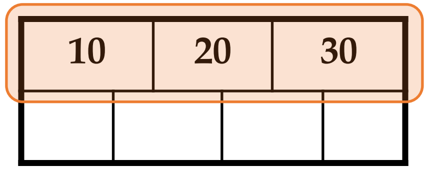
Lecture 13: B+ Tree (continued)

What you will learn about in this section

1. Recap: B+ Trees
2. B+ Trees: Cost
3. B+ Trees: Clustered

1. Recap: B+ Trees

B+ Tree Basics



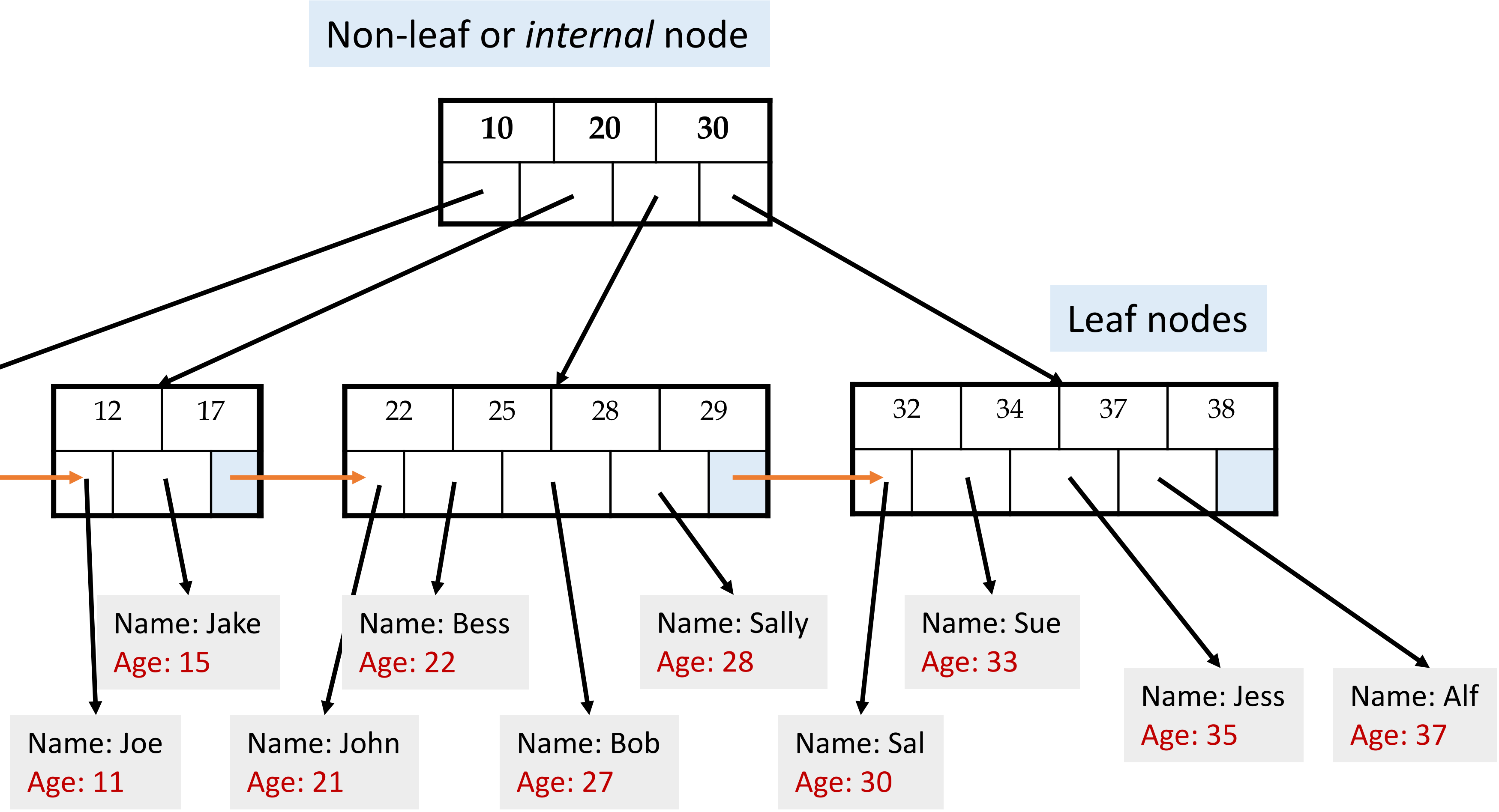
Parameter ***d*** = the order

Each *non-leaf ("interior")* ***node*** has d $\leq m \leq$ 2d ***entries***

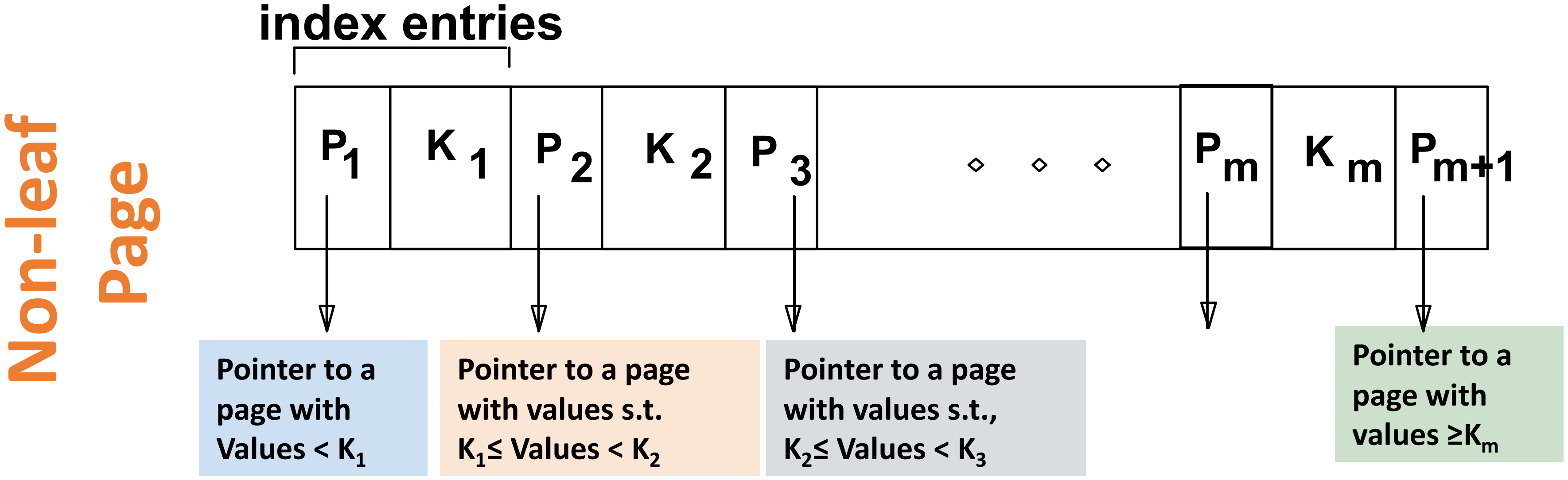
- ***Minimum 50% occupancy***

Root ***node*** has 1 $\leq m \leq$ 2d ***entries***

B+ Tree Basics



B+ Tree Page Format

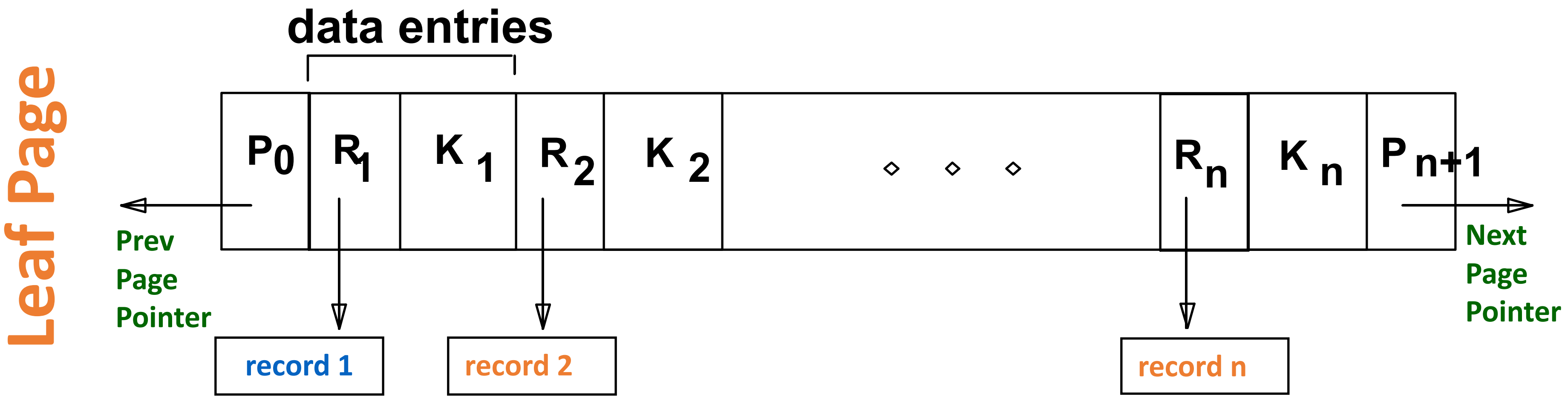


Pointer to a page with Values < K_1

Pointer to a page with values s.t. $K_1 \leq$ Values < K_2

Pointer to a page with values s.t., $K_2 \leq$ Values < K_3

Pointer to a page with values $\geq K_m$



B+ Tree: Search

- start from root
- examine index entries in non-leaf nodes to find the correct child
- traverse down the tree until a leaf node is reached
- non-leaf nodes can be searched using a binary or a linear search

B+ Tree: Insert

- Find correct leaf *L.*
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must ***split*** *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split non-leaf node, redistribute entries evenly, but **pushing up** the middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

B+ Tree: Deleting a data entry

- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to **re-distribute**, borrowing from <u>*sibling*</u> *(adjacent node with same parent as L)*.
    - If re-distribution fails, ***merge*** *L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could **propagate** to root, decreasing height.

