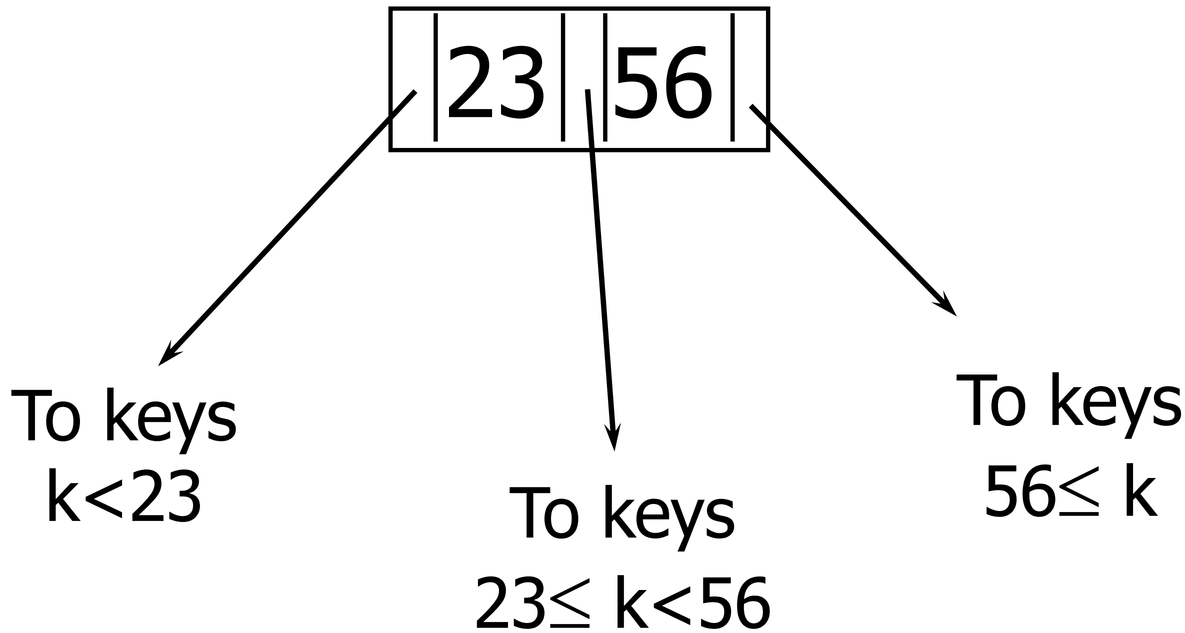
B+Tree

- Most popular index structure in RDBMS
- Advantage
  - Suitable for dynamic updates
  - Balanced
  - Minimum space usage guarantee
- Disadvantage
  - Non-sequential index blocks

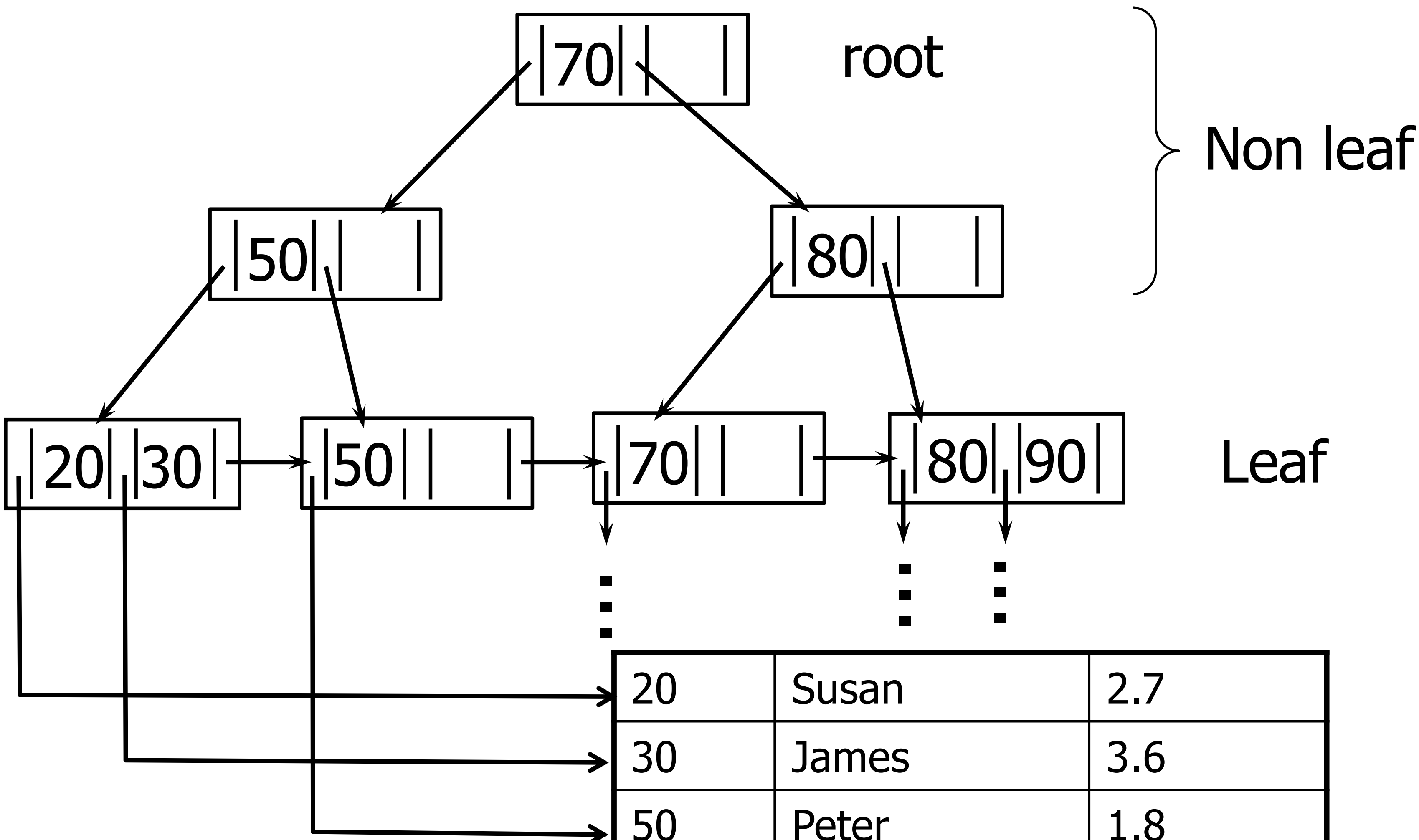
B+ tree: generalizing B trees: e.g., n=3
in practice much larger



- Points to the nodes one-level below
  - No direct pointers to tuples

- At least half of the ptrs used (precisely, $\lceil n/2 \rceil$)
  - except root, where at least 2 ptrs used

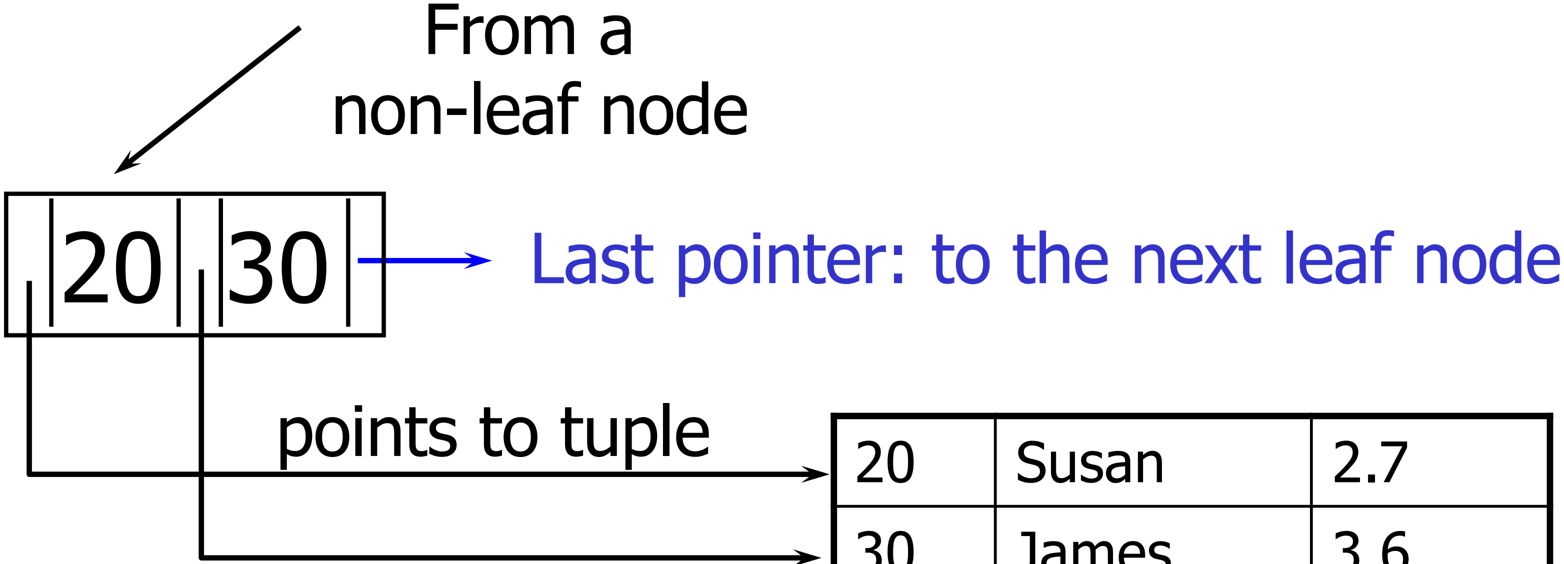
B+Tree Example (n=3)



20	Susan	2.7
30	James	3.6
50	Peter	1.8
…	…	…

Balanced: All leaf nodes are at the same level

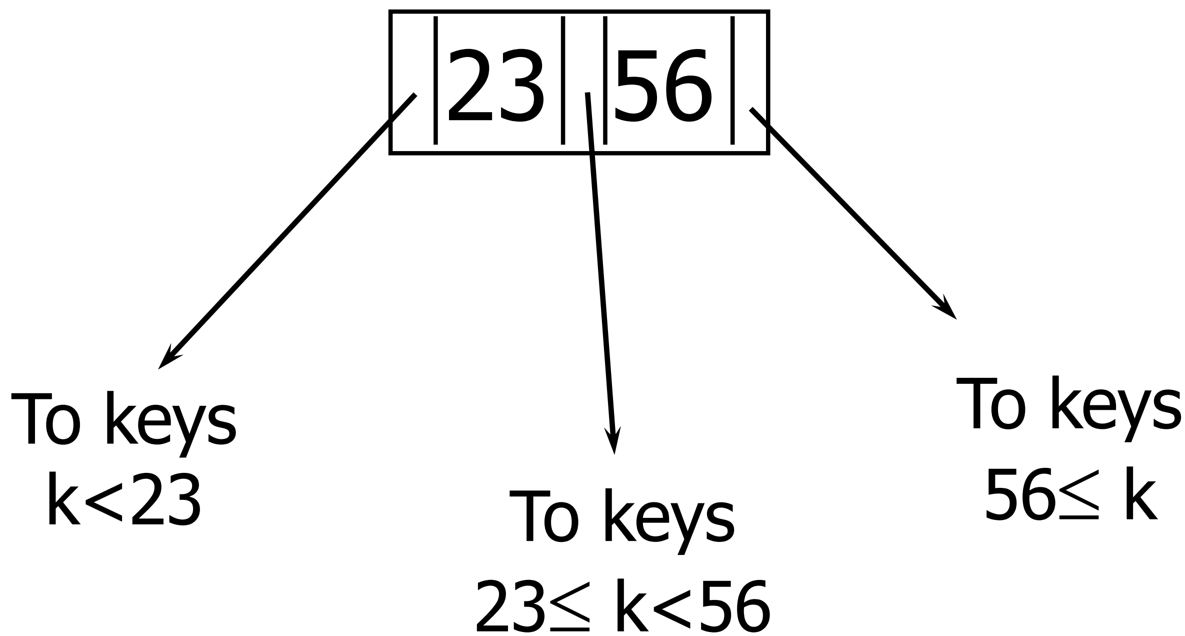
Sample Leaf Node (n=3)



20	Susan	2.7
30	James	3.6
50	Peter	1.8
…	…	…

- n: max # of pointers in a node
- All pointers (except the last one) point to tuples
- At least half of the pointers are used.
  (more precisely, $\lceil (n+1)/2 \rceil$ pointers)

Sample Non-leaf Node (n=3)

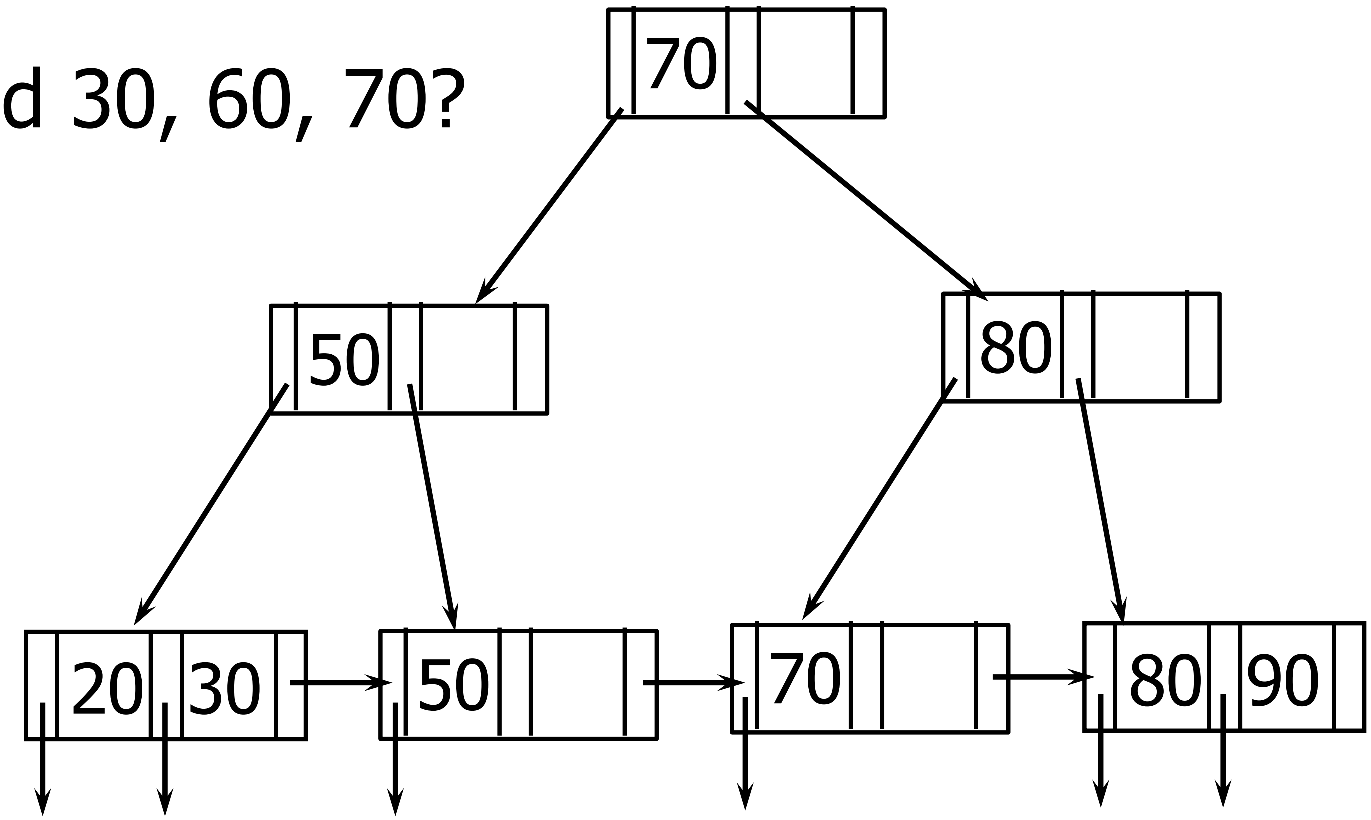


- Points to the nodes one-level below
  - No direct pointers to tuples

- At least half of the ptrs used (precisely, $\lceil n/2 \rceil$)
  - except root, where at least 2 ptrs used

Search on B+tree

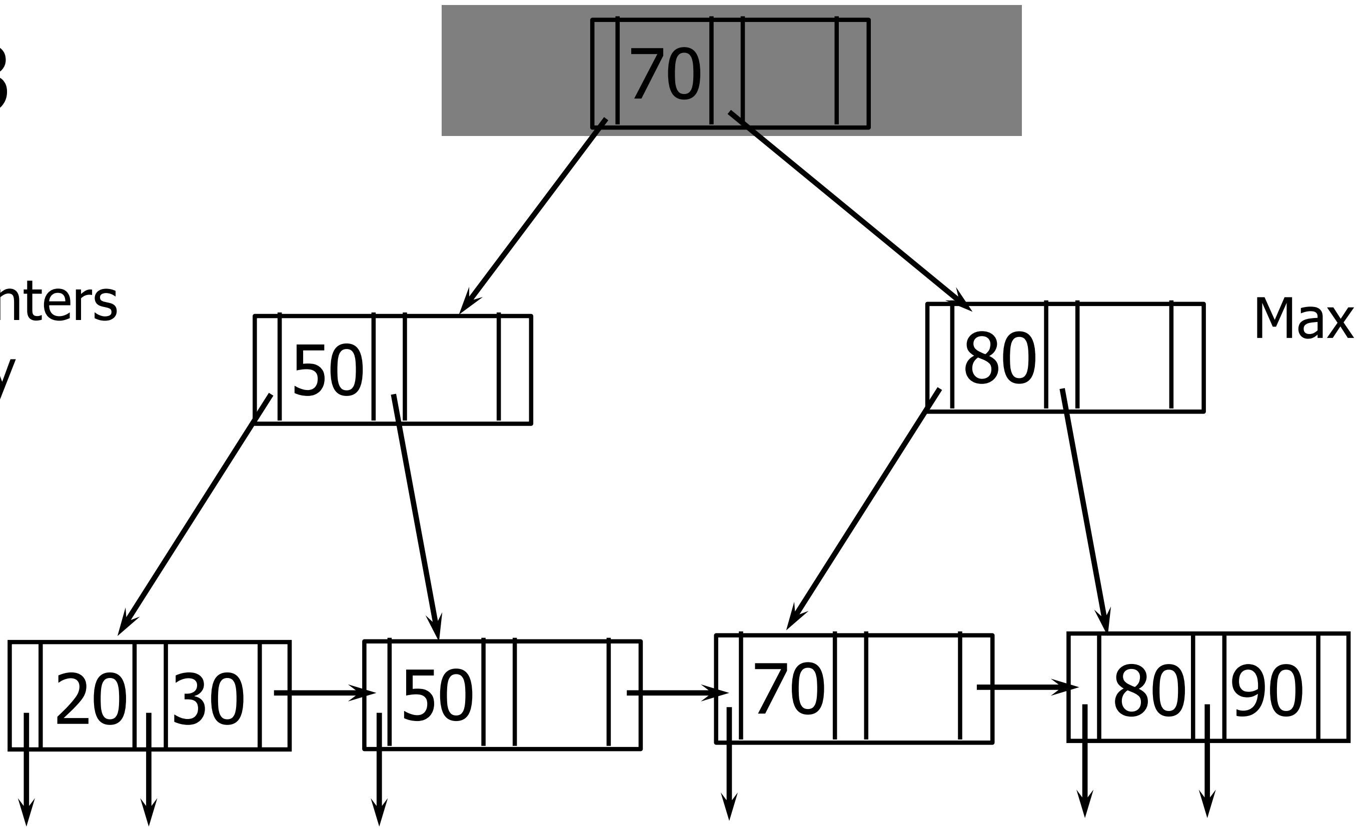
- Find 30, 60, 70?



- Find a greater key and follow the link on the left
  (Algorithm: Figure 12.10 on textbook)

Nodes but the root are never too empty

n=3



min 2 pointers
1 key

Max 3 pointers
2 keys

min 2 pointers
1 key

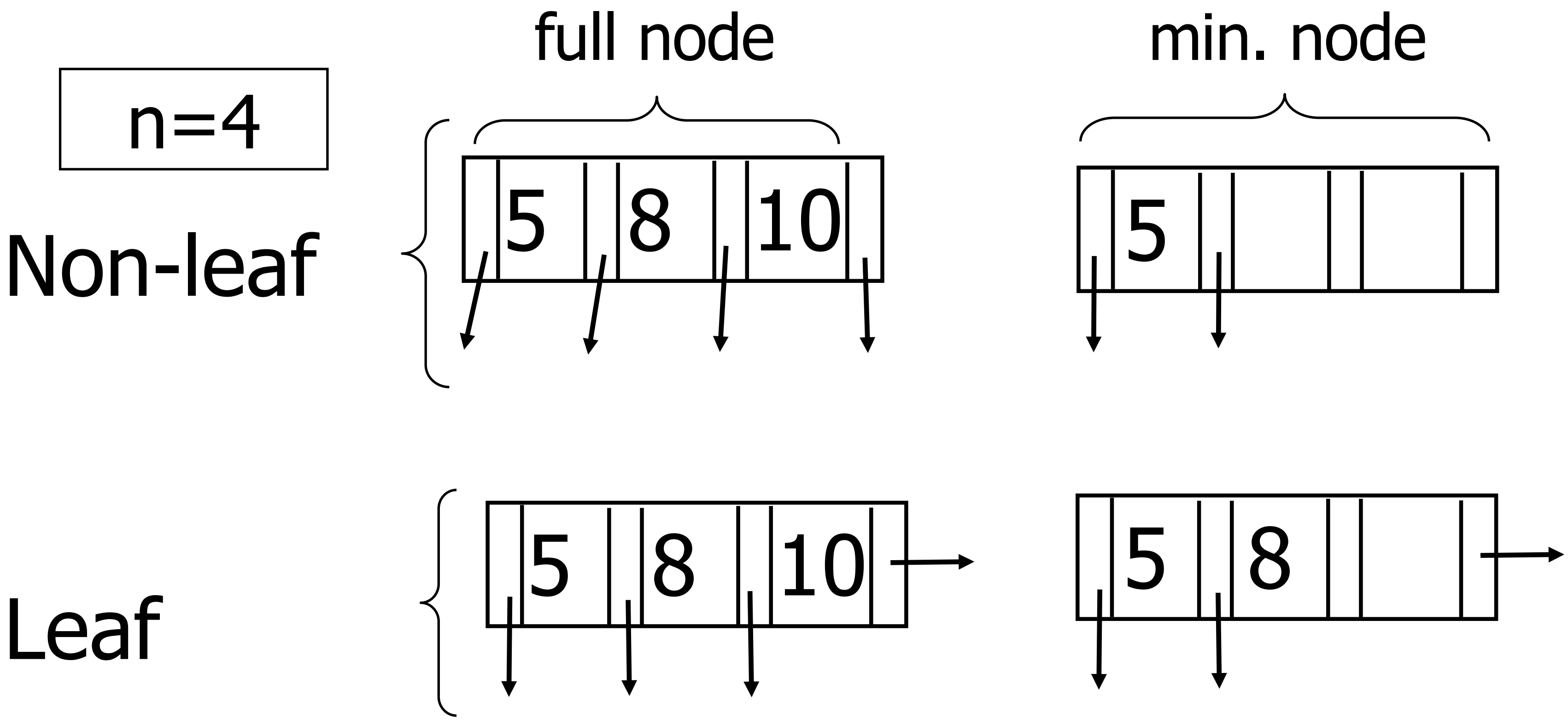
Max 3 pointers
2 keys

Nodes are never too empty

- Use at least

  Non-leaf: $\lceil n/2 \rceil$ pointers

  Leaf: $\lceil (n+1)/2 \rceil$ pointers



Number of Ptrs/Keys for B+tree

	Max Ptrs	Max keys	Min ptrs	Min keys
Non-leaf (non-root)	n	n-1	$\lceil n/2 \rceil$	$\lceil n/2 \rceil$-1
Leaf (non-root)	n	n-1	$\lceil (n+1)/2 \rceil$	$\lceil (n-1)/2 \rceil$
Root	n	n-1	2	1

B+Tree Insertion

(a) simple case (no overflow)

(b) leaf overflow

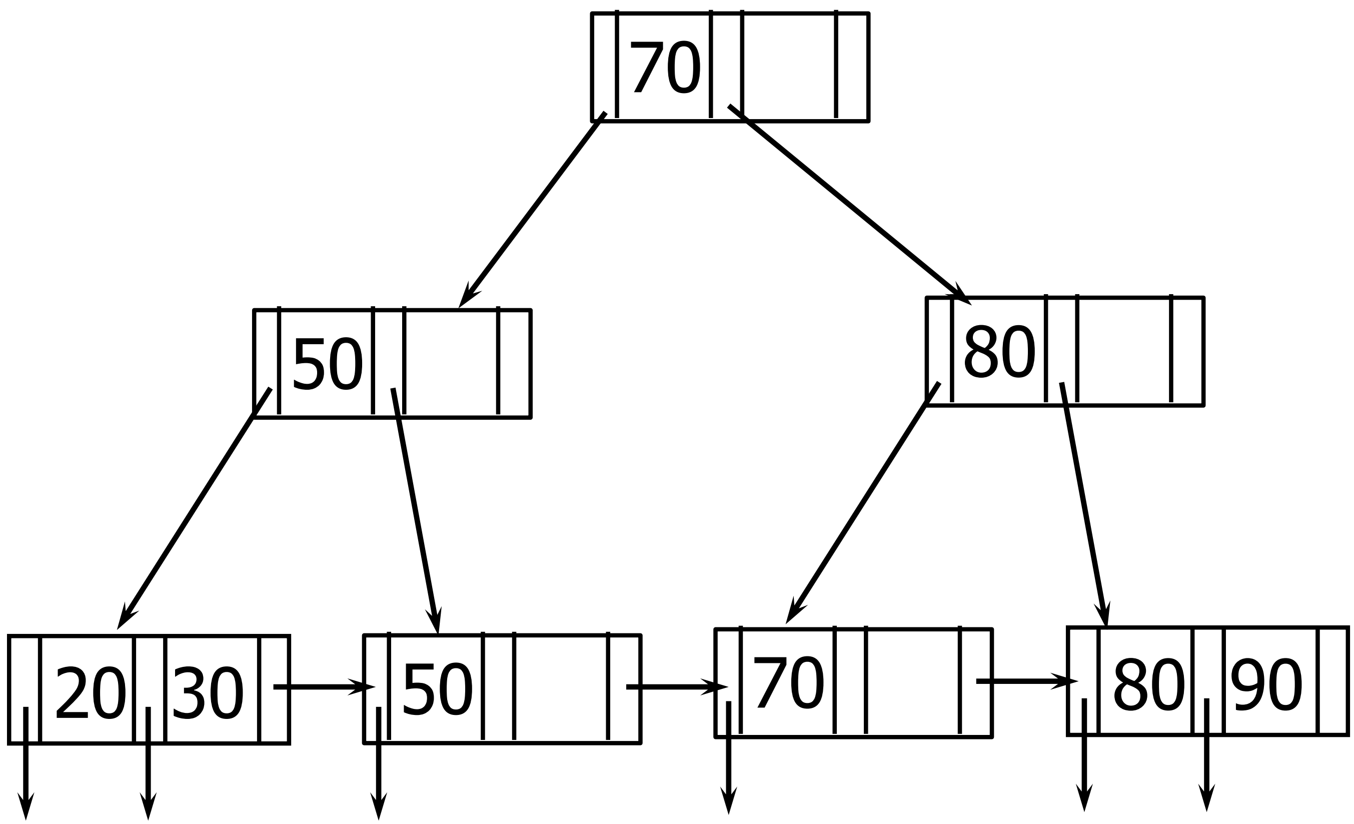
(c) non-leaf overflow

(d) new root

(a) Simple case (no overflow)

