Indexing: B+ Tree & Hash Tables

Doris Xin, **Abdu** Alawini

University of Illinois at Urbana-Champaign

CS411: Database Systems

October 24, 2018

Announcements

- HW 3: Due by Friday 10/26 (23:59)
- Sign up for PT1 midterm demos: Due by Friday 10/26 (23:59)

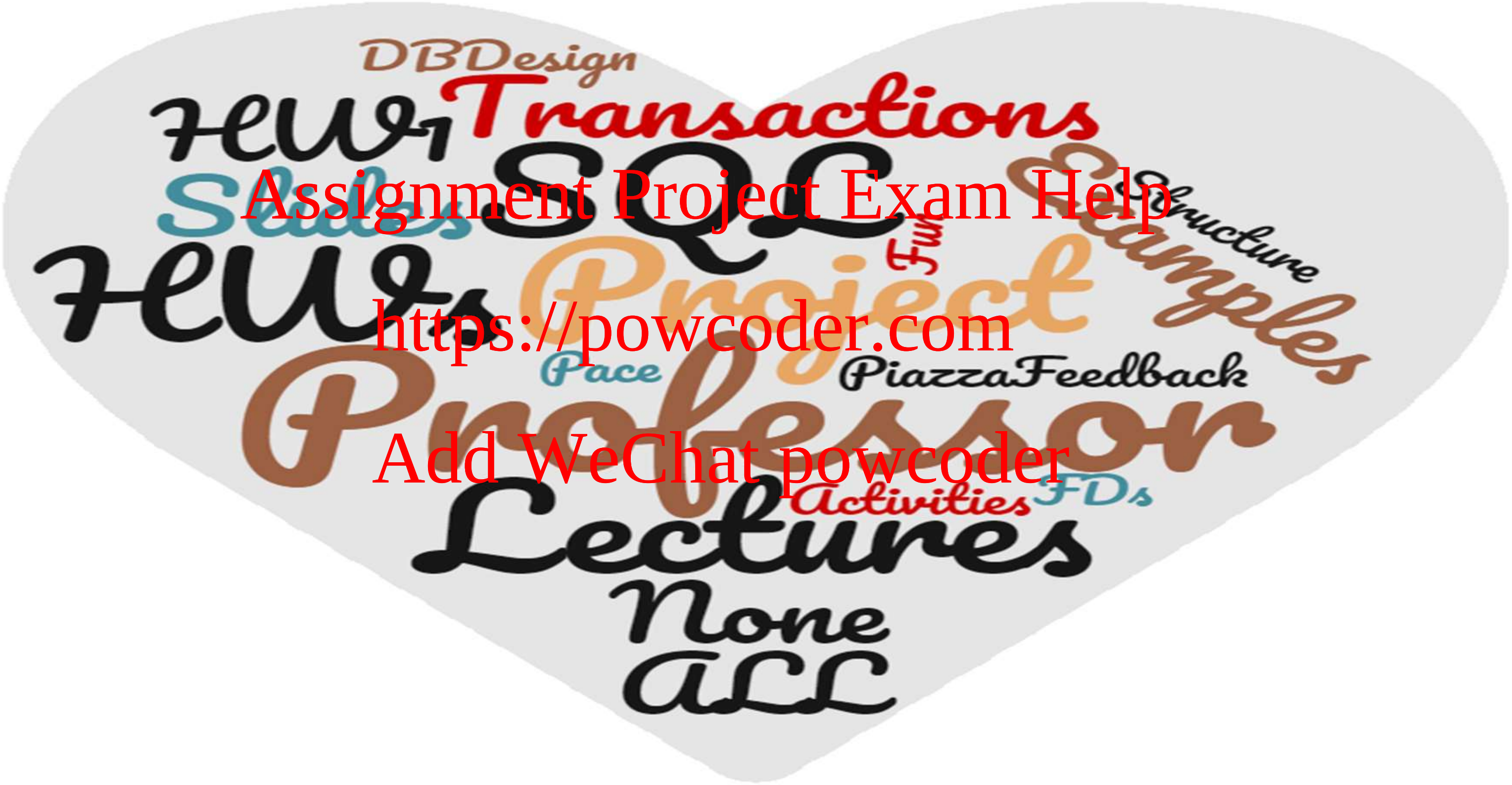
  https://wiki.illinois.edu/wiki/display/cs411sfa18/Project+Track+1+Midterm+Demo+Signup
- Midterm review session: Friday 10/26 (4:00-4:50) SC 1404
  - To suggest topics to discuss in the review session, please fill this form: https://goo.gl/forms/5fDcm80cDjmtMJoH3
- Please fill the early course feedback form:

  https://goo.gl/forms/SC4BYcrDy8dai8PE2
- Midterm: 10/29 in class 11-12:15 pm

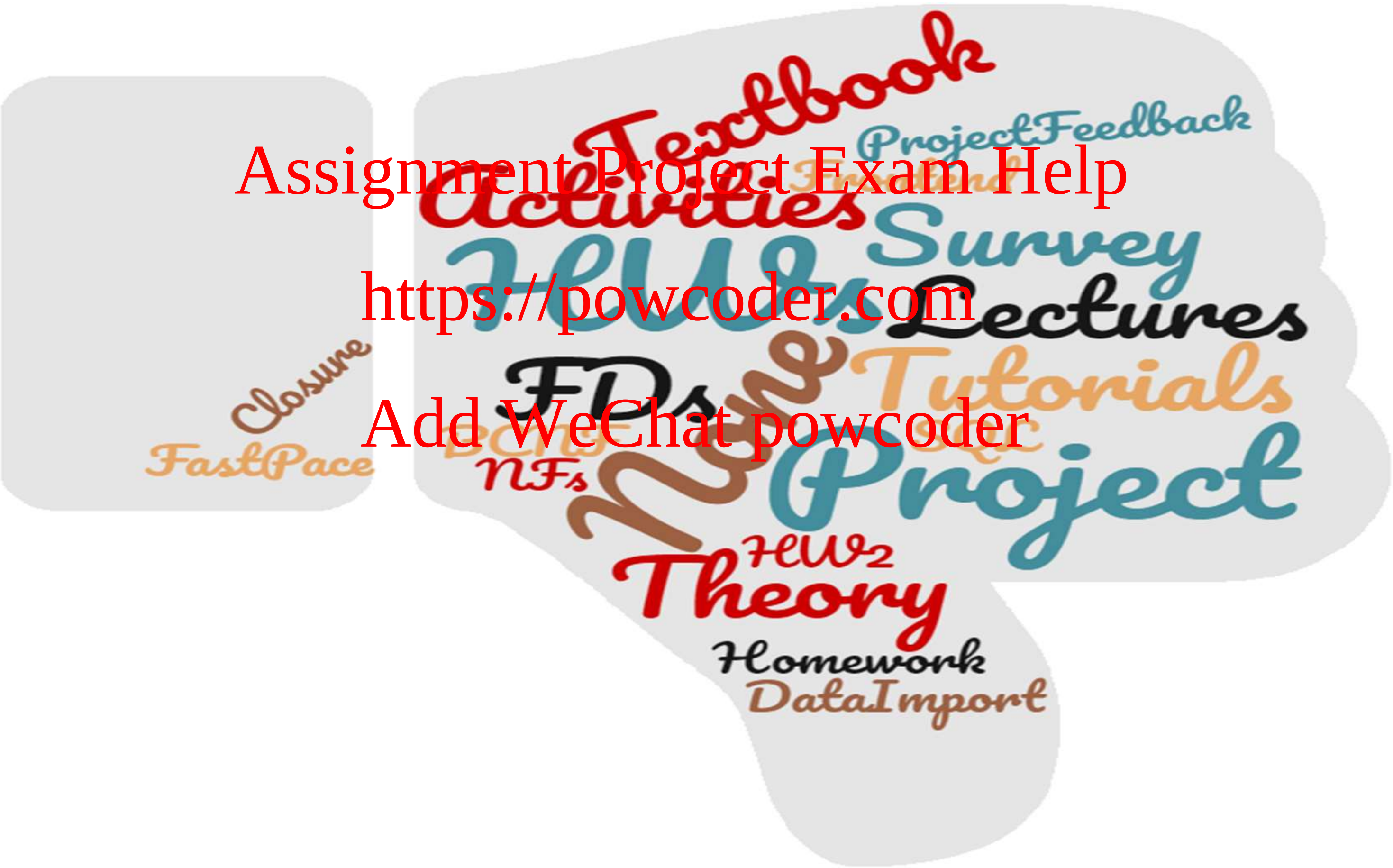
Today's lecture

- Indexing
  - Continue with B+ Trees
  - Hash Tables

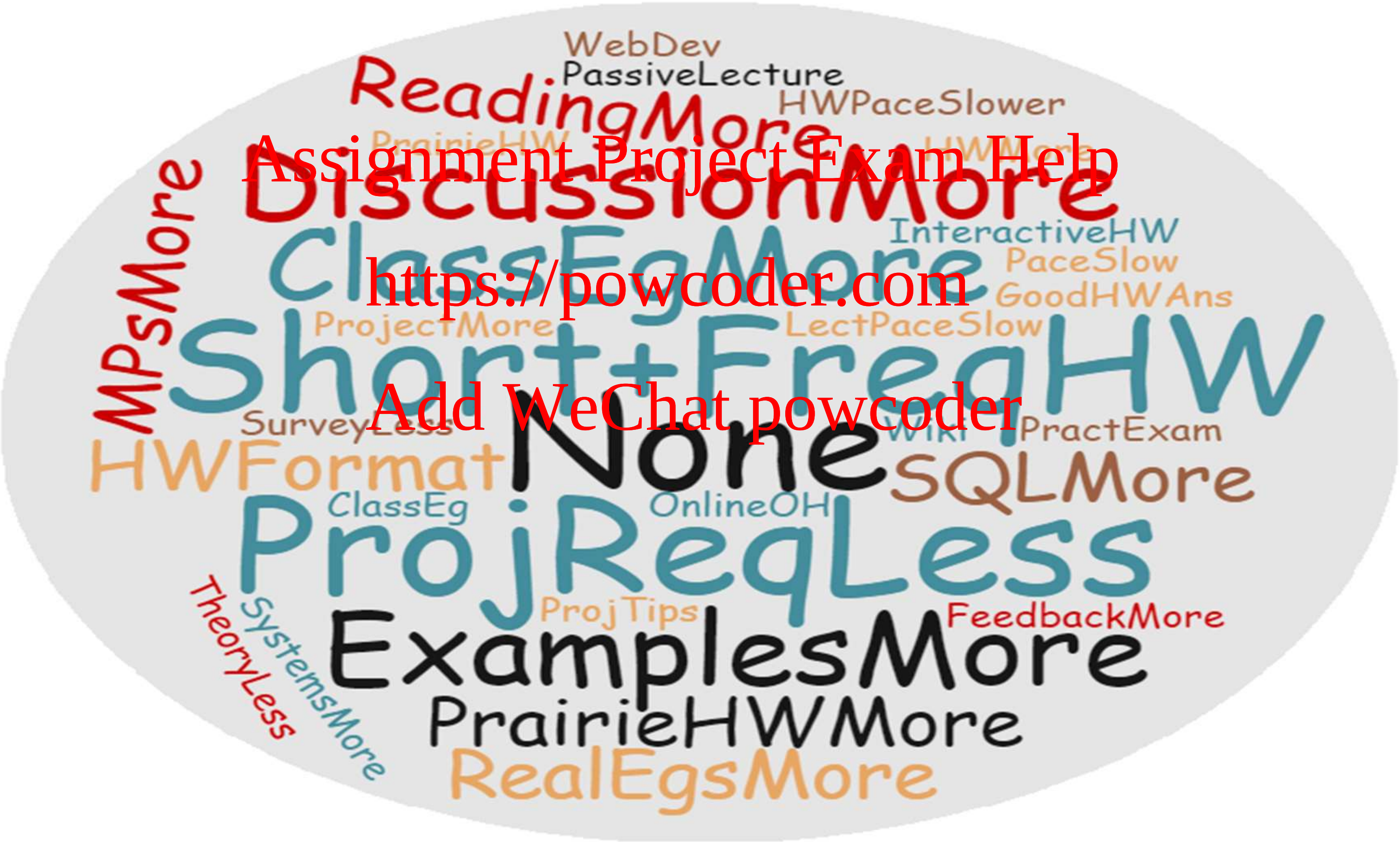
What is the best part of the class so far?



What is the least useful part of the course so far?

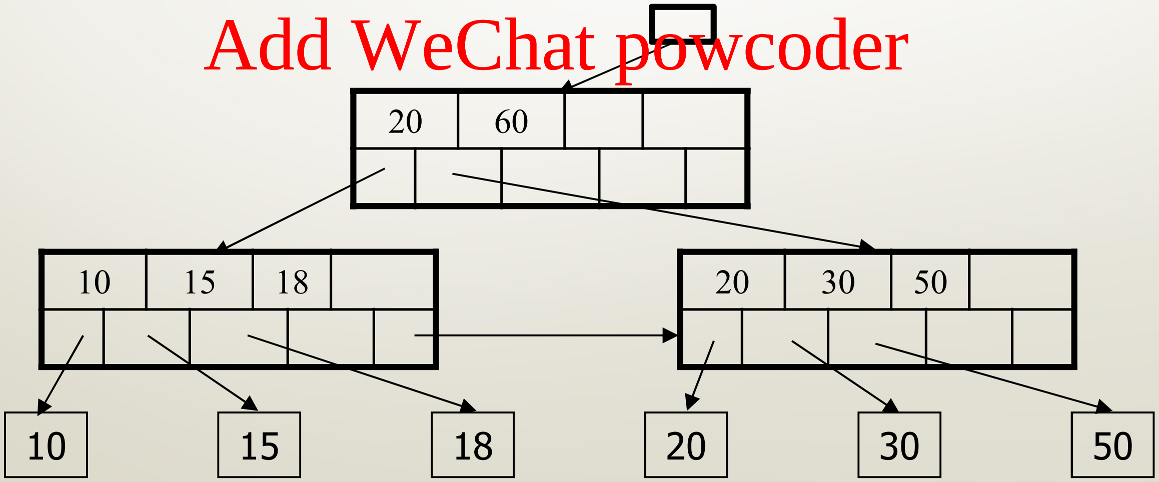


What would you change to make the course even better?



