CSCI3170 Introduction to Database Systems

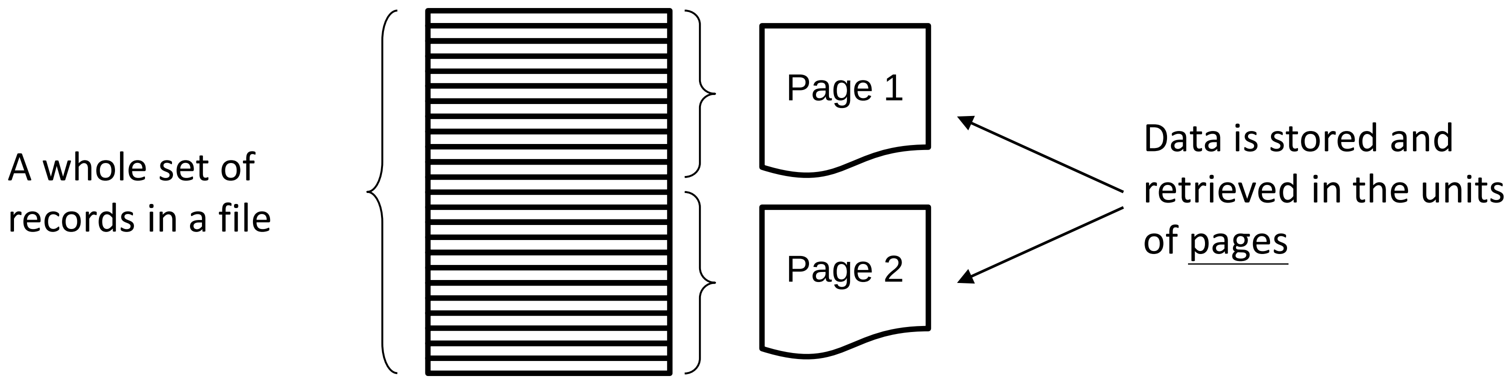
Tutorial 7 – Storage and Indexes

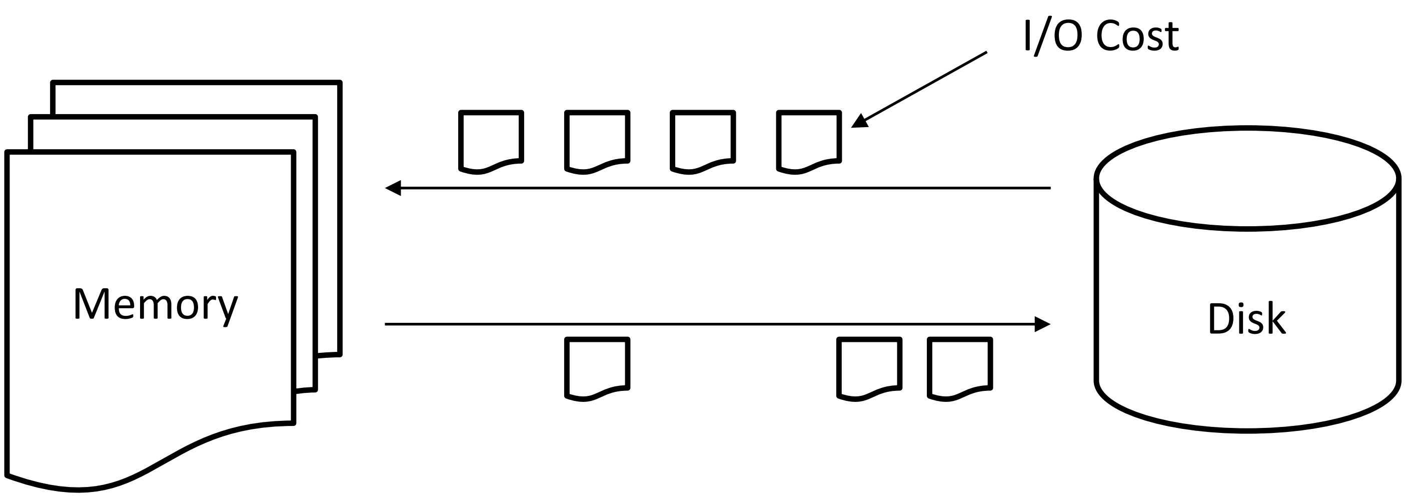
Outline

- Overview of Storage and Indexes
- Tree-structured Indexing
  - B+ Tree
- Hash-based Indexes
  - Static Hashing
  - Dynamic Hashing

Storage

- Files and pages





Record and Index

- Record
  - Record ID = <Page ID>+<Offset>
    - E.g. Record ID: <3>+<10> = The 10^{th} record in the 3^{rd} page
- Index
  - Given a search key K, index can be used to speed up the selection of a set of particular pages.
  - An index file contains a collection of data entries.
  - The data entry of search key K is denoted as K*.
    - K* = <K, Record ID (rid)>

Index Classification

- Primary and Secondary
  - Primary
    - Search key contains primary key.
  - Secondary
    - Otherwise
- Dense and Sparse
  - Dense
    - K* appear for ALL search key
  - Sparse
    - Otherwise

Index Classification (2)

- Clustered and Unclustered
  - Clustered
    - Data entries and records are sorted by K.
    - Order of records are equal/close to the order of data entries in index.
    - Support range search.
  - Unclustered
    - Otherwise

Index Classification: an Example

- Employee = { <u>Name</u> , Office }

contains primary key

in the same order

Name
Bryan
Eric
Fanny
Raymond
Steve
Teddy

Name	Office
Bryan	Rm 1026
Eric	Rm 913
Fanny	Rm 231
Raymond	Rm 1026
Steve	Rm 913
Teddy	Rm 913

Primary	~~Secondary~~
Clustered	~~Unclusterd~~
Dense	~~Sparse~~

Name
Bryan
Eric
Fanny
Raymond
Steve
Teddy

Index Classification: an Example

- Employee = { Name , Office }

not in the same order

contains primary key

Name
Steve
Fanny
Bryan

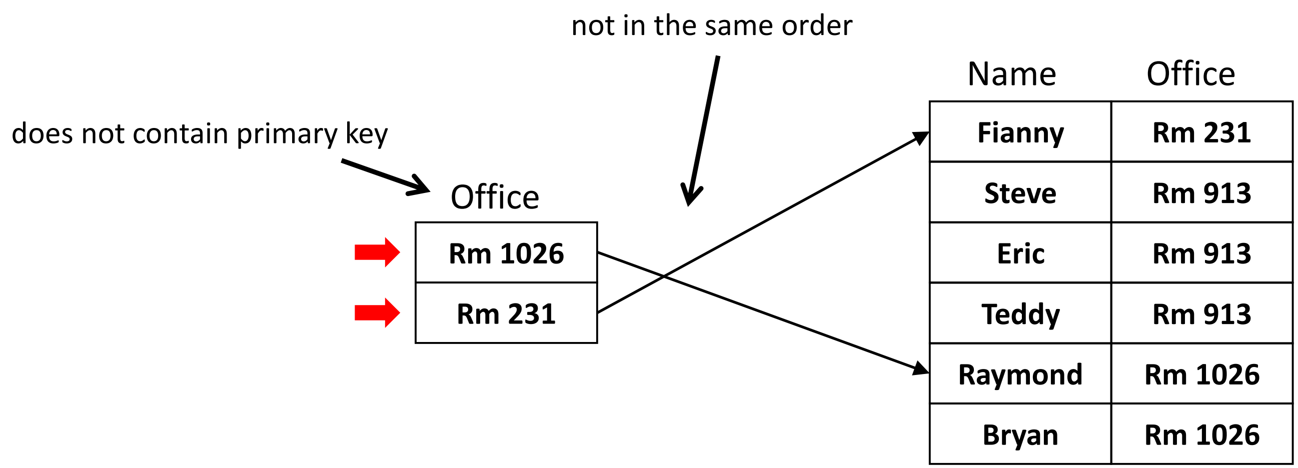
Name	Office
Bryan	**Rm 1026**
Eric	**Rm 913**
Fanny	**Rm 231**
Raymond	**Rm 1026**
Steve	**Rm 913**
Teddy	**Rm 913**

