Database Systems
Lecture13 – Chapter 14: Indexing

SUNG KYUN KWAN UNIVERSITY

Beomseok Nam (남범석)

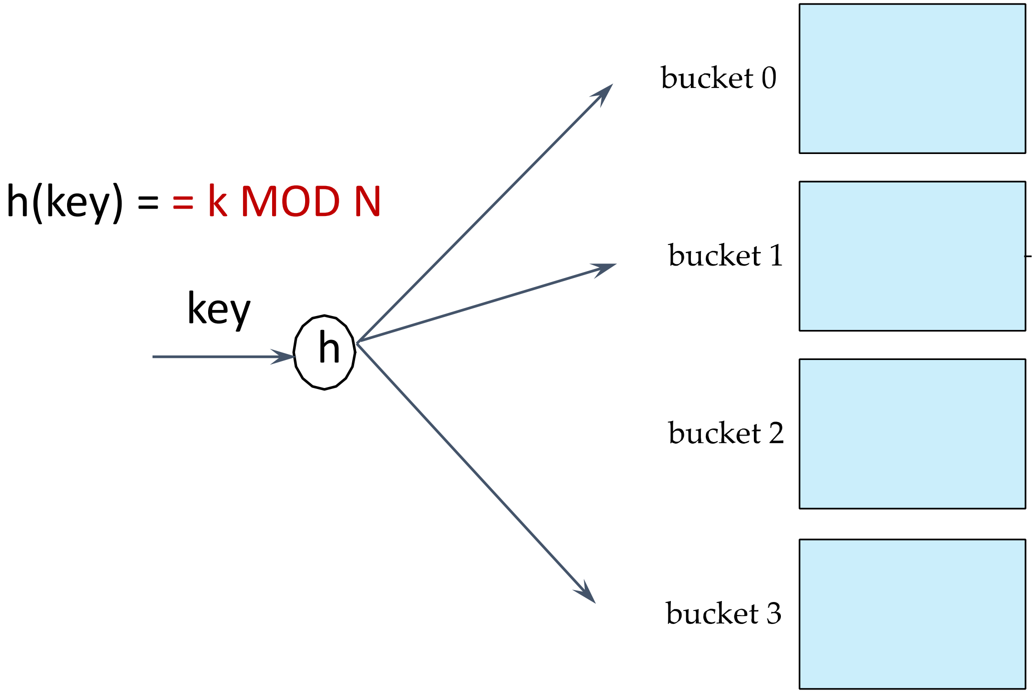
bnam@skku.edu

Static Hashing

- A **bucket** is a storage space containing entries
  - Bucket is typically a disk block (i.e., 4KB)
- Hash function is used to locate buckets
  - Hash function *h* is a function from the set of search-keys to the set of bucket addresses
- Different search-keys may be mapped to the same bucket
  - Entire bucket has to be searched sequentially
- In a **hash index**, buckets store entries with pointers to records
- In a **hash file-organization** buckets store records

Static Hashing

- # of buckets (pages) fixed
- Buckets are allocated sequentially, never de-allocated

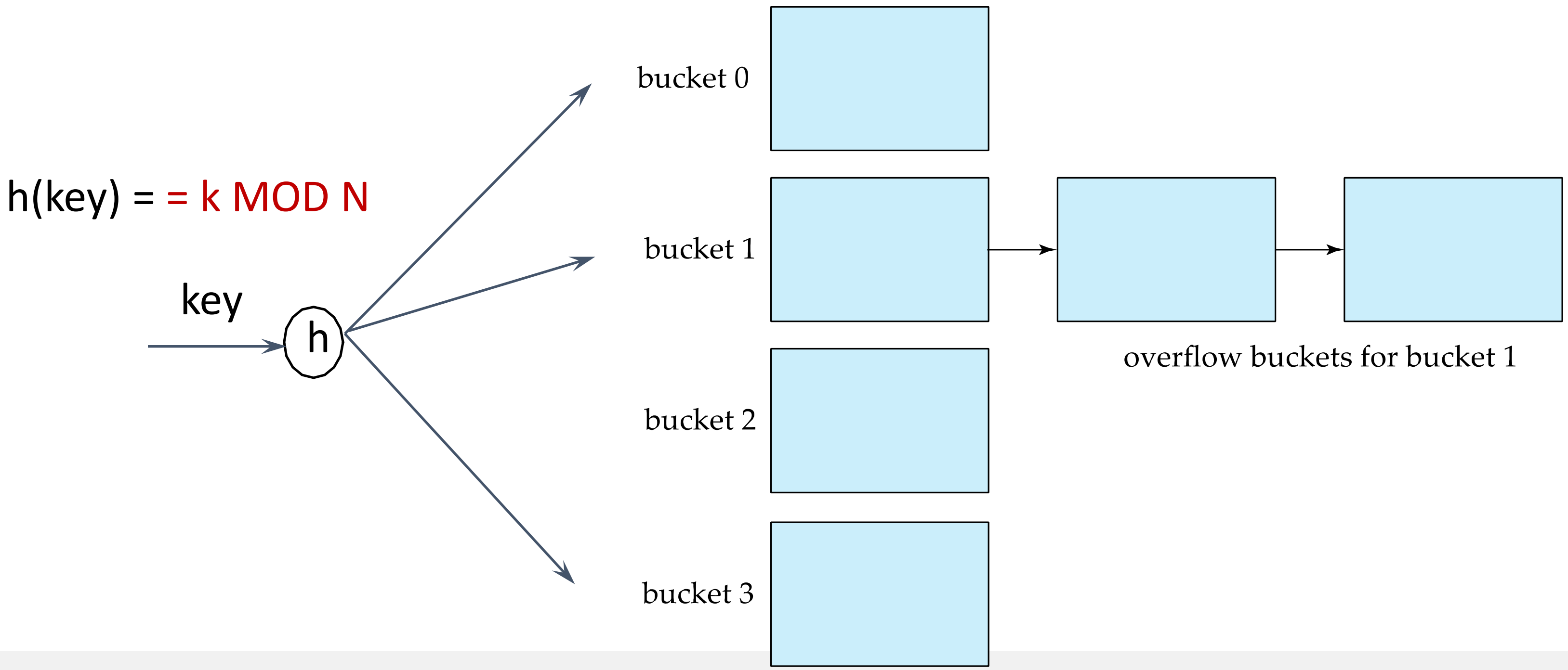


Handling of Bucket Overflows

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records.
- Skew can occur due to two reasons:
  - multiple records have same search-key
  - chosen hash function produces non-uniform distribution of key values
- The probability of bucket overflow can be reduced
- But bucket overflow cannot be eliminated
- Overflow is often handled by using ***overflow buckets***.

Handling of Bucket Overflows (Cont.)

- **Overflow chaining**
  - Overflow buckets are chained together in a linked list.
  - This is often called **Closed addressing (Closed hashing)**
  - An alternative, called **open hashing**, which does not use overflow buckets, is not suitable for database applications.
    - ***Linear probing***: Use the next bucket (in cyclic order) that has space.



Example of Hash File Organization

- Hash file organization of *instructor* file, using *dept_name* as key
- The binary representation of the *i*th character is assumed to be the integer *i.*
- The hash function returns the sum of the binary representations of the characters modulo 10
  - E.g. h(Music) = 1
  - h(History) = 2
  - h(Physics) = 3
  - h(Elec. Eng.) = 3
- The given hash function fails to provide a uniform distribution.

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Deficiencies of Static Hashing

- In static hashing, function *h* maps search-keys to a fixed set of bucket addresses.
- Databases grow or shrink with time.
  - If the number of buckets is too small, performance will degrade due to too much overflows.
  - If the number of buckets is too large, buckets will be underfull and disk space will be wasted.
- One solution: **Rehashing**
  - periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically.

