Storage & Indexing

Abdu Alawini

University of Illinois at Urbana-Champaign

CS411: Database Systems

July 11, 2020

Thanks to Doris Xi for her help with preparing the slides.

Leaning Objectives

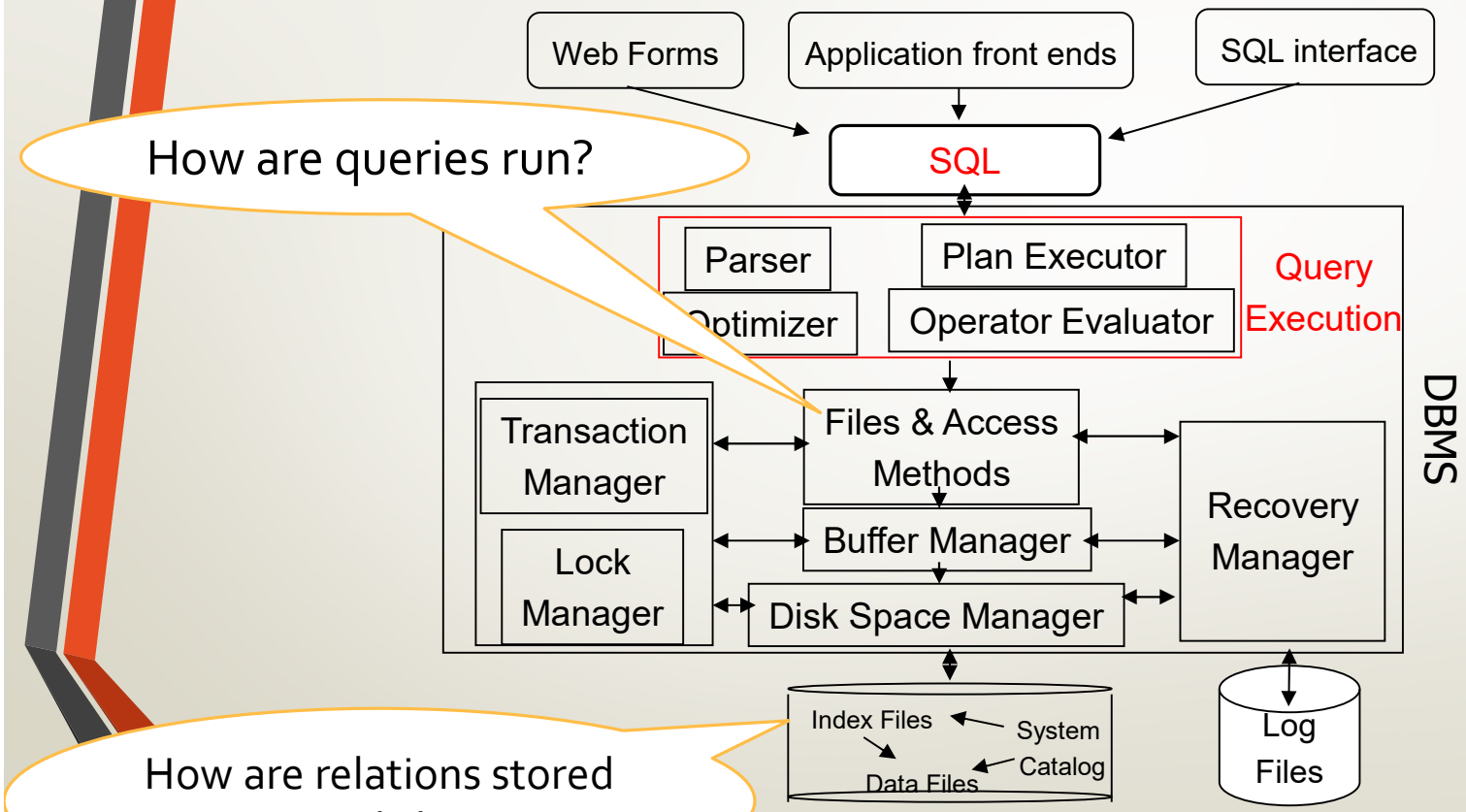
After this lecture, you will learn:

- how relations are stored on disk
- how index structures speed up data access
- the basics of B+ trees
- searching, inserting and deleting keys from B+ Trees
- searching, inserting and deleting keys from
  - Secondary storage Hash Table (HT)
  - Extensible HT
  - Linear HT

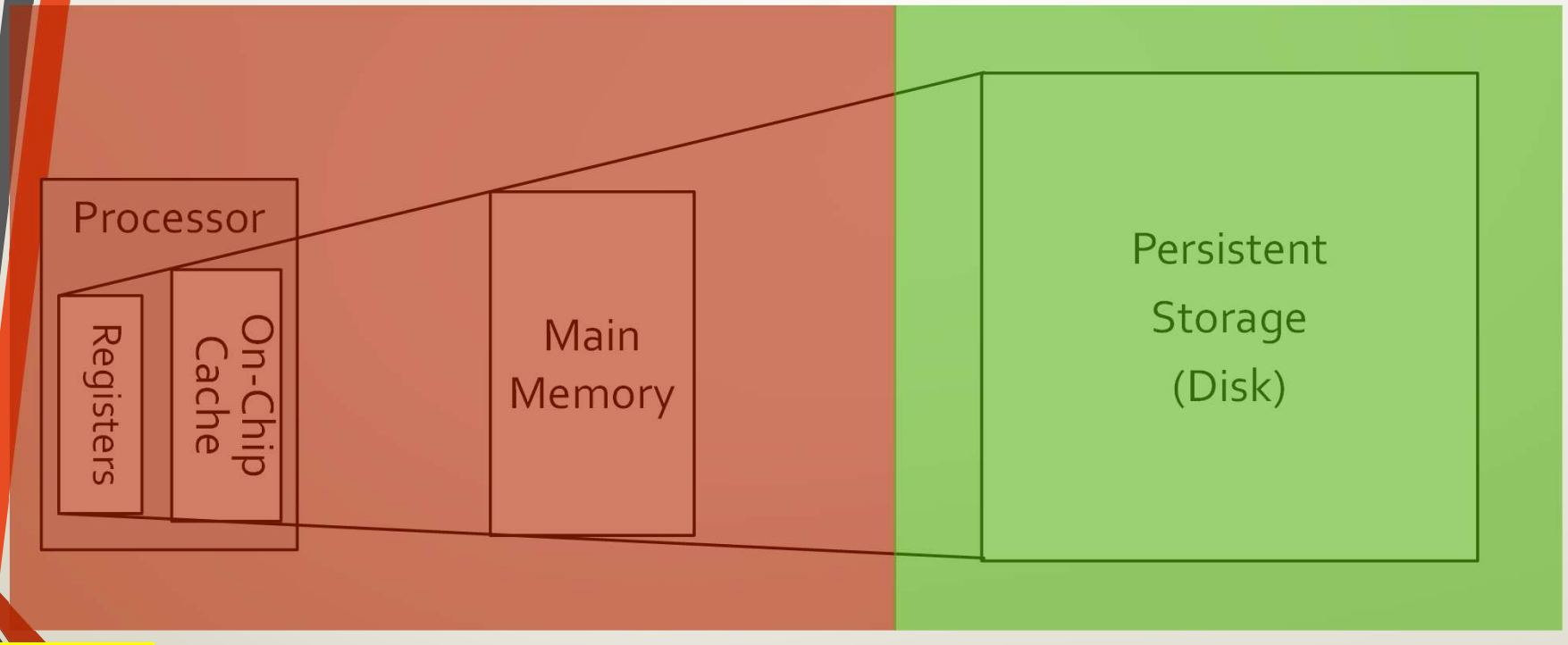
CS411 Goals:
Two Perspectives of DBMS

- USER PERSPECTIVE
  - **how to use a database system?**
  - conceptual data modeling, the relational and other data models, database schema design, relational algebra, SQL and No-SQL query languages.
- SYSTEMS PERSPECTIVE
  - **how to design and implement a database system?**
  - data representation, indexing, query optimization and processing, transaction processing, and concurrency control.
  - NOT COMPLETE: high-level view of implementation; CS511

DBMS Architecture

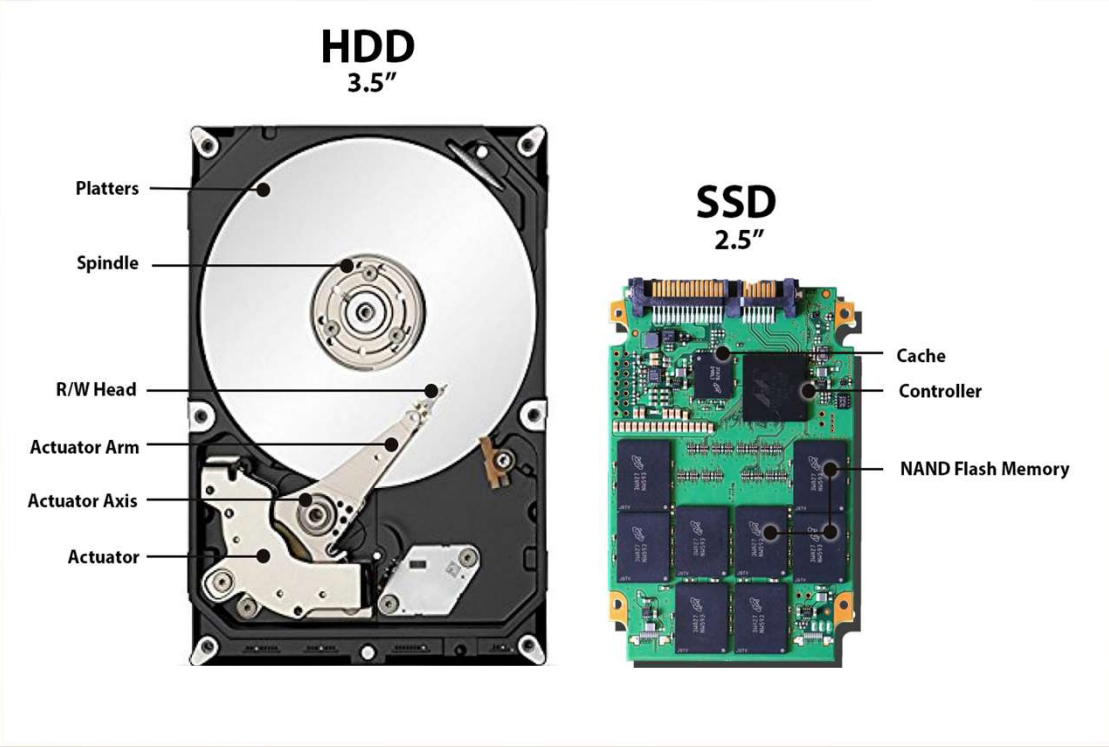


Simplified Computer Architecture



Speed (ns)	10 ns	10^2 ns	10^7 ns
Size	KB	GB	TB

Cost of Accessing Data on Disk



Main Memory

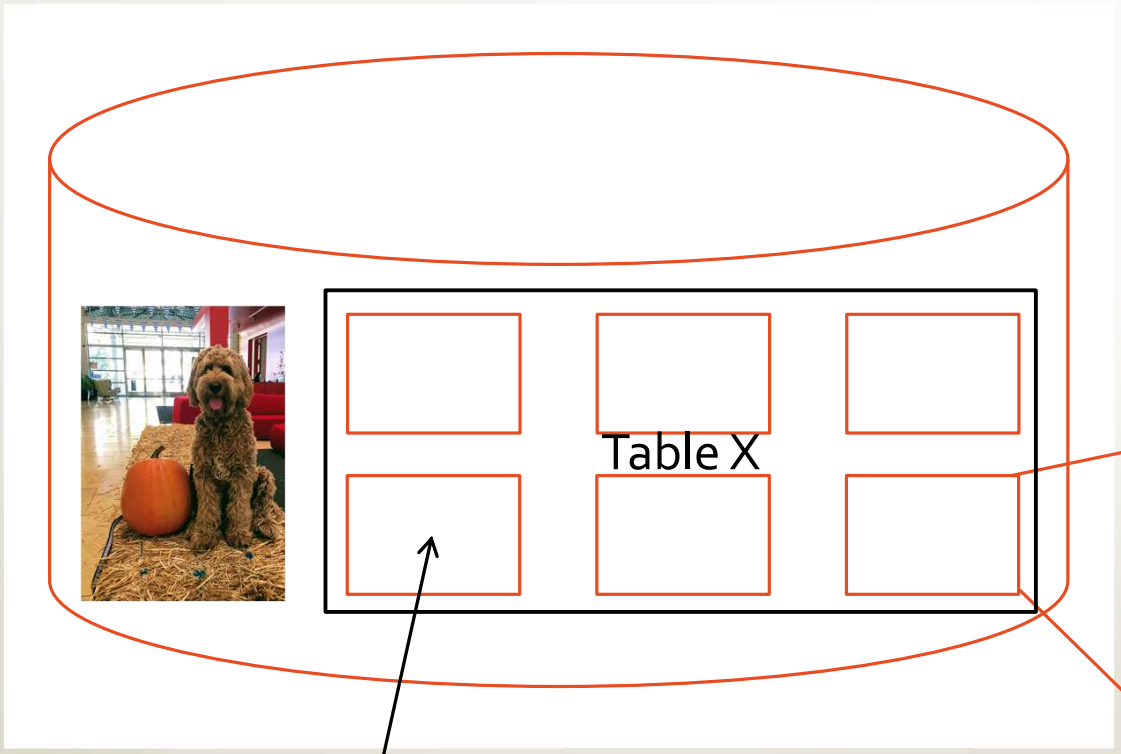
Speed

10^7 ns

10^5 ns

10^2 ns

Block size vs. record size

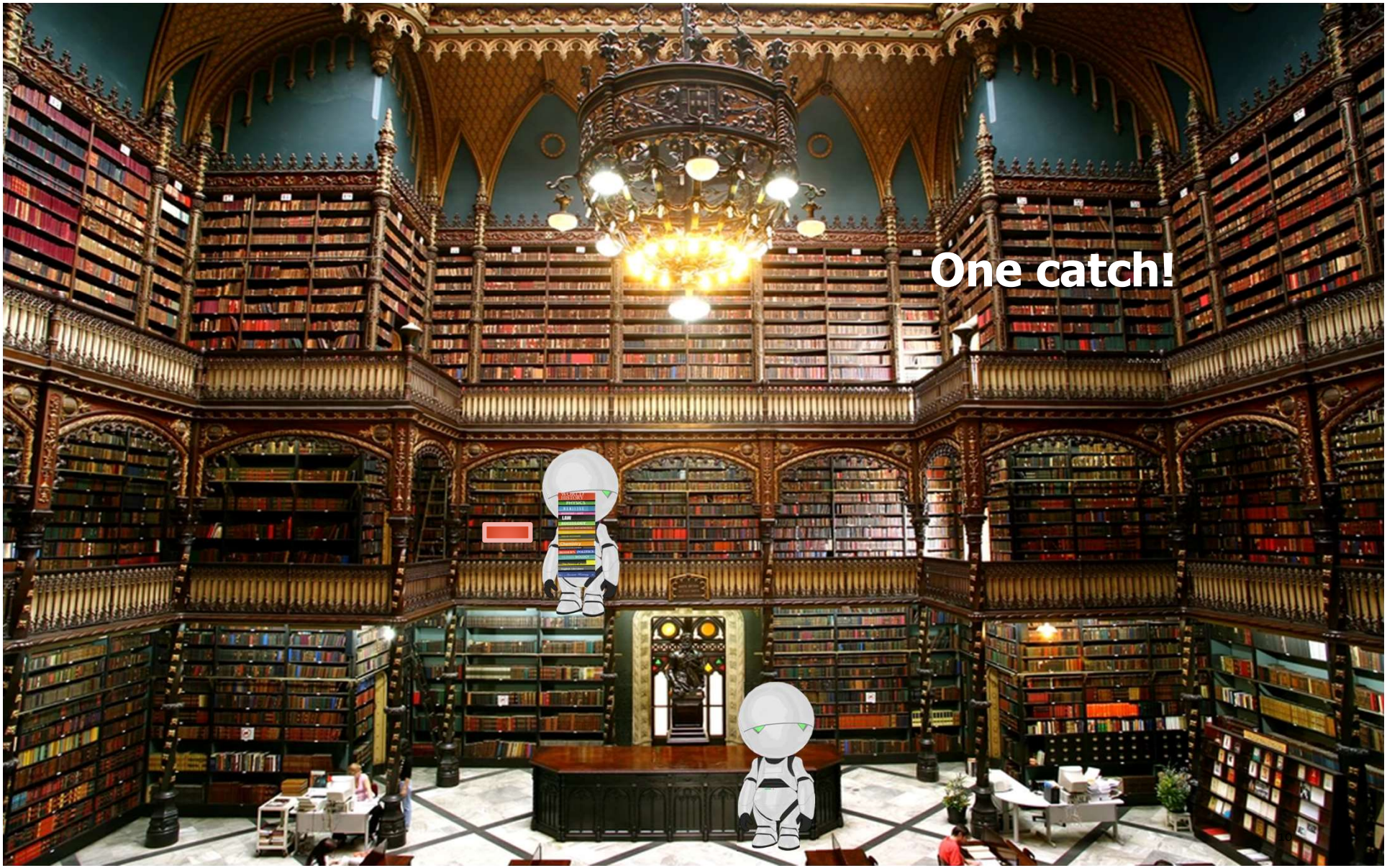


OK. So how do we do simple operations?

Lookups. Insertions. Deletions

Outline

- ✓ Storage
- Indexing
  - What is an index? Why do we need it?
- B+ Trees
  - Basics and Searching
  - Inserting
  - Deletion
- Hash Tables
  - Secondary storage HT
  - Extensible HT
  - Linear HT


One catch!

Indexes in databases

- An *index* speeds up selections on the *search key field(s)*
- Search key = any subset of the fields of a relation
  - *Search key* is not necessarily the same as a *key*
- Entries in an index: (k, r), where:
  - k = the search key
  - r = the record OR record id OR record ids OR pointers

Some terminology

- ***Data file***: has the data corresponding to a relation
- ***Index file***: has the index
- File consists of smaller units called **blocks** (e.g. of size 4 KB or 8 KB)
- # index blocks < # data blocks. Index may even fit into main memory.

An Index is a Function!