Primary	~~Secondary~~
~~Clustered~~	Unclusterd
~~Dense~~	Sparse

Name	
Bryan	✓
Eric	✗
Fanny	✓
Raymod	✗
Steve	✓
Teddy	✗

Index Classification: an Example

- Employee = { <u>Name</u> , Office }

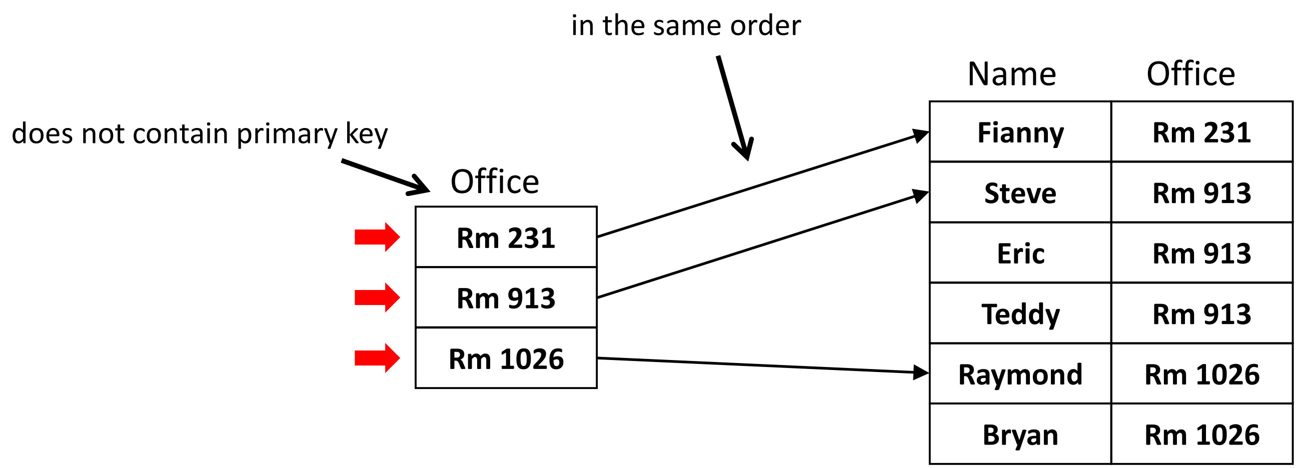


~~Primary~~	Secondary
~~Clustered~~	Unclusterd
~~Dense~~	Sparse

Name	
Rm 231	✓
Rm 913	✗
Rm1026	✓

Index Classification: an Example

- Employee = { Name , Office }

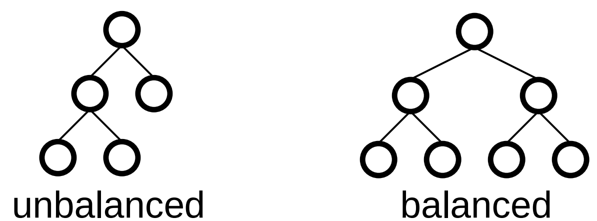


~~Primary~~	Secondary
Clustered	~~Unclusterd~~
Dense	~~Sparse~~

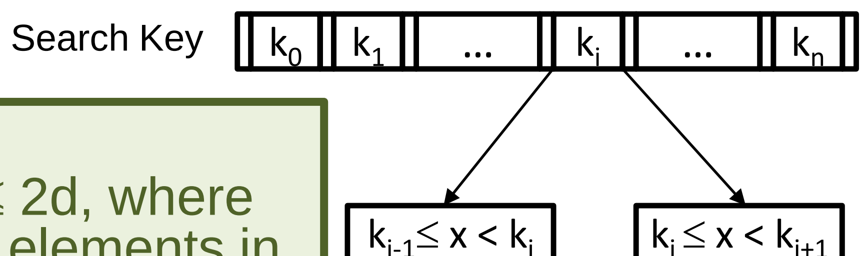
Name
Rm 231 ✓
Rm 913 ✓
Rm1026 ✓

Tree-structured Indexing

- B+ Tree Index Files
  - Balanced Tree – The length of every path from the root to a leaf is the same, maintaining efficient searching



  - Support range search and equality selection

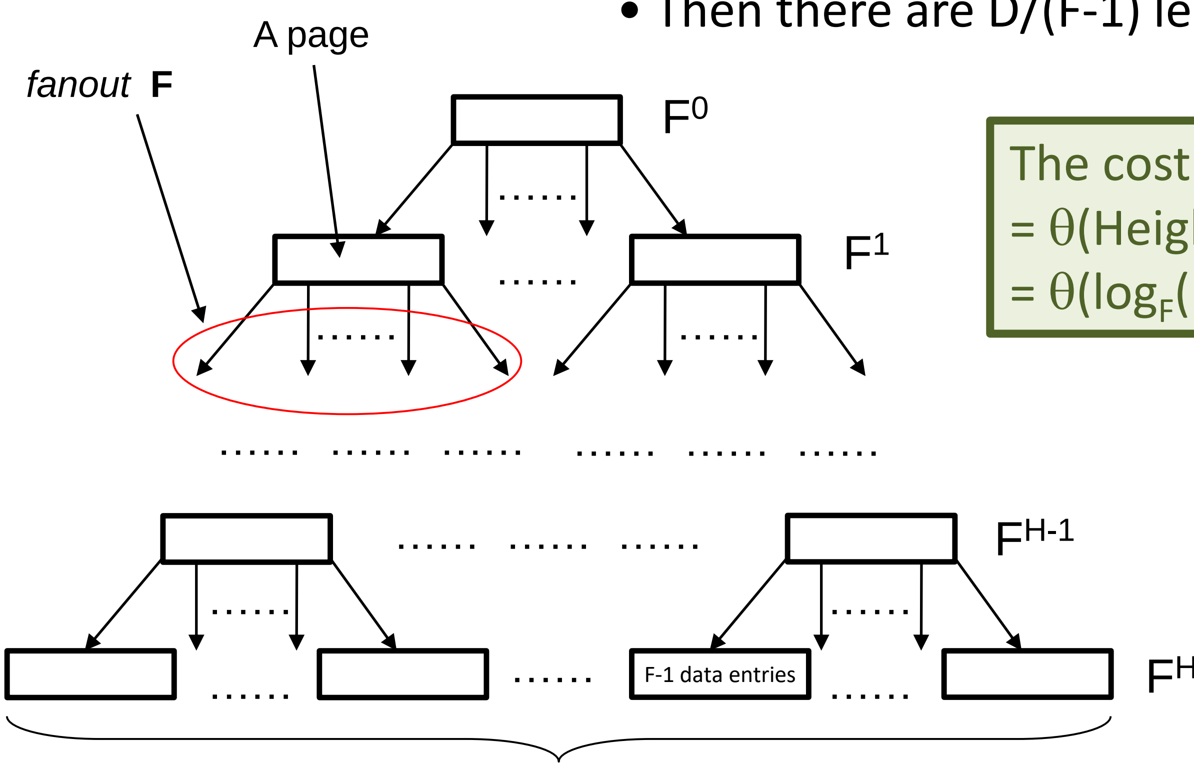


Order: d

- Root node: $1 \leq nr \leq 2d$, where nr is the number of elements in the root node
- Non-root nodes: $d \leq ni \leq 2d$, where ni is the number of elements in any non-root node

Search Cost of B+ Tree

- Let average number of pointers (*fanout*) is **F**
- Each leaf node can at most store F-1 data entries
- Suppose there are totally D data entries
- Then there are D/(F-1) leaf nodes



Totally D Data Entries, D/(F-1) leaves nodes

The cost to reach a leave node:
$= \theta$(Height of tree)
$= \theta(\log_F(D/(F-1)))$

Insertion of B+ Tree

Order 2 B+ Tree
Root node: 1 ≤ # elements ≤ 4
Non-root nodes: 2 ≤ # elements≤ 4

Original Tree

1*	4*	7*	

Insert 13

1*	4*	7*	13*

(No overflow)

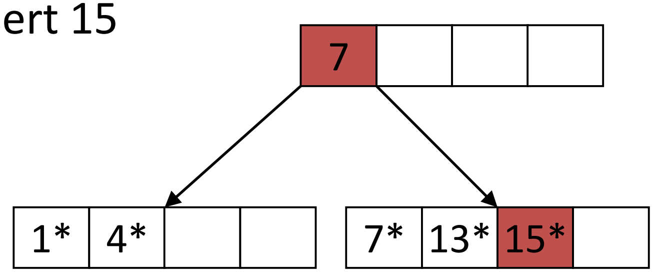
Insert 15

1*	4*	7*	13*	15*

(Overflow)

Leaf node overflow: split leaf node, **copy** middle **key** to the parent.