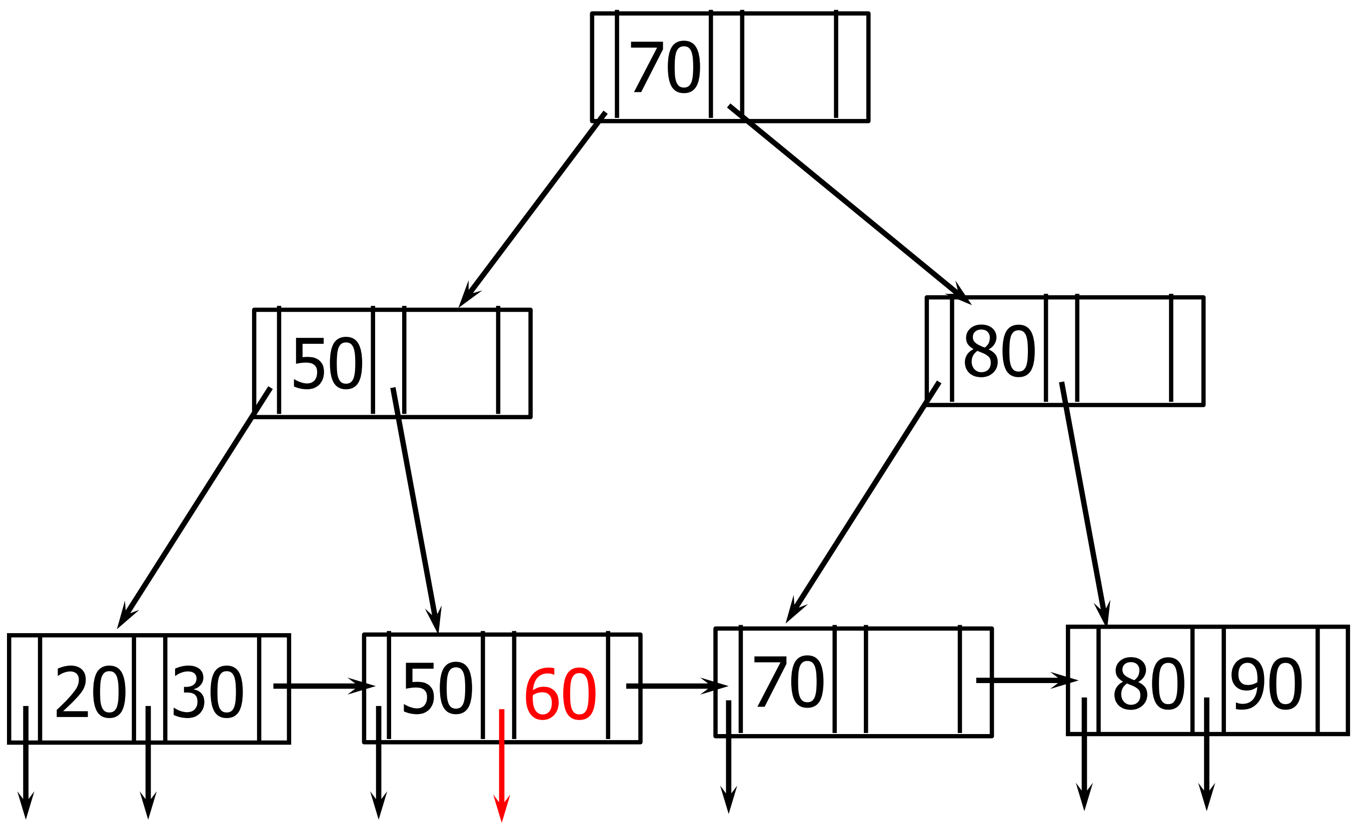
Insertion (Simple Case)

- Insert 60



Insertion (Simple Case)

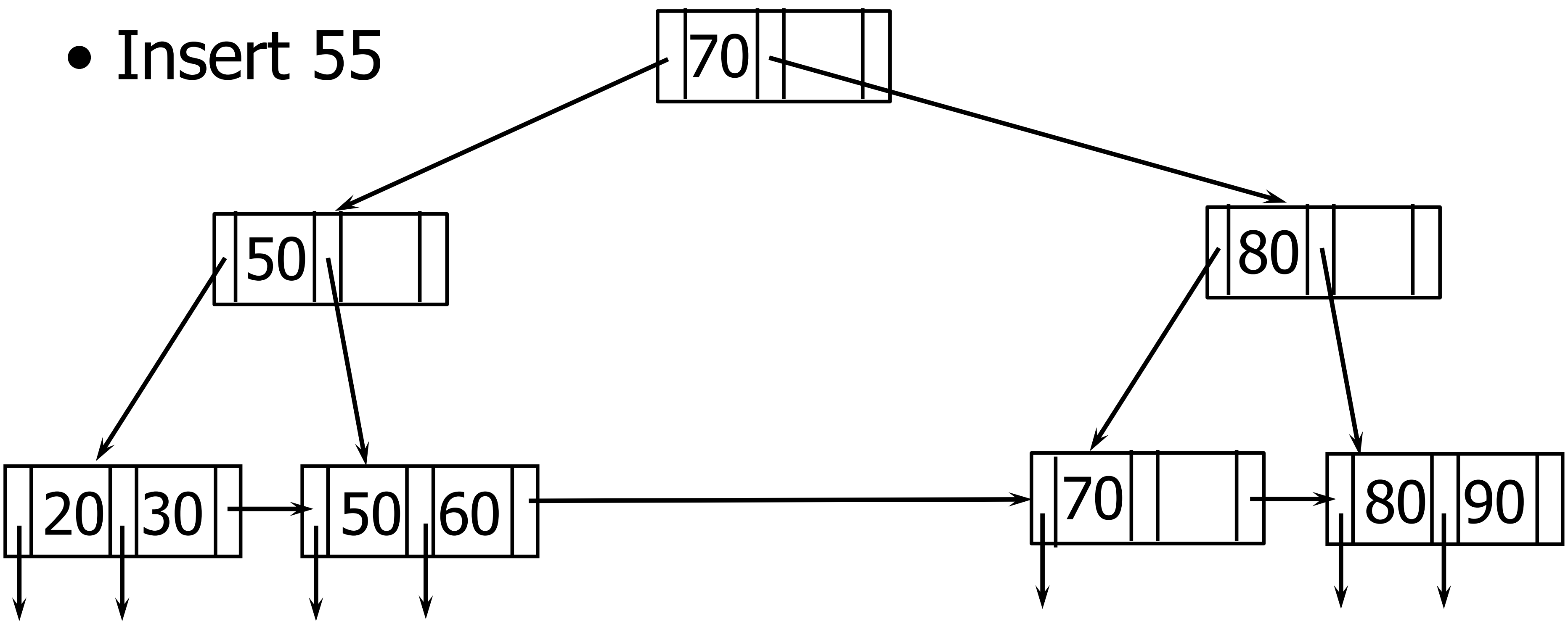
- Insert 60



(b) Leaf overflow

Insertion (Leaf Overflow)

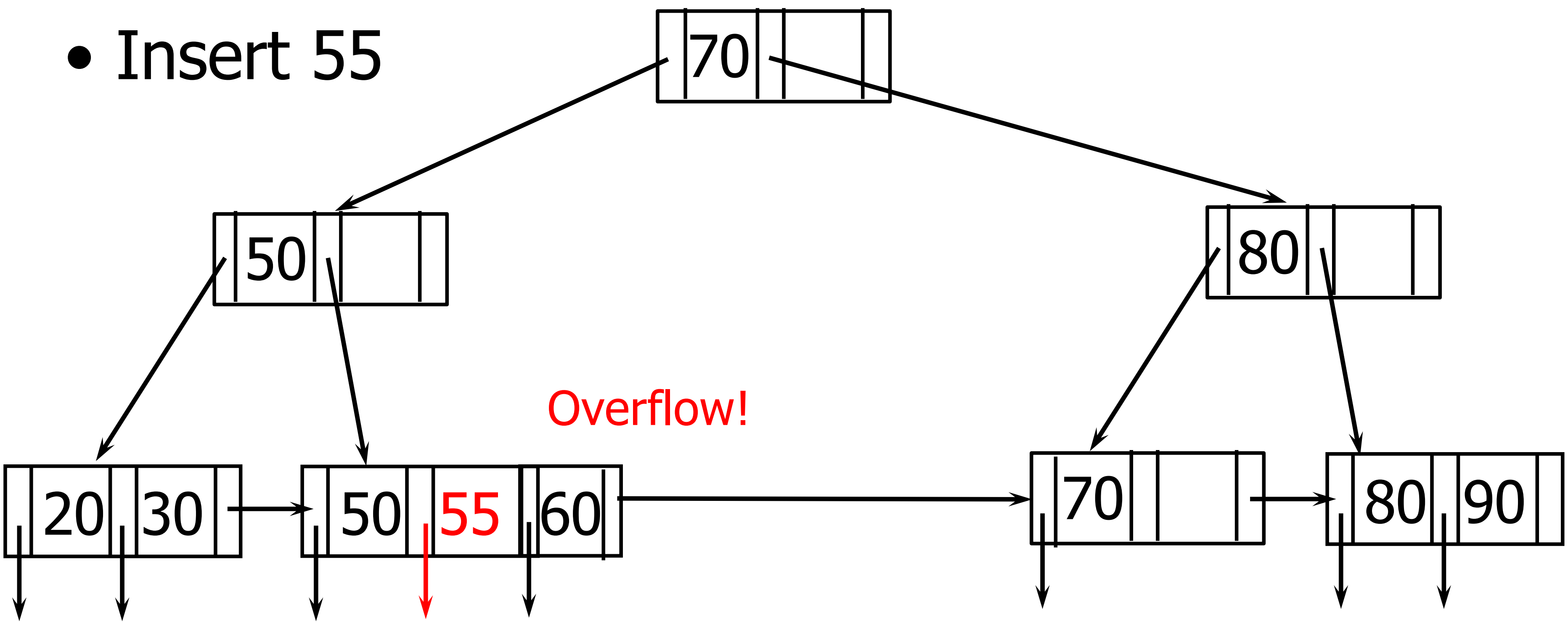
- Insert 55



- No space to store 55

Insertion (Leaf Overflow)

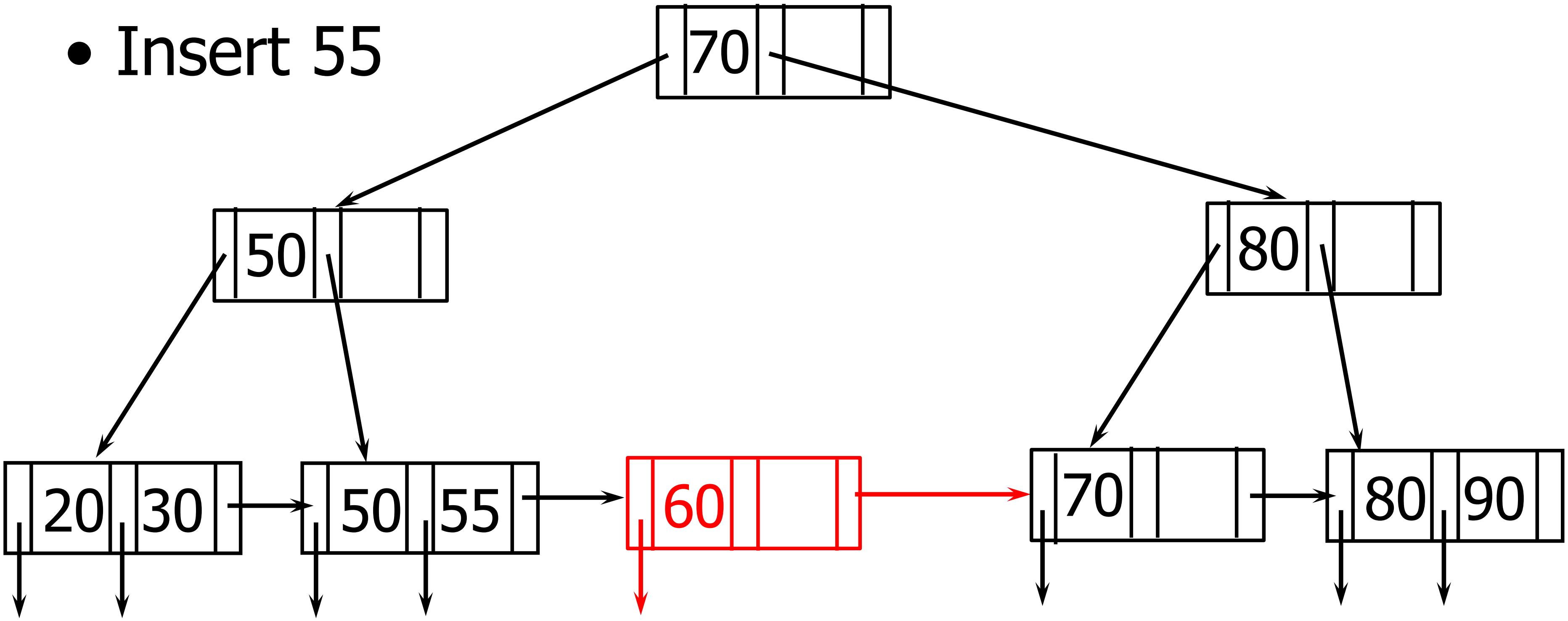
- Insert 55



- Split the leaf into two. Put the keys half and half

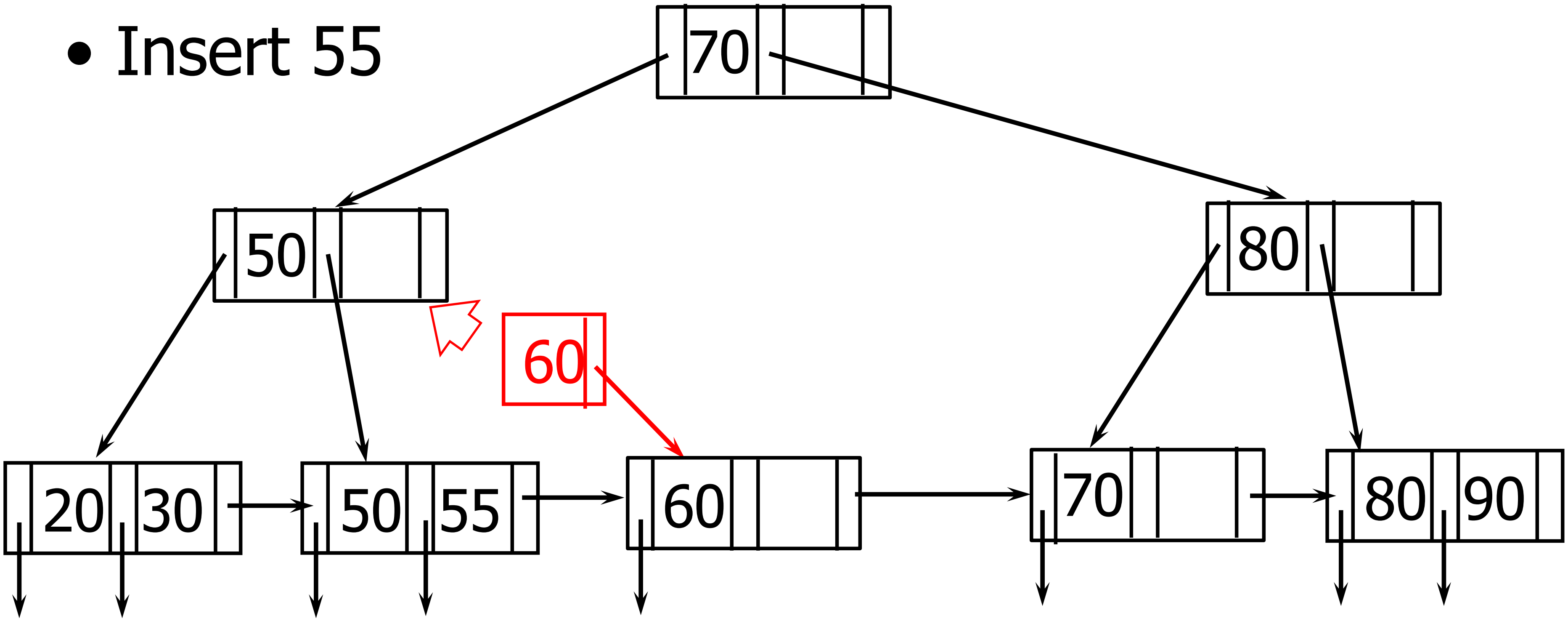
Insertion (Leaf Overflow)

- Insert 55



Insertion (Leaf Overflow)

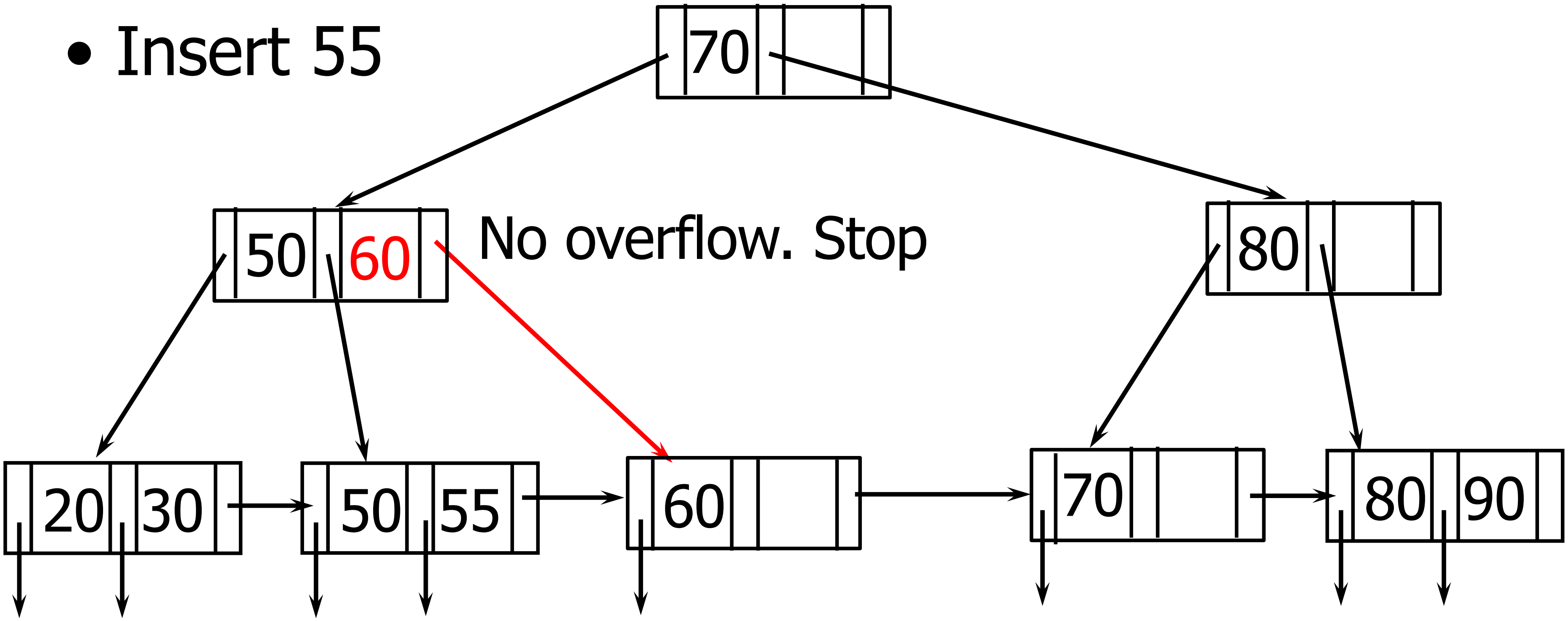
- Insert 55



- *Copy* the first key of the new node to parent

Insertion (Leaf Overflow)

- Insert 55

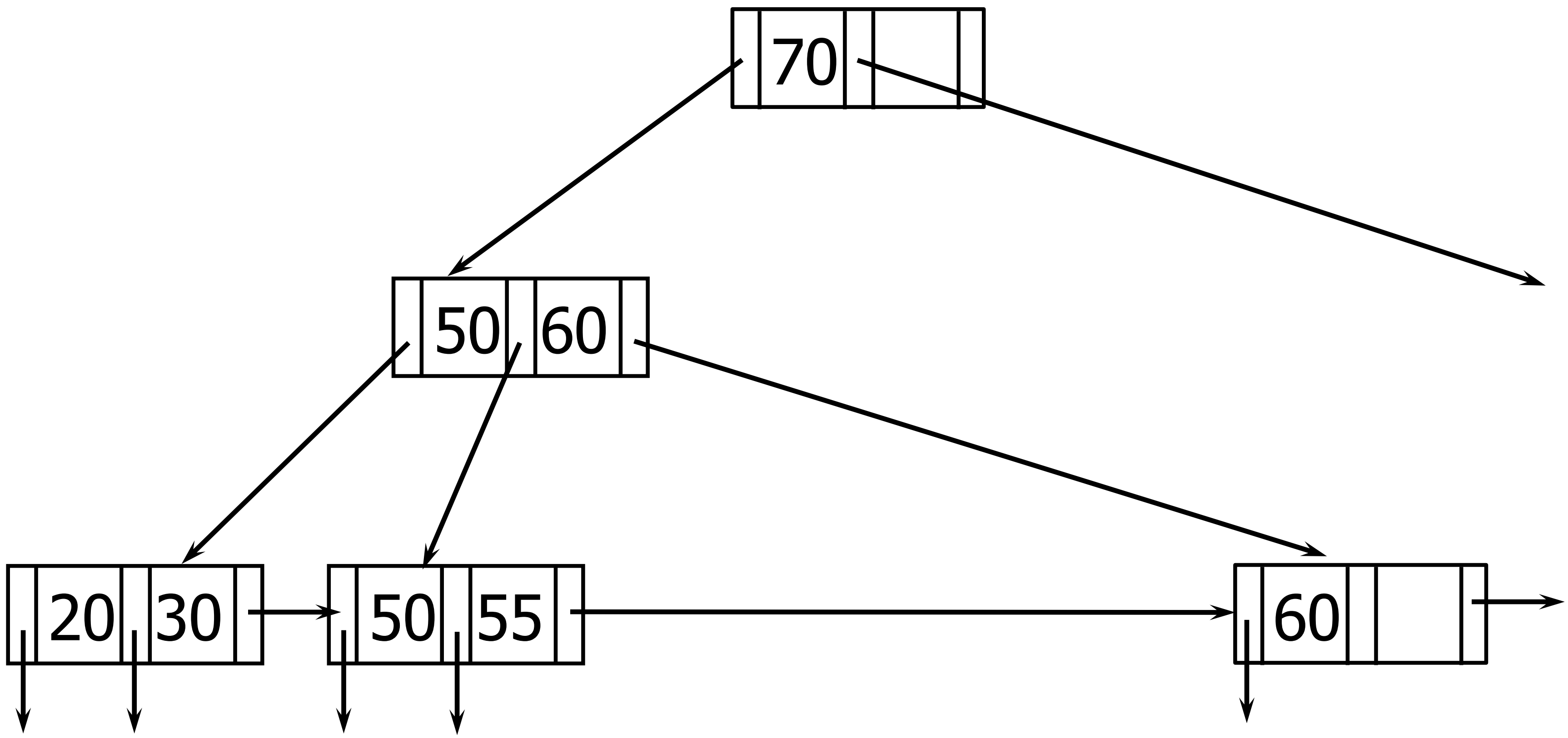


- Q: After split, leaf nodes always half full?

