left[\sum_{i=0}^{n-1} \text{ord}(s_i) \times b^i \right] \% (\alpha \times b) = \left(\sum_{i=0}^{n-1} \left[(\text{ord}(s_i) \times b^i) \% (\alpha \times b) \right] \right) \% (\alpha \times b)$$

characters s_{n-1} through s_y give a contribute = 0 to the value of $H(k(S))$ (in the example, $y = 2$), thus the “useful” string is only y characters long

- With radix = 26 we have problems whenever P has factors 13 and 2

Analysis of an example

- The following experiment was performed by Mullin in 1991 by using, as the key dataset, all 6-char English words, on the alphabet $A = \{a,b,\dots,z\}$, in the Unix spelling checker
- The graph shows the case $b = 26$ with P varying between 420 and 520
 - Distribution peaks are obtained for P values multiple of 13
 - In particular, the maximum is obtained for $P = 507 = 13 \times 13 \times 3$



Load factor

- Given an estimate of the number N of records, and given the bucket capacity C , choosing a load factor d determines the number P of buckets in the primary area
- We should consider that, when d decreases, the percentage of overflow records decreases as well
 - It is therefore not recommended using high values of the load factor
- Common values, representing a good trade-off between memory utilization and operational costs, are included in the range $[0.7,0.8]$

Bucket capacity

- Clearly, a capacity such that reading/writing a single bucket requires more than one I/O operation is not efficient
- On the other hand, it is convenient to have bucket capacity $C > 1$, due to the relationship existing between the value of C and the fraction of overflow records
- When increasing C , when the load factor d is constant, the fraction of overflow records decreases, under the hypothesis of an ideal hash function and a separate overflow area exists
 - Empirically, the result is valid also in the case of non-ideal hash functions

Optimal bucket capacity

- Since increasing the number of overflow record results in poor performance, it is recommended to have maximum bucket capacity C , under the constraints:
 - Reading/writing a bucket should result in a single I/O operation (sequential blocks)
 - Transferring a bucket with capacity C should cost less than transferring two buckets with capacity lower than C

Computing the average number of overflows

- The number of times that the j address is generated is a random variable with a binomial distribution, the average number of overflow in the j -th bucket is:

$$OV_j(C) = \sum_{x_j=C+1}^N (x_j - C) \times \Pr\{X_j = x_j\}$$

that does not depend on the specific bucket j

- For the sake of brevity, we will write $\Pr(x)$ in place of $\Pr\{X_j = x_j\}$

Overflow distribution

- The total number of overflows is obtained as:

$$OV(C) = \sum_{j=0}^{P-1} OV_j(C) = P \times \sum_{x_j=C+1}^N (x_j - C) \times \Pr(x)$$

- For high values of N and P , we can approximate the binomial distribution with the Poisson distribution :

$$\Pr(x) = \binom{N}{x} \left(\frac{1}{P}\right)^x \left(1 - \frac{1}{P}\right)^{N-x} \approx \left(\frac{N}{P}\right)^x \frac{e^{-\frac{N}{P}}}{x!}$$

- Since $P = N/(C \times d)$:

$$\Pr(x) \approx (C \times d)^x \frac{e^{-C \times d}}{x!}$$

Total number of overflows

- The total number of overflows is therefore obtained as :

$$OV(C) \approx P \times \sum_{x=C+1}^N (x-C) \times \frac{(C \times d)^x e^{-(C \times d)}}{x!}$$

- By substituting the variable $i = x - C$:

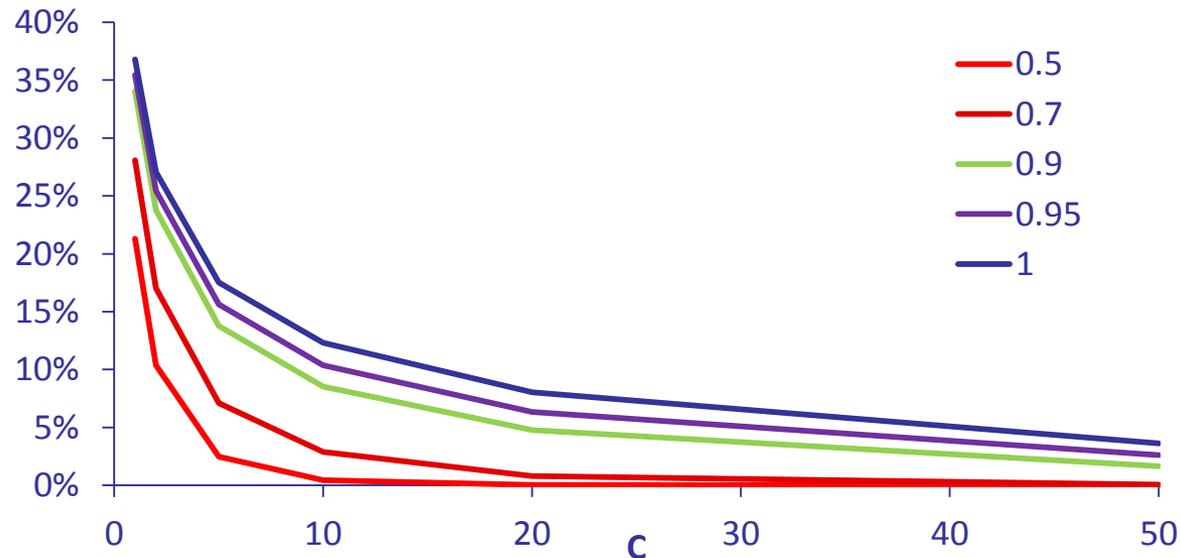
$$OV(C) \approx P \times \frac{(C \times d)^{C+1} e^{-(C \times d)}}{C!} \sum_{i=1}^{N-C} \frac{i \times C^{i-1} \times d^{i-1}}{(C+1)(C+2)\dots(C+i)}$$

- Again, since $P = N/(C \times d)$:

$$OV(C) \approx N \times \frac{(C \times d)^C e^{-(C \times d)}}{C!} \times f(C, d)$$

Example

C\d	0.5	0.7	0.9	0.95	1
1	0.213061	0.280836	0.340633	0.354464	0.367879
2	0.103638	0.170307	0.237853	0.254377	0.270669
5	0.02478	0.071143	0.137768	0.156336	0.17531
10	0.004437	0.028736	0.085344	0.103778	0.123244
20	0.000278	0.008014	0.047641	0.063497	0.080492
50	2.51E-07	0.000496	0.016561	0.026191	0.03622



Managing overflows

- Techniques used to manage overflows aim to reduce the number of bucket accesses required to retrieve the searched record
- Two main strategies:
 - **Chaining**
 - Pointers are used
 - Overflow area might be used
 - **Open addressing**
 - Do not use pointers
 - Buckets in primary area are used to store overflow records

Chaining in primary area

□ Separate lists

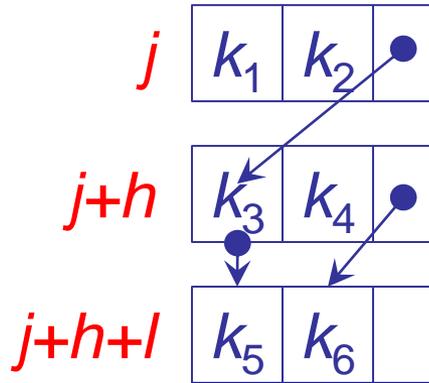
- If bucket j overflows, we insert the new record in the first non-full bucket following j
- Overflow records are linked in a list
- Non-overflow records should be linked, too

□ Coalesced chaining

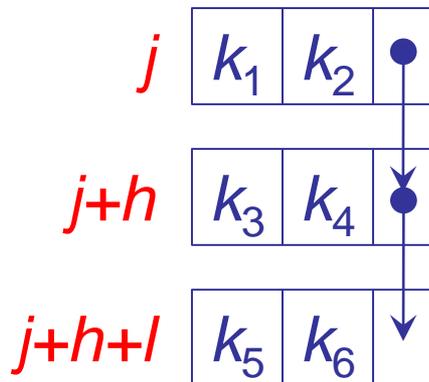
- A single pointer is used for every bucket (not for every record)
- Bucket lists could be merged (coalesced)
 - E.g.: j overflows in $j+h$ and $j+h$ is full, both overflow in $j+h+1$

Examples

- Separate lists



- Coalesced chaining



k	$H(k)$
k_1	j
k_2	j
k_3	j
k_4	$j+h$
k_5	j
k_6	$j+h$

- The coalesced chaining technique simplifies pointers management, but performance is worse

Chaining in overflow area

- As said, the capacity of overflow area buckets C_{ov} could differ from the one of primary area buckets C
- Usually, $C_{ov} < C$ to avoid wasting storage in case of reduced overflows
- For the same reason, we could use coalesced lists
- Clearly, in case a overflow bucket overflows, we would obtain an overflow bucket list

Open addressing

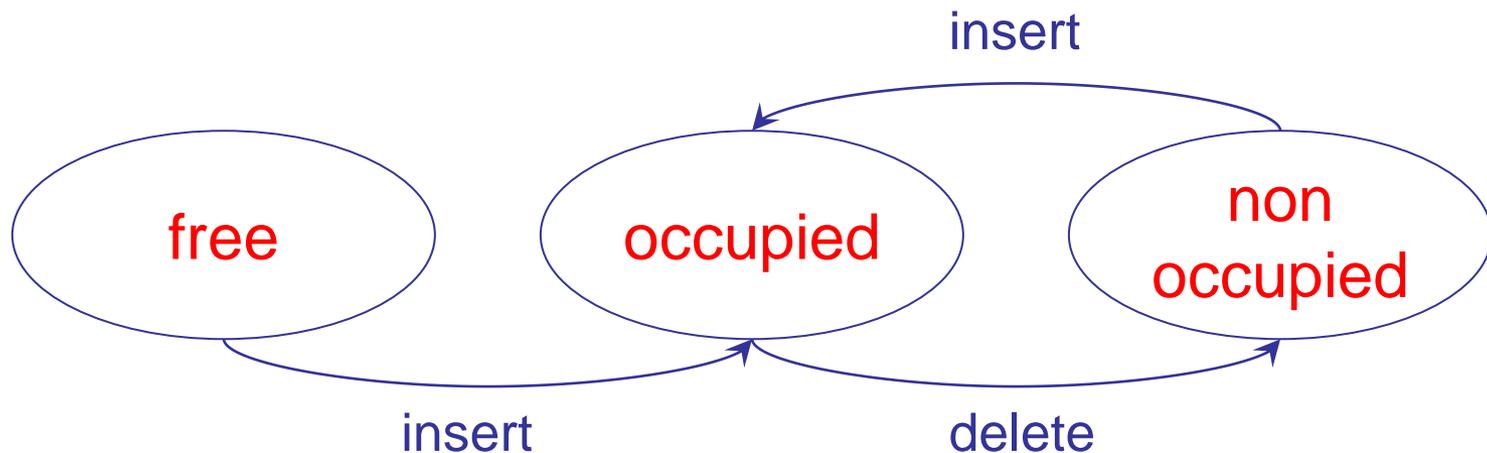
- In open addressing techniques, every key value k_i is associated to **a sequence of addresses** $H_0(k_i), H_1(k_i), \dots, H_l(k_i)$, with $H_0(k_i) = H(k_i)$
- When k_i is inserted, we test all addresses $H_0(k_i), H_1(k_i), \dots, H_l(k_i)$ until the address of a non-full bucket is found

Open addressing: searching

- To search for k_i we should look into all buckets with address $H_0(k_i), H_1(k_i), \dots, H_l(k_i)$ until
 - Either we find k_i (successful search)
 - Or we find a non-full bucket (unsuccessful search)
- With open addressing techniques, we should be very careful **how we delete records**
 - If a record of a full record is deleted, the bucket becomes non-full, thus it “stops” the search

Open addressing: deleting records

- The location of a deleted record is thus marked as “**non occupied**”, and it can be “occupied” (re-used) when new records are inserted
- A bucket is full if and only if all its locations are “occupied”