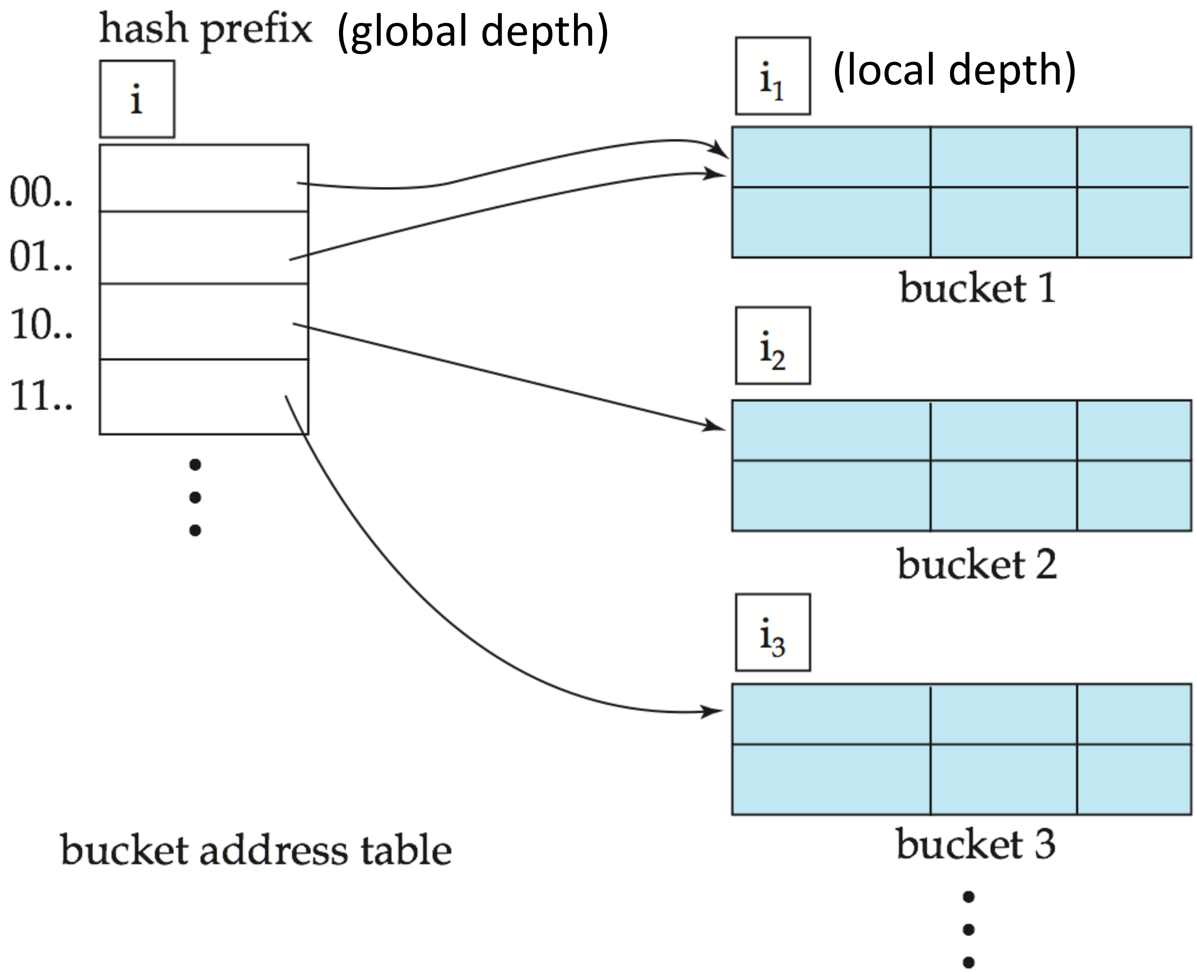
Dynamic Hashing

- **Linear Hashing**
  - Do rehashing in an incremental manner
- **Extendable Hashing**
  - Tailored to disk based hashing, with buckets shared by multiple hash values
  - Doubling of # of entries in hash table, without doubling # of buckets
- Basic idea behind Dynamic Hashing
  - Doubling # of buckets (directories)
  - *Idea*: Add a level of indirection!
    - Use *directory of pointers to buckets*,
    - Double # of buckets by *doubling the directory*
    - Split only the bucket that just overflowed!
  - *Trick lies in how hash function is adjusted!*

Extendable Hashing

- Use only a postfix (or prefix) of the hash key
- Let the length of the postfix (the least significant bits) be i bits, $0 \leq i \leq 32$.
  - Directory (bucket address table) size = 2^{i}.
  - Initially $i = 0$
  - Value of i grows and shrinks as the size of the database grows and shrinks.
- Multiple directory entries may point to a single bucket
- Thus, actual number of buckets is $< 2^i$
  - The number of buckets also changes dynamically due to coalescing and splitting of buckets.

