CS-300: Data-Intensive Systems

Hashing & Sorting

(Chapters 14.5, 15.4, 24.5)

Prof. Anastasia Ailamaki, Prof. Sanidhya Kashyap

EPFL

Today's focus

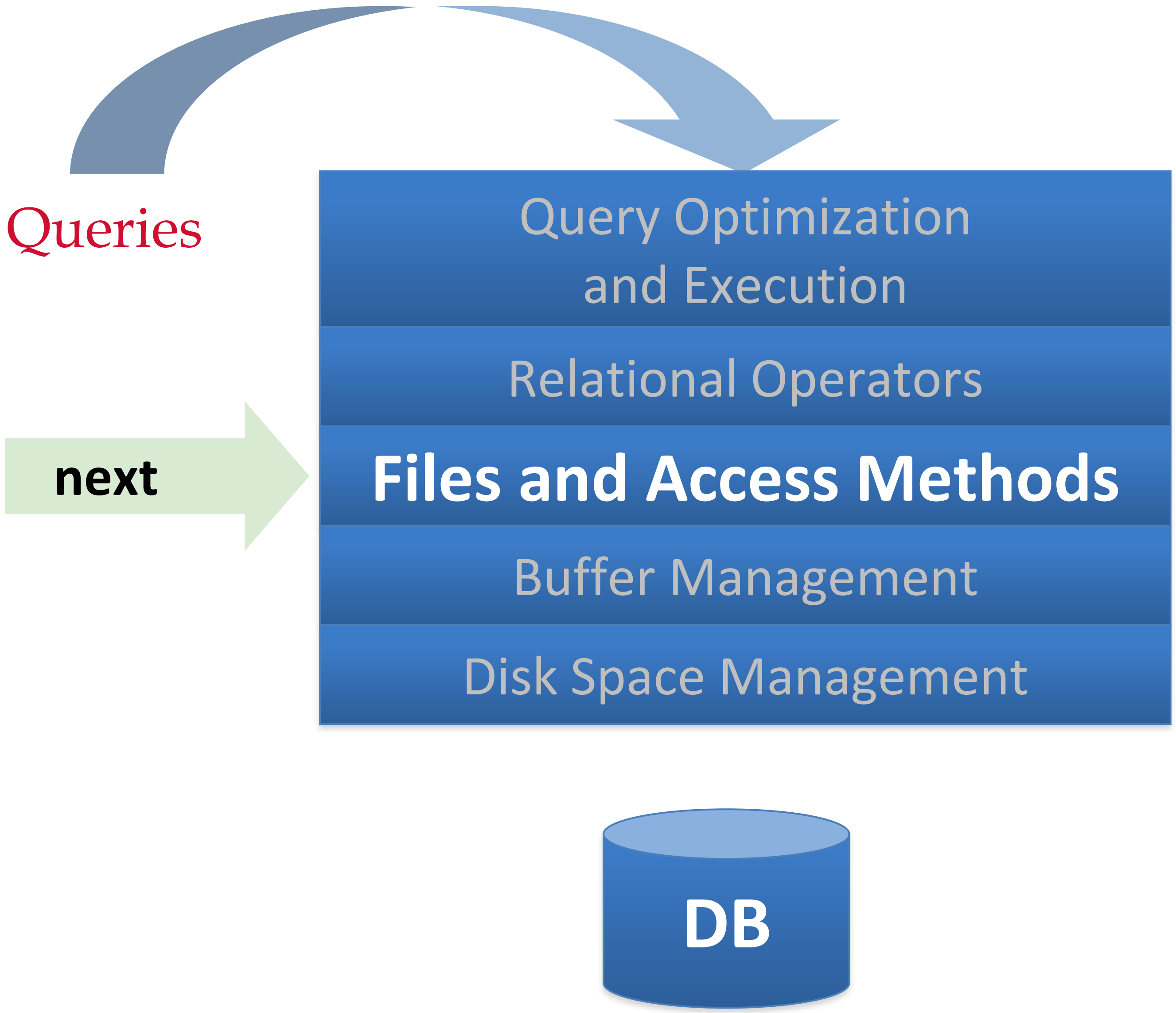
- **Hash-based indexes**
- Sorting

DBMS big picture

Support DBMS execution engine to read/write data from pages!

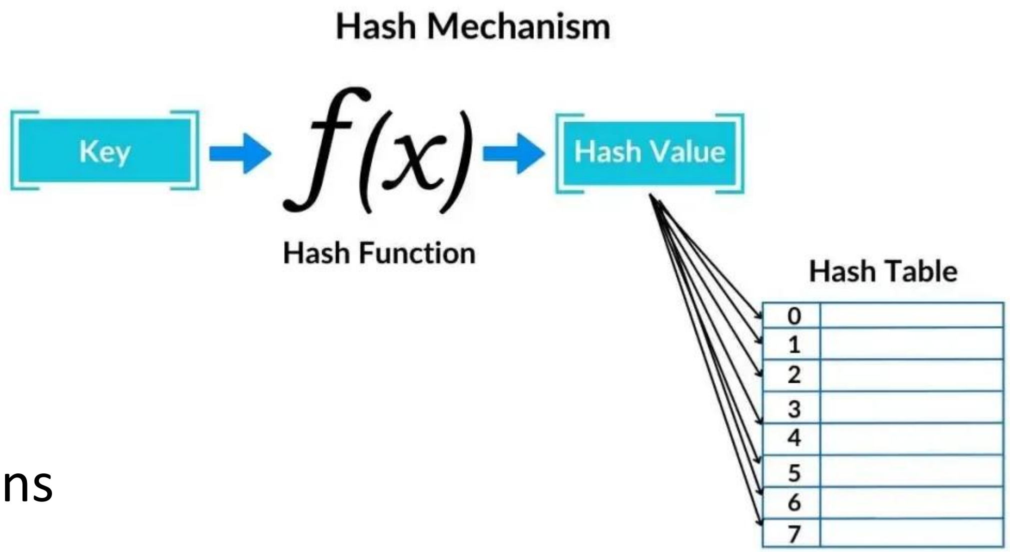
Two types of data structures:

1. Trees (ordered)
2. **Hash tables (unordered)**



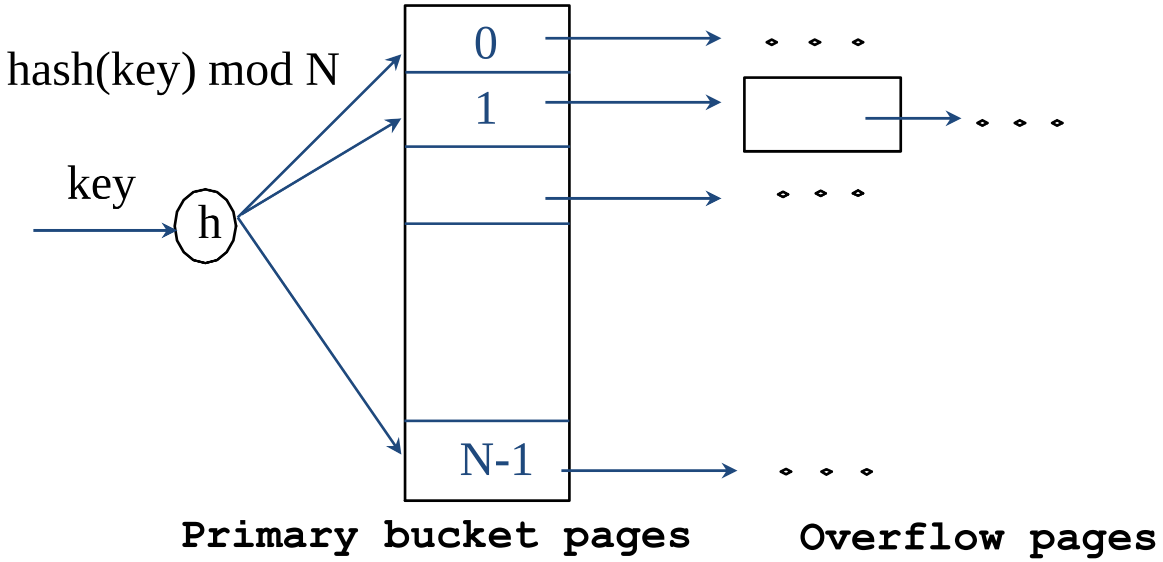
Hash tables

- A **hash table** implements an ***unordered*** associative array that maps keys to values
- It uses a **hash function** to compute an offset into this array for a given key, from which the desired value can be found
- Space complexity: **O(n)**
- Time complexity:
  - Average: **O(1)**
  - Worst: **O(n)**
- Why study hashing?
  - Beneficial if you have only equality selections
  - Very useful in join implementations



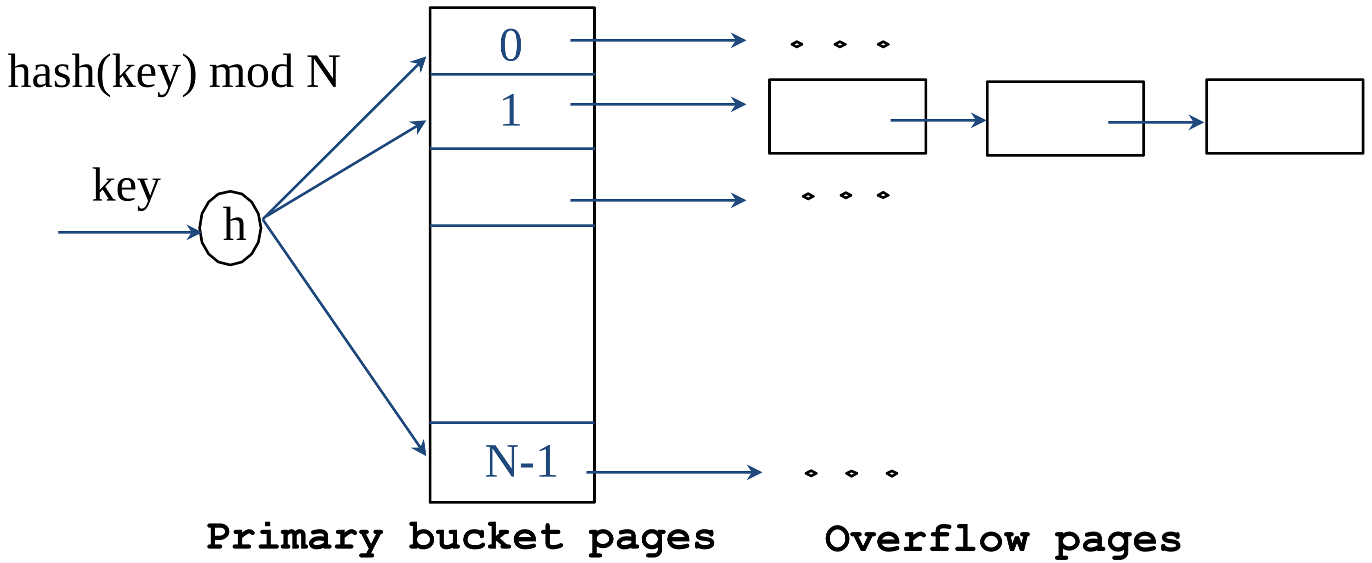
Static hash table

- Hash file is a collection of buckets
- Bucket is a collection of pages
  - 1 primary page and possible one or more overflow pages
- hash(key) % n → bucket to which data entry with key **key** belongs (n → # of buckets)



Static hash table

- N is fixed, primary pages allocated sequentially, never de-allocated; overflow pages if needed.
- Long overflow chains can develop and degrade performance.
    - *Extendible* and *Linear Hashing* fix this problem.



Unrealistic assumptions

- Assumption #1: Number of elements is known ahead of time and fixed

- Assumption #2: Each key is unique

- Assumption #3: Perfect hash function guarantees no collision
  - If **key1 != key2** then

    hash (key1) != hash (key2)

Hash tables

- **Design decision #1: Hash function**
  - Accepts a (fixed- or variable-length) value as input and produces a fixed-sized value output which (ideally) uniquely represents the input
  - Objective: map a large key space into a smaller domain
  - Trade-off between being fast vs. collision rate

- **Design decision #2: Hashing scheme**
  - How to handle key collisions after hashing
  - Trade-off between allocating a large hash table vs. additional instructions to get/put keys

Hashing

- Hash functions
- Static hashing schemes
- Dynamic hashing schemes

Hash functions

- For any input key, return an integer representation of the key
  - Hash function works on search **key** field of record r
  - % n distributes values over range 0 .. n - 1
  - hash(key) = (a * key + b) usually works well; a and b are constants
- Hashing should be **fast** and have a **low collision rate**

- Known hash functions:
  - CRC-64: Used in networking for error detection
  - MurmurHash: fast, general-purpose hash function
  - CityHash: for strings, faster for short keys (<64 bytes)
  - XXHash: very fast parallel hashing

Static hashing schemes

- **Approach #1: Linear probe hashing**

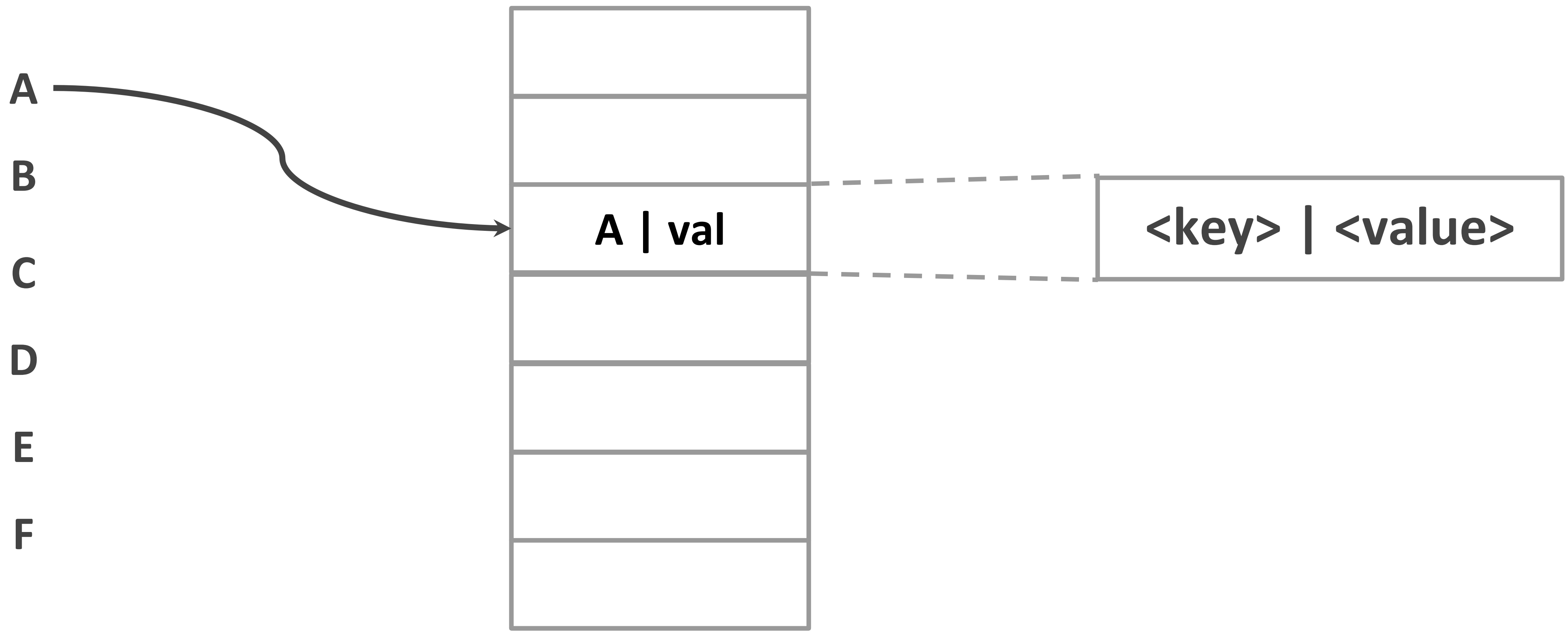
- Several other schemes exist:
  - Cuckoo hashing
  - Hopscotch hashing
  - Robin hood hashing
  - Swiss tables

Linear probe hashing

- A method of **open addressing** (aka **closed hashing**) collision resolution
  - Search through alternative locations in the array (the *probe sequence*) until either the target or an unused array slot is found (search key does not exist),
- **Quadratic probing**: interval between probes increases linearly (eg a quadratic function)
- **Double hashing**: fixed search interval but computed by another hash function.
- **Linear probing**: search in fixed intervals (eg =1): Single giant table of slots
  - Resolve collisions by linearly searching for the next free slot in the table
  - To determine presence of an element, hash to a location in the index and scan for it
  - Must store the key in the index to know when to stop scanning
  - Insertions and deletions are generalizations of lookup