Linear probing

- At each step the address is increased by a **constant value s**
 - $H_j(k_i) = (H_{j-1}(k_i) + s) \% P$
 - Thus: $H_j(k_i) = (H(k_i) + s \times j) \% P$
- To make sure that all addresses can be generated, we need to ensure that s and P have no common factors
 - Otherwise, only $P/\text{MCD}(P,s)$ can be generated
- E.g.: $P=10, s=4, \text{MCD}(10,4)=2$
 - Addresses for $k_i = 3$ are $3, 7, 1, 5, 9, 3, \dots$

Primary clustering

- If bucket j overflows, it is likely that bucket $j+s$ overflows, thus that bucket $j+2s$ overflows...
- There is a **clustering of records** in some of the pages
 - E.g.: $P=31$, $s=3$, the following addresses are generated:
 - 1234 → 25, 28, 0, 3, 6, 9, 12, ...
 - 245 → 28, 0, 3, 6, 9, 12, 15, ...
- The problem is due to the linearity of the probing step s

Quadratic probing

- At each step the address is increased by a **linear value** $a + b(2j - 1)$
 - $H_j(k_i) = (H_{j-1}(k_i) + a + b(2j - 1)) \% P$
 - Thus: $H_j(k_i) = (H(k_i) + a \times j + b \times j^2) \% P$
 - E.g.: $P=31, a=3, b=5$, the following addresses are generated:
 - 1234 → 25, 2, 20, 17, 24, 6, ...
 - 245 → 28, 5, 23, 20, 27, 13, ...
 - Thus lists are not coalescing (e.g., see 20)
- The problem of **secondary clustering** remains, due to keys conflicting for the first address

Double hashing

- We try to avoid the secondary clustering problem by using **two hash functions**, H' and H''
- Address sequences are given by :
 - $H_0(k_i) = H'(k_i)$
 - $H_j(k_i) = (H_{j-1}(k_i) + H''(k_i)) \% P$ (if $j > 0$)
- Two keys generate the same sequence if and only if they collide on both H' and H''

Double hashing: comments

- The double hashing techniques has the collateral effect of producing a high variability of generated addresses, depending on the value of $H''(k_j)$
- Considering the actual allocation of buckets in secondary storage this could this can significantly weigh down I/O operations
 - Subsequent buckets are “far away” from each other, increasing latency
- Double hashing approximates well enough the ideal case of “uniform hash” (random probing), where at the j -th step each address has the same probability of being generated

Comparing different techniques

- Techniques not using overflow area have a storage utilization u equal to d
- With overflow buckets, we obtain:

$$u = \frac{N}{P \times C + P_{ov} \times C_{ov}}$$

Search performance

- Coalesced chaining
 - Successful search: $E \approx 1 + \frac{1}{8d} (e^{2d} - 1 - 2d) + \frac{d}{4}$
 - Unsuccessful search: $A \approx 1 + \frac{1}{4} (e^{2d} - 1 - 2d)$

- Separate chaining with overflow area
 - Successful search: $E \approx 1 + d/2$
 - Unsuccessful search: $A \approx e^{-d} + d$

Performance: open addressing

- Random probing: unsuccessful search
 - With buckets with capacity 1, search cost equals the number of buckets needed to insert a new record
 - Probability of failure at each bucket equals d
 - Probability of having found $r-1$ occupied buckets $\Pr\{\text{cost}=r\}$ thus equals $(1-d) d^{r-1}$
 - Distribution is geometric with average value $d/(1-d)$
 - Average cost for unsuccessful search is thus:
 $A = d/(1-d)+1 = 1/(1-d)$

Performance: open addressing

- Random probing: successful search
 - Average number of accesses equals the average cost for inserting all records

$$E \approx \frac{1}{N} \sum_{i=0}^{N-1} \frac{1}{1-i/P} = \frac{P}{N} \sum_{i=0}^{N-1} \frac{1}{P-i}$$

- The sum can be rewritten as the difference of two armonic sums

$$E \approx \frac{P}{N} \left(\sum_{i=1}^P \frac{1}{i} - \sum_{i=1}^{P-N} \frac{1}{i} \right)$$

Performance: open addressing

- Random probing: successful search

- We have:
$$\sum_{i=1}^N \frac{1}{i} = \log(n) + \text{cost} + O(n^{-1})$$

- Therefore:

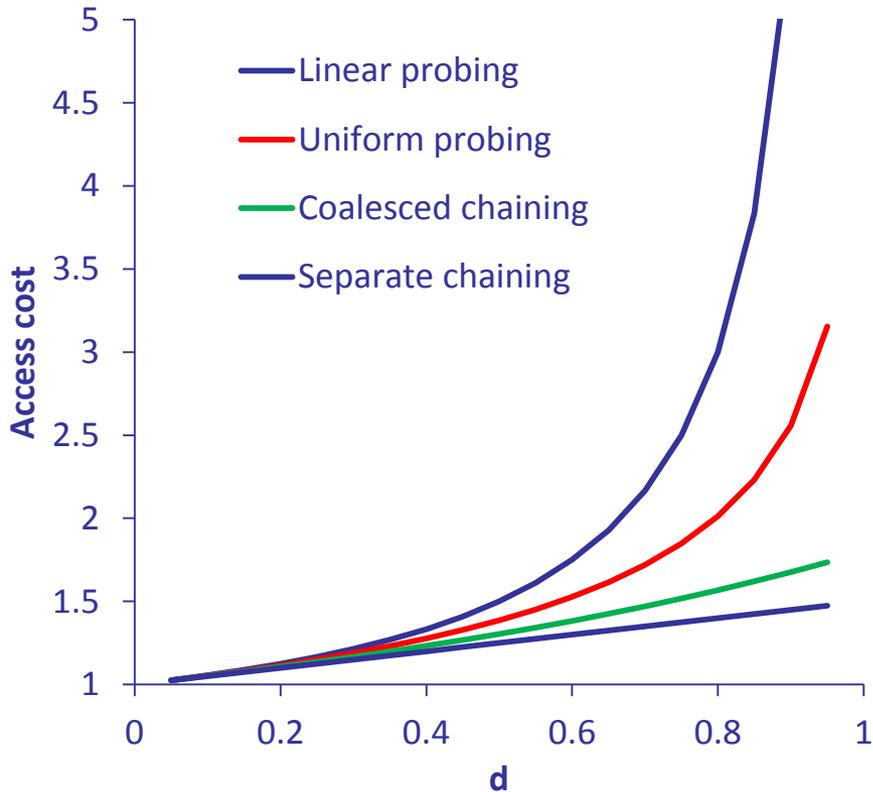
$$E \approx \frac{P}{N} \log\left(\frac{P}{P-N}\right) = \frac{P}{N} \log\left(\frac{1}{1-d}\right) = -\frac{P}{N} \log(1-d)$$

Performance: open addressing

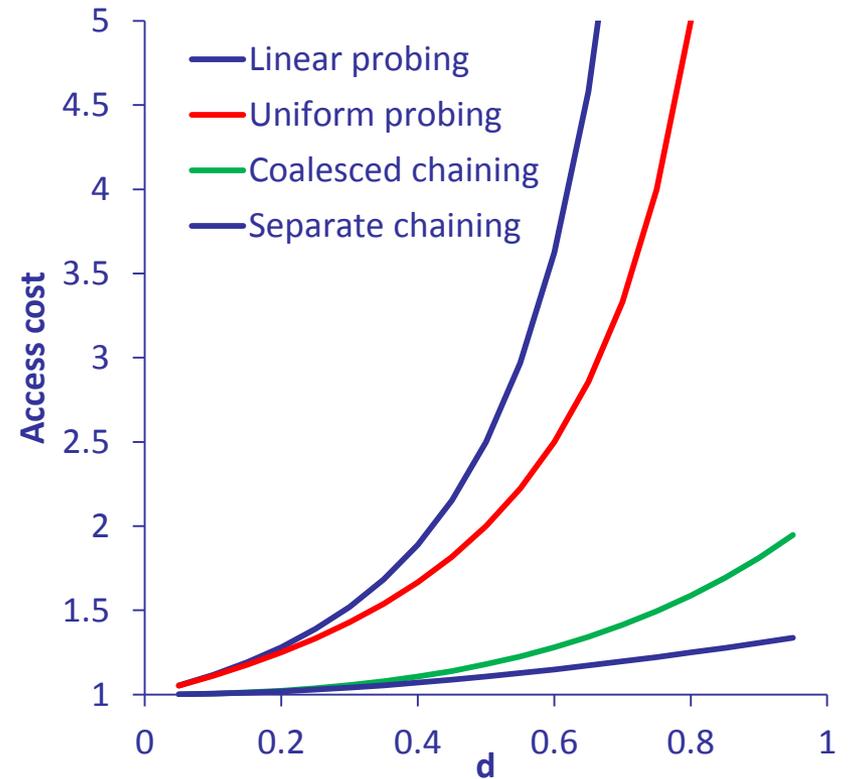
- Linear probing
 - Successful search: $E \approx \frac{1}{2} \left(1 + \frac{1}{1-d} \right)$
 - Unsuccessful search: $A \approx \frac{1}{2} \left(1 + \frac{1}{(1-d)^2} \right)$
- If the bucket capacity C increases, performance of open addressing techniques improve, since:
 - $A(C) = 1 + (A(C=1) - 1)/C$
 - $E(C) = 1 + (E(C=1) - 1)/C$

Comparison

□ Successful search



□ Unsuccessful search



Problems for static organizations

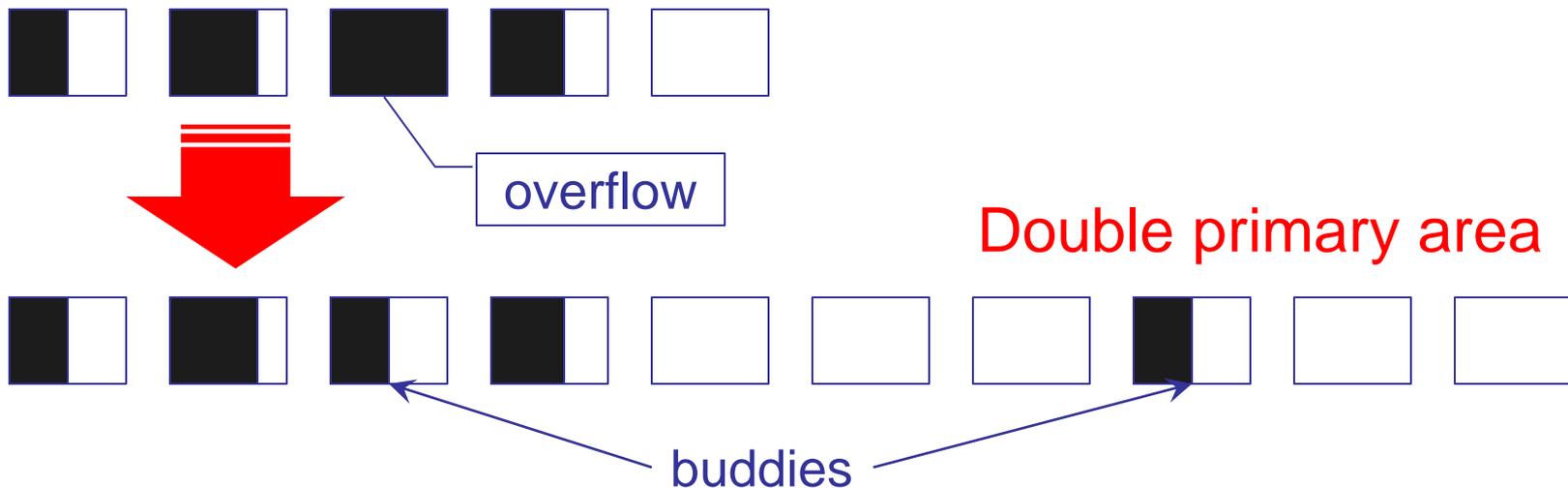
- Storage should be allocated during the initial design
 - If storage is overestimated, it is poorly used
 - If storage is underestimated, a high load factor follows, thus costs increase
- Moreover, if buckets overflow in primary area, we have the constraint $d \leq 1$

Dynamic Hashing

- These techniques adapt the allocation of primary area according to the actual number of records
- They are divided in
 - **With directory**
 - Virtual hashing
 - Dynamic hashing
 - Extendible hashing
 - **Without directory**
 - Linear hashing
 - Spiral hashing