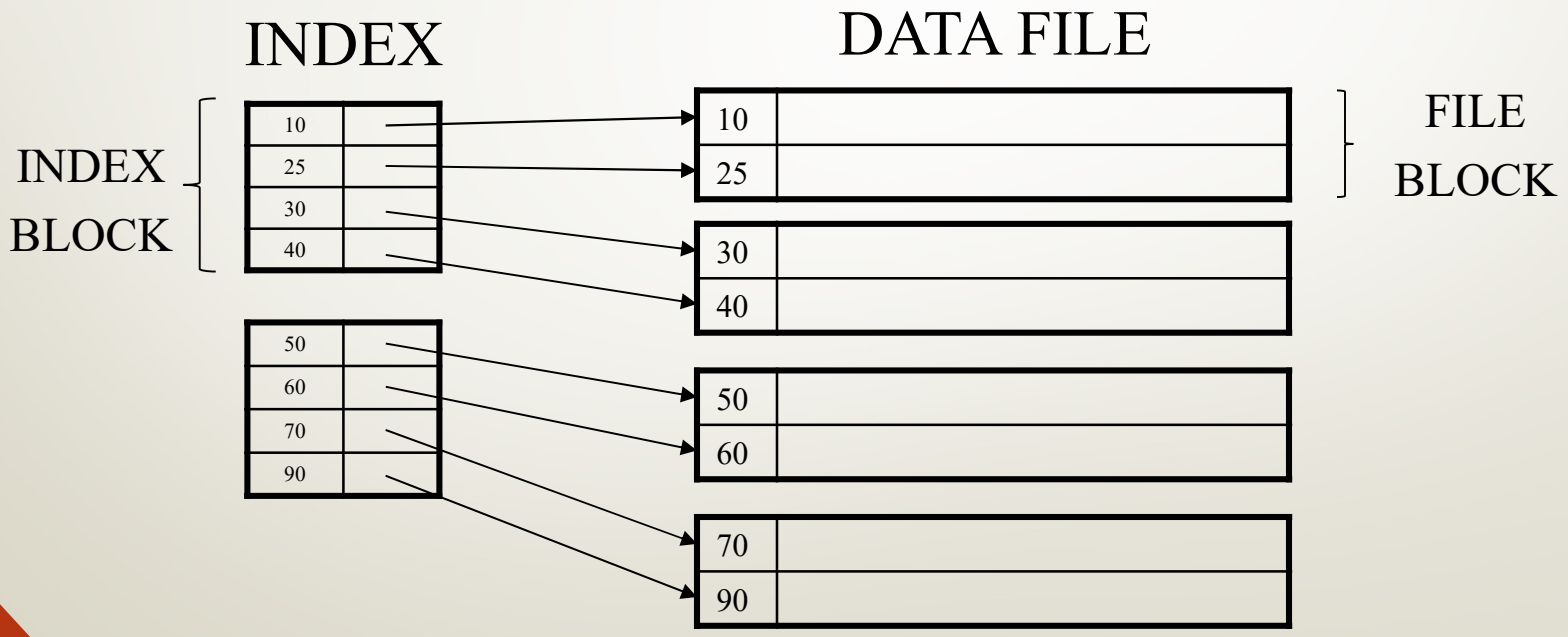
f(what: key) = where: file block

Characteristics of Indexes

- Clustered/unclustered
  - Clustered: records sorted in the search key order
  - Unclustered: records are NOT sorted in the search key order
- Dense/sparse
  - Dense = each record has an entry in the index
  - Sparse = only some records have
- Primary/secondary
  - Primary = on the primary key
  - Secondary = on any attribute

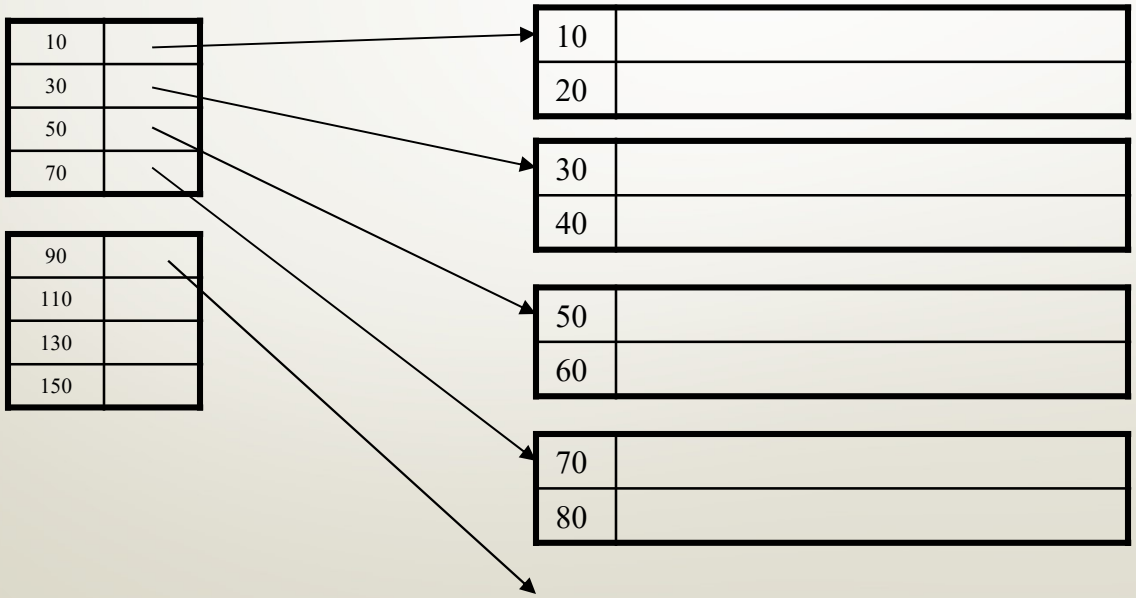
Ex: Clustered, Dense Index

- Clustered: File is sorted on the index attribute
- *Dense*: sequence of (key,pointer) pairs



Clustered, Sparse Index

- *Sparse* index: one key per data block, corresponding to the lowest search key in that block

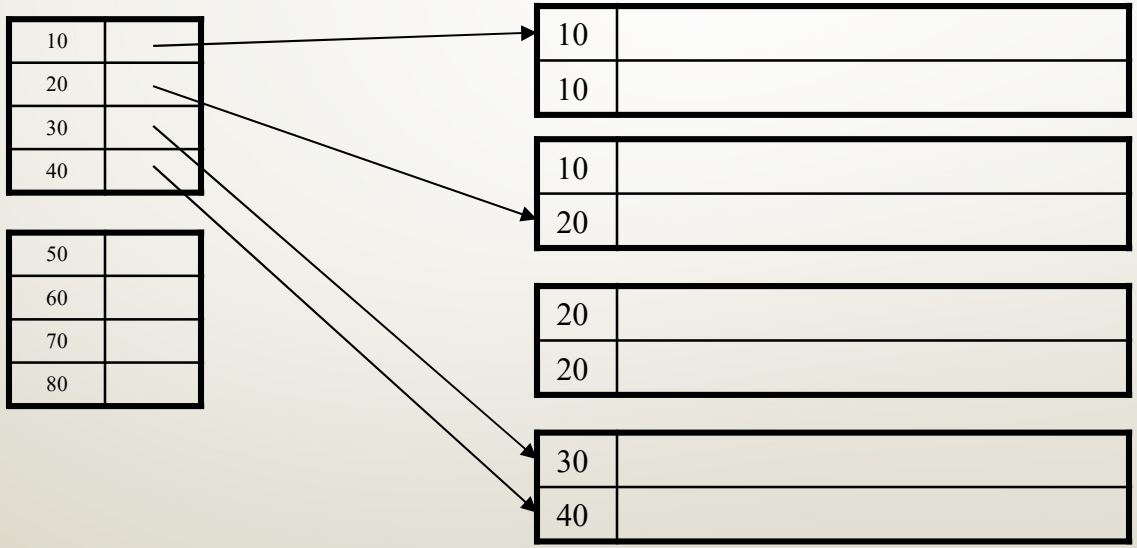


What if there are duplicate keys?

Clustered Index with Duplicate Keys

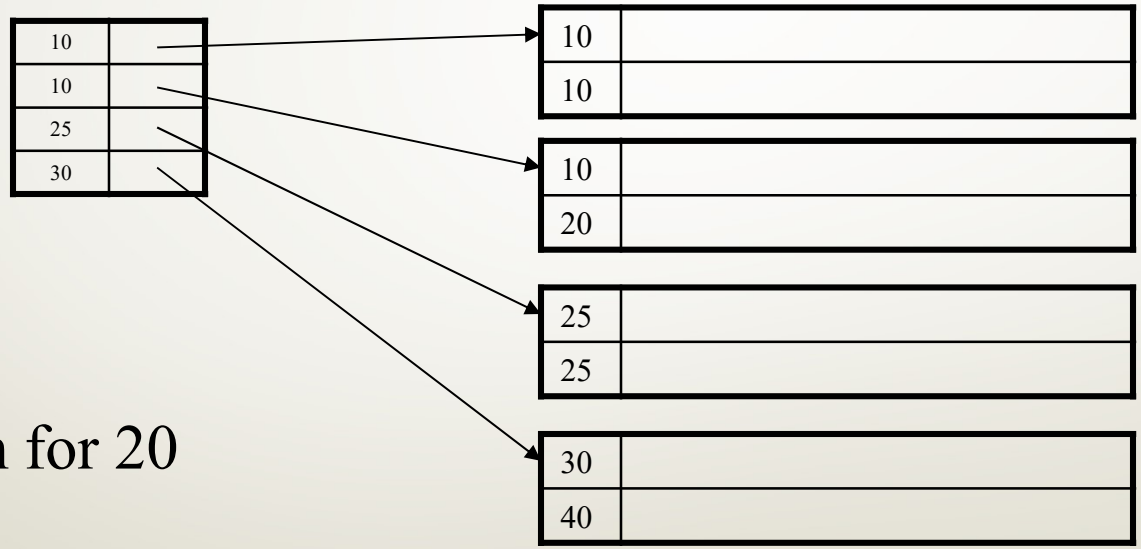
Dense index: point to the first record with that key

(must have a pointer for each new key)



Clustered Index with Duplicate Keys

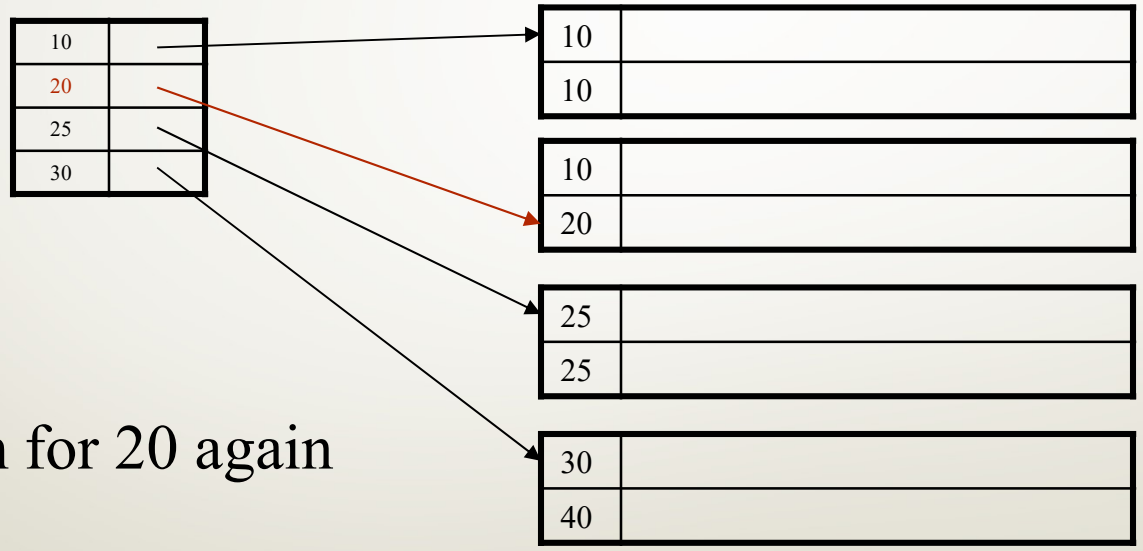
- Sparse index: pointer to lowest search key in each block



Search for 20

Clustered Index with Duplicate Keys

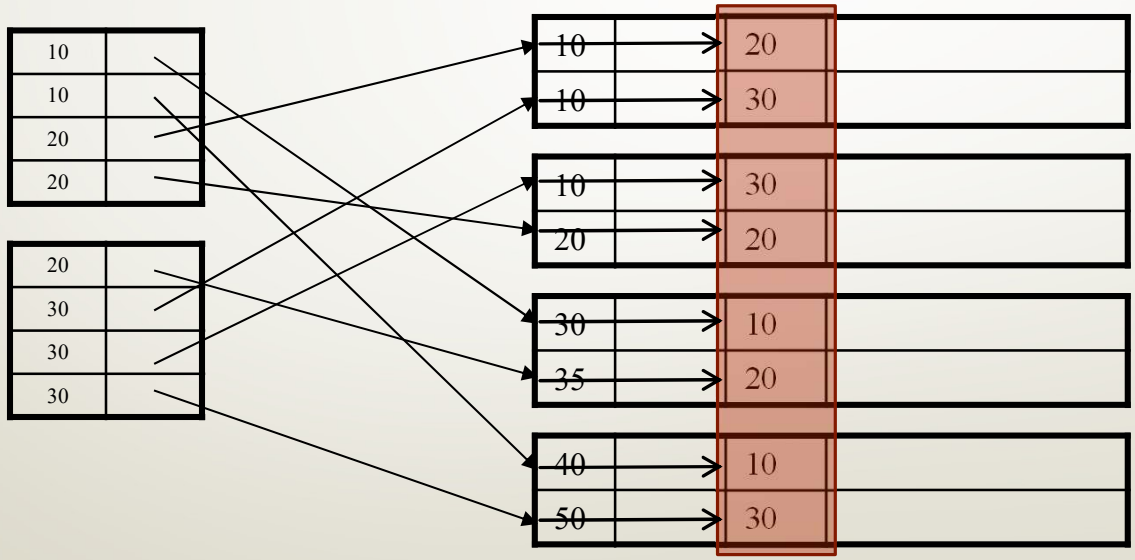
- *Better: pointer to lowest new search key in each block*



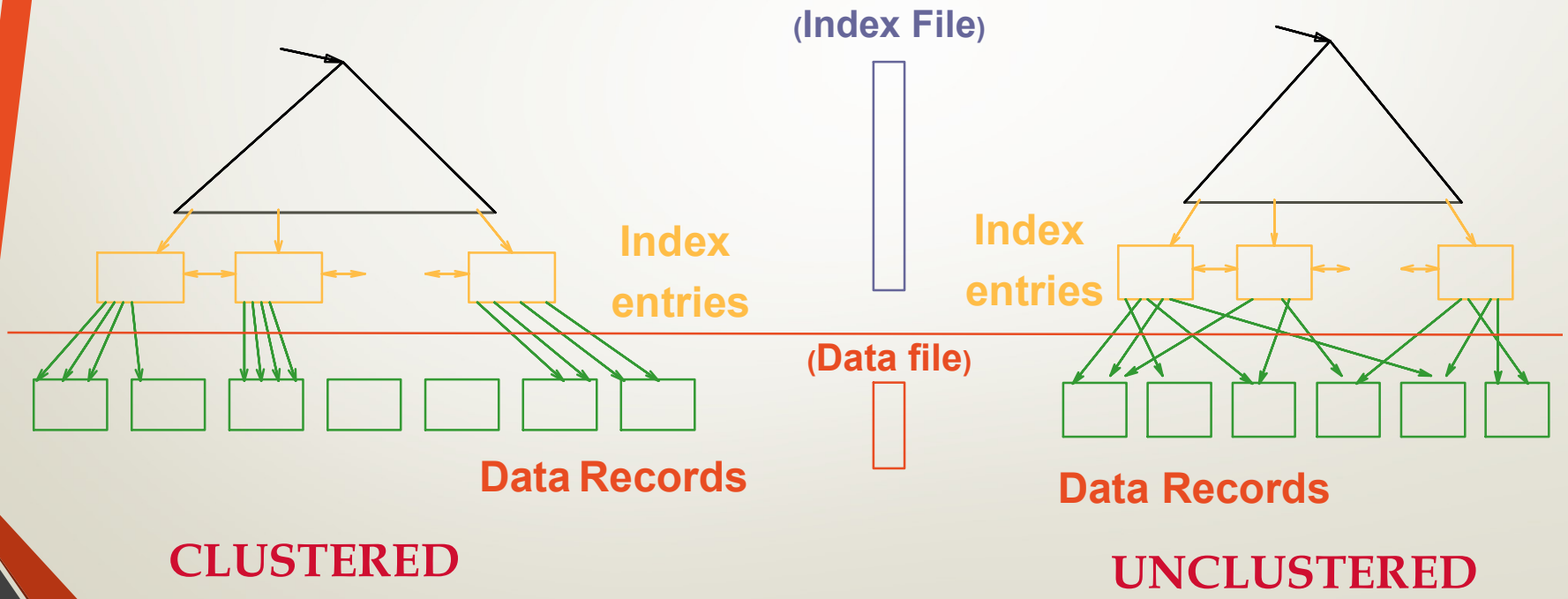
Search for 20 again

Unclustered Indexes

- Often for indexing other attributes than primary key
- Can it be sparse?



Summary Clustered vs. Unclustered Index



Clustered/Unclustered Dense/Sparse

	Dense	Sparse
Clustered	Yes	Yes
Unclustered	Yes	No

Outline

- ✓ Storage
- ✓ Indexing
  - ✓ What is an index? Why do we need it?
- B+ Trees
  - Basics and Searching
  - Inserting
  - Deletion
- Hash Tables
  - Secondary storage HT
  - Extensible HT
  - Linear HT

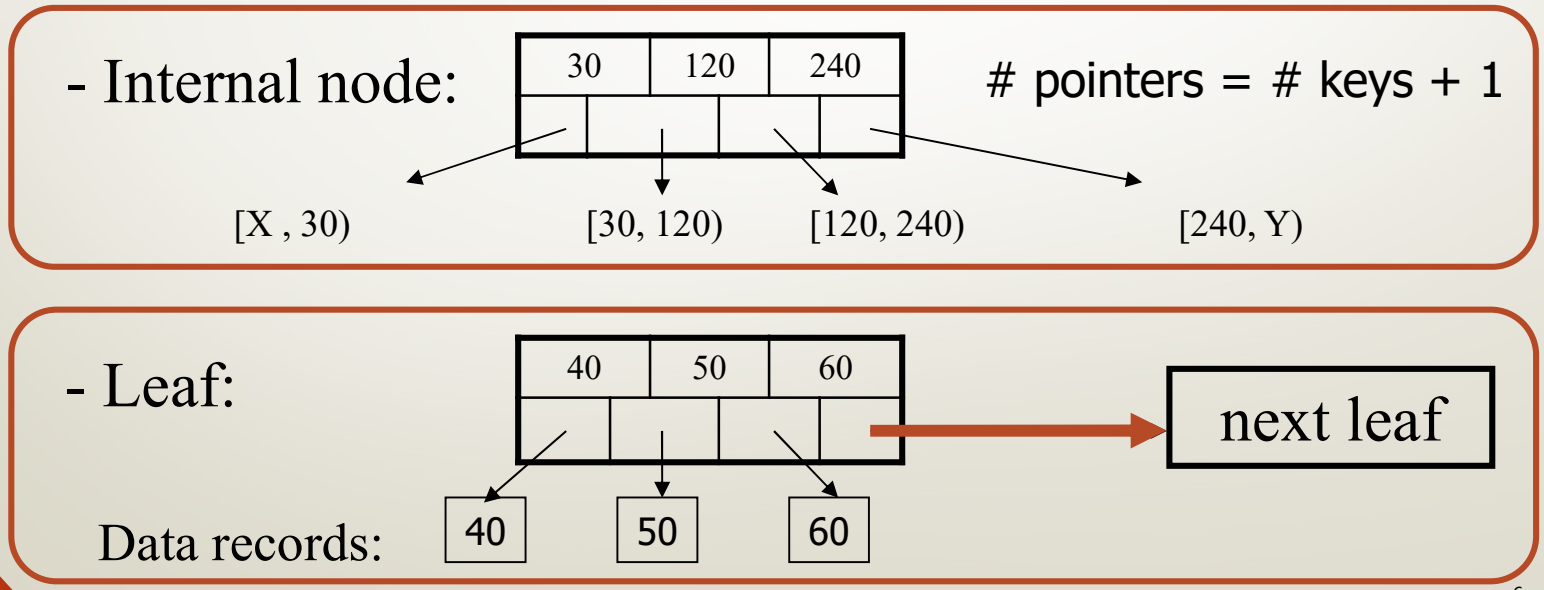
B+ Trees

- Intuition:
  - The index can be very large.
  - Index of index?
  - Index of index of index?
  - How best to create such a multi-level index?
- B+ trees:
  - Textbook refers to B+ trees (a popular variant) as B-trees (as most people do)

Focus on the dense version:
applies to clustered and unclustered settings

B+ Trees Basics

- B+ Trees are trees with nodes:
  Nodes have keys and pointers to:
  - Other nodes [if the node is an internal node]
  - Data Records [if the node is a leaf]



B+ Trees Basics

- Parameter d = the *degree* ; n = max keys
- When n is even [*this is our focus for simplicity*]
  - each node has [d, 2d] keys (except root); $n = 2d$
- At least half full at all times
  - d is the minimum amount it needs to be full.