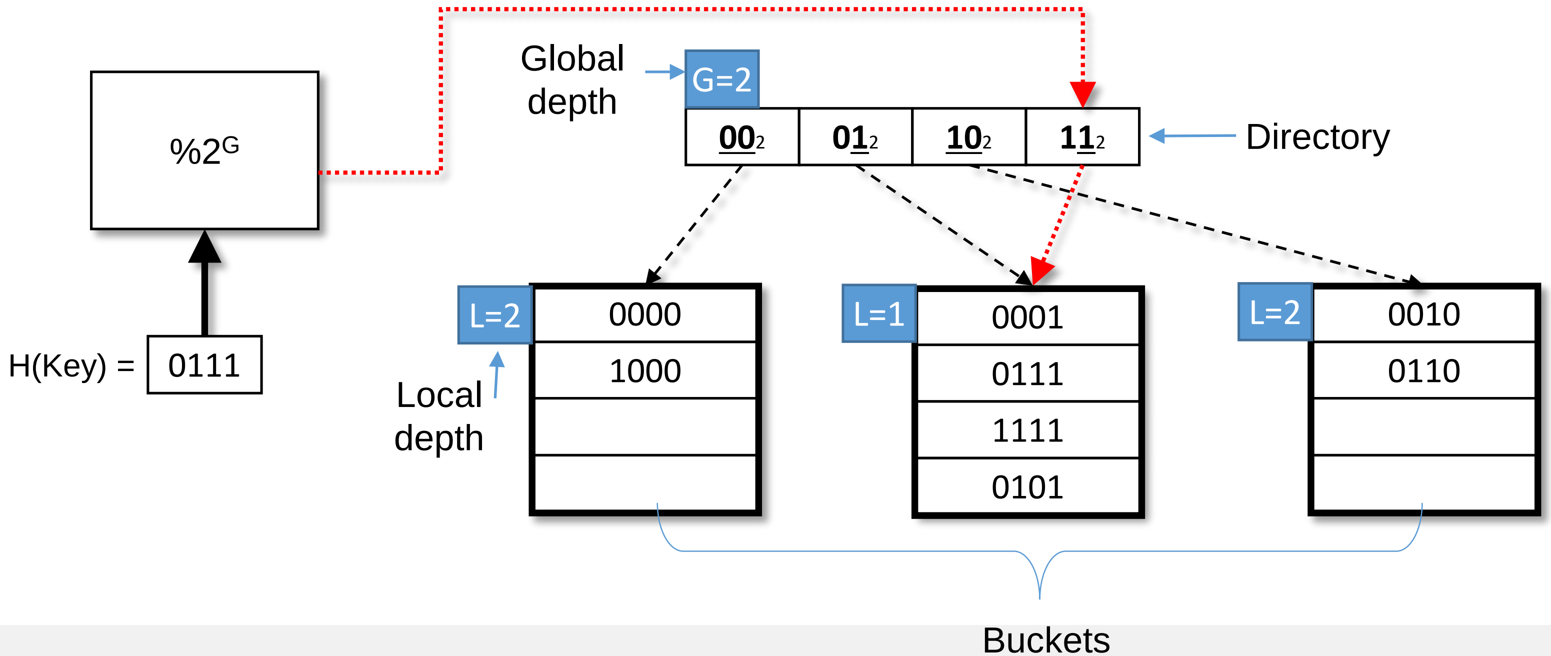
General Extendable Hash Structure



In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$

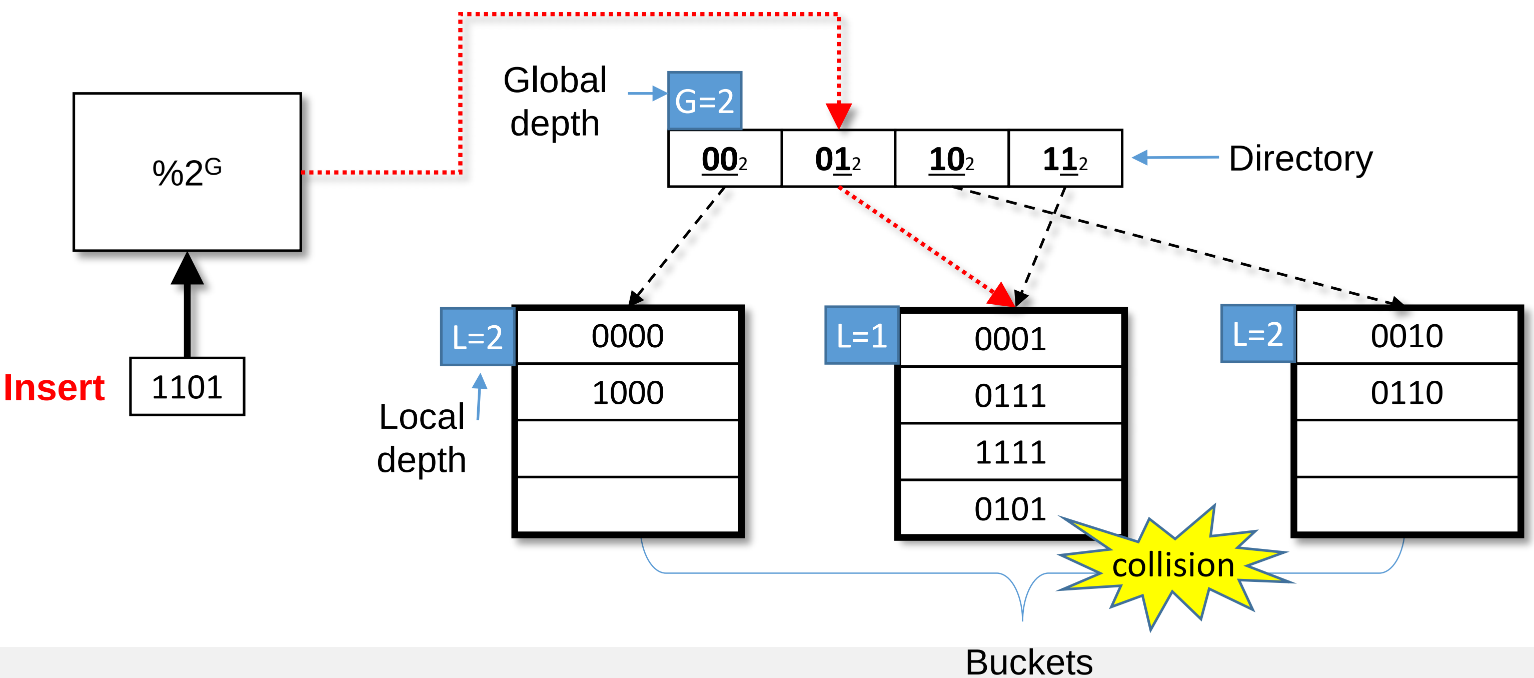
Extendable Hashing – How it Works

- Directory is an array of size 4, so 2 bits are needed.
- To find a bucket for key K, use `global depth'
- global depth = least significant bits of h(K);
- If h(r) = 5 = binary 0101, it is in bucket pointed to by 01.
- If h(r) = 7 = binary 0111, it is in bucket pointed to by 11



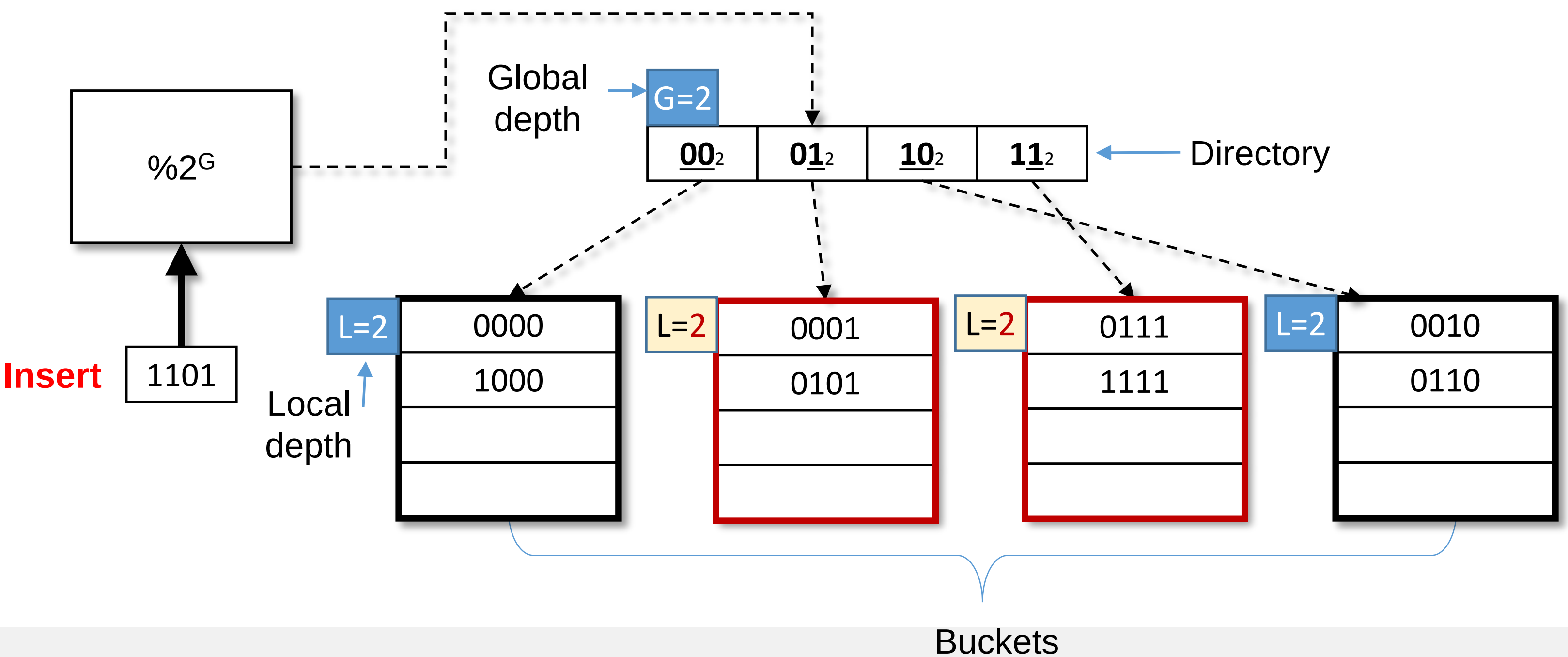
Extendable Hashing – How it Works

- Bucket Split
  - if G> *L* (more than one pointer to bucket)
    - allocate new a bucket, and set *L* = *L+1*
    - Update the directory to point to the new bucket
    - move records in the overflow bucket to the new bucket
    - Further splitting is required if the bucket is still full