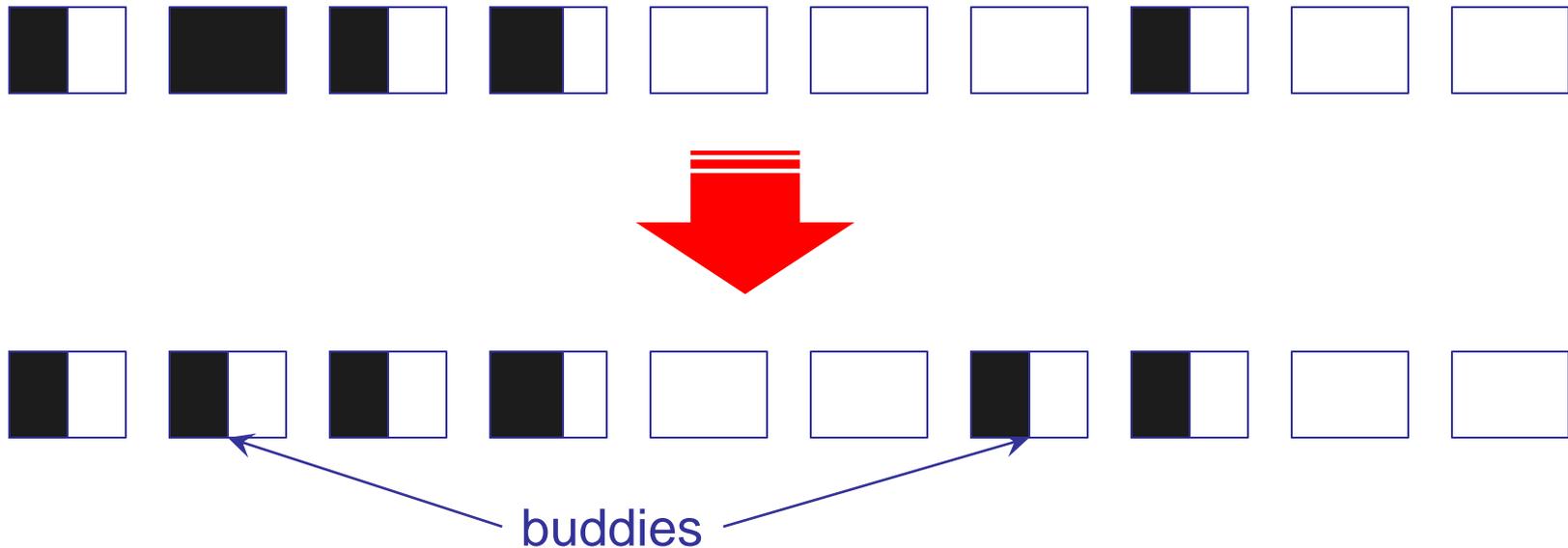
Virtual hashing

- The basic idea of Virtual hashing (Litwin '78) is **doubling the primary area** whenever a bucket overflows
 - Records are distributed between the overflowed bucket and its "**buddy**", by using a new hash function
 - In practice, we "**split**" the overflowed bucket



Virtual hashing: using buddies

- If, afterwards, another bucket in the original primary area overflows, and its buddy is still unused, its record records are distributed between the bucket and its buddy



Virtual hashing: directory

- Since, in a given moment, only some of the buddies are actually used, an auxiliary structure is needed to determine which hash function should be used
- In practice, a bit vector V is used, where $V[i]=1$ if and only if the corresponding bucket is used

Virtual hashing: initializing primary area

- P_0 buckets with capacity C are allocated
- An hash function H_0 with values in $[0, P_0 - 1]$ is used
- We set $l=0$ (number of doublings)
- A binary vector V is created, with length P_0 , and all its elements are set to 1
- After l doublings, the primary area contains $P = 2^l P_0$ buckets, and the buddy of the j -th bucket ($0 \leq j \leq 2^{l-1}P_0 - 1$) is the bucket with address $j + 2^{l-1}P_0$

Virtual hashing: splitting bucket j

- If $l=0$, or $l>0$ but the buddy of j is already used or does not exist ($l>0$, but $V[j+2^{l-1}P_0]=1$ or $j \geq 2^{l-1}P_0 - 1$)
 - $l++$, the primary area and the vector V are doubled
 - New elements of V equal 0, except for $V[j+2^{l-1}P_0]=1$
 - The new hash function H_l is used, with values in $[0, 2^l P_0 - 1]$
 - Keys of bucket j are re-distributed using H_l
- If the buddy of j exists and is not used ($V[j+2^{l-1}P_0]=0$)
 - $V[j+2^{l-1}P_0]=1$
 - Keys of bucket j are re-distributed using H_l

Virtual hashing: example (i)

- $P_0=7, C=3$

112 1176	512 3270 841	723	6851	7830 1075 6647	2840 2665 2385	286
-------------	--------------------	-----	------	----------------------	----------------------	-----

V
1
1
1
1
1
1
1

- We use the family of hash functions
 $H_i(k) = k \% (2^i P_0)$
- We insert the key $k=3820$
 $H_0(3820) = 5$
- Bucket 5 overflows

Virtual hashing: example (ii)

- $P_1=14$

112 1176	512 3270 841	723	6851	7830 1075 6647	2665 2385	286
					3820 2840	

V
1
1
1
1
1
1
1
0
0
0
0
0
1
0

- The primary area and vector V are doubled
- Keys of bucket 5 are re-distributed with its buddy 12, by using the hash function $H_1(k) = k \% 14$

Virtual hashing: example (iii)

□ $P_1=14$

112 1176	512 3270 841	723	6851	7830	2665 2385	286
				1075 6647 3343	3820 2840	

V
1
1
1
1
1
1
1
0
0
0
0
1
1
0

- If we have to insert 3343, we have $H_1(3343)=11$
- Since $V[11]=0$, we should apply $H_0(3343)=4$
- Bucket 4 overflows and is splitted
 - In this case without doubling the primary area
 - $V[11]=1$ and keys are re-distributed
- And if we should insert 5485?

Virtual hashing: search (and insert)

- In order to search for a key value we need to know which hash function was used when it was inserted
 - Vector V is what we need
- The following method returns the address of the bucket where the searched key could be (or where the new key should be inserted)

Address(k, l)

Input: key k , level l

Output: Address of bucket at level l where k can be found

if ($l < 0$) the key does not exist

else if $V[H_l(k)] = 1$ return $H_l(k)$

else return **Address(k, l-1)**

Virtual hashing: hash functions

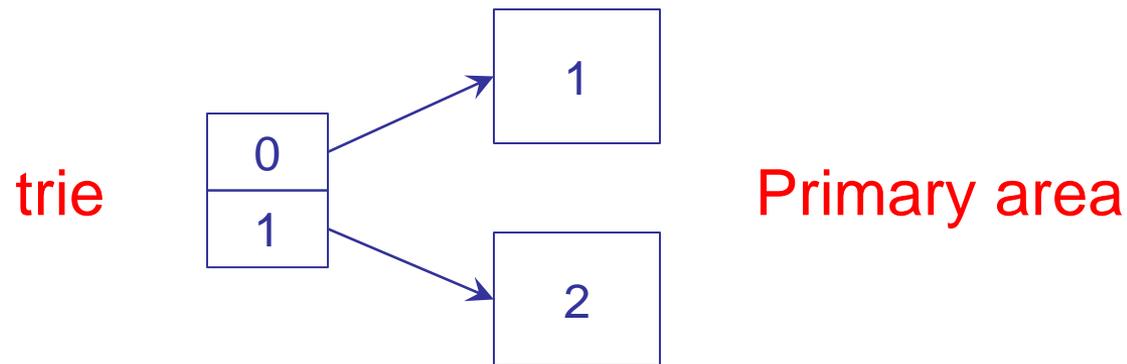
- Virtual hashing requires a **series of hash functions** $H_0, H_1, \dots, H_l, \dots$ satisfying the following properties:
 - **Range condition:** function H_l should have values in $[0, 2^l P_0 - 1]$
 - **Split condition:** for each $l > 0$, for each k , and for each value of $H_l(k)$, the following should hold:
$$H_l(k) = H_{l-1}(k) \quad \text{otherwise } H_l(k) = H_{l-1}(k) + 2^{l-1} P_0$$
that is, when a bucket is split its keys can only generate either the bucket address, or the one of its buddy
- The family of functions $H_l(k) = k \% (2^l P_0)$ satisfies both conditions

Dynamic hashing

- Dynamic hashing (Larson '78) avoids doubling the whole primary area (which limits the storage utilization and slows down the organization during doubling), by using an **auxiliary structure (directory) organized as a binary trie**
- The basic idea (used also by extendible hashing) is to use a hash function that, given the key k , returns not an address, rather a **binary pseudo-key** $H(k) = b_0, b_1, b_2, \dots$
- The ideal case is the one where the set of pseudo-keys is such that $\Pr\{b_i=1\}=1/2$, that is, a balanced partition is obtained for each considered index
- A simple technique to generate pseudo-keys is to use k as the seed of a pseudo-random binary generator

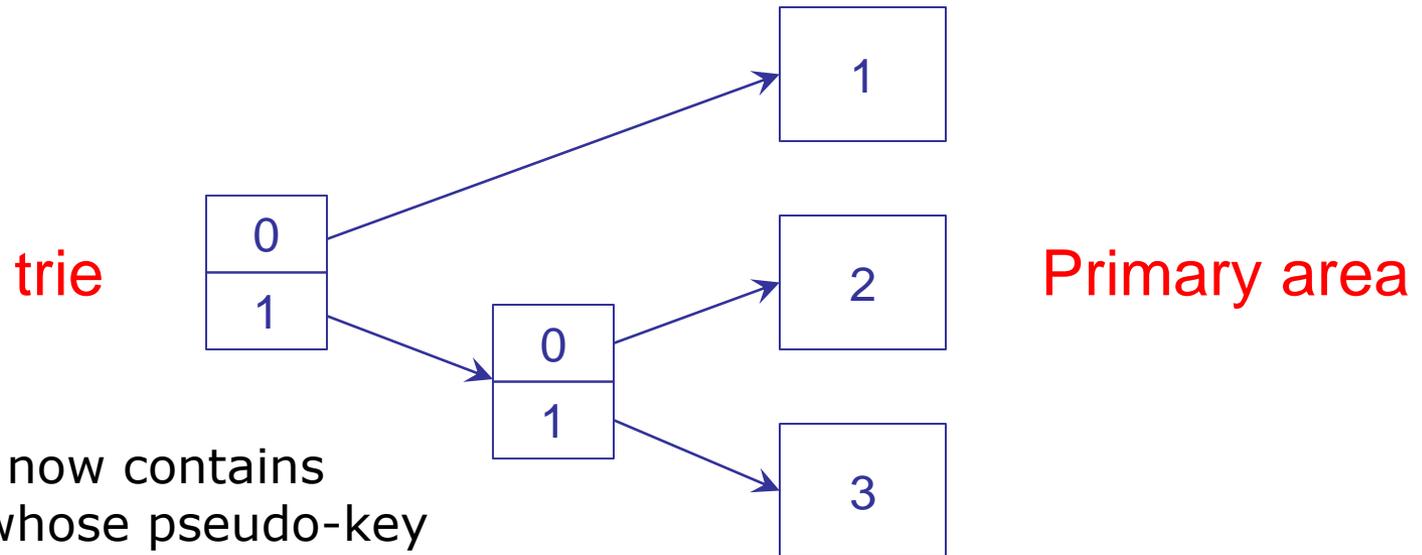
Dynamic hashing: using the trie

- The trie is used to drive the search, and its leaves contain addresses of primary area buckets
- To search for (or to insert) a key **we descend the trie corresponding to the generated pseudo-key, until a leaf is reached**
 - **Example:** bucket **1** contains all key values whose pseudo-key is **0...** and bucket **2** all those with pseudo-key **1...**



Dynamic hashing: overflow

- Expansion of primary area is performed by adding a single bucket at the time, re-distributing records between the overflowed bucket and its buddy, and adding a node to the trie
 - Example:** we have to split bucket 2