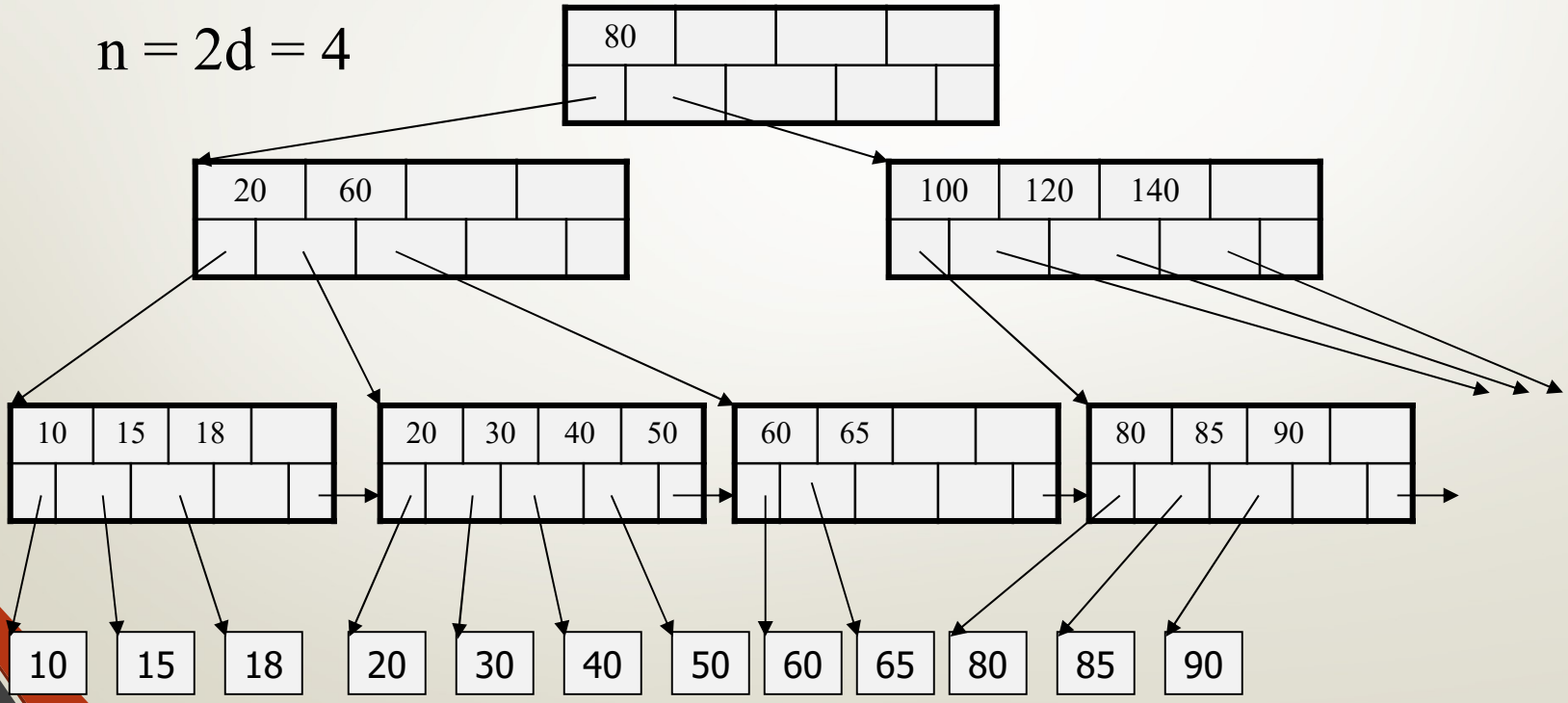
B+ Tree Example

Root can have 1 or more filled in keys

Rest have at least d

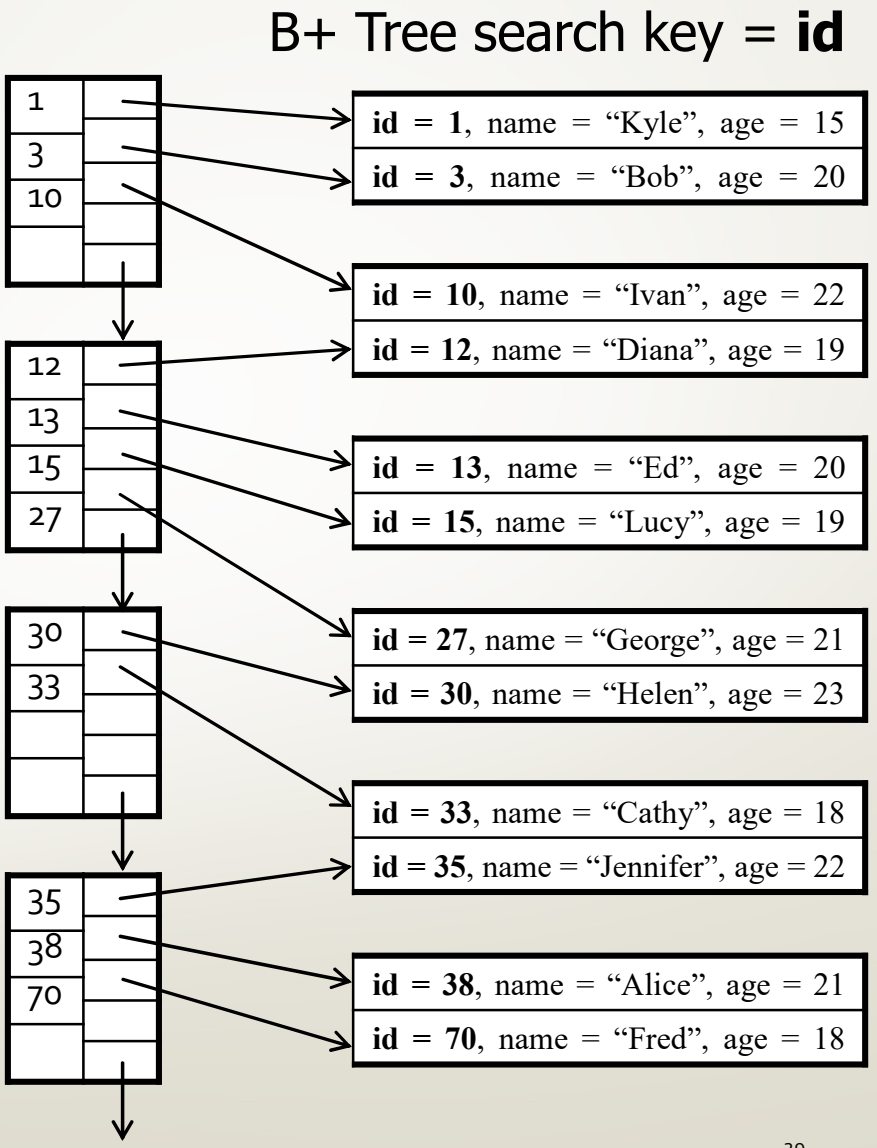
d = 2

n = 2d = 4



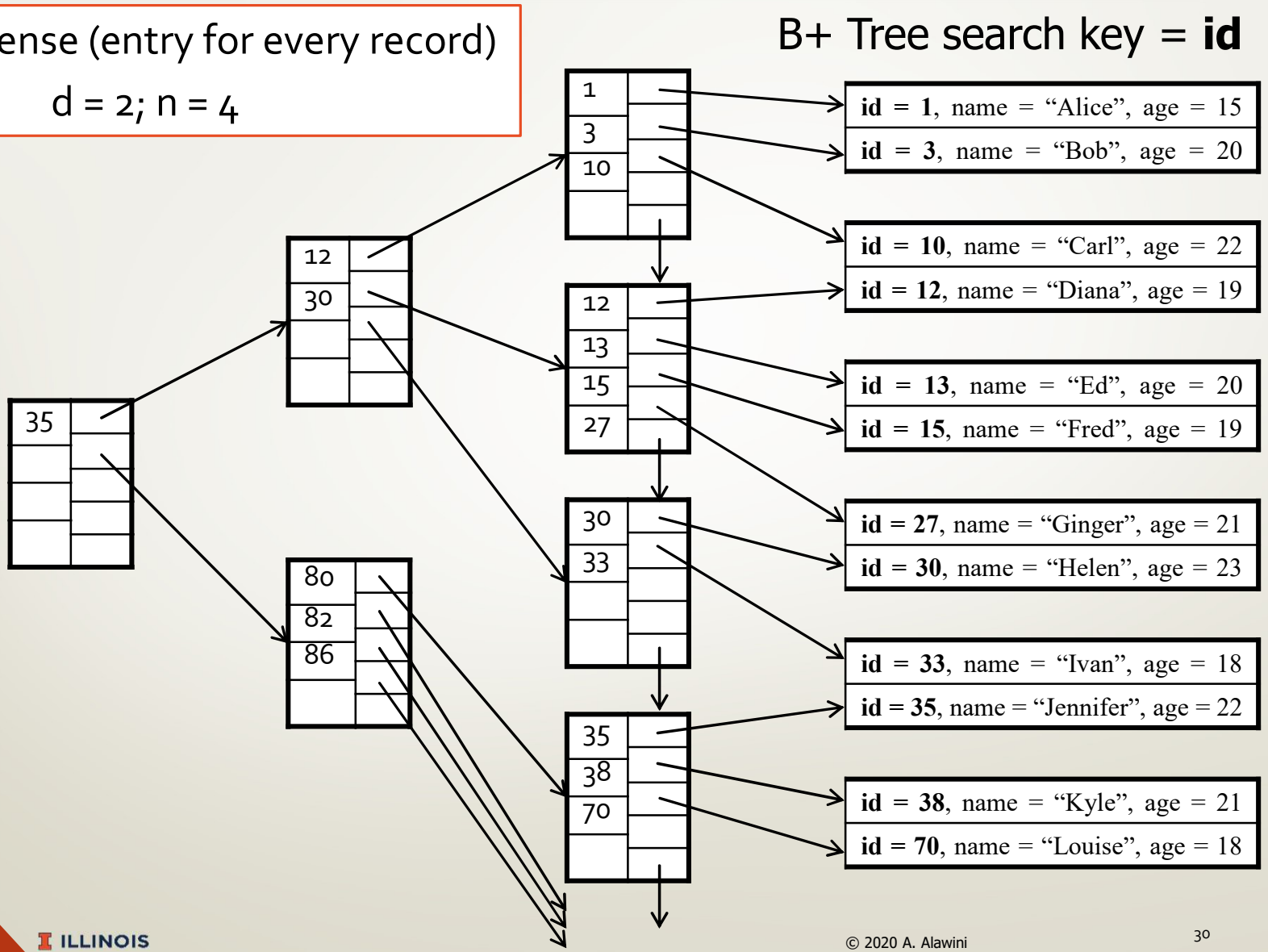
Clustered dense (entry for every record)

d = 2; n = 4



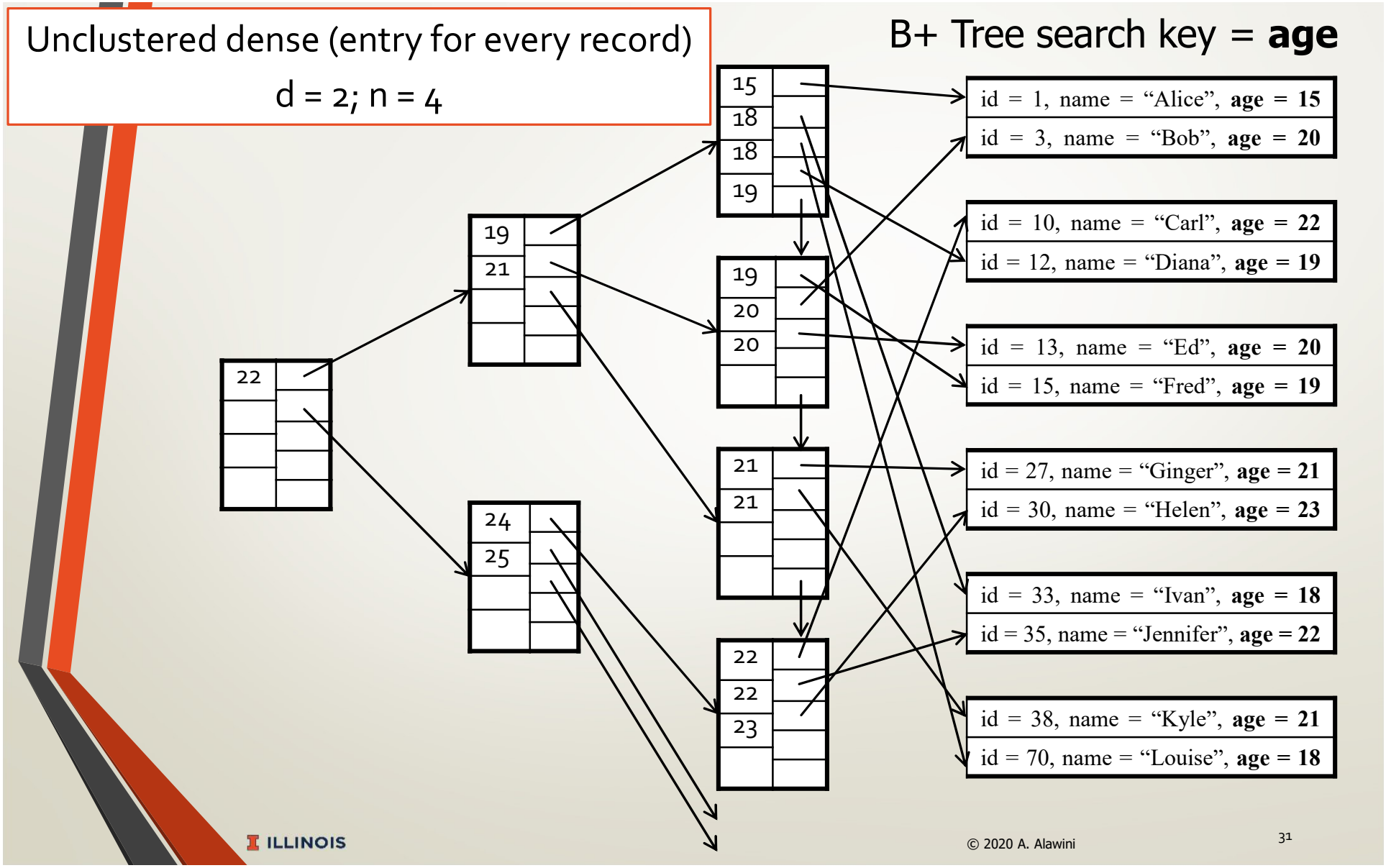
Clustered dense (entry for every record)

d = 2; n = 4



Unclustered dense (entry for every record)

d = 2; n = 4



B+ Tree Design

- How large should *d* be?
- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 byes
- 2d x 4 + (2d+1) x 8 <= 4096
- d = 170; 2d = 340

So up to 340 records in leaf blocks

B+ Trees in Practice

- Typical d: 100. Typical fill-factor: 66.5%.
  - average "fanout" = 66.5 * 2 =133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in main memory:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

When Do B+ Trees Help?