2. B+ Trees: Cost

B+ Tree: High Fanout = Smaller & Lower IO

- As compared to e.g. binary search trees, B+ Trees have **high *fanout*** (***between d+1 and 2d+1***)
- This means that the **depth of the tree is small** → getting to any element requires very few IO operations!
  - Also can often store most or all of the B+ Tree in main memory!
- A TiB = 2^{40} Bytes. What is the height of a B+ Tree (with fill-factor = 1) that indexes it (with 64K pages)?
  - $(2*2730 + 1)^h = 2^{40}$ → ***h = 4***

The fanout is defined as the number of pointers to child nodes coming out of a node

Note that fanout is dynamic- we'll often assume it's constant just to come up with approximate eqns!

Simple Cost Model for Search

- Let:
  - ***f*** = fanout, which is **in [d+1, 2d+1]** ***(we'll assume it's constant for our cost model…)***
  - ***N*** = the total number of *pages* we need to index
  - ***F*** = fill-factor (usually ~= 2/3)

- Our B+ Tree needs to have room to index ***N / F*** pages!
  - We have the fill factor in order to leave some open slots for faster insertions

- What height (*h*) does our B+ Tree need to be?
  - h=1 → Just the root node- room to index f pages
  - h=2 → f leaf nodes- room to index f^2 pages
  - h=3 → f^2 leaf nodes- room to index f^3 pages
  - …
  - h → f^{h-1} leaf nodes- room to index f^h pages!

→ We need a B+ Tree of height h = $\left\lceil \log_f \frac{N}{F} \right\rceil$!

Simple Cost Model for Search

- Note that if we have ***B*** available buffer pages, by the same logic:
  - We can store $\boldsymbol{L_B}$ levels of the B+ Tree in memory
  - where $\boldsymbol{L_B}$ ***is the number of levels such that the sum of all the levels' nodes fit in the buffer:***
    - $B \geq 1 + f + \cdots + f^{L_B - 1} = \sum_{l=0}^{L_B - 1} f^l$

- In summary: to do exact search:
  - We read in one page per level of the tree
  - However, levels that we can fit in buffer are free!
  - Finally we read in the actual record

IO Cost: $\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + 1$

where $B \geq \sum_{l=0}^{L_B - 1} f^l$

Simple Cost Model for Search

- To do range search, we just follow the horizontal pointers

- The IO cost is that of loading additional leaf nodes we need to access + the IO cost of loading each ***page*** of the results- we phrase this as "Cost(OUT)"