B+ Trees (Recap)

- Multi-level index *on a specific search key*
- Nodes contain **keys** and **pointers**
  - **Internal** nodes: point to other nodes
  - **Leaf** nodes: point to data records; last pointer to the next leaf node



B+ Trees (Recap con't)

- Each node can contain up to *n* keys;
  - All nodes have the same capacity (*n* max keys)
  - Degree $d = n/2$ → the *minimum* # of keys per node (assume *n* is even for simplicity)
  - Each node has between $[d, 2d]$ keys at all times
  - Except root node, which can have only one key.
- In practice, each node has its own file block
  - For a 4KB block, we can accommodate up to 340 keys (d=170).
  - In practice, d = 100, 66.5% fill-in factor → 133 keys
  - Visiting one node = one disk read (latency ~ 10^5-10^7 ns)
  - First 3 levels of can be cached in main memory (latency ~ 100 ns) to reduce disk reads

an artificial requirement to make B+ Trees "balanced"

B+ Trees (Recap con't)

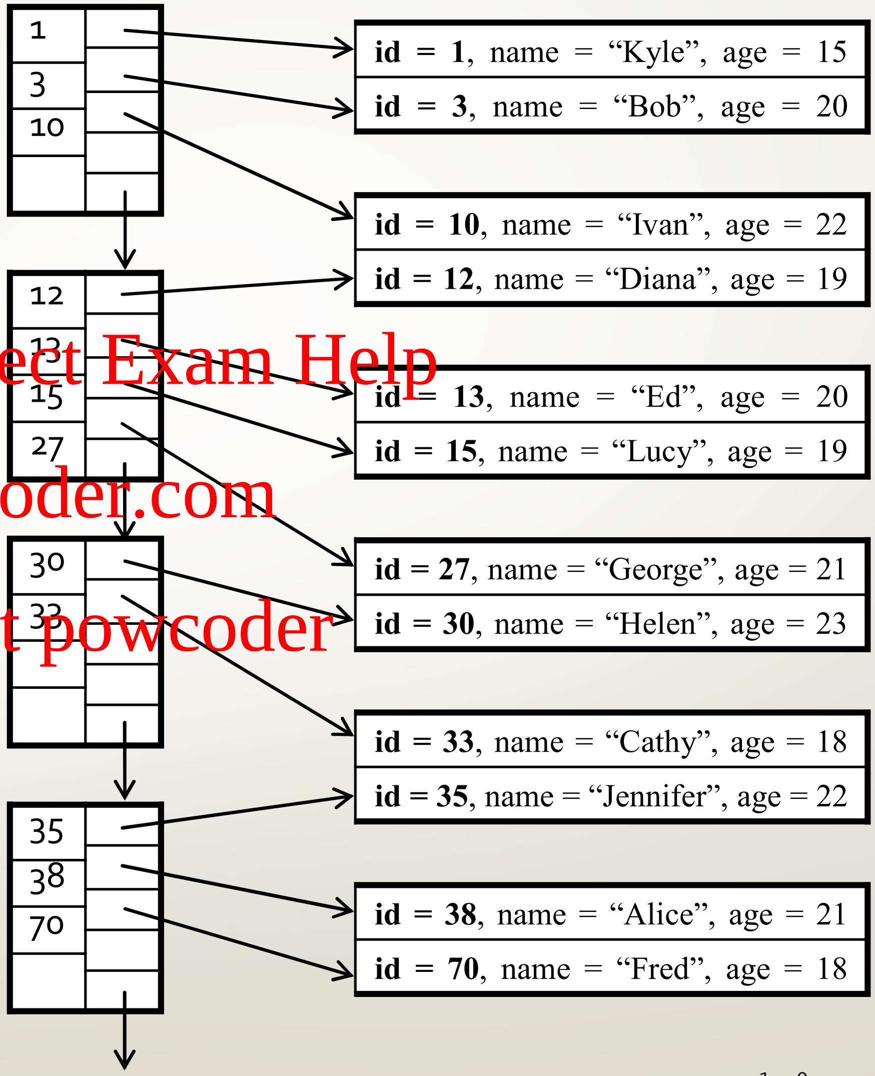
- Do B+ Trees always help?
  - No. e.g., an array of sorted integers.
- Types of queries to answer with a B+ Tree:
  - *Exact key value*, e.g., SELECT name FROM people WHERE age=20
  - *Range queries*, e.g., SELECT name FROM people WHERE age>=20 and age<=70

dense (entry for every record)

d = 2

n = 4

B+ Tree search key = **id**



id = **1**, name = “Kyle”, age = 15

id = **3**, name = “Bob”, age = 20

id = **10**, name = “Ivan”, age = 22

id = **12**, name = “Diana”, age = 19

id = **13**, name = “Ed”, age = 20

id = **15**, name = “Lucy”, age = 19

id = **27**, name = “George”, age = 21

id = **30**, name = “Helen”, age = 23

id = **33**, name = “Cathy”, age = 18

id = **35**, name = “Jennifer”, age = 22

id = **38**, name = “Alice”, age = 21

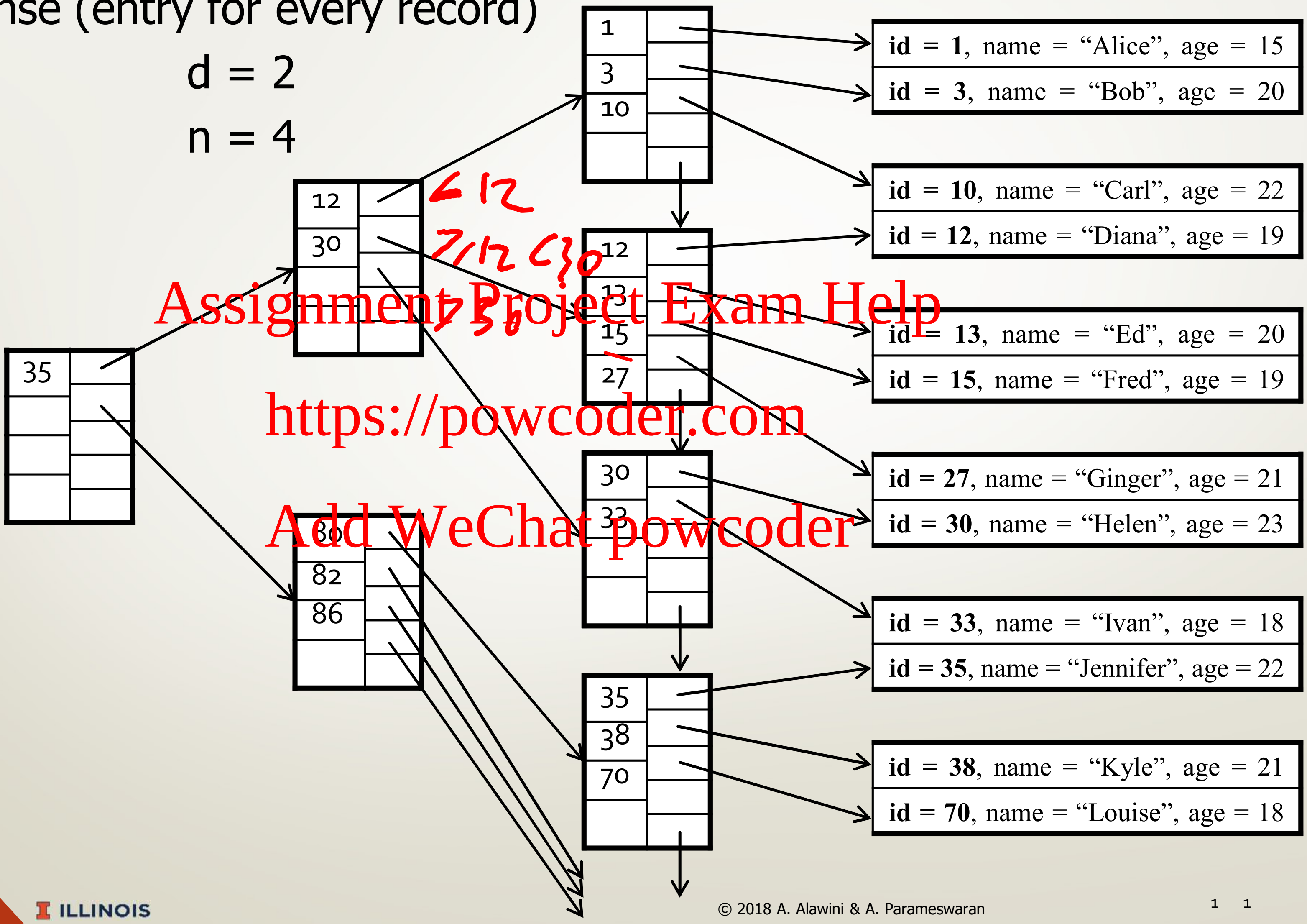
id = **70**, name = “Fred”, age = 18

B+ Tree search key = **id**

dense (entry for every record)

d = 2

n = 4



id = **1**, name = “Alice”, age = 15

id = **3**, name = “Bob”, age = 20

id = **10**, name = “Carl”, age = 22

id = **12**, name = “Diana”, age = 19

id = **13**, name = “Ed”, age = 20

id = **15**, name = “Fred”, age = 19

id = **27**, name = “Ginger”, age = 21

id = **30**, name = “Helen”, age = 23

id = **33**, name = “Ivan”, age = 18

id = **35**, name = “Jennifer”, age = 22

id = **38**, name = “Kyle”, age = 21

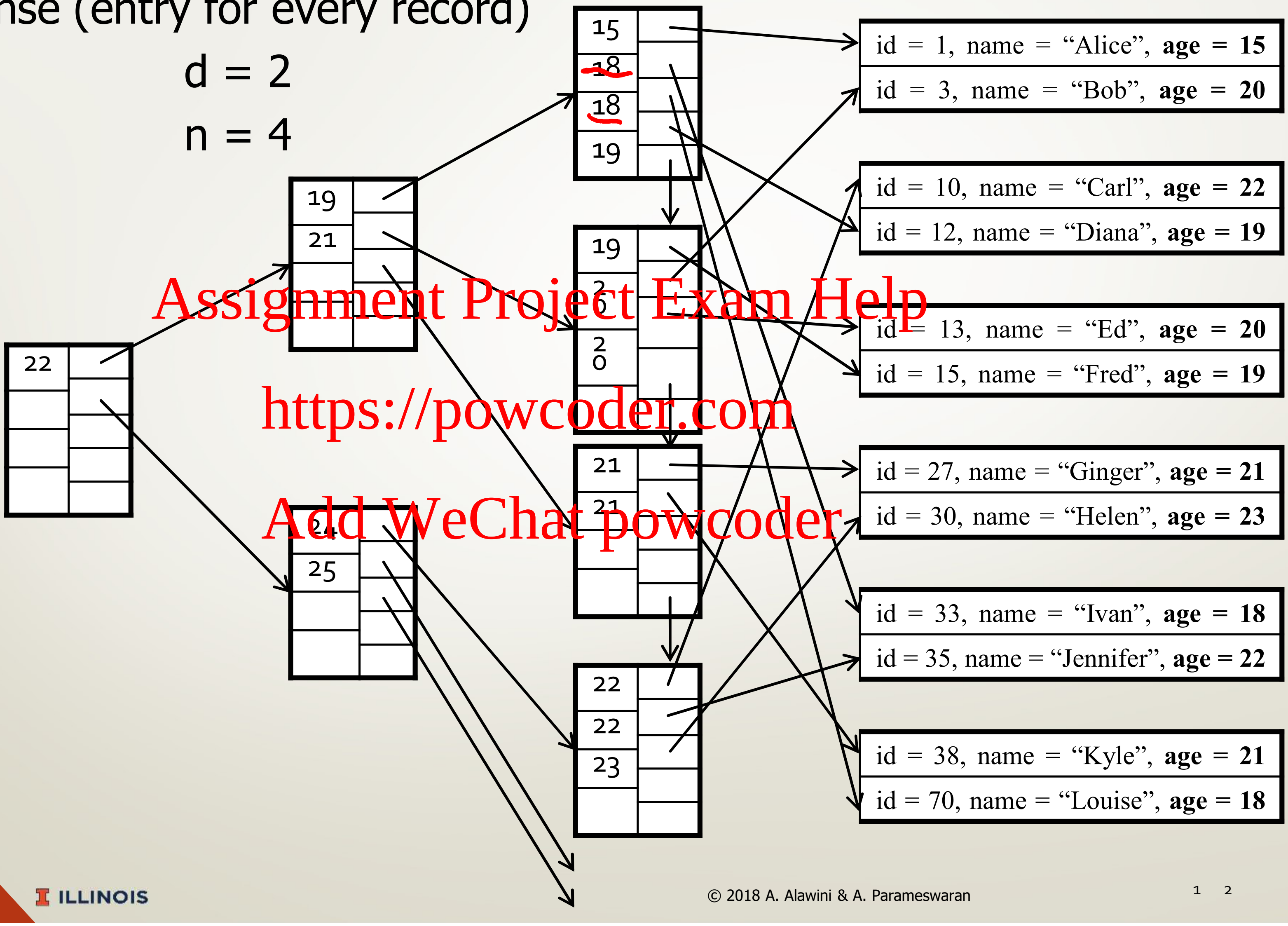
id = **70**, name = “Louise”, age = 18

B+ Tree search key = **age**

dense (entry for every record)

d = 2

n = 4



id = 1, name = “Alice”, **age** = **15**

id = 3, name = “Bob”, **age** = **20**

id = 10, name = “Carl”, **age** = **22**

id = 12, name = “Diana”, **age** = **19**

id = 13, name = “Ed”, **age** = **20**

id = 15, name = “Fred”, **age** = **19**

id = 27, name = “Ginger”, **age** = **21**

id = 30, name = “Helen”, **age** = **23**

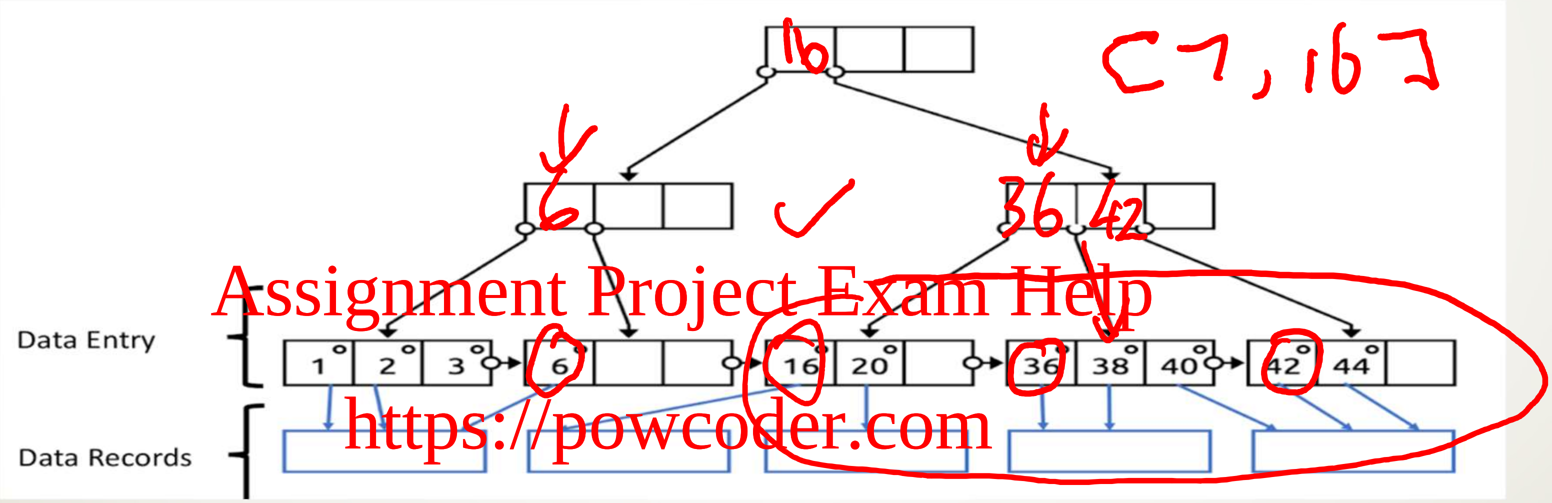
id = 33, name = “Ivan”, **age** = **18**

id = 35, name = “Jennifer”, **age** = **22**

id = 38, name = “Kyle”, **age** = **21**

id = 70, name = “Louise”, **age** = **18**

Think-Pair Share Exercise



Min capacity of a *data entry* page is 1 entry;
Min capacity of an *index page* is 2 pointers/1 key value.
Leaf nodes (data entry pages) point to data pages;
each pointer represents a record ID (RID).