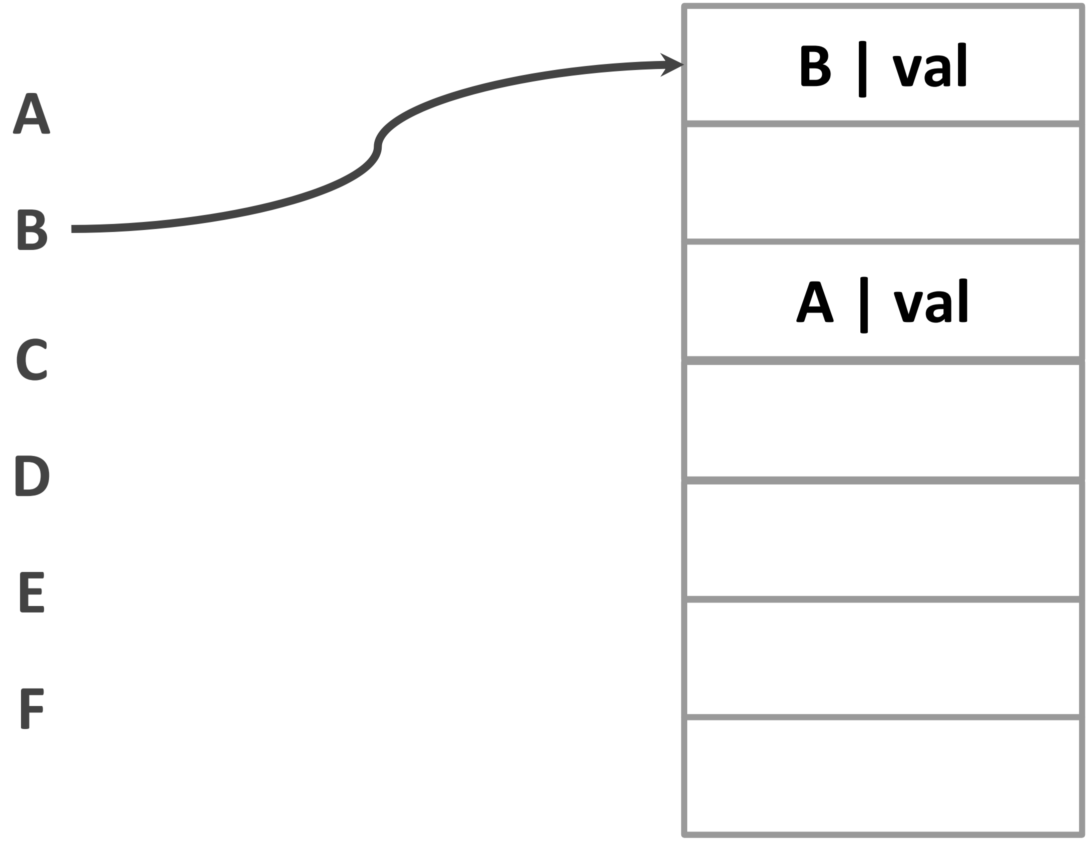
Linear probe hashing

hash(key) % n

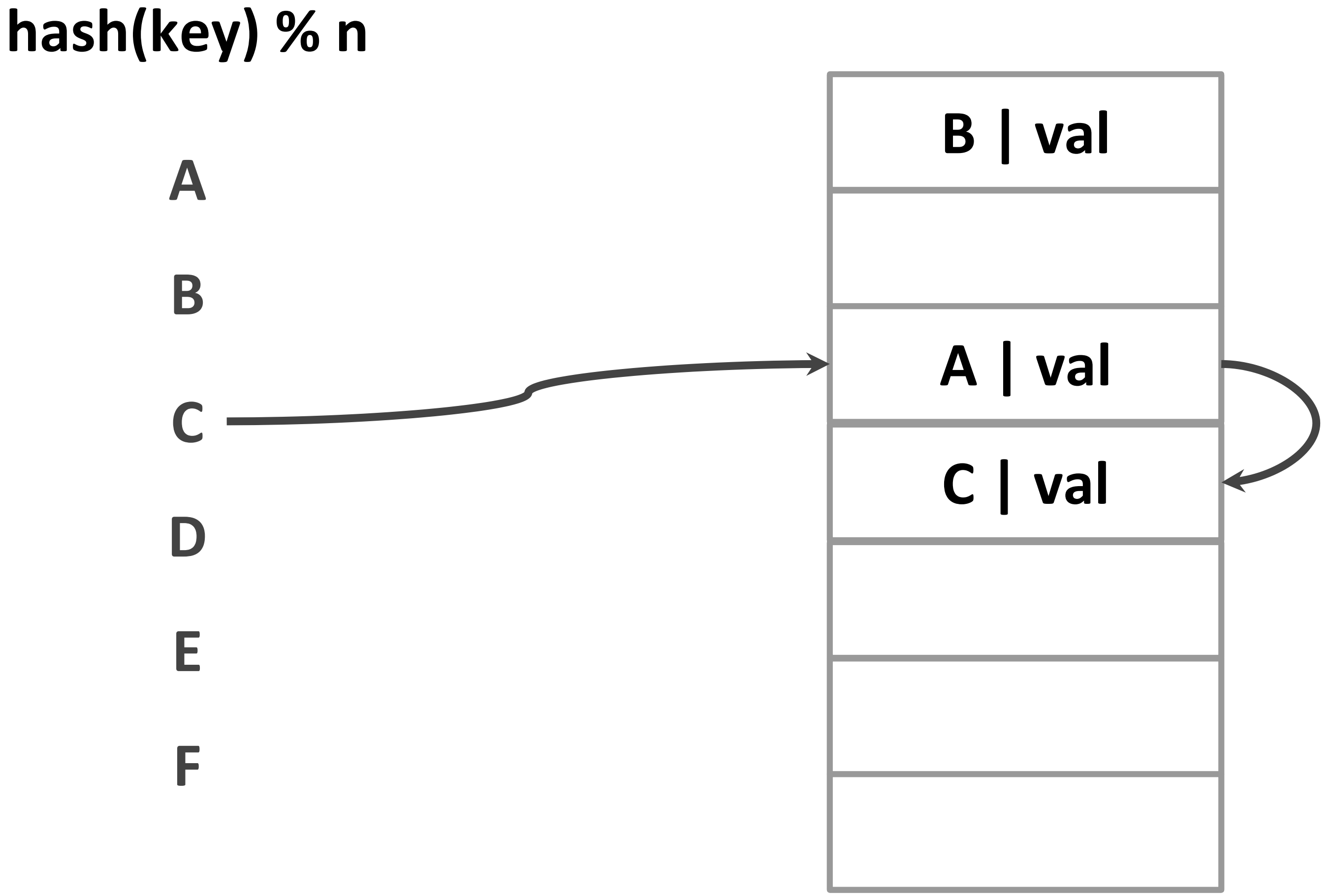


Linear probe hashing

hash(key) % n

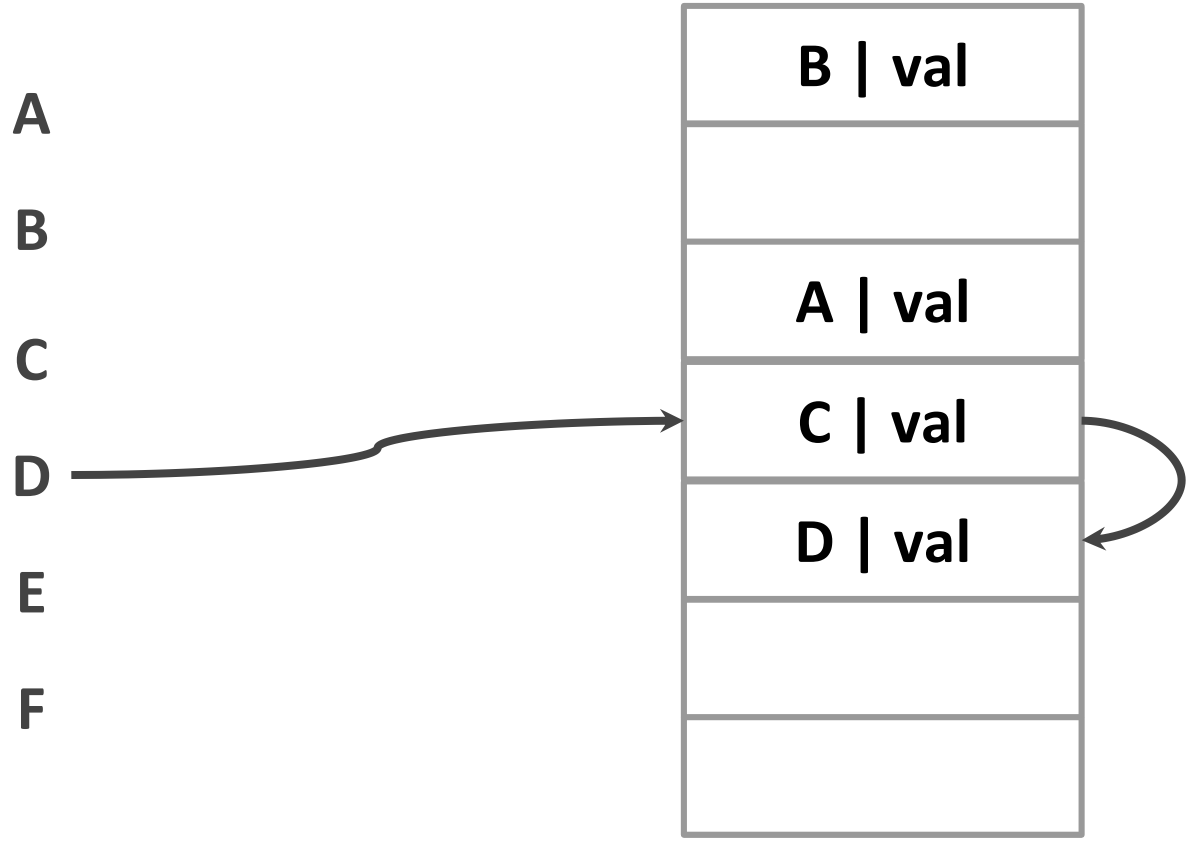


Linear probe hashing



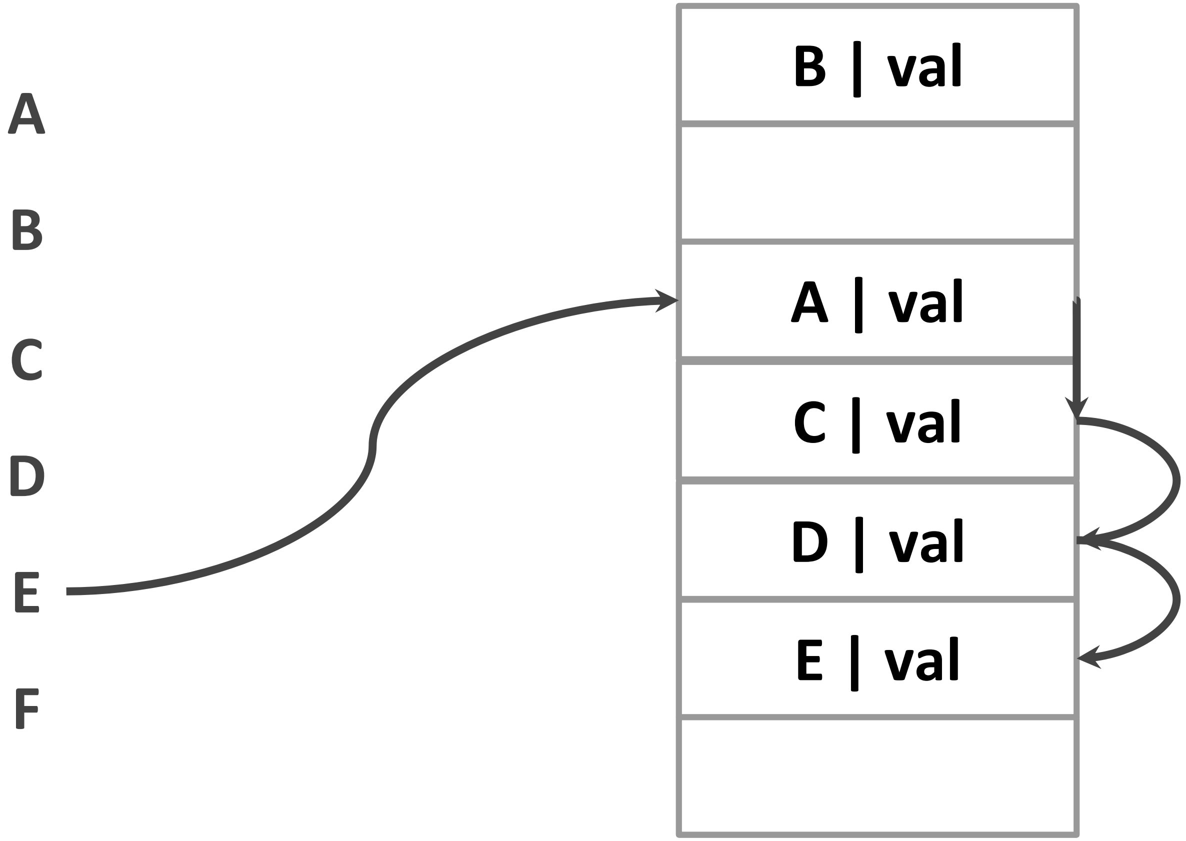
Linear probe hashing

hash(key) % n

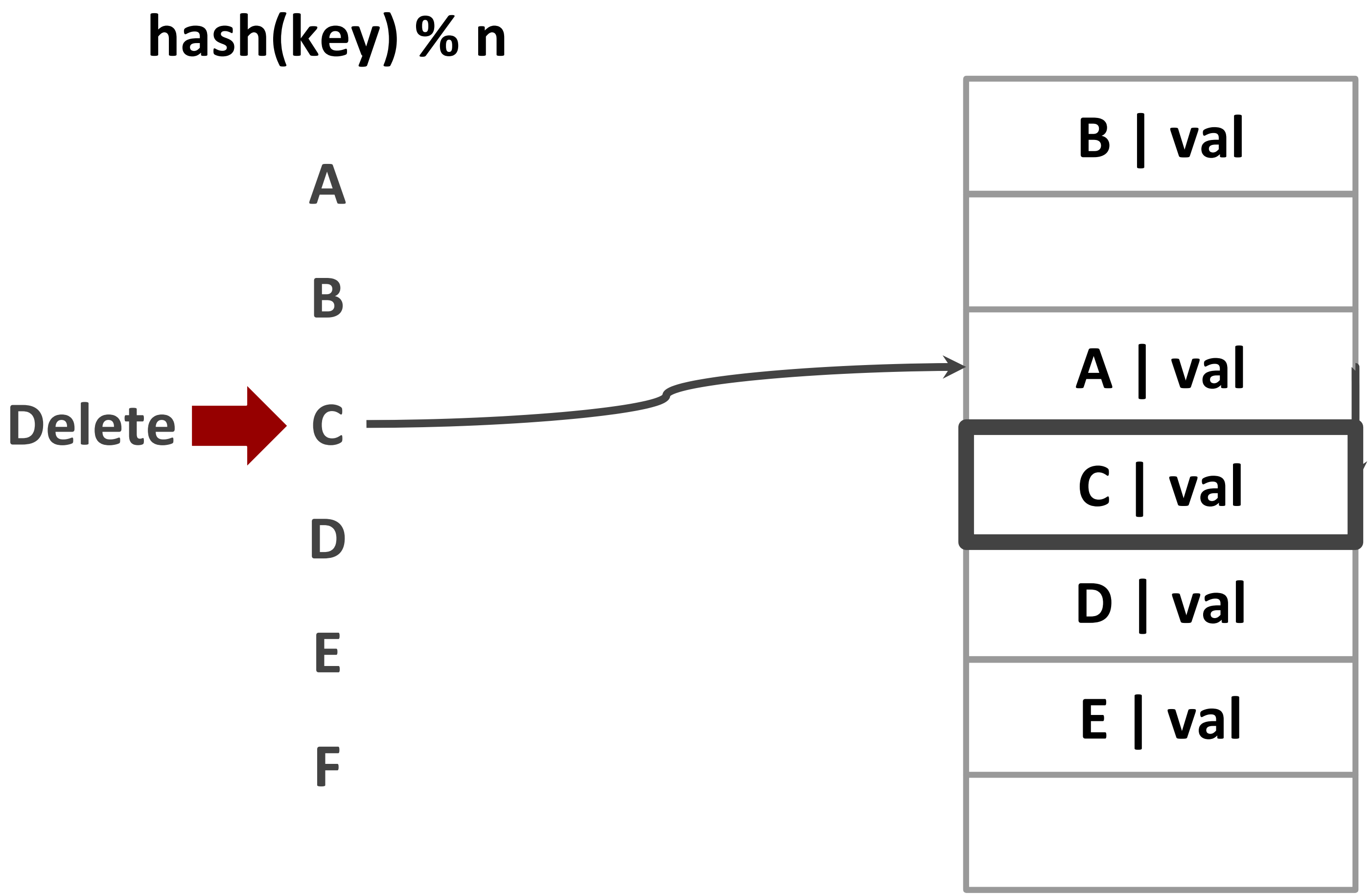


Linear probe hashing

hash(key) % n



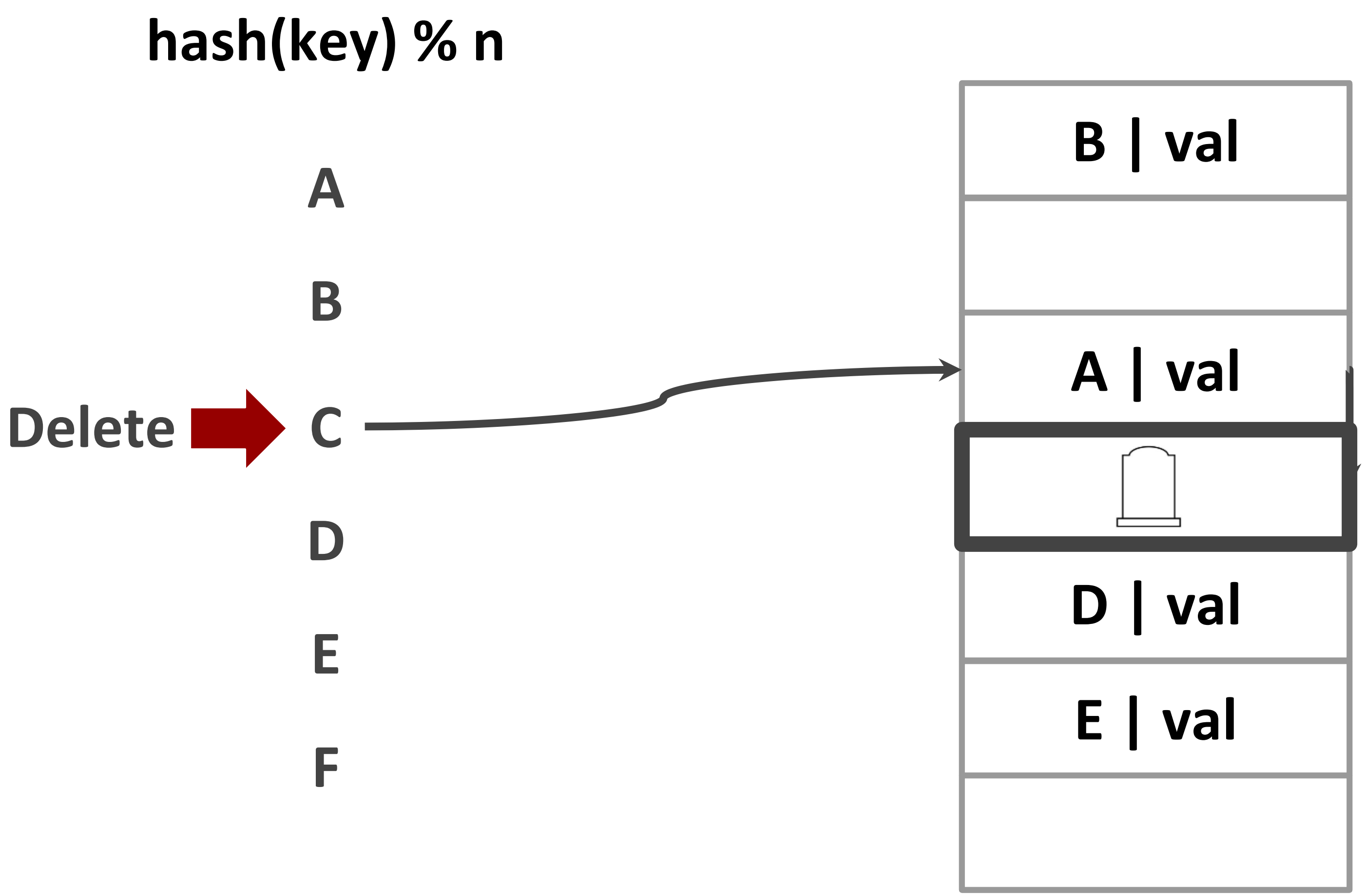
Linear probe hashing: DELETE



Approach: Tombstone

- Set a marker to indicate that the entry in the slot is logically deleted
- Reuse the slot for new keys
- May need periodic garbage collection

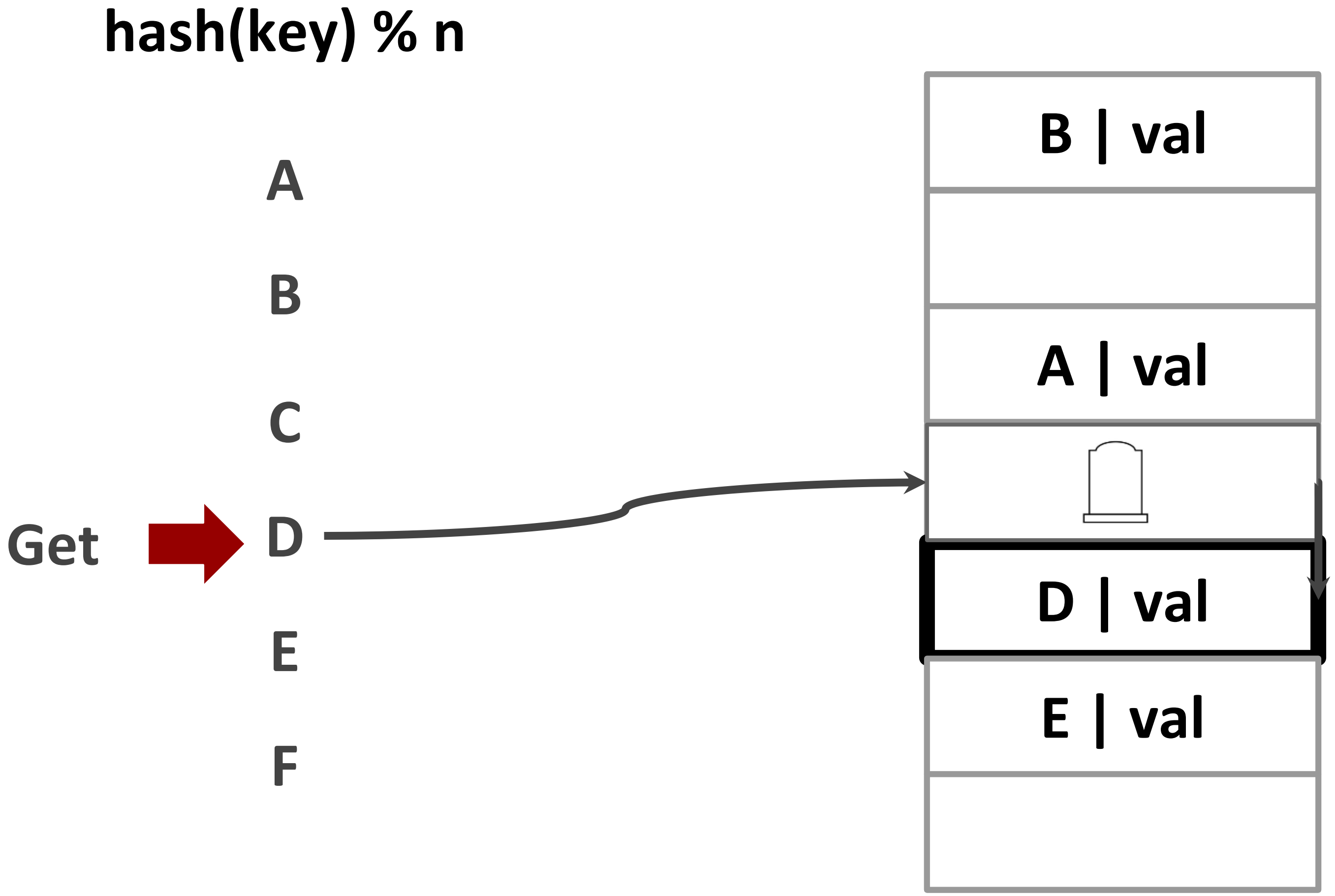
Linear probe hashing: DELETE



Approach: Tombstone

- Set a marker to indicate that the entry in the slot is logically deleted
- Reuse the slot for new keys
- May need periodic garbage collection