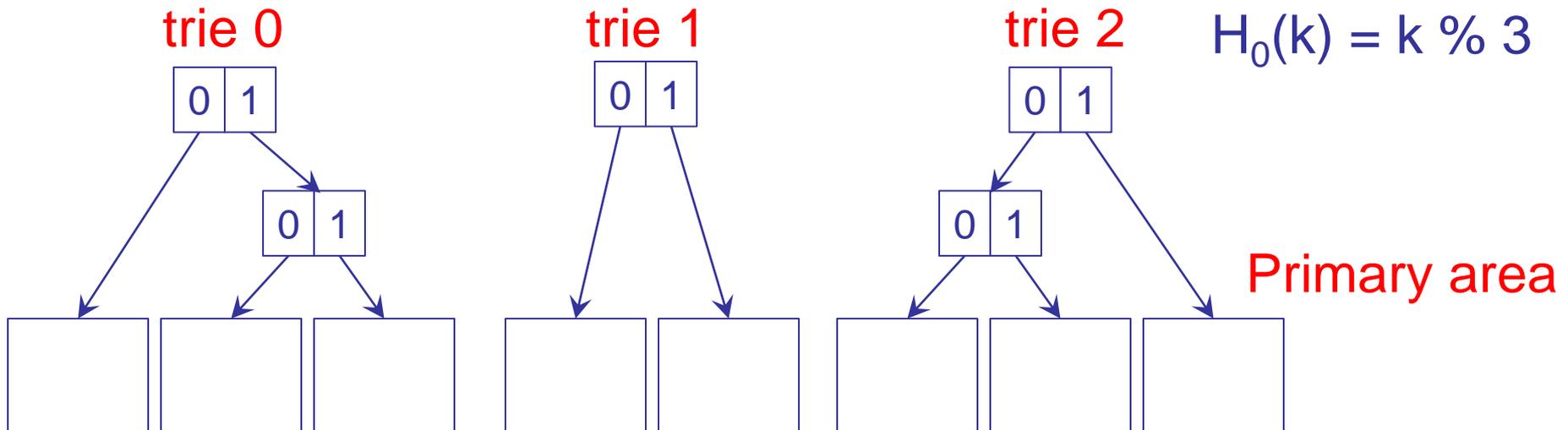
- Bucket 2 now contains all keys whose pseudo-key is 10..., and bucket 3 those with 11...

Dynamic hashing: performance

- If the trie is in main memory, a single access is sufficient to retrieve a record
- If the trie is not in main memory, performance depend on the trie balancing, in terms of number of trie nodes to retrieve
- In the worst case, performance is not so good
 - Depending on the pseudo-key set, inserting a new record could lead to multiple splits
- After deleting a record in a bucket, if the number of records in that bucket is lower than the capacity C , buckets are merged, and a leaf is deleted from the trie
- Average storage utilization is about 70%

Dynamic hashing: variant

- We initially allocate P buckets and use any static hash function H_0
- When a overflow occurs, we generate P tries, whose root nodes are addressed by H_0

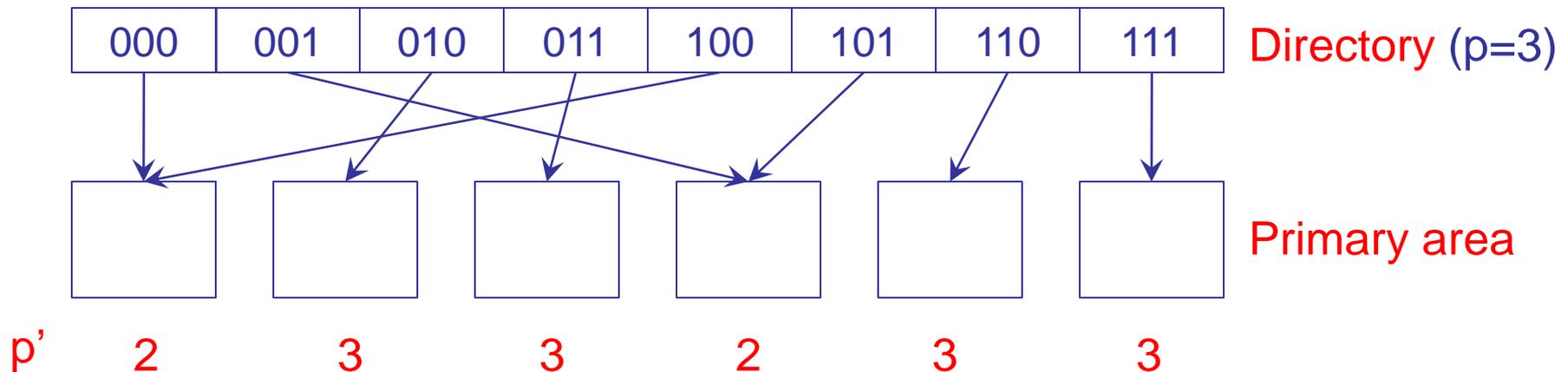


Extendible hashing

- It is quite similar to dynamic hashing, the main difference being the directory management (Fagin et al., '79)
- No more than two I/O accesses are guaranteed
- Directory consists in 2^p cells with addresses $[0, 2^p - 1]$
 - $p \geq 0$ is called **directory depth**
- A hash function associates to each key a **binary pseudo-key**, $H(k) = \dots b_2 b_1 b_0$, for which only the p **least-significant bits** are used to directly access one of the 2^p directory cells, each containing a pointer to a bucket

Extendible hashing: bucket depth

- Every bucket has a **local depth** $p' \leq p$ (value stored in the bucket), indicating the actual number of bits used to allocate keys within the bucket itself
- **Example:** the bucket containing key 258 (...001) has $p'=2$, thus it contains keys with pseudo-key both like ...001 and ...101



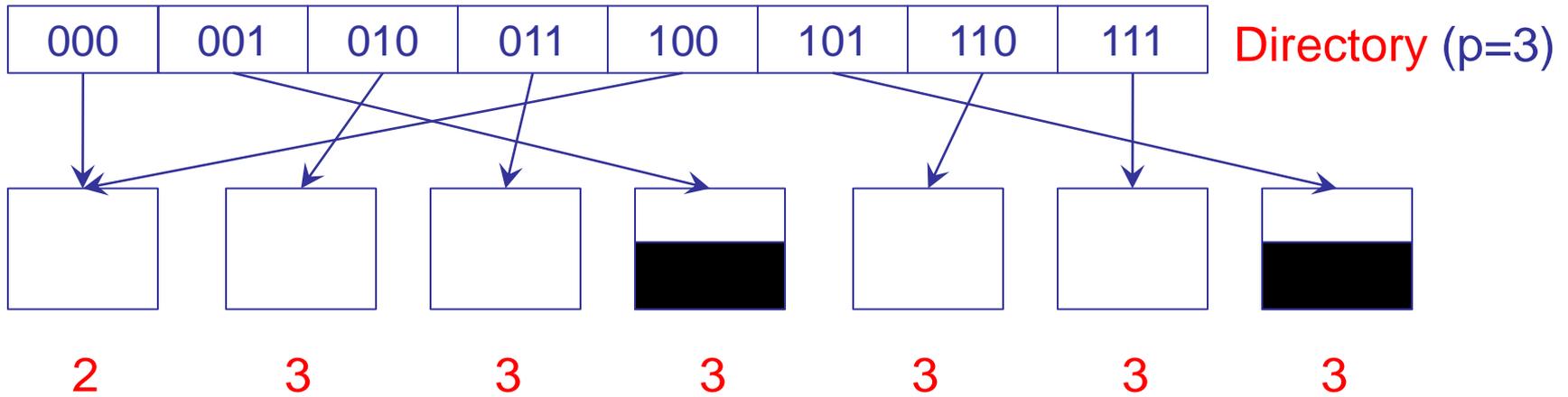
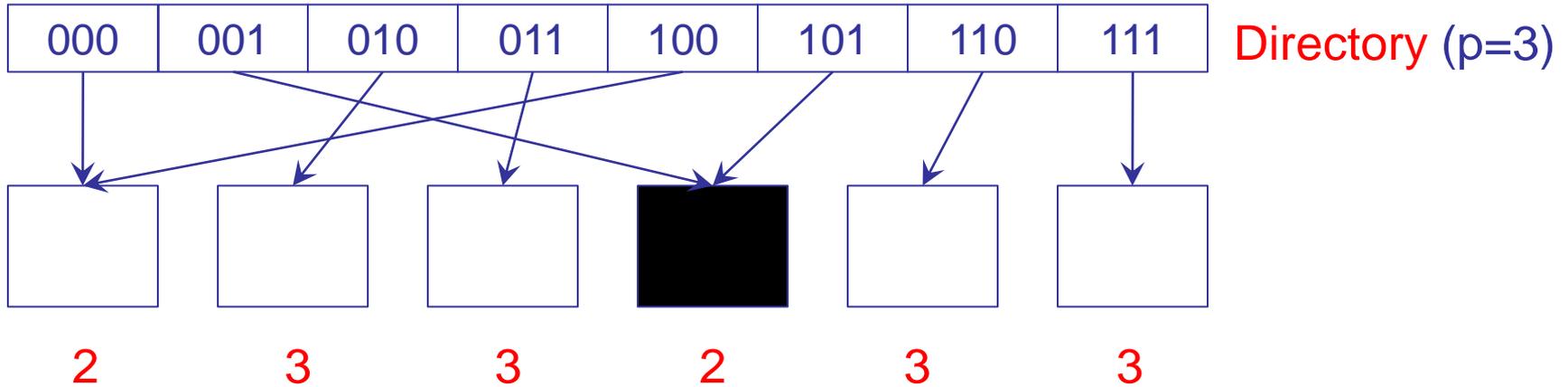
Extendible hashing: splitting a bucket

- Initially, as single bucket exists with $p' = 0$ and $p = 0$
- If a bucket has to be split with local depth p' , 2 cases are possible:
 - $p' < p$
 - $p' = p$

Extendible hashing: splitting a bucket with $p' < p$

- A new bucket is **allocated** and keys are **distributed** between the two buckets using the $(p'+1)$ -th bit of pseudo-keys
 - For both buckets we set local depth at $p'+1$
- Since $p' < p$, at least a cell exists where the address of the new bucket can be arranged
 - We update the pointer of the cell(s)

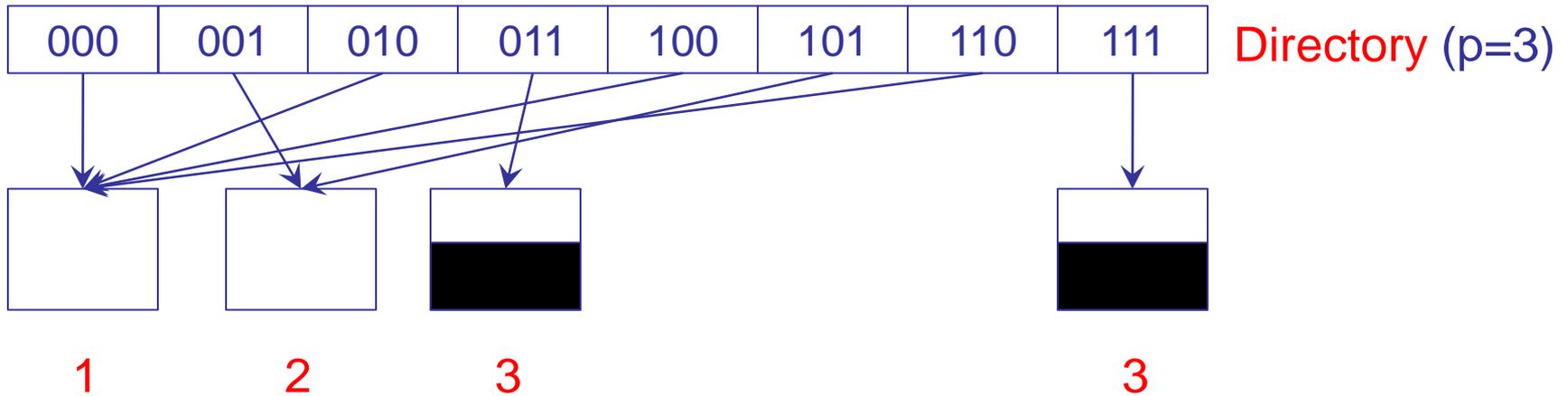
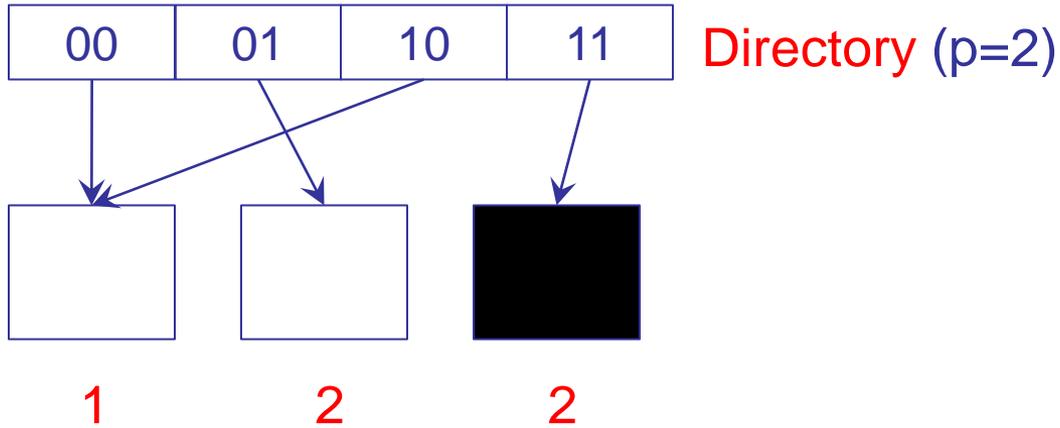
Example: splitting a un bucket with $p' < p$