(c) Non-leaf overflow

Insertion (Non-leaf Overflow)

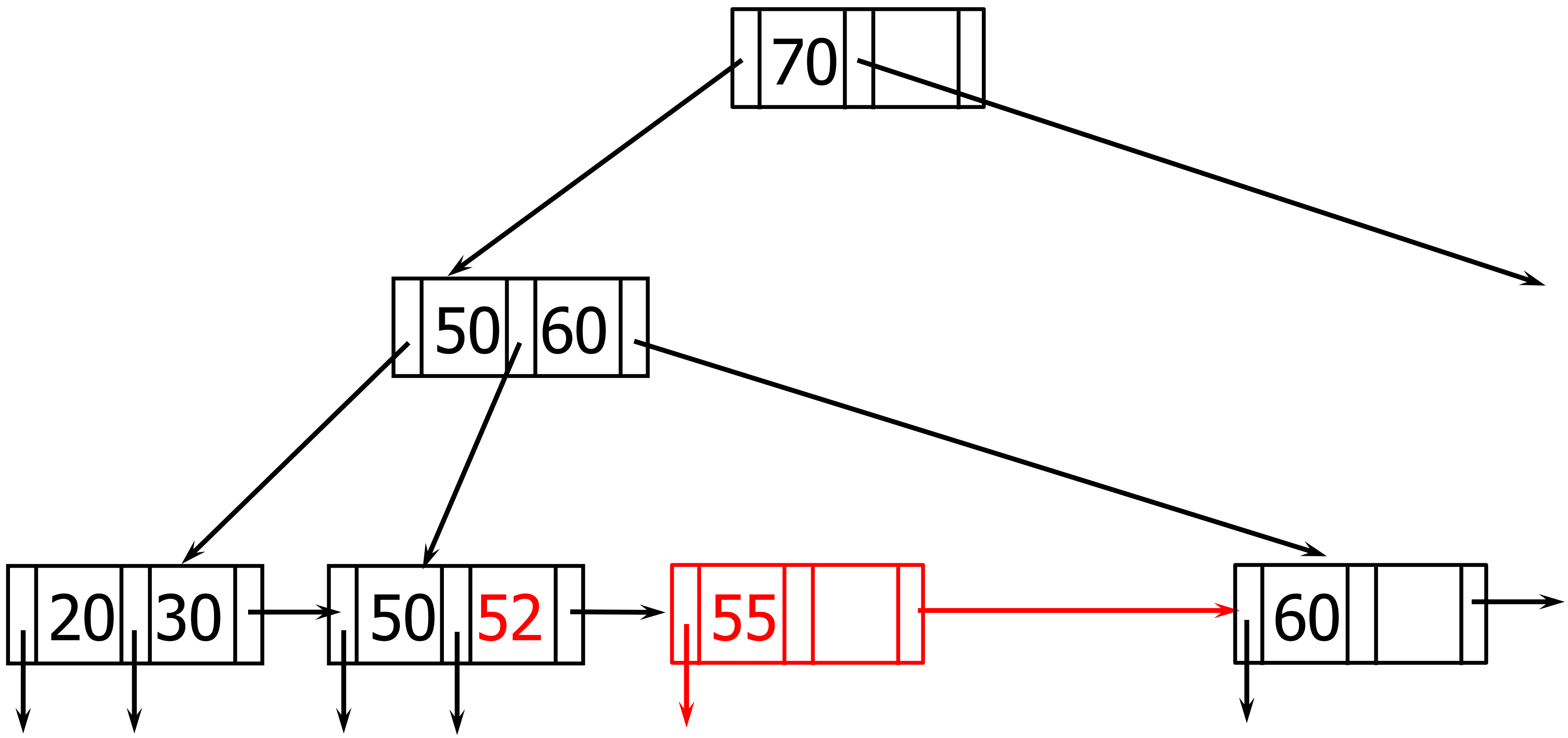
- Insert 52



Leaf overflow. Split and copy the first key of the new node

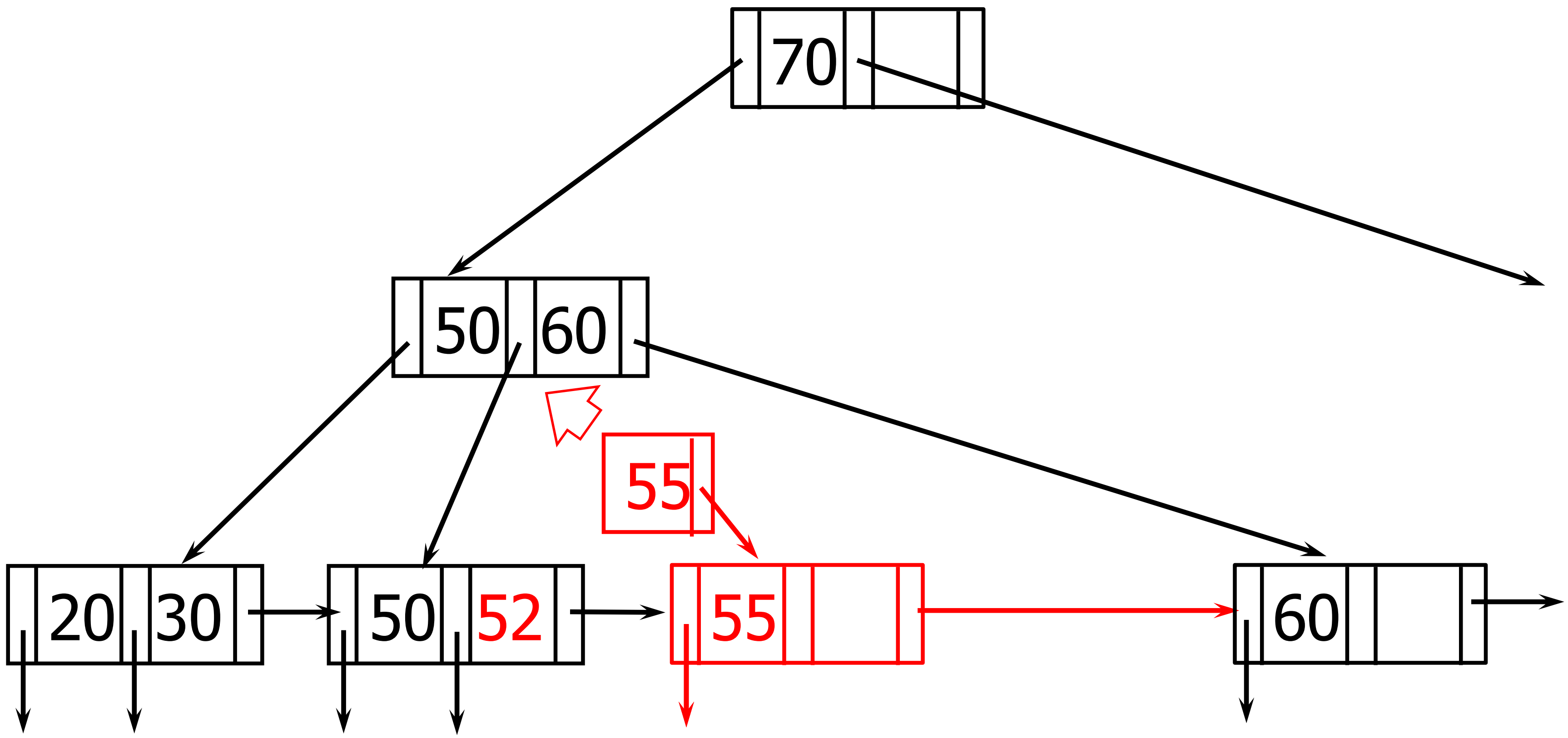
Insertion (Non-leaf Overflow)

- Insert 52



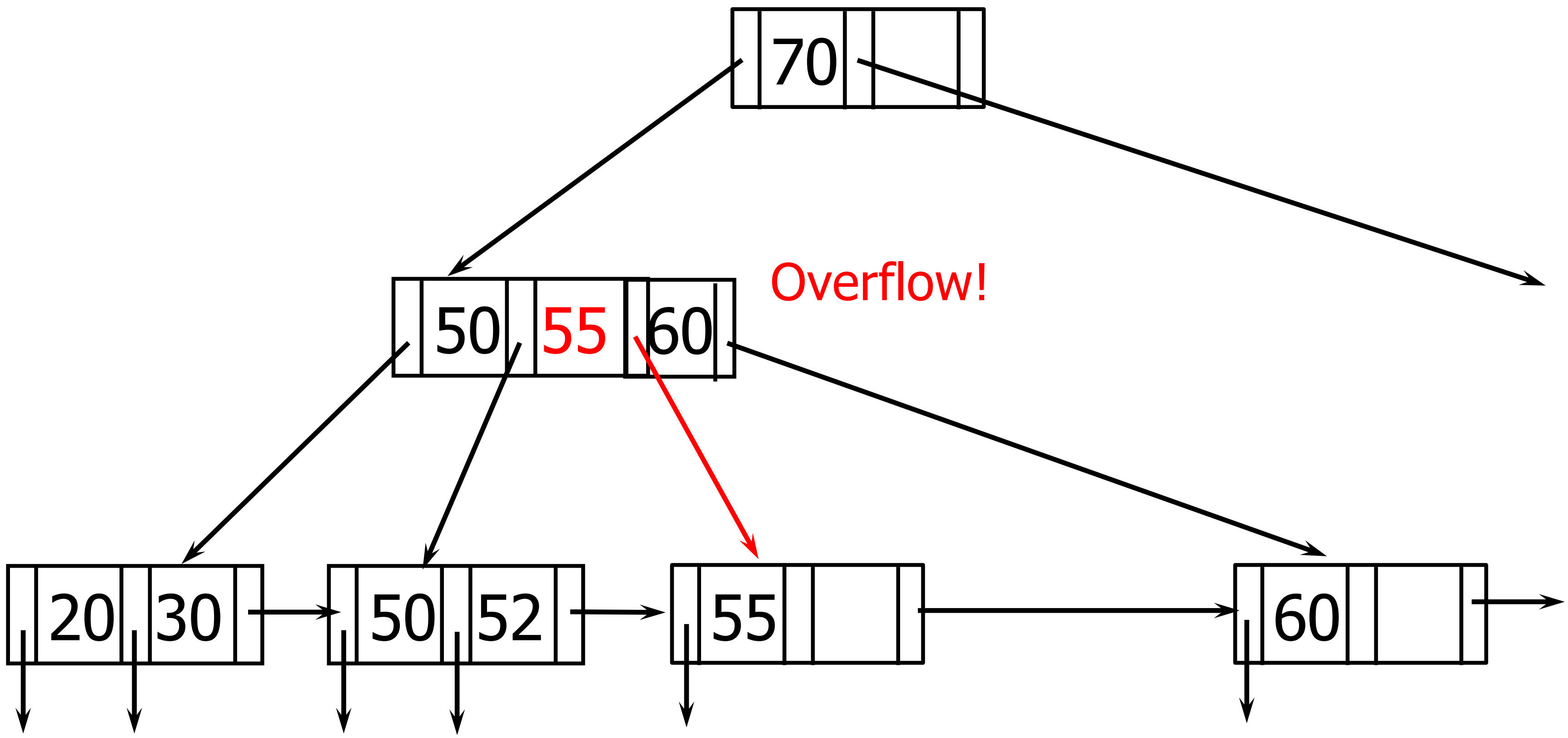
Insertion (Non-leaf Overflow)

- Insert 52



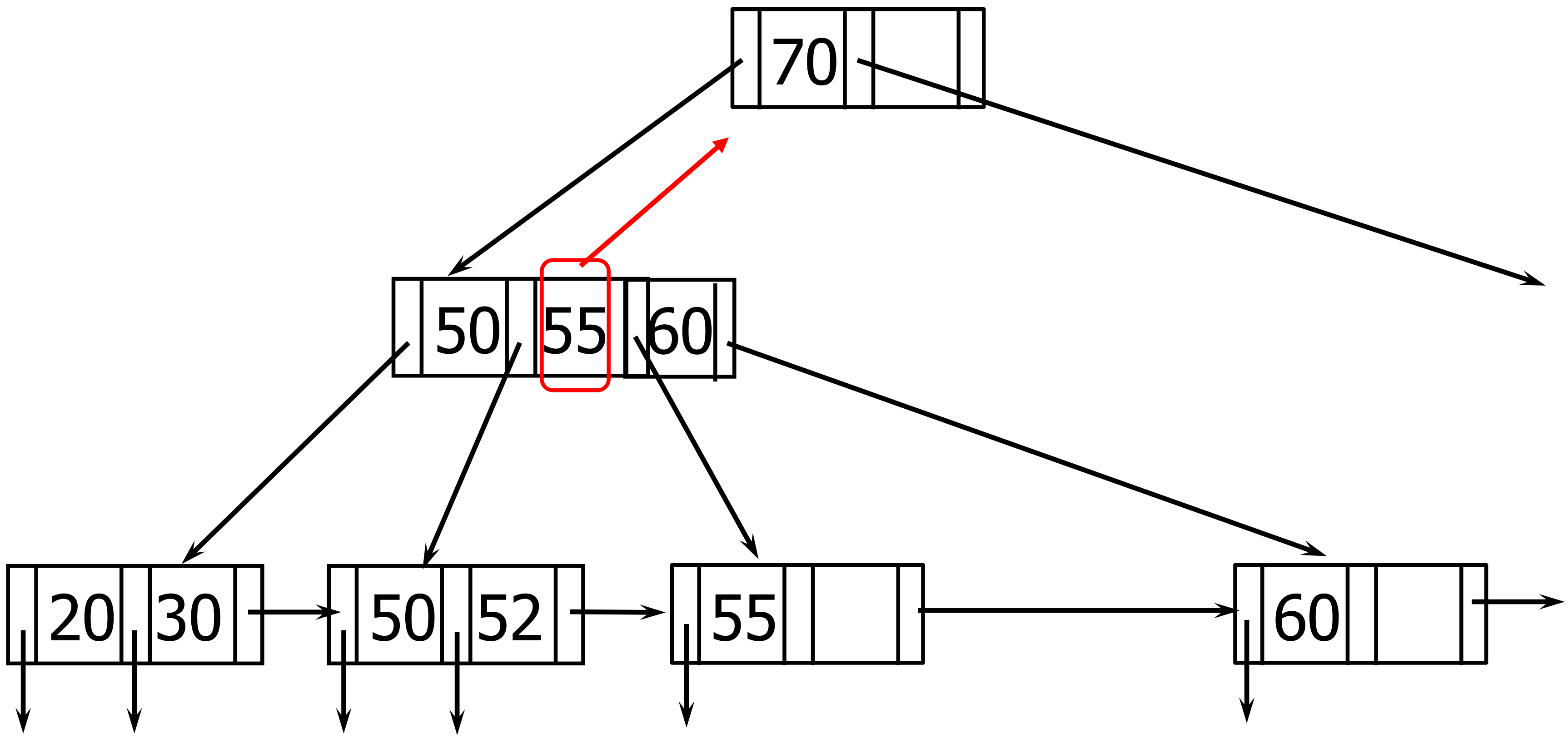
Insertion (Non-leaf Overflow)

- Insert 52



Insertion (Non-leaf Overflow)

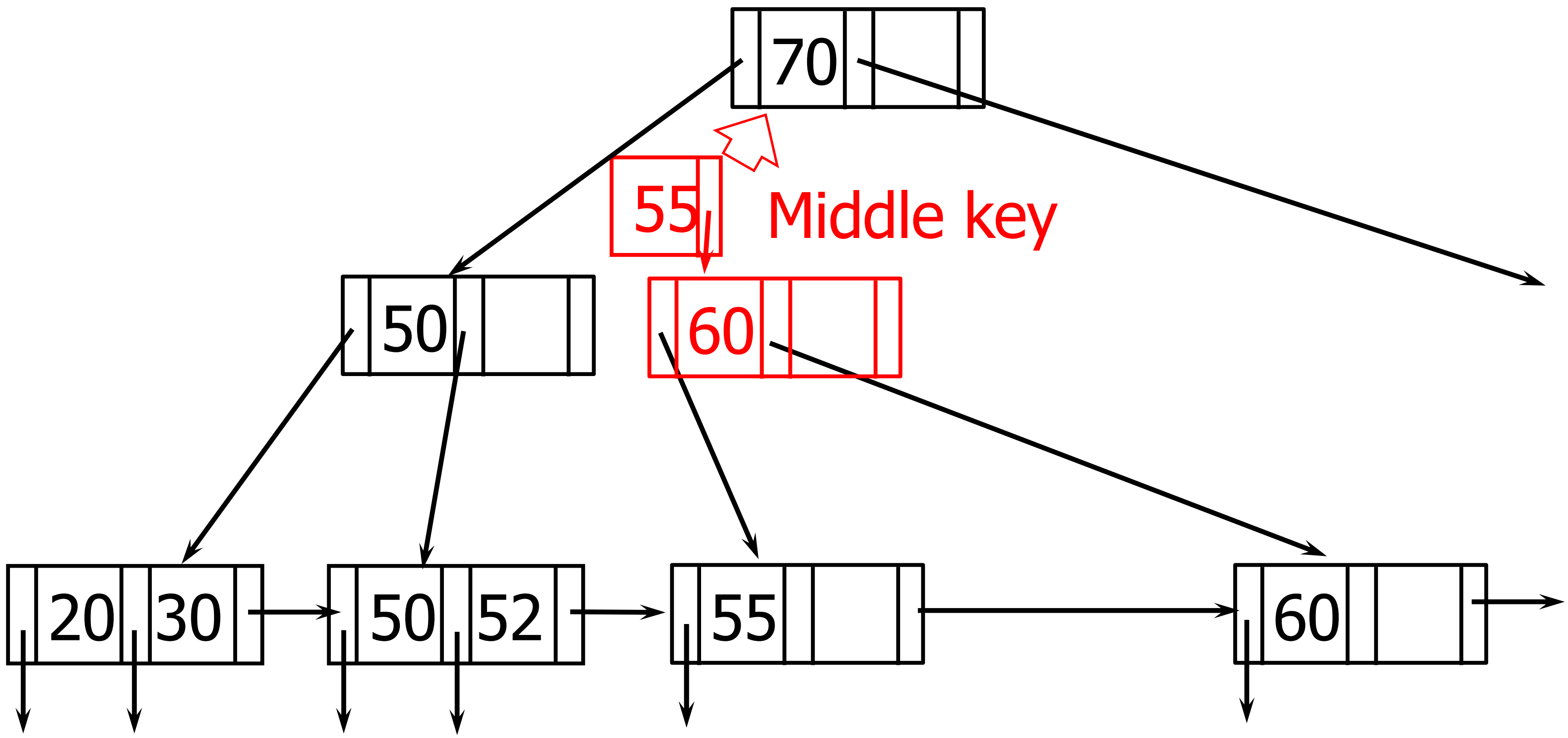
- Insert 52



Split the node into two. *Move* up the key in the middle.

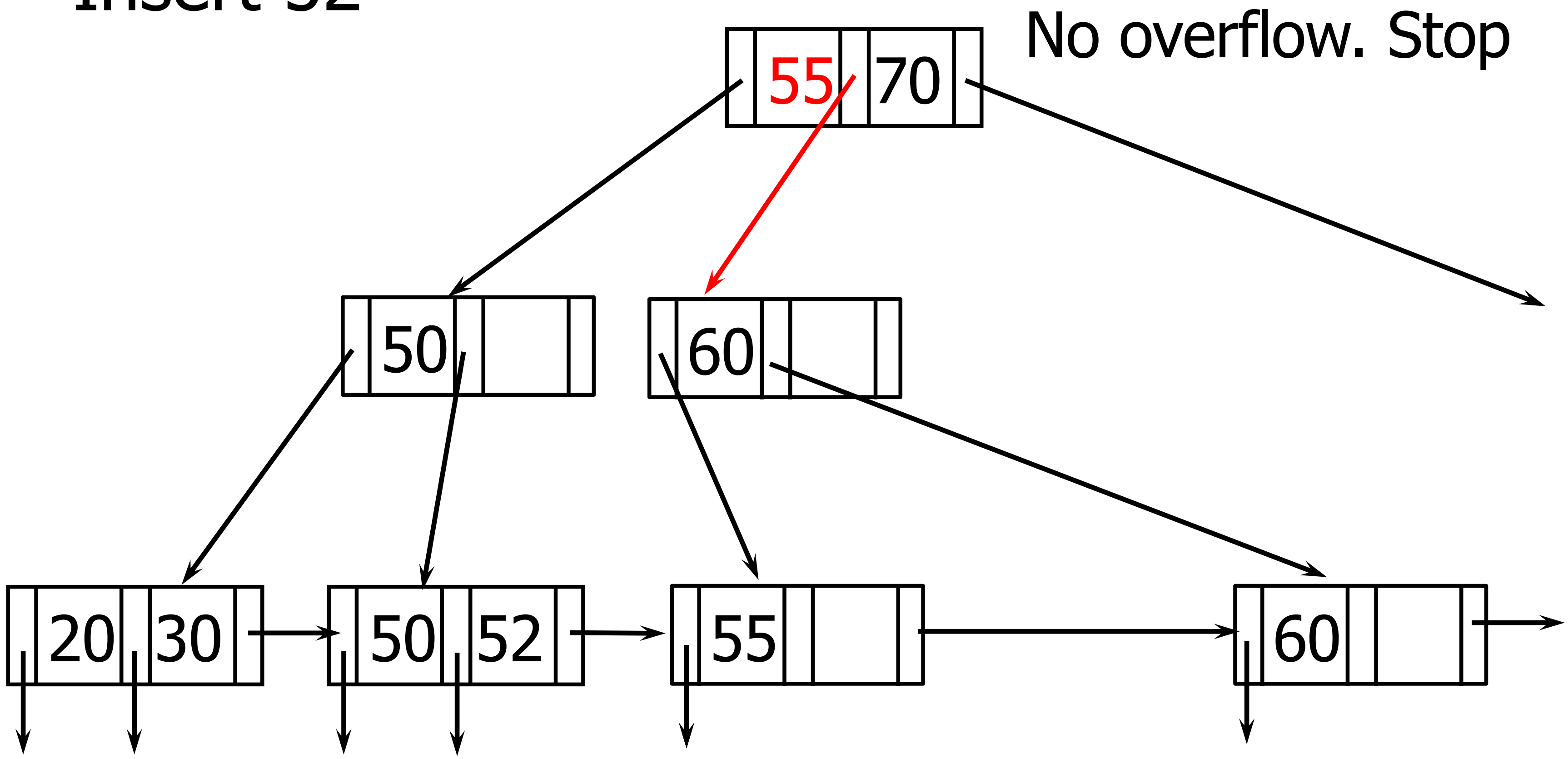
Insertion (Non-leaf Overflow)

- Insert 52



Insertion (Non-leaf Overflow)

- Insert 52

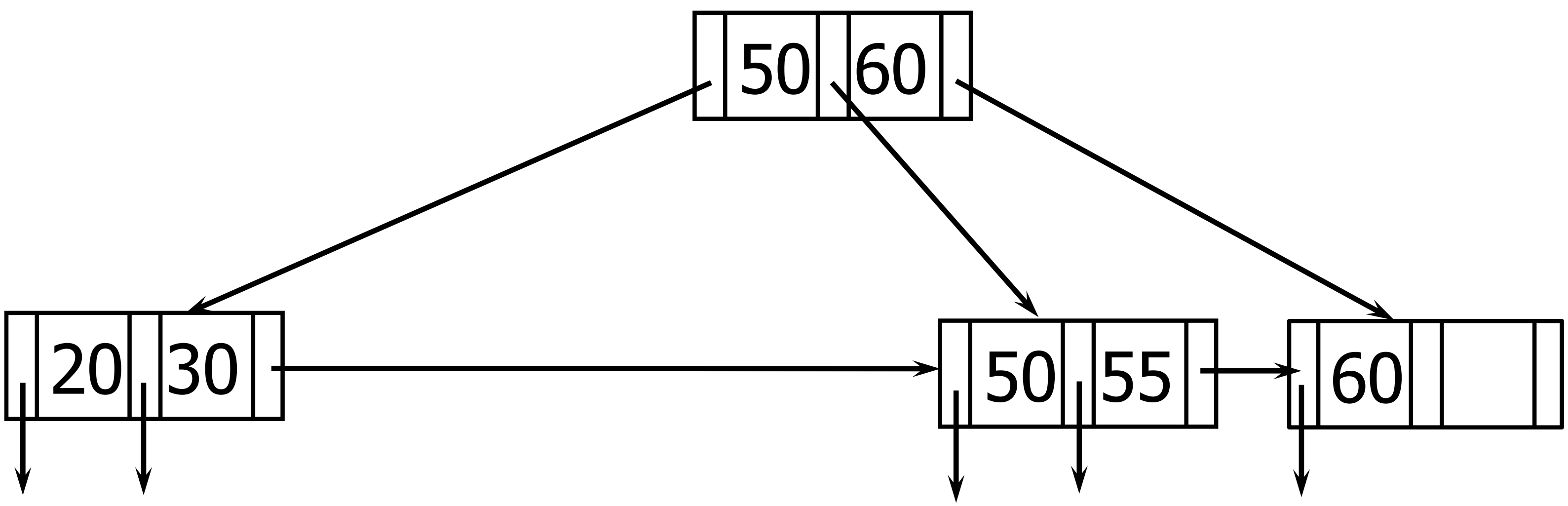


Q: After split, non-leaf at least half full?