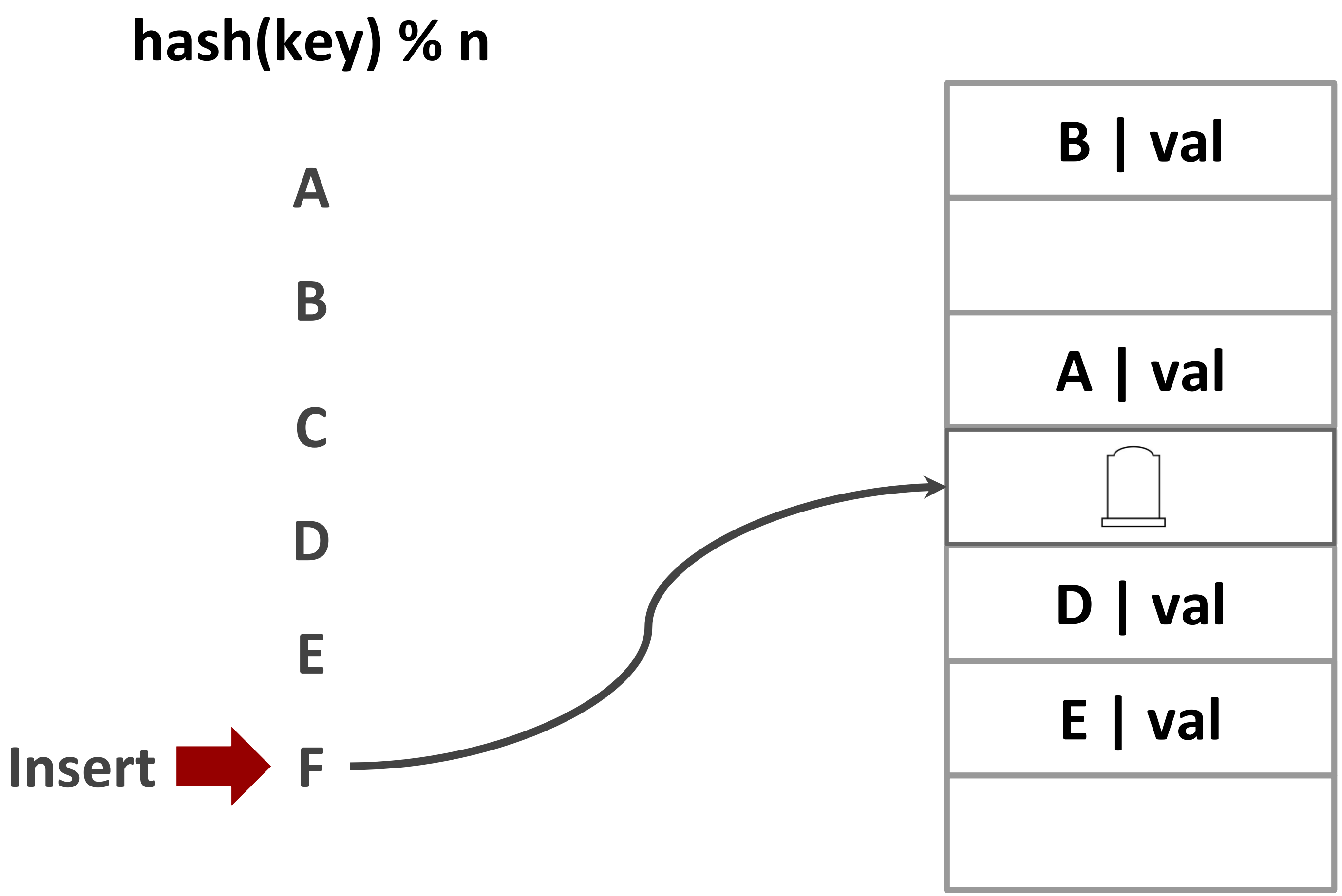
Linear probe hashing: DELETE



Approach: Tombstone

- Set a marker to indicate that the entry in the slot is logically deleted
- Reuse the slot for new keys
- May need periodic garbage collection

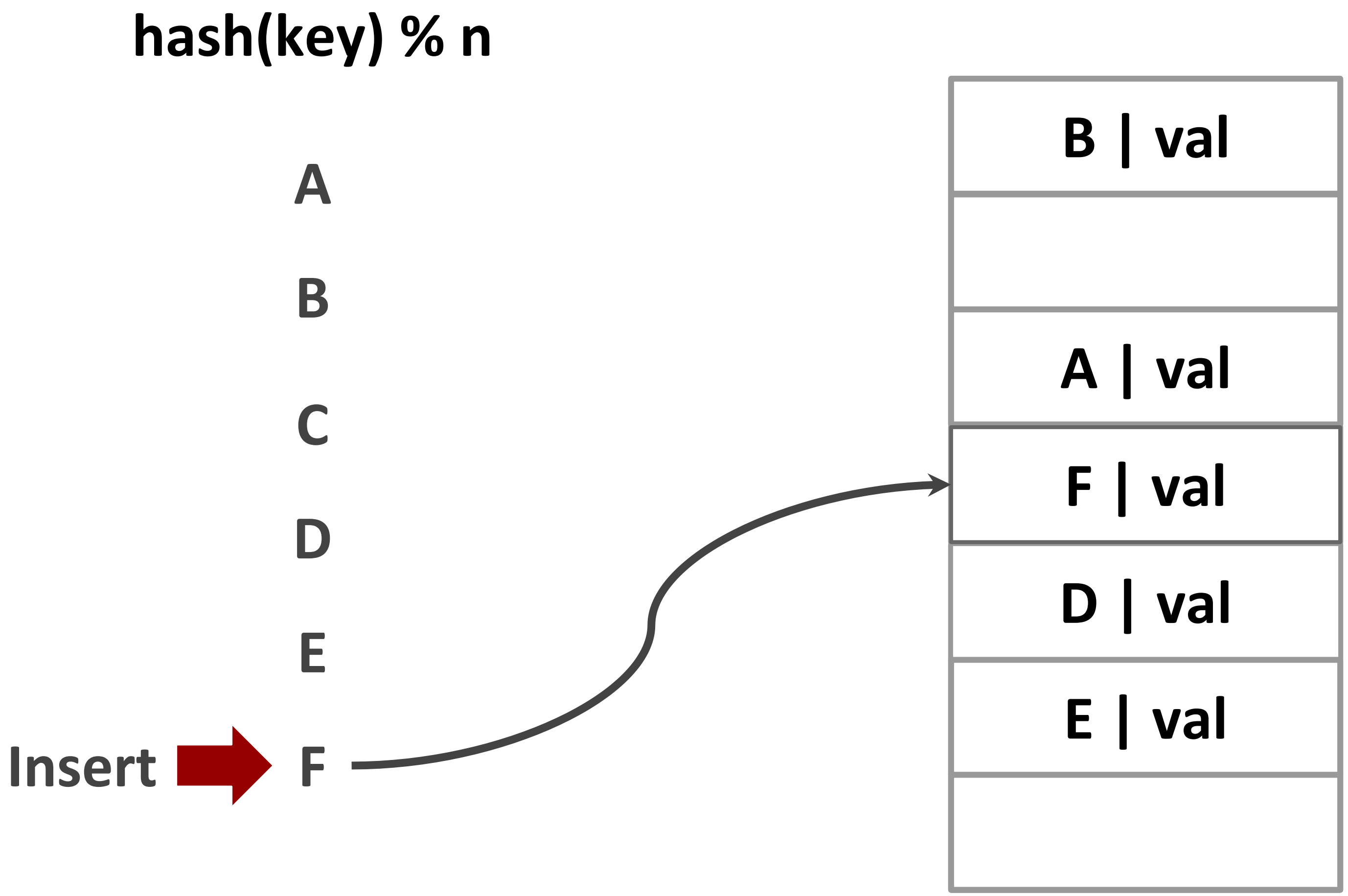
Linear probe hashing: DELETE



Approach: Tombstone

- Set a marker to indicate that the entry in the slot is logically deleted
- Reuse the slot for new keys
- May need periodic garbage collection

Linear probe hashing: INSERT

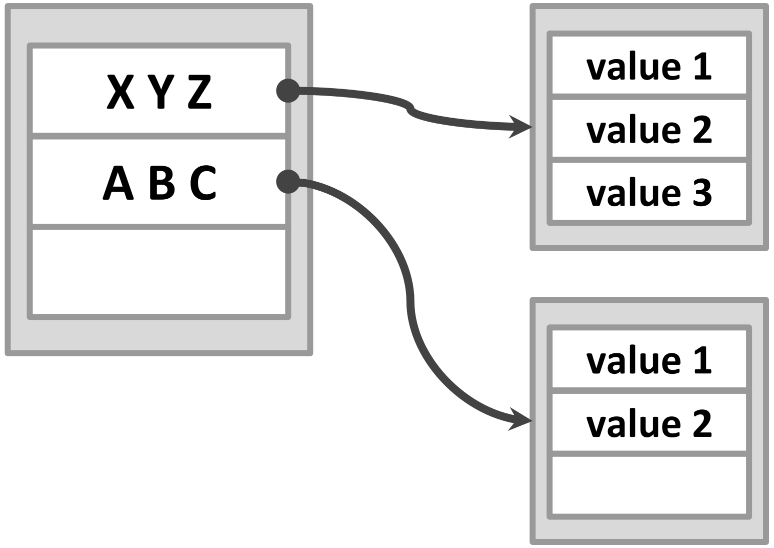


Approach: Tombstone

- Set a marker to indicate that the entry in the slot is logically deleted
- Reuse the slot for new keys
- May need periodic garbage collection

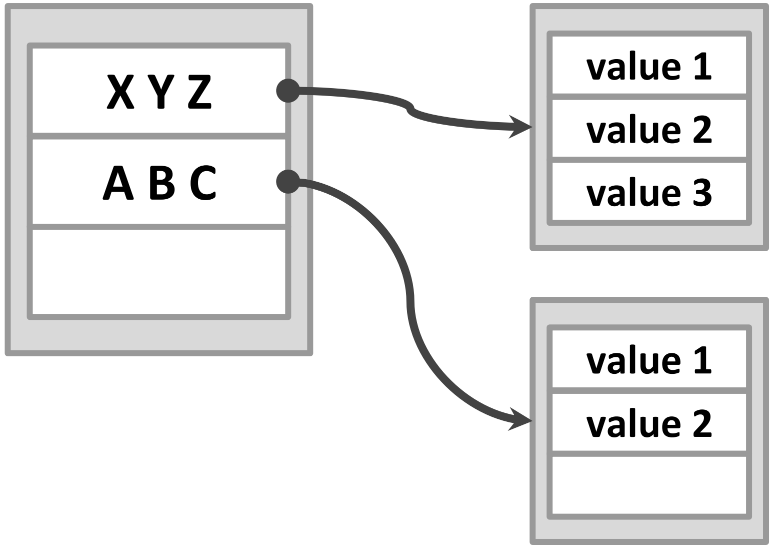
Non-unique keys

- Approach #1: Separate linked list
  - Store values in separate storage area for each key
  - Value lists can overflow to multiple pages if the number of duplicate pages is large

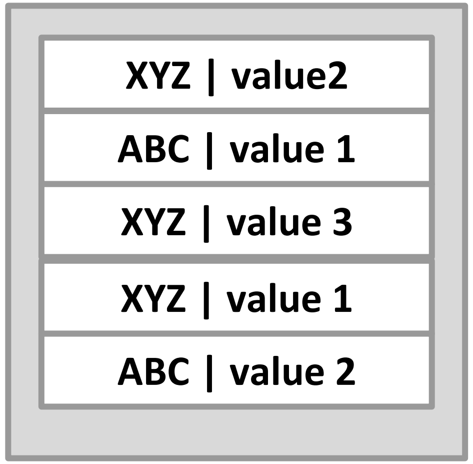


Non-unique keys

- Approach #1: Separate linked list
  - Store values in separate storage area for each key
  - Value lists can overflow to multiple pages if the number of duplicate pages is large



- Approach #2: Redundant keys
  - Store duplicate keys entries together in the hash table
  - Several systems use this approach



Issues with static hash table

- Requires the DBMS to know the number of elements it wants to store
  - Otherwise, it must rebuild the table to grow/shrink it in size
  - This process is costly: Index is blocked and reading/writing all pages is expensive

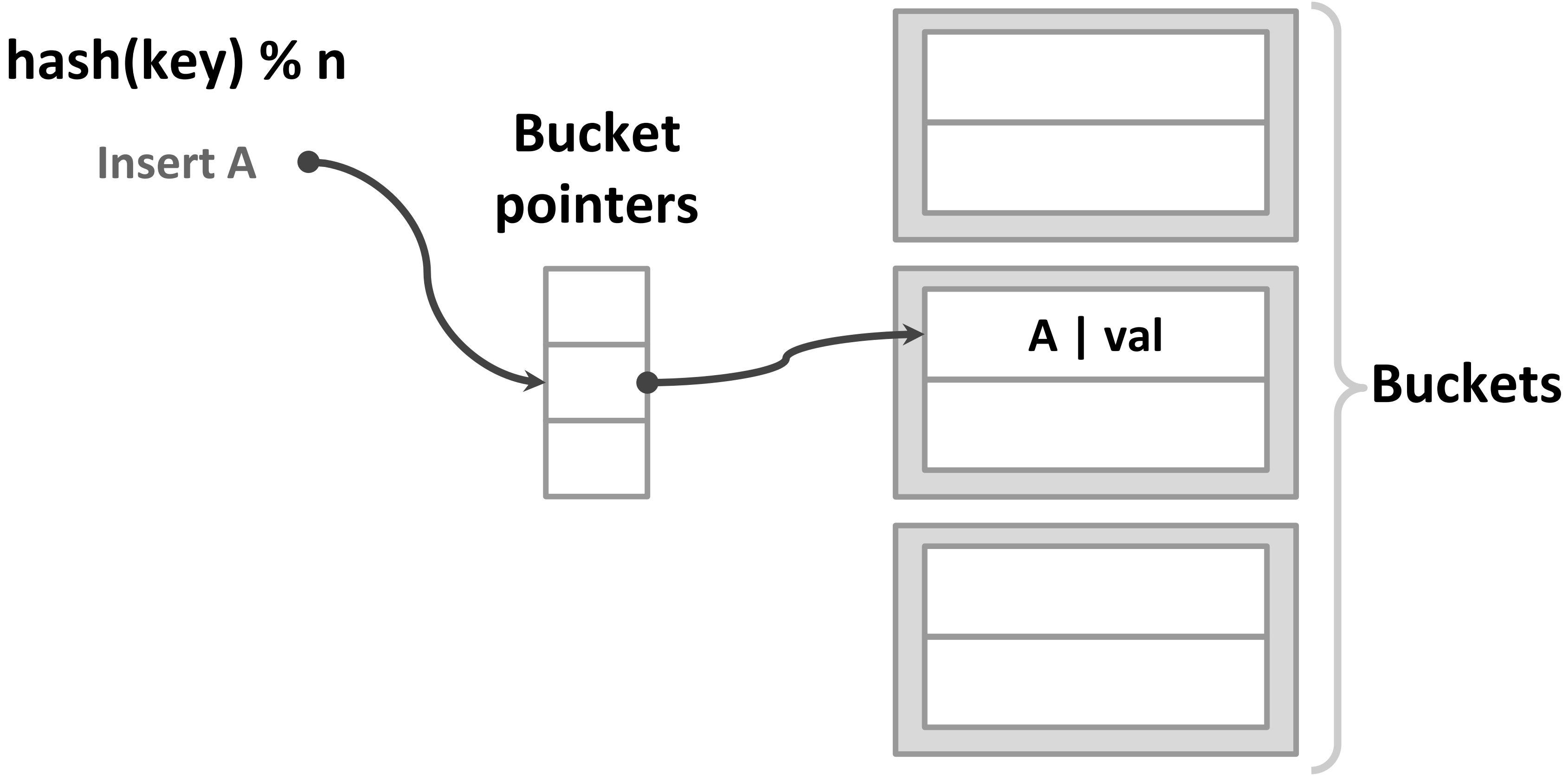
- Dynamic hash tables **incrementally resize** themselves when needed
  - Chained hashing
  - Extendible hashing
  - Linear hashing

Chained hashing

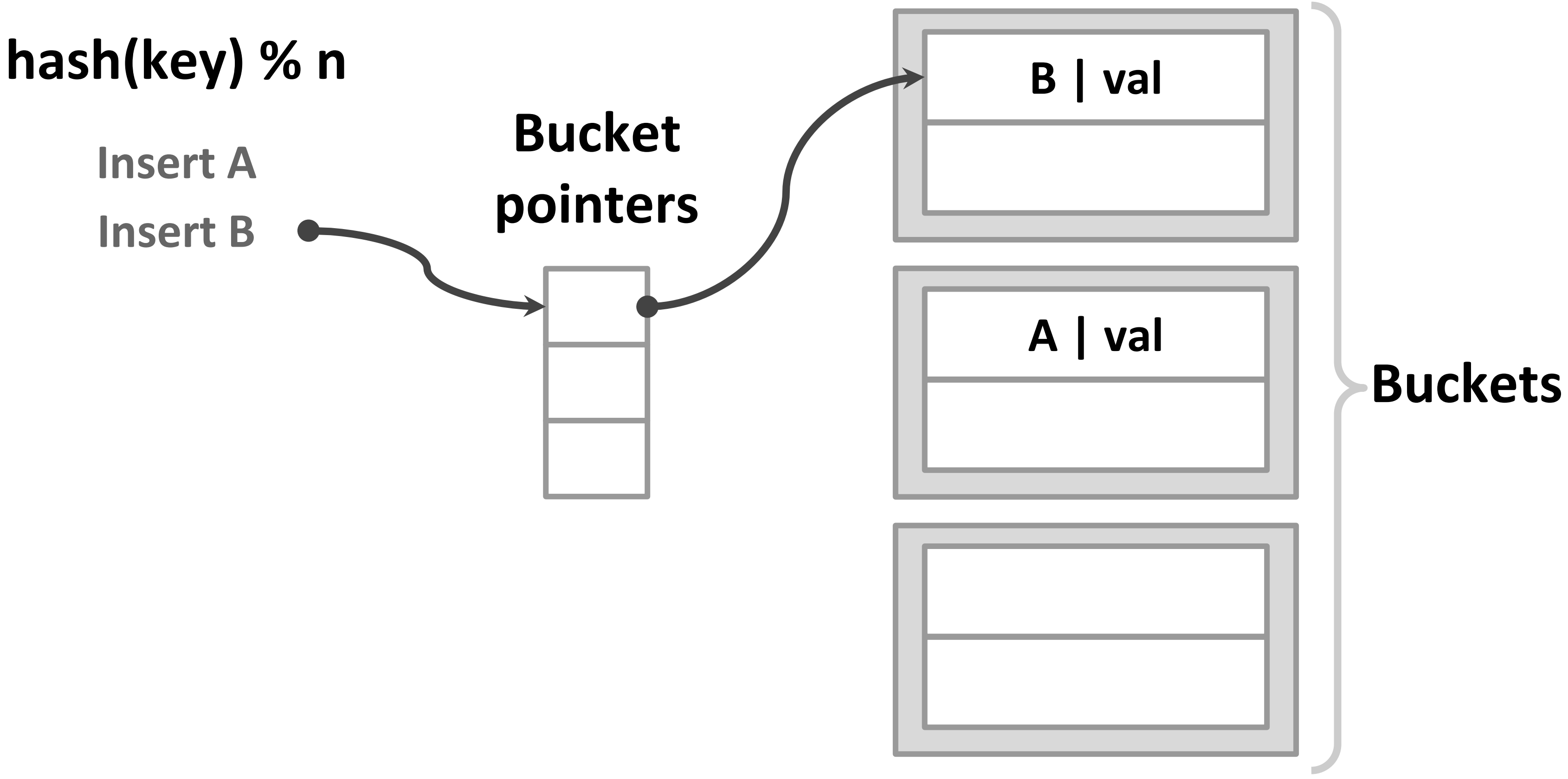
- Maintain a **linked list of buckets** for each slot in the hash table
- Maintain a **directory of pointers to buckets**