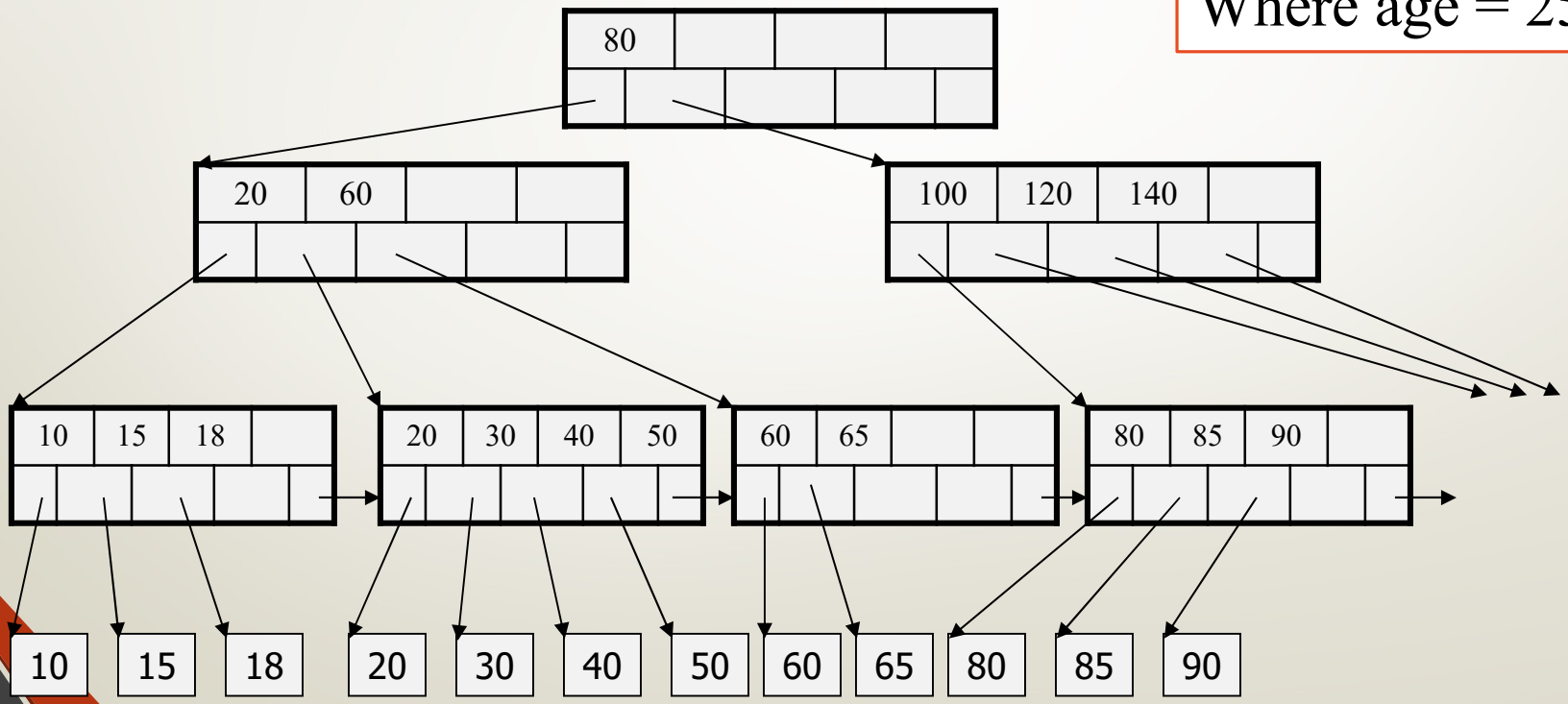
- Do B+ Trees always help?
  - No. e.g., an array of sorted integers.
- Types of queries to answer with a B+ Tree:
  - *Exact key value*, e.g., SELECT name FROM people WHERE age=20
  - *Range queries*, e.g., SELECT name FROM people WHERE age>=20 and age<=70

Searching a B+ Tree

Exact key values:

- Start at the root;
- Proceed down to the leaf

Select name
From people
Where age = 25

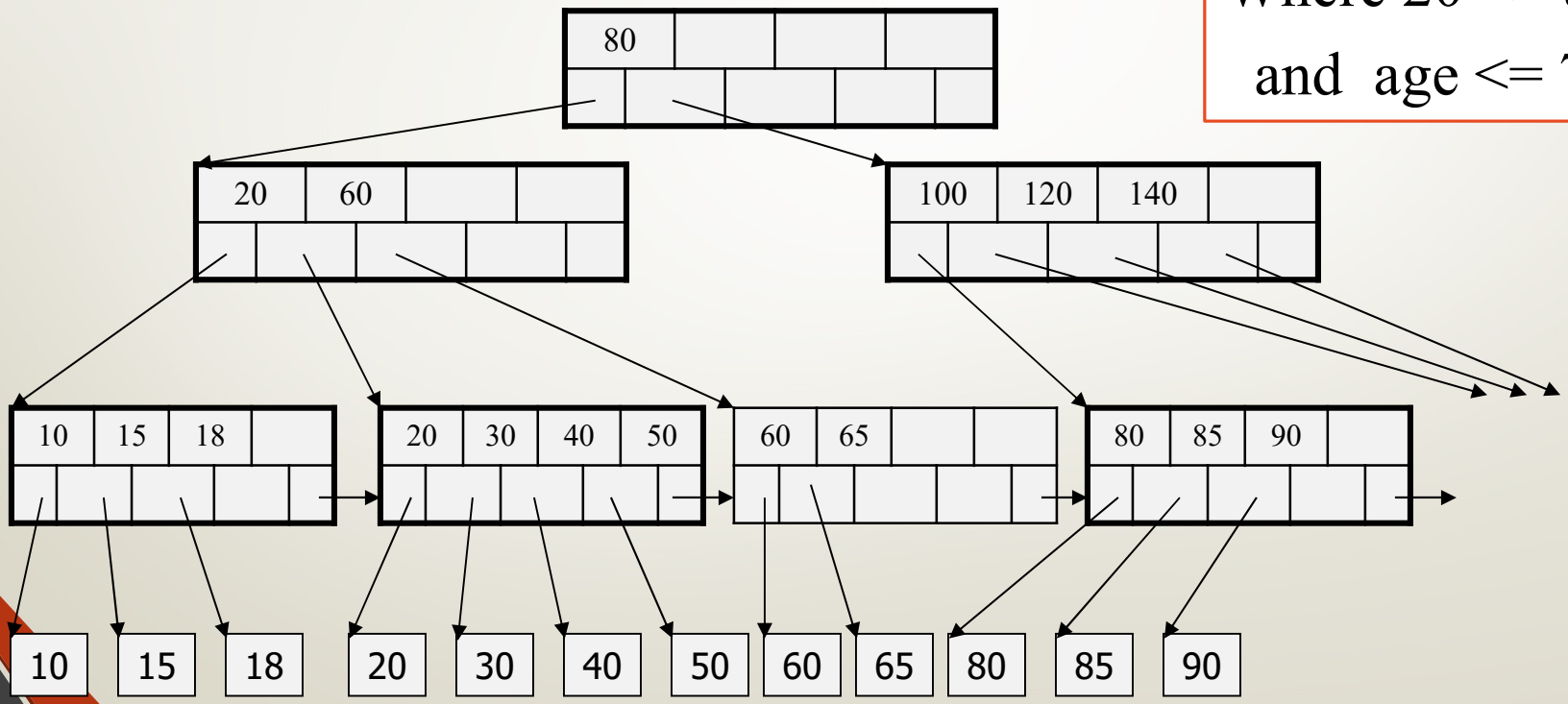


Searching a B+ Tree

Range queries:

- As above
- Then sequential traversal using "next leaf" pointers

Select name
From people
Where 20 <= age
and age <= 70



Outline

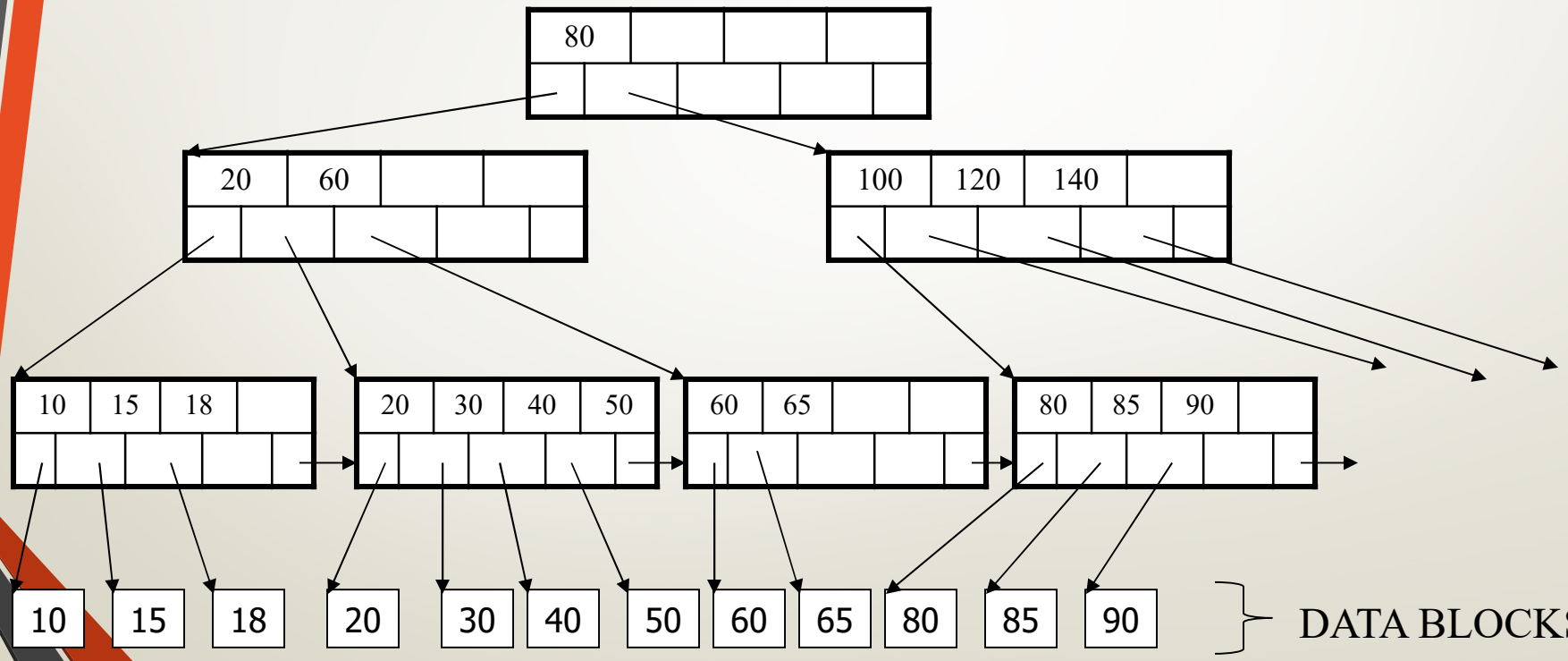
- ✓ Storage
- ✓ Indexing
  - ✓ What is an index? Why do we need it?
- ✓ B+ Trees
  - ✓ Basics and Searching
  - Inserting
  - Deletion
- Hash Tables
  - Secondary storage HT
  - Extensible HT
  - Linear HT

Handling data changes in B+ Trees

Insertion in a B+ Tree

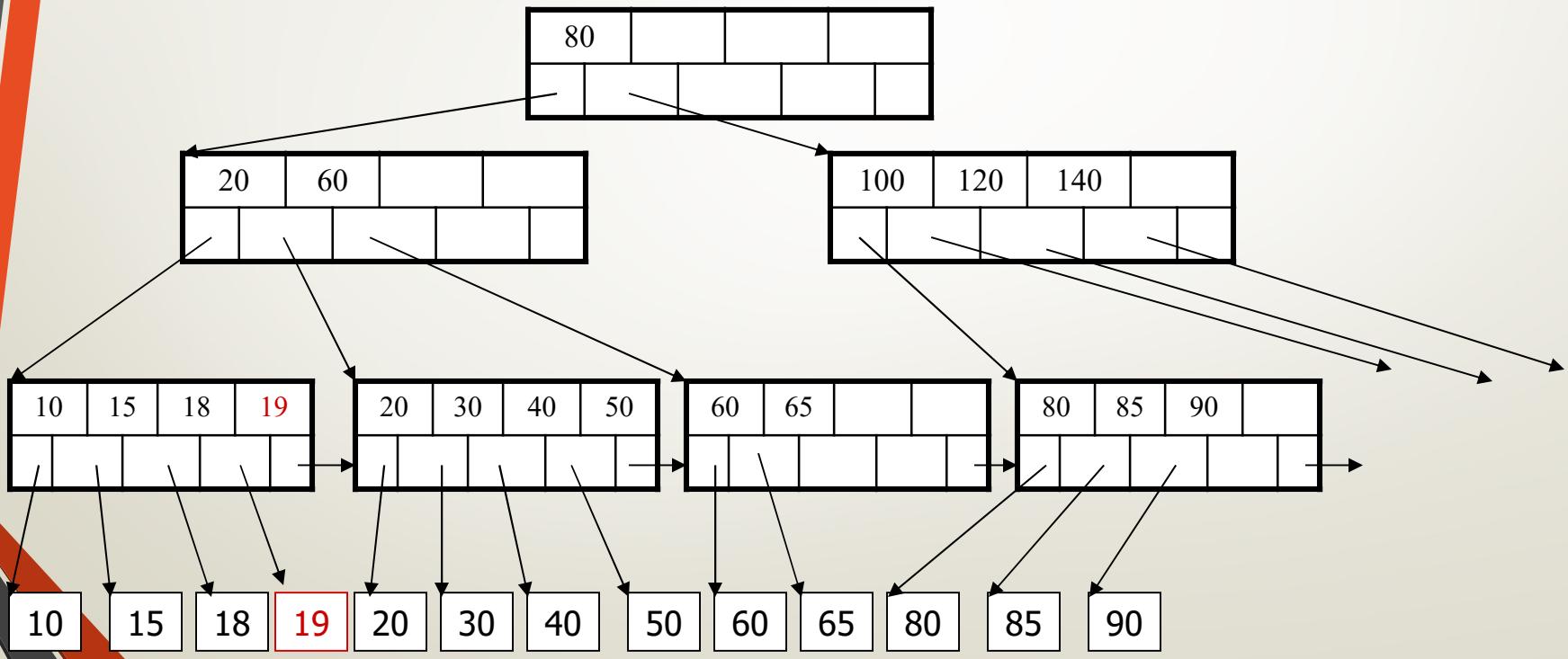
Assume d=2.