IO Cost: $\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + Cost(OUT)$

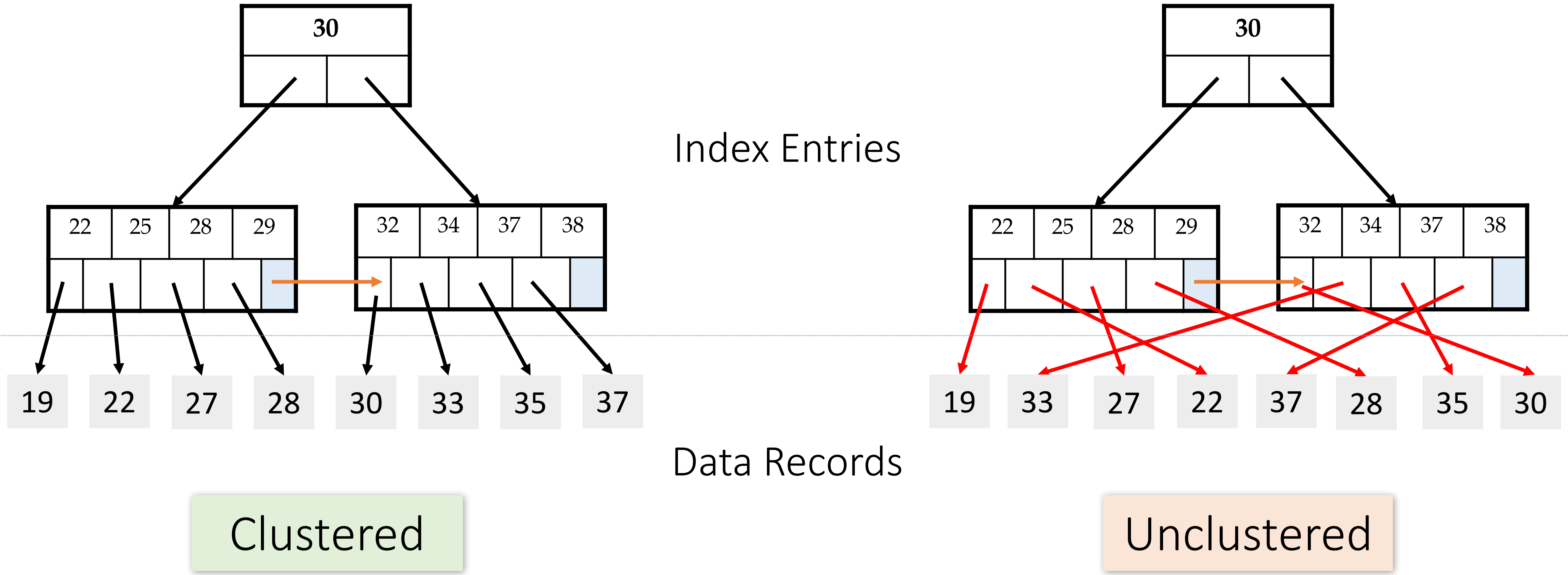
where $B \geq \sum_{l=0}^{L_B - 1} f^l$

3. B+ Trees: Clustered

Clustered Indexes

An index is ***clustered*** if the underlying data is ordered in the same way as the index's data entries.

Clustered vs. Unclustered Index



Index Entries

Data Records

Clustered vs. Unclustered Index

- Recall that for a disk with block access, **sequential IO is much faster than random IO**

- For exact search, no difference between clustered / unclustered

- For range search over R values: difference between **1 random IO + R sequential IO**, and **R random IO**:
  - A random IO costs ~ 10ms (sequential much much faster)
  - For R = 100,000 records- **difference between ~10ms and ~17min!**

Summary

- We create **indexes** over tables in order to support ***fast (exact and range) search*** and ***insertion*** over ***multiple search keys***

- **B+ Trees** are one index data structure which support very fast exact and range search & insertion via ***high fanout***
  - ***Clustered vs. unclustered*** makes a big difference for range queries too

Lecture 14: Hash Indexes

What you will learn about in this section

1. Hash Indexes

2. Static Hashing

3. Extendible Hashing

1. Hash Indexes

Hash Index

- A hash index is a collection of buckets
  - bucket = primary page plus overflow pages
  - buckets contain one or more data entries

- uses a hash function **h**
  - ***h****(r)* = bucket in which (data entry for) record *r* belongs

Hash Index

- A hash index is:
  - good for equality search
  - not so good for range search (use tree indexes instead)

- Types of hash indexes:
  - **Static** hashing
  - **Extendible** hashing (dynamic)
  - Linear hashing (dynamic) – not covered in the course, see 11.3 in the cow book

Operations on Hash Indexes

- **Equality search**
    - apply the hash function on the search key to locate the appropriate bucket
    - search through the primary page (plus overflow pages) to find the record(s)
- **Deletion**
    - find the appropriate bucket, delete the record
- **Insertion**
    - find the appropriate bucket, insert the record
    - if there is no space, create a new overflow page