- Resolve collision by placing all elements with the same hash key into the same bucket
  - To determine whether an element is present, hash to its bucket and scan for it
  - Insertions and deletions are generalizations of lookups

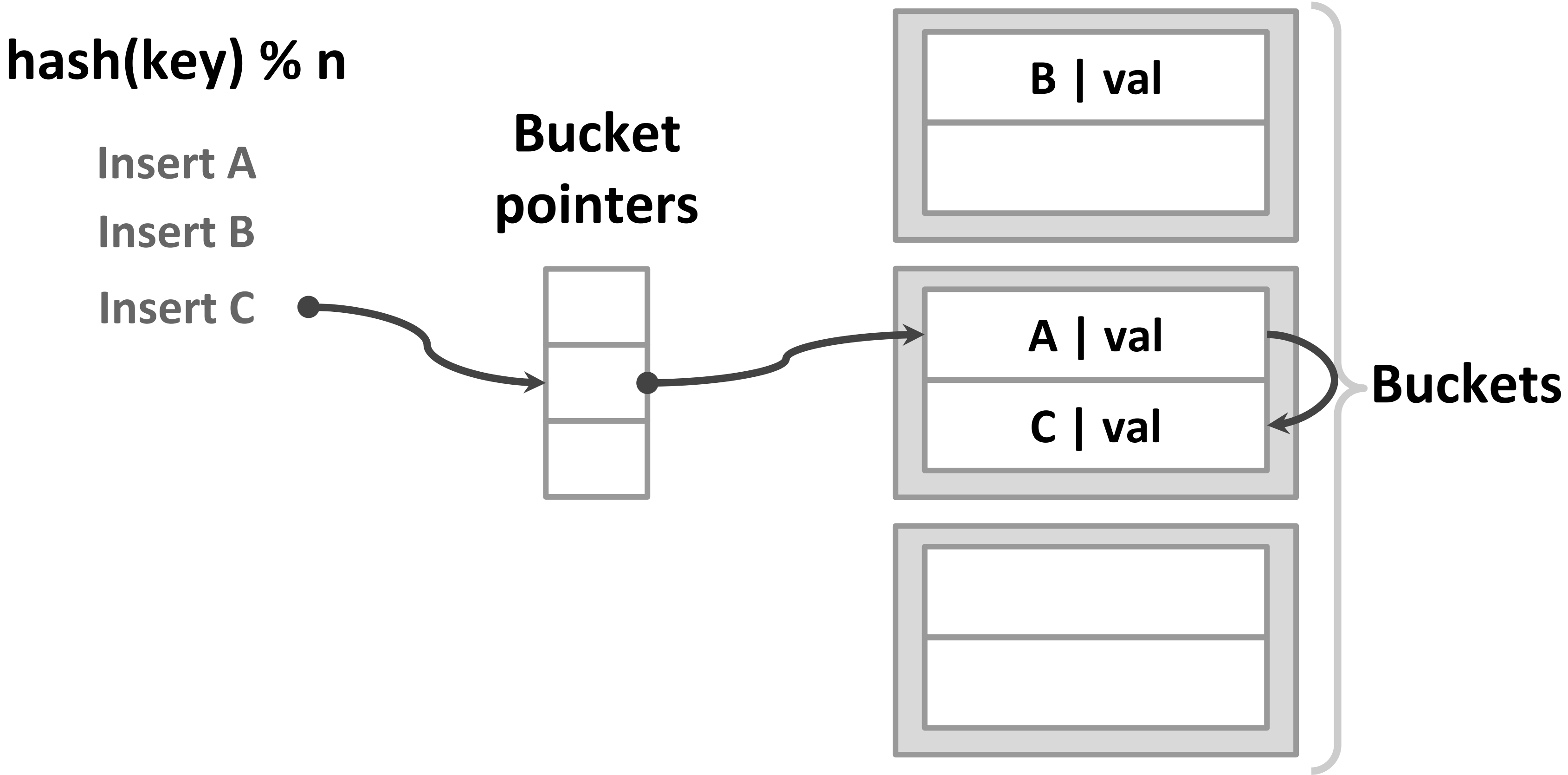
Chained hashing



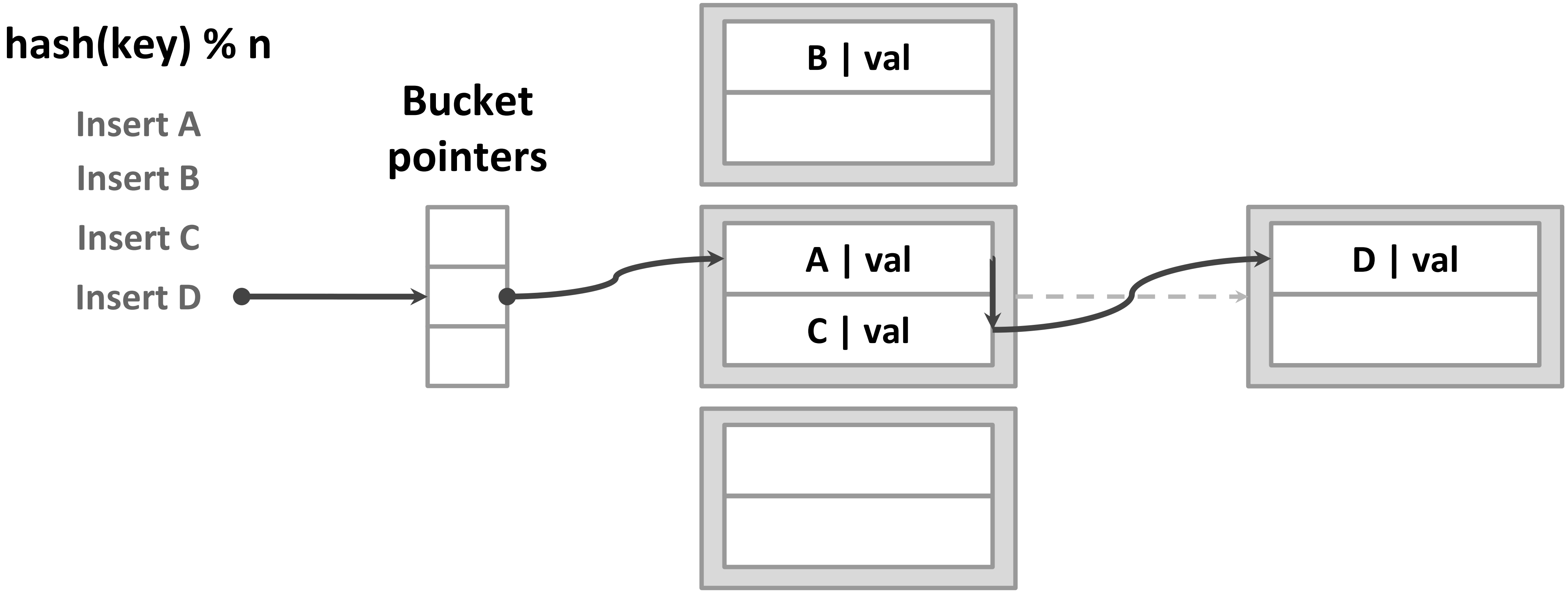
Chained hashing



Chained hashing



Chained hashing



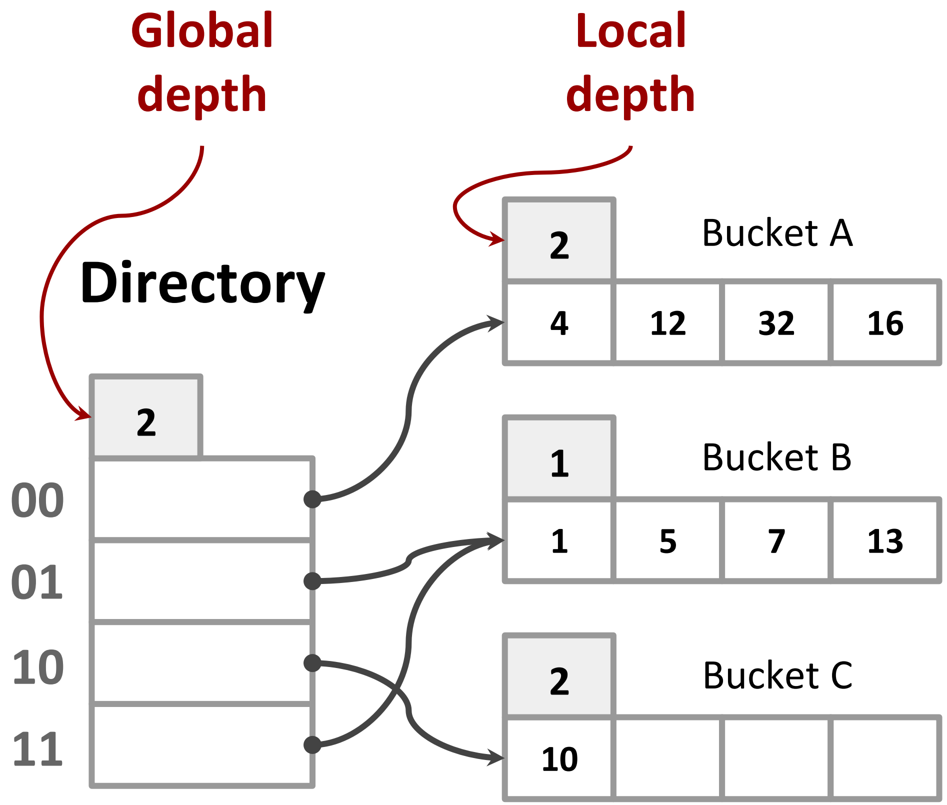
Extendible hashing

- Issues with chained-hashing:
  - Linked list can grow forever (not space efficient + pointer chasing)
  - Cannot have constant time lookups

- Extendible hashing is a variant of chained-hashing approach that **splits buckets incrementally** instead of letting the *linked list grow forever*
- Use **directory of pointers to buckets**
  - Double the number of buckets by doubling the directory, splitting only the bucket that overflowed
  - Data movement is localized to just the split chain

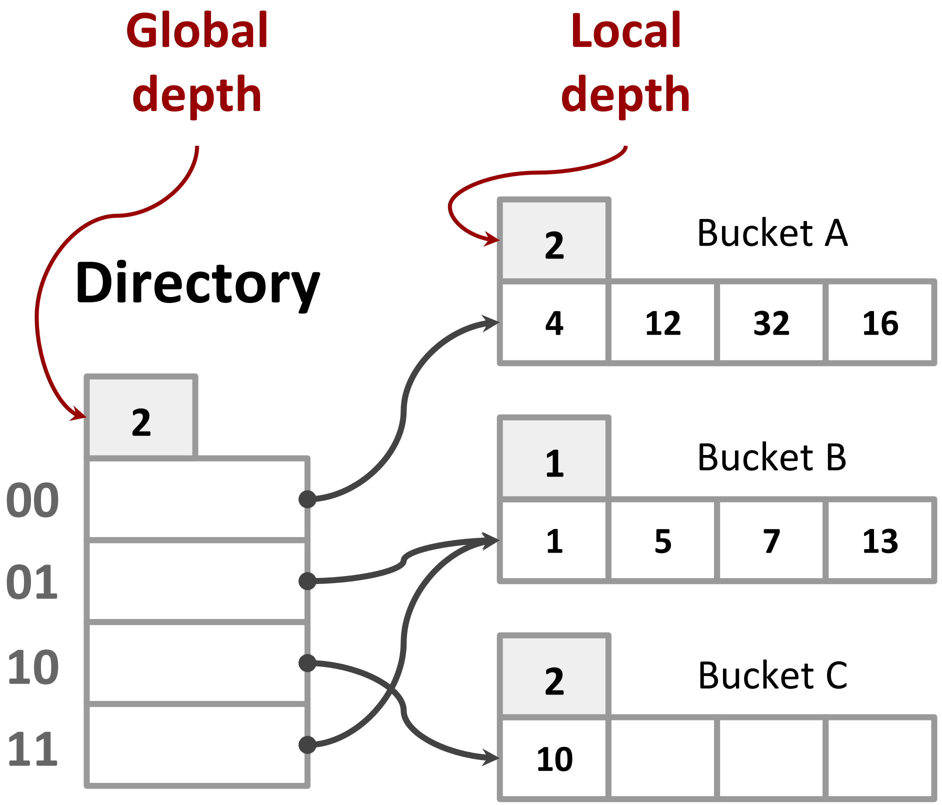
Extendible hashing: Example

- Bucket for record **r** has an entry with index = **`global depth`-least** significant bits of hash(r)
- E.g. directory is array of size 4 (global depth=2)
    - If hash(r) = 5 ⇒ 101 in binary
        - It is in bucket pointed to by 01
    - If hash(r) = 7 ⇒ 111 in binary
        - It is in bucket pointed by 11



Extendible hashing: Example (contd.)

- Assume Hash(x) = x for simplicity
- The location of the hash table corresponds to the least significant bits (LSB) to point to a bin in the directory table
  - Global depth of 2: use 2 LSB of the hash function
- Each bucket has a local depth: LSB shared by all bucket members, i.e., keys duplicate on at least *n bits*
  - Bucket A: All keys duplicate on the least significant 2 bits
  - Bucket B: All keys duplicate on the least significant 1 bit

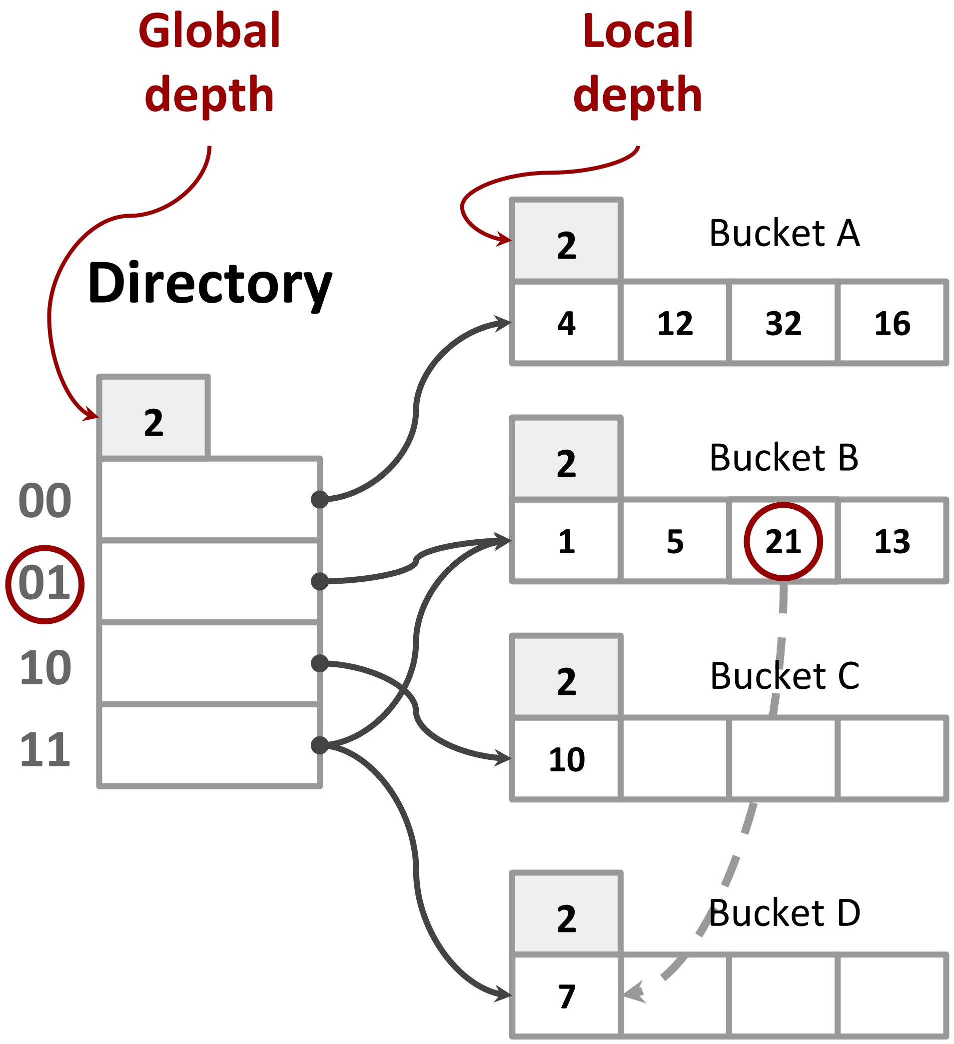


Extendible hashing: Inserts

- Find the bucket where record belongs
- If there is room, put the record there
- Else, if the bucket is full, split it:
    - Increment the local depth of the original page
    - Allocate a new page with new local depth
    - Add entry for the new page to the directory
    - Re-distribute records from the original page
- If the local depth > global depth:
    - double the hash table size
    - Remap pointers from the hash table to their respective bins based on the local depth value

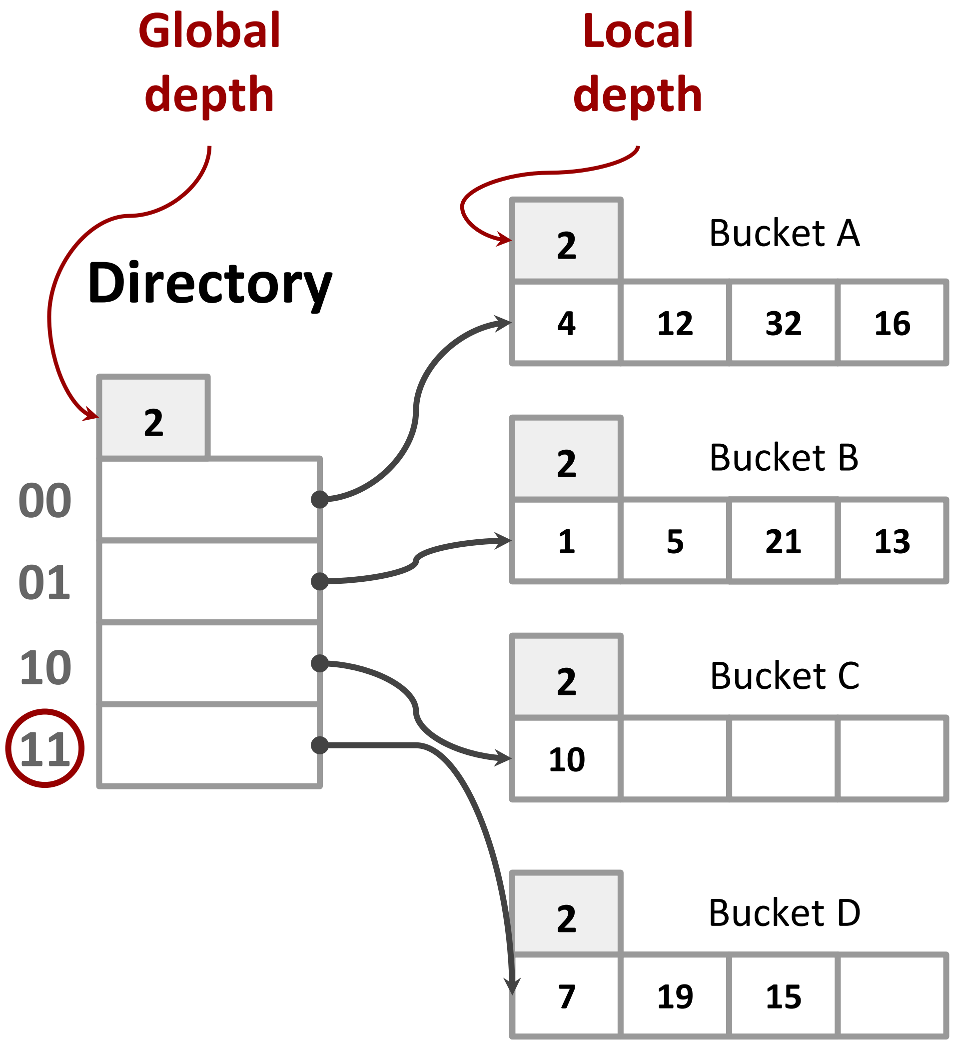
Extendible hashing: Insert 21 (10101)

- 21 (10101) goes to slot 01 pointing to bucket B
- Bucket B is full, increment the local depth by 1
- Allocate a new page (bucket D) with new local depth
- Both 01 and 11 point to bucket B, we can move key 7 (111) to Bucket D and update the hash table pointer for 11 to point to bucket D
- Add 21 to bucket B
- Nothing to balance as all elements are already distributed according to the global depth bits