After Insert 15

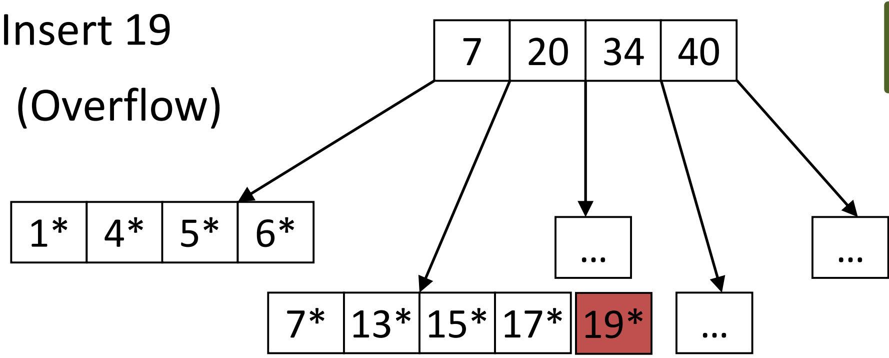


Tree grows

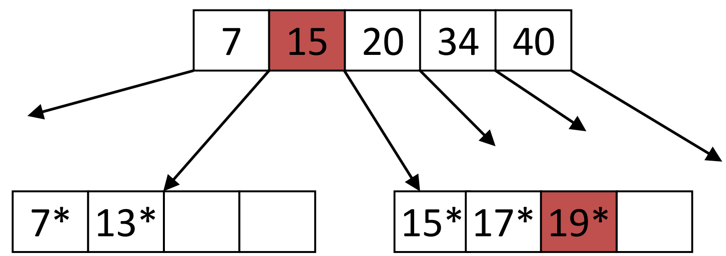
Insertion of B+ Tree

Insert 19

(Overflow)

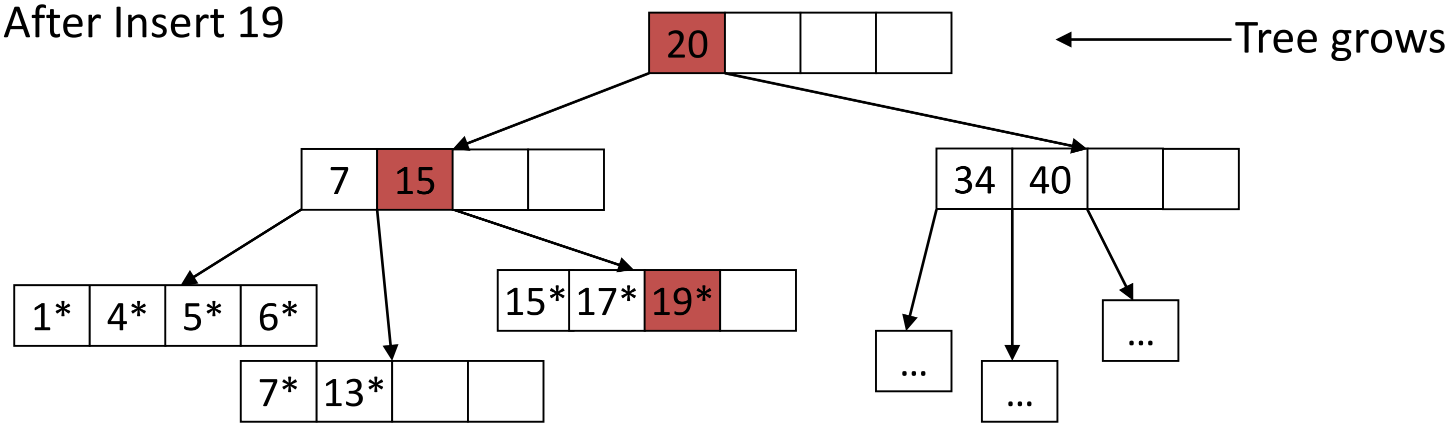


1. Leaf node overflow: split leaf node.



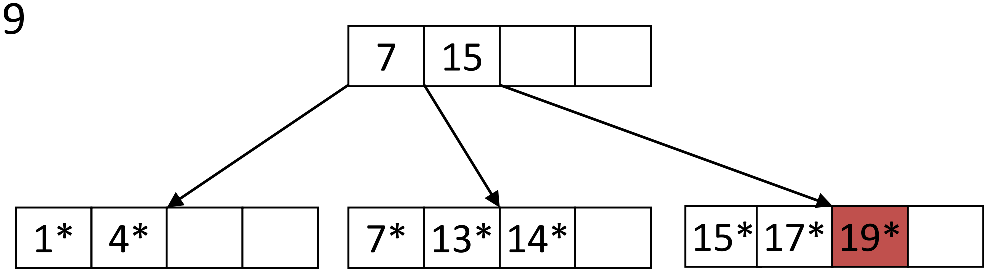
2. Non-leaf node overflow: split non-leaf node, **push** middle key up.

After Insert 19



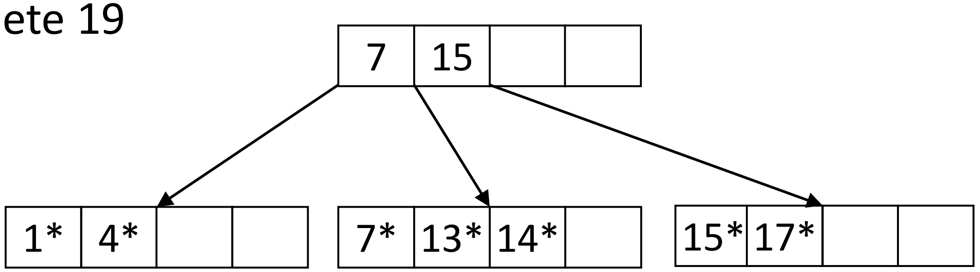
Deletion of B+ Tree

Delete 19



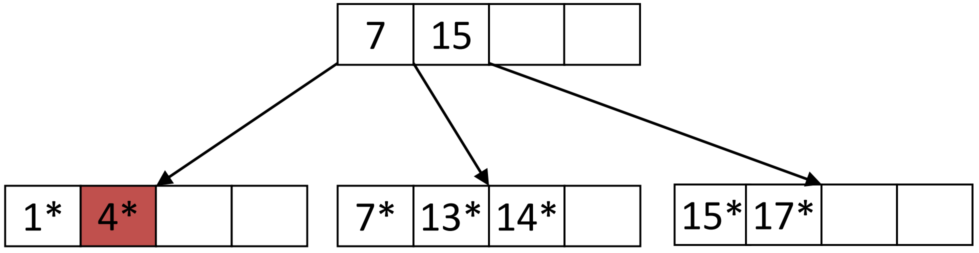
(No underflow)