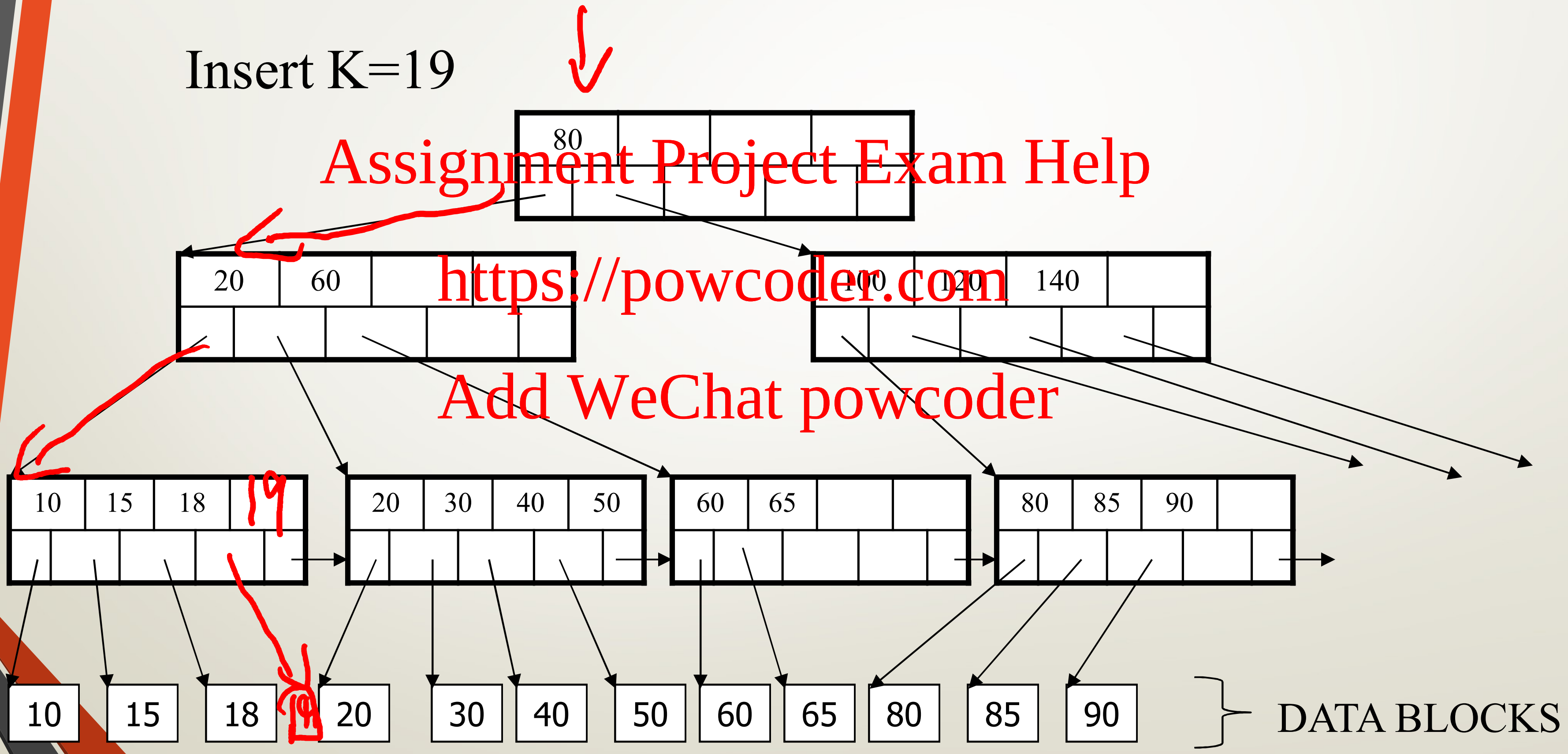
Write the key value index entries for the root and the intermediate nodes in the tree above.

Handling data changes in B+ Trees

Insertion in a B+ Tree

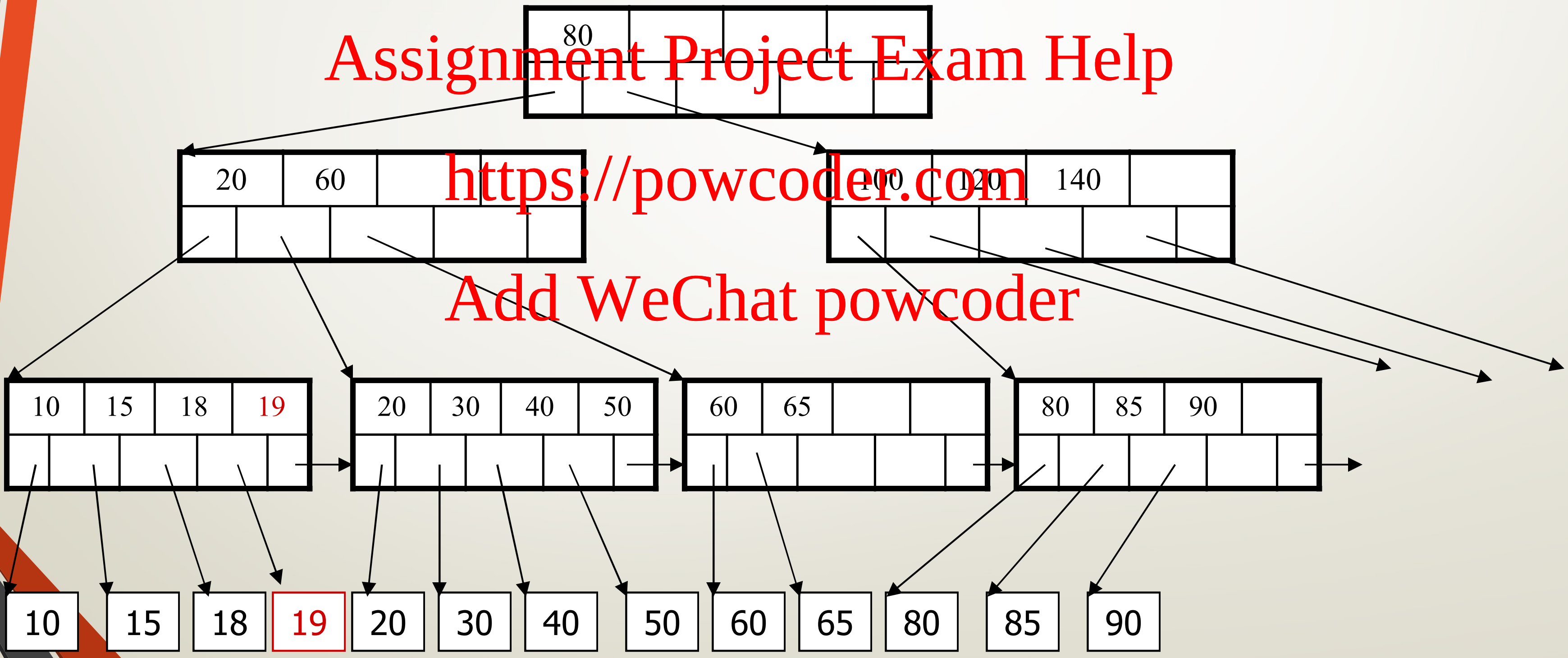
Assume d=2.