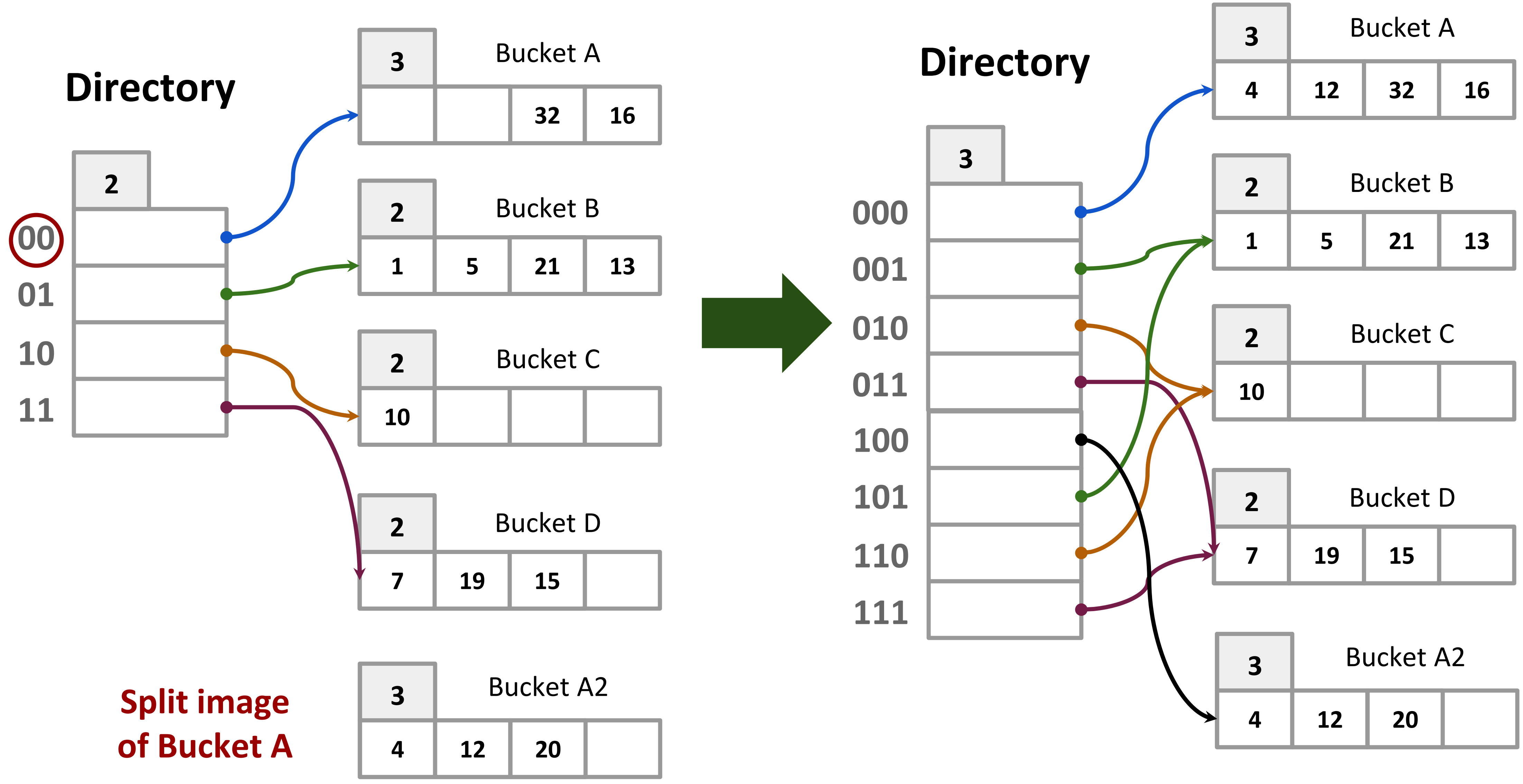


Extendible hashing: Insert 19 (10011), 15 (01111)

- Both 19 and 15 will go to bucket D which has enough space to accommodate



Extendible hashing: Insert 20 (10100)



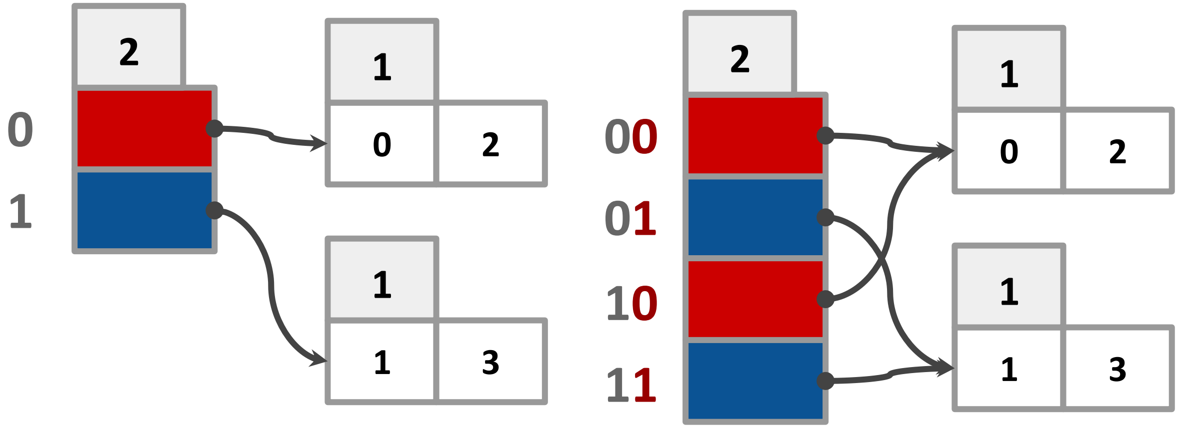
Directory doubling

Can double directory size based on least or most-significant bits (LSB or MSB)

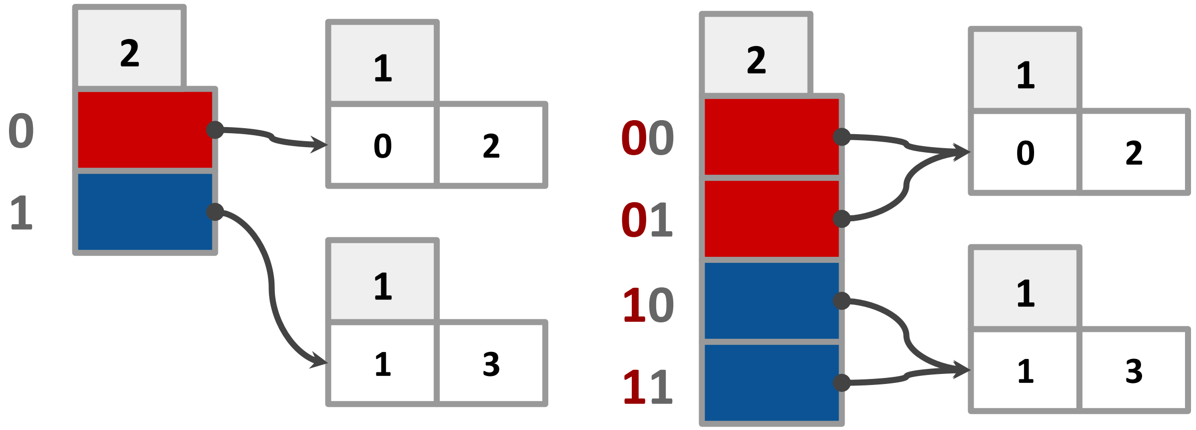
- LSB: directly append a new copy to the original page
- MSB: requires updating pointers for the earlier bins

→ Look at the colors of the bins

Least significant bits



Most significant bits



Linear hashing: Overflow chains without directory

- The hash table maintains a **pointer** that tracks the next bucket to split
  - When any bucket overflows, split the bucket the pointer points to!

- Avoids directory by using **temporary** overflow pages
- Avoids long overflow chains by choosing the bucket to split in a **round-robin** fashion

- Seamlessly handles duplicates and collision
- Flexible in trading off performance for space usage

Linear hashing: Main idea

- Uses a family of hash functions h_0, h_1, h_2, h_3 … to find the right bucket for a given key
  - h_{i+1} doubles the range of h_i

- $h_i(key) = h(key) mod(2^i N)$
  - N → Initial # buckets, **h** is a hash function
  - Apply hash function **h** and look at the last d_i bits
- Example: N = 4
  - $h_0(key) = h(key) mod(4)$
  - $h_1(key) = h(key) mod(8)$
  - $h_2(key) = h(key) mod(16)$

Linear hashing: algorithm

The algorithm proceeds in **rounds.**

Current round number is the hashing **level** ("i" in previous slide)

- There are N_{level} (= $N * 2^{level}$) buckets at the beginning of the round
- **next** is the bucket that will be split
  - When any bucket overflows, split the **next** bucket and then increment **next**
- Buckets **0** to **next-1** have been split; Buckets **next** to $\mathbf{N_{level}}$ have not been split yet in this round
- Rounds end when all **initial** buckets have been split, i.e., **next =** $\mathbf{N_{level}}$
- To start the next round: increment **level** by 1 and reset the **next** to 0

Linear hashing: search algorithm

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

  - If $h_{level}(r)$ >= next (i.e., $h_{level}(r)$ is a bucket that has not been involved in a split (in this round) then **r** belongs in that bucket for sure

  - Else, r could belong to bucket $h_{level}(r)$ OR bucket $h_{level}(r) + N_{level}$
    - Must also apply $h_{level+1}(r)$ to find out

Linear hashing: insert algorithm

- First find the appropriate bucket

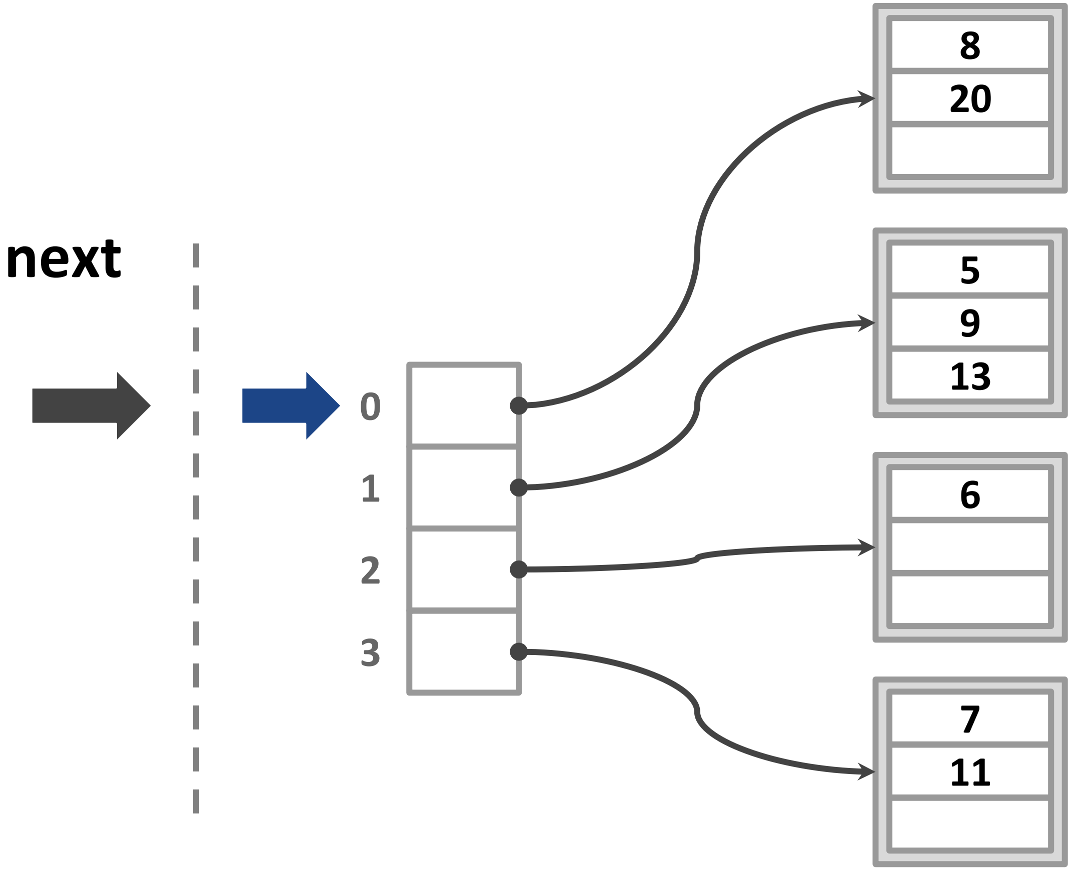
- If that bucket is full:
  - Add overflow page and insert data entry
  - Split **next** bucket and increment **next**
    - **Note:** This is likely NOT the bucket where the insertion happens
  - To split a bucket, create a new bucket and use $\mathbf{h_{level+1}}$ to re-distribute entries

- Since buckets are split in a round-robin fashion, long overflow chains do not occur