(d) New root

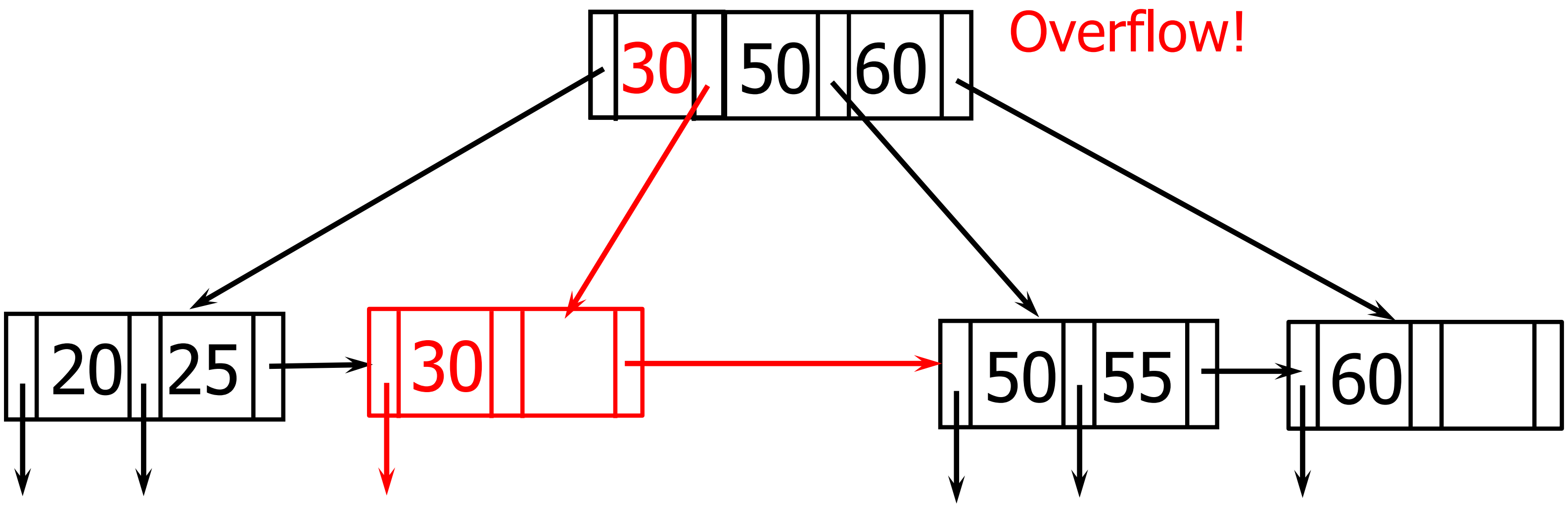
Insertion (New Root Node)

- Insert 25



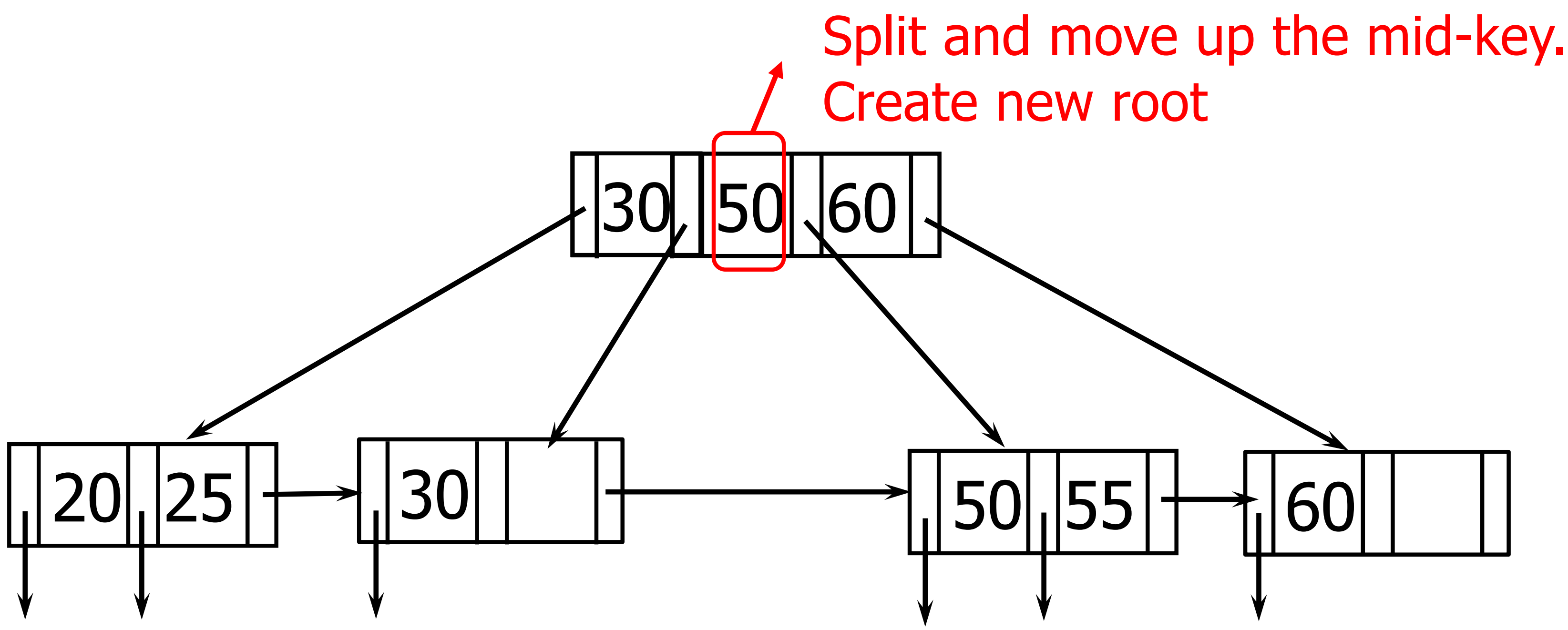
Insertion (New Root Node)

- Insert 25



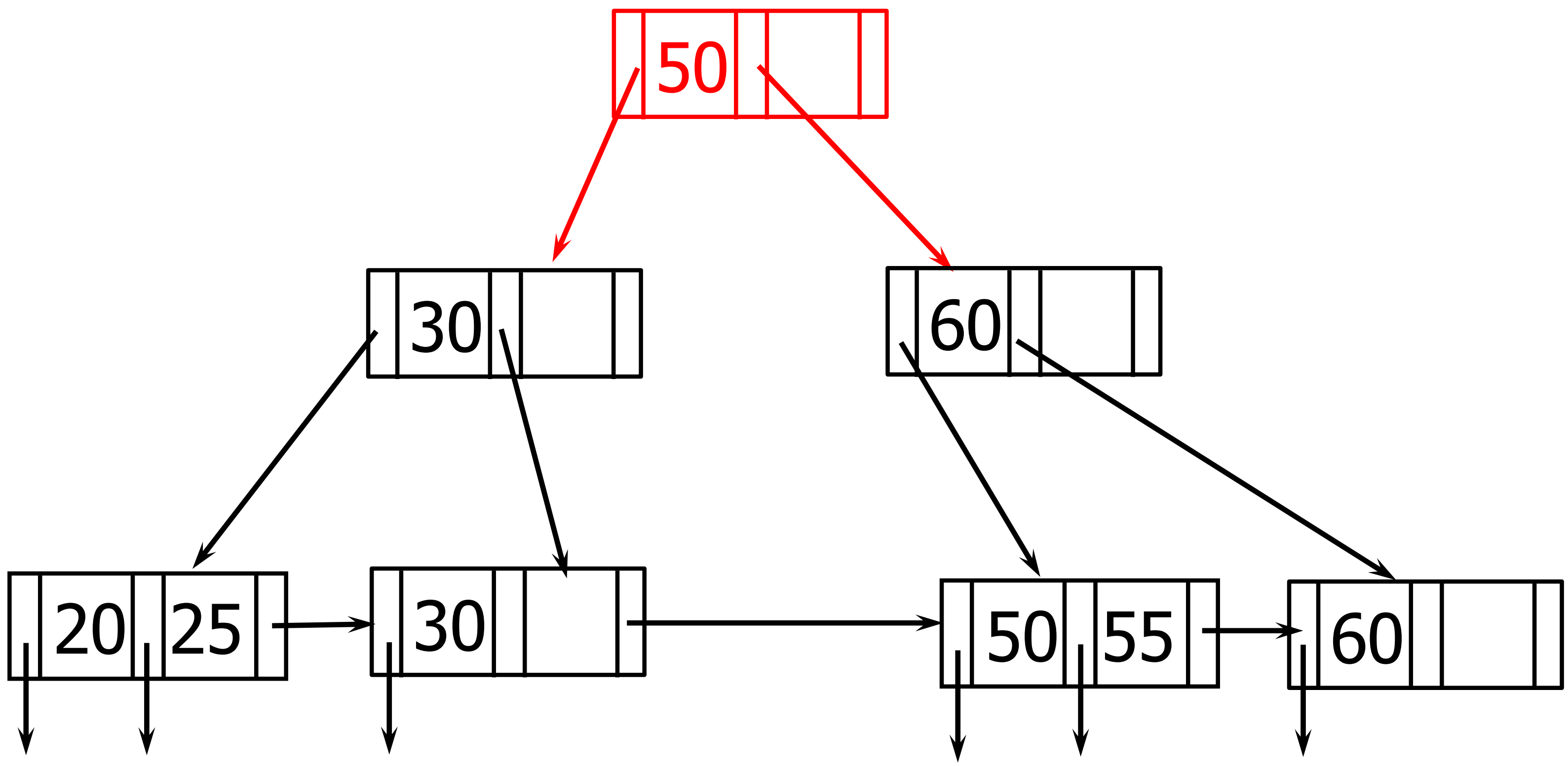
Insertion (New Root Node)

- Insert 25



Insertion (New Root Node)

- Insert 25
- Q: At least 2 ptrs at root?



B+Tree Insertion

- Leaf node overflow
  - The first key of the new node is *copied* to the parent
- Non-leaf node overflow
  - The middle key is *moved* to the parent
- Detailed algorithm: see textbook

B+Tree Deletion

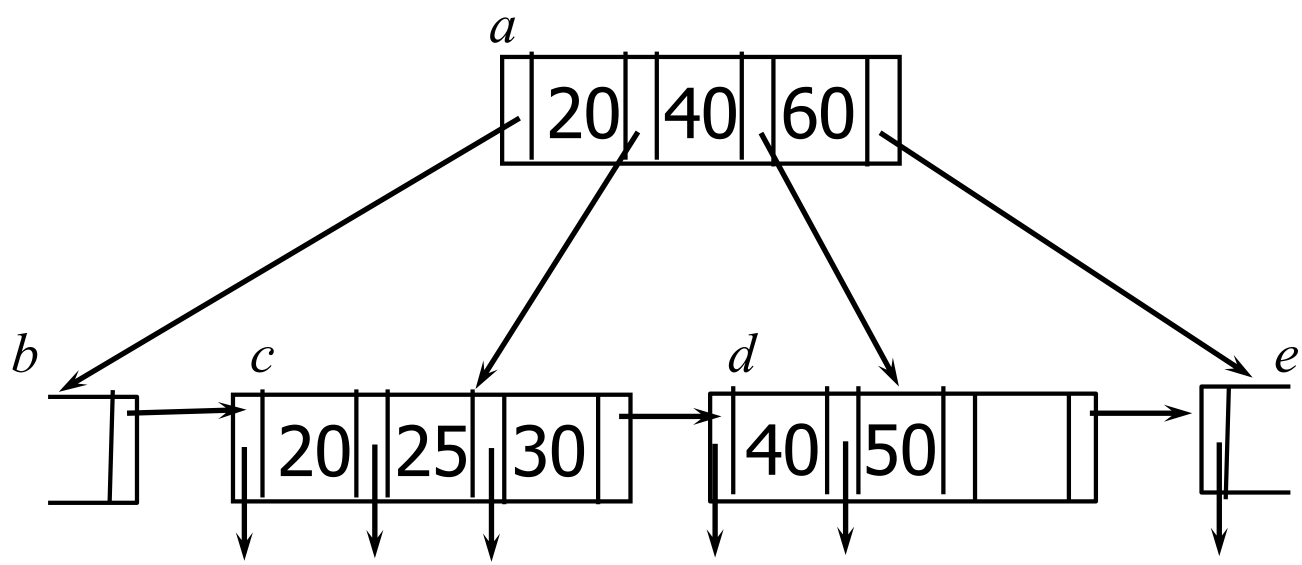
(a) Simple case (no underflow)
(b) Leaf node, coalesce with neighbor
(c) Leaf node, redistribute with neighbor
(d) Non-leaf node, coalesce with neighbor
(e) Non-leaf node, redistribute with neighbor

In the examples, n = 4

- Underflow for non-leaf when fewer than $\lceil n/2 \rceil = 2$ ptrs
- Underflow for leaf when fewer than $\lceil (n+1)/2 \rceil = 3$ ptrs
- Nodes are labeled as *a, b, c, d, …*

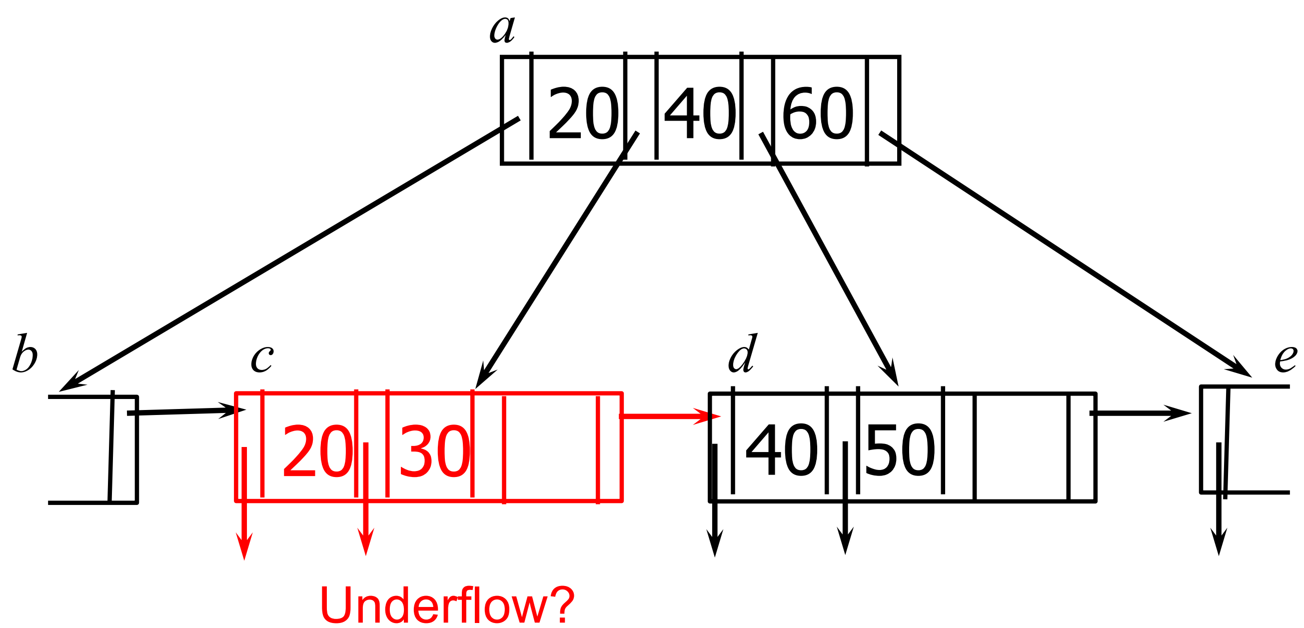
(a) Simple case
(no underflow)

(a) Simple case



- Delete 25

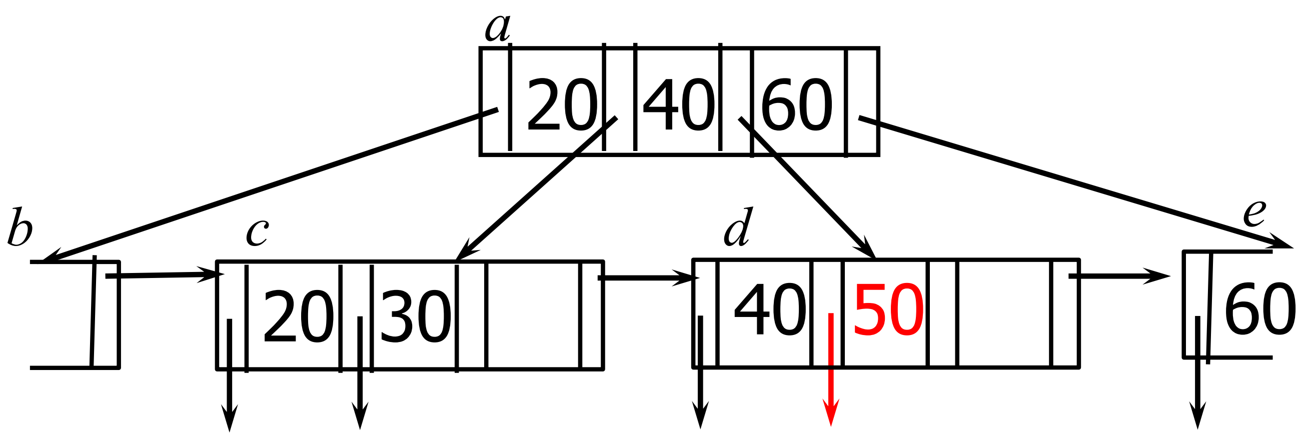
(a) Simple case



- Delete 25
  - Underflow? Min 3 ptrs. Currently 3 ptrs

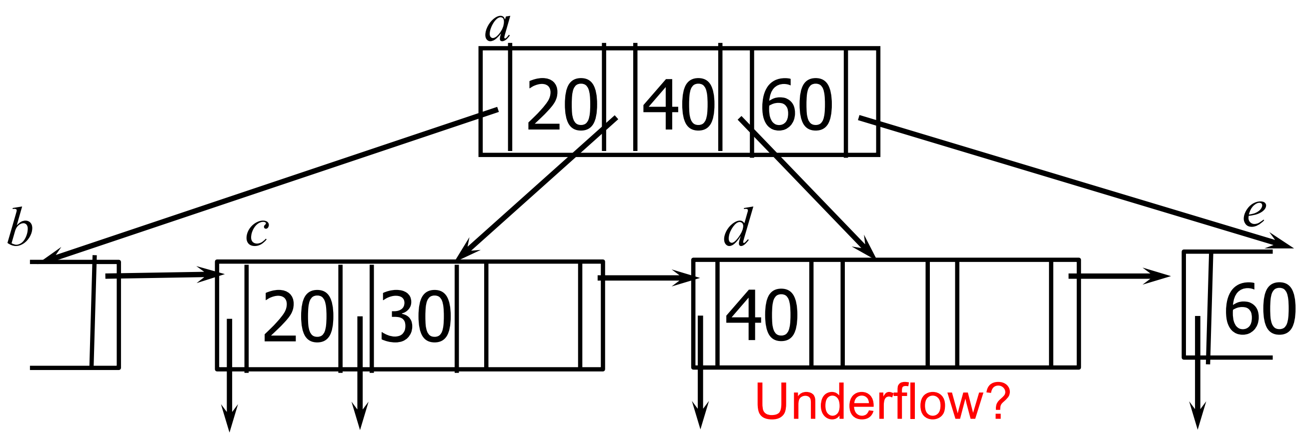
(b) Leaf node, coalesce with neighbor

(b) Coalesce with sibling (leaf)



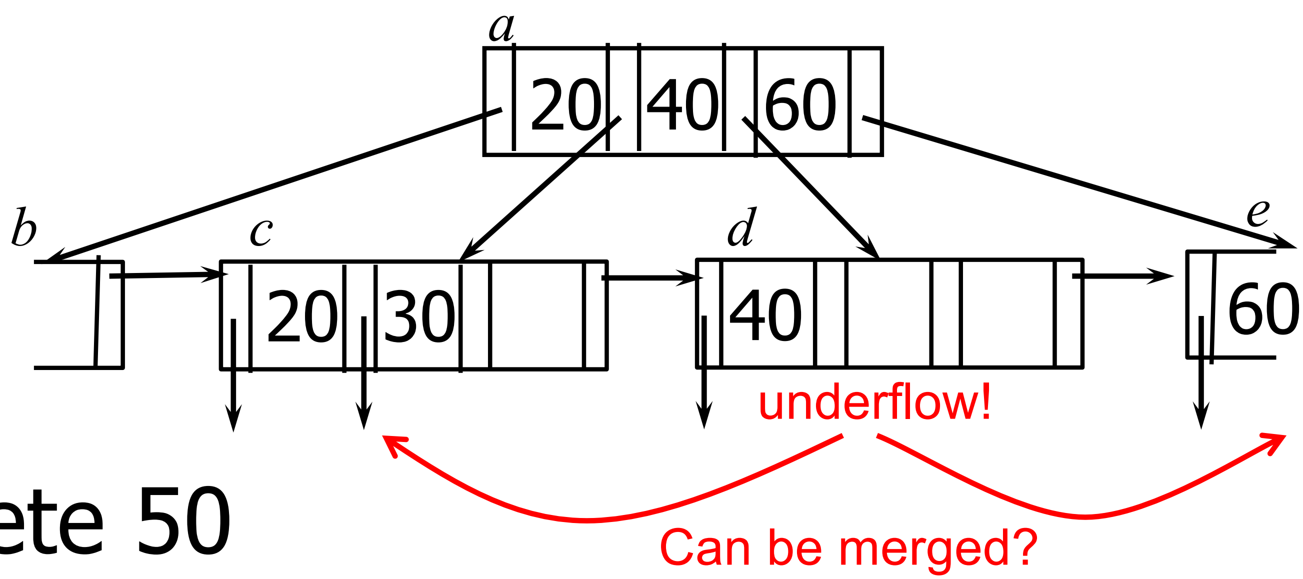
- Delete 50

(b) Coalesce with sibling (leaf)



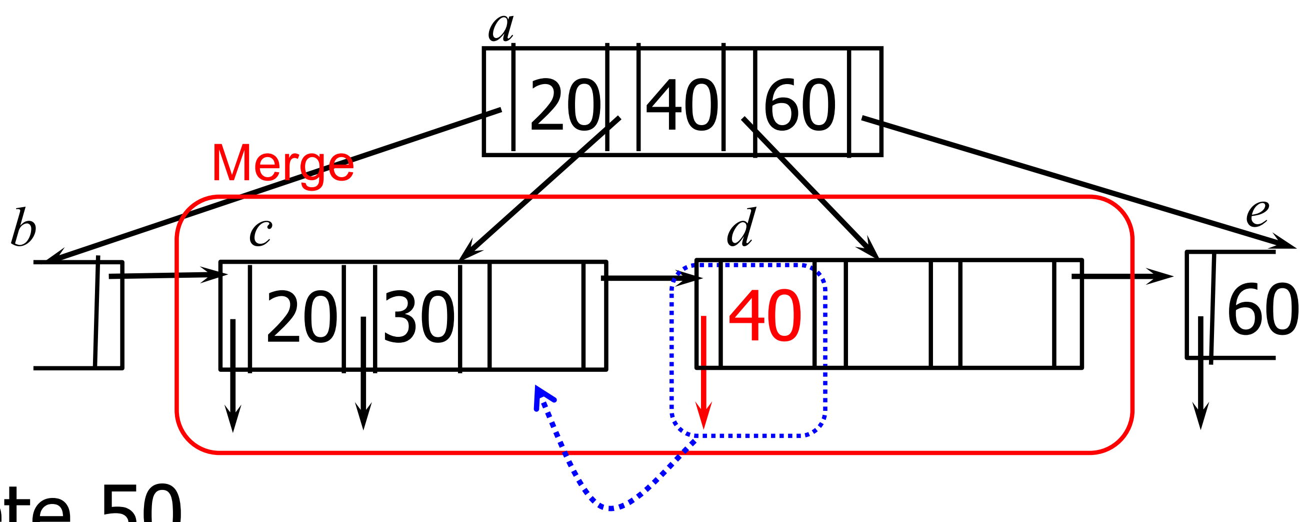
- Delete 50
  - Underflow? Min 3 ptrs, currently 2.

(b) Coalesce with sibling (leaf)



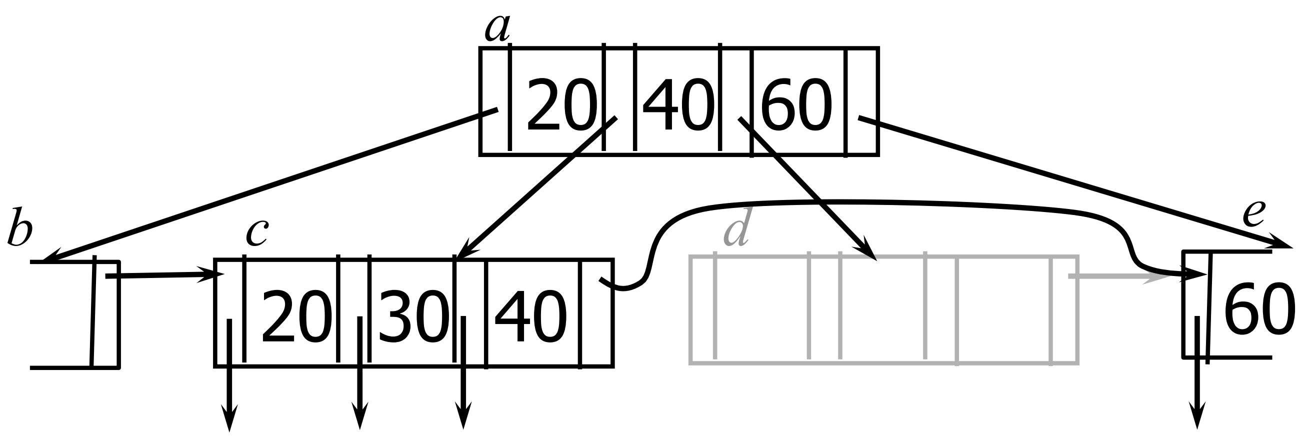
- Delete 50
  - Try to merge with a sibling

(b) Coalesce with sibling (leaf)



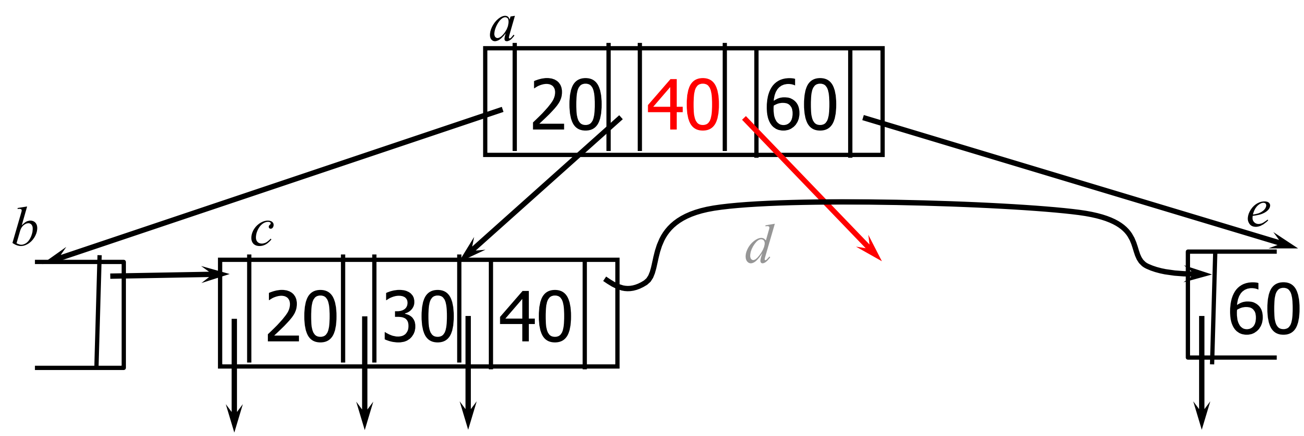
- Delete 50
  - Merge c and d. Move everything on the right to the left.