Insert K=19



Insertion in a B+ Tree

After insertion



80

20 60

100 120 140

10 15 18 19

20 30 40 50

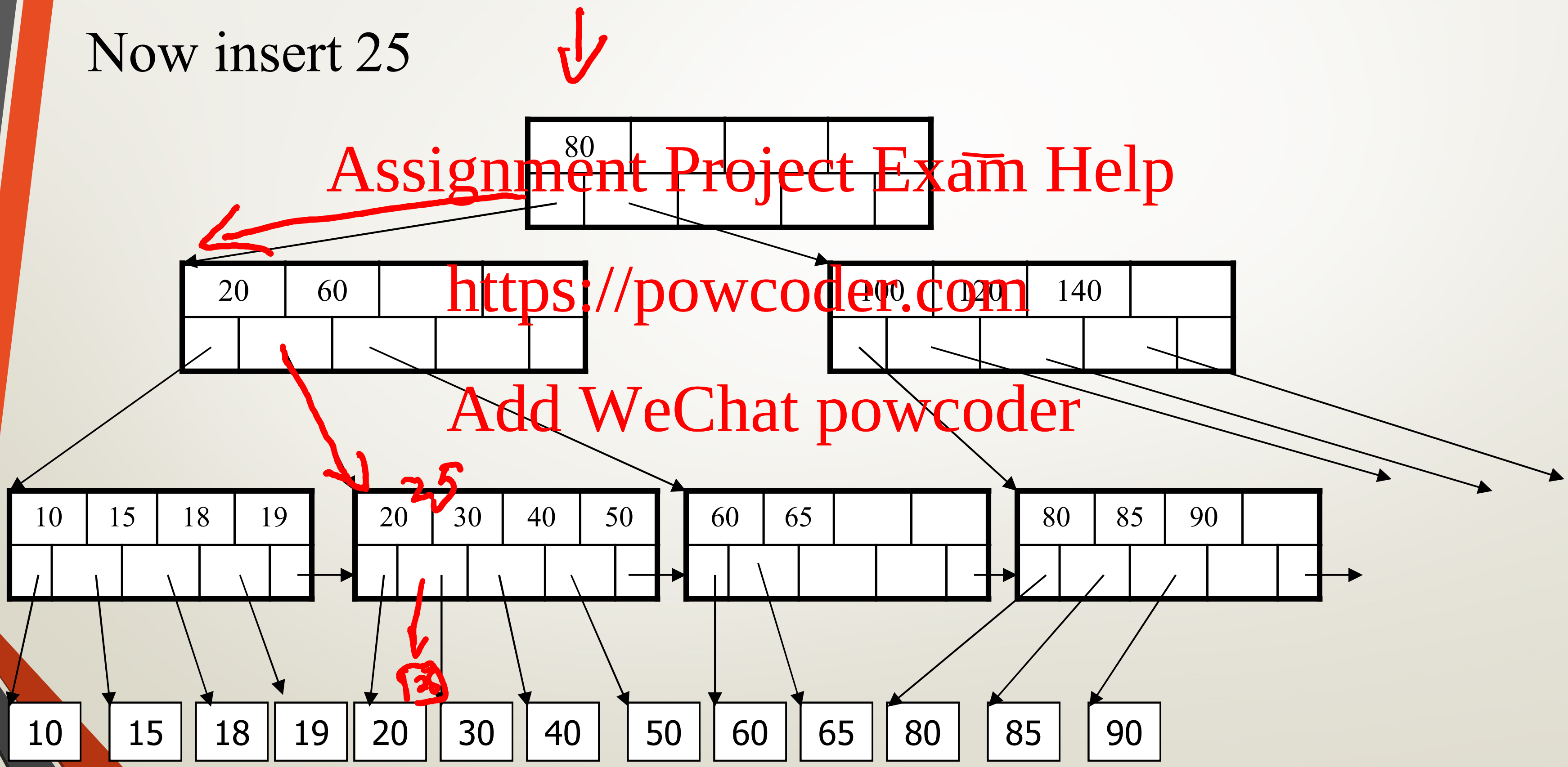
60 65

80 85 90

10 15 18 19 20 30 40 50 60 65 80 85 90

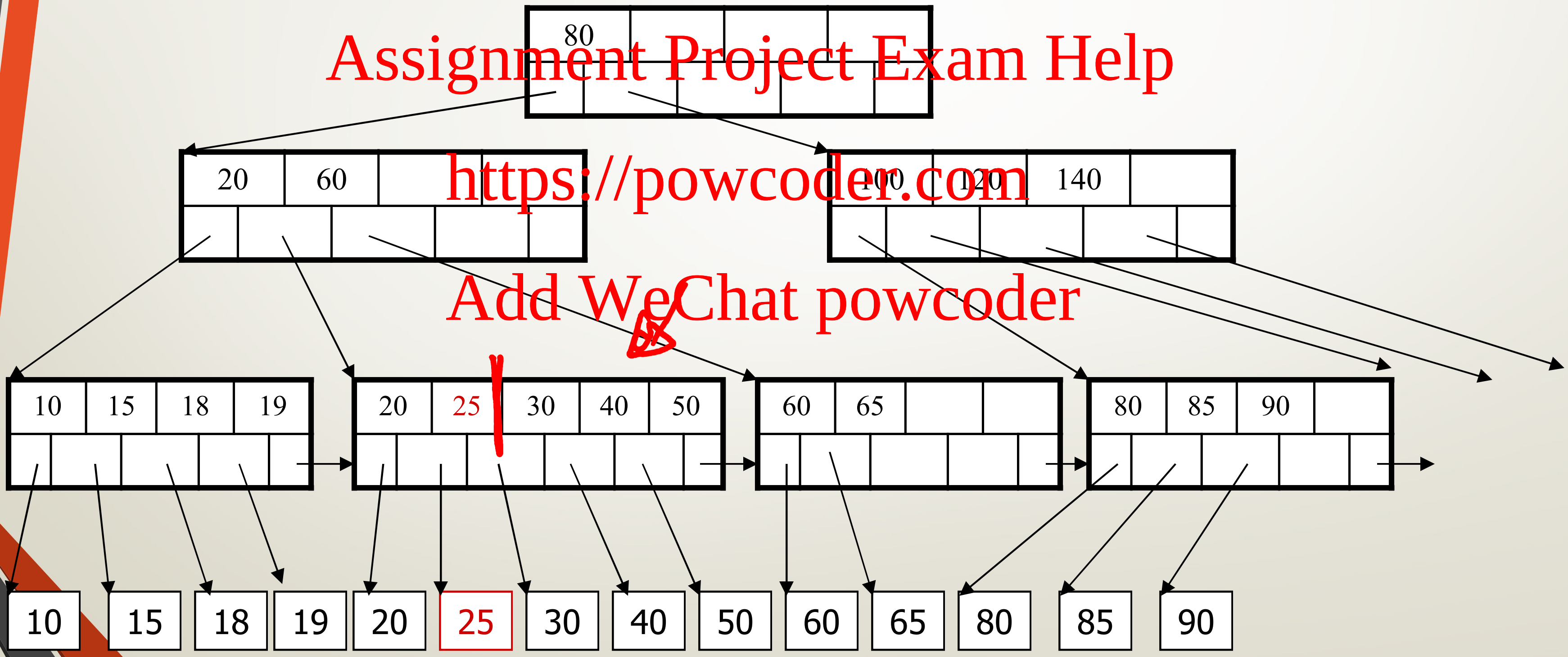
Insertion in a B+ Tree

Now insert 25



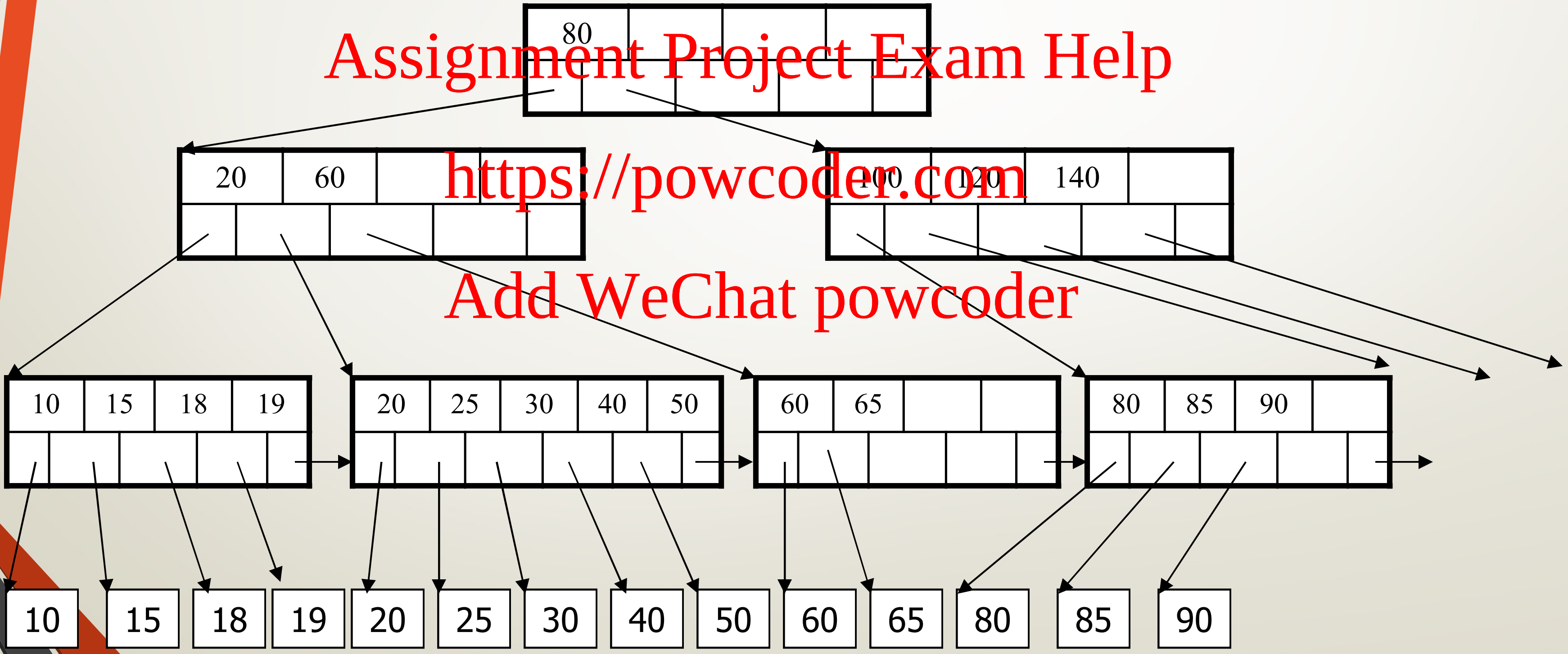
Insertion in a B+ Tree

After insertion



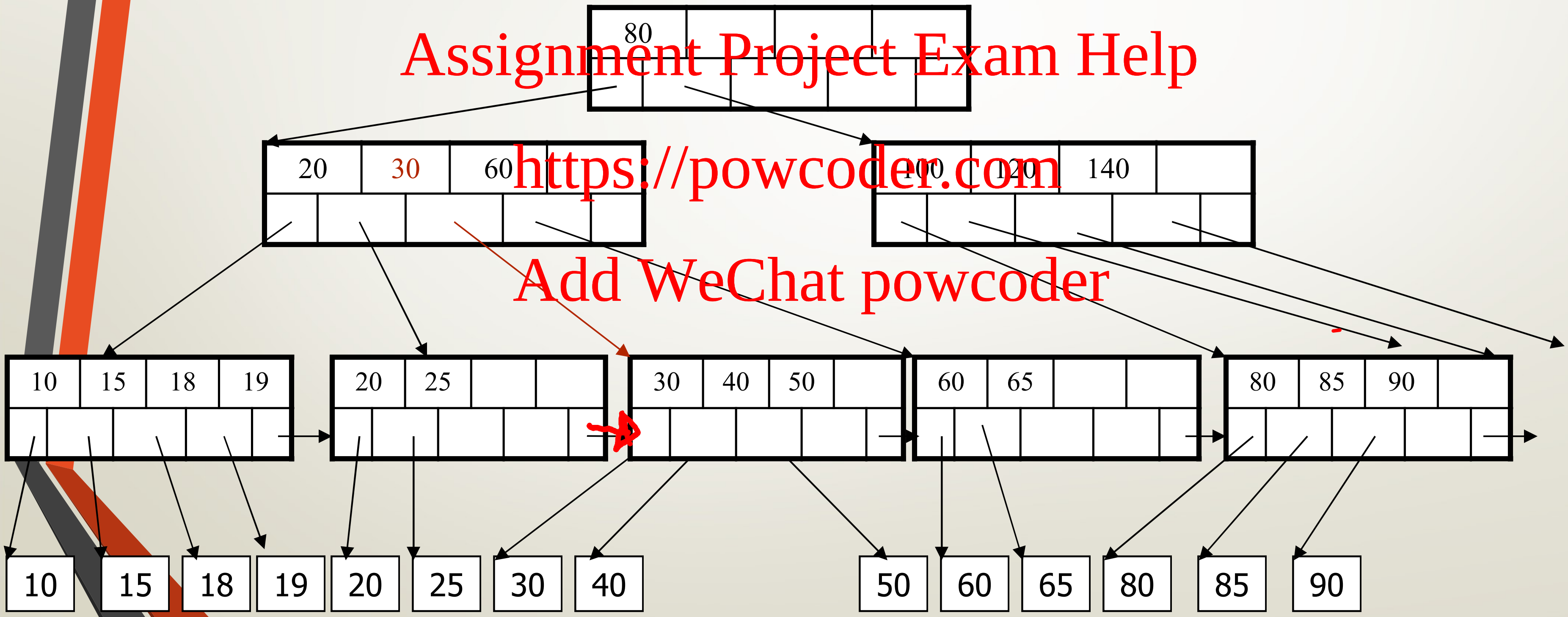
Insertion in a B+ Tree

But now have to split !