After Delete 19



Deletion of B+ Tree

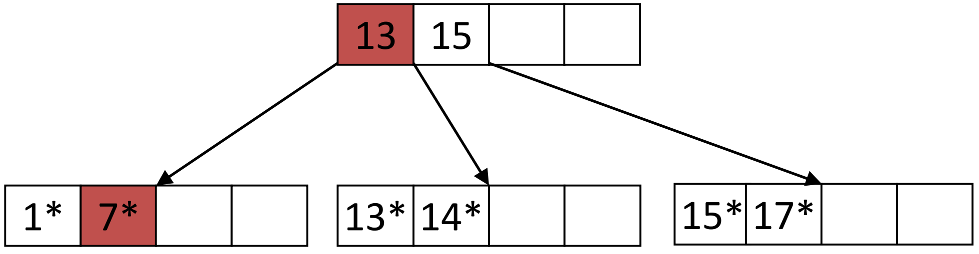
Delete 4



(Underflow)

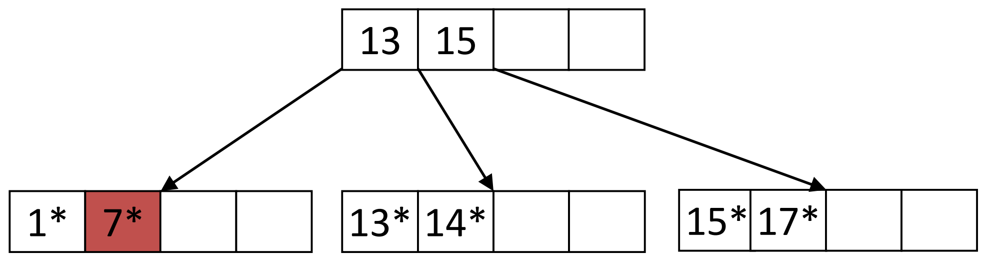
After delete 4, the page will underflow, need to borrow from sibling, update the parent.

After Delete 4



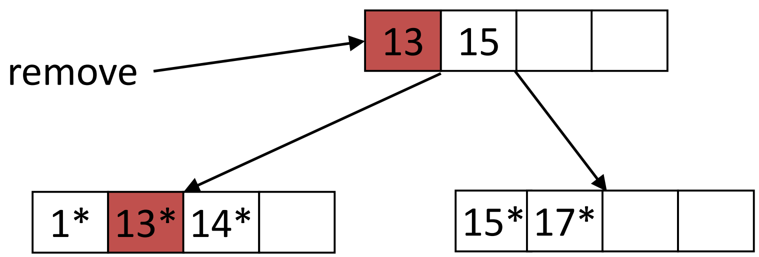
Deletion of B+ Tree

Delete 7

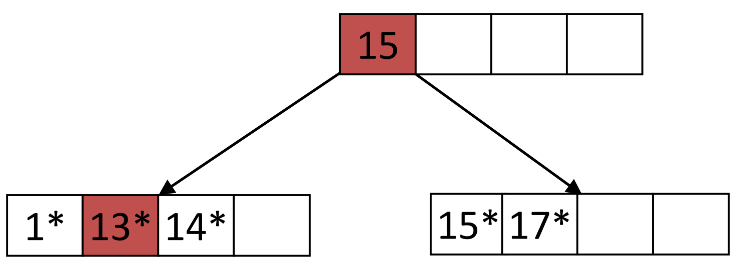


(Underflow)

After delete 7, the page will underflow, can’t borrow from sibling, need to merge with sibling and update the parent.



After Delete 7

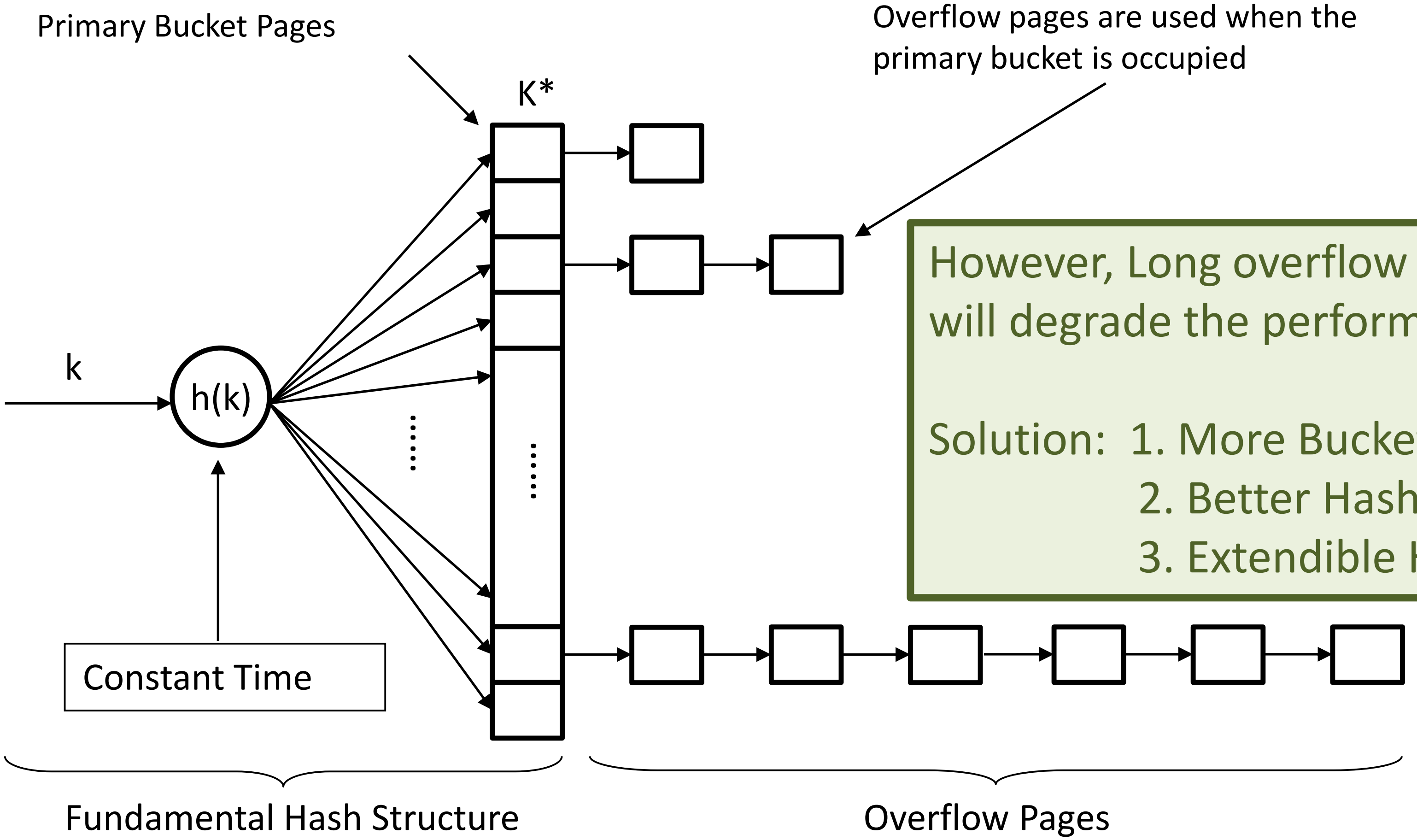


Hashing

- Properties
  - Data entries are kept in bucket.
  - Hashing function h(k) = address of the bucket.
  - h(k) should distribute K* uniformly.
  - Best for equality selection.
  - <u>Not support range search</u>.
  - Constant time retrieval.

Static Hashing

No. of primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.