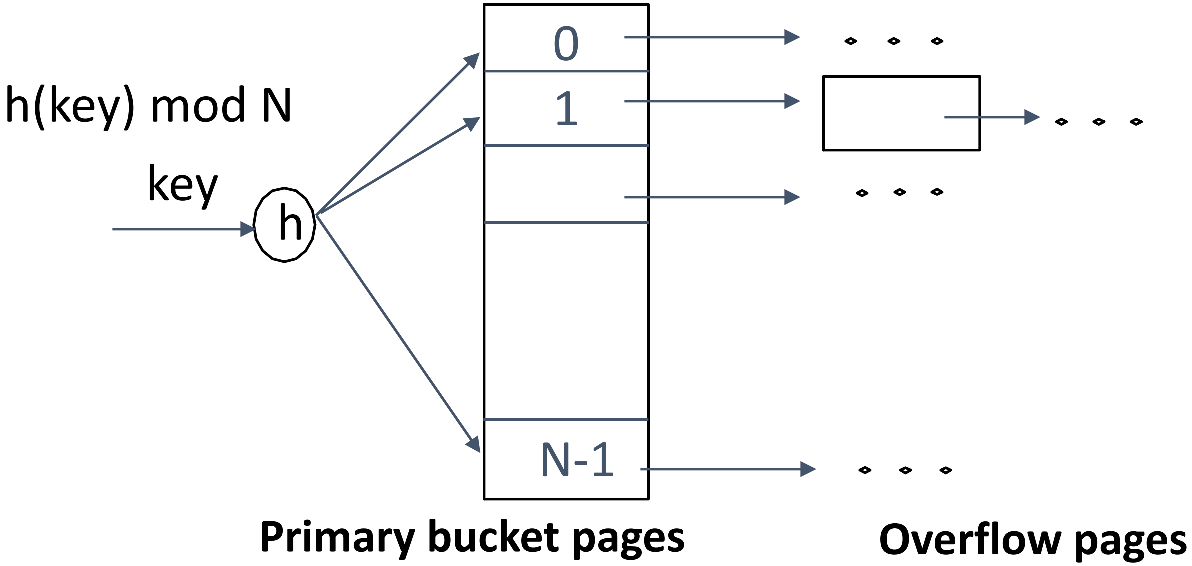
Hash Functions

- An *ideal* hash function must be uniform: each bucket is assigned the same number of key values
- A *bad* hash function maps all search key values to the same bucket
- Examples of good hash functions:
  - $\boldsymbol{h}(k) = a * k + b$, where a and b are constants
  - a random function

2. Static Hashing

Static Hashing

- # primary bucket pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- **h(*k*) mod N** = bucket to which data entry with key *k* belongs. (N = # of buckets)

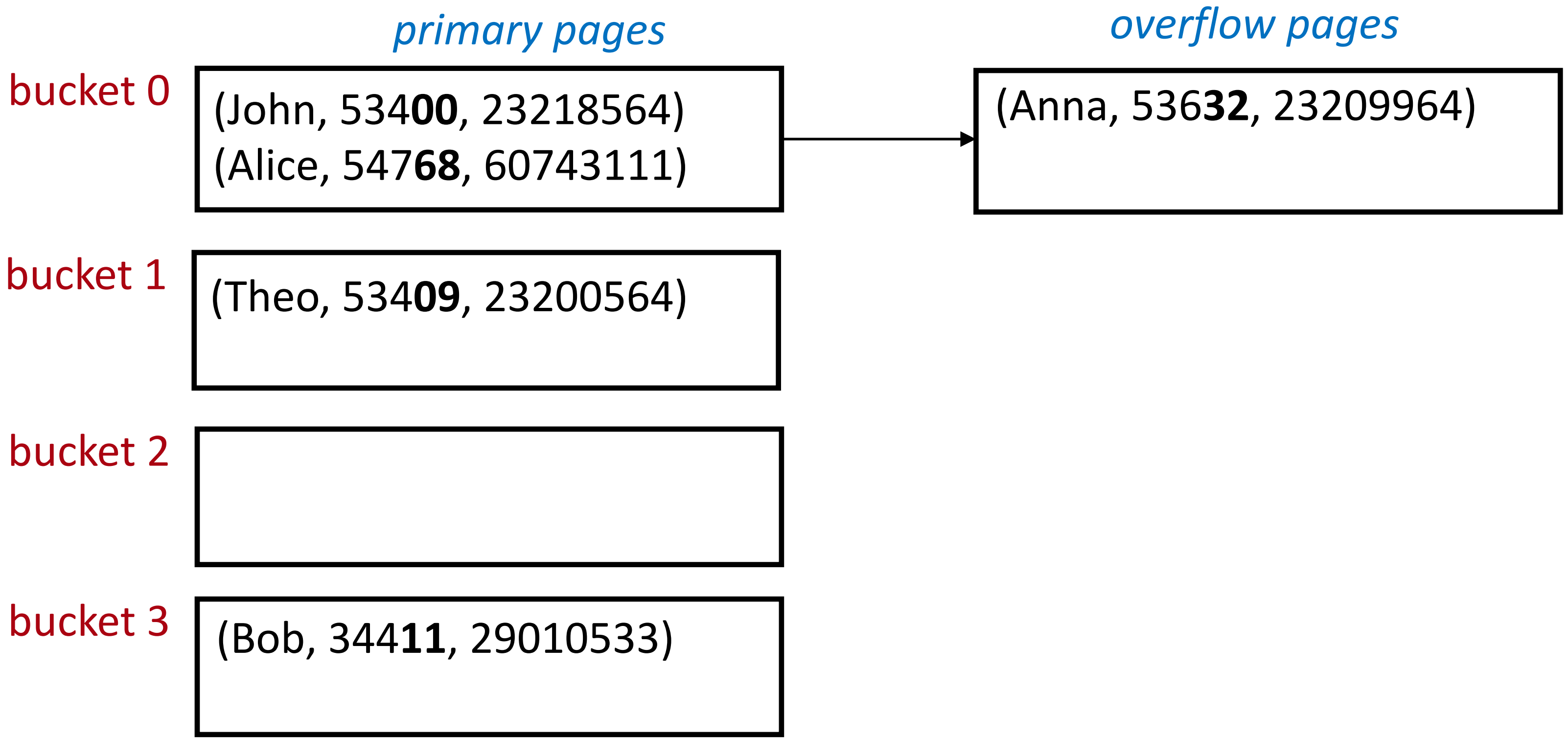


Static Hashing: Example

`**Person**(name,zipcode,phone)`

- *search key*: zipcode
- *hash function **h***: last 2 digits

- 4 buckets
- each bucket has 2 data entries (full record)



primary pages

overflow pages

bucket 0

(John, 534**00**, 23218564)
(Alice, 547**68**, 60743111)