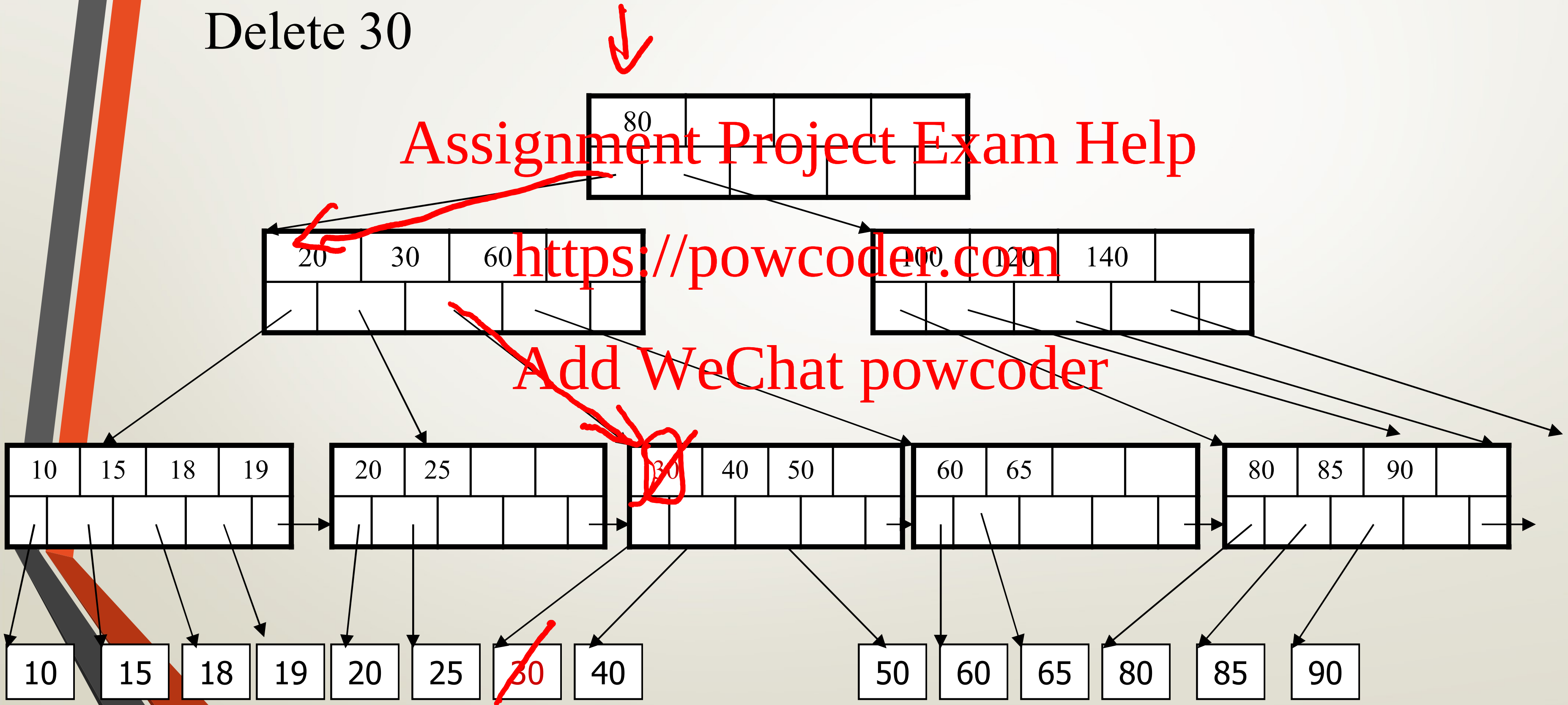
Insertion in a B+ Tree

After the split



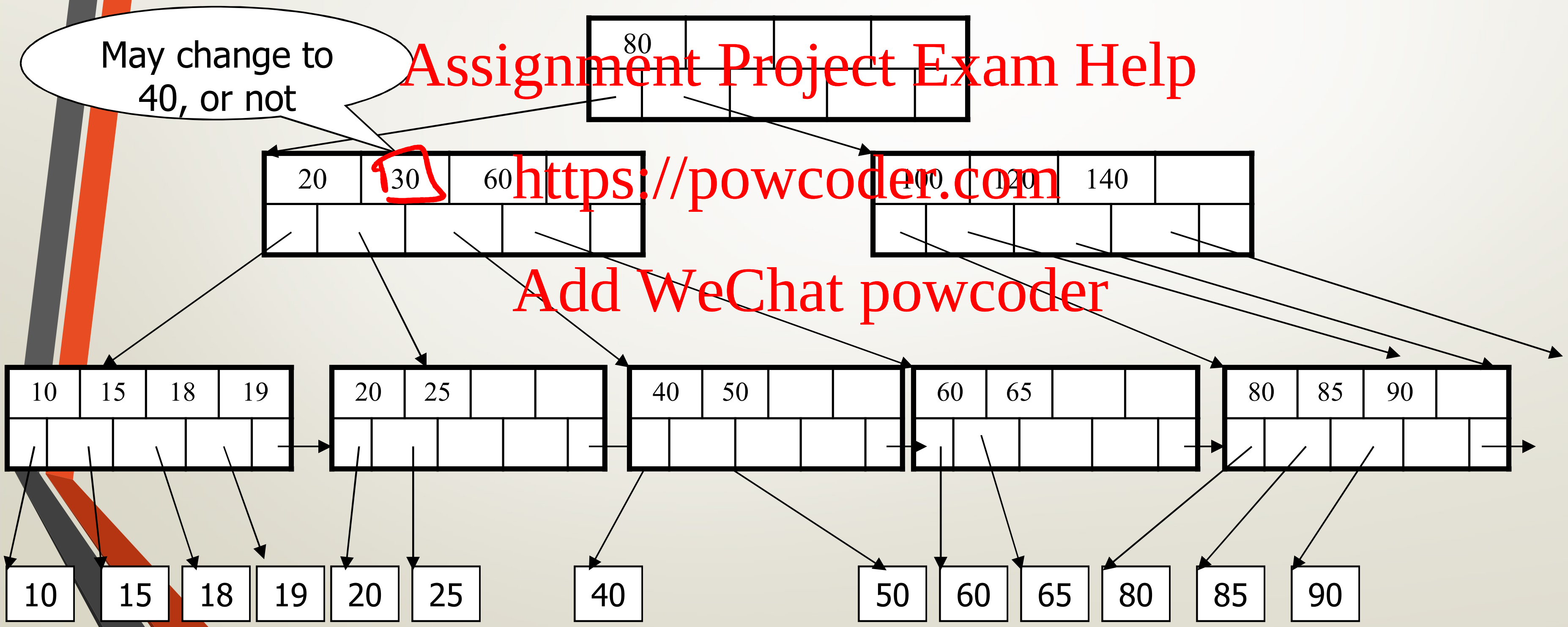
Deletion from a B+ Tree

Delete 30



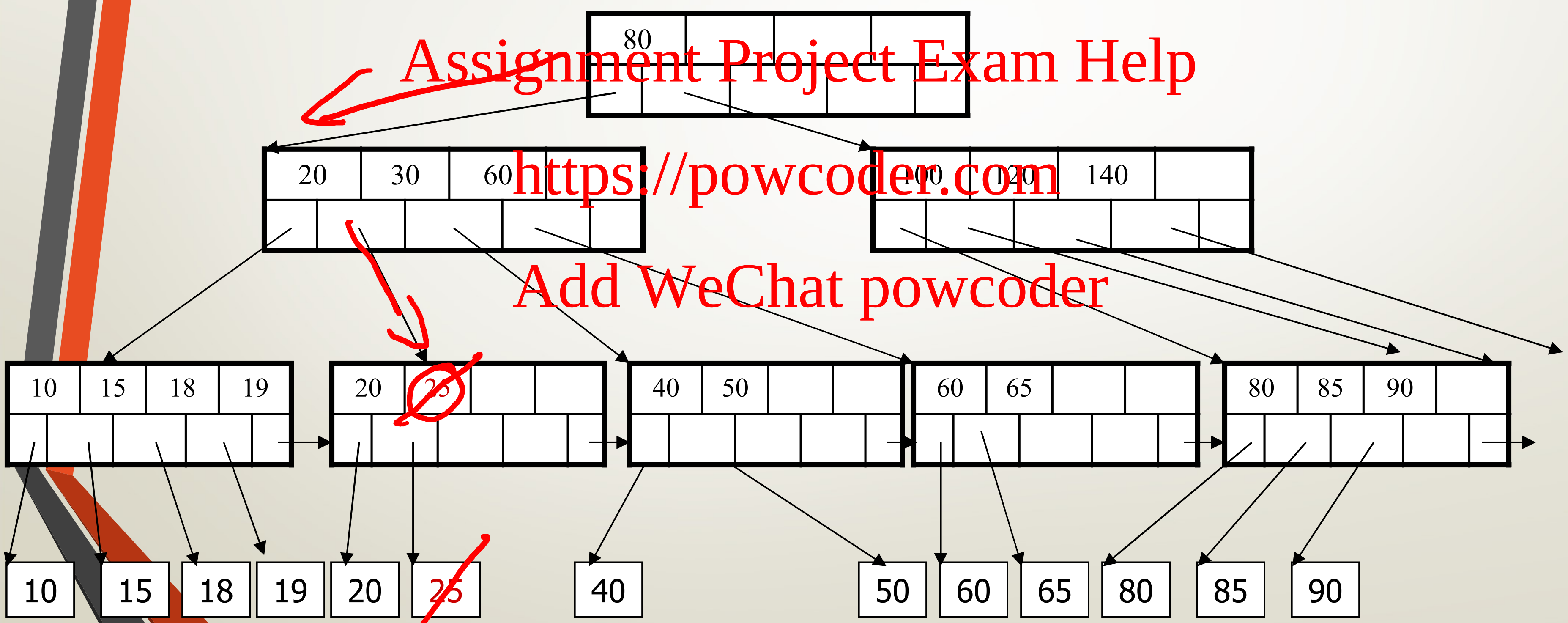
Deletion from a B+ Tree

After deleting 30



Deletion from a B+ Tree

Now delete 25



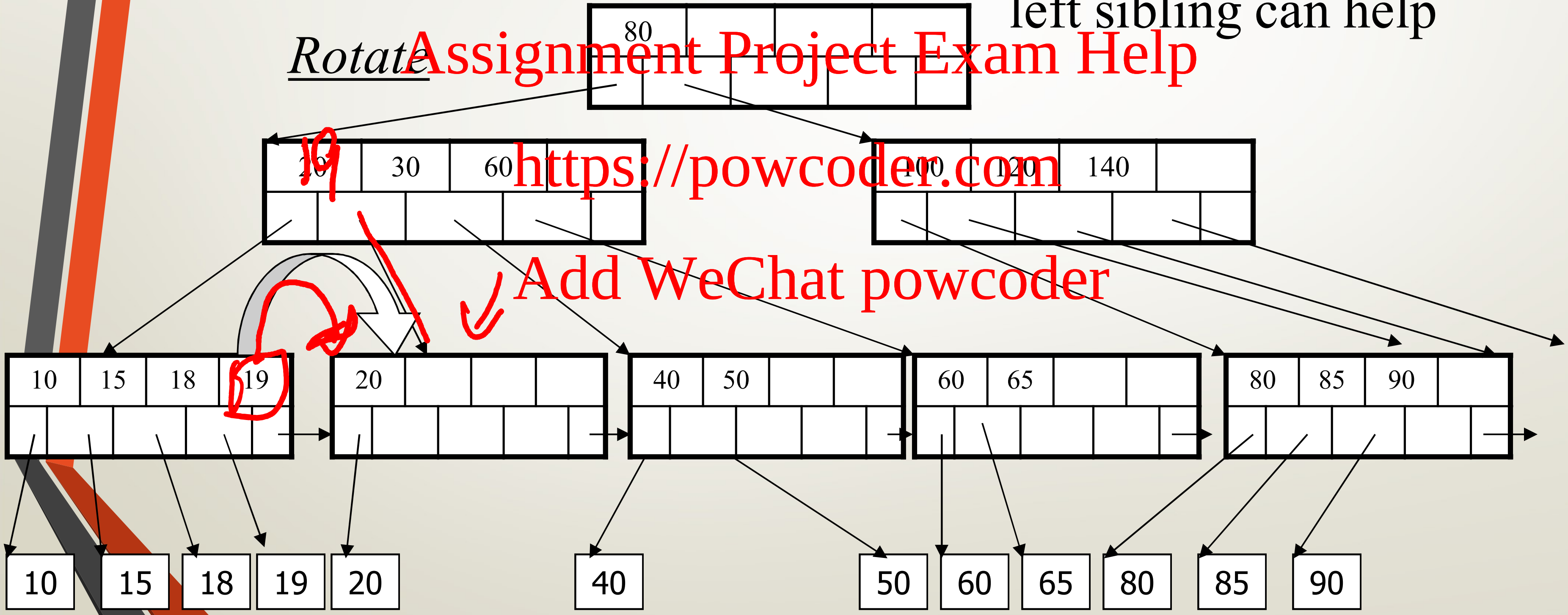
Deletion from a B+ Tree

After deleting 25

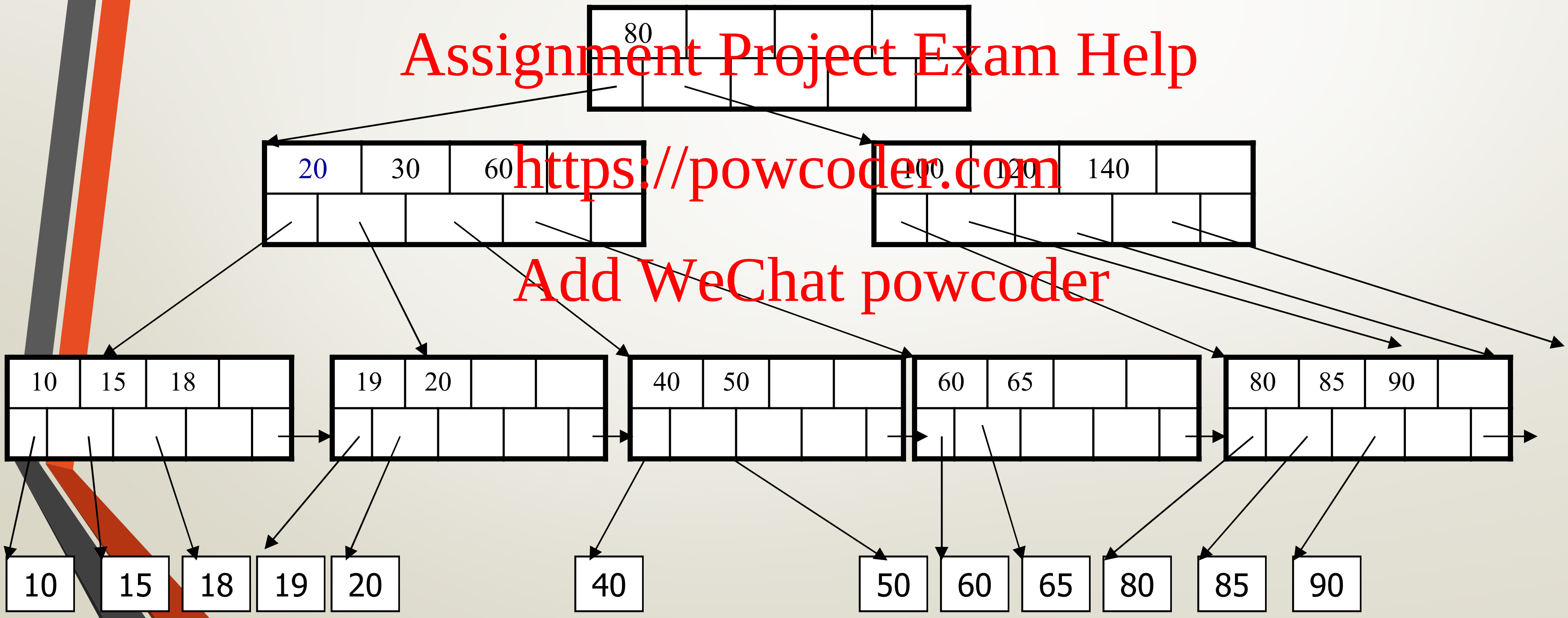
Need to rebalance

Rotate

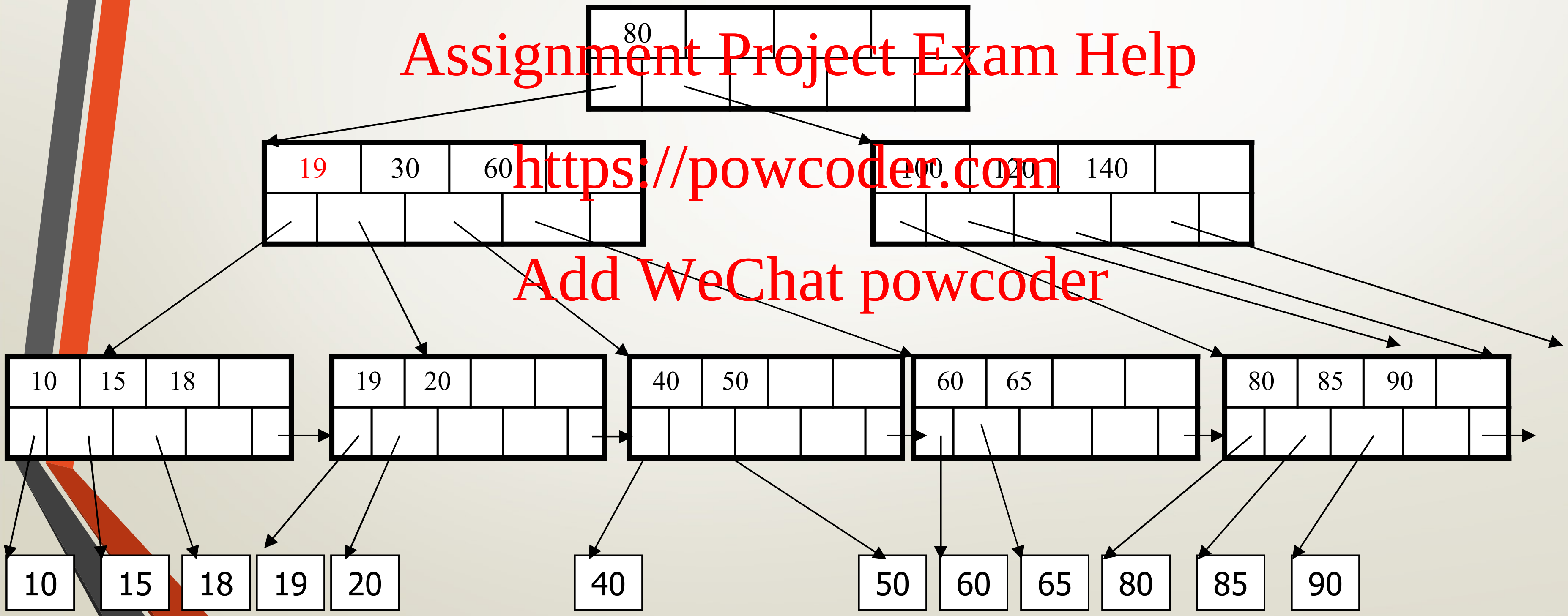
Rotation in general can involve either sibling, but here only the left sibling can help