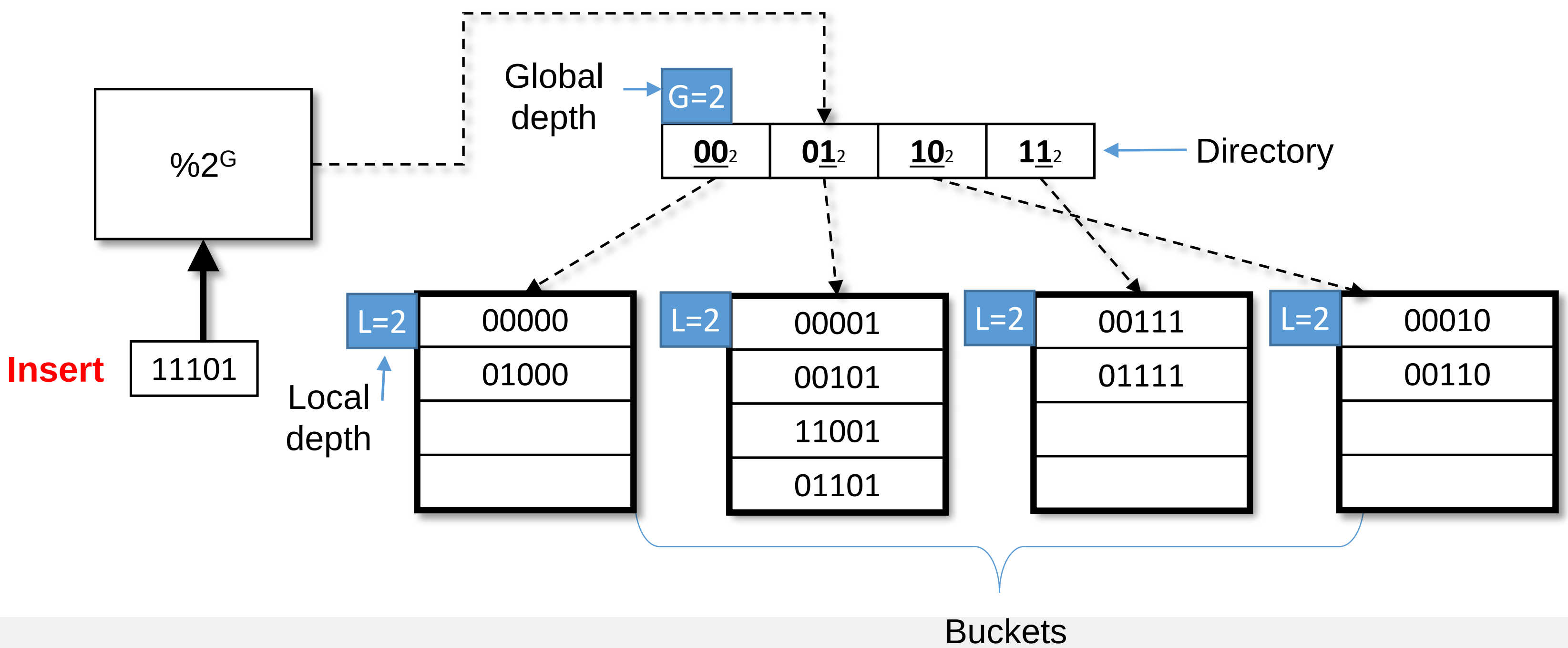
Extendable Hashing – How it Works

- Bucket Split
  - if G> *L* (more than one pointer to bucket)
    - allocate new a bucket, and set *L* = *L+1*
    - Update the directory to point to the new bucket
    - move records in the overflow bucket to the new bucket
    - Further splitting is required if the bucket is still full



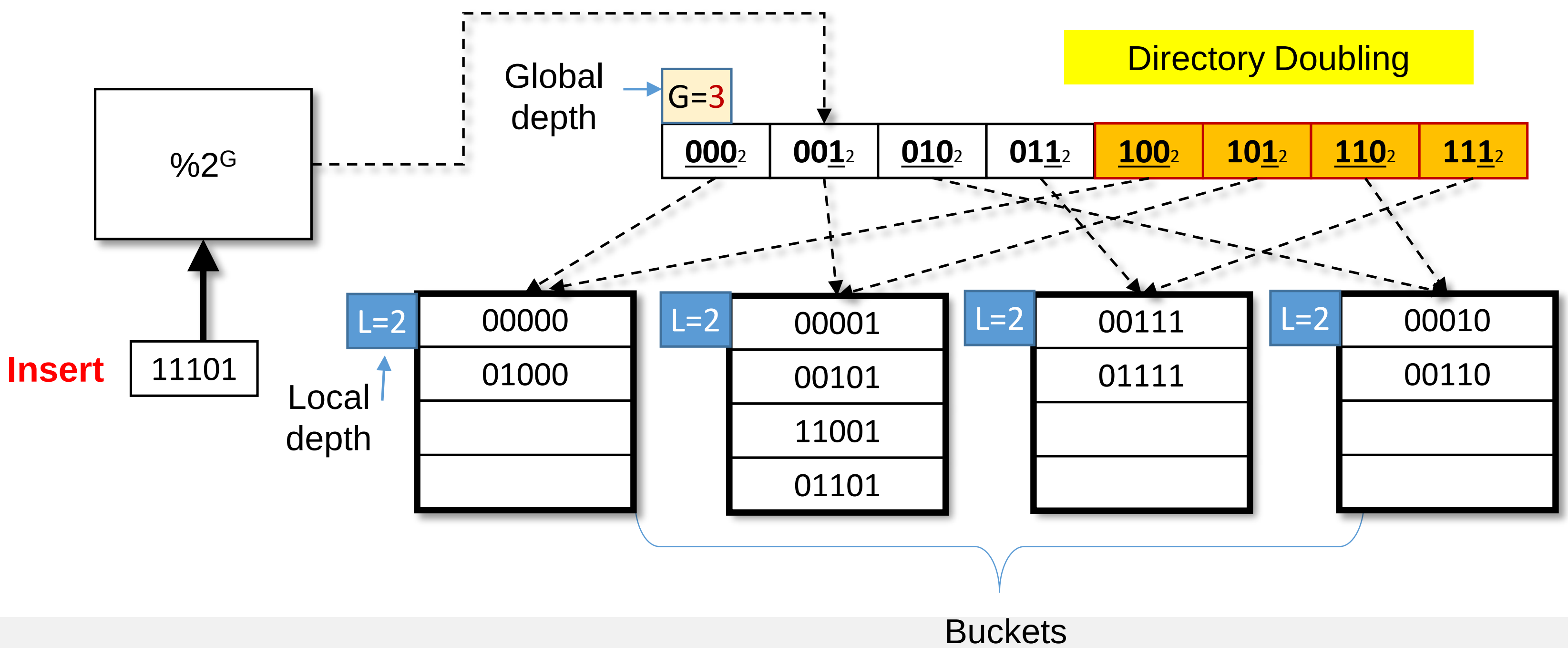
Extendable Hashing – How it Works

- Bucket Split → Directory Doubling
  - If $G = L$ (only one pointer to bucket)
    - increment G and double the size of directory.
    - replace each entry by two entries that point to the same bucket.
    - Now $G > L$, so use the first case in the previous slide.



Extendable Hashing – How it Works

- Bucket Split → Directory Doubling
  - If *G* = *L* (only one pointer to bucket)
    - increment *G* and double the size of directory.
    - replace each entry by two entries that point to the same bucket.
    - Now *G*> *L*, so use the first case in the previous slide.



Extendable Hashing vs. Other Schemes

- Benefits of extendable hashing:
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
- Disadvantages of extendable hashing
  - Extra level of indirection to find desired record
  - Bucket address table may itself become very big (larger than memory)
    - Cannot allocate very large contiguous areas on disk either
    - Solution: B^+-tree structure for bucket address table
  - Changing size of bucket address table is an expensive operation
- Linear hashing is an alternative mechanism
  - Allows incremental growth of its directory (equivalent to bucket address table)
  - At the cost of more bucket overflows

Linear Hashing – a lazy approach

- A dynamic hashing scheme that handles the problem of long overflow chains without using a directory.

- Directory is avoided
- Temporary overflow pages are used
- The bucket to split is chosen in a round-robin fashion

- When any bucket overflows split the bucket that the “Next” pointer currently points to and then increment that pointer to the next bucket.

- Use a family of hash functions h_0, h_1, h_2, ...
- $h_i(key) = h(key) \bmod (2^i N)$
  - N = initial # buckets (must be a power of 2)
  - h is some hash function
- h_{i+1} doubles the range of h_i (similar to directory doubling)

Linear Hashing - algorithm

- Algorithm proceeds in \`rounds'.
- Current round number is "*Level*". (Level = round)

- There are $N_{Level} = N * 2^{Level}$ buckets at the beginning of a round
  - i.e. $N = N_0$
- Buckets 0 to *Next-1* have been split
- *Next* to N_{Level} have not been split yet this round.
- Round ends when all initial buckets have been split.
  - i.e. $Next = N_{Level}$
- To start next round:

  Level++;

  Next = 0;

Linear Hashing - Insert

- Find a bucket, if fits, then DONE.
- else, if there's no space:
  - Add a temporary overflow page and insert data entry into that.
  - Split *Next* bucket and increment *Next*.
    - This is likely NOT the bucket that just overflew!!!
    - After split, use $\mathbf{h}_{Level+1}$ to re-distribute entries.