(Anna, 536**32**, 23209964)

bucket 1

(Theo, 534**09**, 23200564)

bucket 2

bucket 3

(Bob, 344**11**, 29010533)

Hash Functions

- An *ideal* hash function must be uniform: each bucket is assigned the same number of key values
- A *bad* hash function maps all search key values to the same bucket
- Examples of good hash functions:
  - $\mathbf{h}(k) = a * k + b$, where a and b are constants
  - a random function

Bucket Overflow

- Bucket *overflow* can occur because of
  - insufficient number of buckets
  - *skew* in distribution of records
    - many records have the same search-key value
    - the hash function results in a non-uniform distribution of key values

- Bucket overflow is handled using *overflow buckets*

Problems of Static Hashing

- In static hashing, there is a **fixed** number of buckets in the index
- Issues with this:
  - if the database grows, the number of buckets will be too small: long overflow chains degrade performance
  - if the database shrinks, space is wasted
  - reorganizing the index is expensive and can block query execution

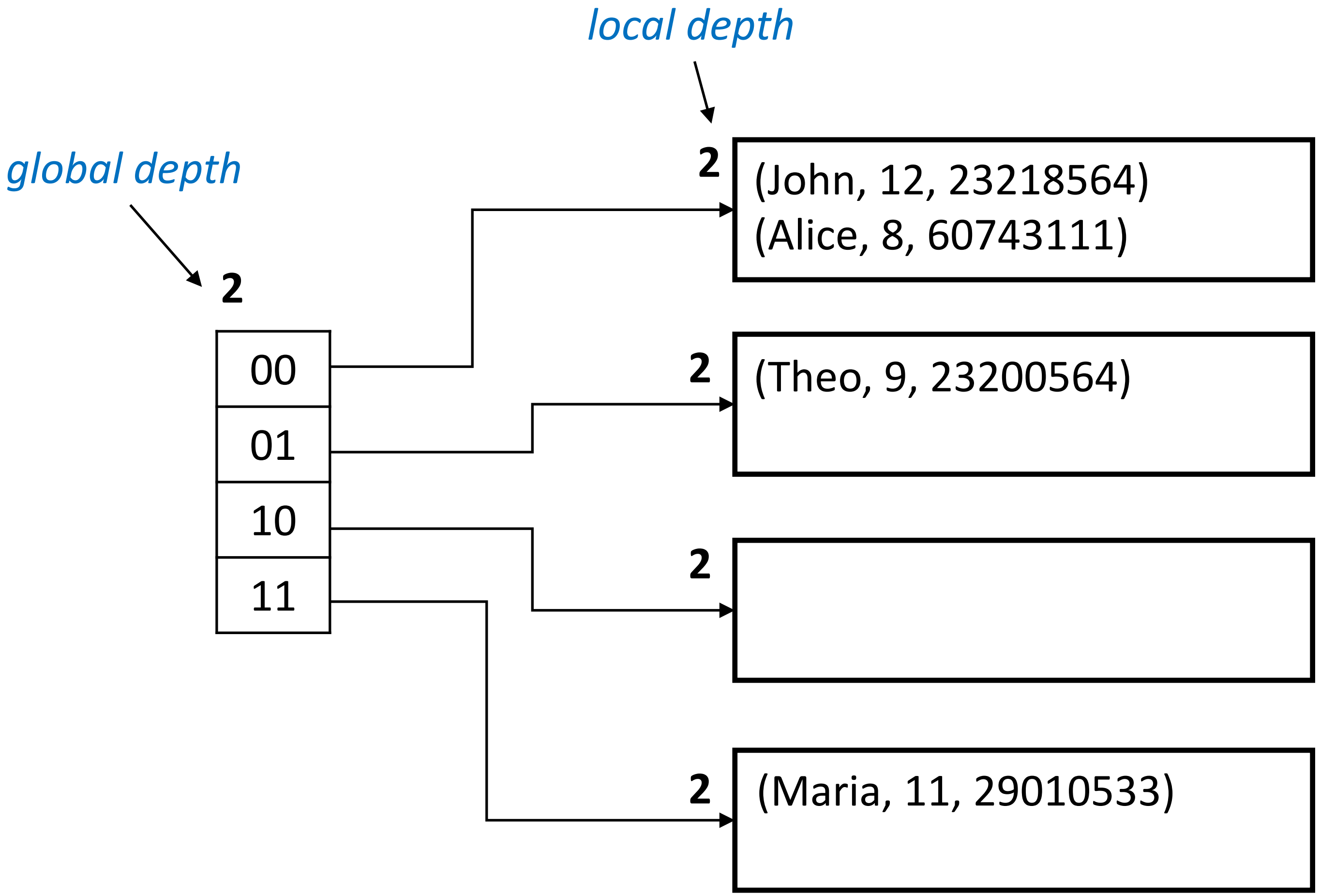
3. Extendible Hashing

Extendible Hashing

- **Extendible hashing** is a type of *dynamic* hashing
- It keeps a directory of pointers to buckets
- On overflow, it reorganizes the index by doubling the directory (and not the number of buckets)

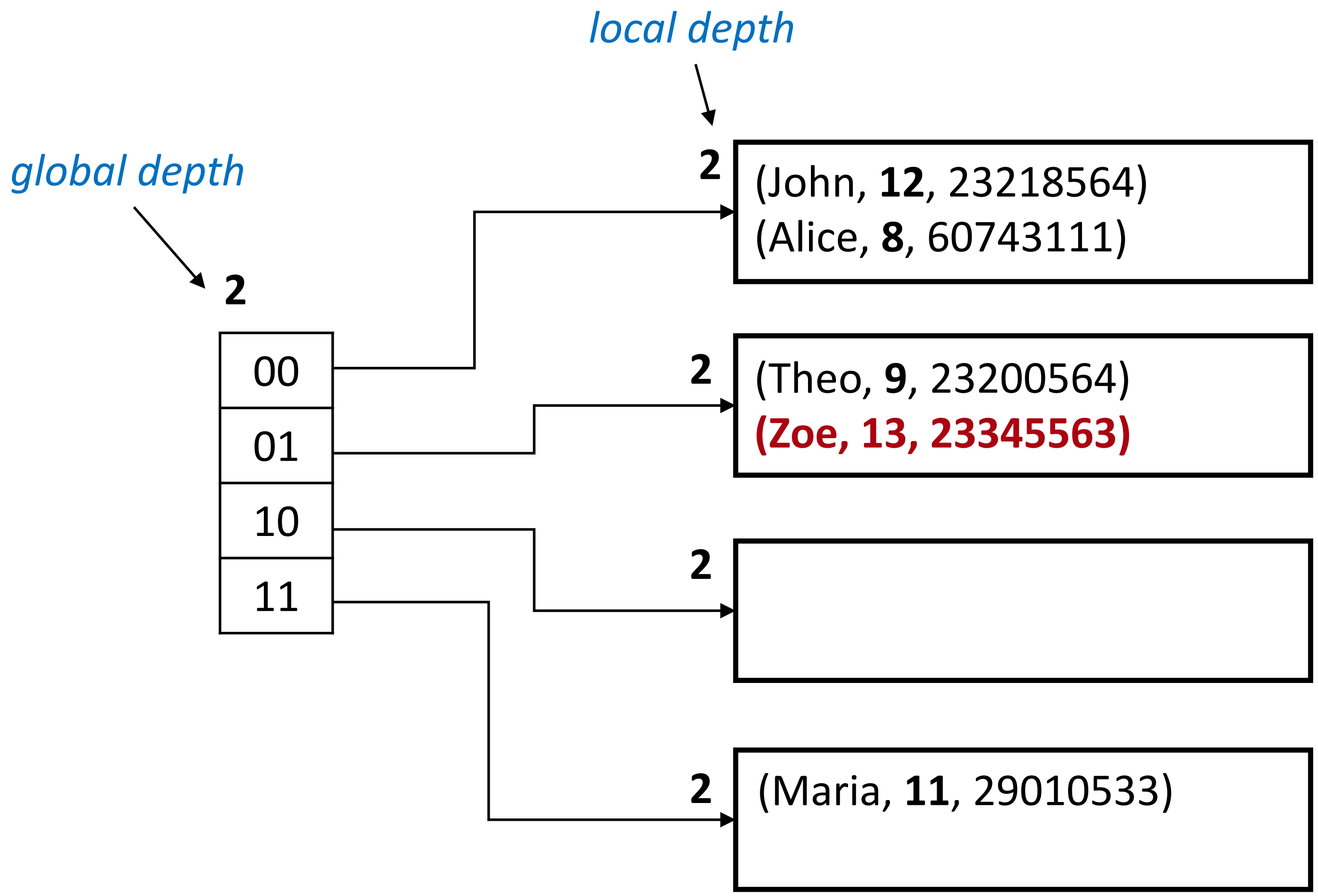
Extendible Hashing

To search, use the last **2** digits of the **binary** form of the search key value