Extendible hashing: splitting a bucket with $p' = p$

- Since $p' = p$, no cell exists to contain the address of the new bucket (obtained from splitting the overflowed bucket)
- We **double** the directory, increasing p by **1**
- We copy pointers in the new cells (second half of the directory)
- We perform split as when $p' < p$

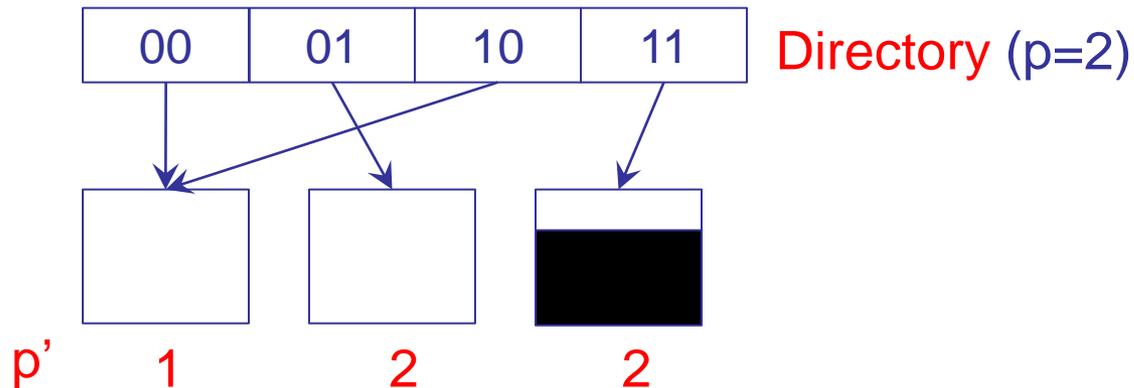
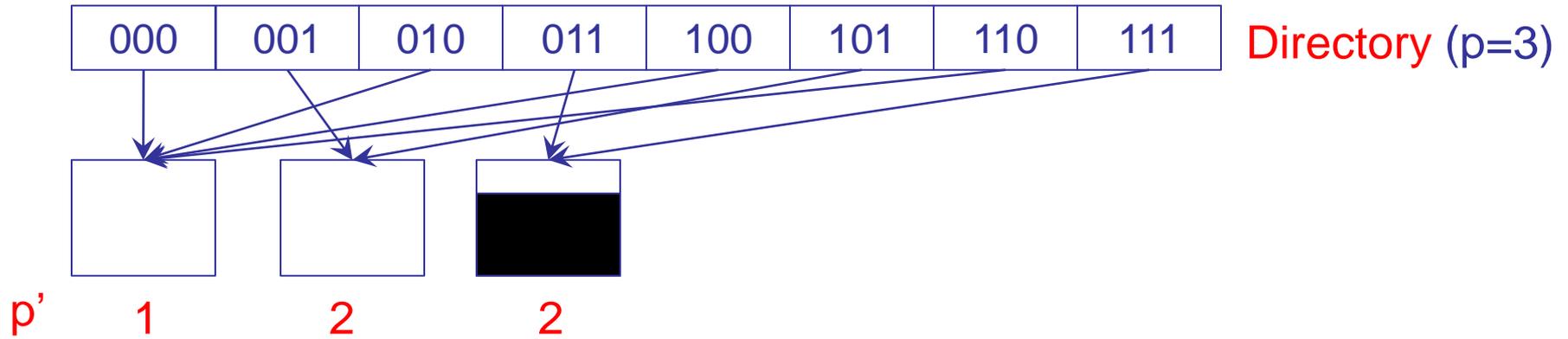
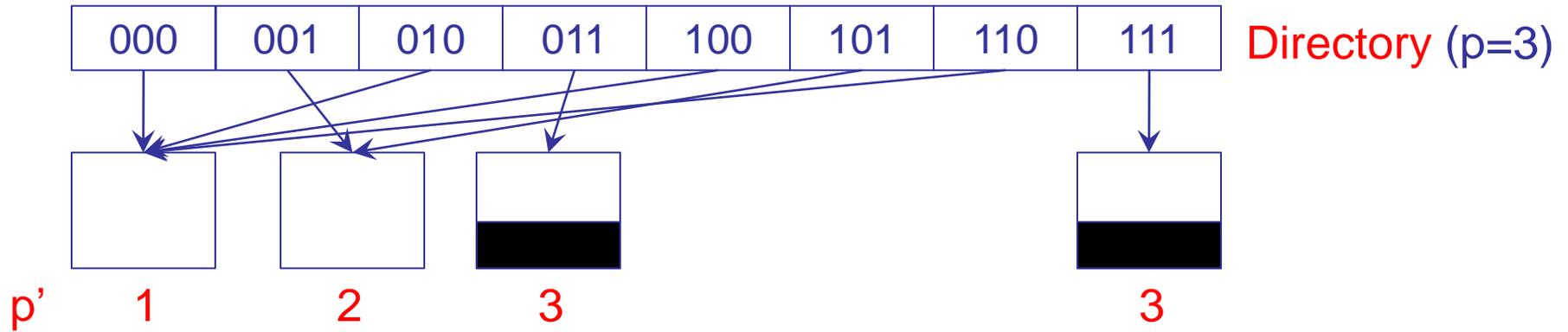
Example: splitting a un bucket with $p'=p$



Extendible hashing: deletion

- If a record is deleted in a bucket with depth p' , and the number of records in the bucket and in its buddy is not higher than the capacity C , buckets **are merged**
 - The local depth of the resulting bucket is $p'-1$
- If the only buckets at depth $p' = p$ are merged, it is possible to **contract the directory**, halving it
 - Since testing that no bucket exists with local depth p requires, in the worst case, to read all the buckets, it is appropriate using a local depth table which, for every value $p' \leq p$, stores the number, $P(p')$, of buckets having depth p'

Example: directory halving



Extendible hashing: comments

- Every doubling affects the whole directory
 - Problems in case of concurrent operations
- A solution exploits a multi-level directory
 - The index is no longer binary

Linear hashing

- **Linear expansion** technique

“We do not split the overflowed bucket,
rather we split another bucket,
chosen according to a specific criterion”

- No directory is required
 - We have to manage overflows (in primary or separate area)
 - Primary area grows “linearly”
- In the case of linear hashing, the bucket to be split is the one **following** the last split bucket

Linear hashing: management of primary area

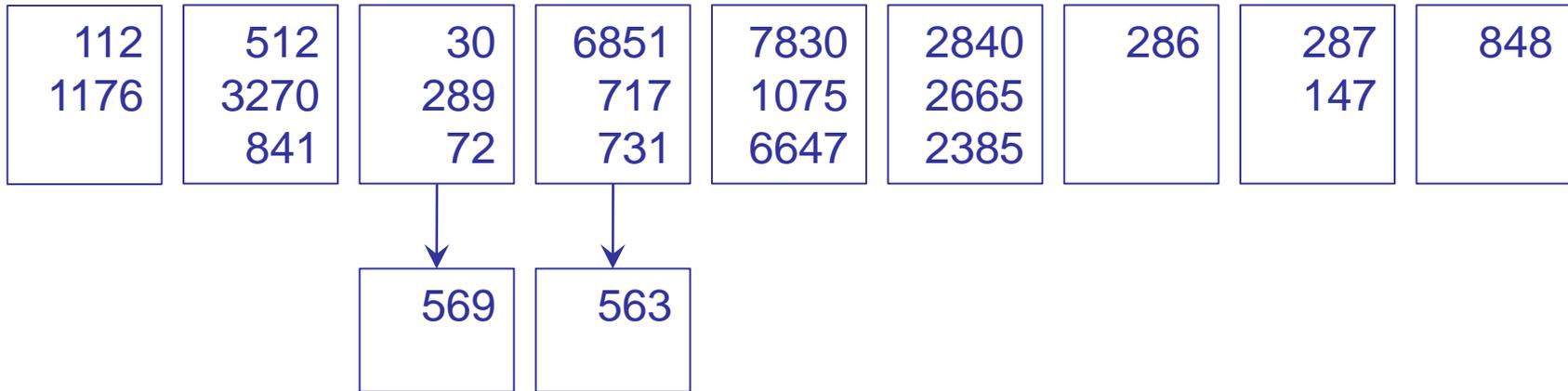
- Initially, P_0 buckets are allocated and we use the hash function $H_0(k) = k \% P_0$
- We keep a pointer (split pointer, SP) to the next bucket to be split
 - Initially, $SP = 0$
- In case of an overflow
 - We add a new bucket with address $P_0 + SP$
 - We re-distribute records in SP (including those that may be in overflow area) by using the new hash function $H_1(k) = k \% (2P_0)$
 - We increase SP by 1

Linear hashing: doubling the primary area

- After P_0 overflows, we have a complete expansion of primary area, since the number of buckets is now $2 P_0$
- We prepare for a new expansion, setting $SP = 0$, $H_0(k) = H_1(k)$, and $H_1(k) = k \% (2^2 P_0)$
- During the j -th expansion we use the hash functions $H_0(k) = k \% (2^{j-1} P_0)$ and $H_1(k) = k \% (2^j P_0)$
- The address of the home bucket for a key is $H_0(k)$ if $H_0(k) \geq SP$, $H_1(k)$ otherwise

Linear hashing: example(i)

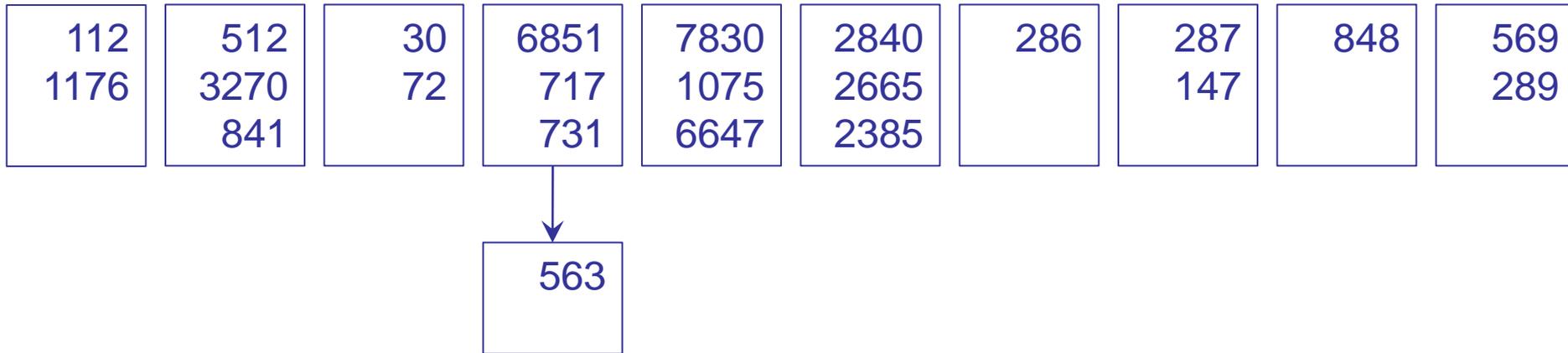
- $P_0=7, C=3, C_{ov}=2, SP=2$



- The overflow of buckets 2 e 3 caused the split of buckets 0 and 1
- Inserting the key 3820 ($H_0(3820) = 5$) bucket 5 overflows
 - We split bucket 2