However, Long overflow chains will degrade the performance

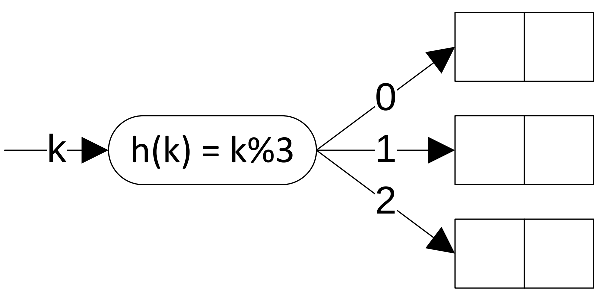
Solution: 1. More Buckets
2. Better Hash Function
3. Extendible Hashing

Static Hashing: an example

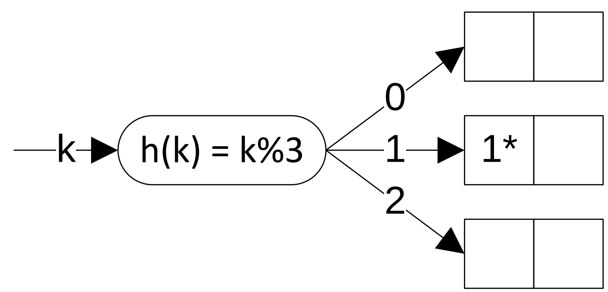
- Suppose the hash function is **h(x) = x mod 3** and a bucket can hold at most 2 data entries. Below show a static hash structure after inserting 1, 4, 5, 7, 8, 2.

h(1) = 1

Insert 1



After insert 1



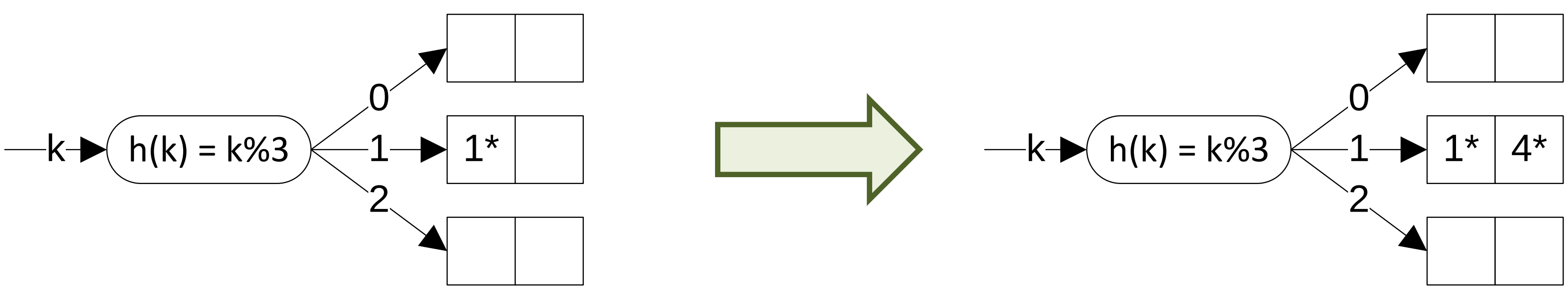
Static Hashing: an example

- Suppose the hash function is **h(x) = x mod 3** and a bucket can hold at most 2 data entries. Below show a static hash structure after inserting 1, 4, 5, 7, 8, 2.

h(4) = 1

Insert 4

After insert 4

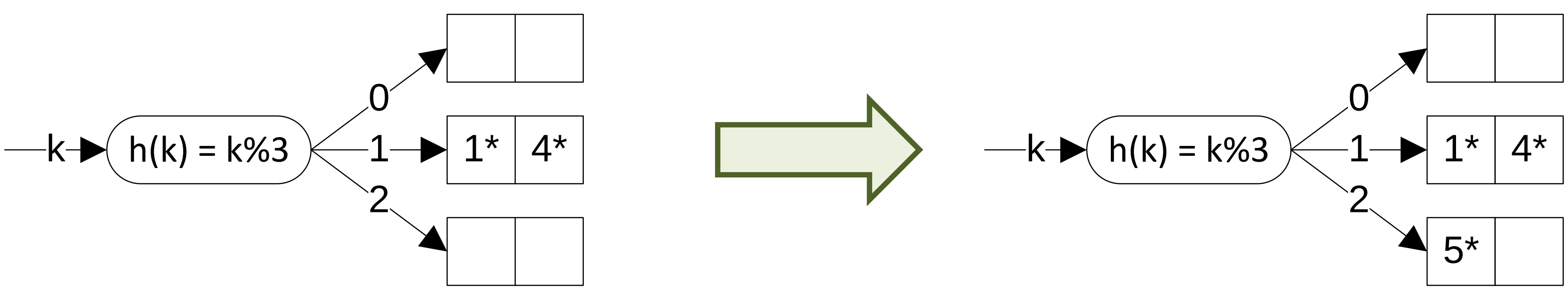


Static Hashing: an example

- Suppose the hash function is **h(x) = x mod 3** and a bucket can hold at most 2 data entries. Below show a static hash structure after inserting 1, 4, 5, 7, 8, 2.

h(5) = 2

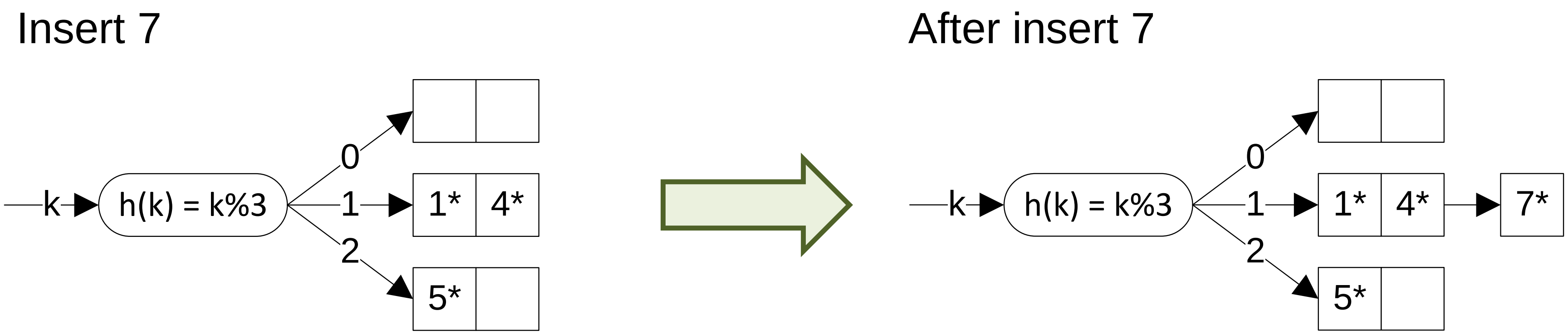
Insert 5 After insert 5



Static Hashing: an example

- Suppose the hash function is **h(x) = x mod 3** and a bucket can hold at most 2 data entries. Below show a static hash structure after inserting 1, 4, 5, 7, 8, 2.

h(7) = 1

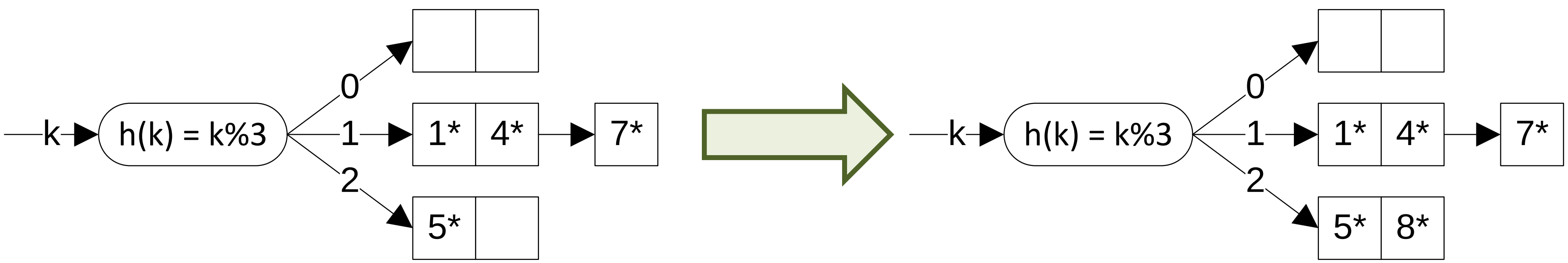


Static Hashing: an example

- Suppose the hash function is **h(x) = x mod 3** and a bucket can hold at most 2 data entries. Below show a static hash structure after inserting 1, 4, 5, 7, 8, 2.