(b) Coalesce with sibling (leaf)



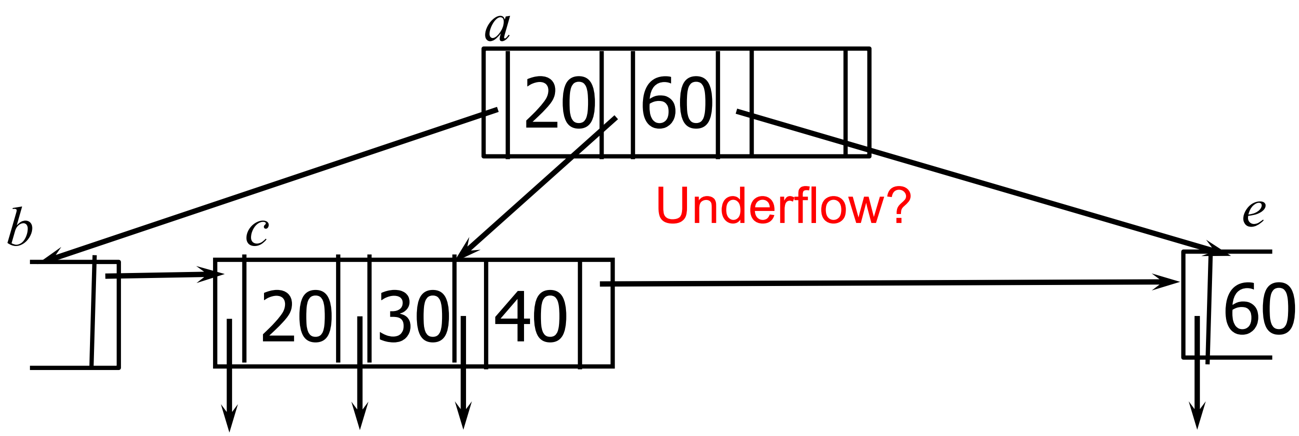
- Delete 50
  - Once everything is moved, delete *d*

(b) Coalesce with sibling (leaf)



- Delete 50
  - After leaf node merge,
    - From its parent, delete the pointer and key to the deleted node

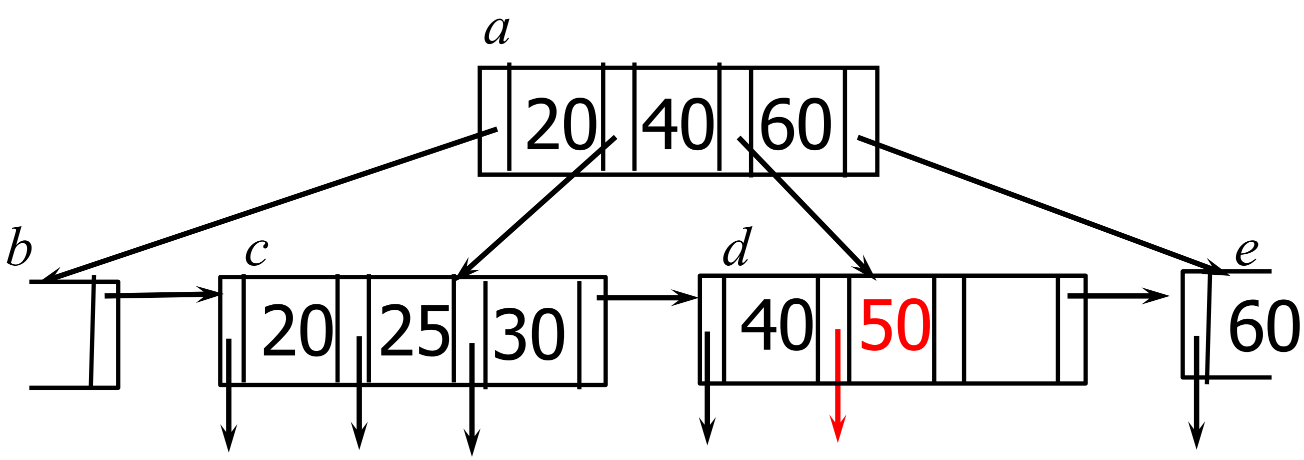
(b) Coalesce with sibling (leaf)



- Delete 50
  - Check underflow at a. Min 2 ptrs, currently 3

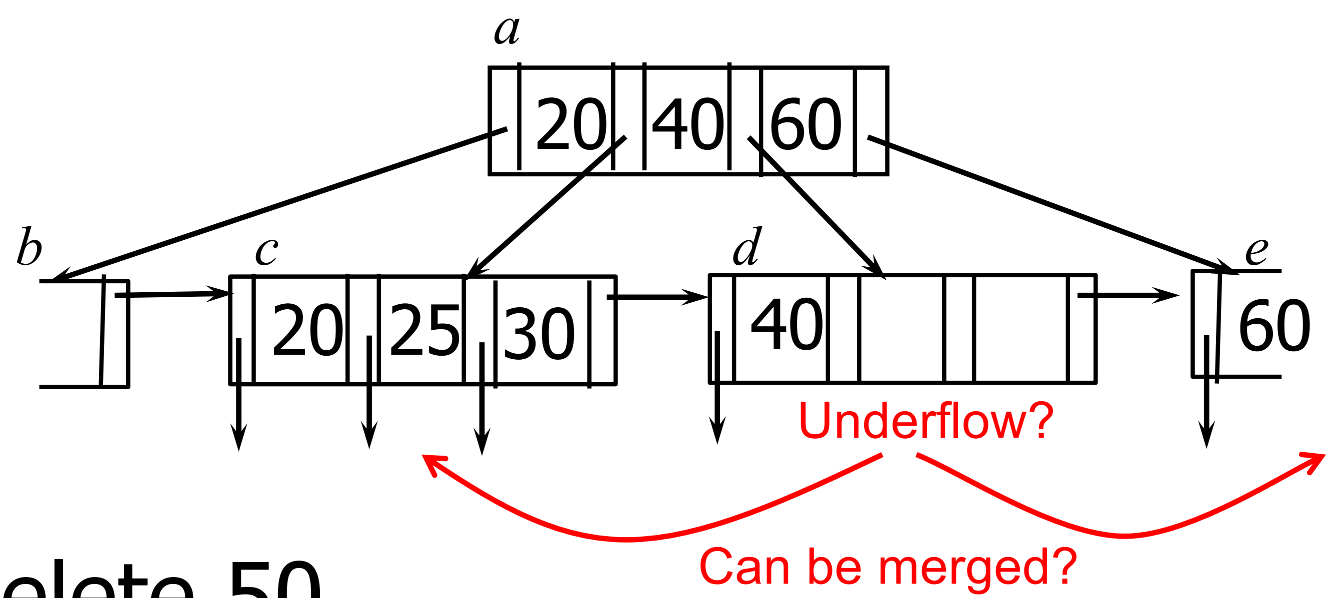
(c) Leaf node, redistribute with neighbor

(c) Redistribute (leaf)



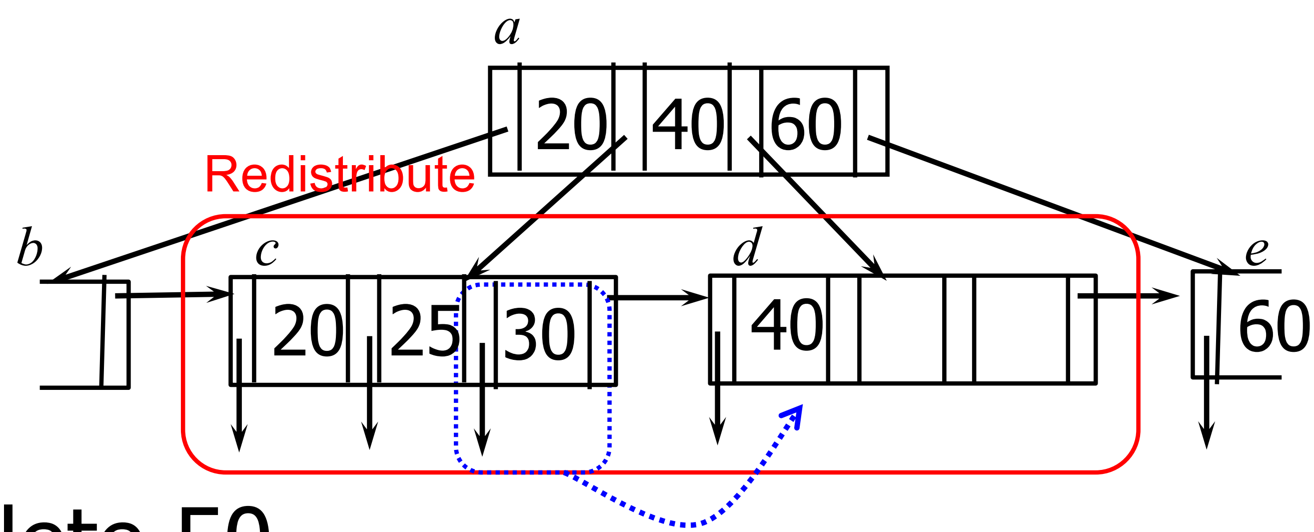
- Delete 50

(c) Redistribute (leaf)



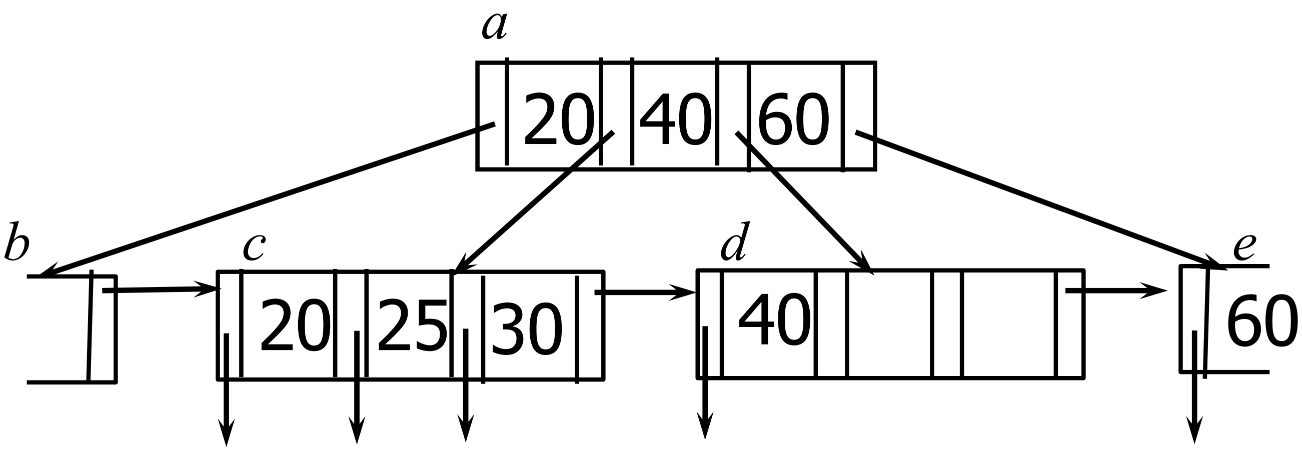
- Delete 50
  - Underflow? Min 3 ptrs, currently 2
  - Check if d can be merged with its sibling c or e
  - If not, redistribute the keys in d with a sibling
    - Say, with c

(c) Redistribute (leaf)



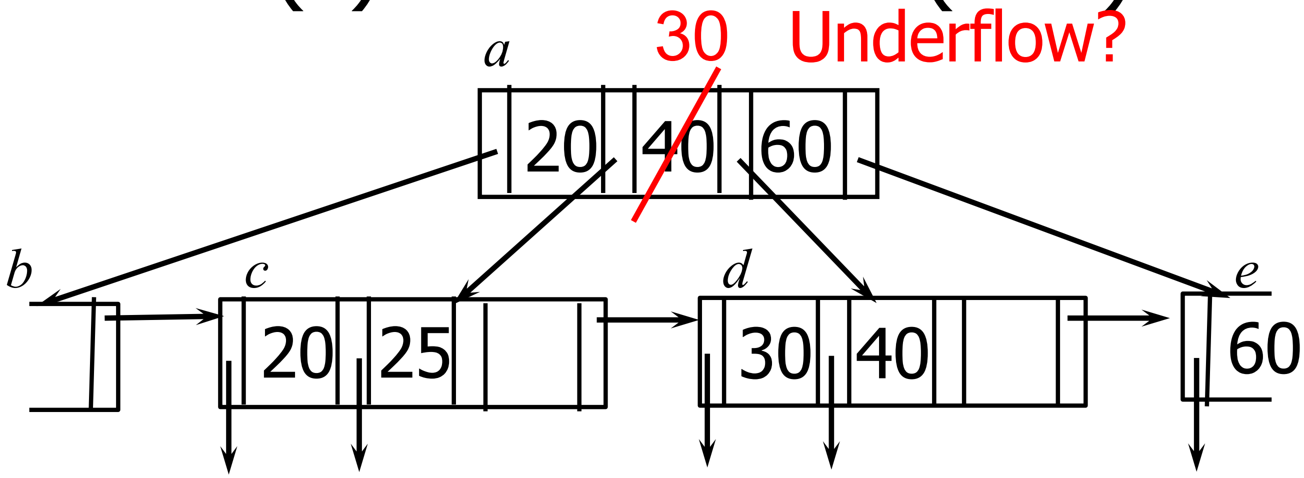
- Delete 50
  - Redistribute c and d, so that nodes c and d are roughly "half full"
    - Move the key 30 and its tuple pointer to the d

(c) Redistribute (leaf)



- Delete 50
  - Update the key in the parent

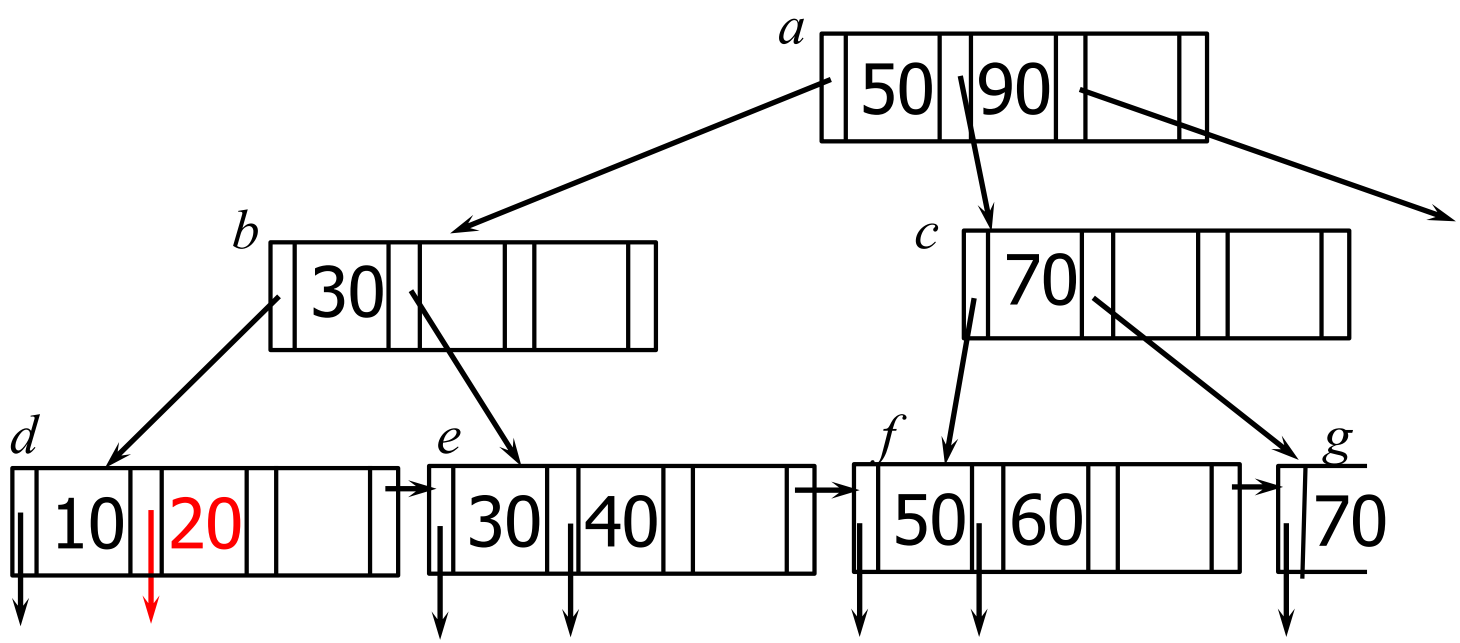
(c) Redistribute (leaf)



- Delete 50
  - No underflow at a. Done.

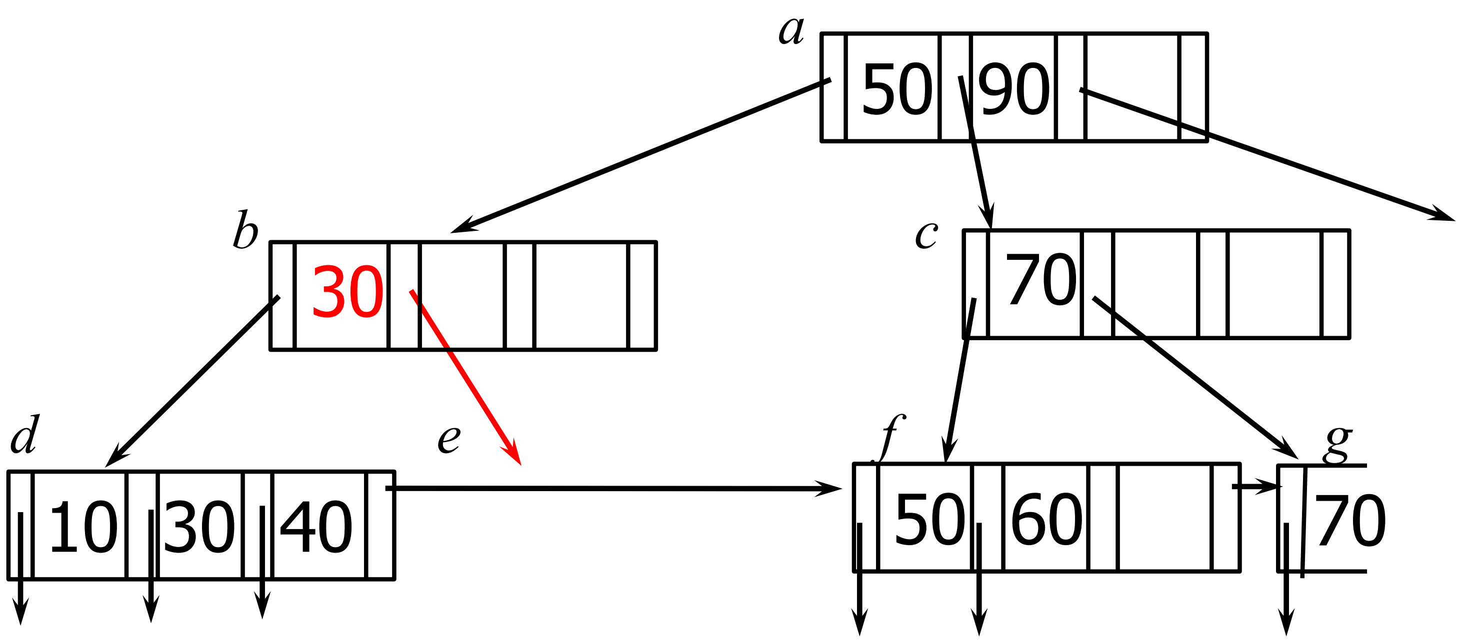
(d) Non-leaf node, coalesce with neighbor

(d) Coalesce (non-leaf)



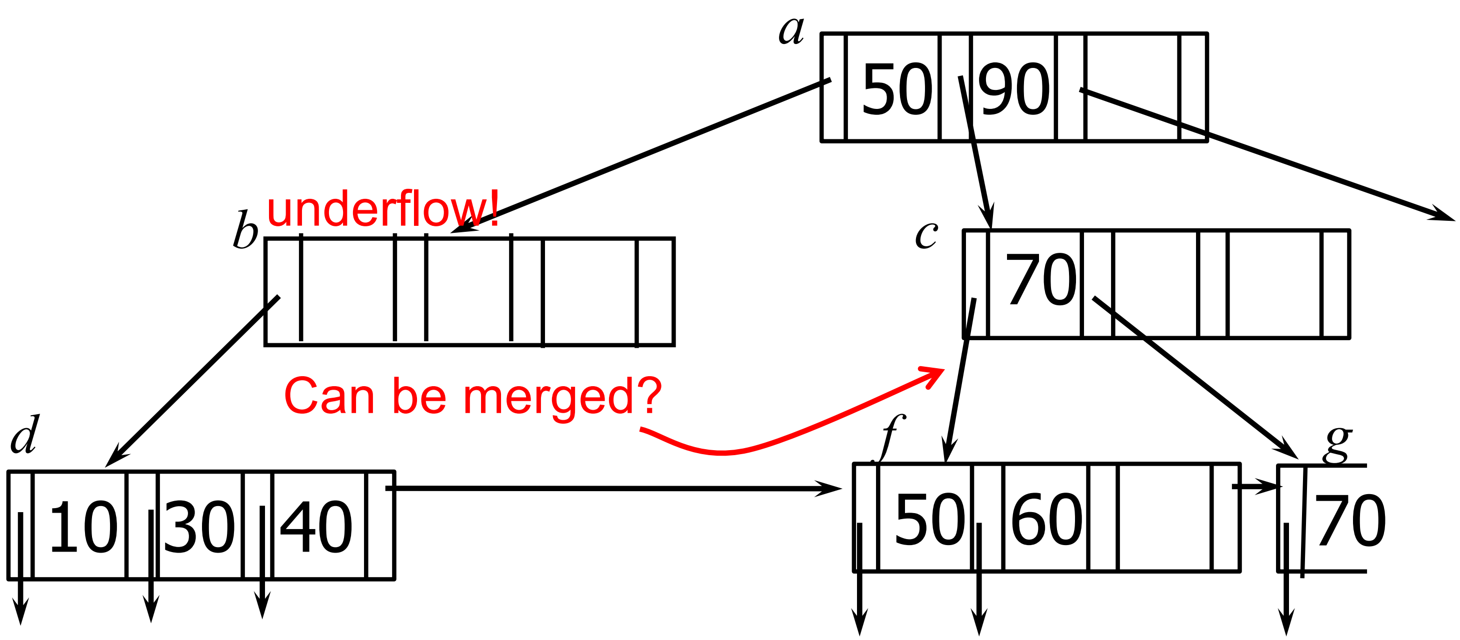
- Delete 20
  - Underflow! Merge *d* with *e*.
    - Move everything in the right to the left

(d) Coalesce (non-leaf)



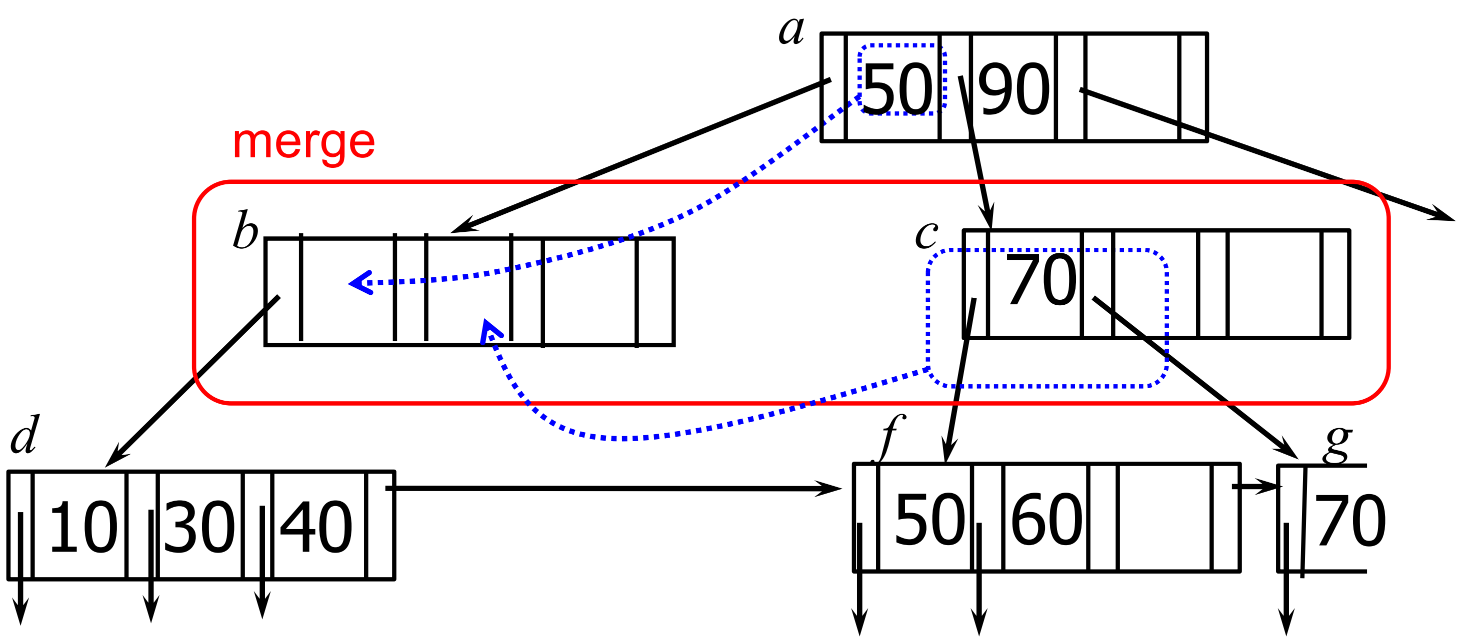
- Delete 20
  - From the parent node, delete pointer and key to the deleted node

(d) Coalesce (non-leaf)



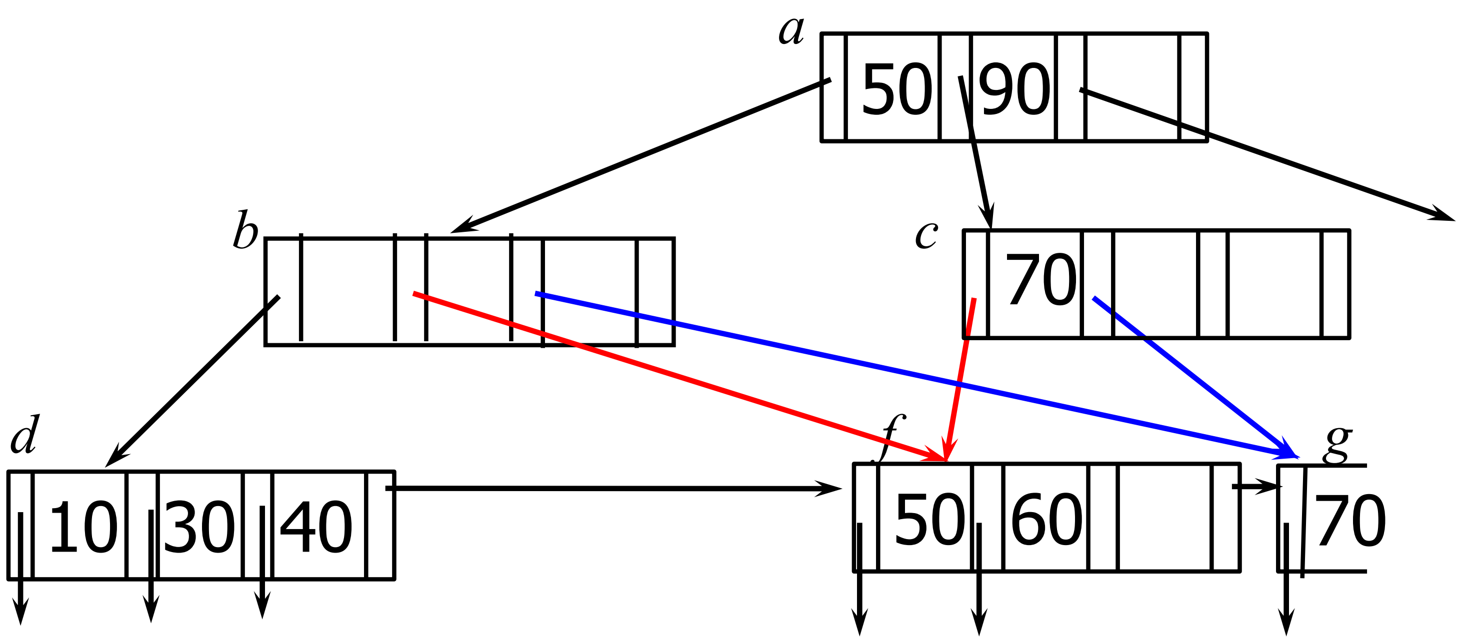
- Delete 20
  - Underflow at b? Min 2 ptrs, currently 1.
  - Try to merge with its sibling.
    - Nodes b and c: 3 ptrs in total. Max 4 ptrs.
    - Merge b and c.