Insert K=19



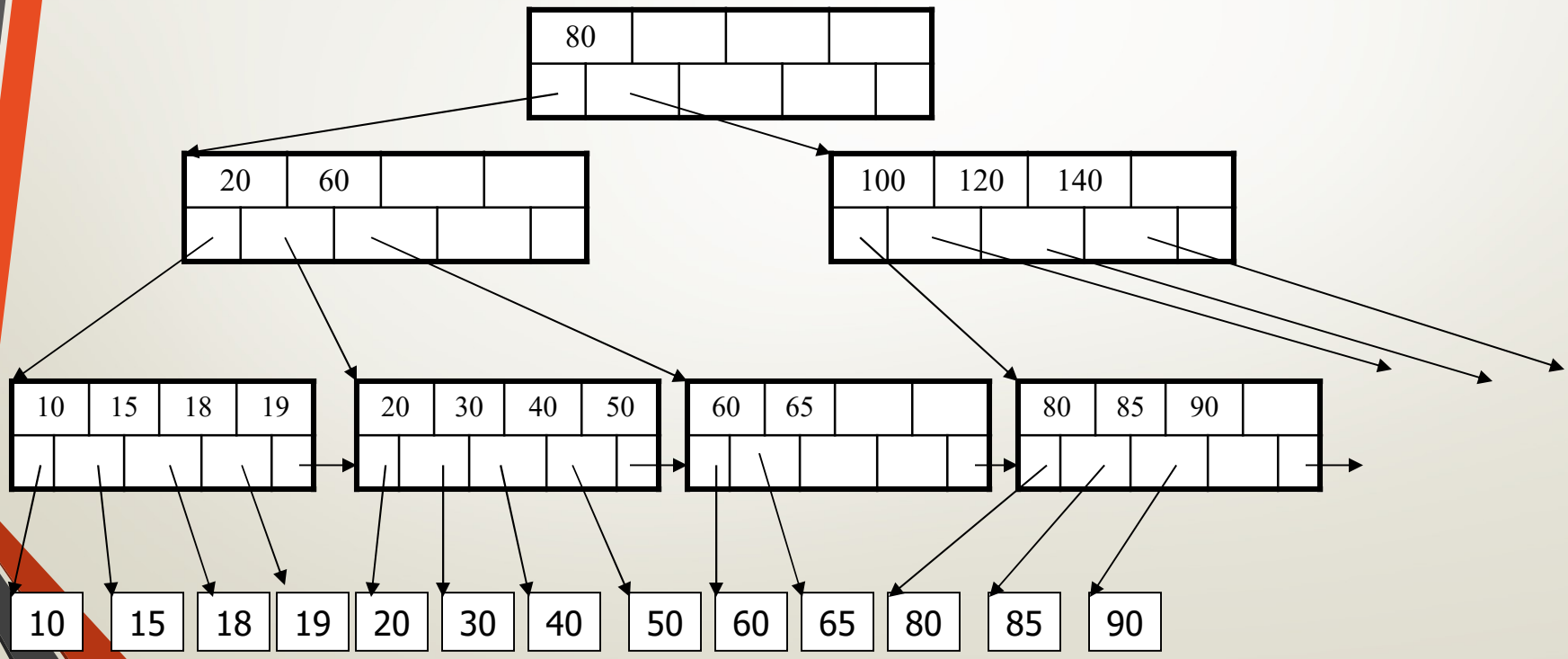
Insertion in a B+ Tree

After insertion



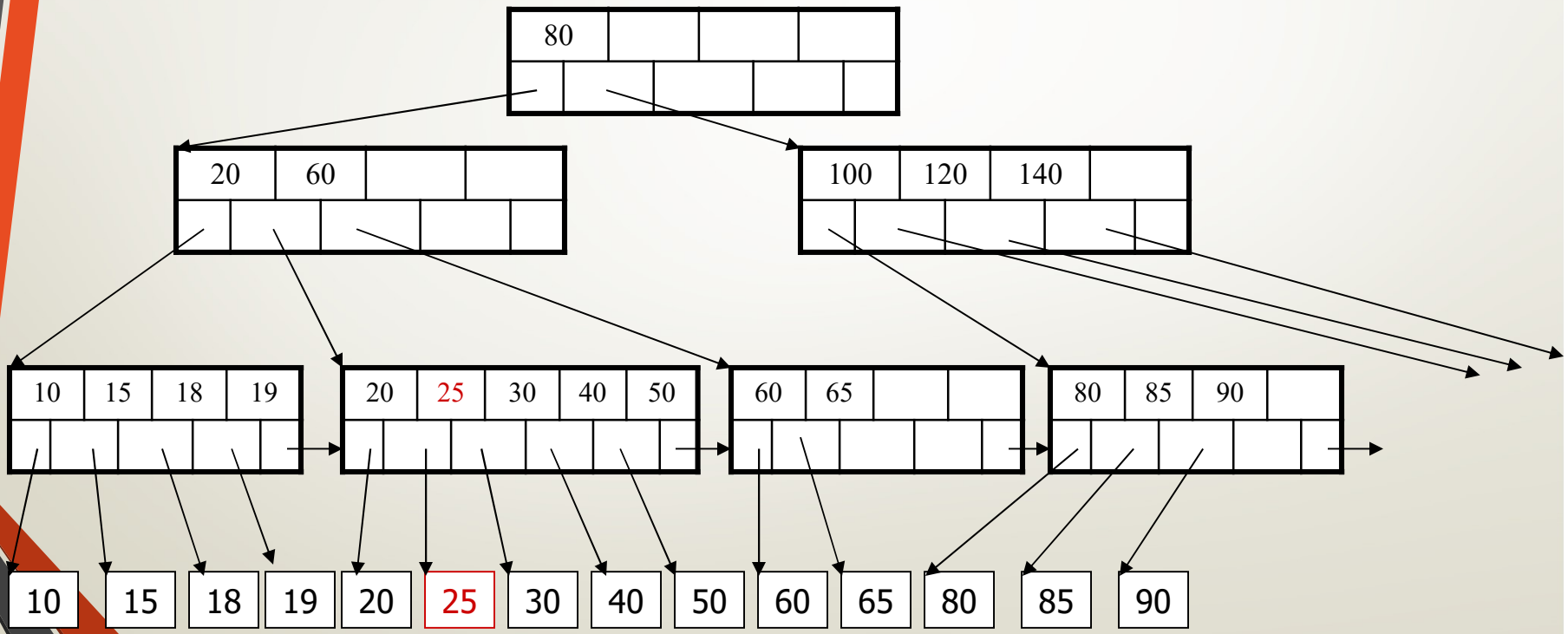
Insertion in a B+ Tree

Now insert 25



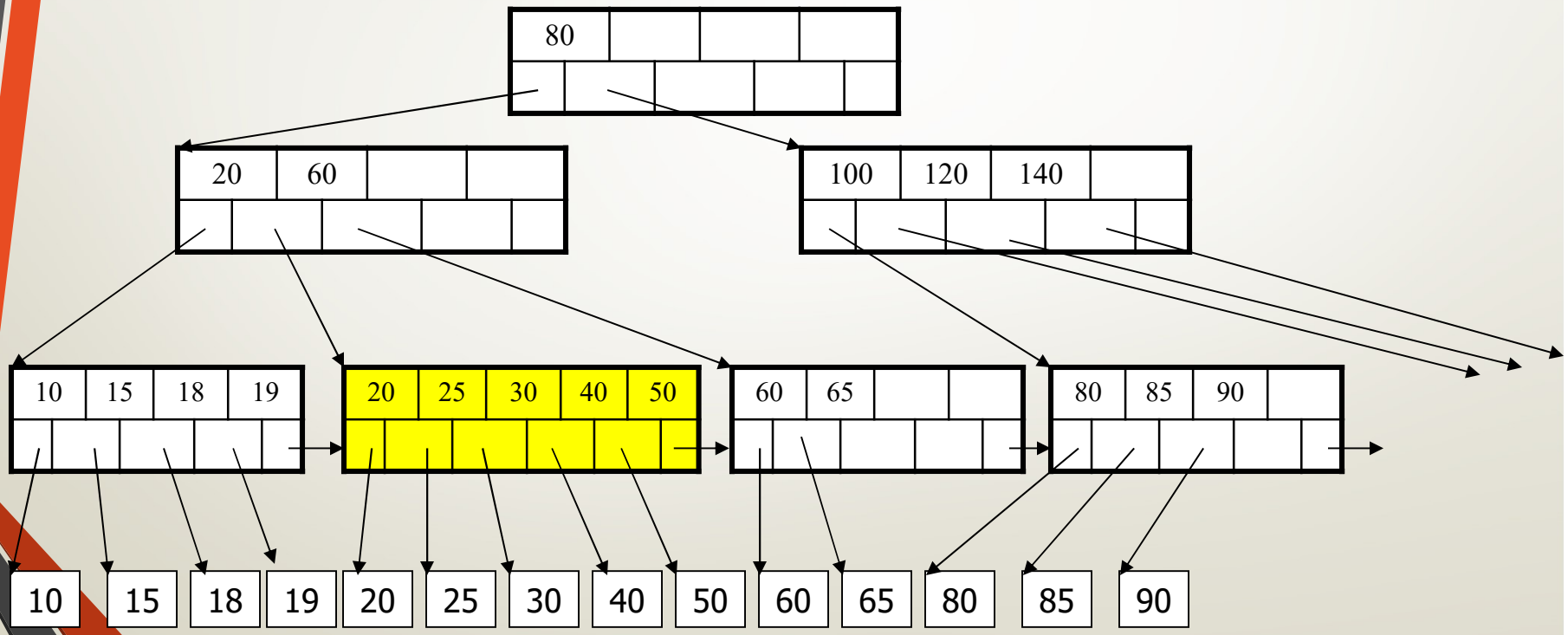
Insertion in a B+ Tree

After insertion



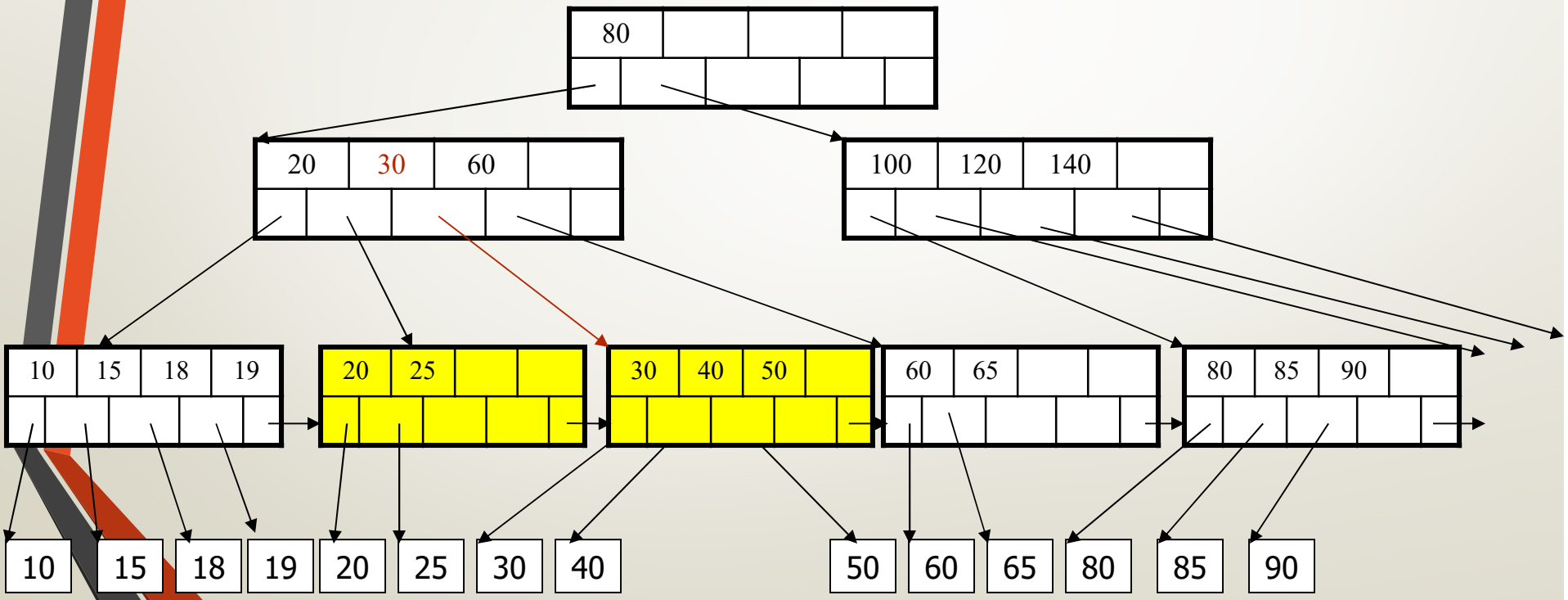
Insertion in a B+ Tree

But now have to split !



Insertion in a B+ Tree

After the split

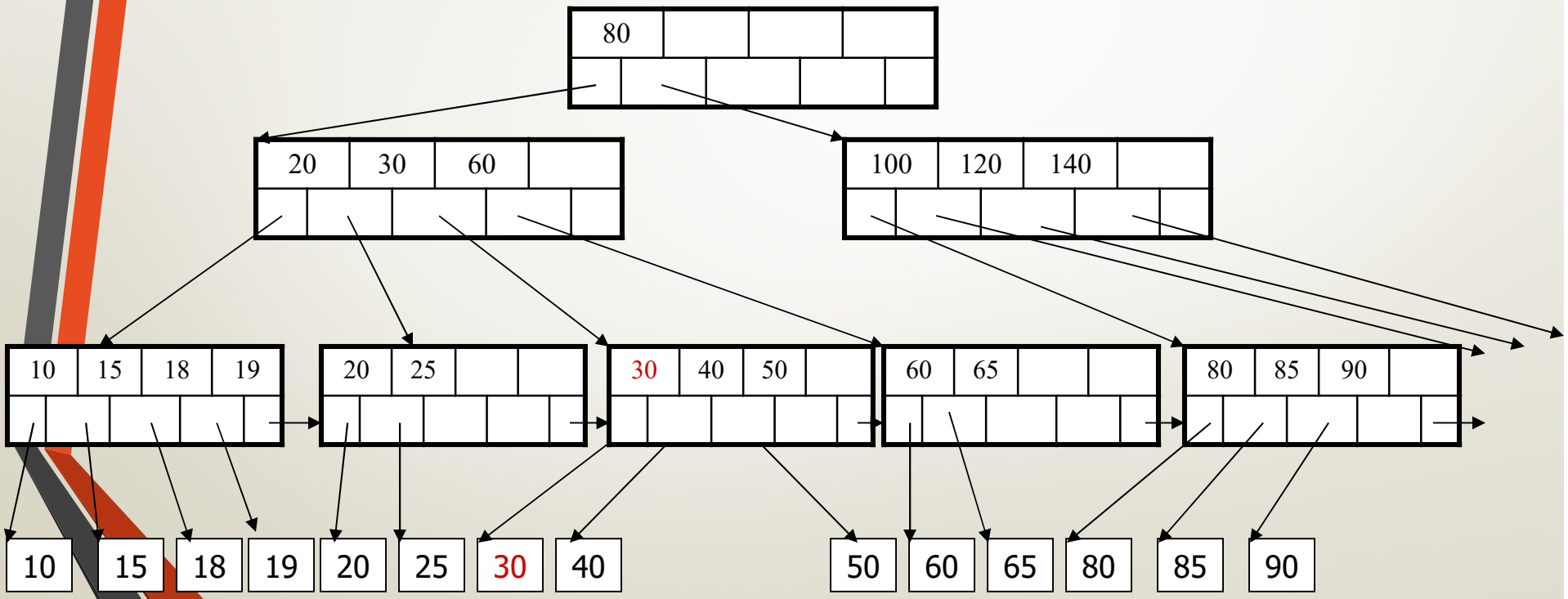


Outline

- ✓ Storage
- ✓ Indexing
  - ✓ What is an index? Why do we need it?
- ✓ B+ Trees
  - ✓ Basics and Searching
  - ✓ Inserting
  - • Deletion
- • Hash Tables
  - • Secondary storage HT
  - • Extensible HT
  - • Linear HT