Extendible Hashing: Insert

If there is space in the bucket, simply add the record



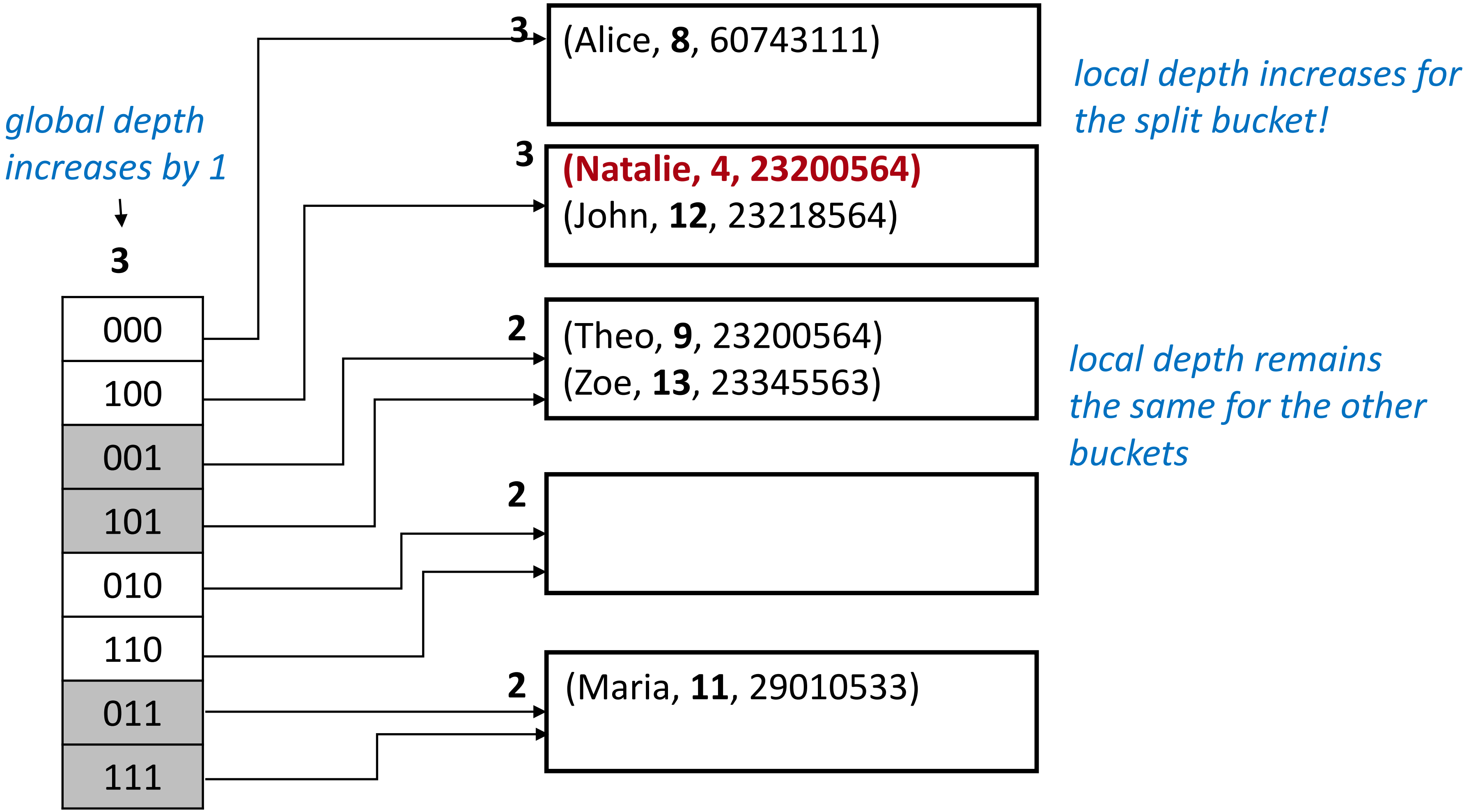
(John, **12**, 23218564)
(Alice, **8**, 60743111)

(Theo, **9**, 23200564)
(Zoe, 13, 23345563)

(Maria, **11**, 29010533)

Extendible Hashing: Insert

If the bucket is full, split the bucket and redistribute the entries



Extendible Hashing: Delete

- Locate the bucket of the record and remove it
- If the bucket becomes empty, it can be removed (and update the directory)
- Two buckets can also be coalesced together if the sum of the entries fit in a single bucket
- Decreasing the size of the directory can also be done, but it is expensive

More on Extendible Hashing

- How many disk accesses for equality search?
    - One if directory fits in memory, else two
- Directory grows in spurts, and, if the distribution of hash values is skewed, the directory can grow very large
- We may need overflow pages when multiple entries have the same hash