(d) Coalesce (non-leaf)



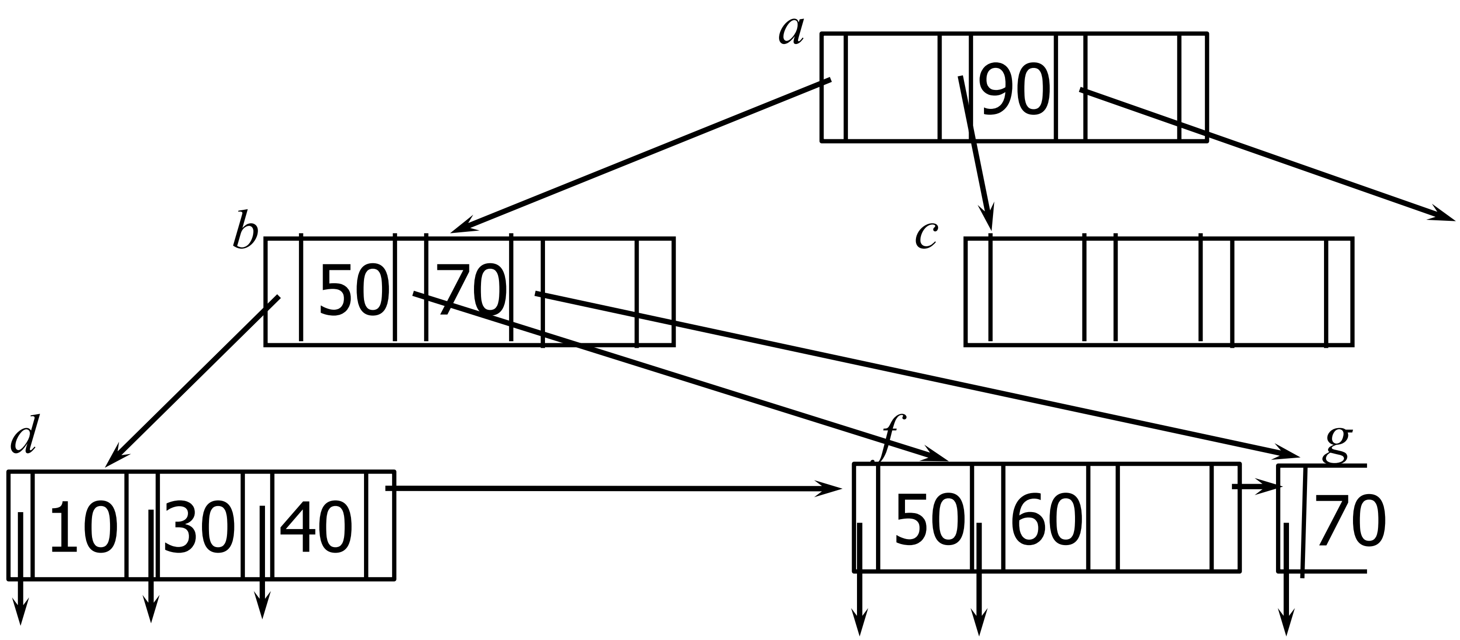
- Delete 20
  - Merge b and c
    - Pull down the mid-key 50 in the parent node
    - Move everything in the right node to the left.
- Very important: when we merge <u>*non-leaf nodes*</u>, we always pull down the mid-key in the parent and place it in the merged node.

(d) Coalesce (non-leaf)



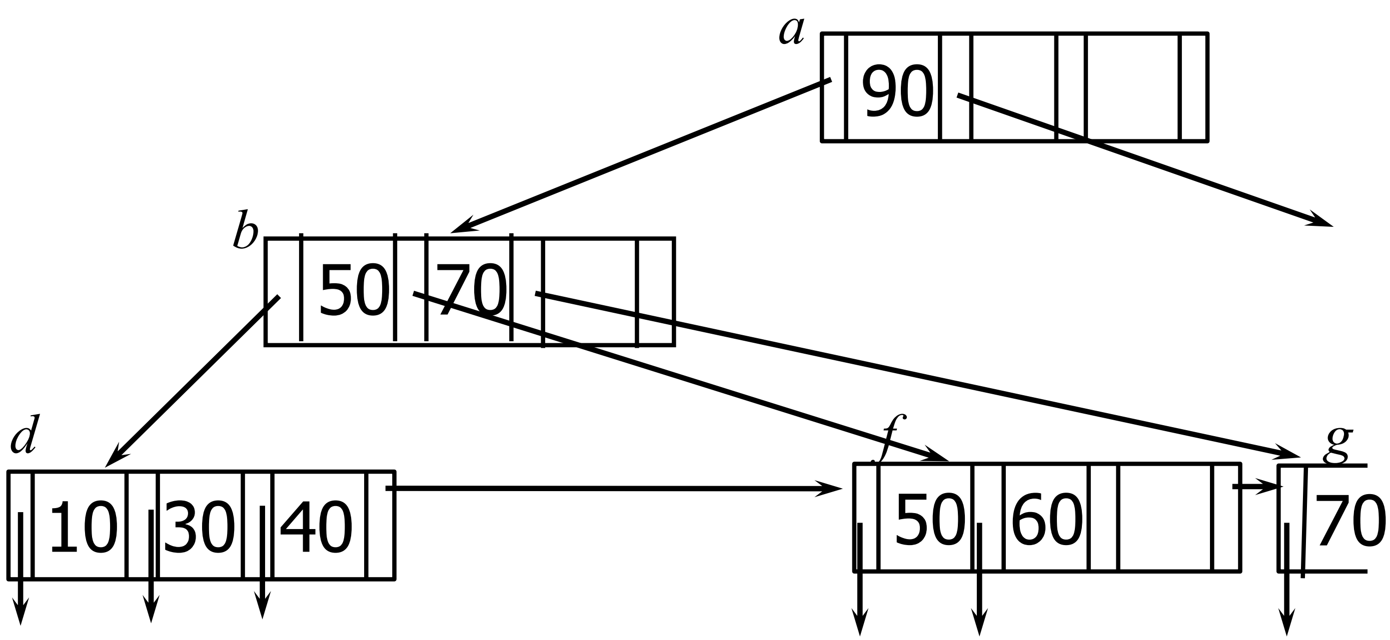
- Delete 20
  - Merge b and c
    - Pull down the mid-key 50 in the parent node
    - Move everything in the right node to the left.
- Very important: when we merge *non-leaf nodes*, we always pull down the mid-key in the parent and place it in the merged node.

(d) Coalesce (non-leaf)



- Delete 20
  - Delete pointer to the merged node.

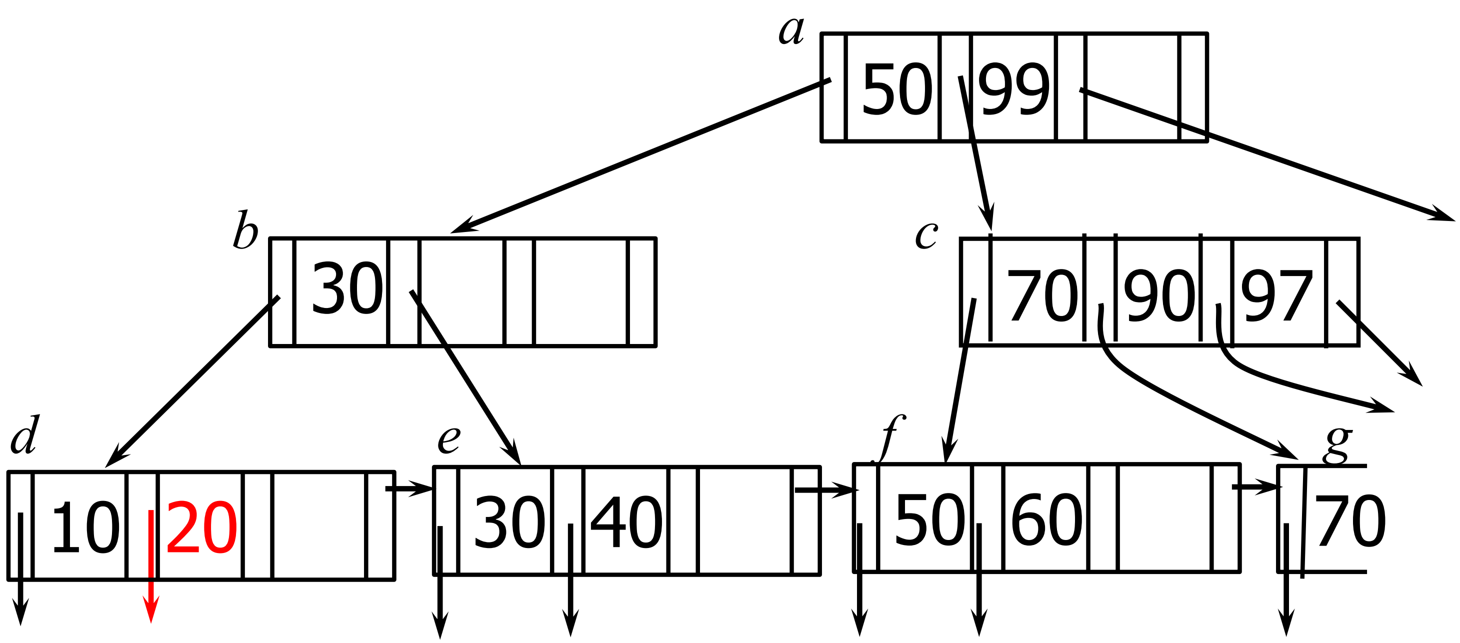
(d) Coalesce (non-leaf)



- Delete 20
  - Underflow at a? Min 2 ptrs. Currently 2. Done.

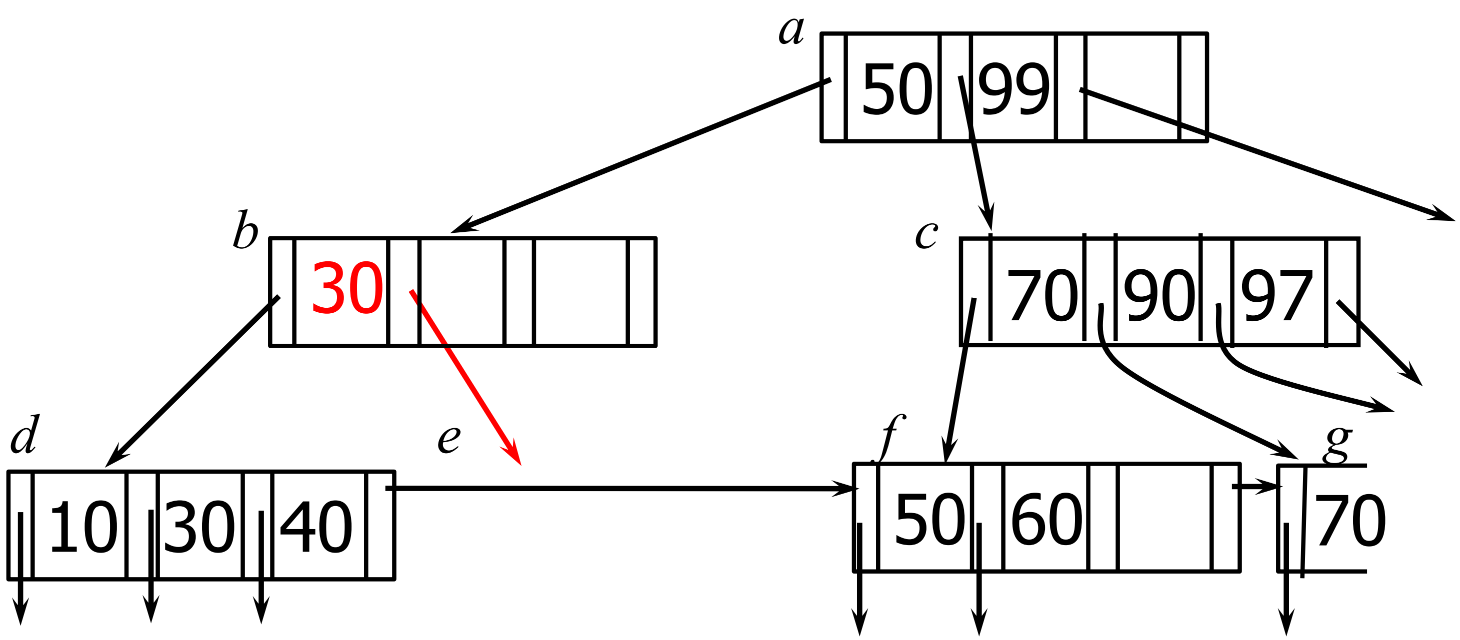
(e) Non-leaf node, redistribute with neighbor

(e) Redistribute (non-leaf)



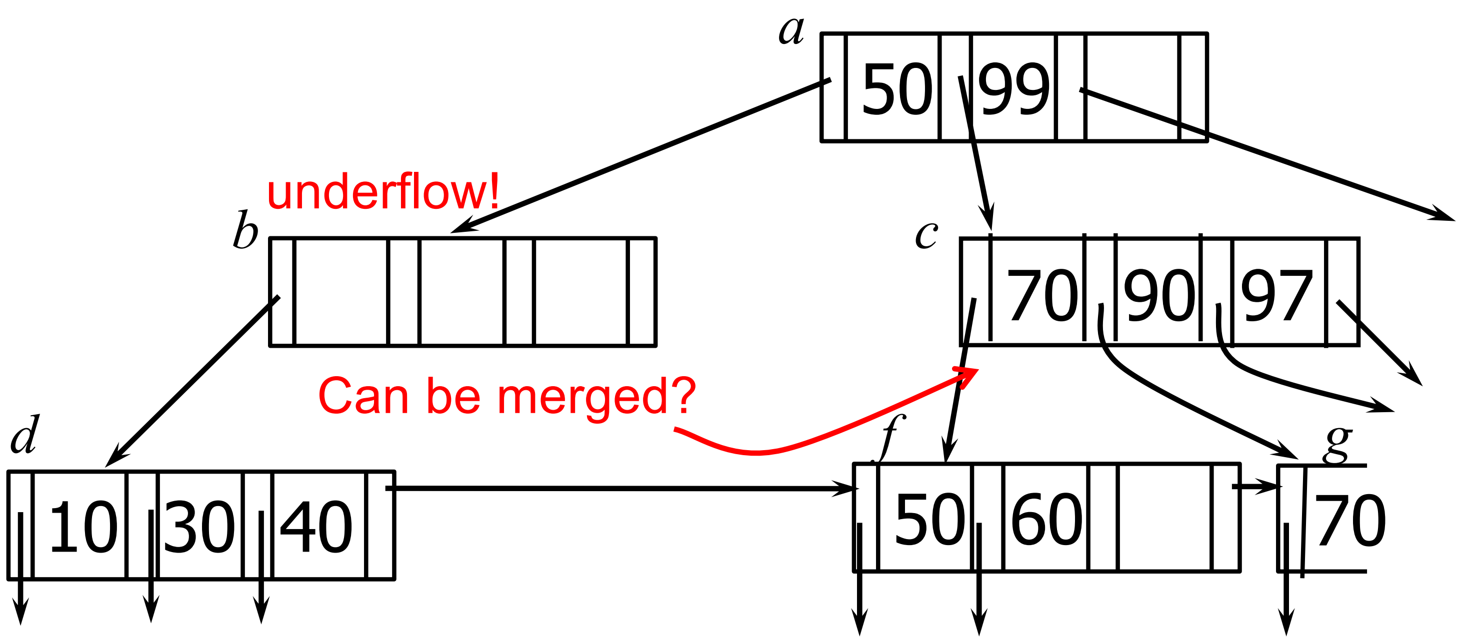
- Delete 20
  - Underflow! Merge *d* with *e*.

(e) Redistribute (non-leaf)



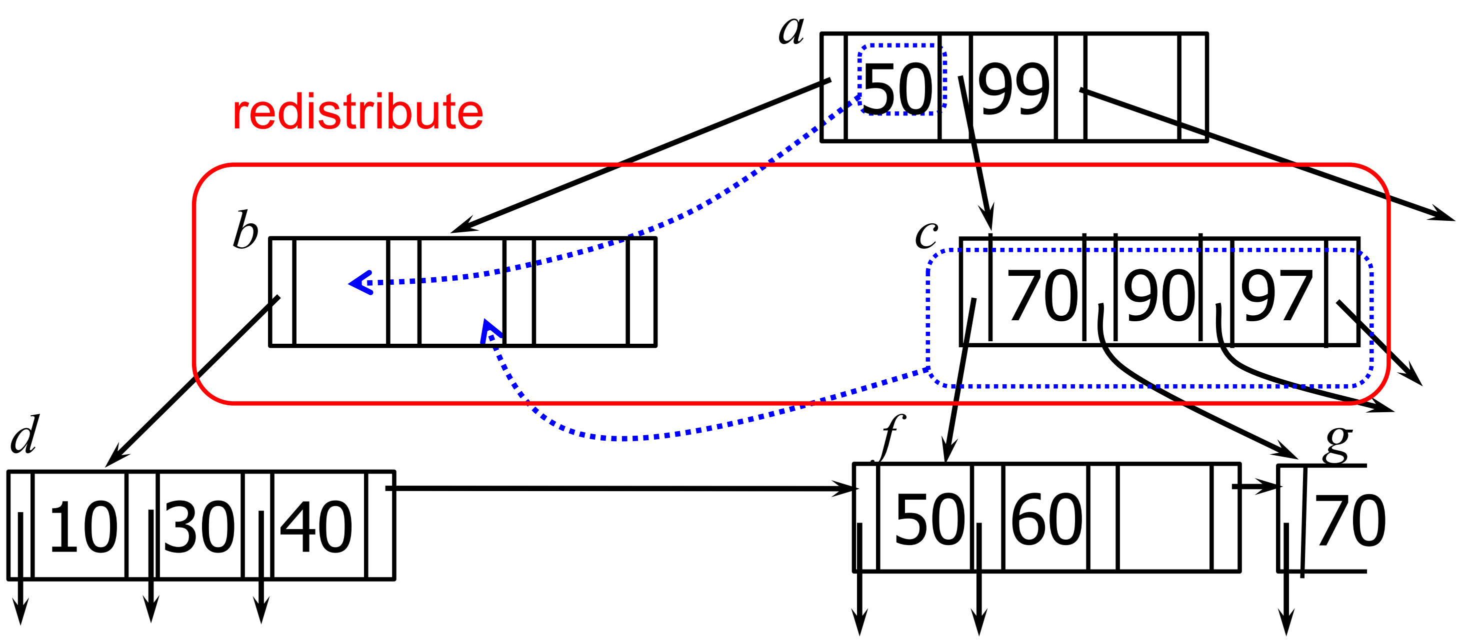
- Delete 20
  - After merge, remove the key and ptr to the deleted node from the parent

(e) Redistribute (non-leaf)



- Delete 20
  - Underflow at b? Min 2 ptrs, currently 1.
  - Merge b with c? Max 4 ptrs, 5 ptrs in total.
  - If cannot be merged, redistribute the keys with a sibling.
    - Redistribute b and c

(e) Redistribute (non-leaf)

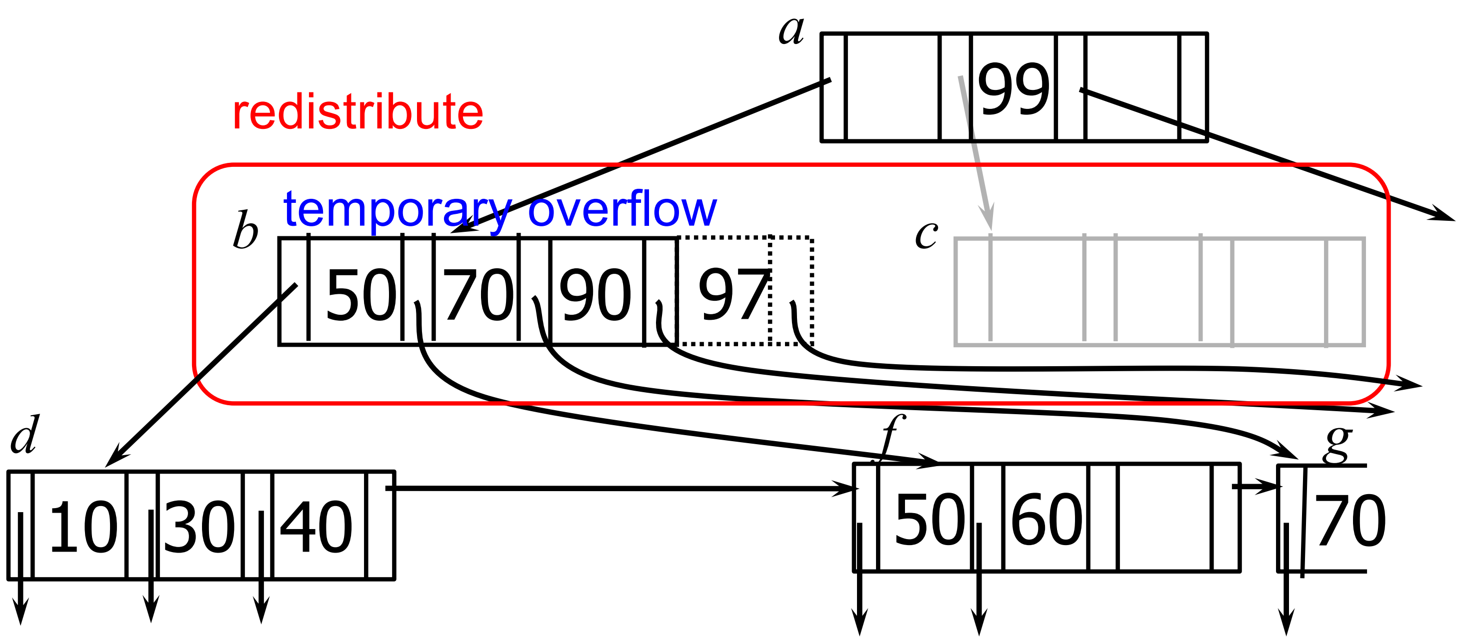


- Delete 20

<u>Redistribution at a non-leaf node is done in two steps</u>.

Step 1: Temporarily, make the left node b “overflow” by pulling down the mid-key and moving everything to the left.

(e) Redistribute (non-leaf)

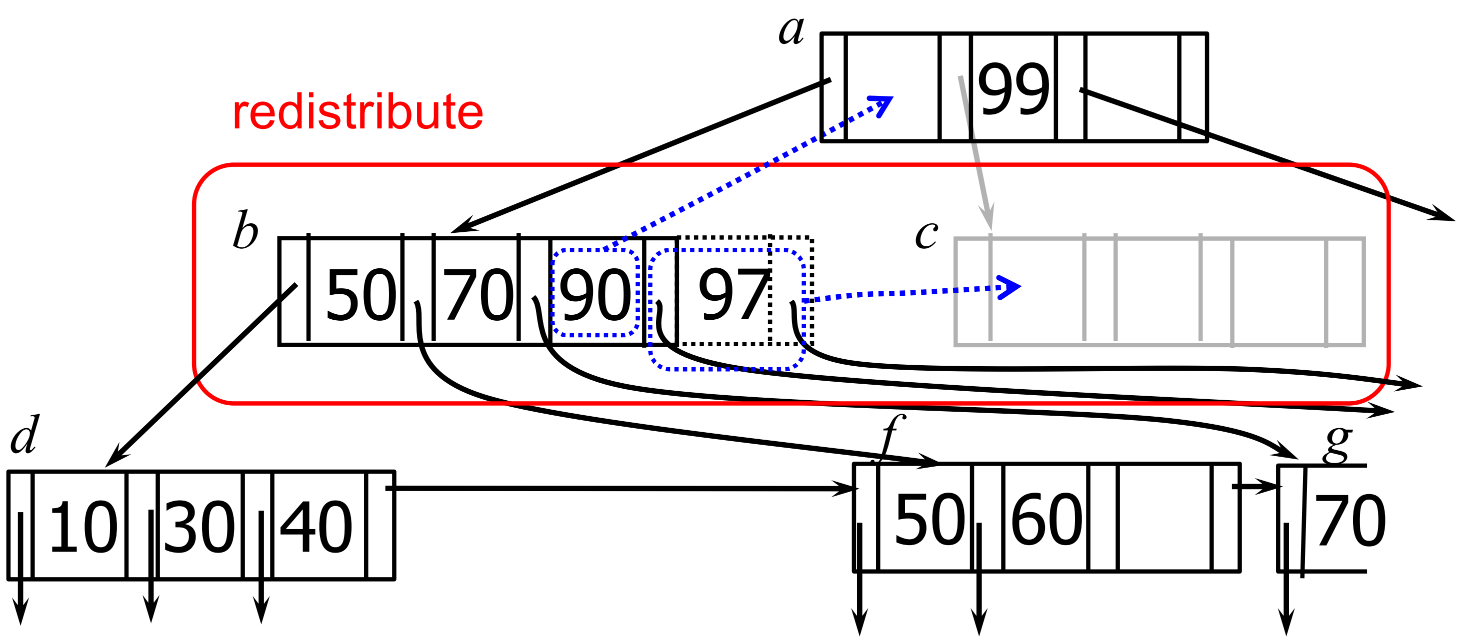


- Delete 20

Step 2: Apply the “overflow handling algorithm” (the same algorithm used for B+tree insertion) to the overflowed node

  – Detailed algorithm in the next slide

(e) Redistribute (non-leaf)

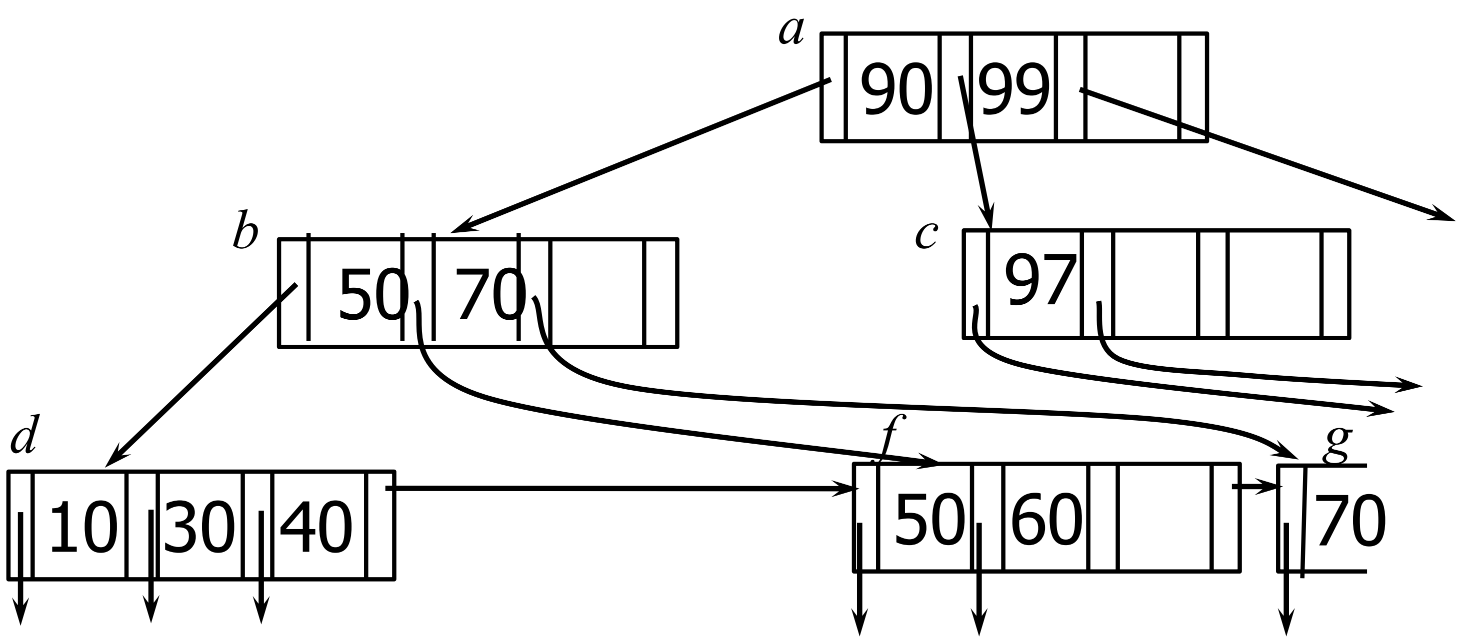


- Delete 20

Step 2: "overflow handling algorithm"

- Pick the mid-key (say 90) in the node and move it to parent.
- Move everything to the right of 90 to the empty node c.