Deletion from a B+ Tree

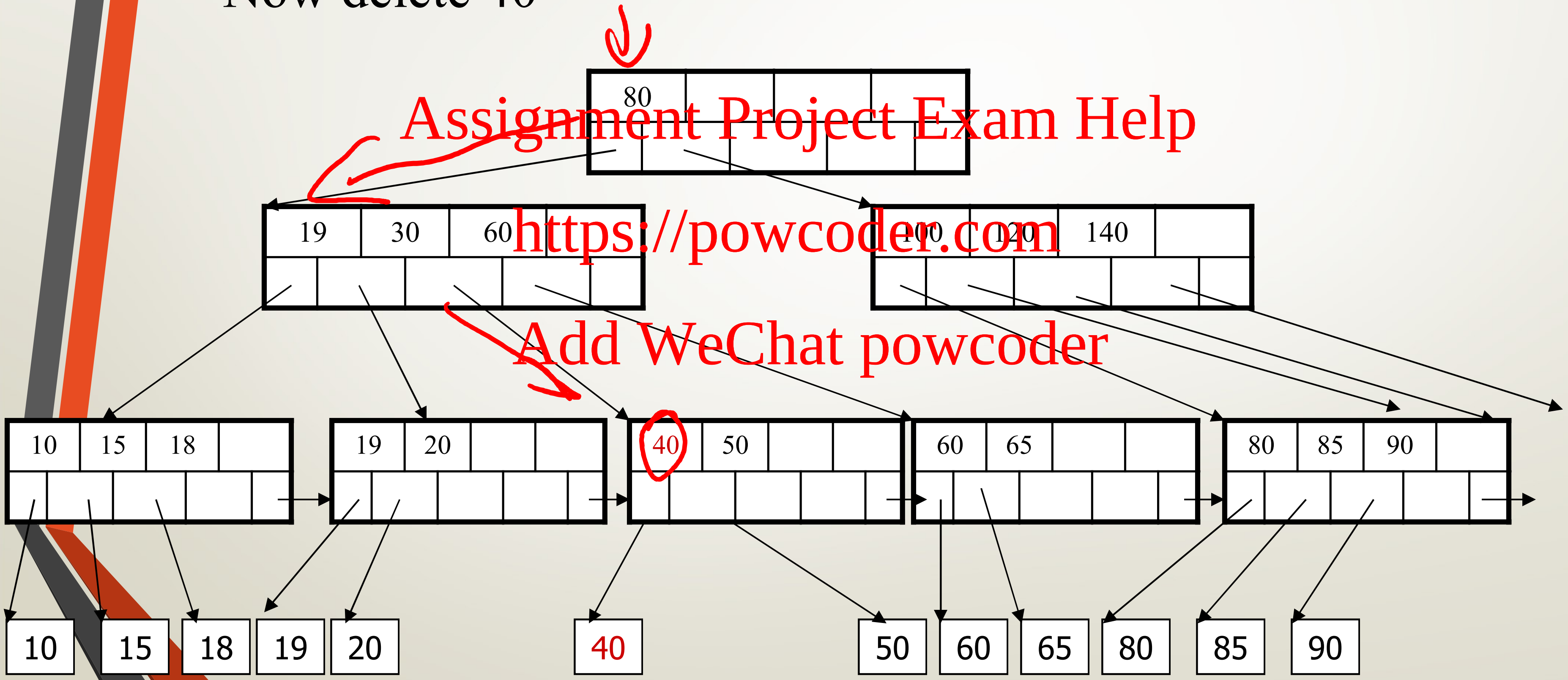


Deletion from a B+ Tree



Deletion from a B+ Tree

Now delete 40

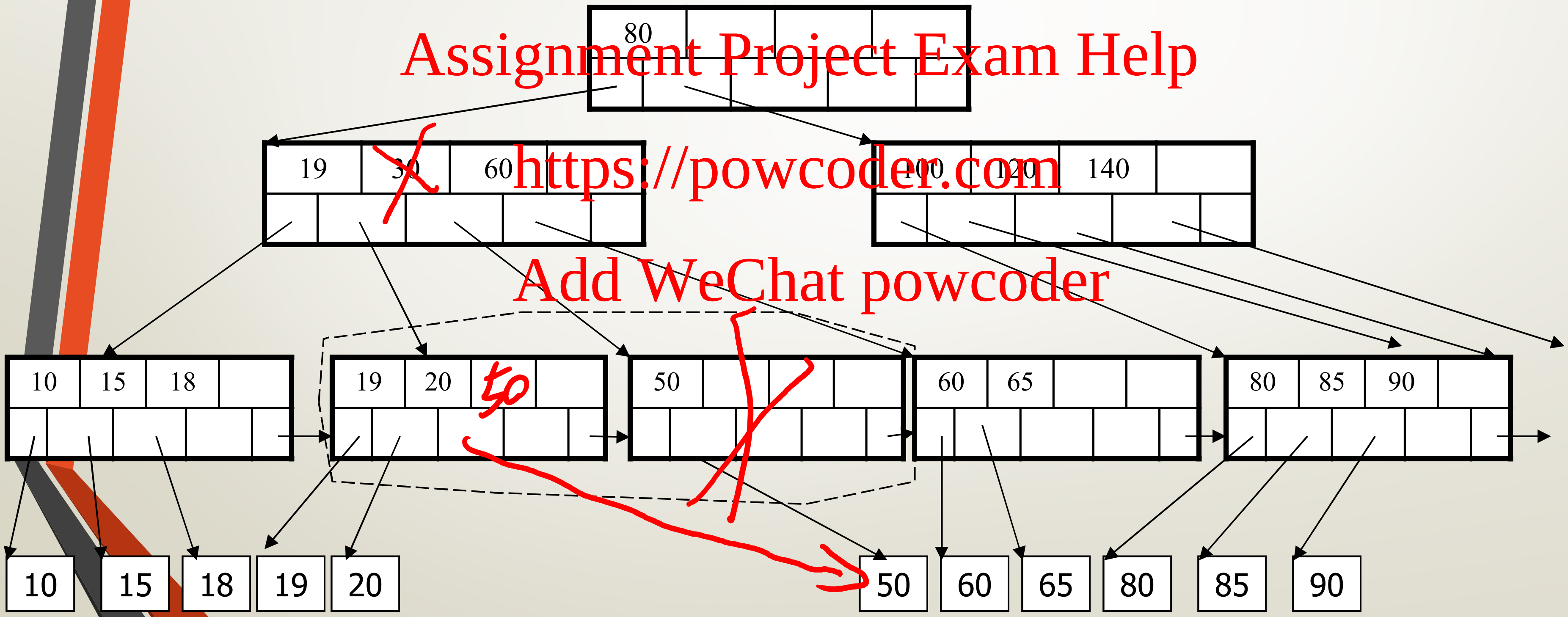


Deletion from a B+ Tree

After deleting 40

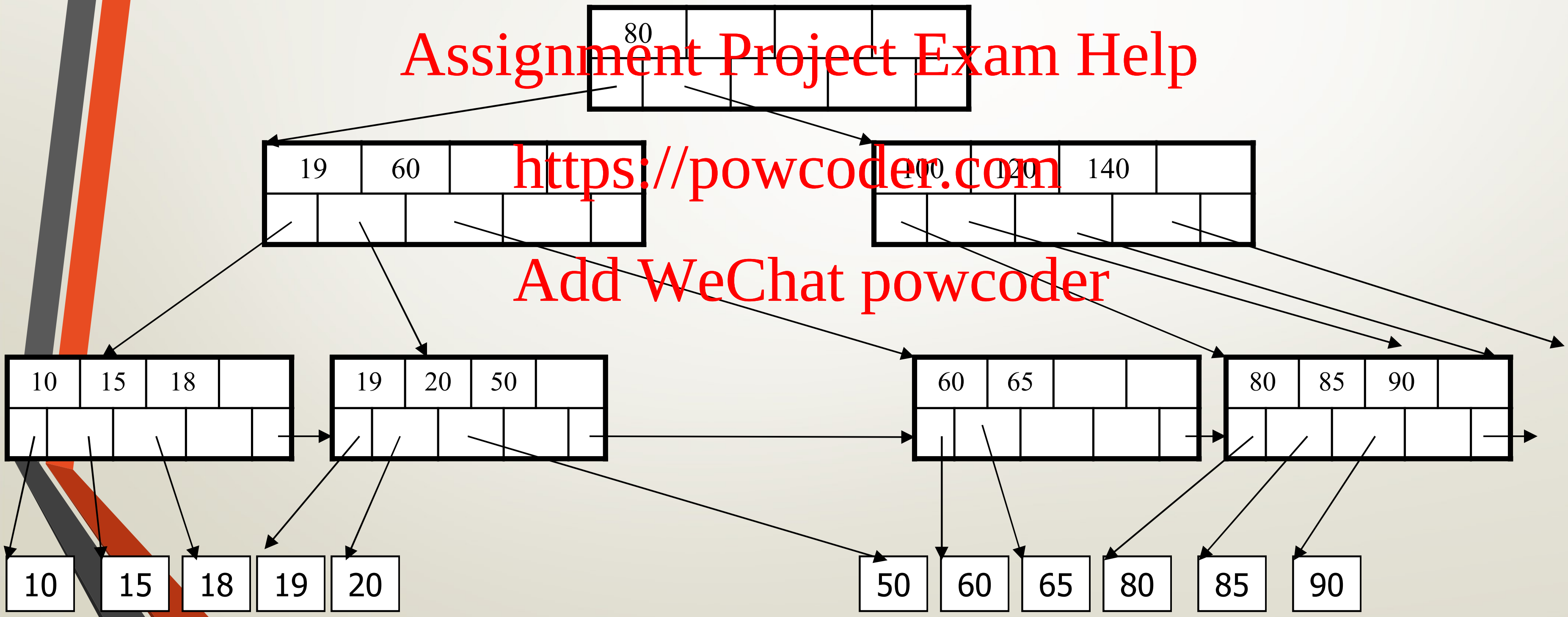
Rotation not possible

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

- Indexing
  - ✓ B+ Trees
  - Hash Tables

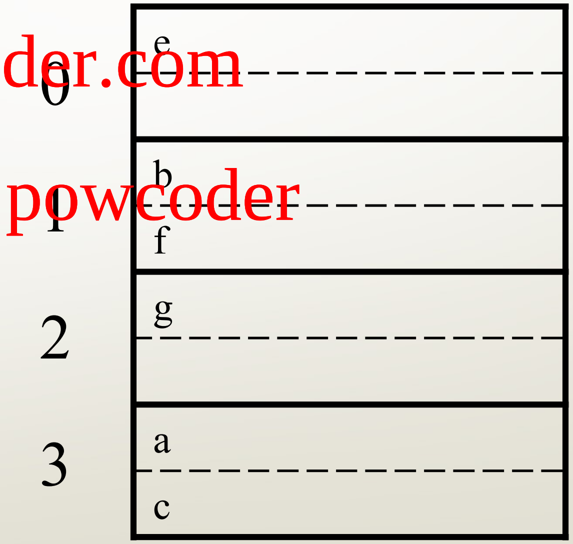
Hash Tables

Hash Tables

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

Hash Table Example

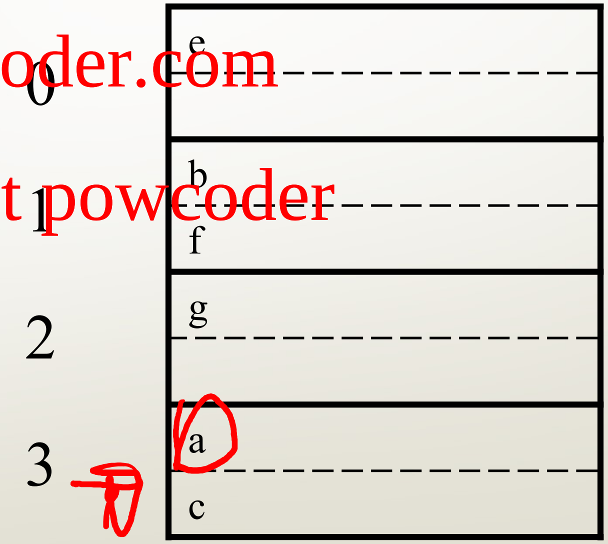
- Assume 1 bucket (block) stores 2 records
- h(e)=0
- h(b)=h(f)=1
- h(g)=2
- h(a)=h(c)=3



Searching in a Hash Table

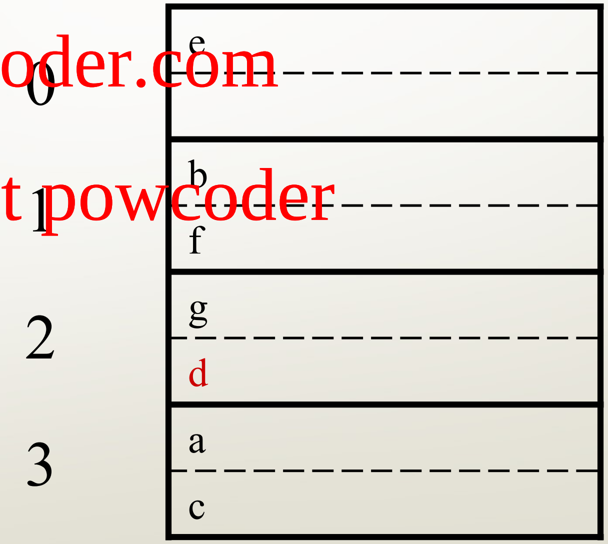
- Search for a:
- Compute h(a)=3
- Read bucket (block) 3
- 1 disk access

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



0 e

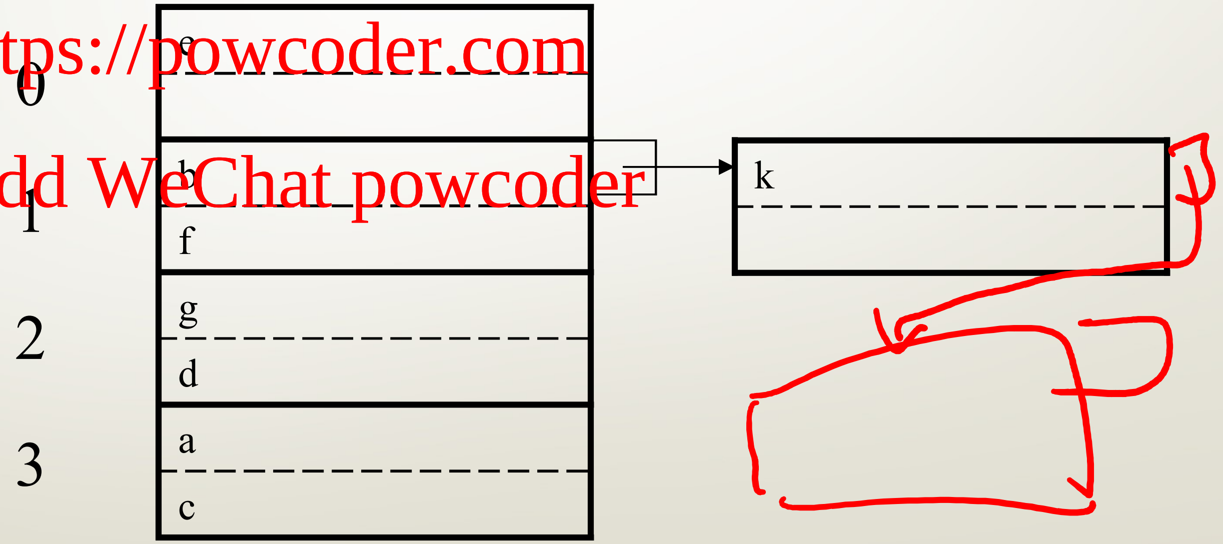
1 b f

2 g d

3 a c

Insertion in Hash Table

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



- More overflow 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 fix this by allowing the number of buckets to grow

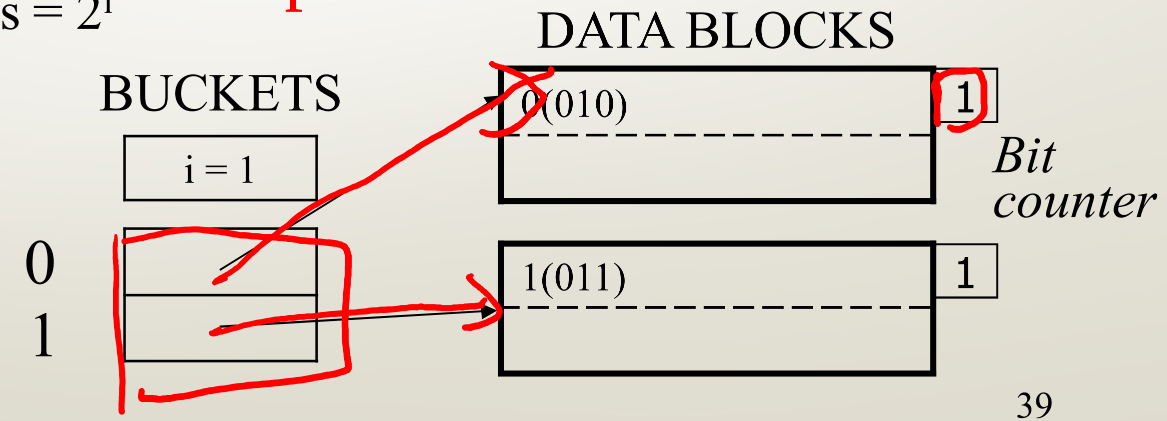
Extensible Hash Table

- 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)

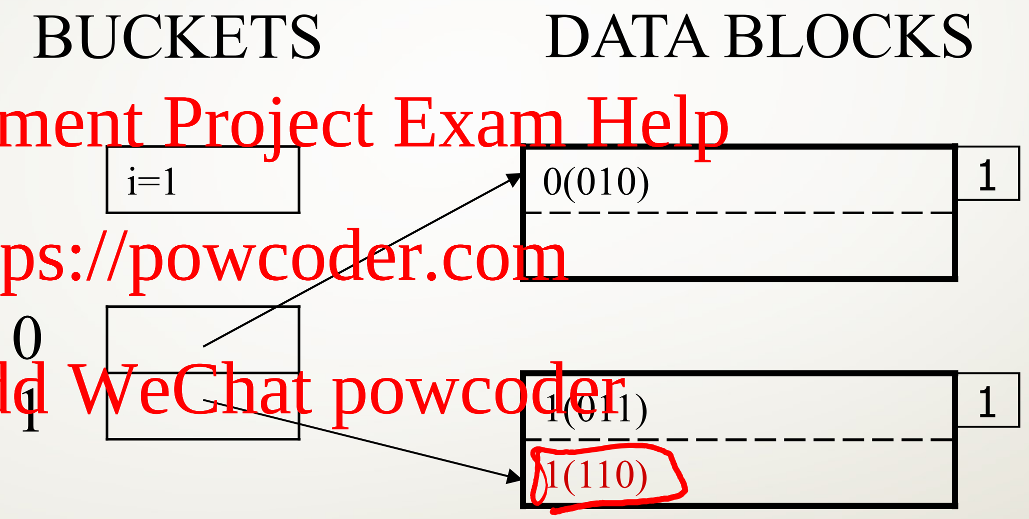
  - Only use the first i << k bits to determine bucket
  - Number of buckets = 2^i