h(8) = 2

Insert 8 After insert 8



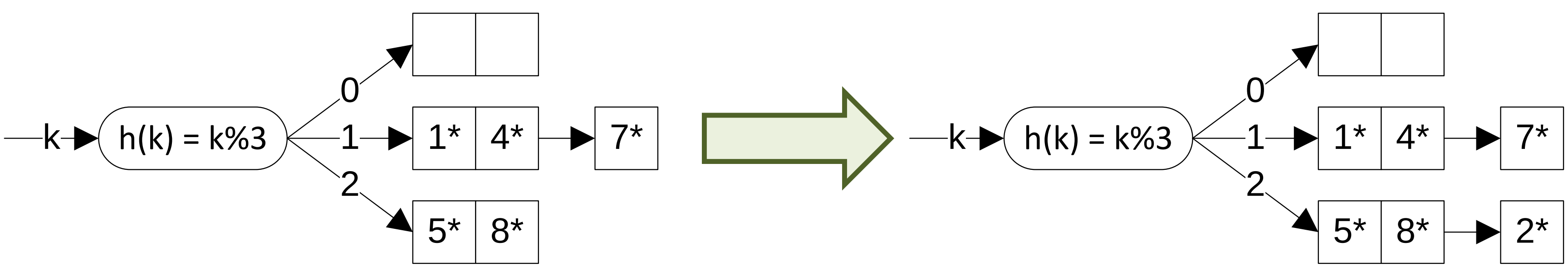
Static Hashing: an example

- Suppose the hash function is **h(x) = x mod 3** and a bucket can hold at most 2 data entries. Below show a static hash structure after inserting 1, 4, 5, 7, 8, 2.

h(2) = 2

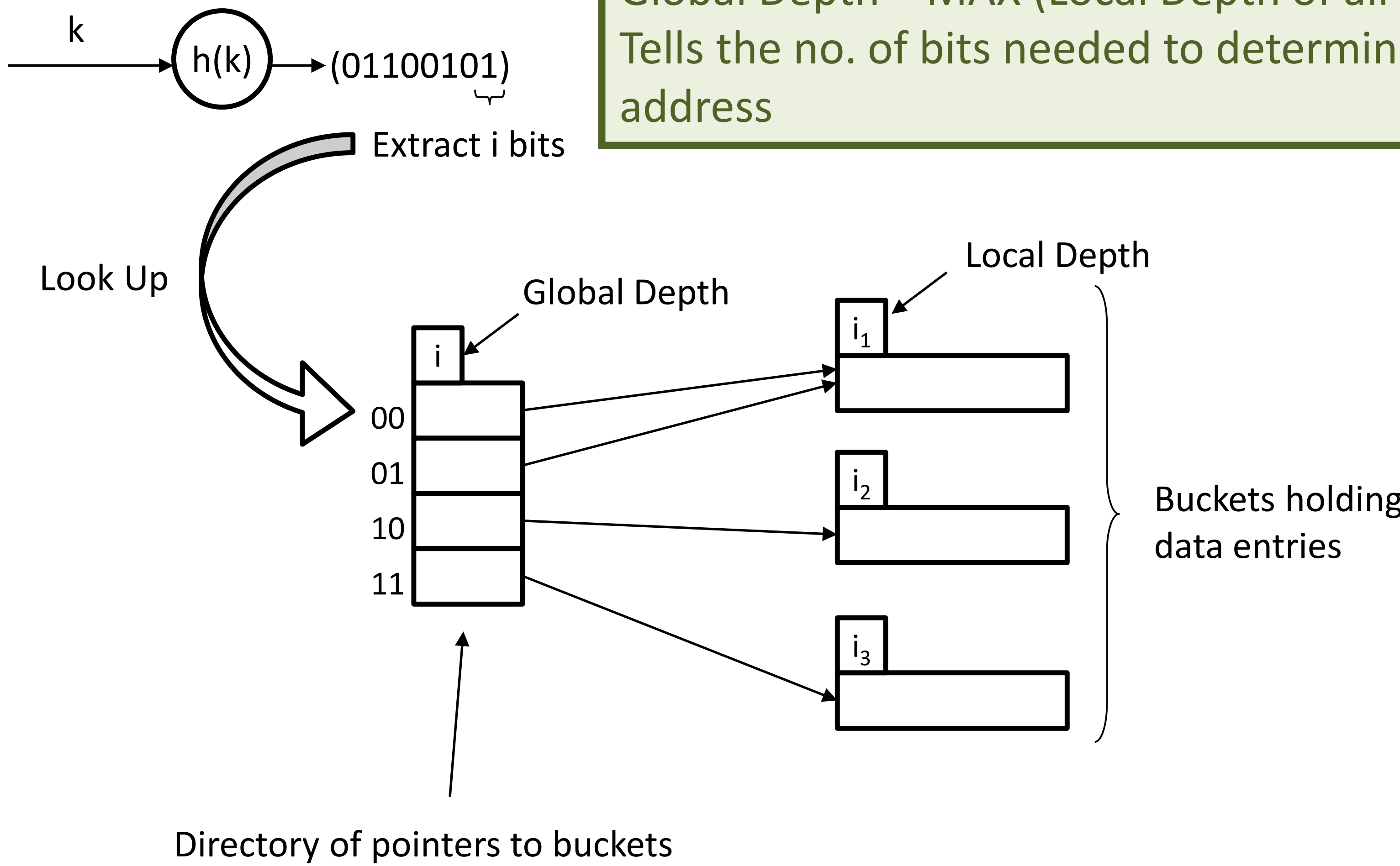
Insert 2

After insert 2



Extendible Hashing

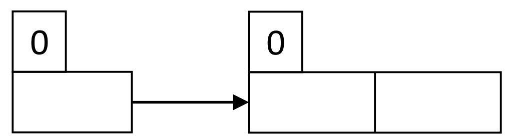
Global Depth = MAX (Local Depth of all buckets)
Tells the no. of bits needed to determine the address



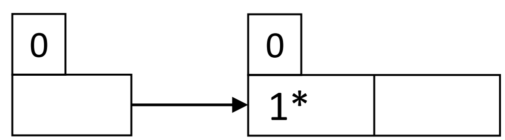
Insertion of Extendible Hashing

- Suppose the hash function is h(x) = x mod 8 and each bucket can hold at most 2 data entries. Below show a extendable hash structure after inserting 1, 4, 5, 7, 8, 2.

Initial:



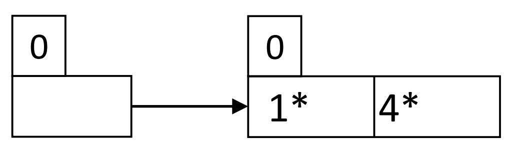
After insert 1:



(ok)

h(1) = 001

After insert 4:

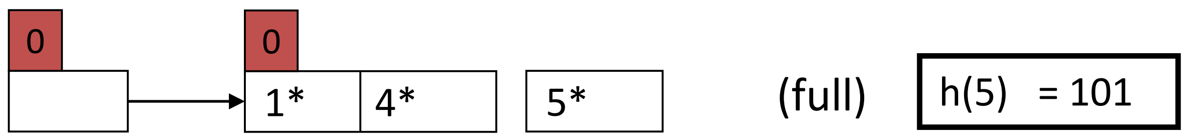


(ok)

h(1) = 001
h(4) = 100

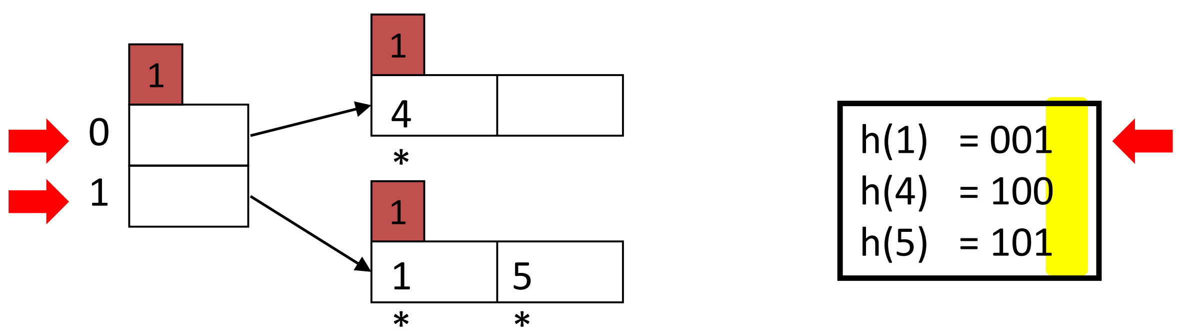
Insertion of Extendible Hashing

Insert 5:



If bucket with local depth = global depth, then double the directory and split the bucket.