Linear hashing: example (ii)

- $P_0=7, C=3, C_{ov}=2, SP=3$



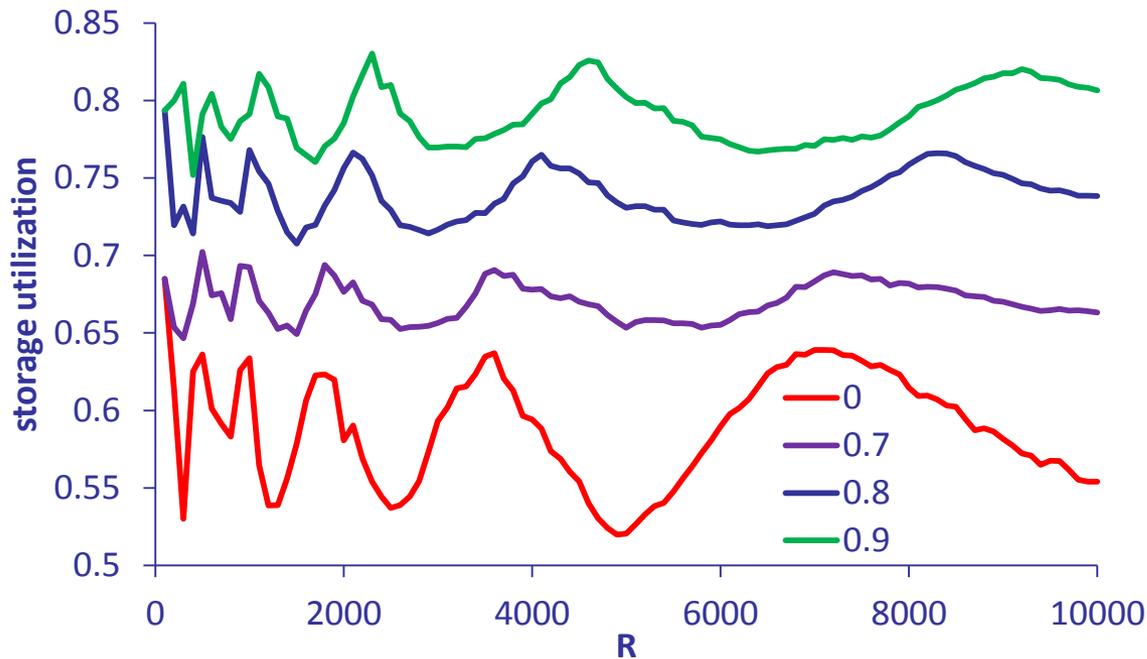
- Allocation of bucket 9 caused the deletion of an overflow bucket
 - The next overflow would split bucket 3
- After 7 split the primary area is doubled and we restart from $SP=0$

Linear hashing: pros and cons

- + The lack of a directory and the management of splits make implementing the structure very easy
- + Management of primary area (expansion and contraction) is immediate, since buckets are always added (and removed) at the end
- Storage utilization is rather low (variable between 0.5 and 0.7)
- Management of overflow area introduces problems similar to the ones of a static organization
- Overflow chains of high-address, not yet split, buckets can be very long

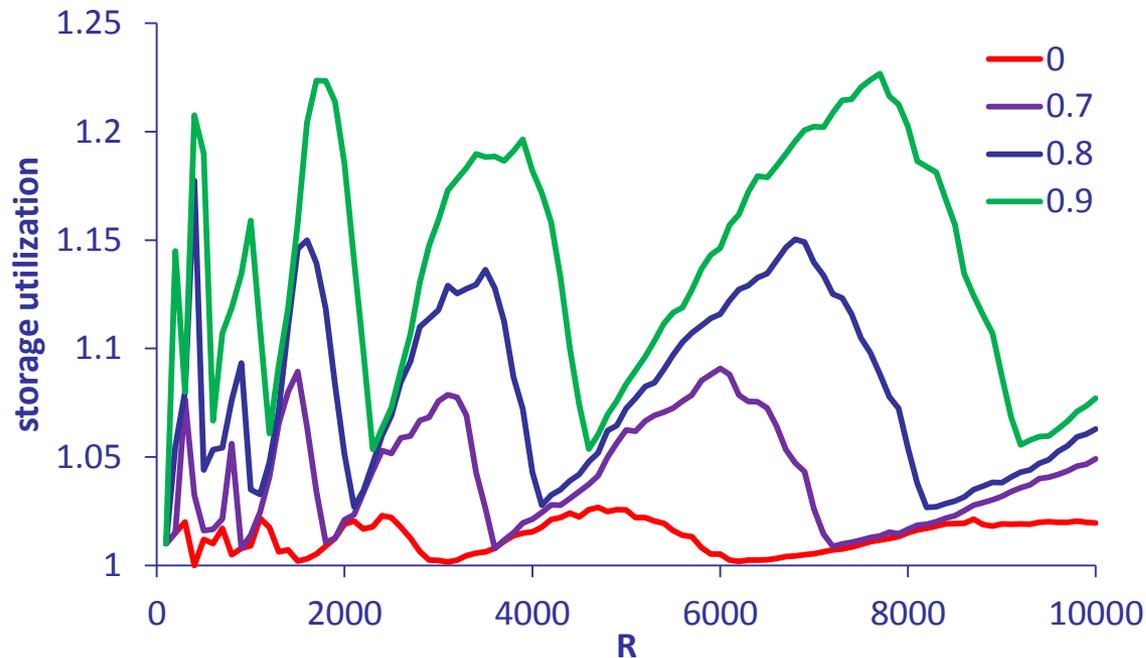
Linear hashing: storage utilization

- A variant, improving storage utilization, does not split a bucket if this does not reach a minimum utilization u_{\min}



Linear hashing: performance

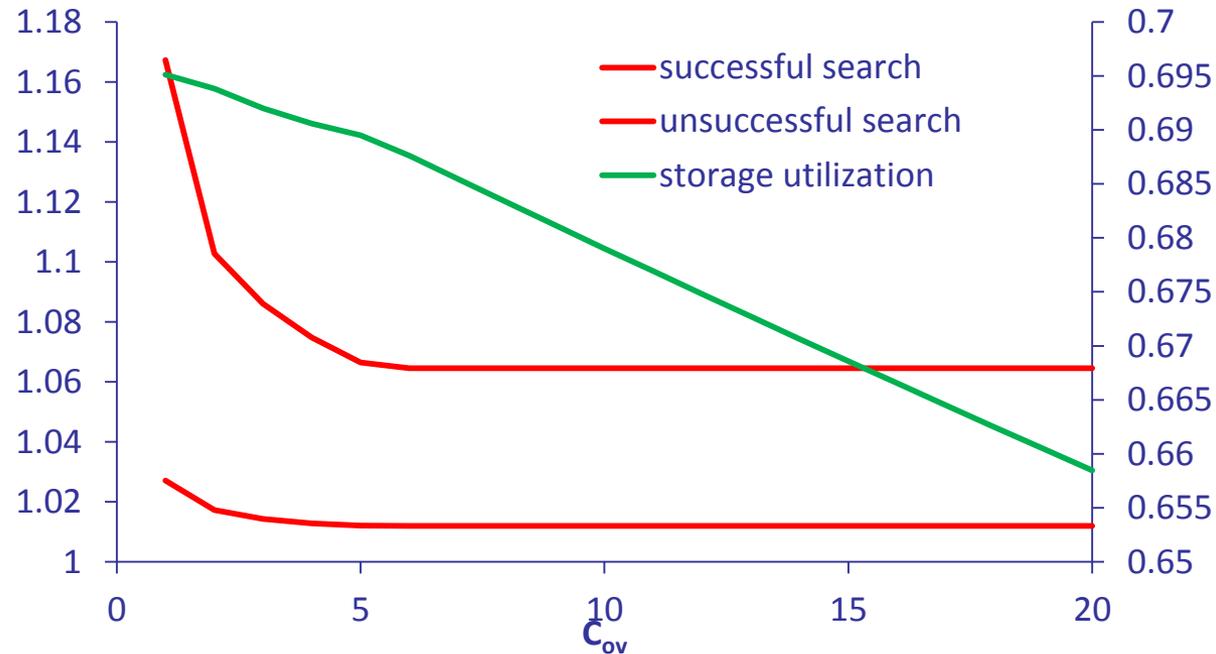
- When storage utilization increases, search costs increase as well, since overflow records increase
- Successful search costs



Minima of access costs correspond to maxima of storage utilization

Linear hashing: overflow capacity

- Increasing C_{ov} reduces the length of overflow chains
 - Search cost reduces as well
- Beyond a given point, we only waste storage



Recursive linear hashing

- Principal characteristic of Recursive linear hashing (Ramamohanarao & Sacks-Davis, '84) is the management of overflow area, **dynamically organized** using Linear hashing
- Different levels of dynamic hash files are created (on average, no more than 3), with the file at level h ($h=0$ primary area) storing its overflow records into the file at level $h+1$
- At level h we keep the split pointer SP_h

Recursive linear hashing: operations

- Insertion:
 - We compute the address $H^0(k)$
 - If the bucket overflows, we move to level 1 using function $H^1(k)$
 - If also the bucket at level L overflows, we add a new level
- Split:
 - When the j -th bucket at level h is split, records to distribute are those of j -th bucket and those in overflow, which are contained in buckets at levels $(h+1), \dots, L$

Recursive linear hashing: search

- Searching for a key could require a number of disk accesses equal to the number of levels

$h = 0;$

while ($h \leq L$)

if ($H^h_0(k) < SP_h$) **Address** = $H^h_1(k);$

else **Address** = $H^h_0(k);$

if **EXISTS**($k, \text{Address}, h$) **return** true;

else if **FULL**(**Address**, h) $h++;$

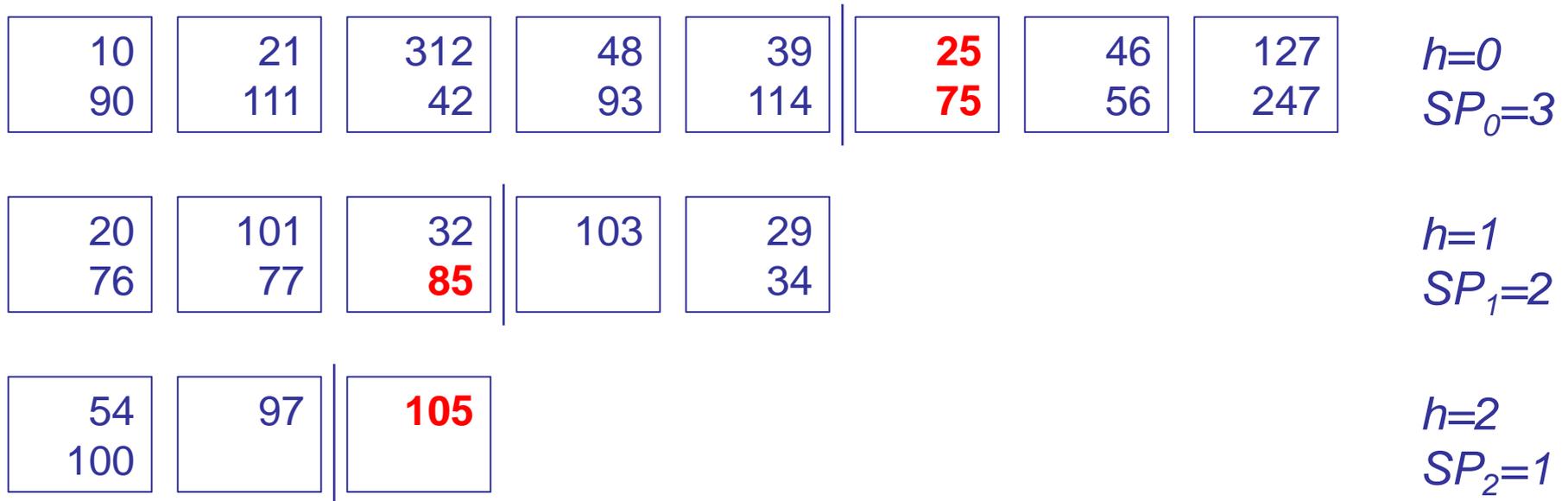
else return false;