Linear hashing illustration

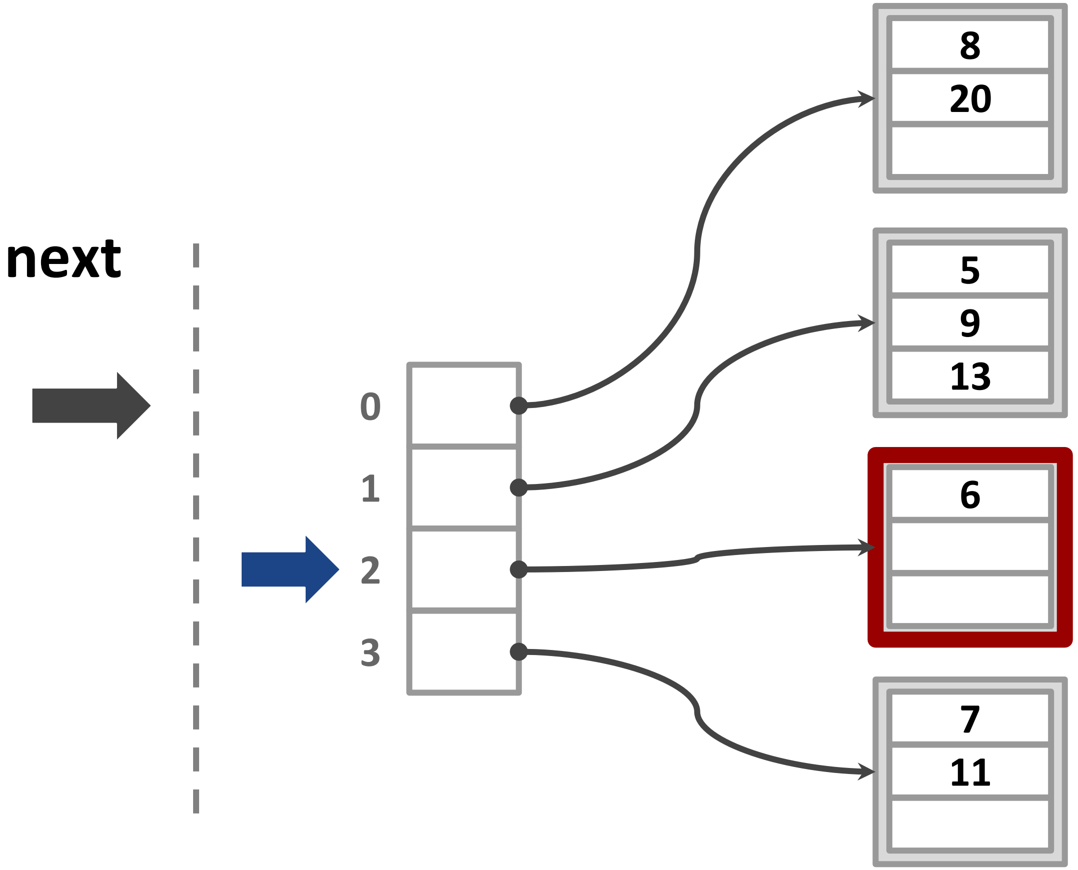
Search for 6

`hash(6) = 6 % 4 = 2`



`hash(key) = key % n`

Linear hashing illustration

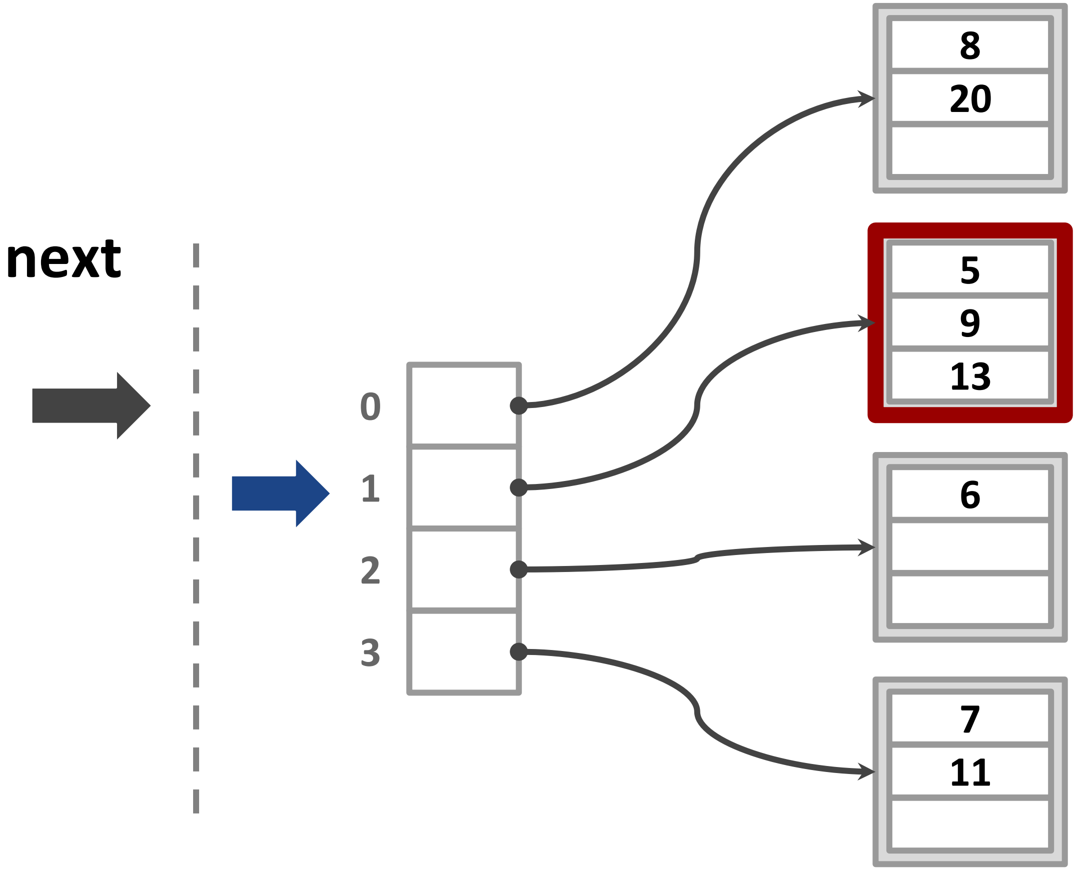


`hash(key) = key % n`

Search for 6

`hash(6) = 6 % 4 = 2`

Linear hashing illustration



`hash(key) = key % n`

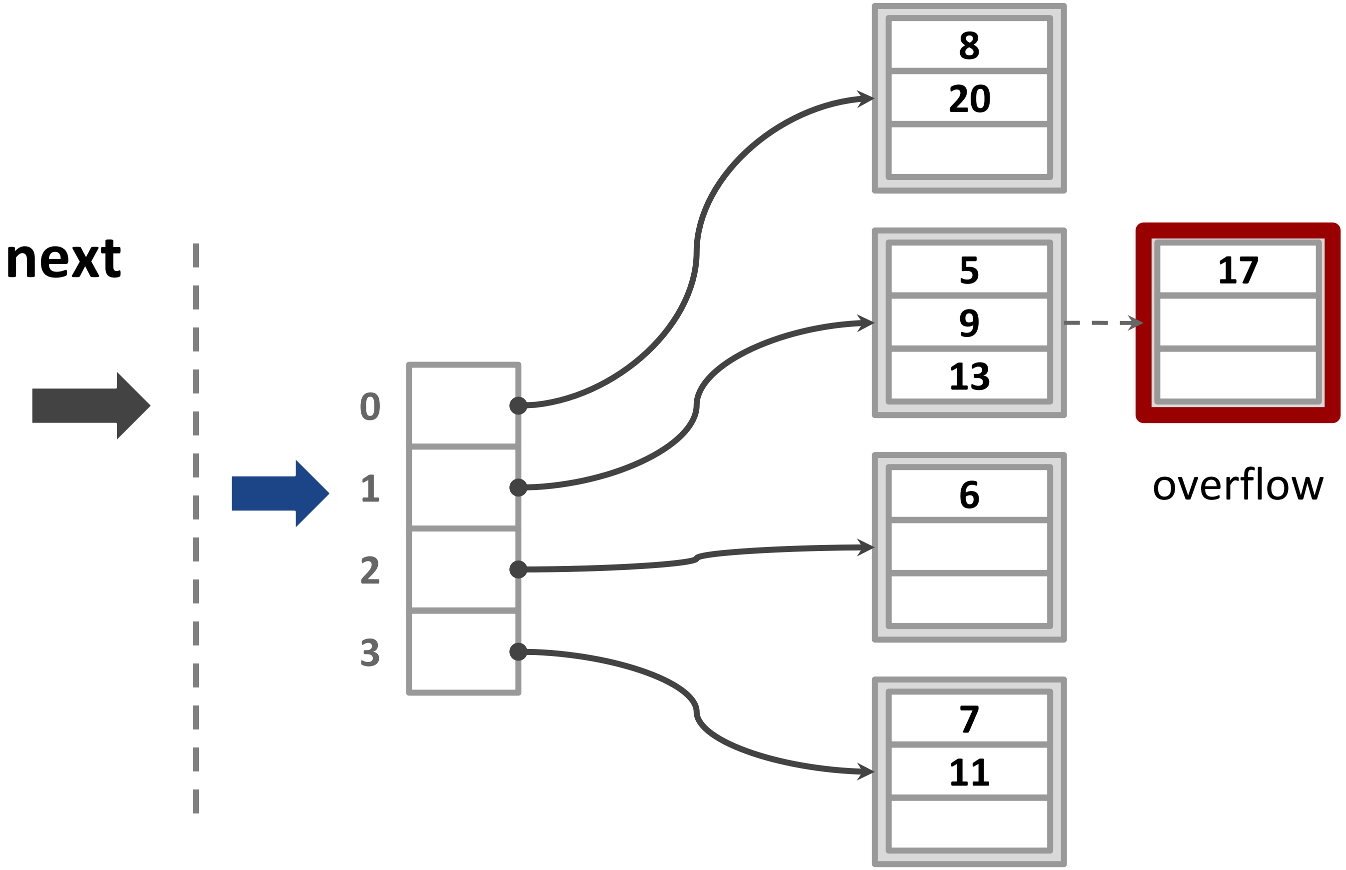
Search for 6

`hash(6) = 6 % 4 = 2`

Insert 17

`hash(17) = 17 % 4 = 1`

Linear hashing illustration



`hash(key) = key % n`

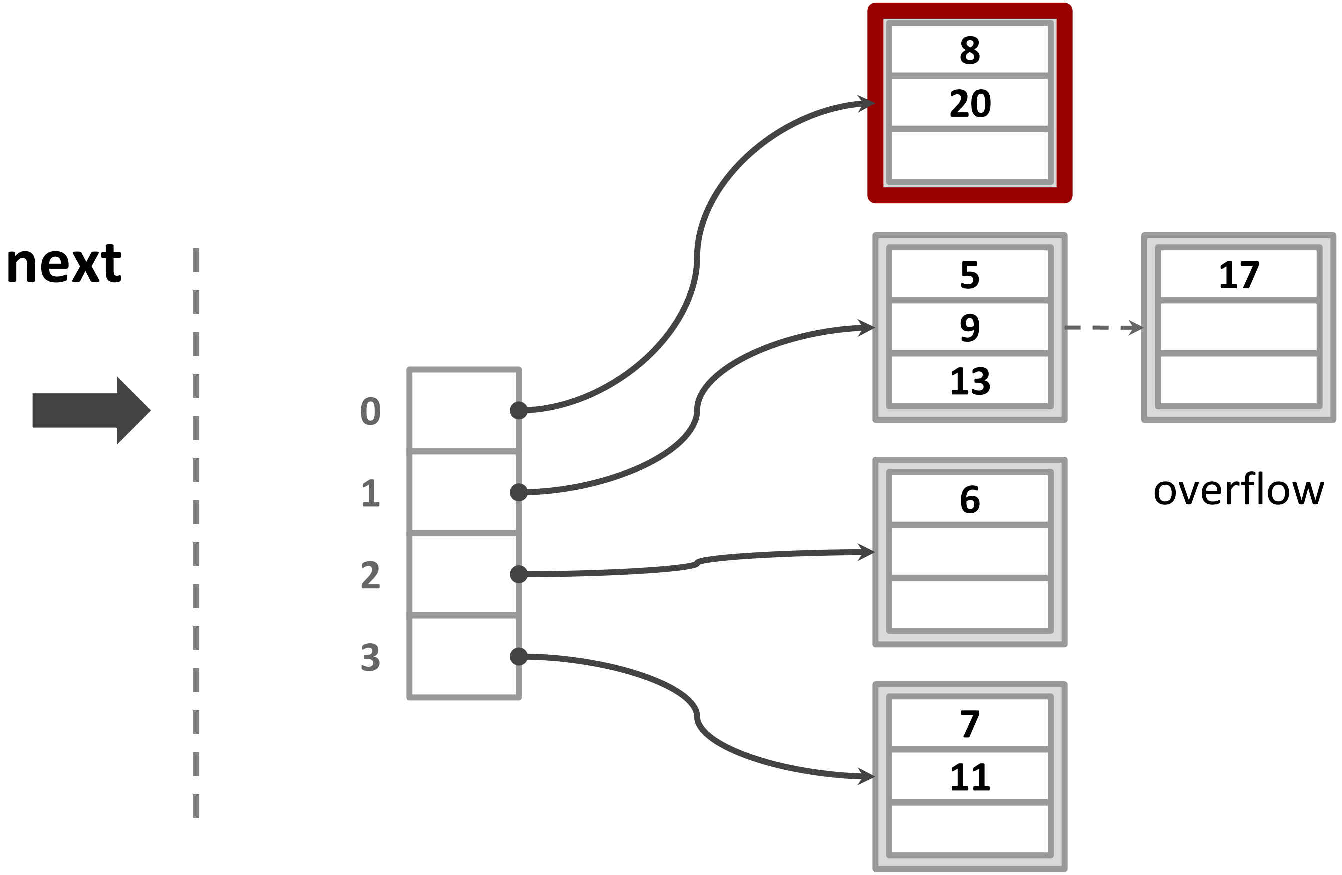
Search for 6

`hash(6) = 6 % 4 = 2`

Insert 17

`hash(17) = 17 % 4 = 1`

Linear hashing illustration



`hash(key) = key % n`

`hash(key) = key % 2n`

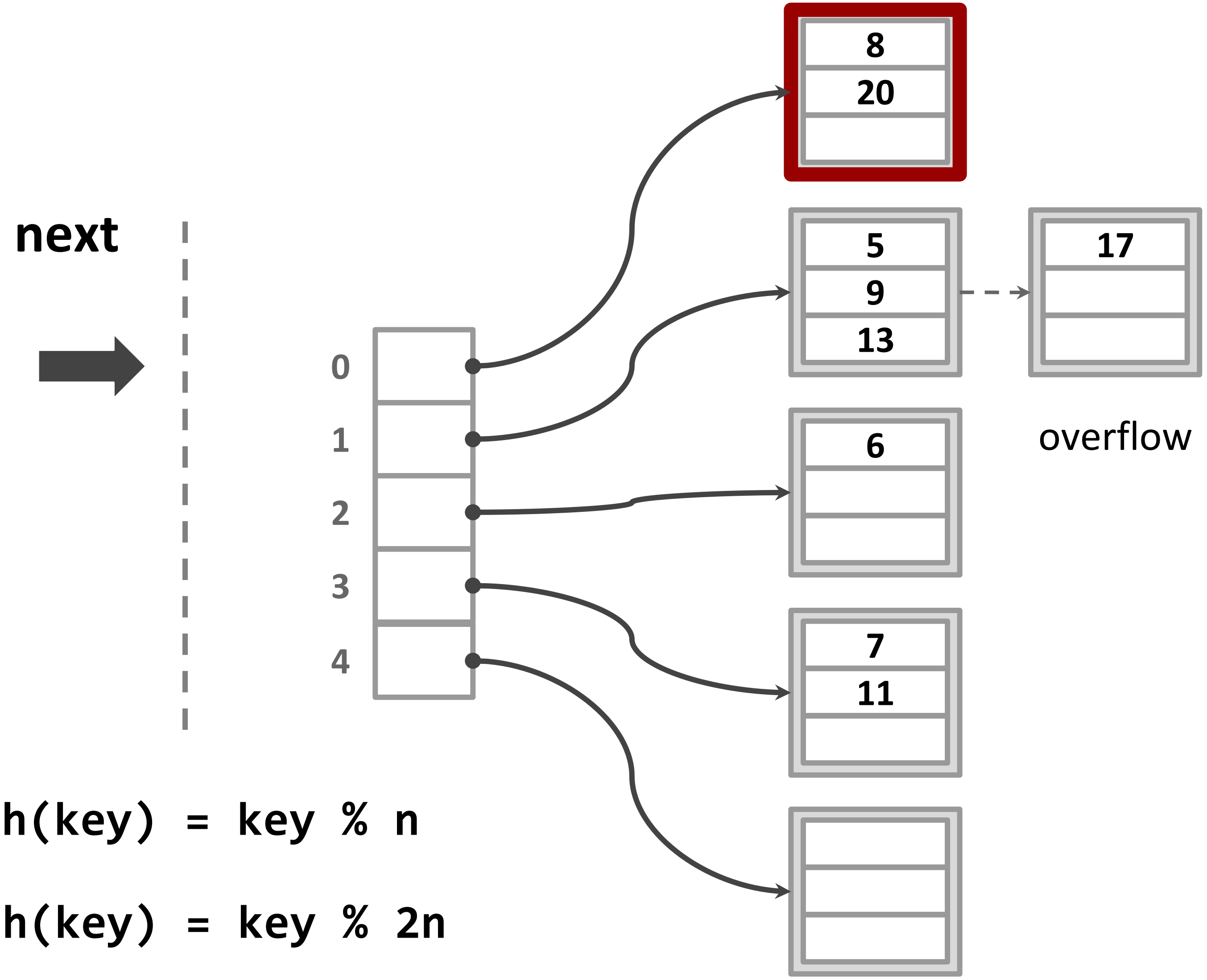
Search for 6

`hash(6) = 6 % 4 = 2`

Insert 17

`hash(17) = 17 % 4 = 1`

Linear hashing illustration



hash(key) = key % n

hash(key) = key % 2n

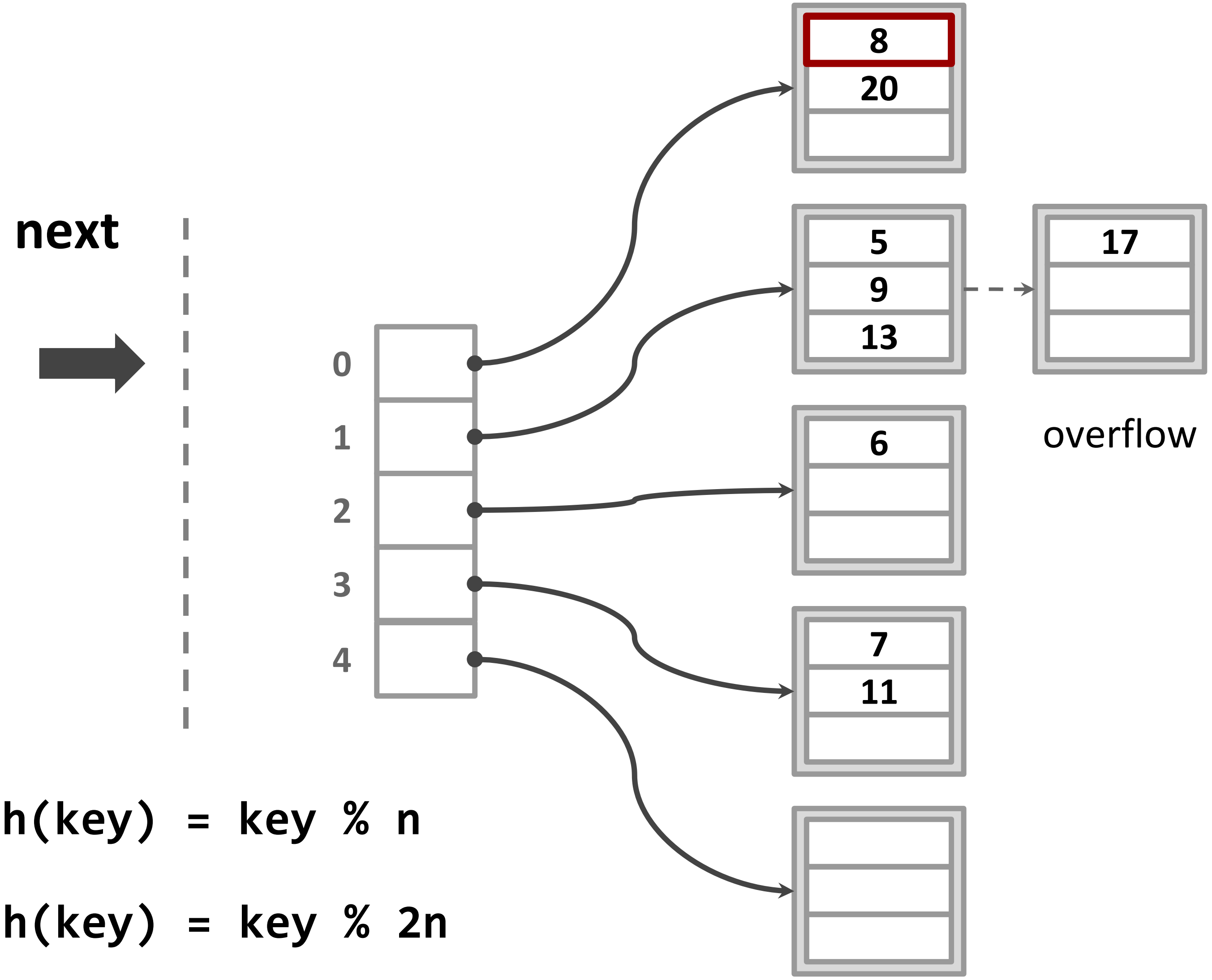
Search for 6

hash(6) = 6 % 4 = 2

Insert 17

hash(17) = 17 % 4 = 1

Linear hashing illustration



`hash(key) = key % n`

`hash(key) = key % 2n`

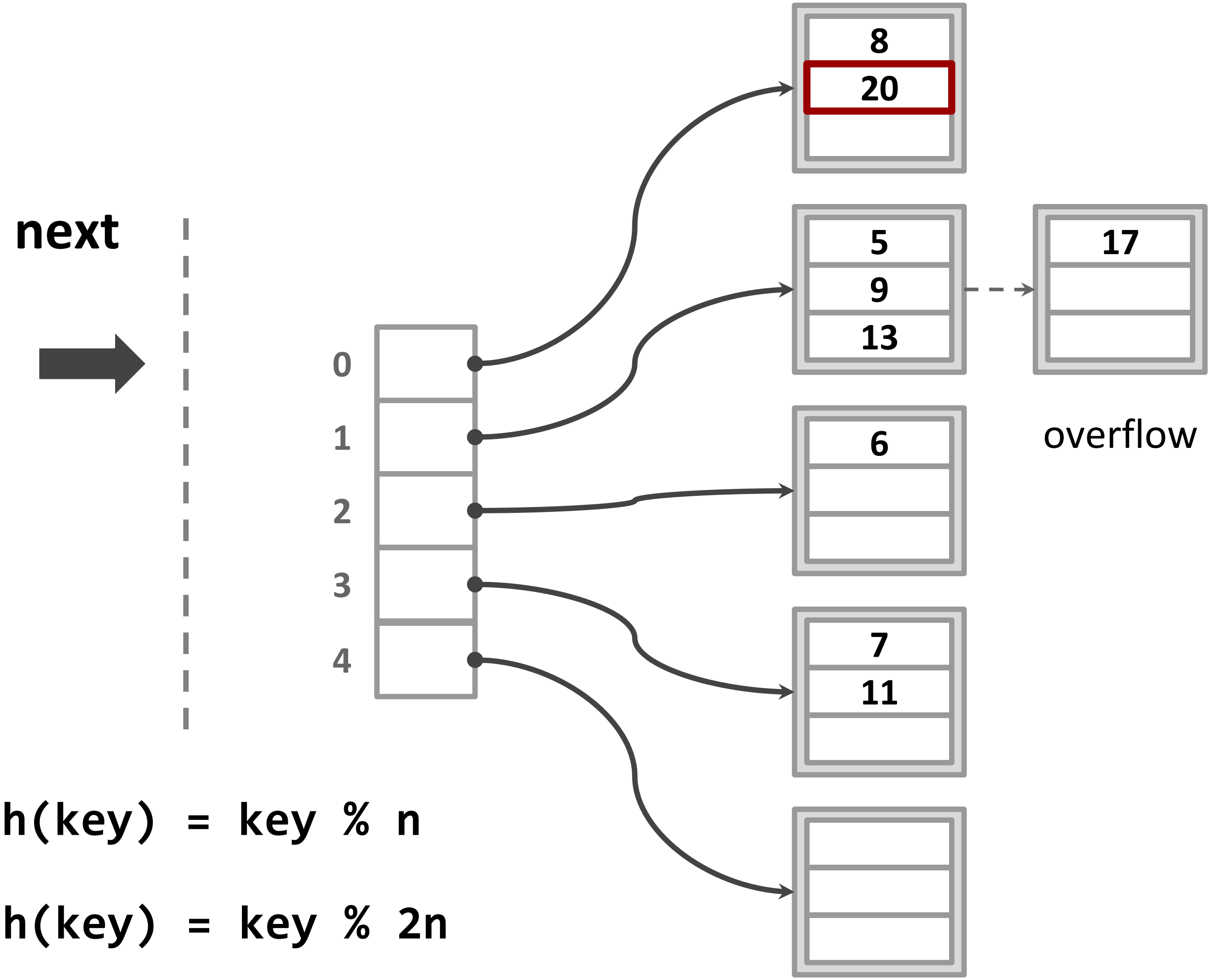
Search for 6

`hash(6) = 6 % 4 = 2`

Insert 17

`hash(17) = 17 % 4 = 1`
`hash(8)  =  8 % 8 = 0`

Linear hashing illustration




```
hash(key) = key % n

hash(key) = key % 2n
```

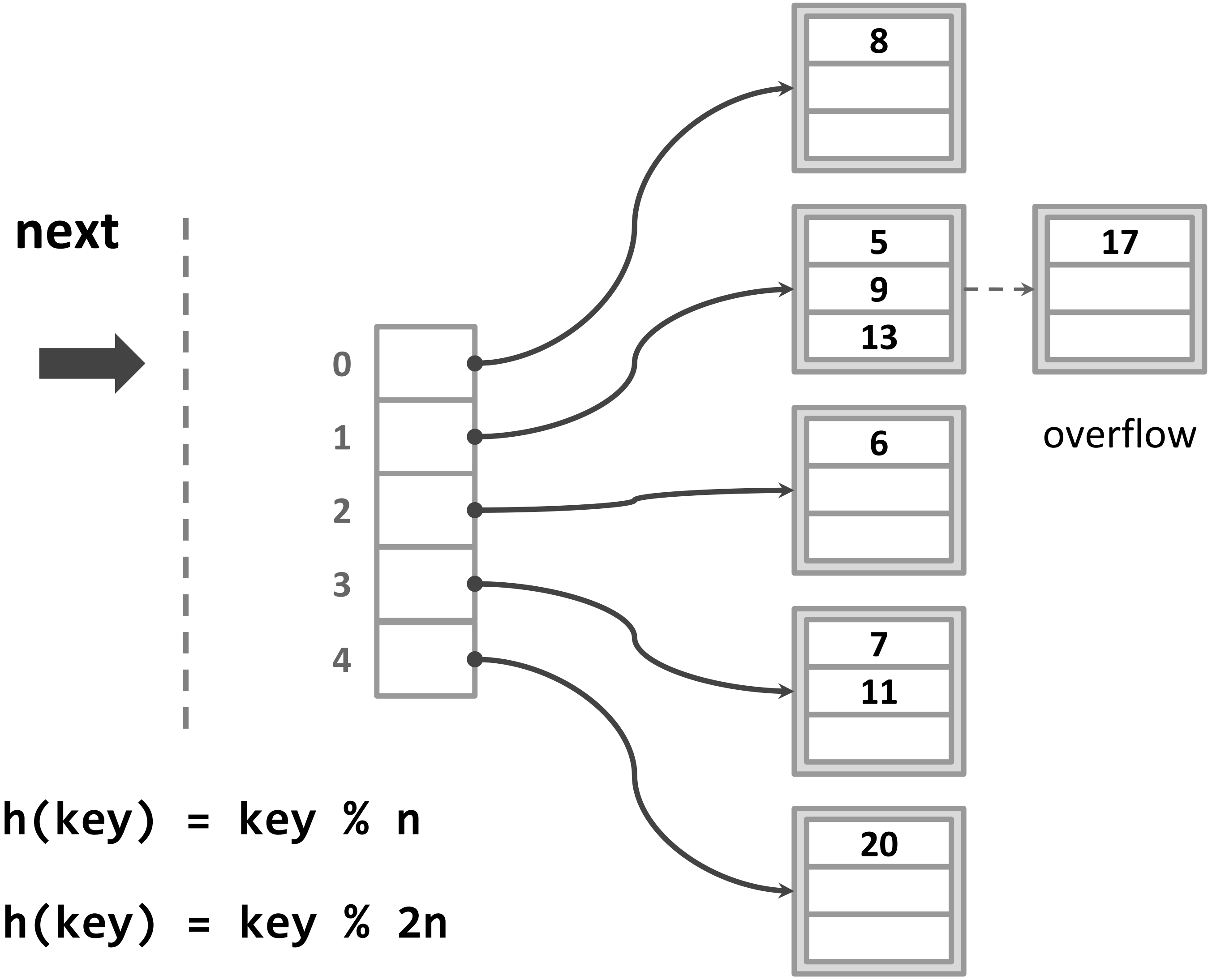
Search for 6

```
hash(6) = 6 % 4 = 2
```