(e) Redistribute (non-leaf)



- Delete 20
  - Underflow at a? Min 2 ptrs, currently 3. Done

Important Points

- Remember:
  - For *leaf node* merging, we *delete* the mid-key from the parent
  - For *non-leaf node* merging/redistribution, we *pull down* the mid-key from their parent.
- In practice
  - Coalescing is often not implemented
    - Too hard and not worth it

Where does *n* come from?

- *n* determined by
  - – Size of a node
  - – Size of search key
  - – Size of an index pointer
- Q: 1024B node, 10B key, 8B ptr → *n*?
- Computation: 8 bytes for final pointer. Then 1024-8=1016 bytes left for Key+pointer pair, each taking 18 bytes. Thus floor(1016/18)= 56. Thus n=57.
- Or use formulas

Summary on tree index

- Issues to consider
  - Sparse vs. dense
  - Primary (clustering) vs. secondary (non-clustering)
- Indexed sequential file (ISAM)
  - Simple algorithm. Sequential blocks
  - Not suitable for dynamic environment
- B+trees
  - Balanced, minimum space guarantee
  - Insertion, deletion algorithms

Index Creation in SQL

- `CREATE INDEX <indexname> ON <table>(<attr>,<attr>,…)`
- Example
  - `CREATE INDEX stidx ON Student(sid)`
    - Creates a B+tree on the attributes
    - Speeds up lookup on sid