- Since buckets are split round-robin, long overflow chains are likely to develop!

Linear Hashing – How it Works

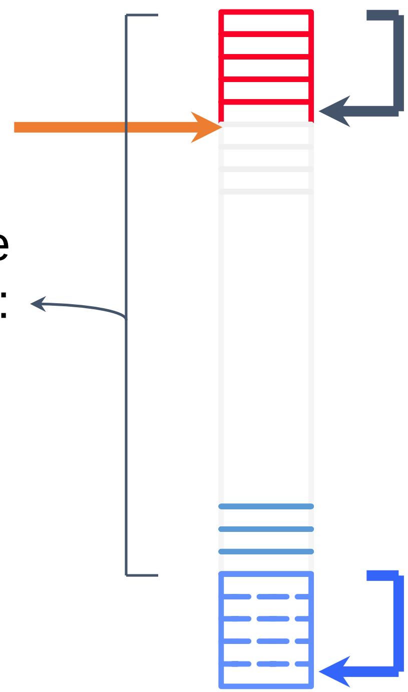
- Overview

Bucket to be split

Next

Buckets that existed at the beggining of this round: this is the range of

$N_{Level} = N*2^{Level}$

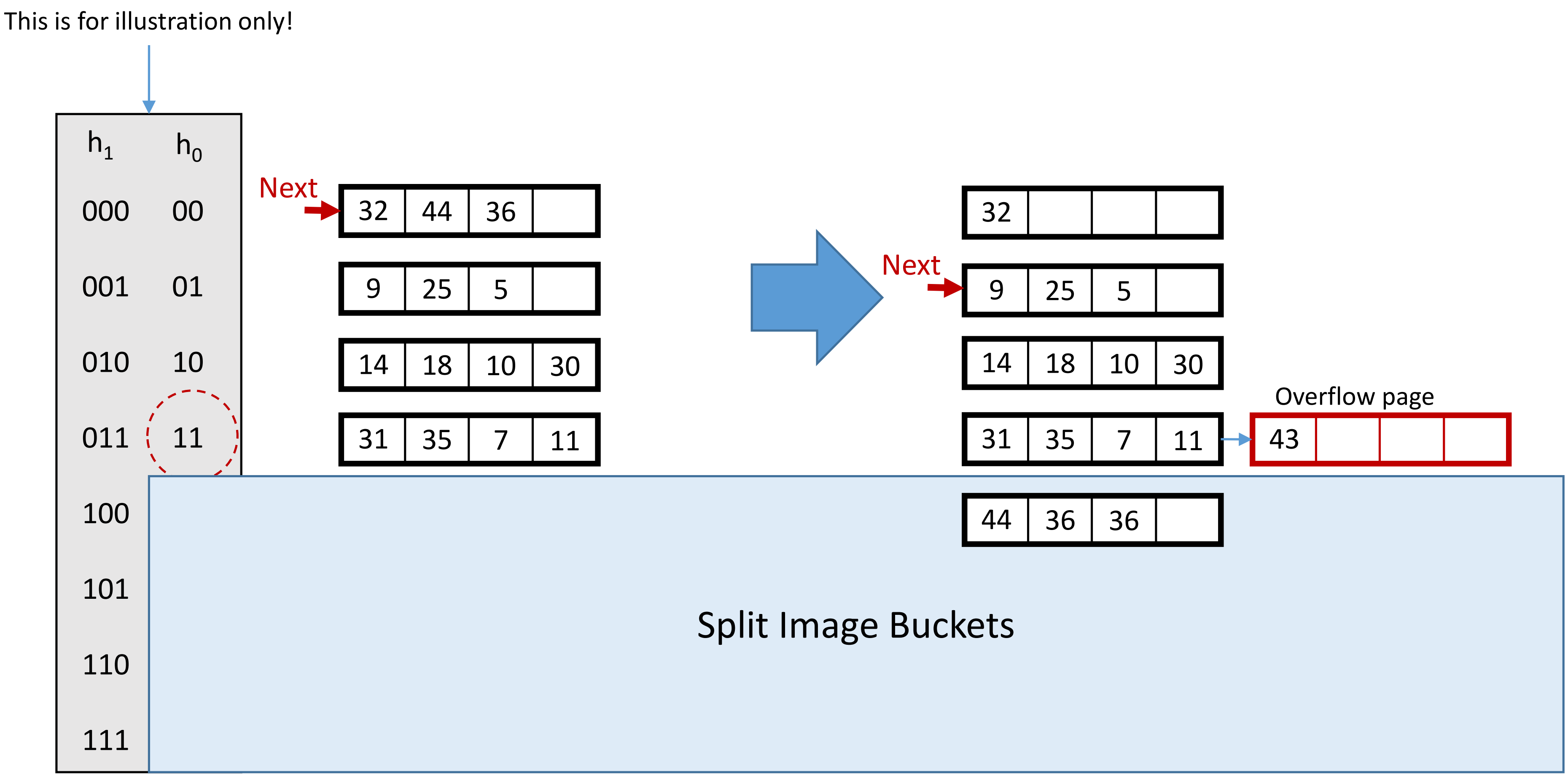


Buckets splitted in this round: If h_{Level}(search key) is in this range, must use $h_{Level+1}$(search key) to decide if entry is in `split image' bucket.

`split image' buckets: created (through splitting of other buckets) in this round

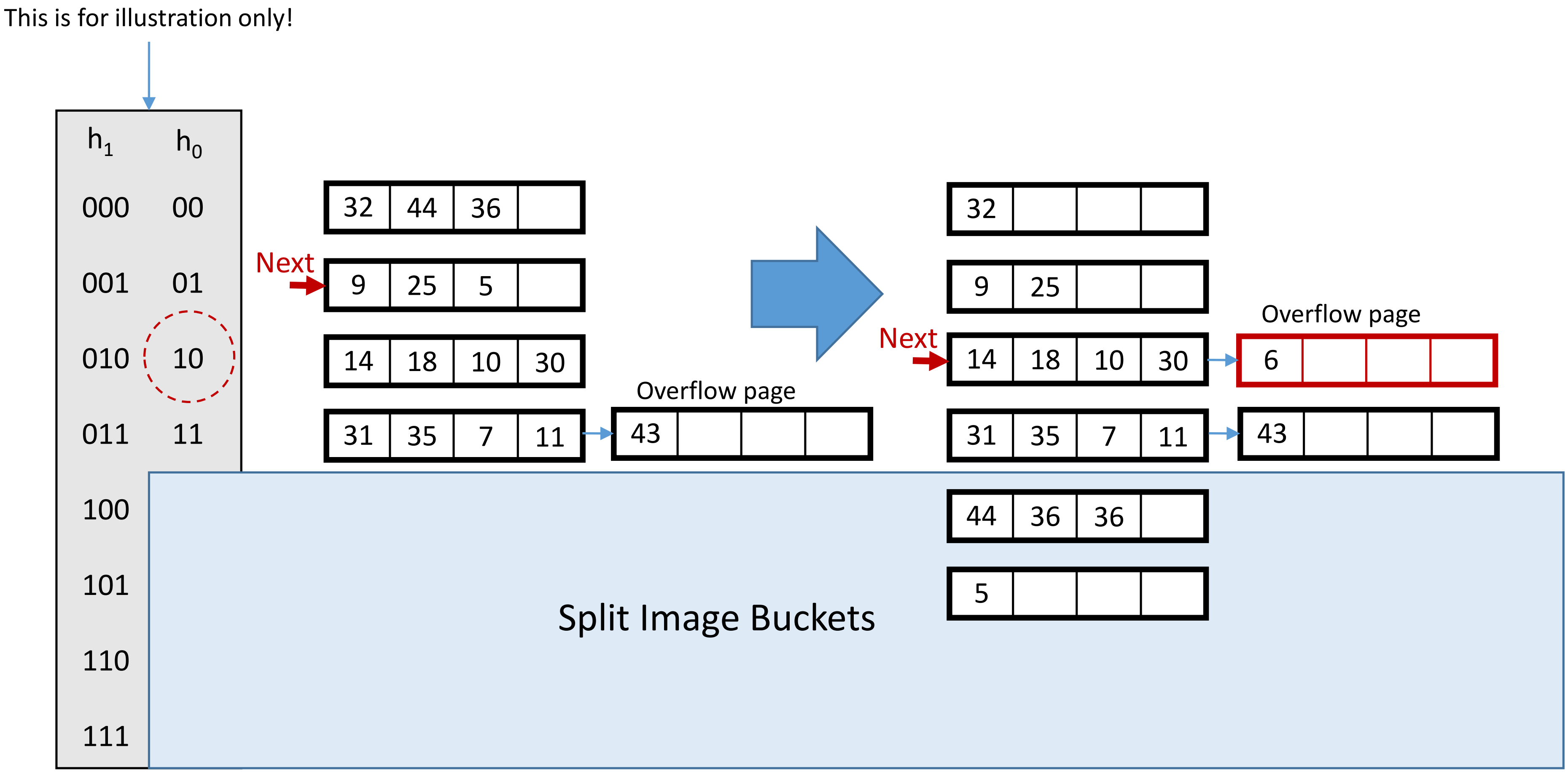
Linear Hashing – How it Works

- Insert 43 ($101011_{(2)}$) (Level 0, N=4)