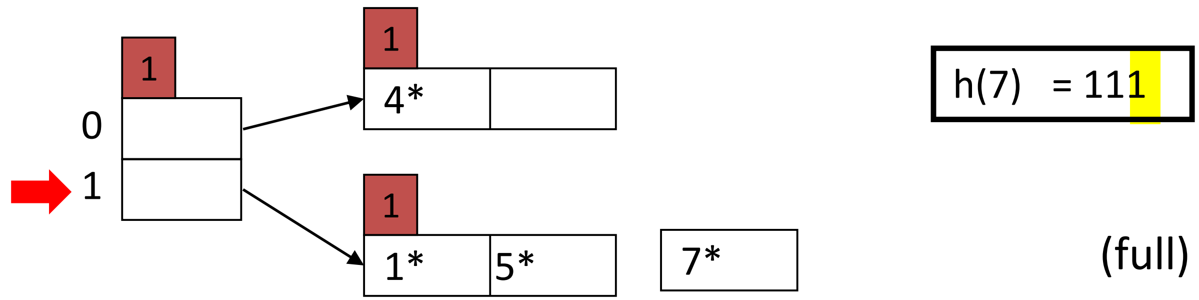
After insert 5:



Increase the global depth and local depth of the new buckets by 1.

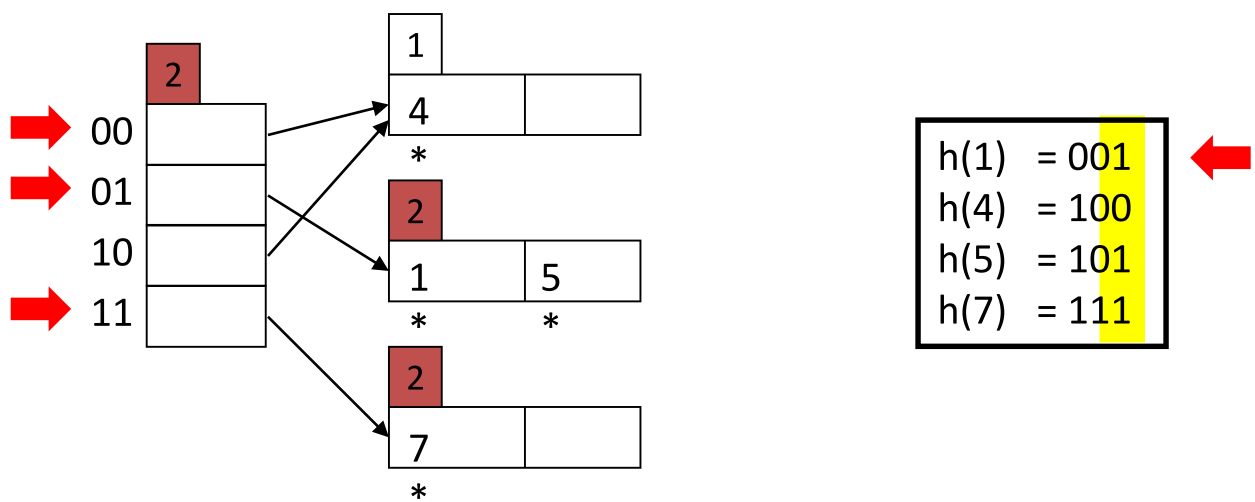
Insertion of Extendible Hashing

Insert 7:



If bucket with local depth = global depth, then double the directory and split the bucket.

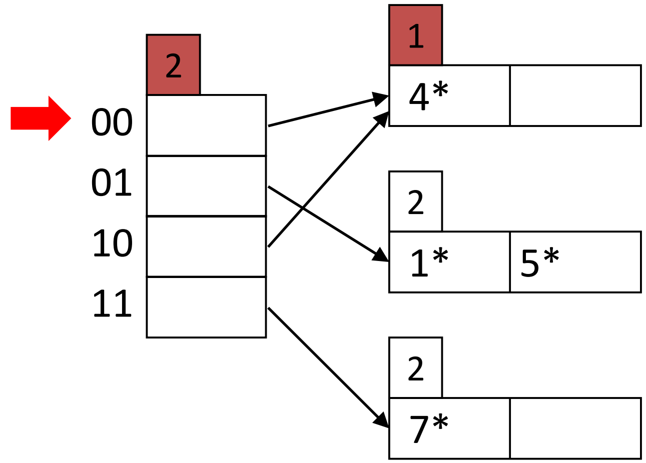
After insert 7:



Increase the global depth and local depth of the new buckets by 1.

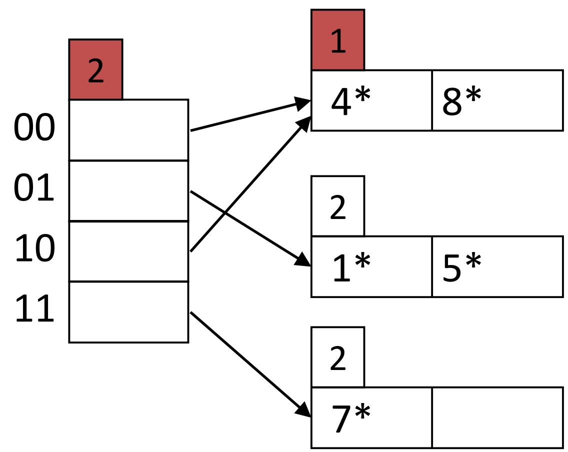
Insertion of Extendible Hashing

Insert 8:



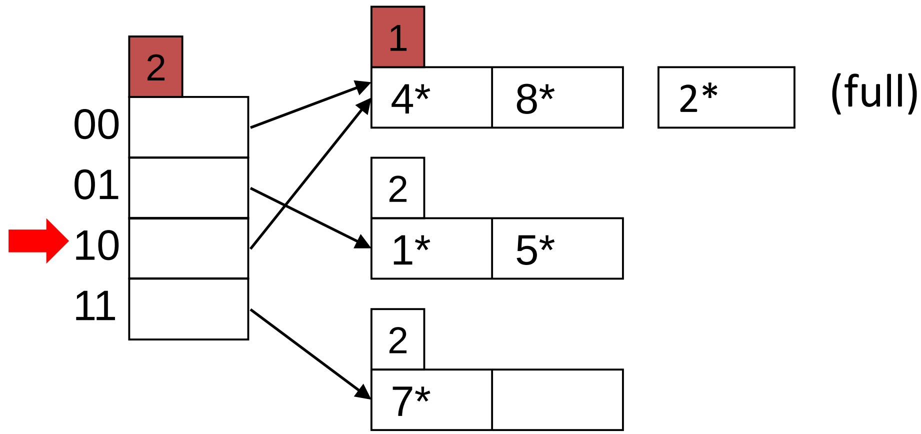
h(8) = 000

After insert 8:



Insertion of Extendible Hashing

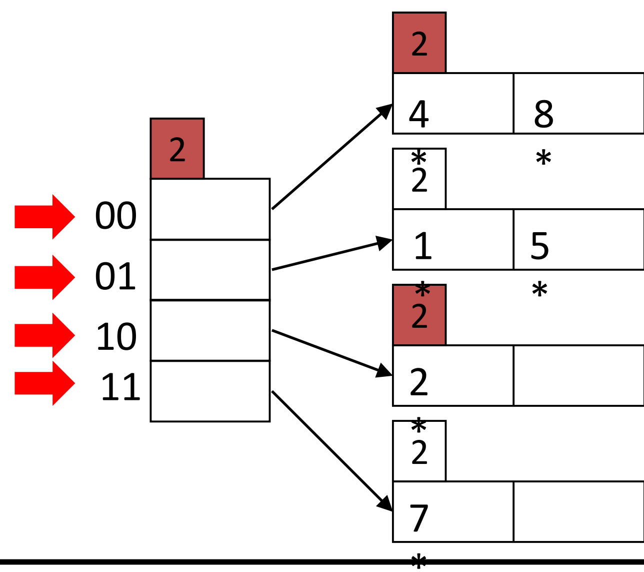
Insert 2:



h(2) = 010

If bucket with local depth < global depth, then split the bucket and update the pointers.

After insert 2:

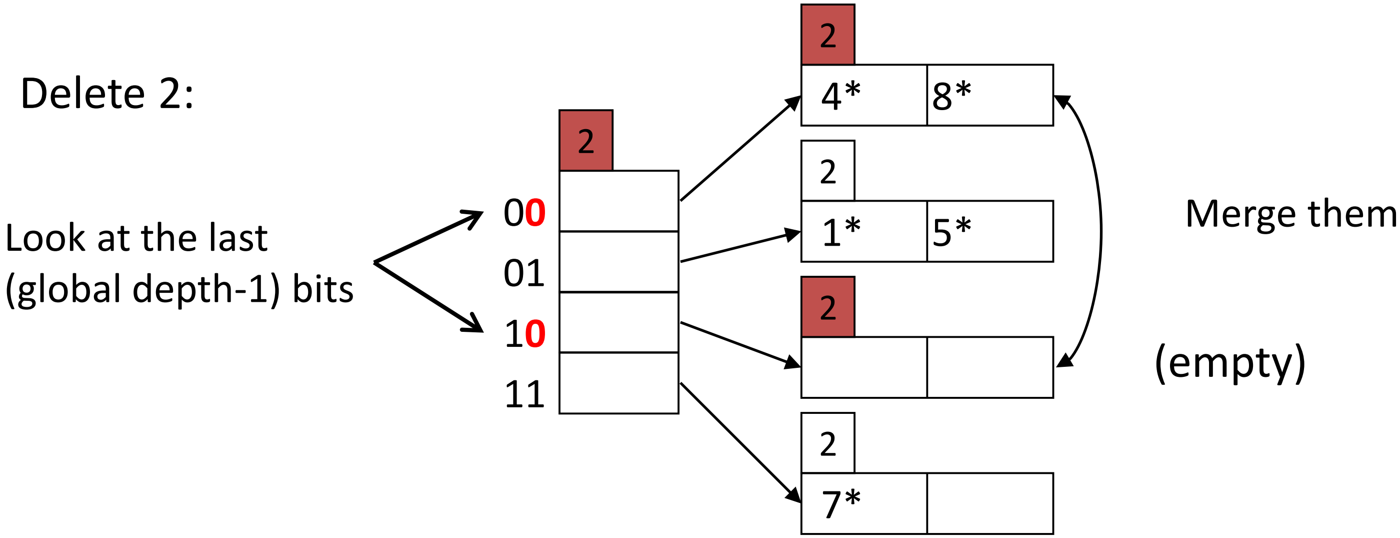


h(1) = 0001
h(2) = 0010
h(4) = 0100
h(5) = 0101
h(7) = 0111
h(8) = 1000

Increase the local depth of the new buckets by 1.

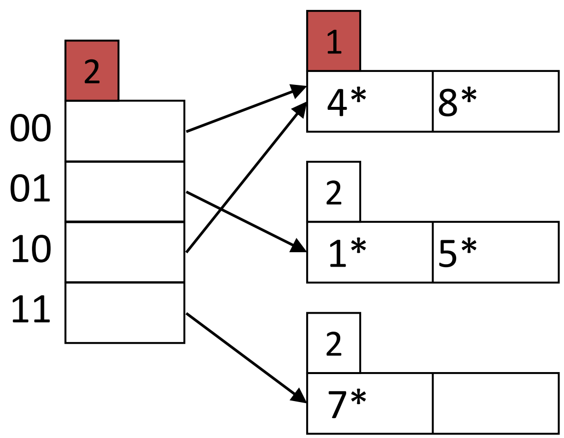
Deletion of Extendible Hashing

Delete 2:



If the removal makes the bucket empty, then remove the bucket and update the pointer.

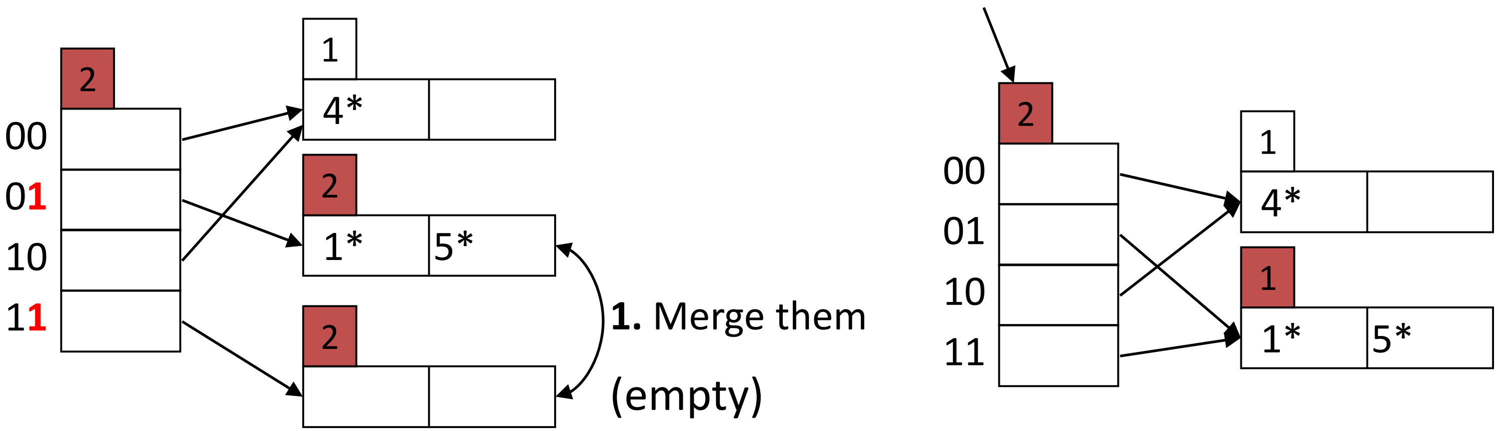
After delete 2:



Decrease the local depth of the new bucket by 1.

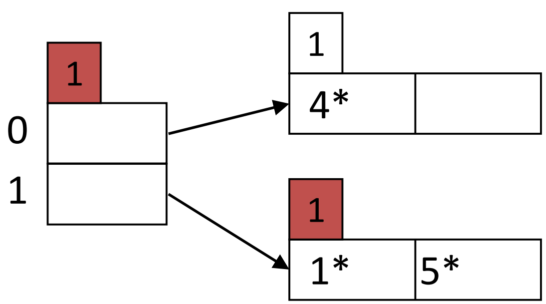
Deletion of Extendible Hashing

Delete 7:



If the mergence makes max(local depth of all buckets) < global depth, then halve the size of directory.

After delete 7:



Decrease the global depth of the new bucket by 1.