Recursive linear hashing: addressing

- Overflows at level h are stored using as the new “key” the address of the bucket at level h itself
- Allocation is not performed using the key value, k , otherwise, when a bucket is split, we would not know where its overflow records are stored

Recursive linear hashing: example

- $P_0=5, P_1=3, P_2=2, C=2$



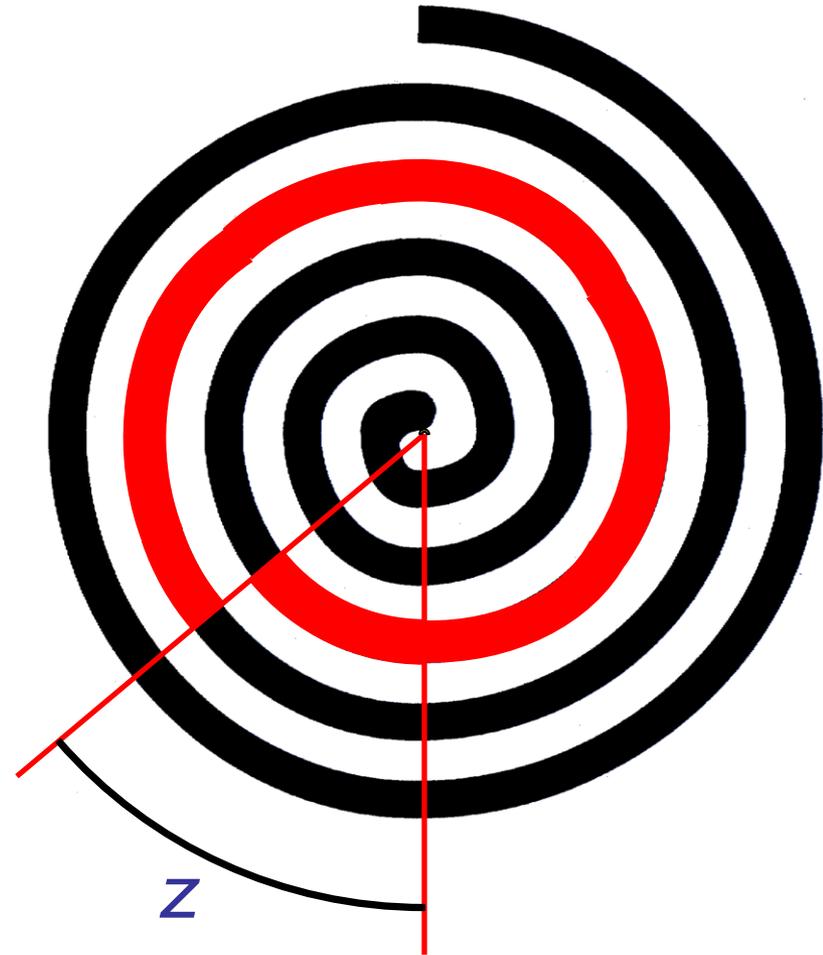
- Overflow records of bucket 5 at level 0 are stored in bucket $5\%3=2$ at level 1 (together with those of bucket 2)
- Overflow records of that bucket are stored at level 2 in bucket $2\%4=2$

Spiral hashing

- With Linear hashing there is a higher chance that buckets yet-unsplit during current expansion will
 - In fact, using an uniform hash function means that every value of $H_0(k)$ is equi-probable, but buckets for which $H_0(k) < SP$ have been already split
- Lo Spiral hashing (Martin, '79) tries to solve this issue by exploiting **an exponential function**, allowing to store records more densely in the initial portion of primary area
- The name of this organization derives from the fact that the storage area is considered as a spiral, rather than a line, and the primary area is a revolution of that spiral, univocally defined by an angle z

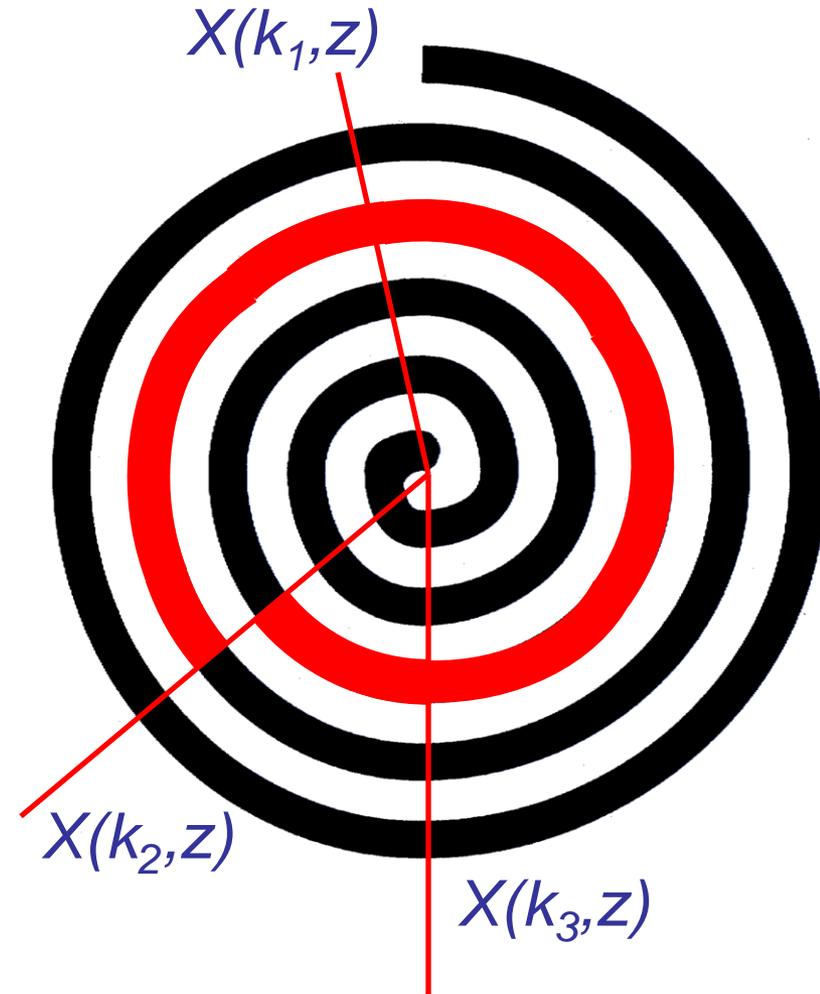
Spiral hashing: intuition

- Primary area expands with a **growth factor w**
 - The higher w , the quicker the growth of primary area
- Primary area lies along **a revolution of the spiral**
- The number of pages in the primary area is $P = \lfloor w^{z+1} \rfloor - \lfloor w^z \rfloor$
- The value of z is increased at each expansion
- Initially, $z=0$ ($P=1$)



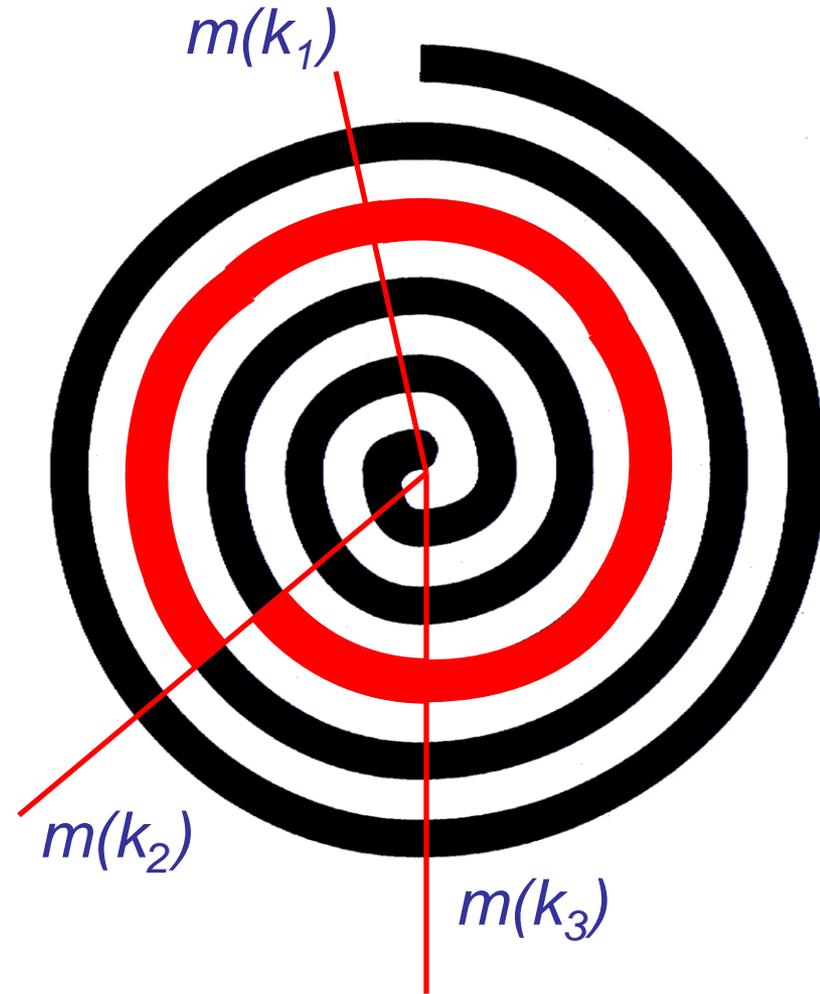
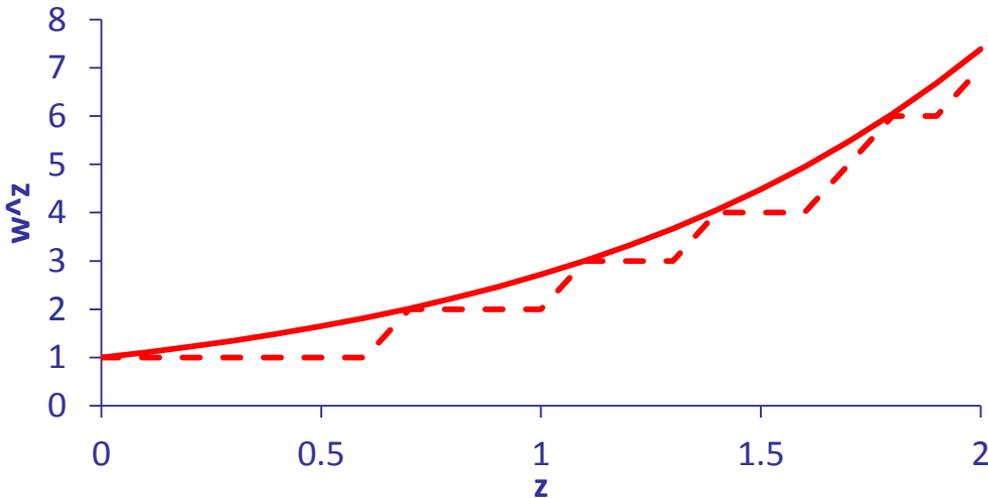
Spiral hashing: logical addresses (i)

- Let $H(k)$ be a hash function with values in $[0,1[$
- The “direction” of a key k is given by function $X(k,z) = \lceil z - H(k) \rceil + H(k)$ with values in $[z, z+1[$ and discontinuous in $z - \lfloor z \rfloor$



Spiral hashing: logical addresses (ii)

- The **logical address** of k is given by the exponential function $m = \lfloor w^{X(k,z)} \rfloor$
 - Minimum value $\lfloor w^z \rfloor$
 - Maximum value $\lfloor w^{z+1} \rfloor - 1$
- The **relative address** is $m_r = \lfloor w^{X(k,z)} \rfloor - \lfloor w^z \rfloor$