- Bit counter on each block indicates how much are used for that block



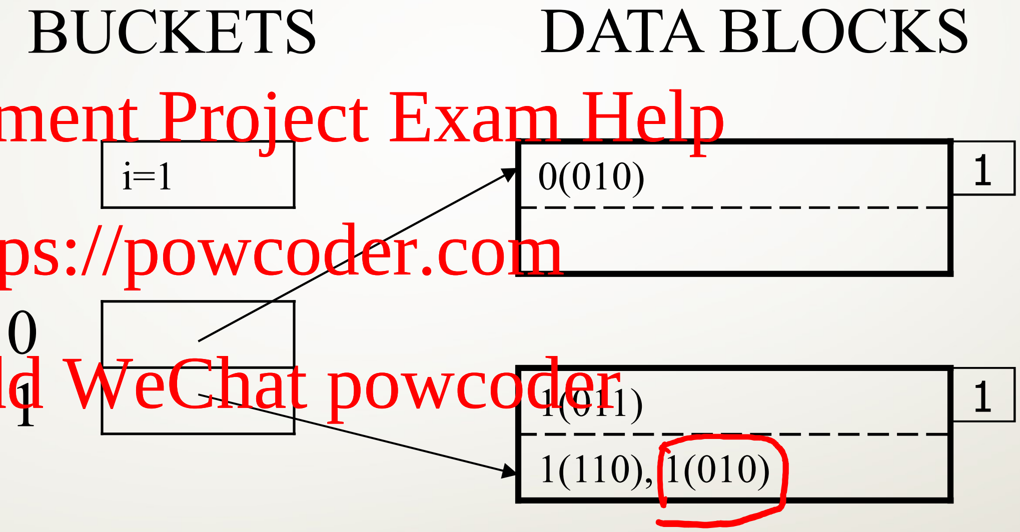
Insertion in Extensible Hash Table

- Insert 1110



Insertion in Extensible Hash Table

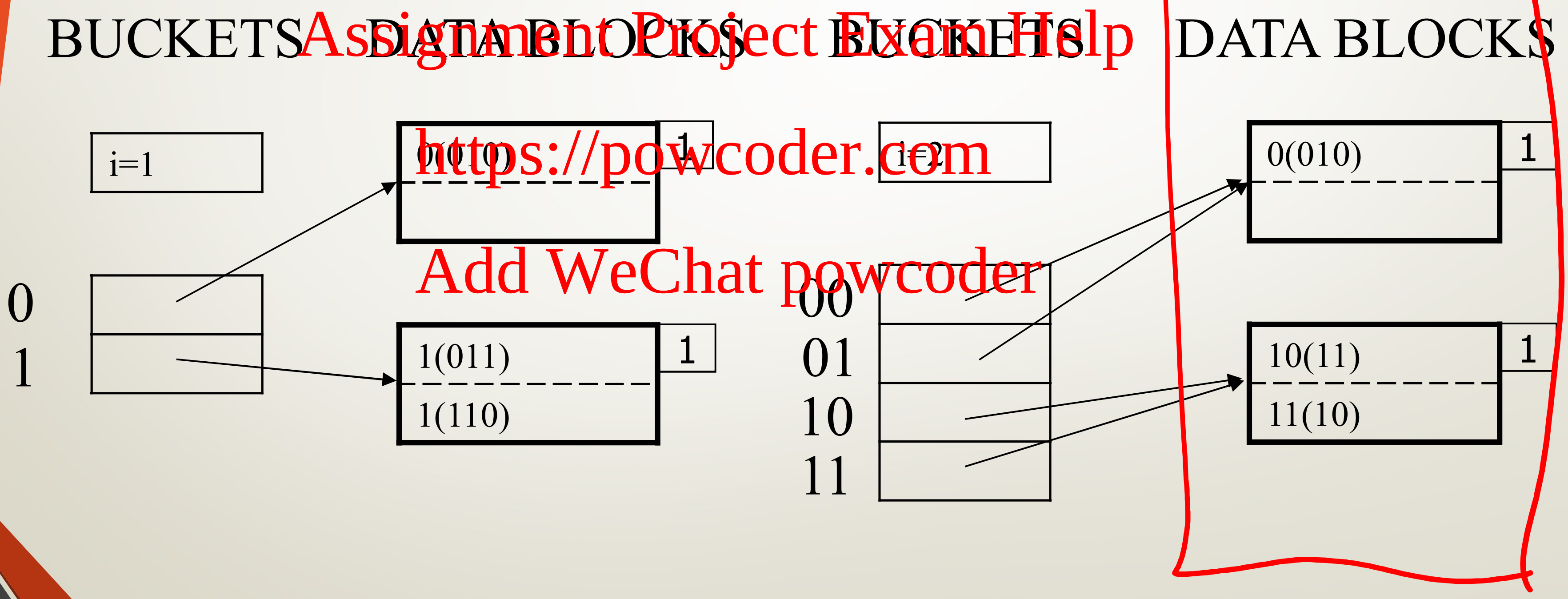
- Now insert 1010



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

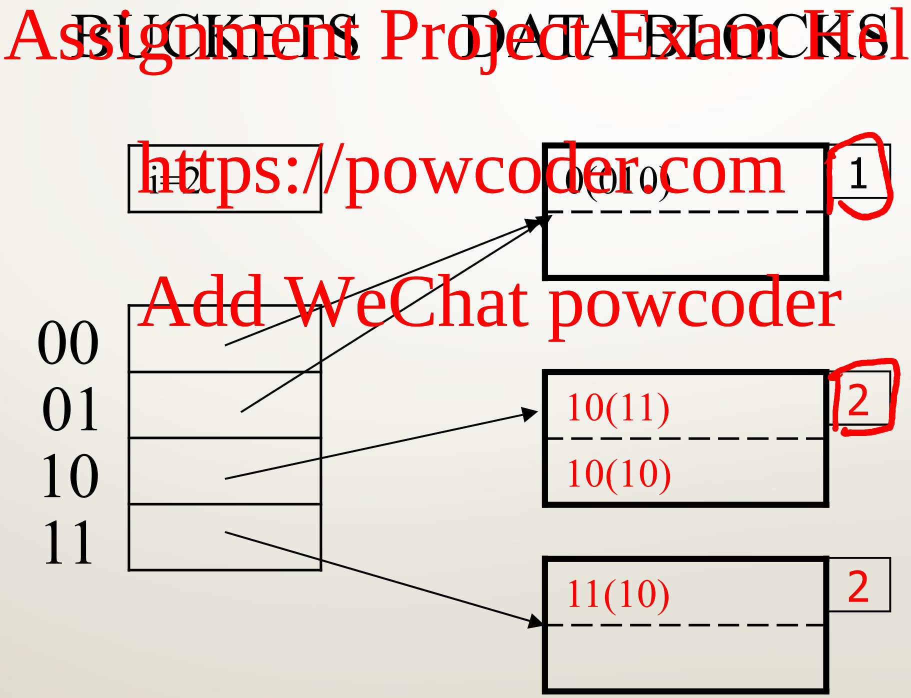
Insertion in Extensible Hash Table

Step 1: Extend the buckets



Insertion in Extensible Hash Table

Step 2: Now try to insert 1010



BUCKETS DATA BLOCKS

i=2

00
01
10
11

0(010)

1

10(11)

10(10)

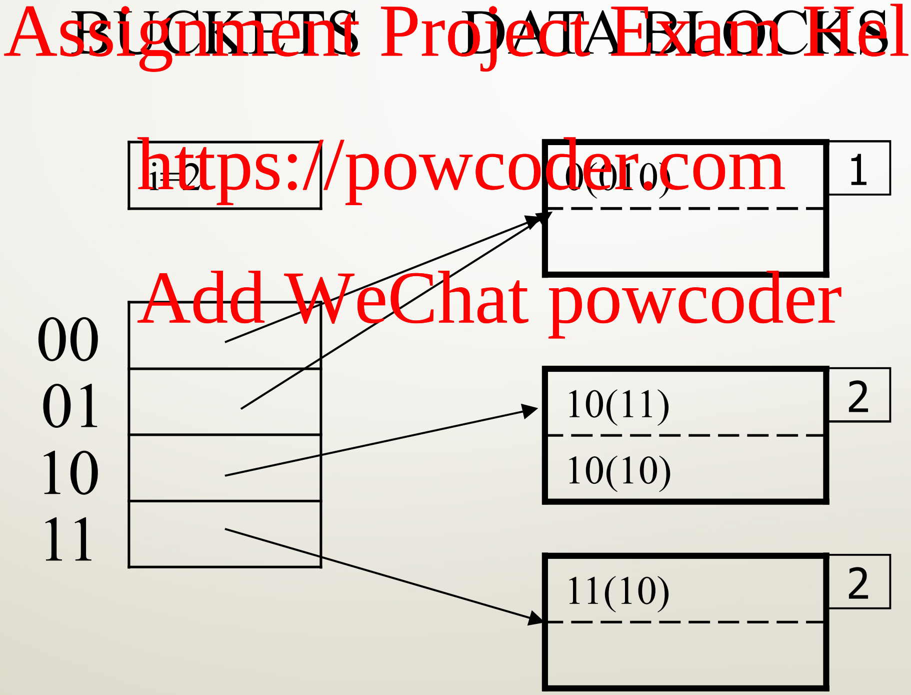
2

11(10)

2

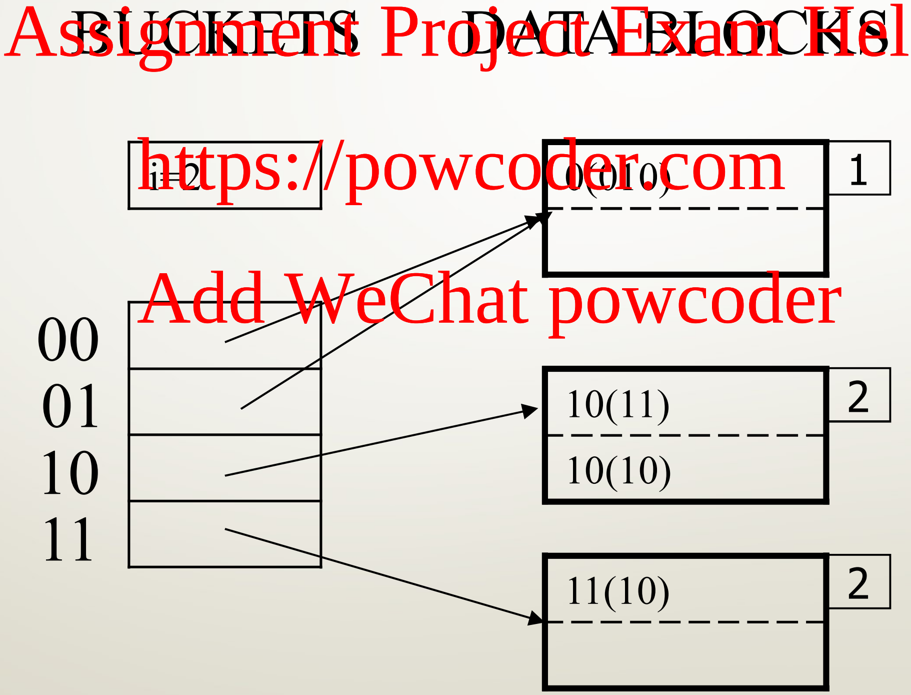
Insertion in Extensible Hash Table

Done



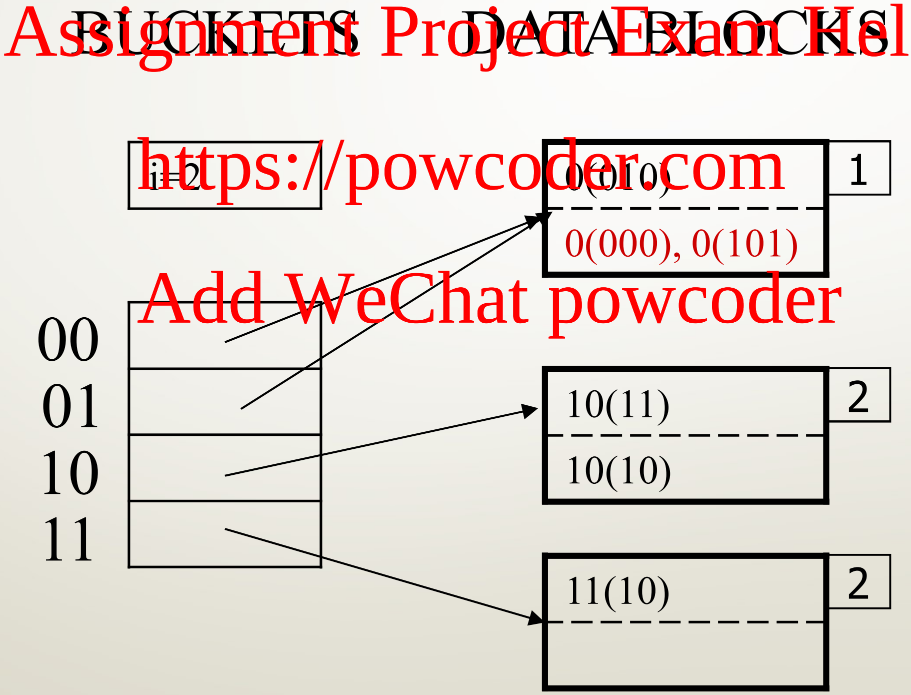
Insertion in Extensible Hash Table

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



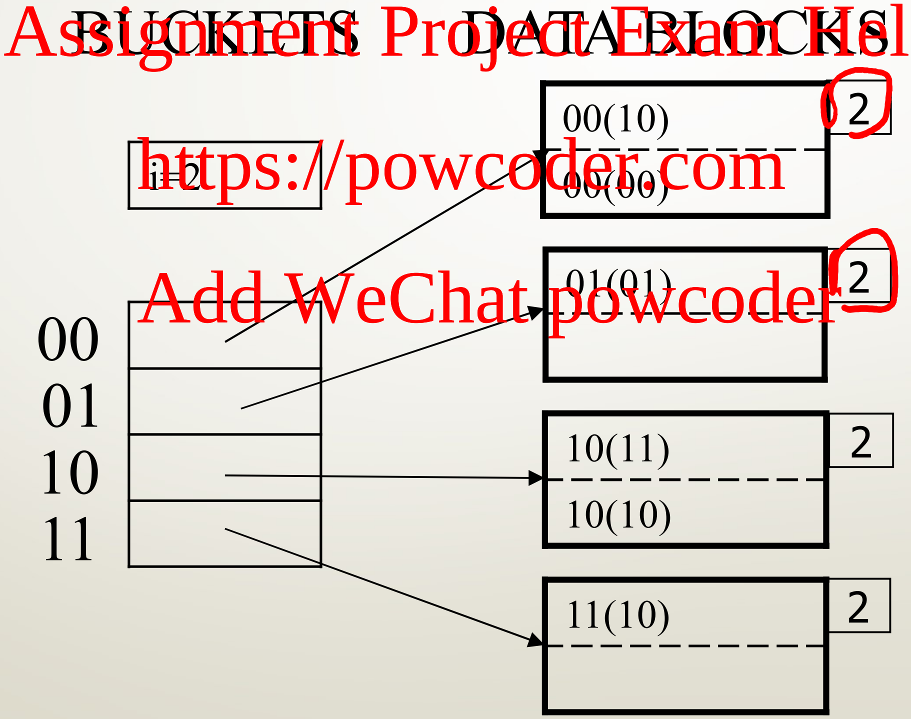
Insertion in Extensible Hash Table

- 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.