Insert 17

```
hash(17) = 17 % 4 = 1
hash(8)  =  8 % 8 = 0
hash(20) = 20 % 8 = 4
```

Linear hashing illustration



`hash(key) = key % n`

`hash(key) = key % 2n`

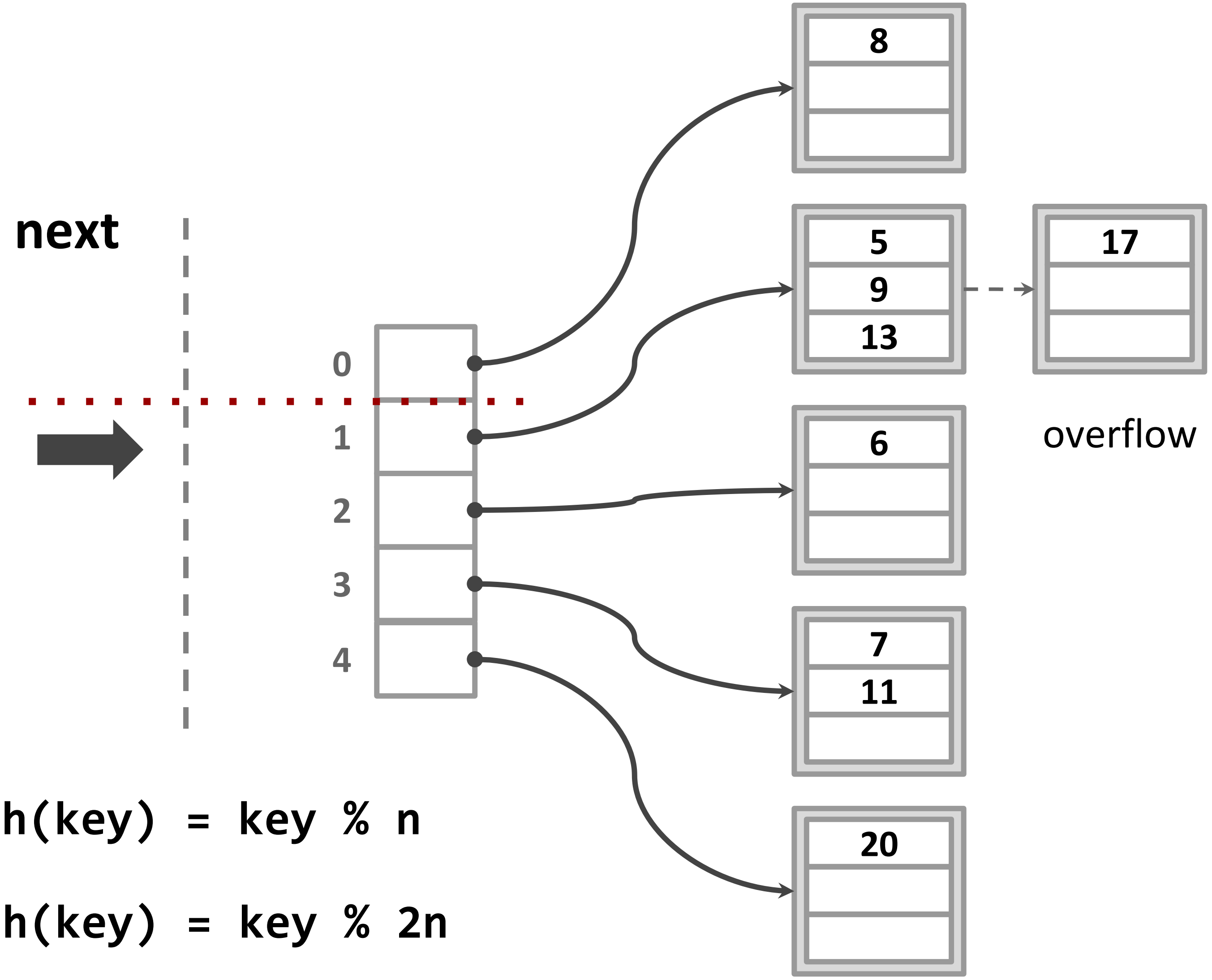
Search for 6

`hash(6) = 6 % 4 = 2`

Insert 17

```
hash(17) = 17 % 4 = 1
hash(8)  =  8 % 8 = 0
hash(20) = 20 % 8 = 4
```

Linear hashing illustration




```
hash(key) = key % n

hash(key) = key % 2n
```

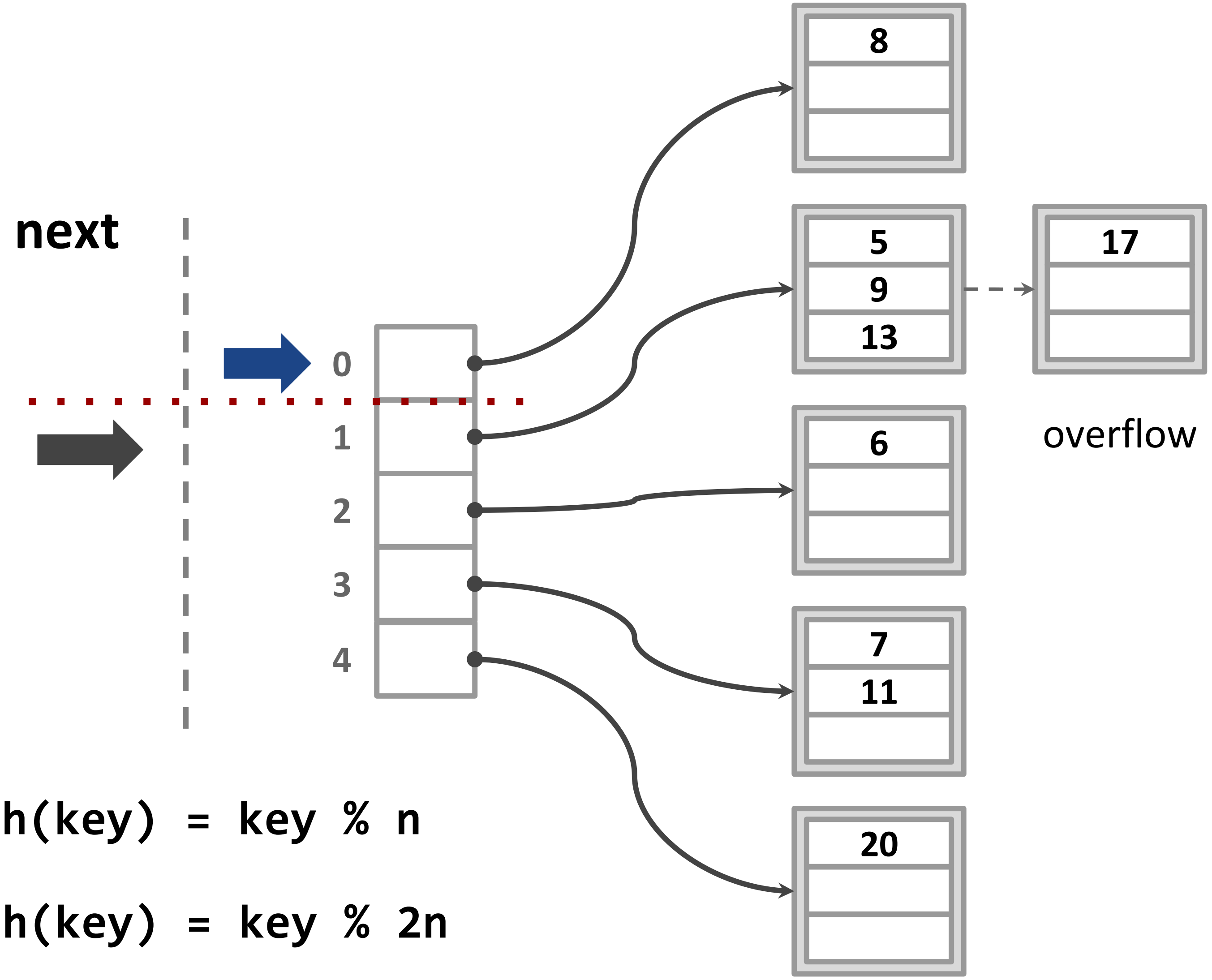
Search for 6

```
hash(6) = 6 % 4 = 2
```

Insert 17

```
hash(17) = 17 % 4 = 1
hash(8)  =  8 % 8 = 0
hash(20) = 20 % 8 = 4
```

Linear hashing illustration



`hash(key) = key % n`

`hash(key) = key % 2n`

Search for 6

```
hash(6) = 6 % 4 = 2
```

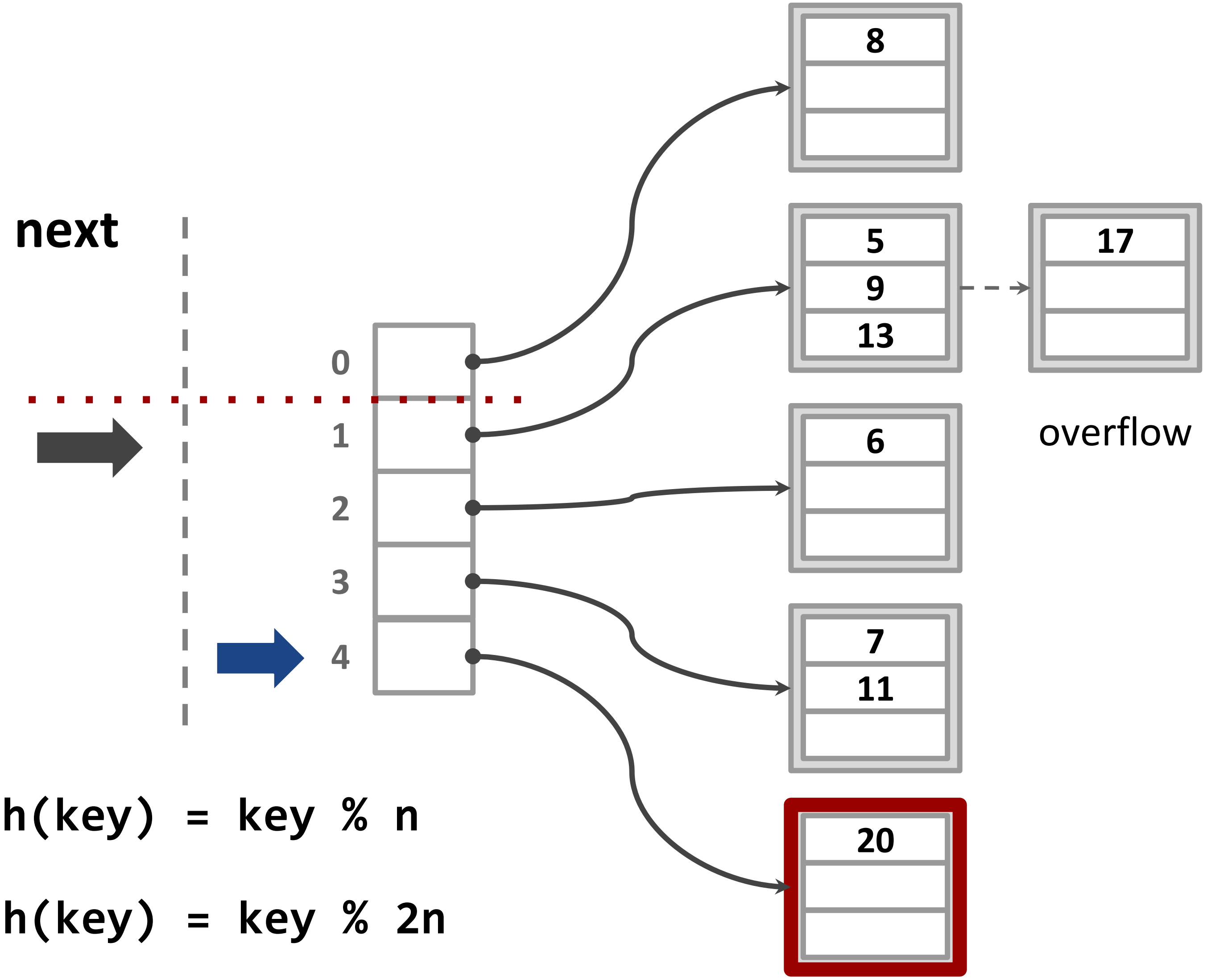
Insert 17

```
hash(17) = 17 % 4 = 1
hash(8)  =  8 % 8 = 0
hash(20) = 20 % 8 = 4
```

Search for 20

```
hash(20) = 20 % 4 = 0
```

Linear hashing illustration



`hash(key) = key % n`

`hash(key) = key % 2n`

Search for 6

`hash(6) = 6 % 4 = 2`

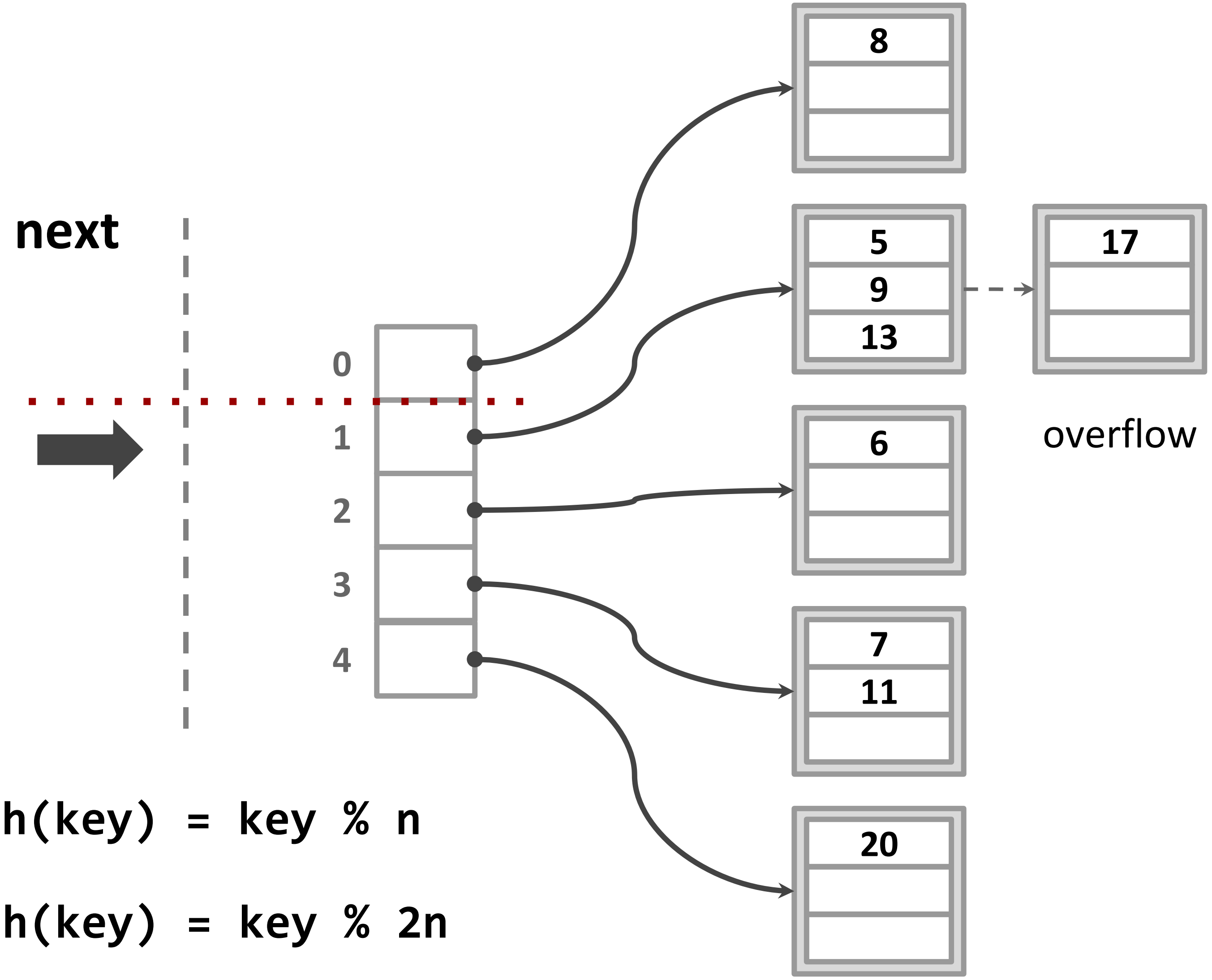
Insert 17

```
hash(17) = 17 % 4 = 1
hash(8)  =  8 % 8 = 0
hash(20) = 20 % 8 = 4
```

Search for 20

```
hash(20) = 20 % 4 = 0
hash(20) = 20 % 8 = 4
```

Linear hashing illustration




```
hash(key) = key % n

hash(key) = key % 2n
```

Search for 6

```
hash(6) = 6 % 4 = 2
```

Insert 17

```
hash(17) = 17 % 4 = 1
hash(8)  =  8 % 8 = 0
hash(20) = 20 % 8 = 4
```

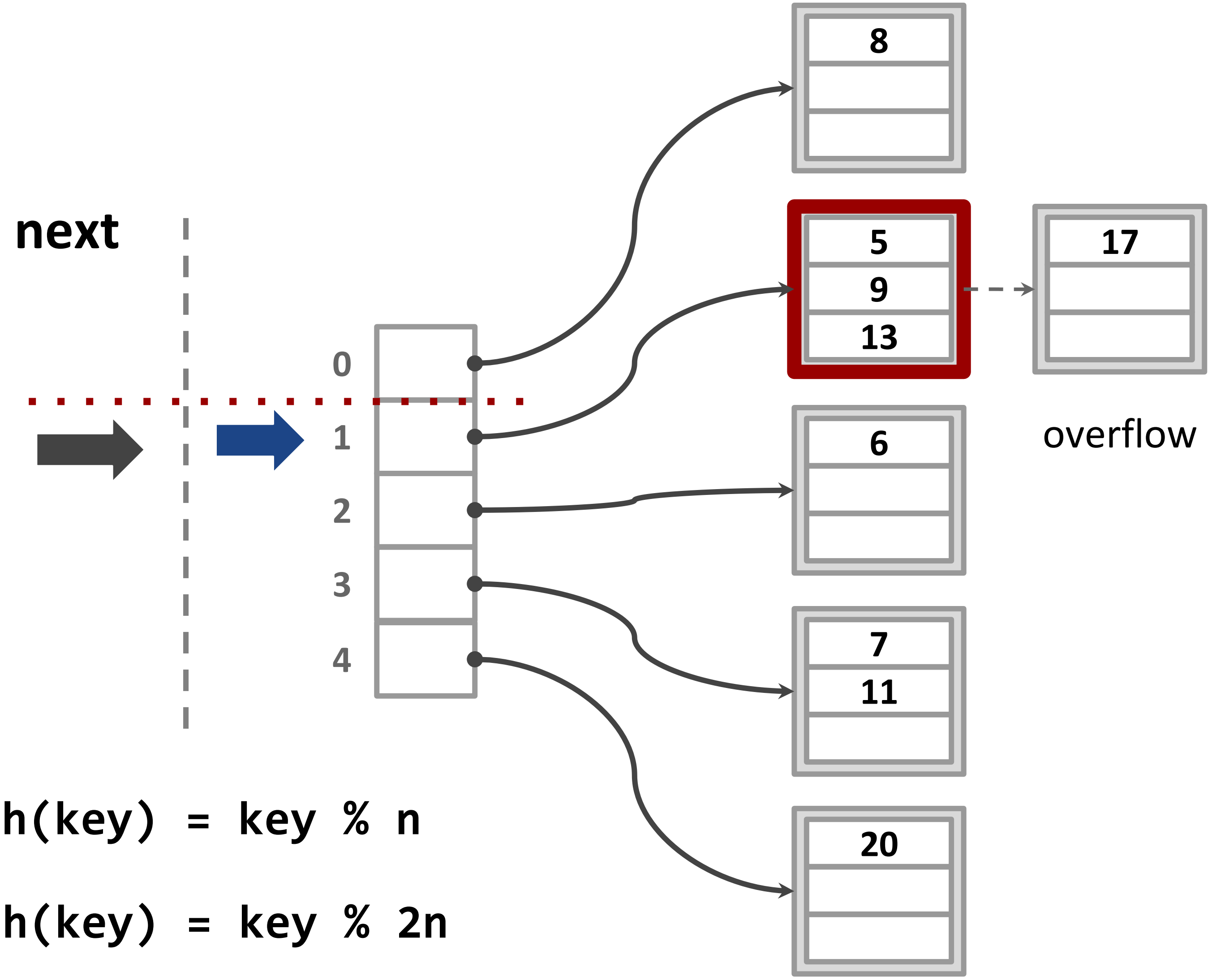
Search for 20

```
hash(20) = 20 % 4 = 0
hash(20) = 20 % 8 = 4
```

Search for 9

```
hash(9) = 9 % 4 = 1
```

Linear hashing illustration




```
hash(key) = key % n

hash(key) = key % 2n
```

Search for 6

```
hash(6) = 6 % 4 = 2
```

Insert 17

```
hash(17) = 17 % 4 = 1
hash(8)  =  8 % 8 = 0
hash(20) = 20 % 8 = 4
```

Search for 20

```
hash(20) = 20 % 4 = 0
hash(20) = 20 % 8 = 4
```

Search for 9

```
hash(9) = 9 % 4 = 1
```

Linear hashing: Resizing

- The splitting bucket strategy (based on the split pointer) will eventually reach all overflowed buckets
  - When the **next** pointer reaches the last slot, remove the old hash function and move assign the pointer back to the first bucket

Linear hashing: why do we need it?