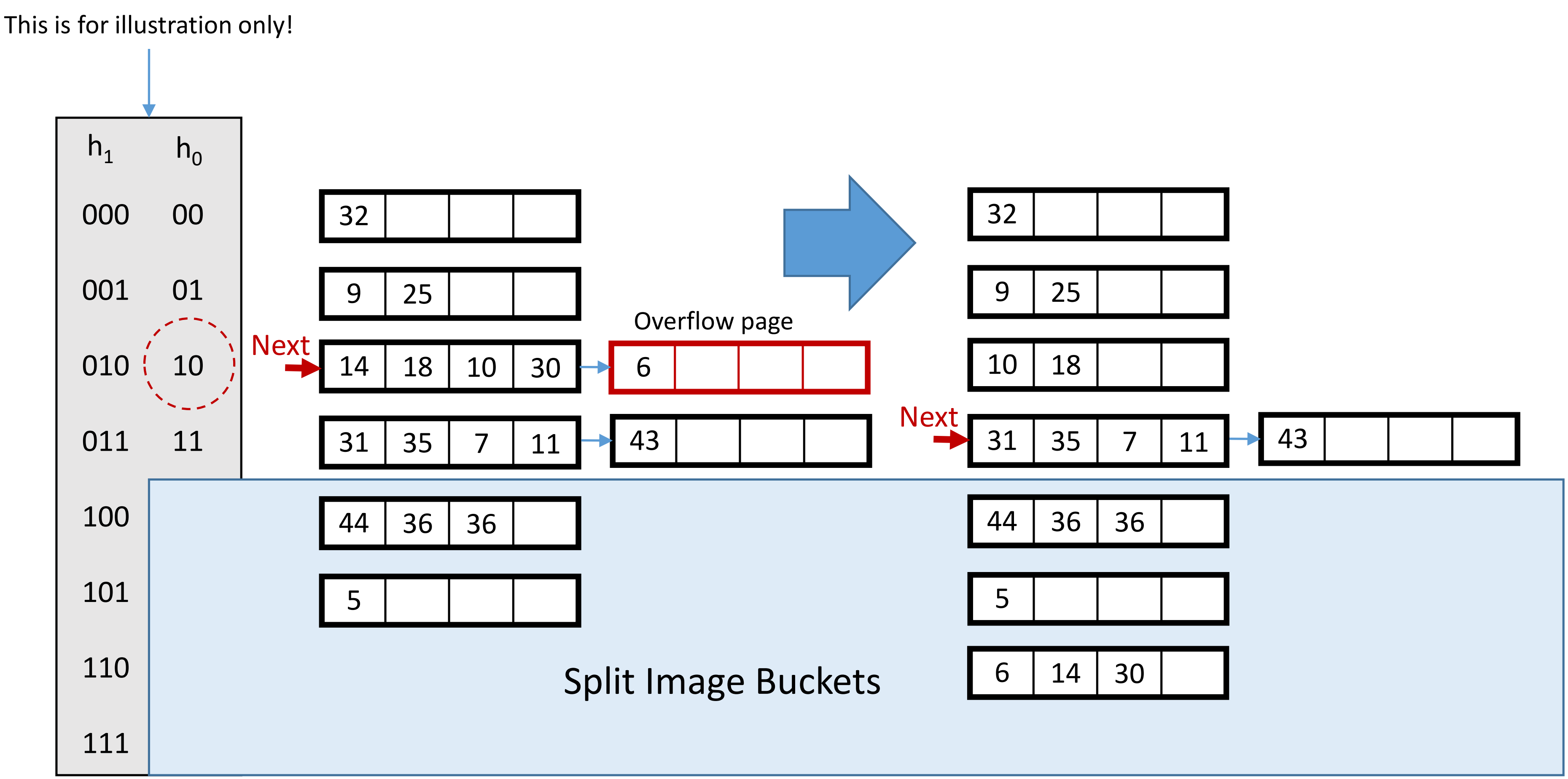
Linear Hashing – How it Works

- Insert 6 ($110_{(2)}$) (Level 0, N=4)



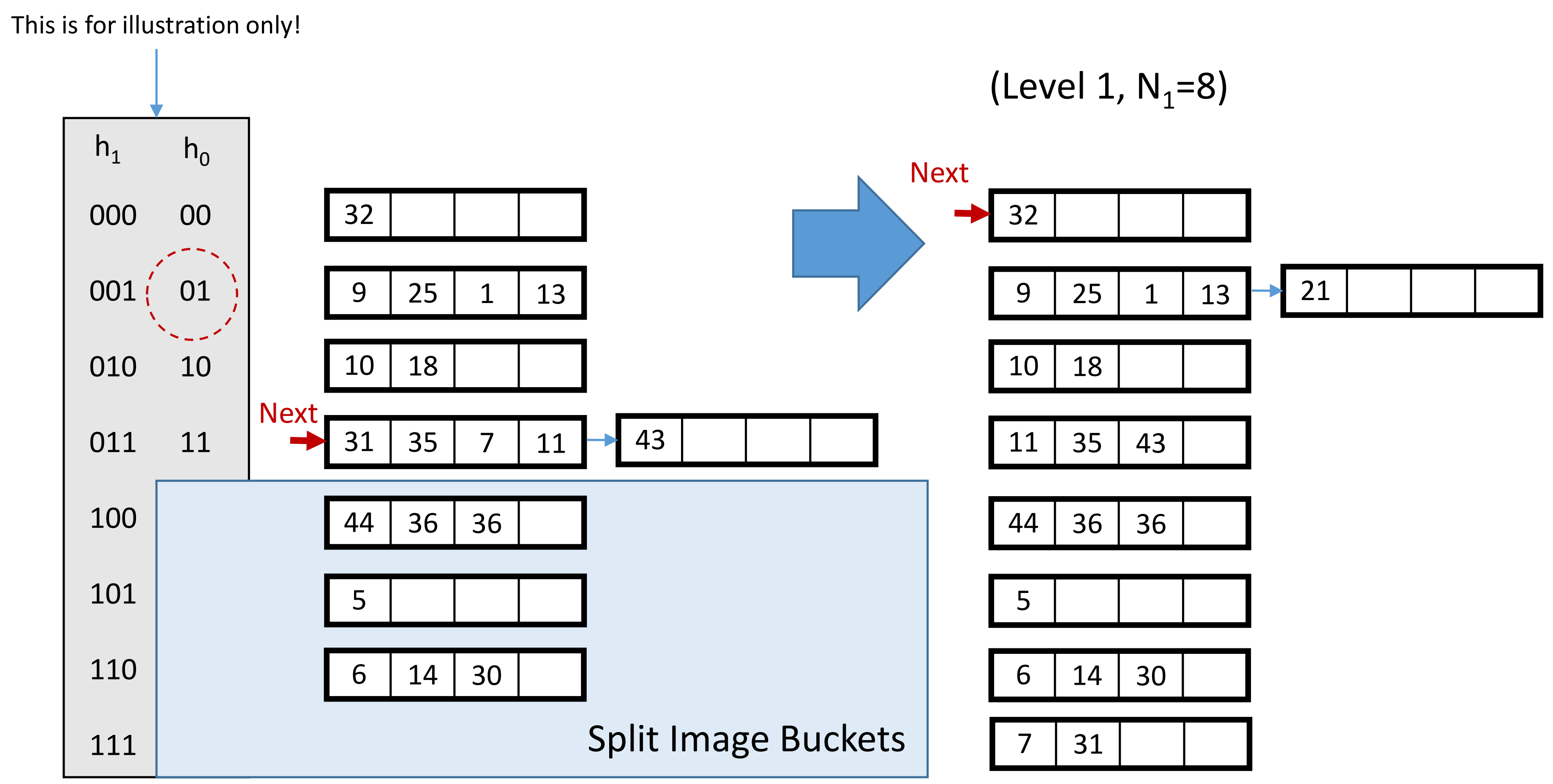
Linear Hashing – How it Works

- Suppose another bucket overflow causes a bucket split.