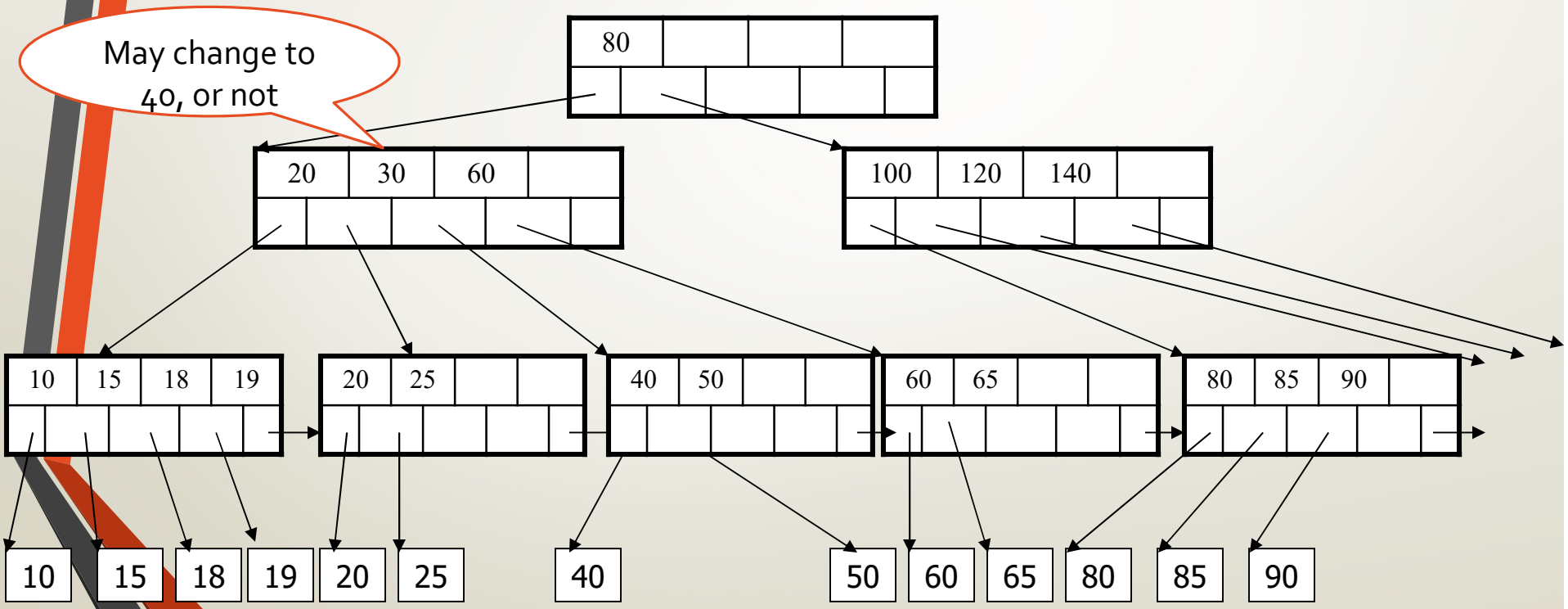
Deletion from a B+ Tree

Delete 30



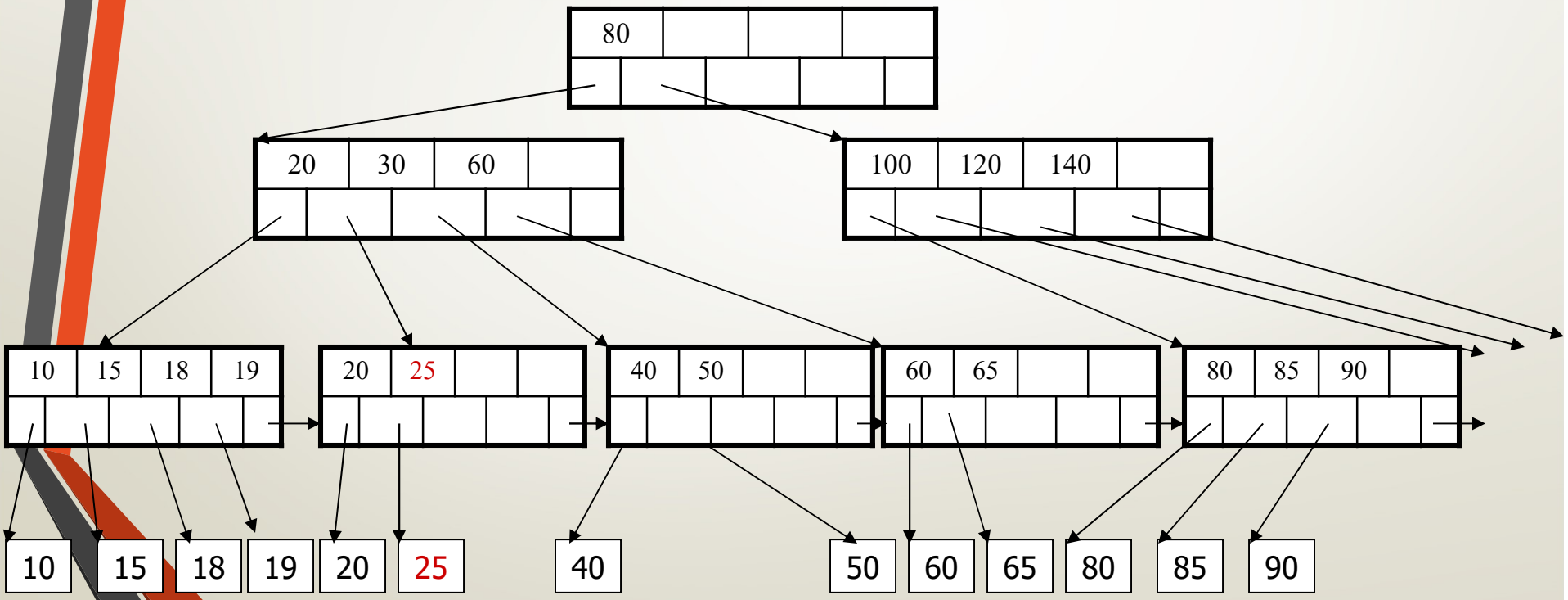
Deletion from a B+ Tree

After deleting 30



Deletion from a B+ Tree

Now delete 25



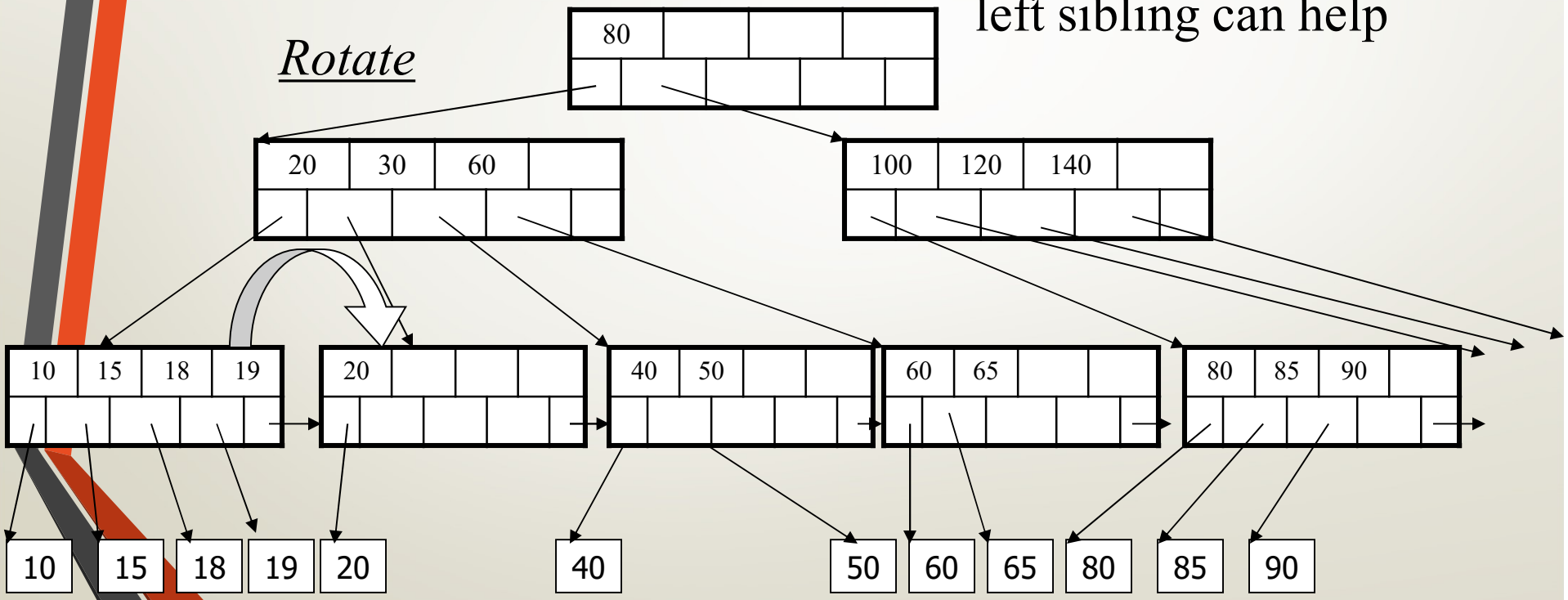
Deletion from a B+ Tree

After deleting 25

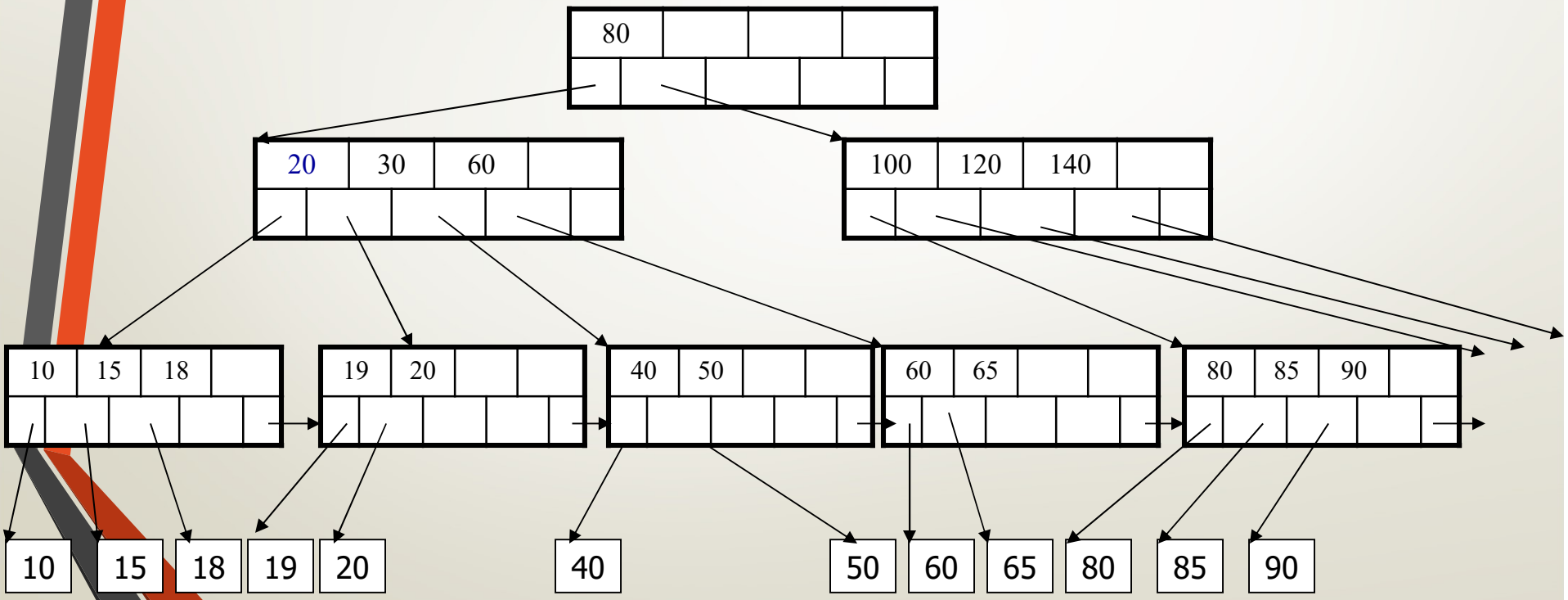
Need to rebalance

Rotate

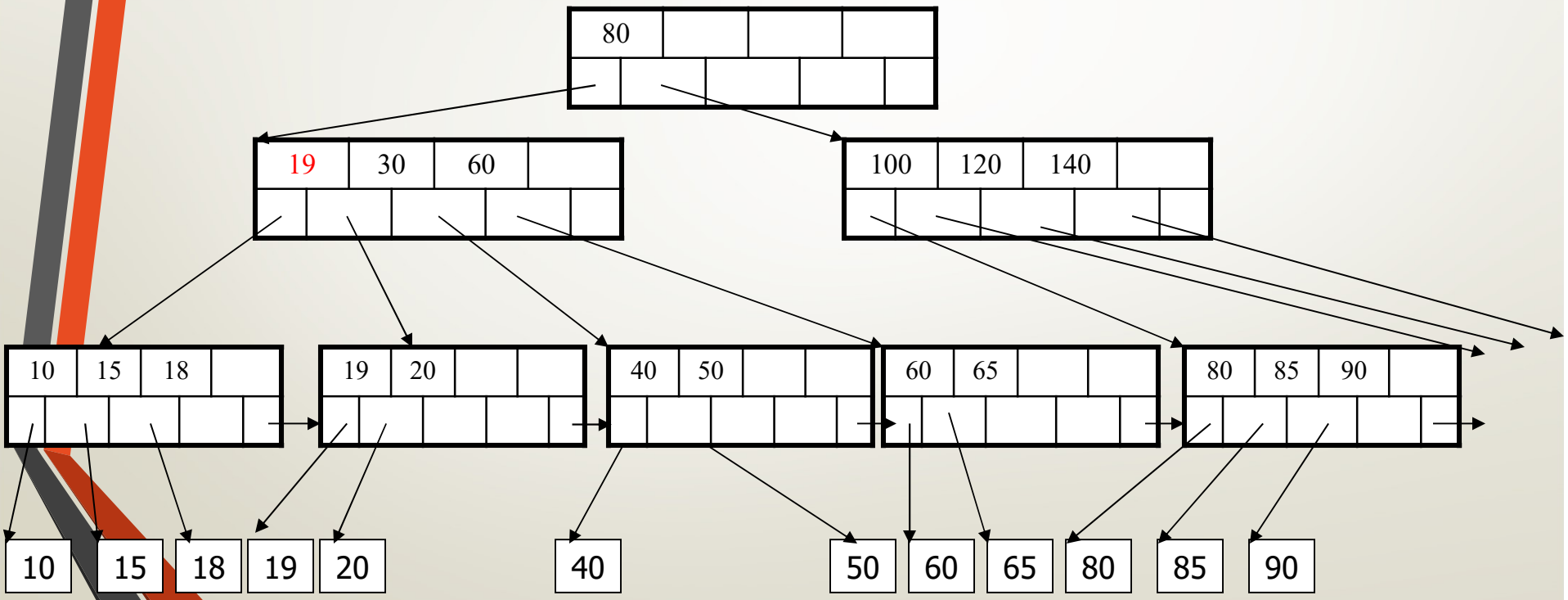
Rotation in general can involve either sibling, but here only the left sibling can help



Deletion from a B+ Tree

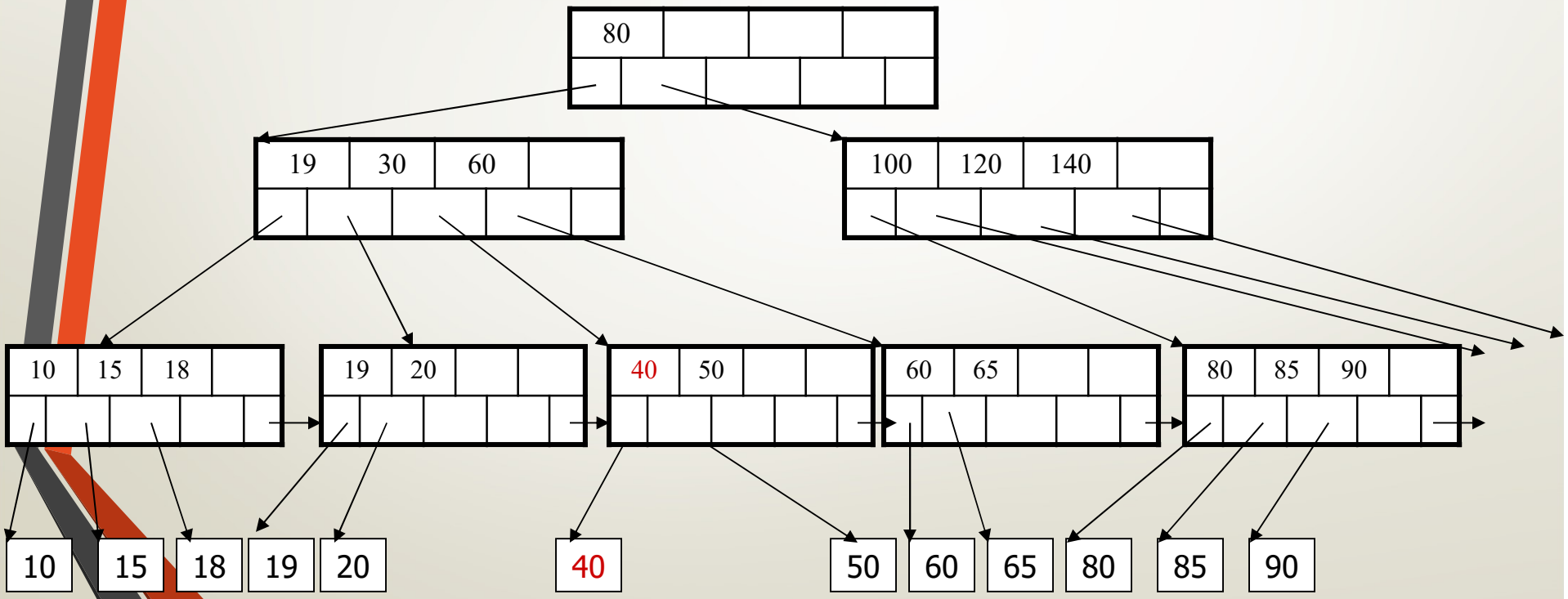


Deletion from a B+ Tree



Deletion from a B+ Tree

Now delete 40

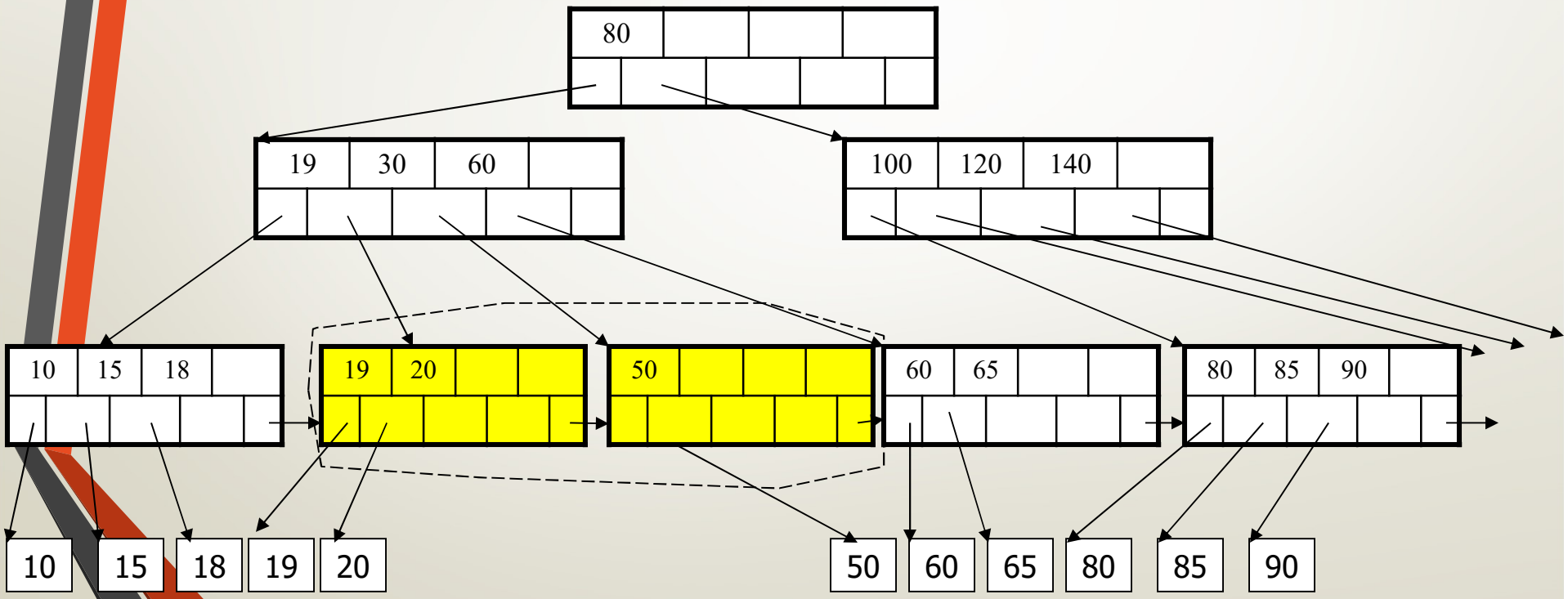


Deletion from a B+ Tree

After deleting 40

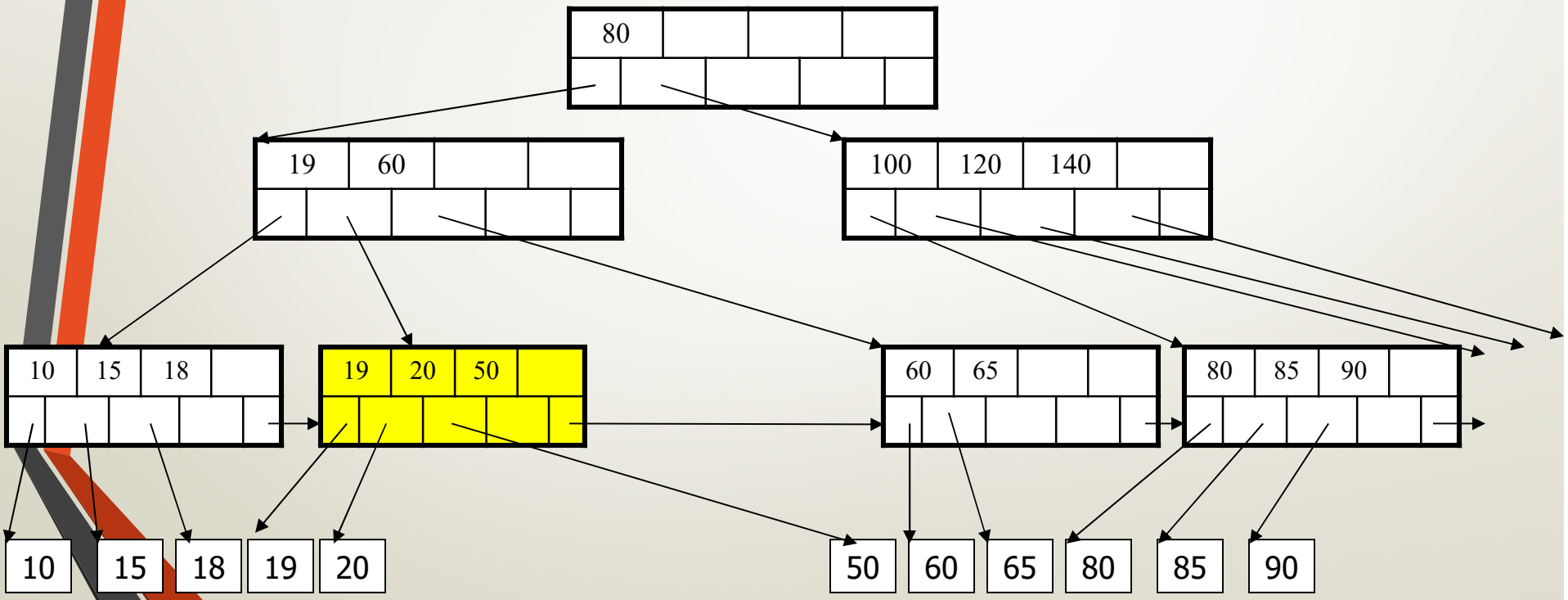
Rotation not possible

Need to *merge* nodes



Deletion from a B+ Tree

Final tree



Advantages of B+Trees

- Balanced → Uniform space utilization
  - Predictable organization
  - Predictable time (logarithmic); unbalanced can be linear in worst case
- Good for range queries

Can we do better?

Outline

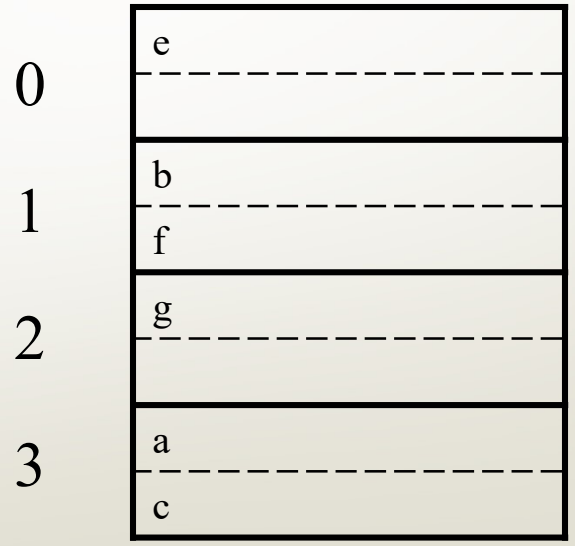
- ✓ Storage
- ✓ Indexing
  - ✓ What is an index? Why do we need it?
- ✓ B+ Trees
  - ✓ Basics and Searching
  - ✓ Inserting
  - ✓ Deletion
- Hash Tables
  - Secondary storage HT
  - Extensible HT
  - Linear HT

Hash Tables

- Secondary storage hash tables are much like main memory ones
- Recall basics:
  - There are B *buckets*
  - A hash function h(k) maps a key k to {0, 1, …, B-1}
  - Store in bucket h(k) a pointer to record with key k
- Secondary storage: bucket = block
  - Store in the block of bucket h(k) any record with key k
  - use overflow blocks when needed

Hash Table Example

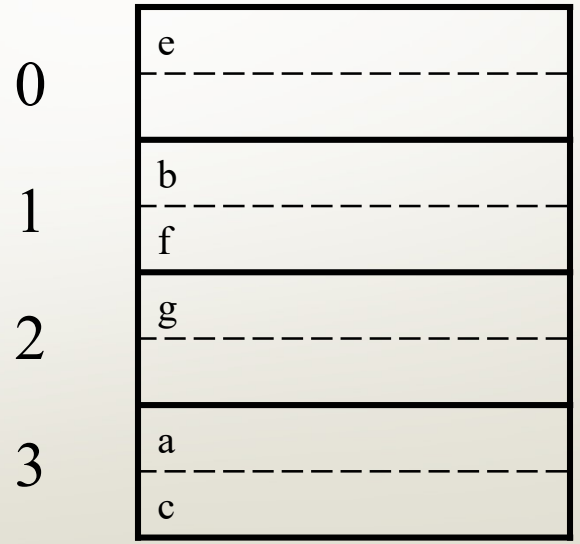
- Assume 1 bucket (block) stores 2 records
- h(e)=0
- h(b)=h(f)=1
- h(g)=2
- h(a)=h(c)=3



Searching in a Hash Table

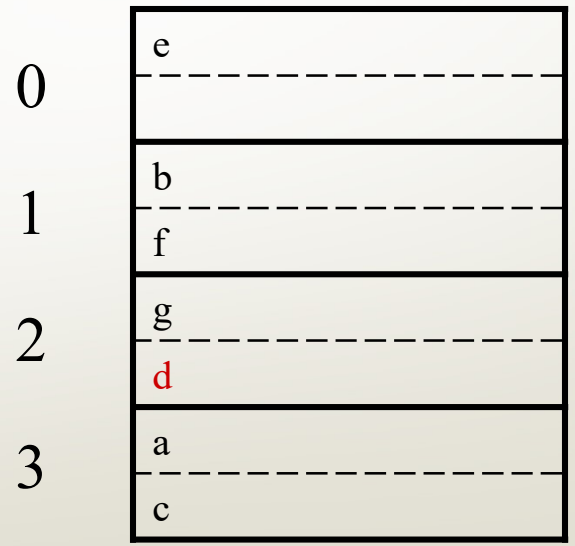
- Search for a:
- Compute h(a)=3
- Read bucket (block) 3
- 1 disk access

Main memory may have an array of pointers (to buckets) accessible by bucket number.