- Handles data insertion in a more gradual and controlled fashion
- Spreads the rehashing across insertions (more concurrency)
  - Only one bin/page is rehashed at a time…
  - …while other threads can access other parts of the table
  - Better than extendible hashing: it needs to rehash only when the global-depth changes
- Good for cases where dataset size changes over time
- But:
  - Needs a good hash function
  - Increased access time due to overflow tables

Summary of hash table indexes

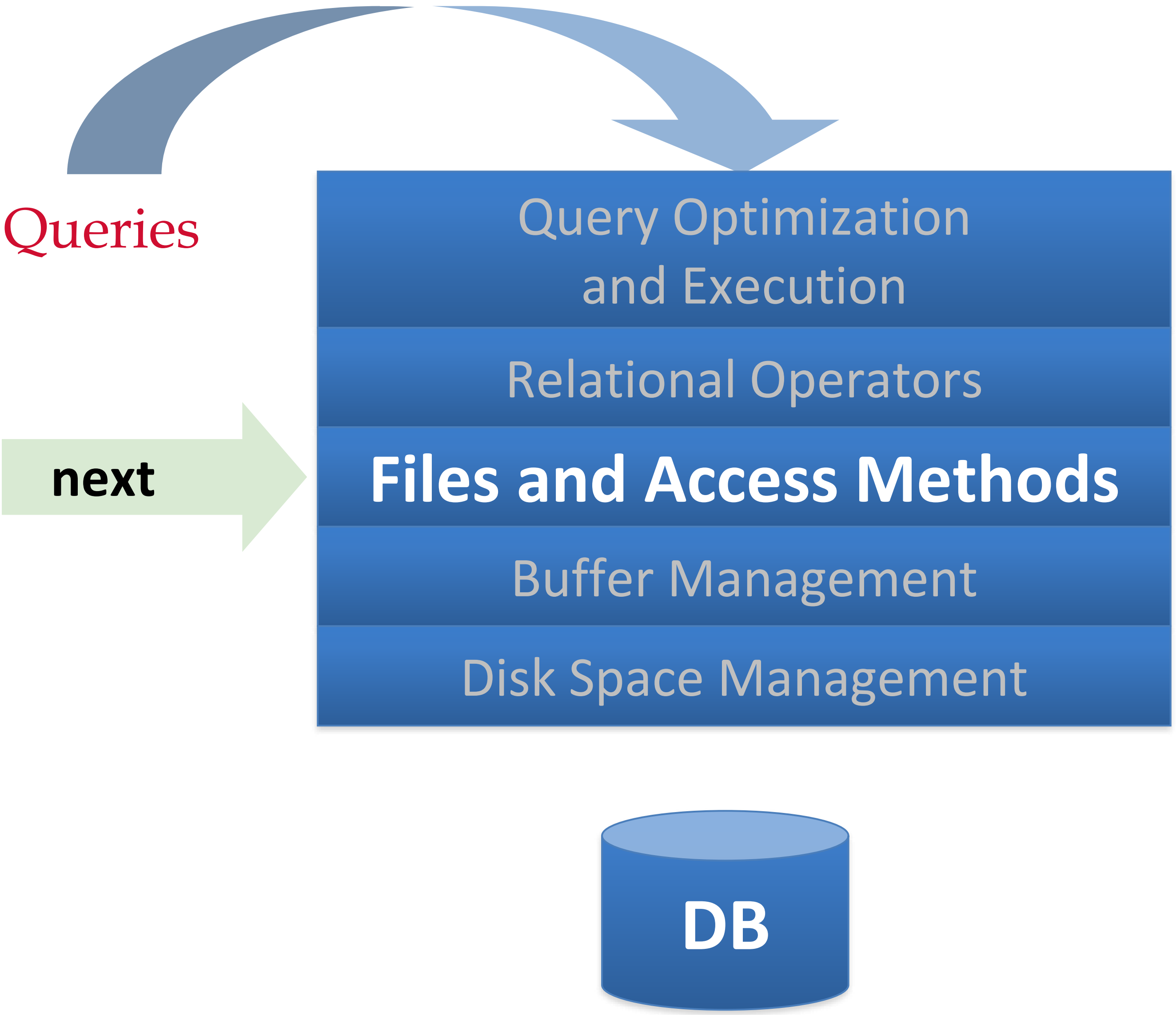
- Hash-based indexes are best for equality searches but do not support range searches
- Static hashing can lead to long overflow chains
- Extendible hashing
  - Avoids overflow pages by splitting a full bucket when a new data entry is to be added to it
  - Directory can keep track of buckets, doubles periodically
  - Can get large with skewed data; additional IO if the table does not fit in main memory

Today's focus

- Hash-based indexes
- **Sorting**

DBMS bigger picture

How DBMS executes queries using the DBMS components, when data can be **unsorted**



Disk-oriented DBMS

- A DBMS does not assume that a table fits entirely in main memory, a disk-oriented DBMS cannot assume that a query result can fit in memory

- We use the buffer pool to implement algorithms that need to spill to disk

- We prefer algorithms that maximize the amount of sequential IO
  - Better utilization of disk (sequential IO > random IO)

Need for sorting data

- Relational model/SQL is **unsorted**
- Queries may request that tuples are sorted in a specific way (**ORDER BY)**
- But even if a query does not specify an order, we may still want to sort to do other things:
  - Remove duplicates (**DISTINCT**)
  - Bulk sorted tuples into B+-tree index is faster
  - Aggregations (**GROUP BY)**

- Sorting in memory: well-studied problem (quicksort, heapsort)
- In DBMS: sort 100 GB with 100 MB of memory

Sorting

- 2-way external sorting
- General external sorting and performance analysis
- Using B+-trees for sorting

2-way external sort

- A simple example of a 2-way external (merge) sort
  - "2" is the number of runs that we are going to merge into a new run for each pass

- Data is broken up into **N** pages

- DBMS has a finite number of **B** buffer pool pages to hold input and output data

Simplified 2-way external sort

Pass #0

- Read one page of the table into memory
- Sort the page into a "run" and write it back to disk
- Repeat until the whole table has been sorted into runs

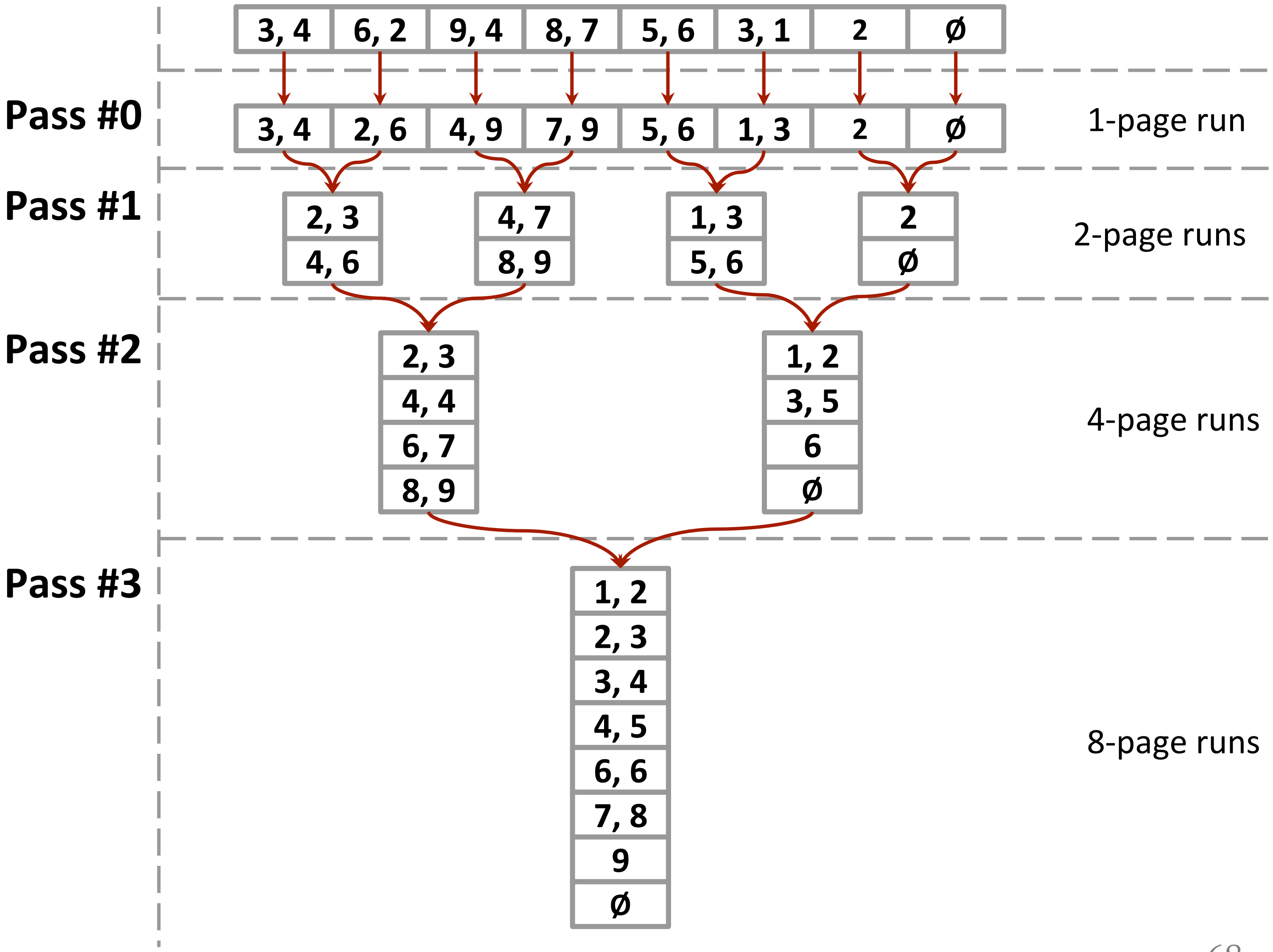
Pass #1,2,3 …

- Recursively merge pairs of runs into runs twice as long
- Needs at least 3 buffer pages (2 for input and 1 for output)

Simplified 2-way external sort

- In each pass, we read and write every page in the file
- Number of passes $= 1 + \lceil \log_2 N \rceil$
- Total IO cost $= 2N * (1 + \lceil \log_2 N \rceil)$

Idea: Divide and conquer: sort subfiles and merge



General external sort

Pass #0

- Use **B** buffer pages
- Produce **⌈N/B⌉** sorted runs of size **B**

Pass #1,2,3 …

- Merge **B-1** runs (i.e., k-way merge)

Number of passes = $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$

Total I/O cost = $2N * (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$

General external sort: example

Determine how many passes it takes to sort 108 pages with 5 buffer pool pages

N = 108, B = 5

- **Pass #0:** $\lceil N/B \rceil = \lceil 108/5 \rceil = 22$ sorted runs of 5 pages each (last run is 3 pages)
- **Pass #1:** $\lceil N'/B-1 \rceil = \lceil 22/4 \rceil = 6$ sorted runs of 20 pages each (last run is 8 pages)
- **Pass #2:** $\lceil N''/B-1 \rceil = \lceil 6/4 \rceil = 2$ sorted runs of 80 pages and 28 pages
- **Pass #3:** Sorted file of 108 pages

$$1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil = 1 + \lceil \log_4 \lceil 22 \rceil \rceil = 1 + \lceil 2.229 \rceil = 4 \text{ passes}$$

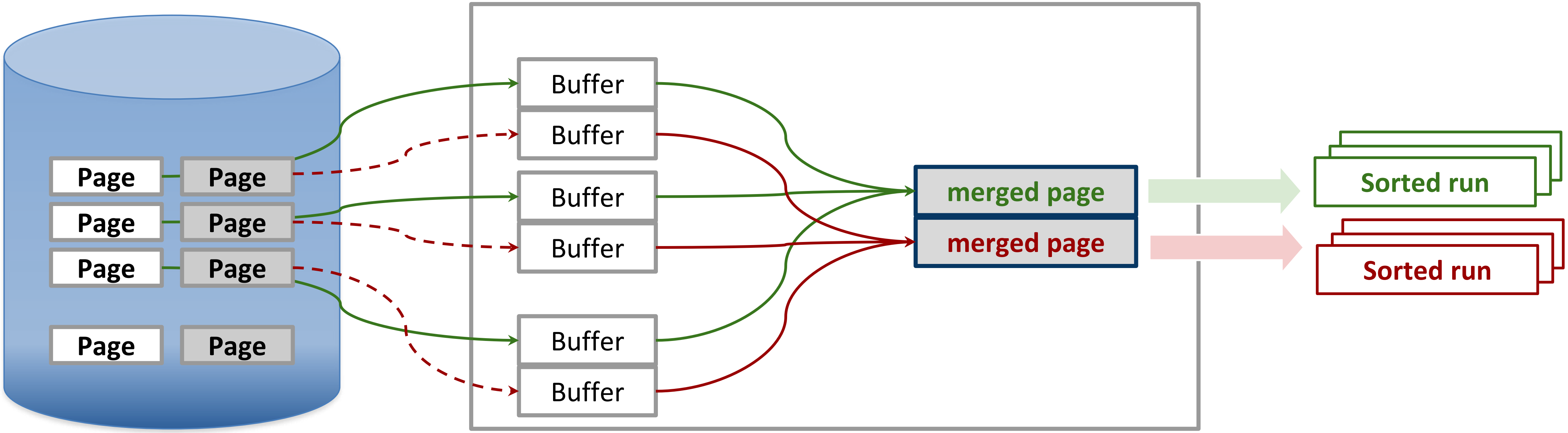
Double buffering optimization

- Prefetch the next run in the background and store it in a second buffer while the system is processing the current run

  - Reduces the wait time for IO requests at each step by overlapping disk transfer time with computation

Double buffering optimization

- Prefetch the next run in the background and store it in a second buffer while the system is processing the current run
  - Overlaps CPU and IO operations
- Reduces the effective "B" by half
- Reduces the response time



Using B^+ Tree for sorting

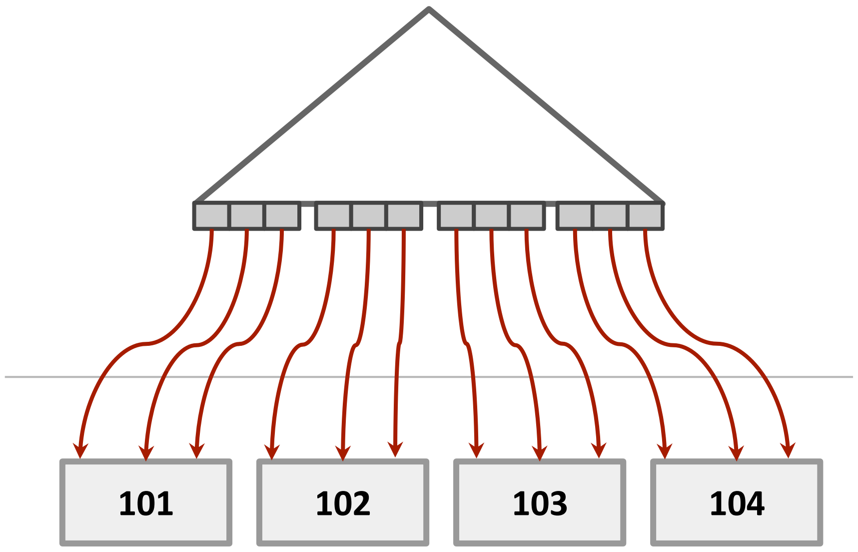
- If the table that must be sorted already has a B^+ Tree index on the sort attribute(s), then we can use that to accelerate sorting

- Retrieve records in desired sort order by simply traversing the leaf pages of the tree

- Consider the case:
  - Clustered B^+ Tree: **Good idea**
  - Unclustered B^+ Tree: **Could be a very bad idea**

Sort Using a Clustered B^+ Tree…

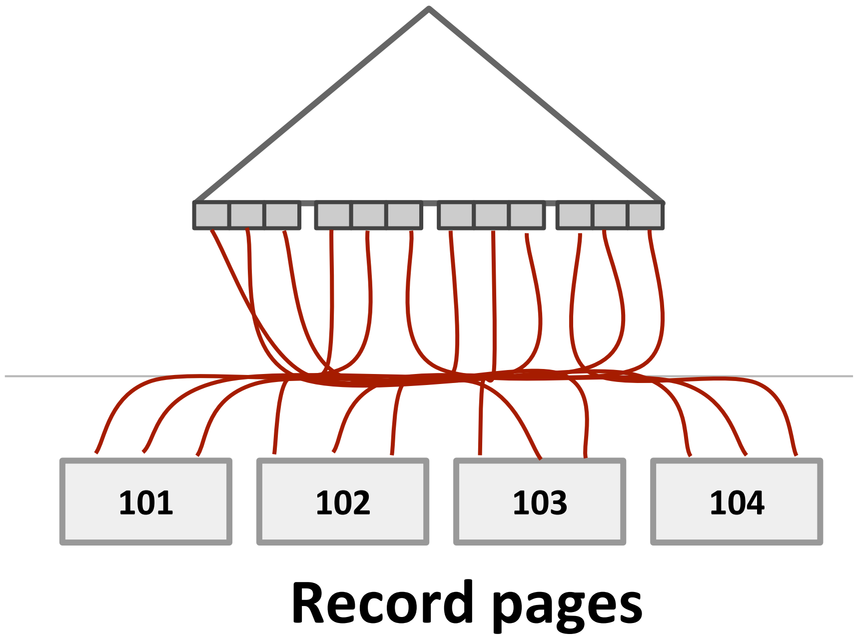
- Traverse to the left-most leaf page, and then retrieve records from all leaf pages
- This is always better than external sorting:
  - No computational cost
  - All disk accesses are sequential



Record pages

…or Sort Using an Unclustered B^+ Tree

- Chase each pointer to the page that contains the data
- Worst case, one I/O per data record
- Always a bad idea! Instead, sorting is a better idea



External sorting: summary

- Sorting a file while optimizing for I/O is very useful for query processing
- External merge sort minimizes disk I/O cost as follows:
  - # runs merged at a time depends on B and block size
  - Larger block size: lower I/O cost and smaller number of runs merged
  - In practise, # of runs rarely more than 2 or 3
- Choice of internal sort affects the performance
  - Quicksort is better, heap is slower (2x)
- Clustered B^+Tree is good for sorting
- Unclustered B^+Tree is usually very bad