Linear Hashing – How it Works

- Insert 21 ($10101_{(2)}$) (Level 0, N=4)



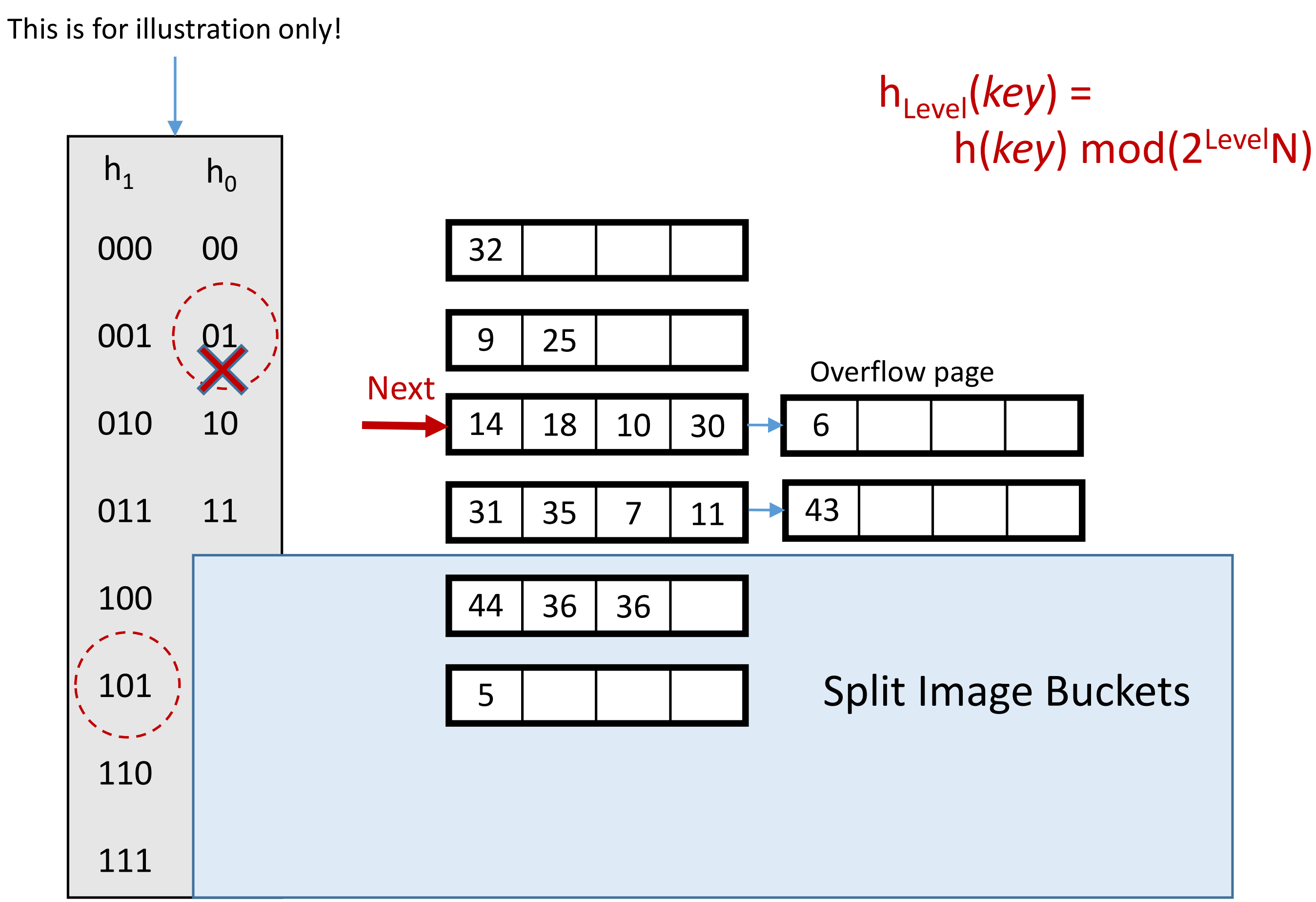
Linear Hashing Search Algorithm

- To find bucket for data entry r, find $h_{Level}(r)$:

- Buckets not yet split this round:
  - If $\mathbf{h}_{Level}(r)$ >= Next (i.e., $\mathbf{h}_{Level}(r)$ is a bucket that hasn't been involved in a split this round) then *r* belongs in that
    bucket for sure.
- Buckets split already this round:
  - Else, r could belong to bucket $\mathbf{h}_{Level}(r)$ or bucket $\mathbf{h}_{Level}(r) + N_{Level}$
  - must apply $\mathbf{h}_{Level+1}(r)$ to find out.

Linear Hashing – How it Works

- Search 5 ($00101_{(2)}$) (Level 0, N=4)



Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better for point query
  - If range queries are common, ordered indices are to be preferred

- In practice:
  - PostgreSQL supports hash indices, but discourages use due to poor performance
  - Oracle supports static hash organization, but not hash indices
  - SQLServer supports only B^+-trees

Multiple-Key Access and Bitmap Index

- Use multiple indices for certain types of queries.
- Example:

  select *ID*

  from *instructor*

  where *dept_name* = "Finance" **and** *salary* = 80000

- Possible strategies for processing query using indices on single attributes:
  1. Use index on *dept_name* to find instructors with department name Finance; test *salary = 80000*
  2. Use index on *salary* to find instructors with a salary of $80000; test *dept_name* = "Finance".
  3. Take intersection of both sets

Multiple-Key Access and Bitmap Index

- Bitmap indices are designed for efficient querying on multiple keys
- Records are assumed to be numbered sequentially from, say, 0
  - Given a number *n* it must be easy to retrieve record *n*
    - Particularly easy if records are of fixed size
- Applicable on attributes that take on a small number of distinct values
  - E.g., gender, country, state, …
  - E.g., income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000-infinity)
- A bitmap is simply an array of bits

Bitmap Indices (Cont.)

- a bitmap index has a bitmap for each value of the attribute
  - Bitmap has as many bits as records
  - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise
- Example

record number	*ID*	*gender*	*income_level*
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Bitmap Indices (Cont.)

- Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
  - Intersection (and)
  - Union (or)
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
  - E.g., 100110 AND 110011 = 100010
    100110 OR 110011 = 110111
    NOT 100110 = 011001
  - Males with income level L1: 10010 AND 10100 = 10000
    - Can then retrieve required tuples.
    - Counting number of matching tuples is even faster

Bitmap Indices (Cont.)