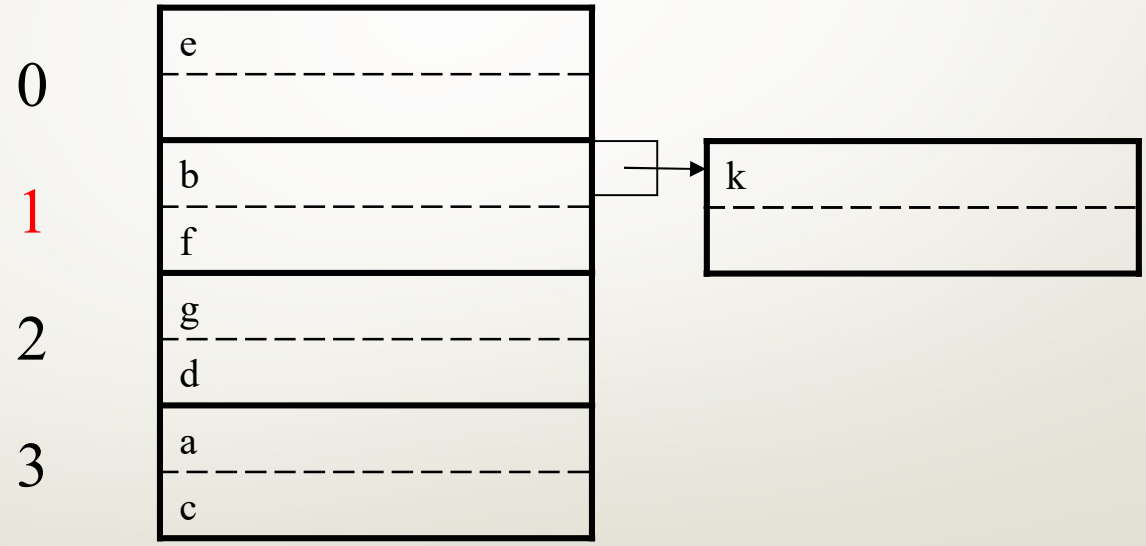
Insertion in Hash Table

- Place in right bucket (block), if space
- E.g. h(d)=2



Insertion in Hash Table

- Create overflow block, if no space
- E.g. h(k)=1



More over-flow blocks may be needed

Hash Table Performance

- Fixed number of buckets
- Excellent, if no overflow blocks
- Degrades considerably when there are many overflow blocks.
  - Might need to go through a chain of overflow blocks

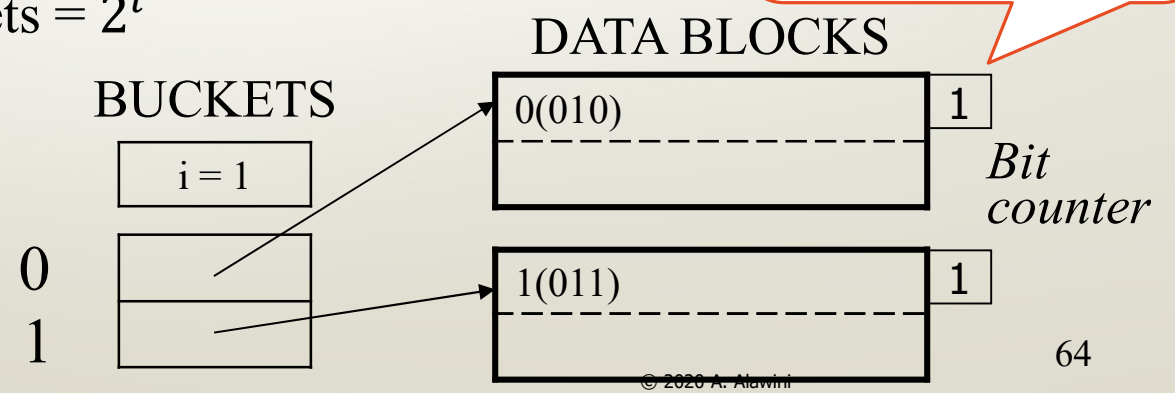
Can improve this by allowing the number of buckets to grow

Outline

- ✓ Storage
- ✓ Indexing
  - ✓ What is an index? Why do we need it?
- ✓ B+ Trees
  - ✓ Basics and Searching
  - ✓ Inserting
  - ✓ Deletion
- Hash Tables
  - ✓ Secondary storage HT
  - Extensible HT
  - Linear HT

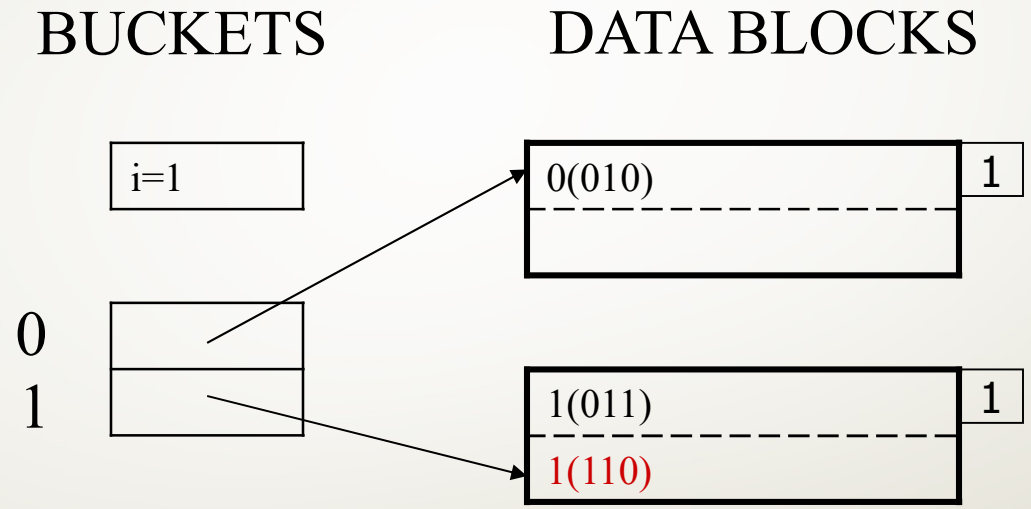
Extensible Hash Table

- Array of pointers to blocks instead of array of blocks
- Size of array is allowed to grow. 2x size when it grows
- Don't need a block per bucket. Sparse buckets share a block
- Hash function returns k-bit integers (e.g., k=32)
  - Only use the first i << k bits to determine bucket
  - Number of buckets = 2^i



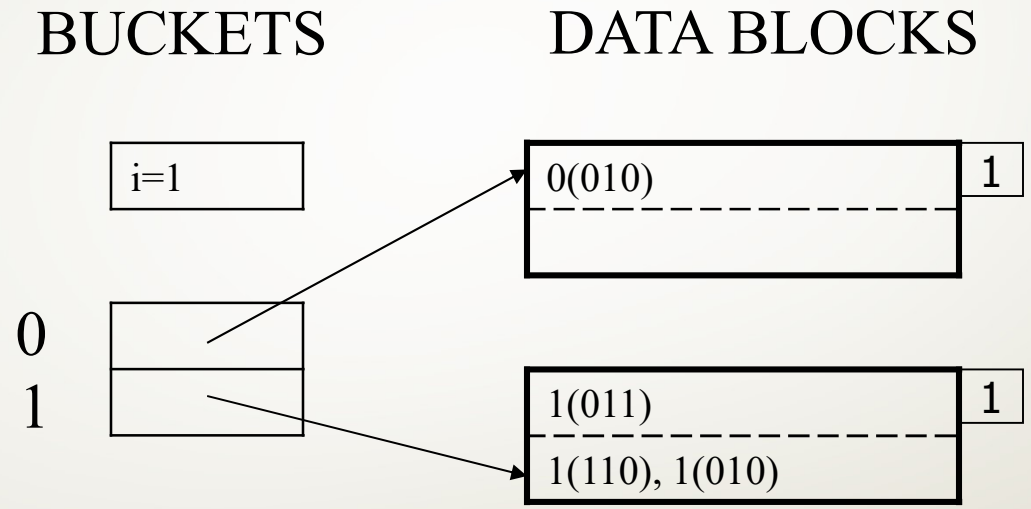
Insertion in Extensible Hash Table

- Insert 1110



Insertion in Extensible Hash Table

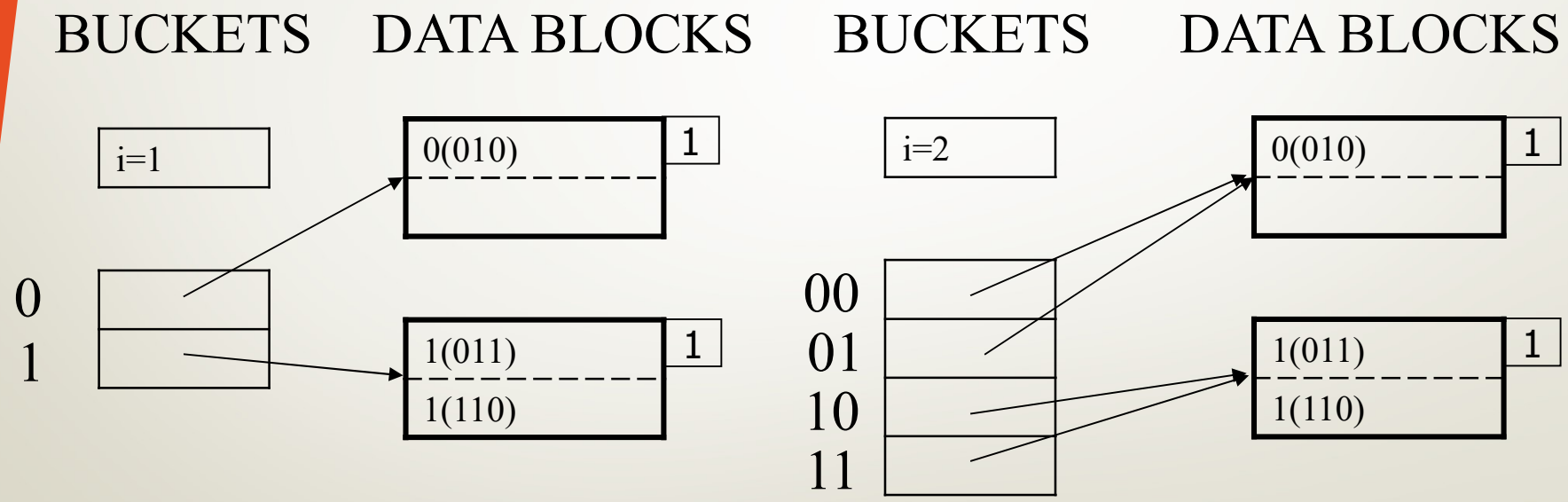
- Now insert 1010



- Need to split block and extend bucket array
- i becomes 2: done in two steps

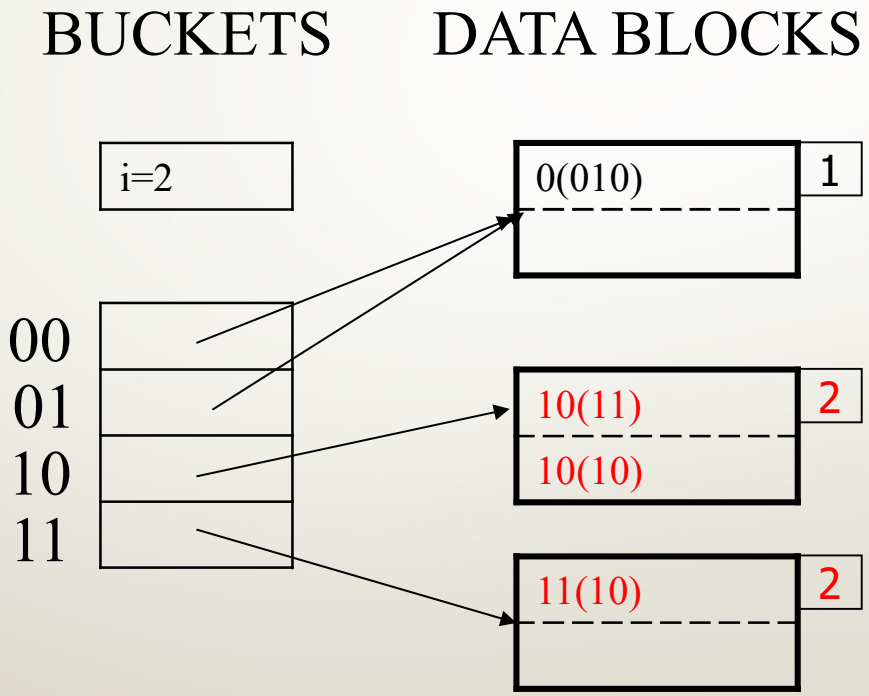
Insertion in Extensible Hash Table

Step 1: Extend the buckets



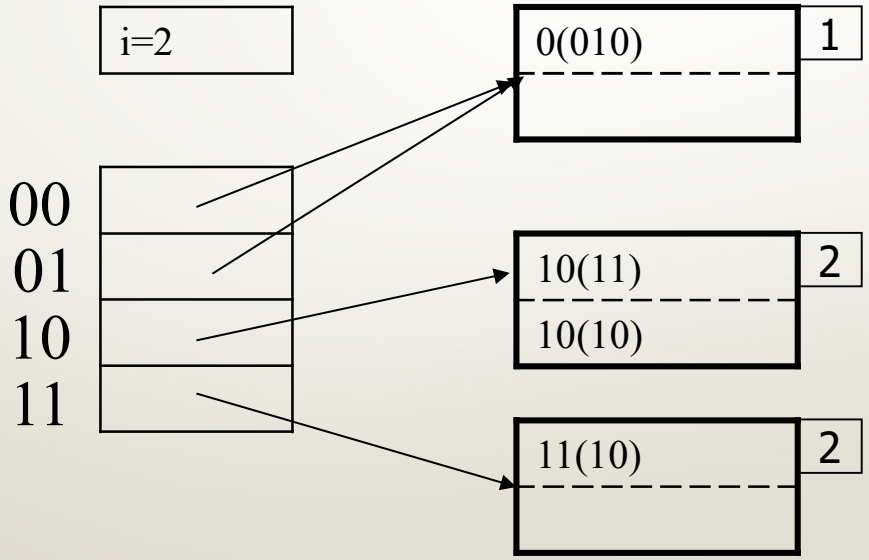
Insertion in Extensible Hash Table

Step 2: Now try to insert 1010



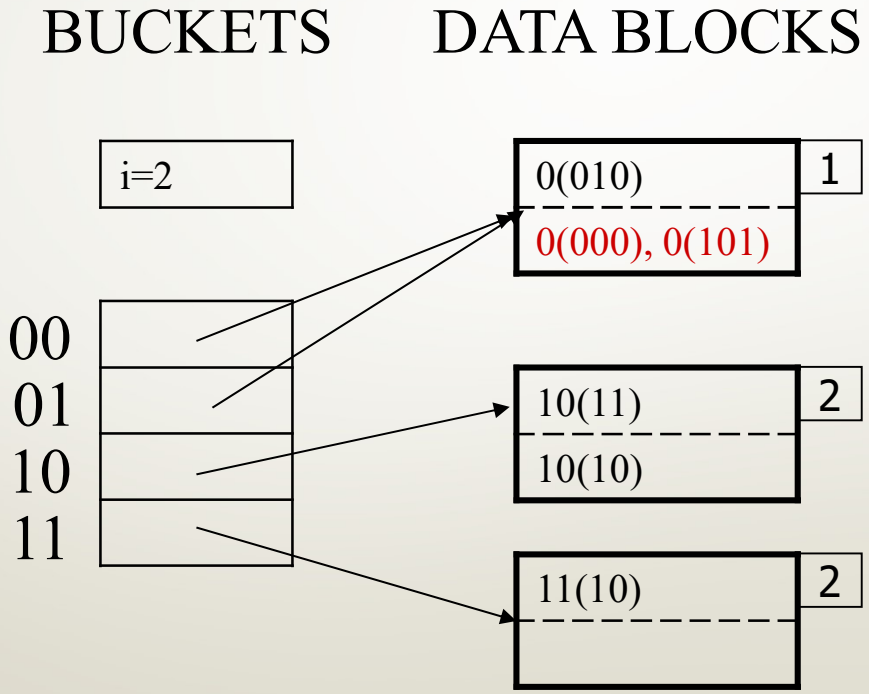
Insertion in Extensible Hash Table

- Now insert 0000: where would it go? Then 0101?
- Need to split block, but not bucket array



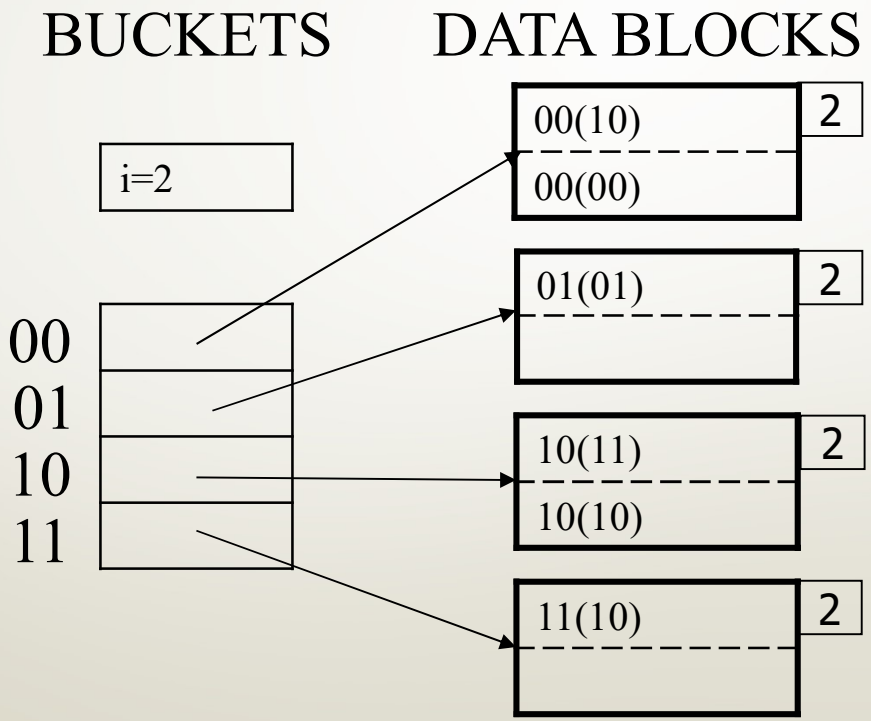
Insertion in Extensible Hash Table

- Now insert 0000: where would it go? Then 0101?
- Need to split block, but not bucket array



Insertion in Extensible Hash Table

- Now insert 0000: where would it go? Then 0101?
- Need to split block, but not bucket array



Performance: Extensible Hash Table

- No overflow blocks: access always one read for distinct keys
- BUT:
  - Extensions can be costly and disruptive
  - After an extension bucket table may no longer fit in memory
  - Imagine three records whose keys share the first 20 bits. These three records cannot be in same block (assume two records per block). But a block split would require setting i = 20, i.e., accommodating for 2^20 = 1 million buckets, even though there may be only a few hundred records.