Linear Hash Table

- Idea 1: add only one bucket at a time

  Problem: n = no longer a power of 2

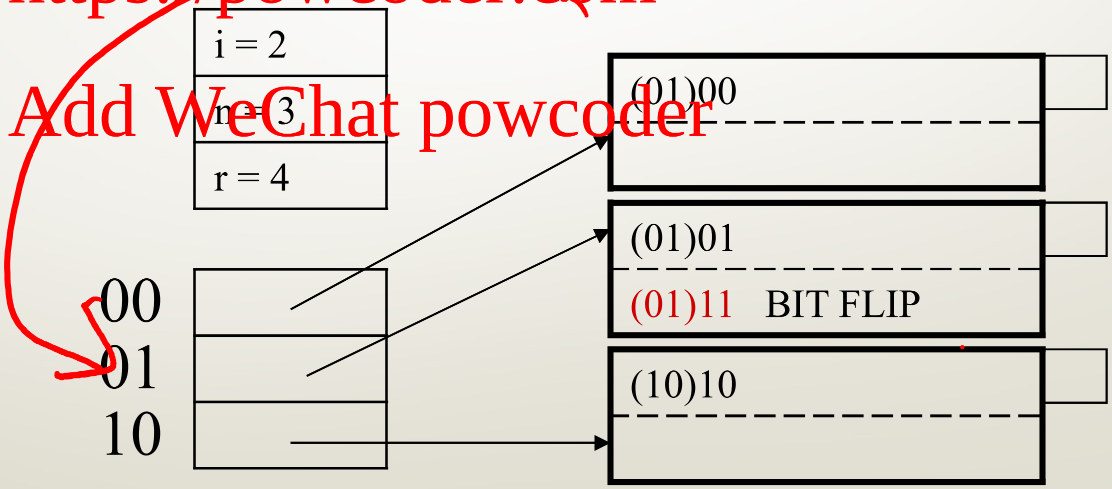
- Let i be # bits necessary to address n buckets.
  - i = ceil($\log^2$ n)

- After computing h(k), use ***last*** i bits:

  - If last i bits represent a number >= 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
  - When inserting 0111, 11 is flipped => 01



i = 2

n = 3

r = 4

00

01

10

(01)00

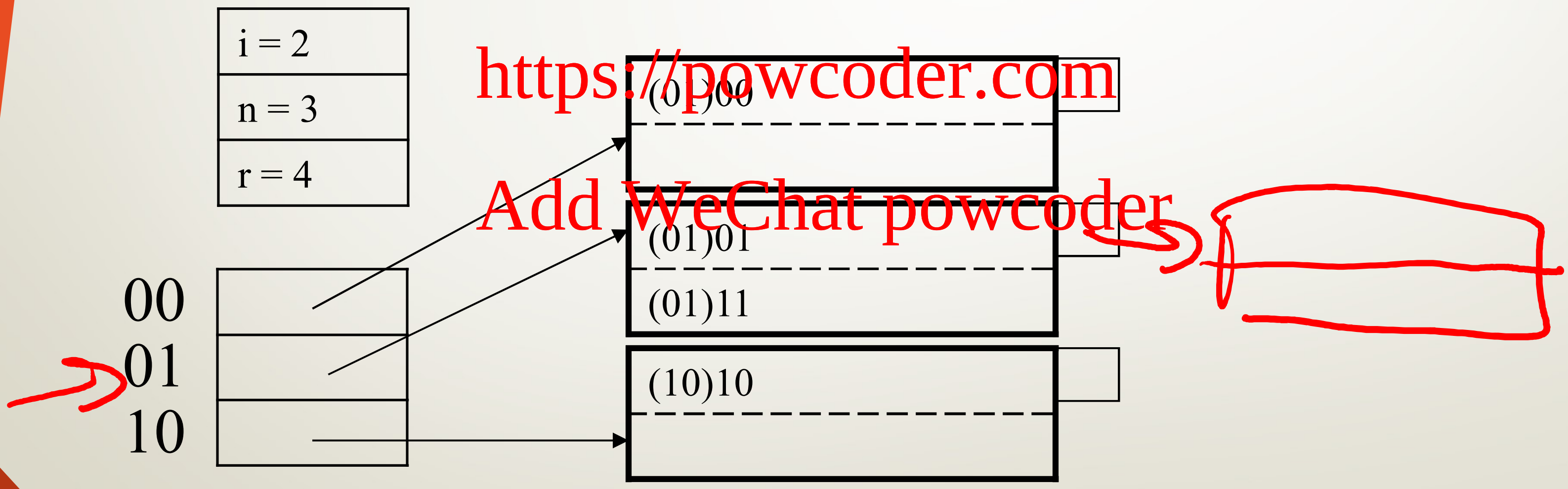
(01)01

(01)11 BIT FLIP

(10)10

Linear Hash Table Example

- Insert 1001:



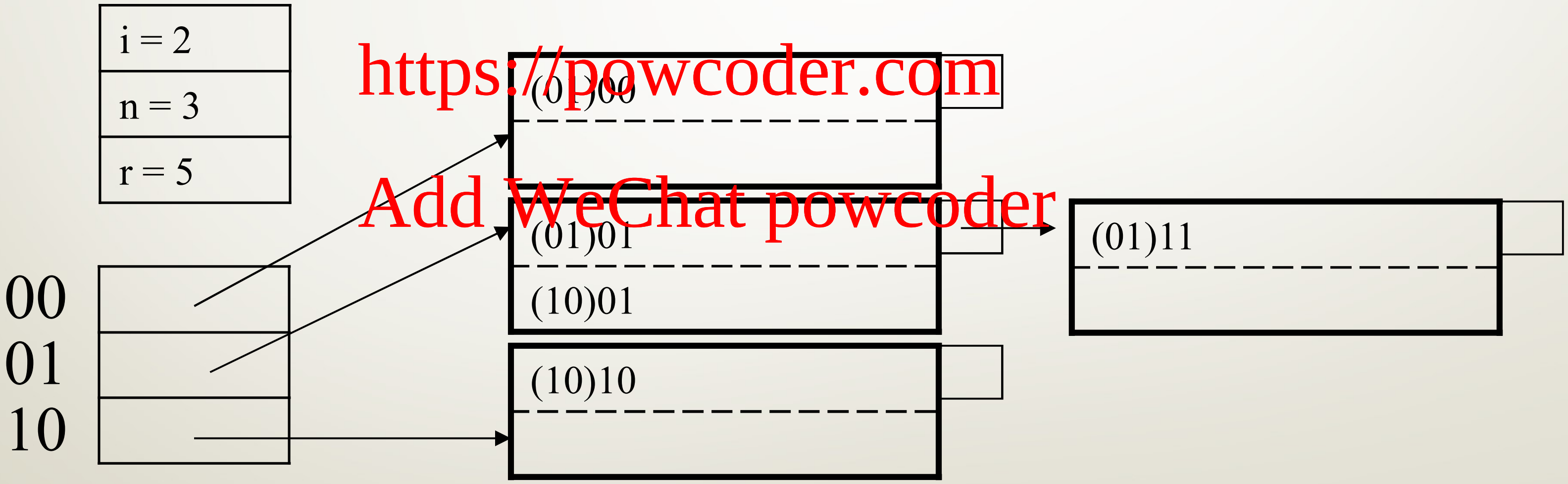
i = 2

n = 3

r = 4

Linear Hash Table Example

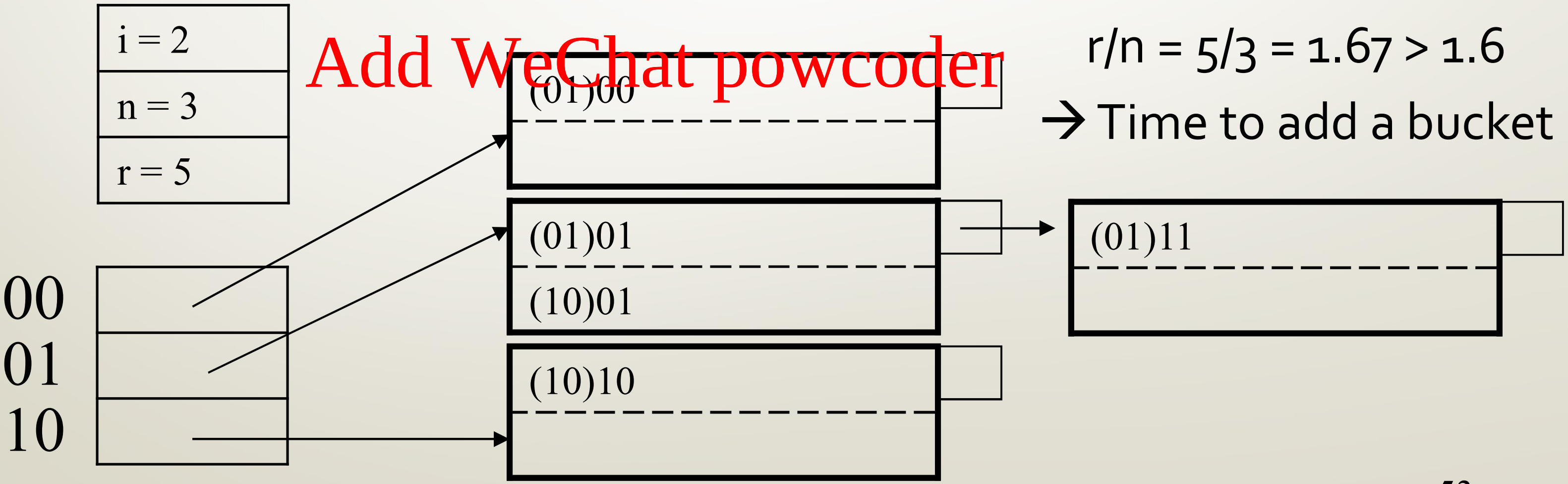
- Insert 1001: overflow blocks…



Linear Hash Tables

- Extend n → n+1 when average number of records per bucket exceeds (say) 80% of total number of records per block
  - e.g., r/n <= 0.8 * 2 = $\boxed{1}$.6 (for block size = 2)
- Until then, use overflow blocks (cheaper than adding buckets)

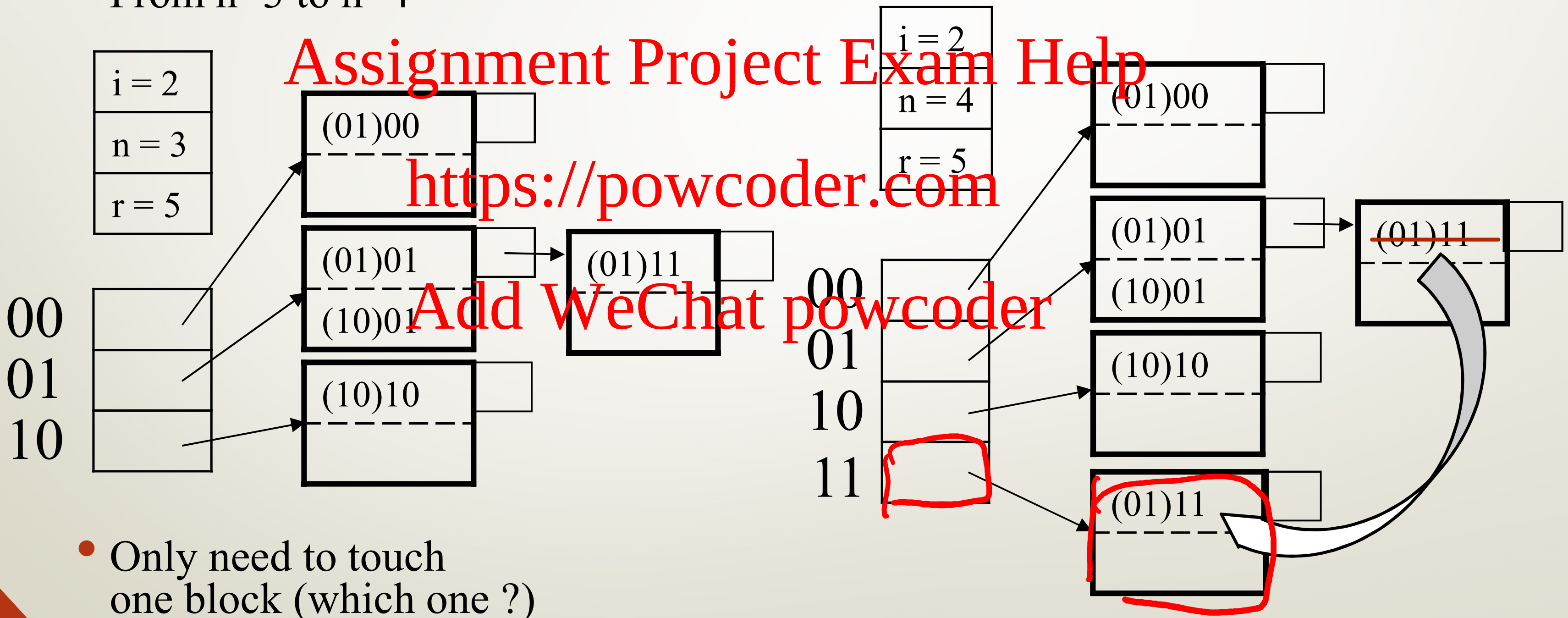
r/n = 5/3


i = 2
n = 3
r = 5
00
01
10
(01)00
(01)01
(10)01
(10)10
(01)11

r/n = 5/3 = 1.67 > 1.6
→ Time to add a bucket

Linear Hash Table Extension

- From n=3 to n=4

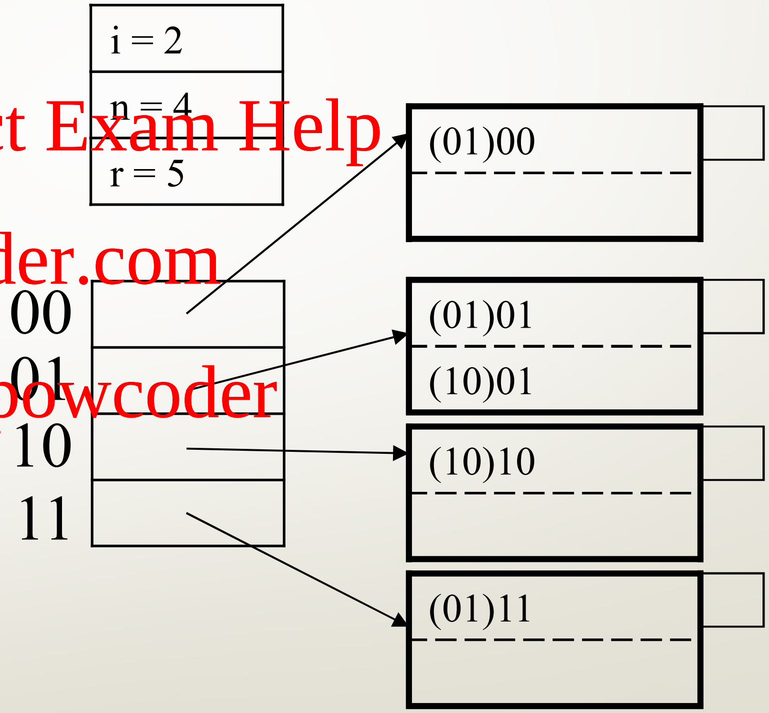


- Only need to touch one block (which one ?)

Linear Hash Table Extension

- From n=3 to n=4 finished

r/n = 5/4 = 1.25 < 1.6 ✓



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

Index 2.0

- Learn the best index from the data and queries logs
- Machine Learning to the rescue!
- Recall, an index is a function
- Machine learning are good at learning functions from data
- What's your cool idea for a better index?