- Bitmap indices generally very small compared with relation size
  - E.g., if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation.
    - If number of distinct attribute values is 8, bitmap is only 1% of relation size

Creation of Indices

- Example

  create index *takes_pk*

  on *takes* (*ID,course_ID, year, semester, section*)

  drop index *takes_pk*

- Most database systems allow specification of type of index, and clustering.
- Indices on primary key created automatically by all databases
- Some database also create indices on foreign key attributes
  - Index on foreign key would be useful for this query:
    - $takes \bowtie \sigma_{name='Shankar'}(student)$
- Indices can greatly speed up lookups, but impose cost on updates

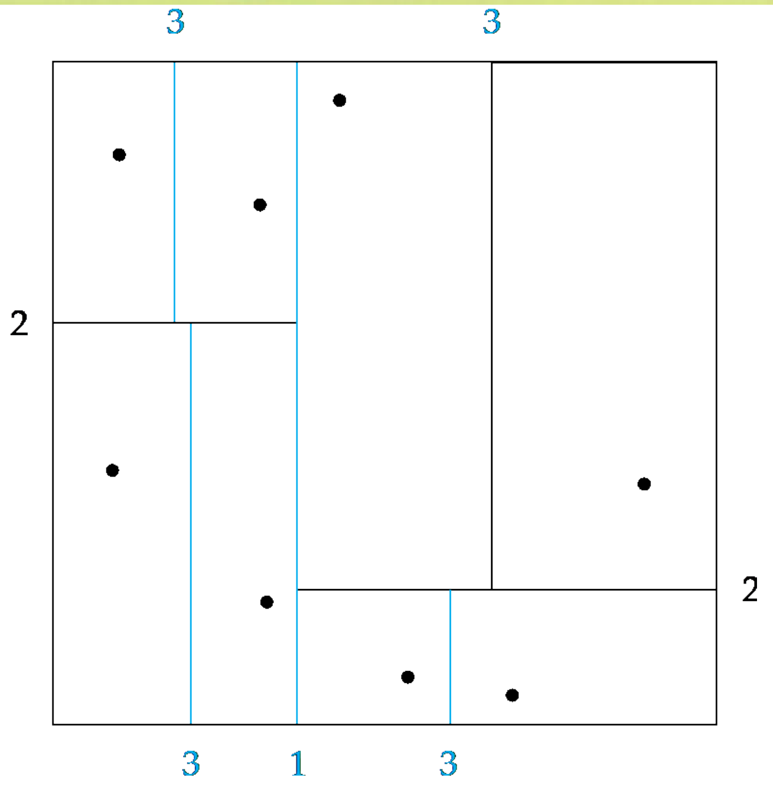
Spatial Indices

Spatial Data

- Databases can store data types such as lines, polygons, in addition to raster images
  - allows relational databases to store and retrieve spatial information
  - Queries can use spatial conditions (e.g. contains or overlaps).
  - queries can mix spatial and nonspatial conditions
- **Nearest neighbor queries**, given a point or an object, find the nearest object that satisfies given conditions.
- **Range queries** deal with spatial regions. e.g., ask for objects that lie partially or fully inside a specified region.
- Queries that compute intersections or unions of regions.
- **Spatial join** of two spatial relations with the location playing the role of join attribute.

Indexing of Spatial Data

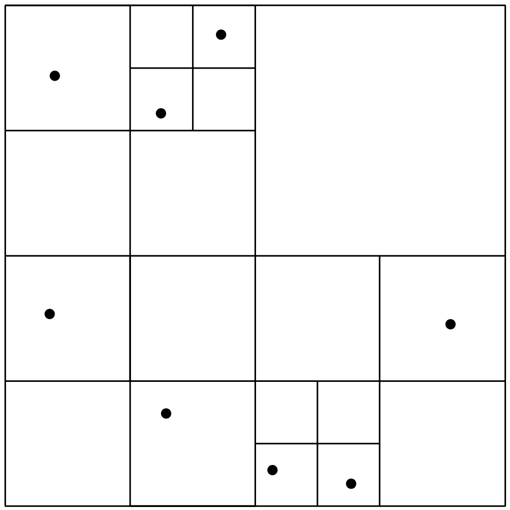
- **k-d tree** - early structure used for indexing in multiple dimensions.
- Each level of a *k-d* tree partitions the space into two.
  - Choose one dimension for partitioning at the root level of the tree.
  - Choose another dimensions for partitioning in nodes at the next level and so on, cycling through the dimensions.
- In each node, approximately half of the points stored in the sub-tree fall on one side and half on the other.
- Partitioning stops when a node has less than a given number of points.



- The **k-d-B tree** extends the *k-d* tree to allow multiple child nodes for each internal node; well-suited for secondary storage.

Division of Space by Quadtrees

- Each node of a quadtree is associated with a rectangular region of space; the top node is associated with the entire target space.
- Each non-leaf nodes divides its region into four equal sized quadrants
  - correspondingly each such node has four child nodes corresponding to the four quadrants and so on
- Leaf nodes have between zero and some fixed maximum number of points (set to 1 in example).

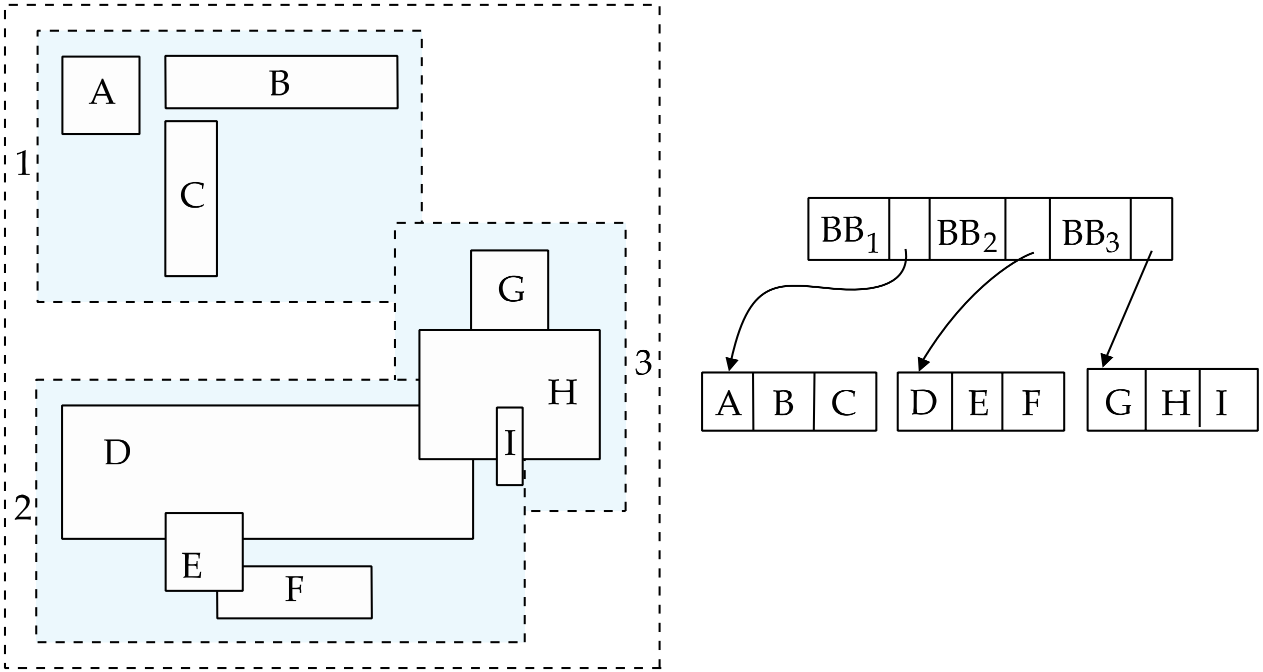


R-Trees

- **R-trees** are a N-dimensional extension of B^+-trees, useful for indexing sets of rectangles and other polygons.
- Supported in many modern database systems, along with variants like R^+ -trees and R^*-trees.
- Basic idea: generalize the notion of a one-dimensional interval associated with each B+ -tree node to an N-dimensional interval, that is, an N-dimensional rectangle.
- Will consider only the two-dimensional case ($N = 2$)
  - generalization for $N > 2$ is straightforward, although R-trees work well only for relatively small N
- The **bounding box** of a node is a minimum sized rectangle that contains all the rectangles/polygons associated with the node
  - *Bounding boxes of children of a node are allowed to overlap*

Example R-Tree

- A set of rectangles (solid line) and the bounding boxes (dashed line) of the nodes of an R-tree for the rectangles.
- The R-tree is shown on the right.



Search in R-Trees

- To find data items intersecting a given query point/region, do the following, starting from the root node:
  - If the node is a leaf node, output the data items whose keys intersect the given query point/region.
  - Else, for each child of the current node whose bounding box intersects the query point/region, recursively search the child
- Can be very inefficient in worst case since multiple paths may need to be searched, but works acceptably in practice.