Outline

- ✓ Storage
- ✓ Indexing
  - ✓ What is an index? Why do we need it?
- ✓ B+ Trees
  - ✓ Basics and Searching
  - ✓ Inserting
  - ✓ Deletion
- Hash Tables
  - ✓ Secondary storage HT
  - ✓ Extensible HT
  - Linear HT

Linear Hash Table

- Idea 1: add only one bucket at a time

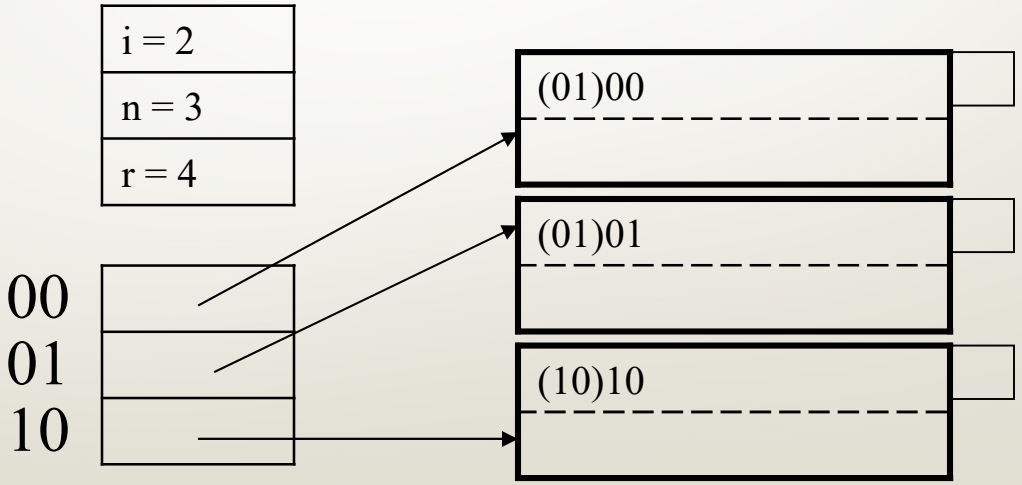
  Problem: n = no longer a power of 2
- Let i be # bits necessary to address n buckets.
  - i = ceil(log_2 n)
- After computing h(k), use ***last*** i bits:
  - If last i bits represent a number (say m) < n, store the key in bucket m
  - If m >= n, change msb from 1 to 0 (get a number < n)
- Idea 2: allow overflow blocks (not expensive to overflow)
- Convention: Read from the right (as opposed to the left)

Linear Hash Table Example

- N=3 <= $2^2 = 4$
  - Therefore, only buckets until 10

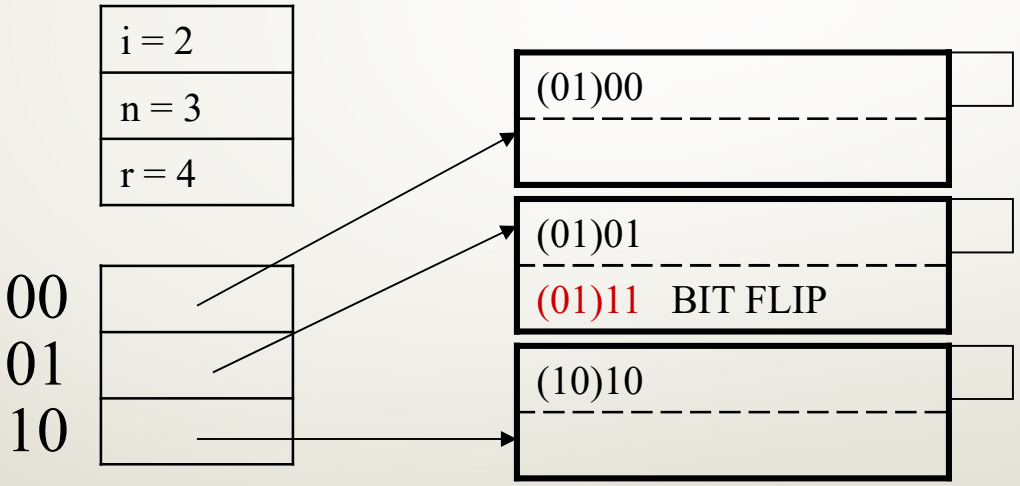
Try to insert 01<u>11</u>

11 is flipped => 01



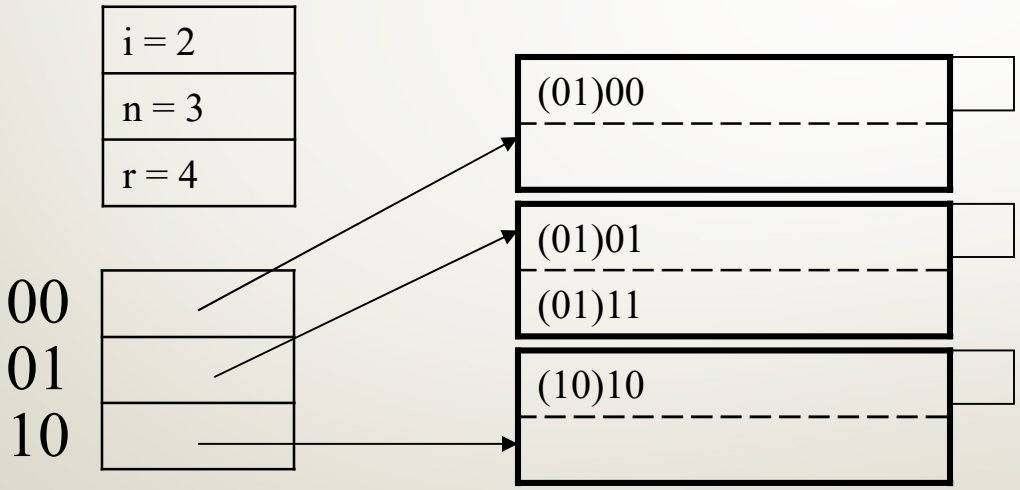
Linear Hash Table Example

- After inserting 0111



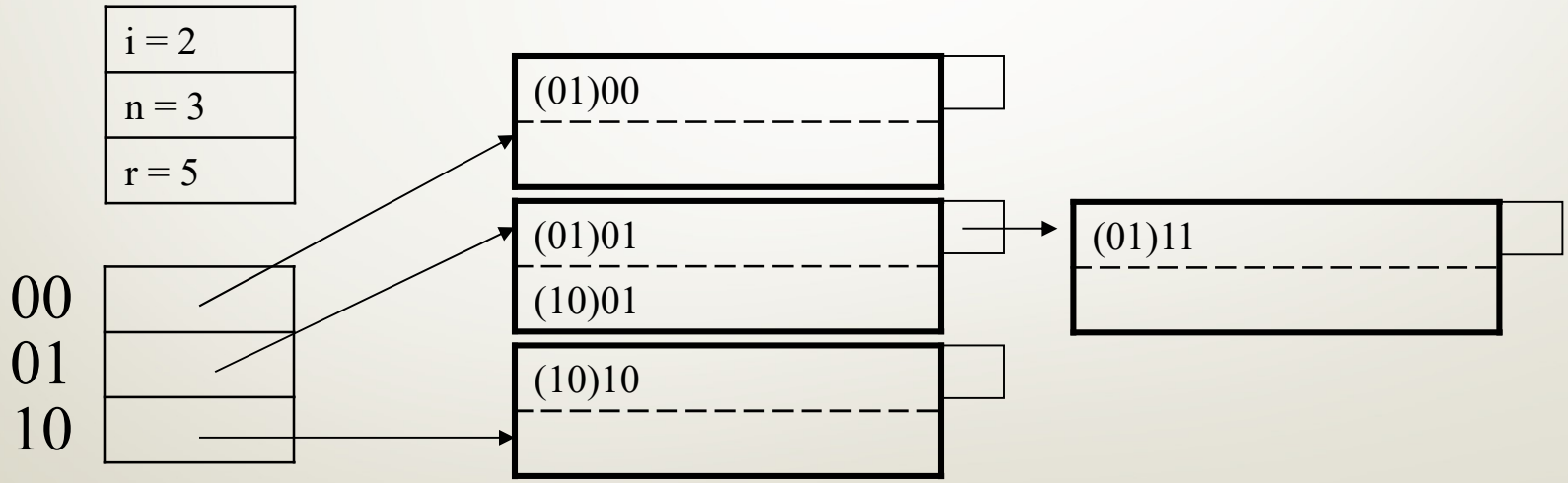
Linear Hash Table Example

- Insert 1001:



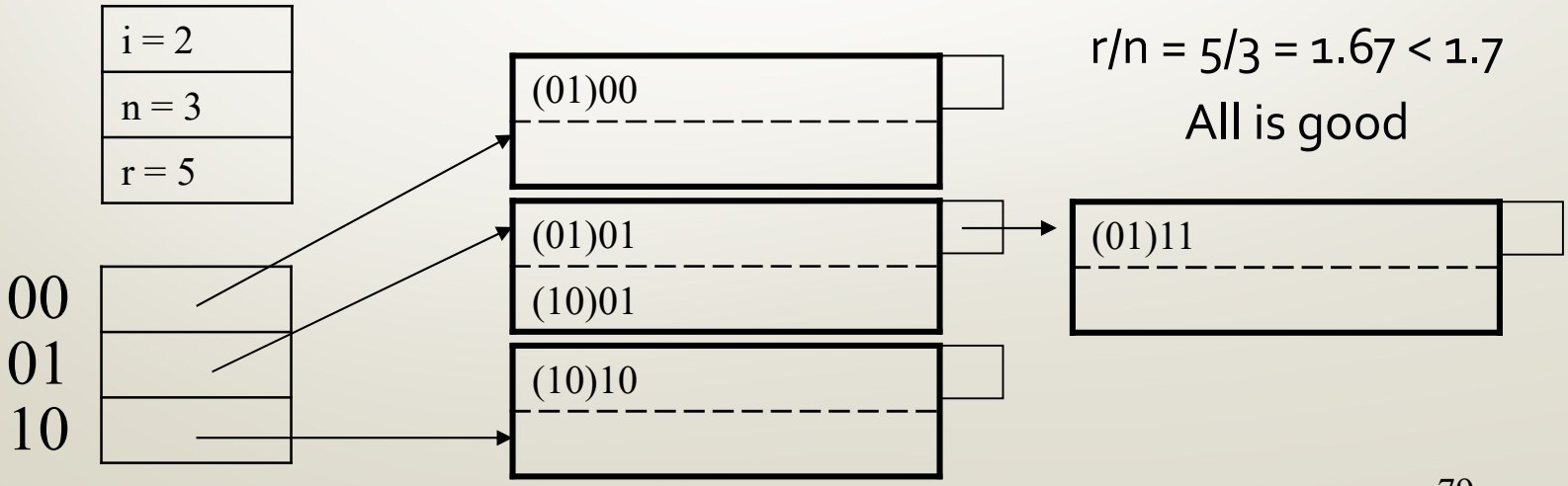
Linear Hash Table Example

- Insert 1001: overflow blocks…



Linear Hash Tables

- Extend n → n+1 when average number of records per bucket exceeds (say) 85% of total number of records per block
  - e.g., r/n <= 0.85 * 2 = 1.7 (for block size = 2)
- Until then, use overflow blocks (cheaper than adding buckets)

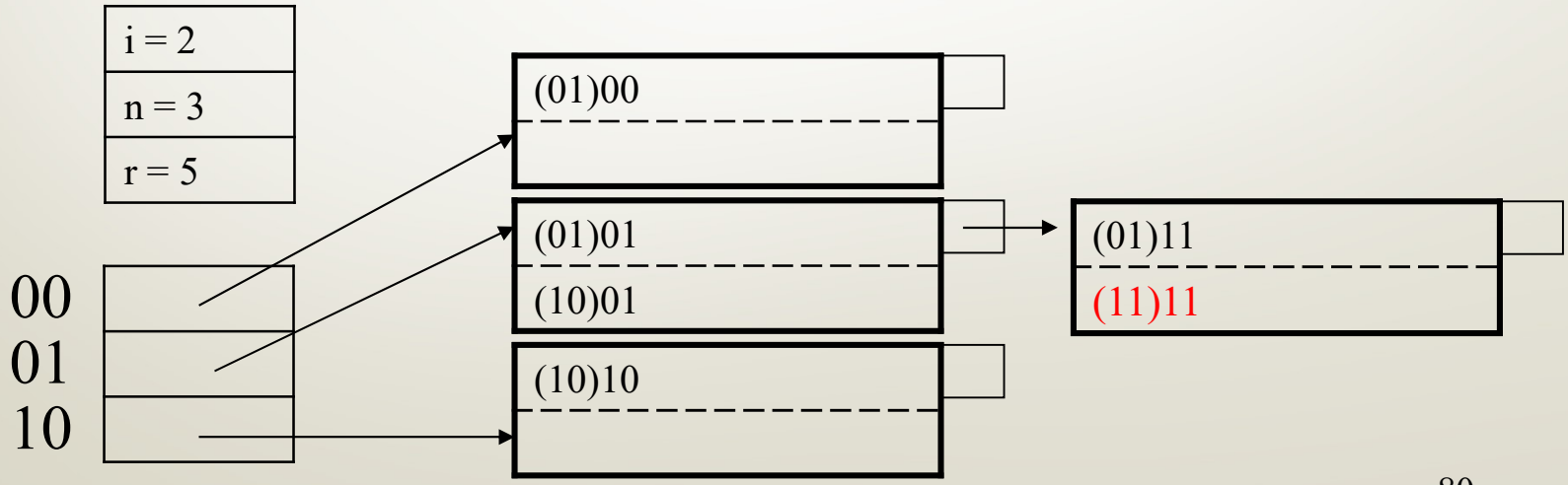


Linear Hash Tables

- Try to insert 1111

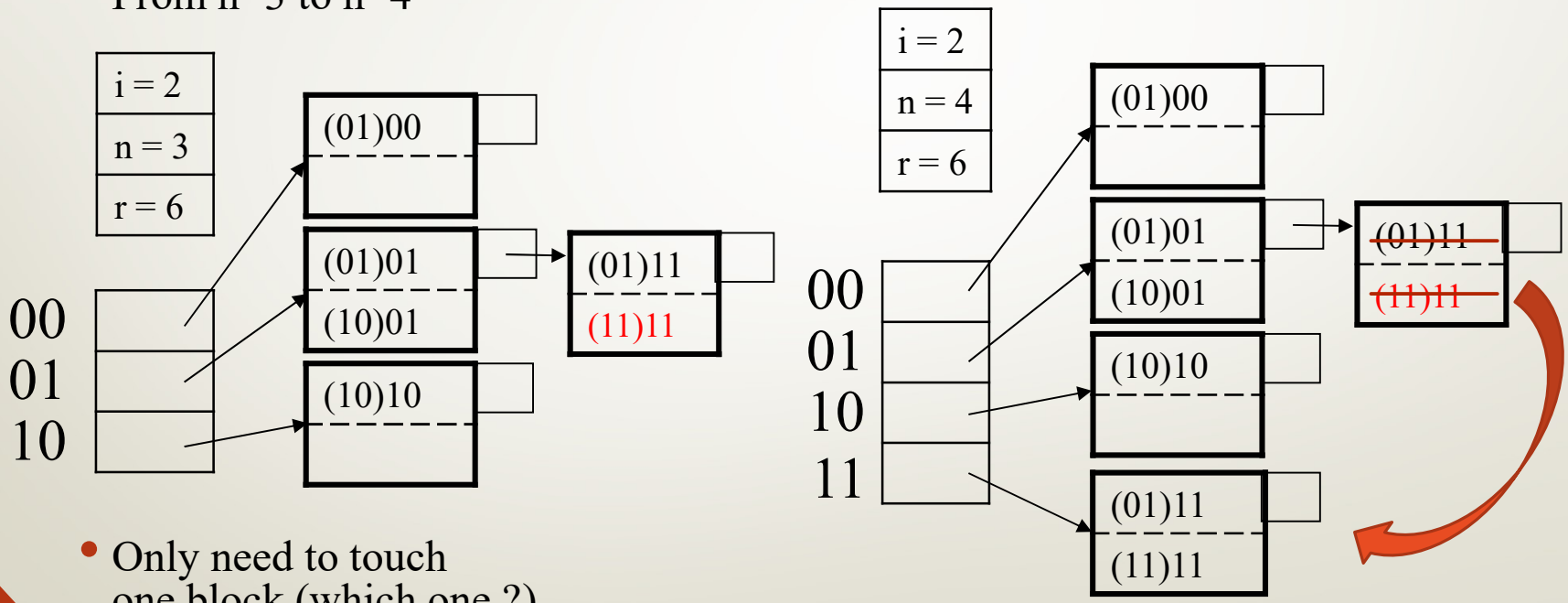
r/n = 6/3 = 2 > 1.7

→ Time to add a bucket



Linear Hash Table Extension

- From n=3 to n=4

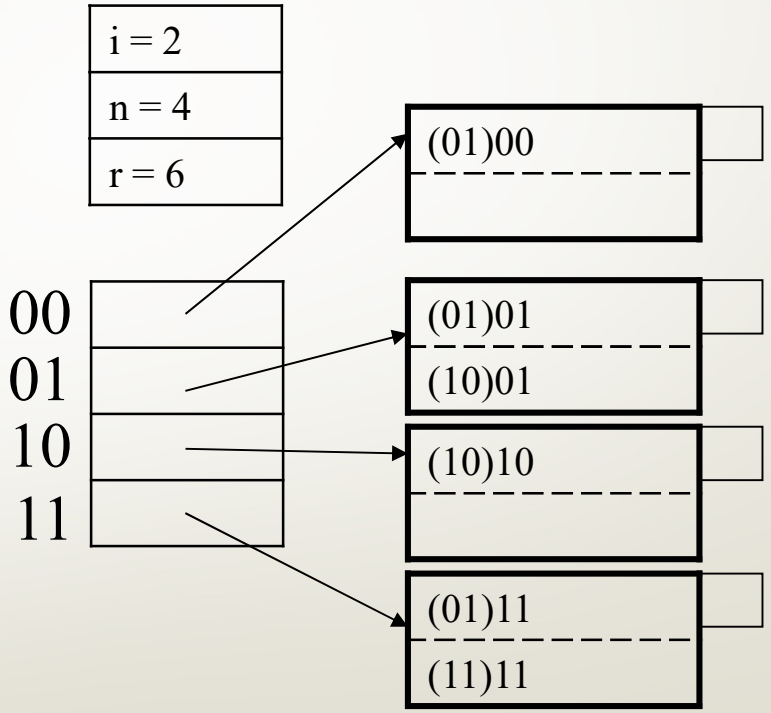


- Only need to touch one block (which one ?)

Linear Hash Table Extension

- From n=3 to n=4 finished

r/n = 6/4 = 1.5 < 1.7 ✔



Outline

- ✓ Storage
- ✓ Indexing
  - ✓ What is an index? Why do we need it?
- ✓ B+ Trees
  - ✓ Basics and Searching
  - ✓ Inserting
  - ✓ Deletion
- Hash Tables
  - ✓ Secondary storage HT
  - ✓ Extensible HT
  - ✓ Linear HT

Summary

- B+ Trees (search, insertion, deletion)
  - Good for point and range queries
  - Log time lookup, insertion and deletion because of balanced tree
- Hash Tables (search, insertion)
  - Static hash tables: one I/O lookup, unless long chain of overflow
  - Extensible hash tables: one I/O lookup, extension can take long
  - Linear hash tables: ~ one I/O lookup, cheaper extension
- No panacea; dependent on data and use case