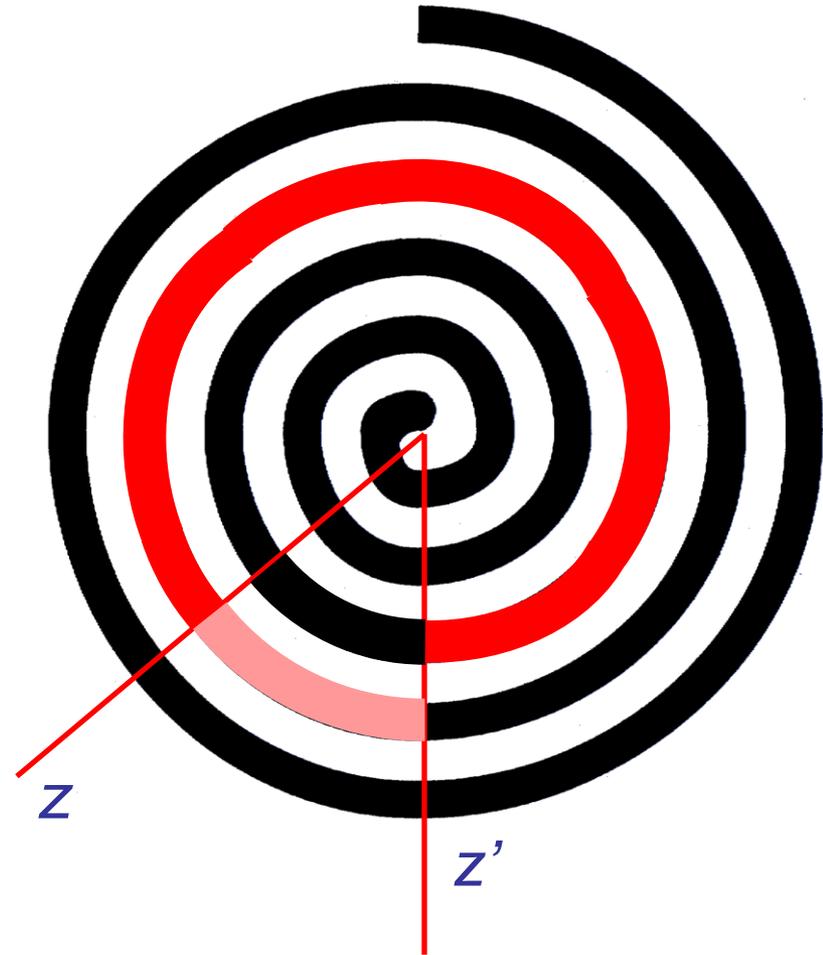


Spiral hashing: proprieties of addresses

- If the function $H(k)$ is uniform, keys are distributed uniformly in $[z, z+1[$
- The exponential behavior allows pages with small addresses to receive more records than pages with high addresses
 - Inversely depends on the graph slope
- Expansion of primary area is performed, as for linear hashing, “removing” records from the first bucket and moving them to a new bucket created at the end of the file

Spiral hashing: split

- We always split the bucket with lowest address
 - Actually, the address of the overflowed bucket is no longer generated
- Records of bucket $\lfloor w^z \rfloor$ are moved in new buckets created at the end of the file
- Value of the angle is thus increased from z to z'



Spiral hashing: new value of z

- The value of z' should guarantee that $\lfloor w^z \rfloor$ is no longer generated: $\lfloor w^{z'} \rfloor = \lfloor w^z \rfloor + 1$
- Thus: $z' = \log_w(\lfloor w^z \rfloor + 1)$
- Therefore, the number of added buckets is:
$$\lceil w^{z'+1} \rceil - \lceil w^{z'+1} \rceil = \lceil w(w^{z'+1}) \rceil - \lceil w \cdot w^z \rceil$$
that is, $\lceil w \rceil$ or $\lfloor w \rfloor$ depending on the value of z
- Thus, $w - 1$ determines the growth of primary area

Spiral hashing: example

- With $w=2$ and $z=0$, we obtain:

1	$z = 0, [\lceil 2^0 \rceil, \lceil 2^{0+1} \rceil - 1]$
---	---

2	3	$z' = \log_2(\lfloor 2^0 \rfloor + 1) = 1, [\lceil 2^1 \rceil, \lceil 2^{1+1} \rceil - 1]$
---	---	--

3	4	5	$z' = \log_2(\lfloor 2^1 \rfloor + 1) = \log_2 3,$ $[\lceil 2^{\log_2 3} \rceil, \lceil 2^{\log_2 3 + 1} \rceil - 1]$
---	---	---	--

4	5	6	7	$z' = \log_2(\lfloor 2^{\log_2 3} \rfloor + 1) = 2,$ $[\lceil 2^2 \rceil, \lceil 2^{2+1} \rceil - 1]$
---	---	---	---	--

5	6	7	8	9	$z' = \log_2(\lfloor 2^2 \rfloor + 1) = \log_2 5,$ $[\lceil 2^{\log_2 5} \rceil, \lceil 2^{\log_2 5 + 1} \rceil - 1]$
---	---	---	---	---	--

Spiral hashing: expansion

- Function $X(k,z)$ guarantees that logical addresses of keys in buckets other than the first are not modified
- In fact, $X(k,z)$ and $X(k,z')$ only differ for those keys having $H(k) \in [z - \lfloor z \rfloor, z' - \lfloor z' \rfloor[$, that is, between the two discontinuity points
- Since $z' = \log_w(\lfloor w^z \rfloor + 1)$, such keys are only those with logical address $\lfloor w^z \rfloor$

Spiral hashing: physical addresses

- Deleting the first bucket in the file poses the problem of **reusing** such pages
- It is therefore required to “map” logical addresses m into physical addresses $ph(m)$
- The basic schema is the following:
 - Numbering starts at 0
 - The first added bucket replaces the one deleted at the beginning
 - Other buckets are added at the end

Spiral hashing: allocation example

- With $w=3$ e $z=0$, we obtain:

$$\begin{array}{|c|} \hline 1 \\ \hline \end{array} \begin{array}{|c|} \hline 2 \\ \hline \end{array} \quad \lceil \lceil 3^0 \rceil, \lceil 3^{0+1} \rceil - 1 \rceil$$

$$\begin{array}{|c|} \hline 3 \\ \hline \end{array} \begin{array}{|c|} \hline 2 \\ \hline \end{array} \begin{array}{|c|} \hline 4 \\ \hline \end{array} \begin{array}{|c|} \hline 5 \\ \hline \end{array} \quad \lceil \lceil 3^{\log_3 2} \rceil, \lceil 3^{\log_3 2+1} \rceil - 1 \rceil$$

$$\begin{array}{|c|} \hline 3 \\ \hline \end{array} \begin{array}{|c|} \hline 6 \\ \hline \end{array} \begin{array}{|c|} \hline 4 \\ \hline \end{array} \begin{array}{|c|} \hline 5 \\ \hline \end{array} \begin{array}{|c|} \hline 7 \\ \hline \end{array} \begin{array}{|c|} \hline 8 \\ \hline \end{array} \quad \lceil \lceil 3^{\log_3 3} \rceil, \lceil 3^{\log_3 3+1} \rceil - 1 \rceil$$

$$\begin{array}{|c|} \hline 9 \\ \hline \end{array} \begin{array}{|c|} \hline 6 \\ \hline \end{array} \begin{array}{|c|} \hline 4 \\ \hline \end{array} \begin{array}{|c|} \hline 5 \\ \hline \end{array} \begin{array}{|c|} \hline 7 \\ \hline \end{array} \begin{array}{|c|} \hline 8 \\ \hline \end{array} \begin{array}{|c|} \hline 10 \\ \hline \end{array} \begin{array}{|c|} \hline 11 \\ \hline \end{array}$$

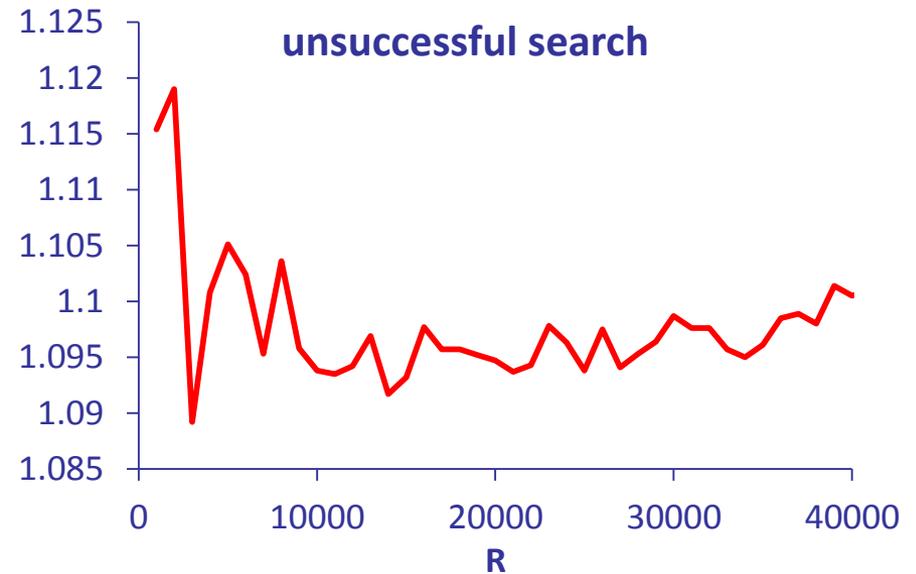
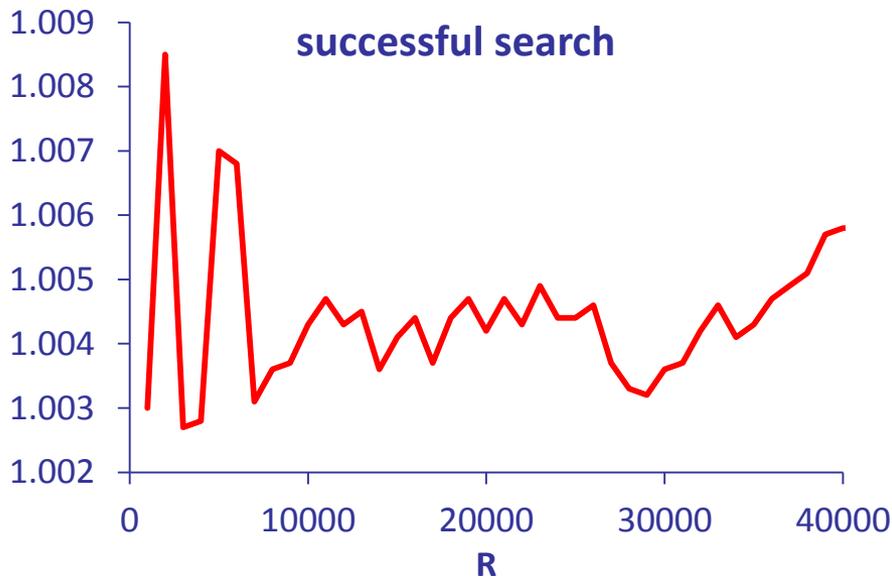
$$\lceil \lceil 3^{\log_3 4} \rceil, \lceil 3^{\log_3 4+1} \rceil - 1 \rceil$$

Spiral hashing: converting addresses

- Let $l = \lfloor (m-1)/w \rfloor$ and $h = \lfloor m/w \rfloor$, $ph(m)$ equals:
 - $ph(h)$, if $l < h$
 - $m - h - 1$, otherwise
- In the previous example, for bucket 6 we have:
 - $l = \lfloor 5/3 \rfloor = 1$, $h = \lfloor 6/3 \rfloor = 2$
- Therefore, we need to compute $ph(2)$ such that:
 - $l = \lfloor 1/3 \rfloor = 0$, $h = \lfloor 2/3 \rfloor = 0$
- Thus, $ph(6) = ph(2) = 2 - 0 - 1 = 1$
- For bucket 7, on the other hand: $l = 2$, $h = 2$, thus $ph(7) = 7 - 2 - 1 = 4$

Spiral hashing: performance (i)

- The main advantage of spiral hashing is the clear reduction of oscillatory phenomena of linear hashing
- $w=2, C=10, C_{ov}=3$



Spiral hashing: performance (ii)

- With controlled splits, increasing w increases search costs
- With uncontrolled splits, this does not happen, but storage utilization is reduced
- $C=10, C_{ov}=3, R=15000$