Primary (Clustering) Index

- MySQL:
  - Primary key becomes the clustering index
- DB2:
  - **`CREATE INDEX idx ON Student(sid) CLUSTER`**
  - Tuples in the table are sequenced by sid
- Oracle: Index-Organized Table (IOT)
  - ```
    CREATE TABLE T (
        ...
    ) ORGANIZATION INDEX
    ```
  - B+tree on primary key
  - Tuples are stored at the leaf nodes of B+tree
- Periodic reorganization may still be necessary to improve range scan performance
    ```

Next topic

- Hash index
  - Static hashing
  - Extendible hashing

What is a Hash Table?

- Hash Table
  - Hash function
    - h(k): key → integer [0…n]
    - e.g., h('Susan') = 7
  - Array for keys: T[0…n]
  - Given a key k, store it in T[h(k)]

h(Susan) = 4
h(James) = 3
h(Neil) = 1

0	
1	Neil
2	
3	James
4	Susan
5	

Overflow and Chaining

- Insert

  h(a) = 1

  h(b) = 2

  h(c) = 1

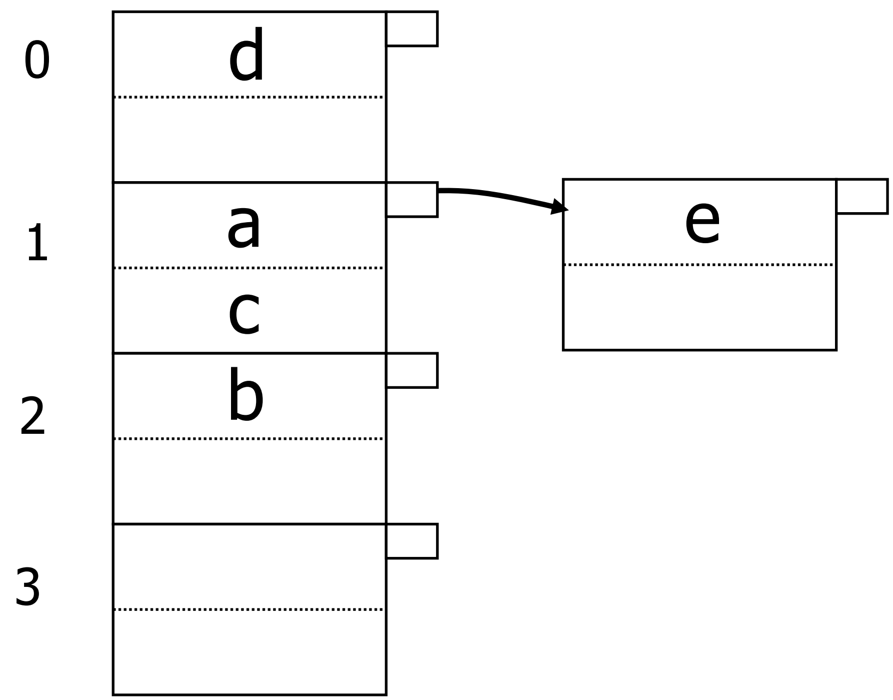
  h(d) = 0

  h(e) = 1

- Delete

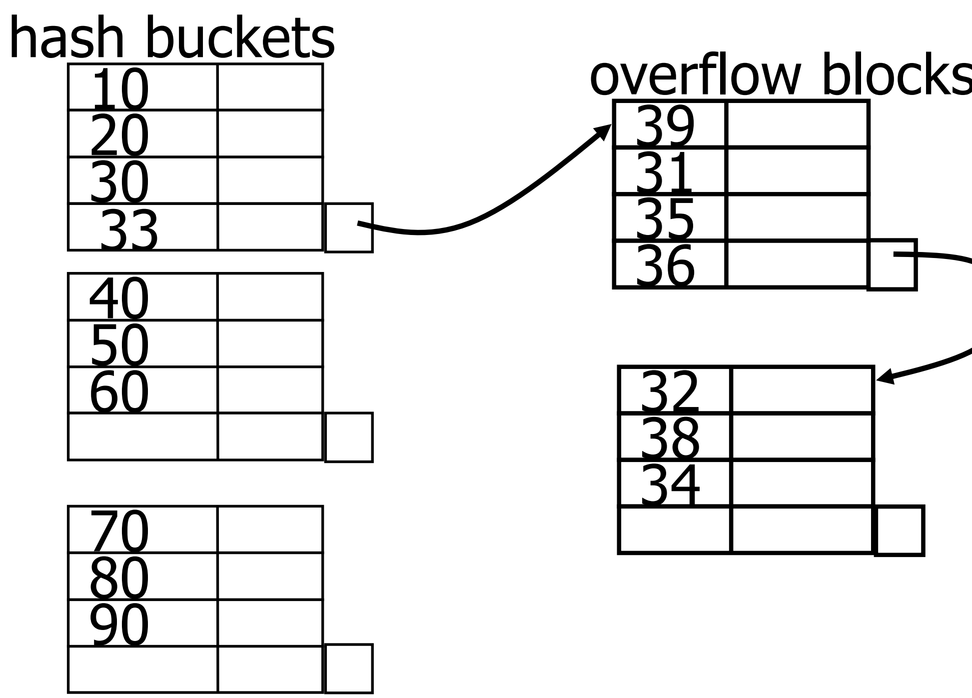
  h(b) = 2

  h(c) = 1



Major Problem of Static Hashing

- How to cope with growth?
  - Data tends to grow in size
  - Overflow blocks unavoidable



Extendible Hashing
(two ideas)

(a) Use *i* of *b* bits output by hash function

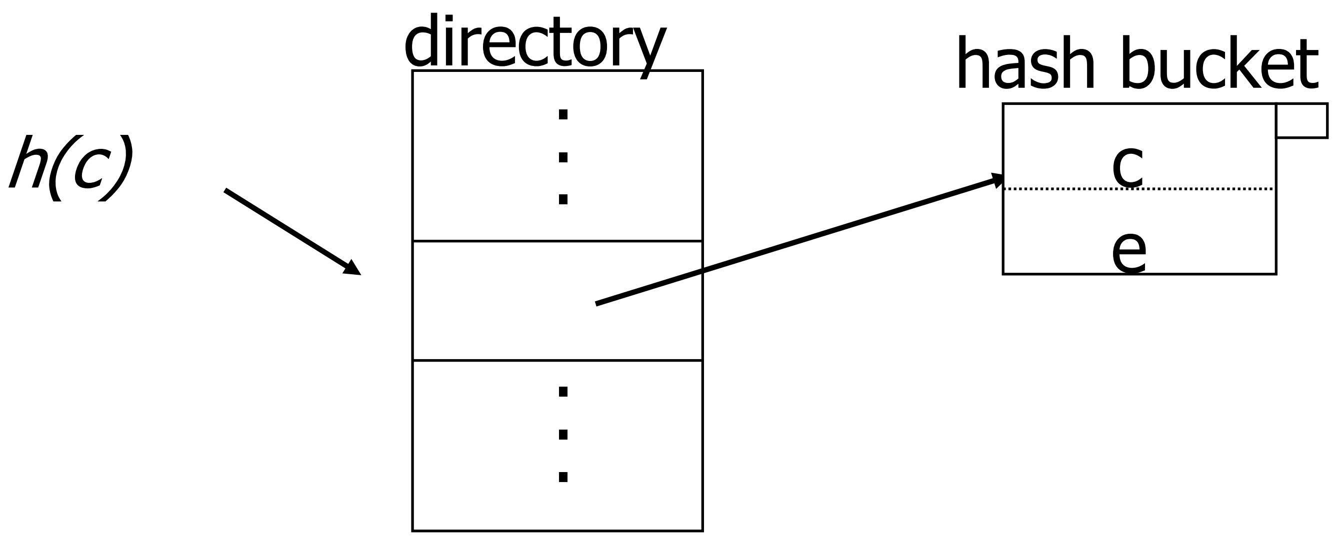
$$\longleftarrow \; b \; \longrightarrow$$

$$h(K) \rightarrow \boxed{\underbrace{00}_{}110101}$$

use *i* $\rightarrow$ grows over time

Extendible Hashing
(two ideas)

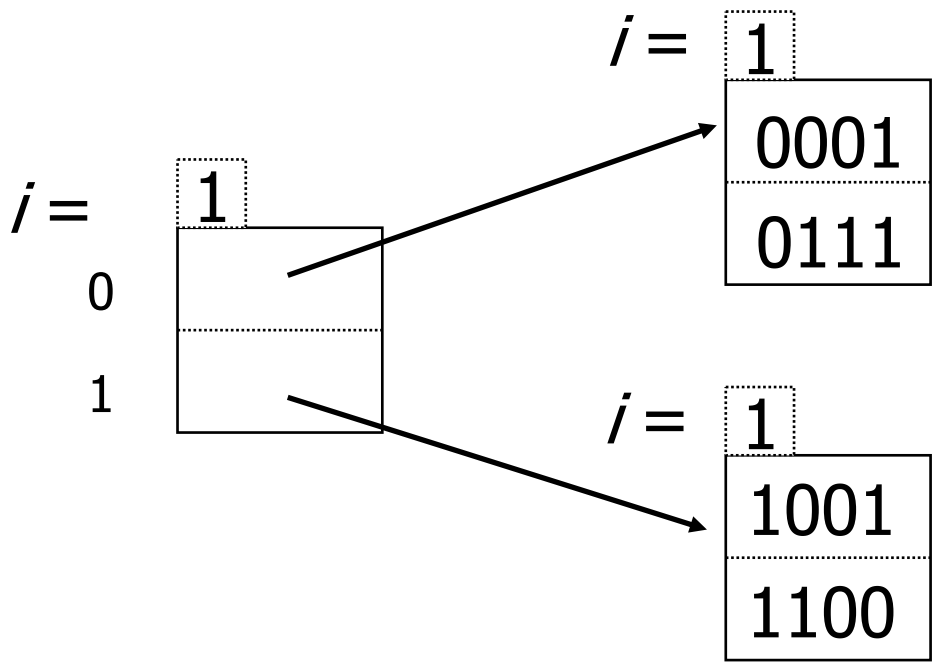
(b) Use directory that maintains pointers to hash buckets (indirection)



Example

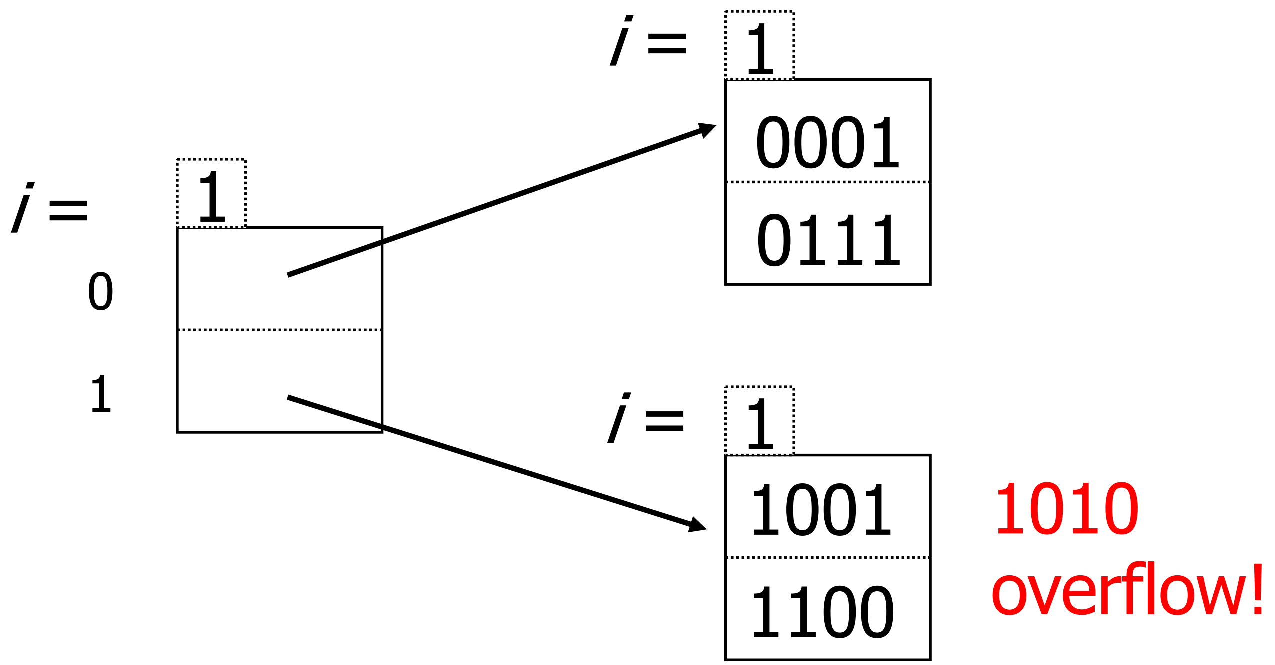
- h(k) is 4 bits; 2 keys/bucket

Insert 0111



Example

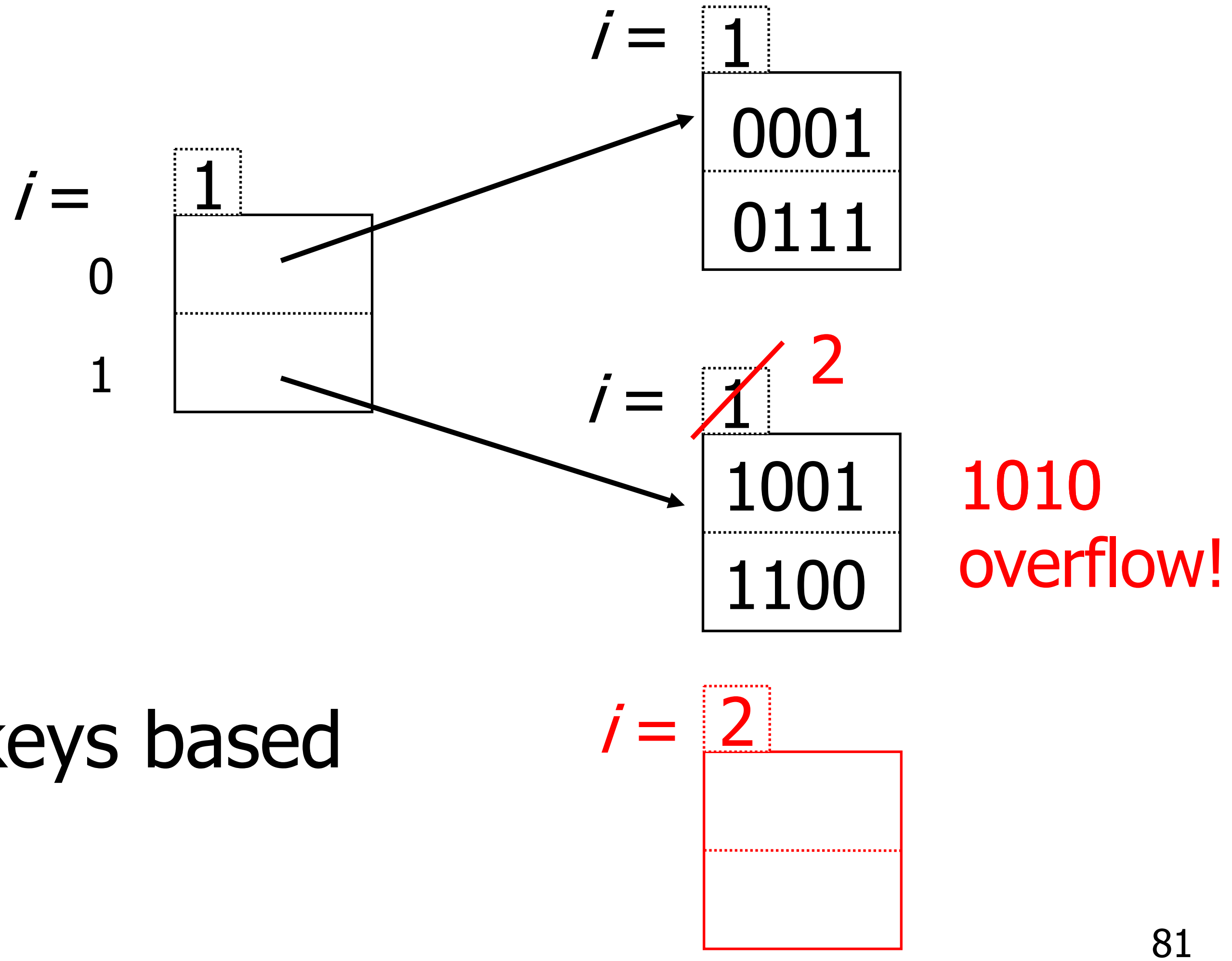
Insert 1010



Increase i of the bucket. Split it.

Example

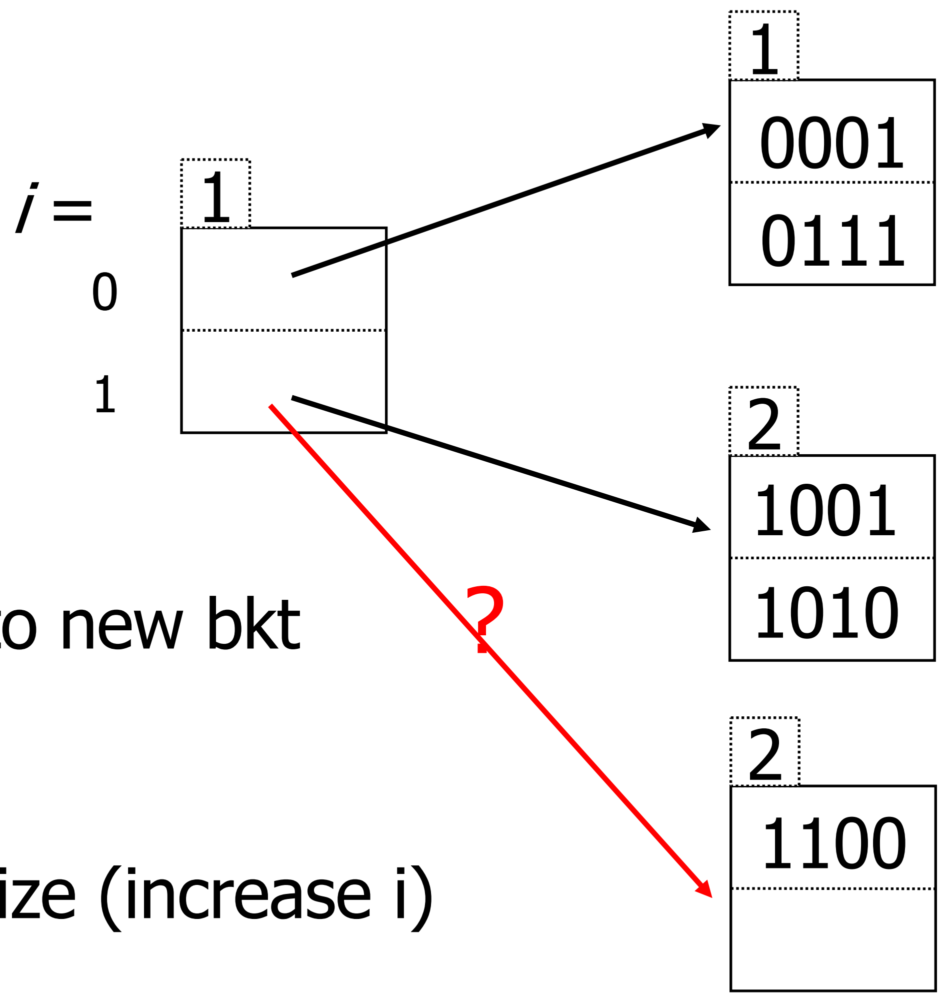
Insert 1010



Redistribute keys based on first i bits

Example

Insert 1010

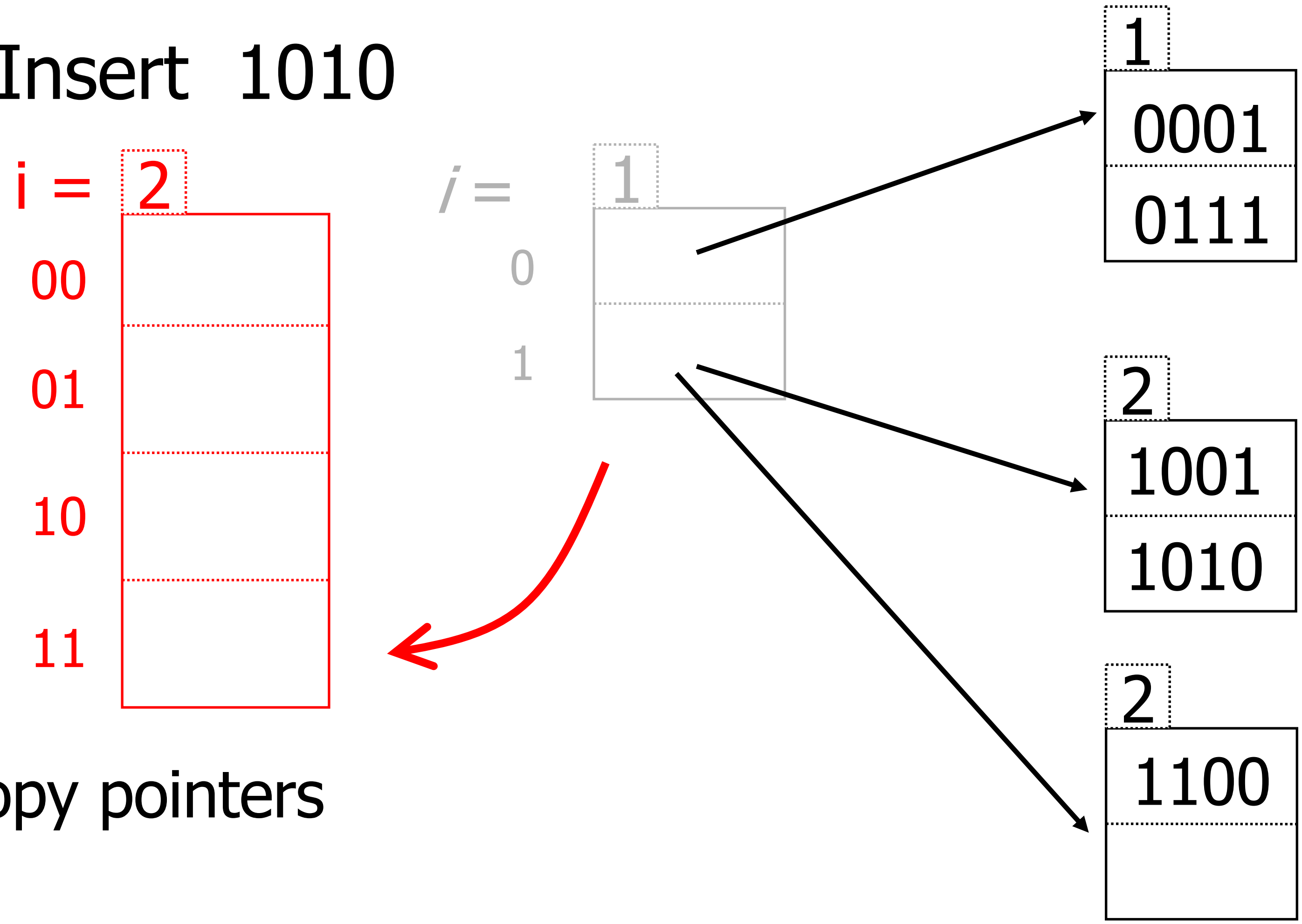


Update ptr in dir to new bkt

If no space,
double directory size (increase i)

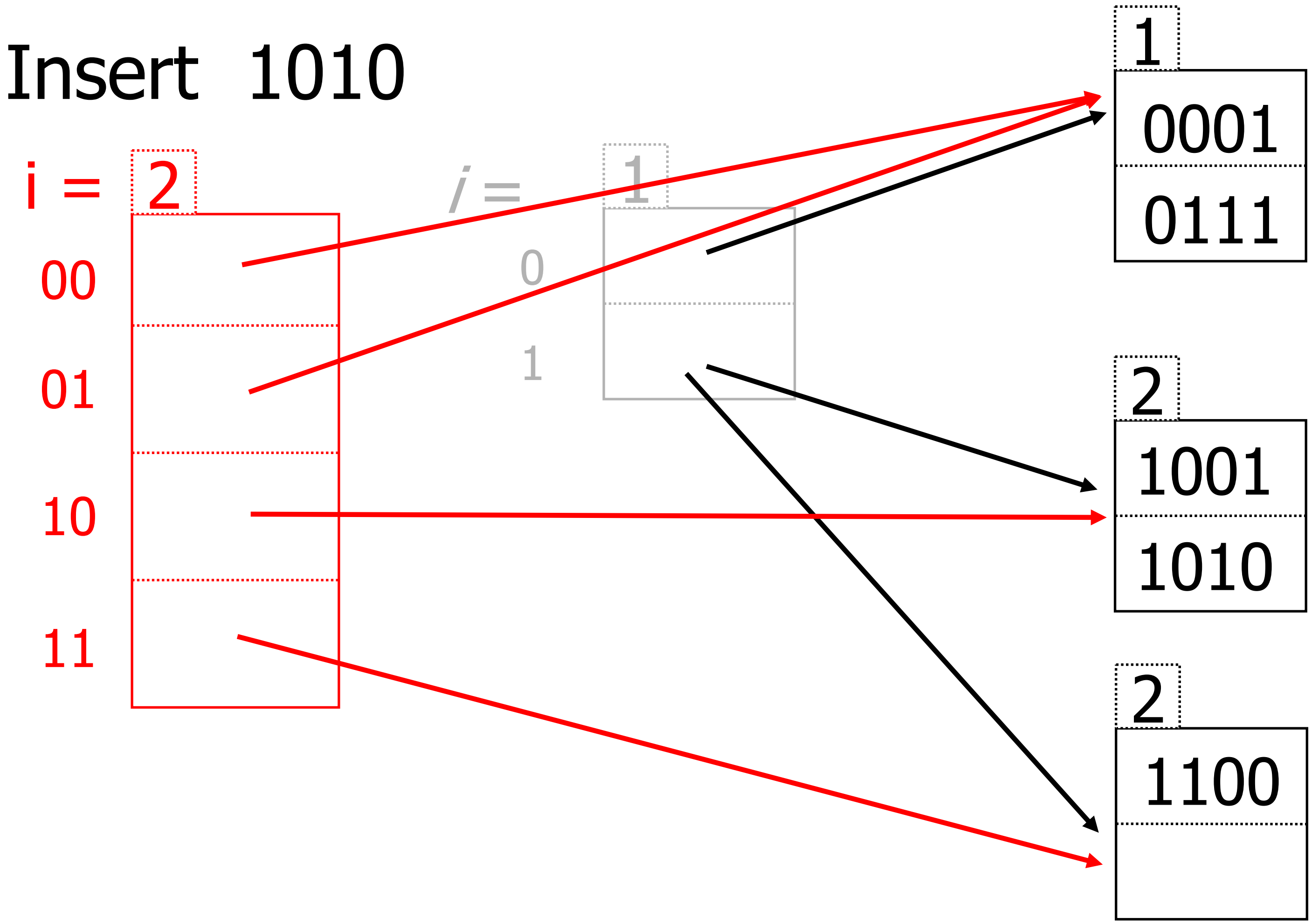
Example

Insert 1010



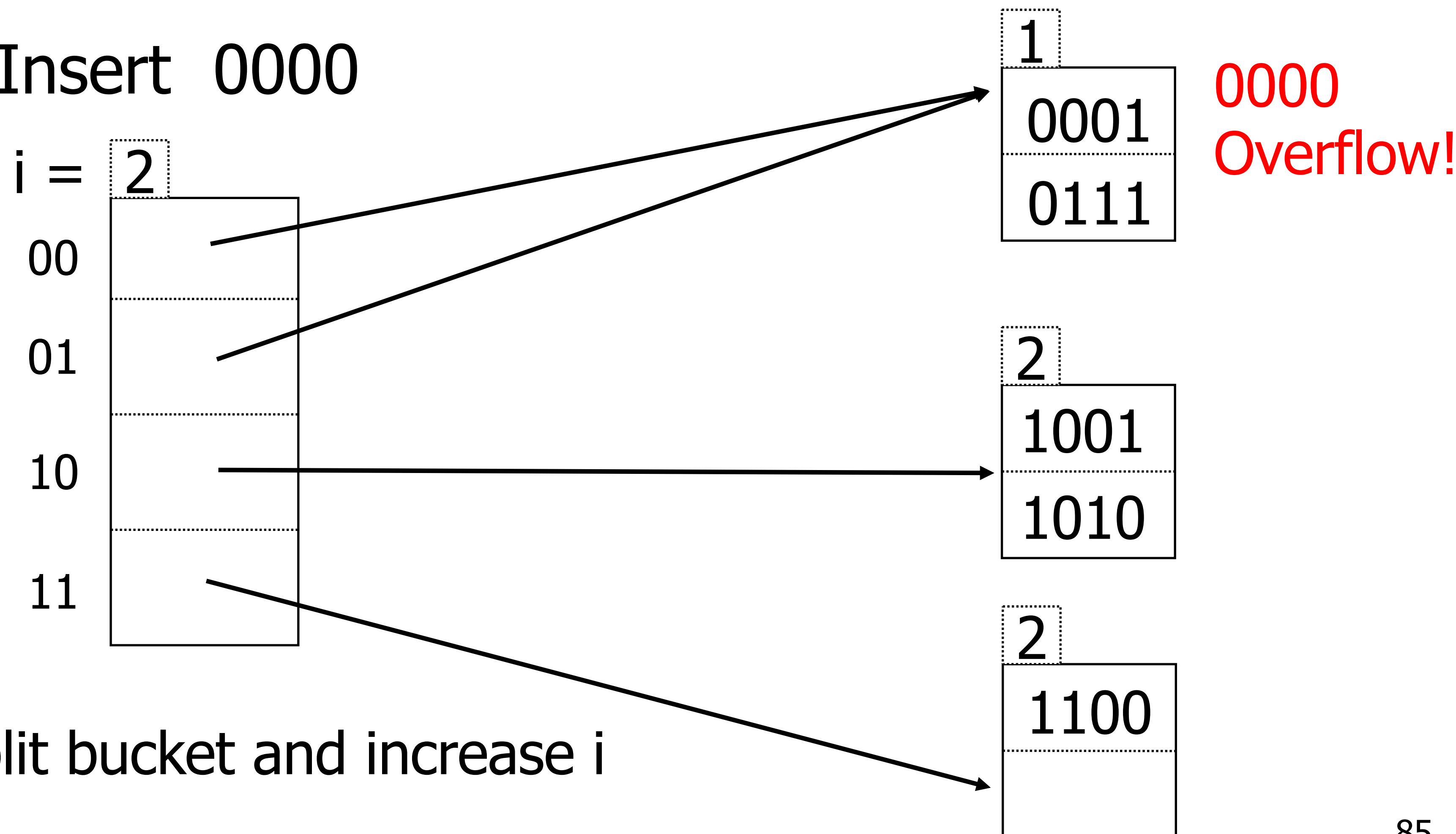
Copy pointers

Example



Example

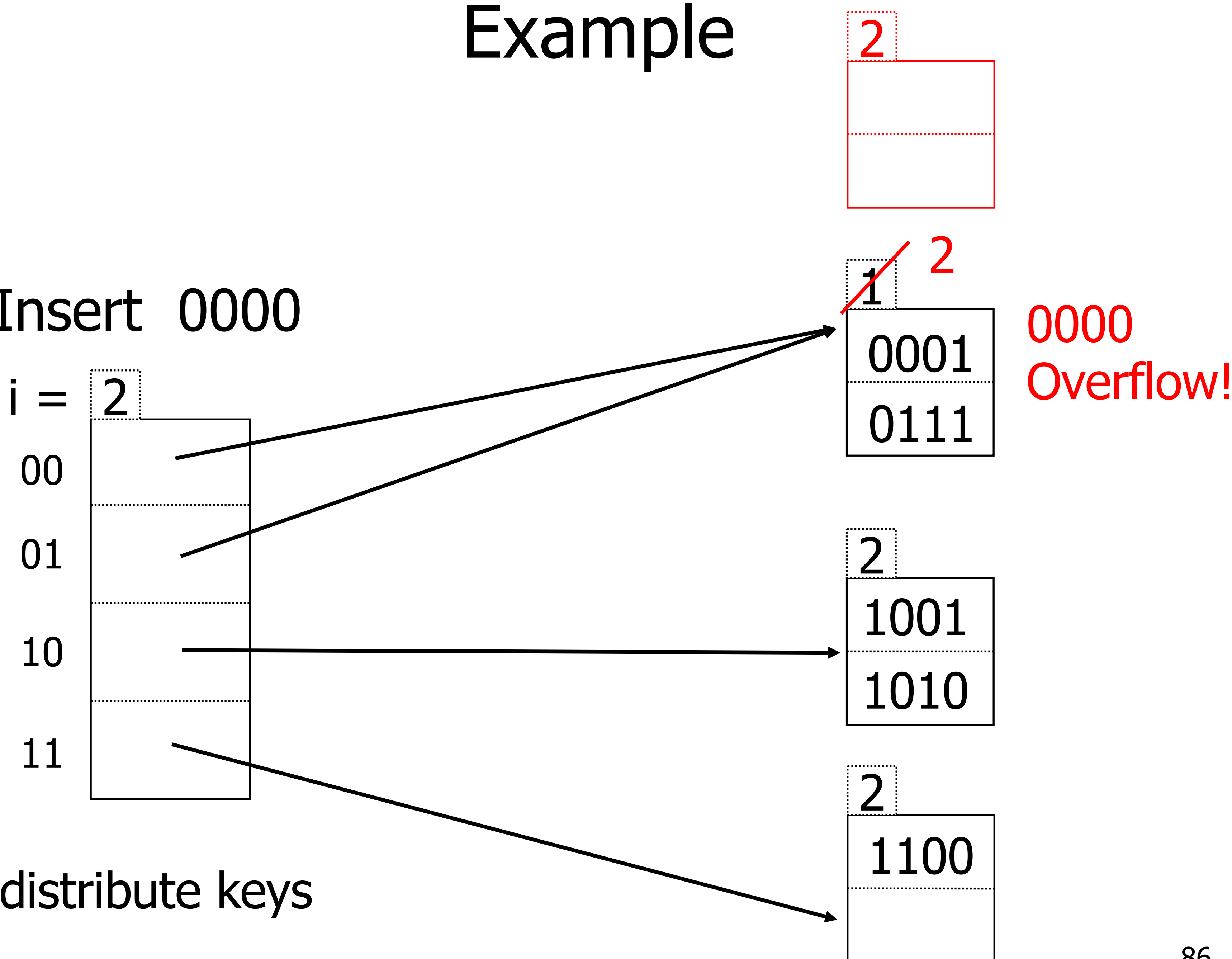
Insert 0000



Split bucket and increase i

Example

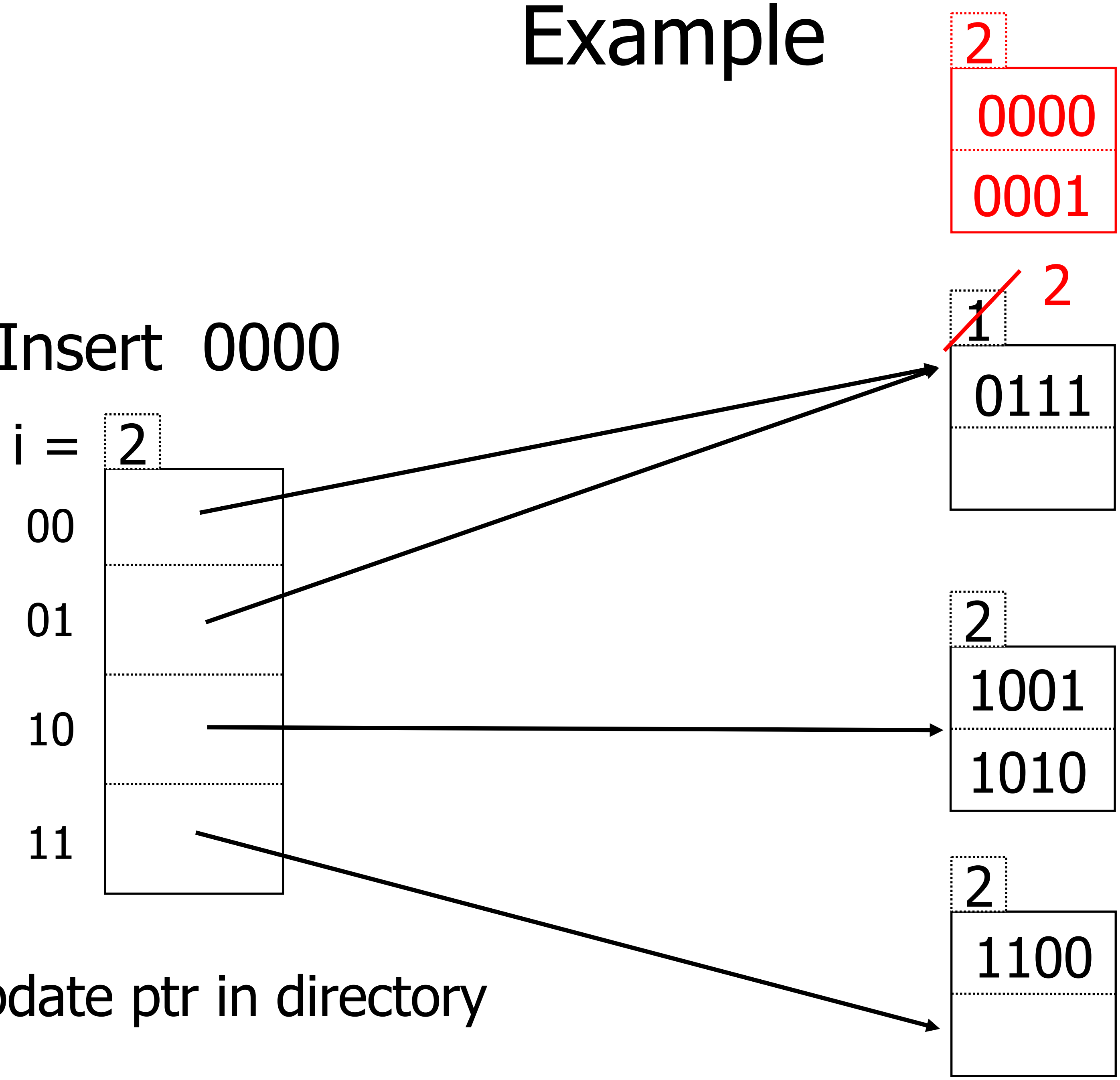
Insert 0000



Redistribute keys

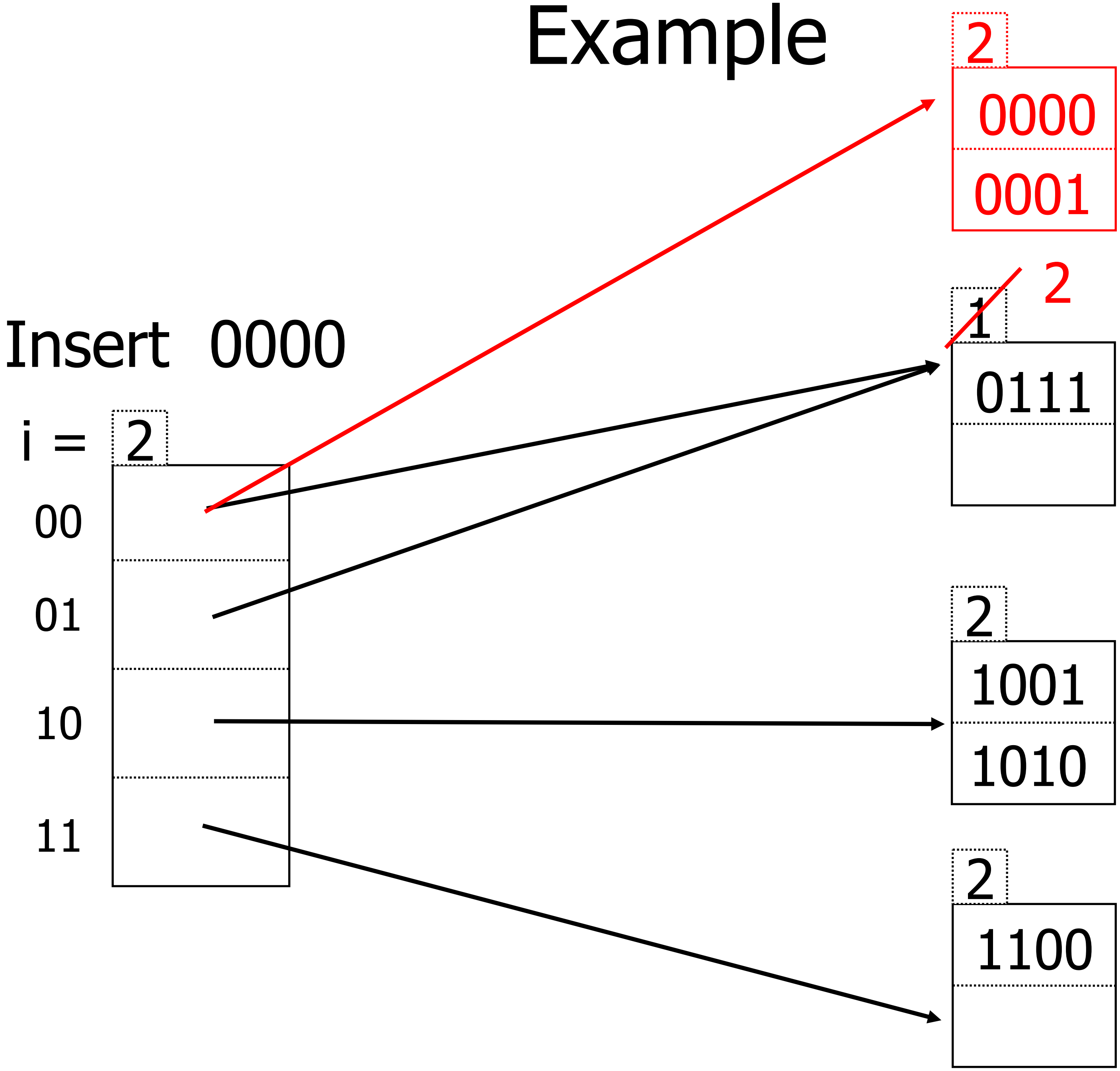
Example

Insert 0000

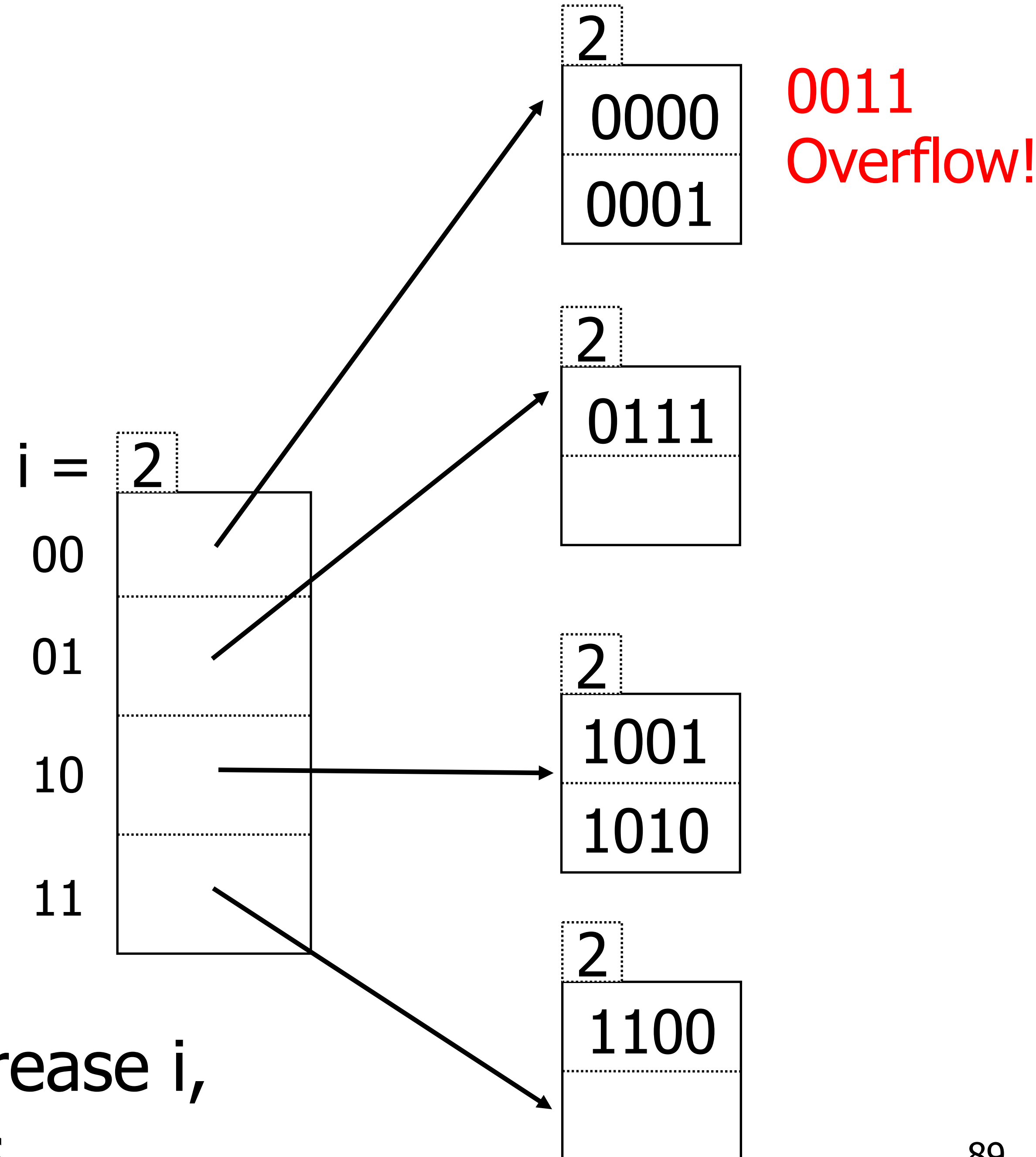


Update ptr in directory

Example

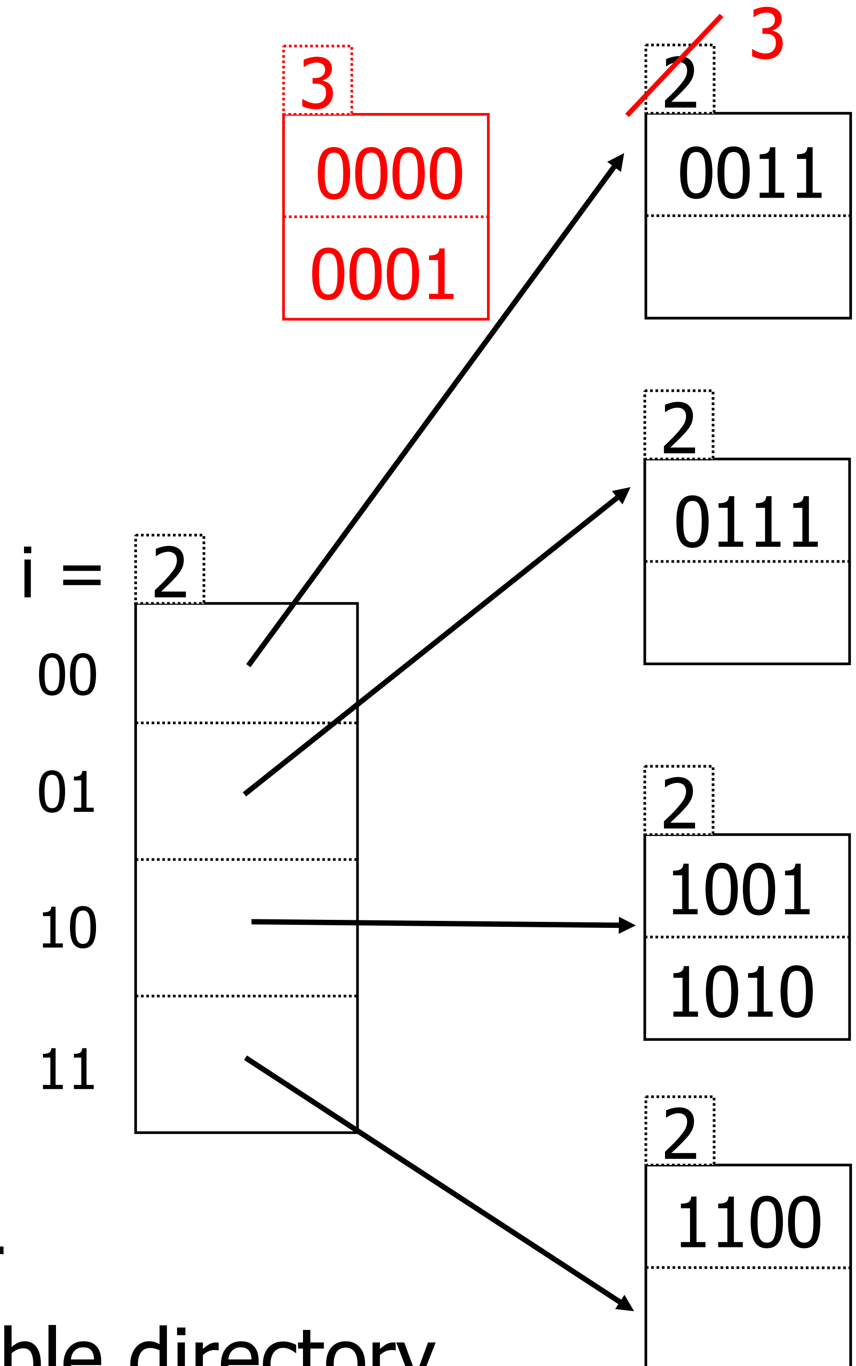


Insert 0011



Split bucket, increase i, redistribute keys

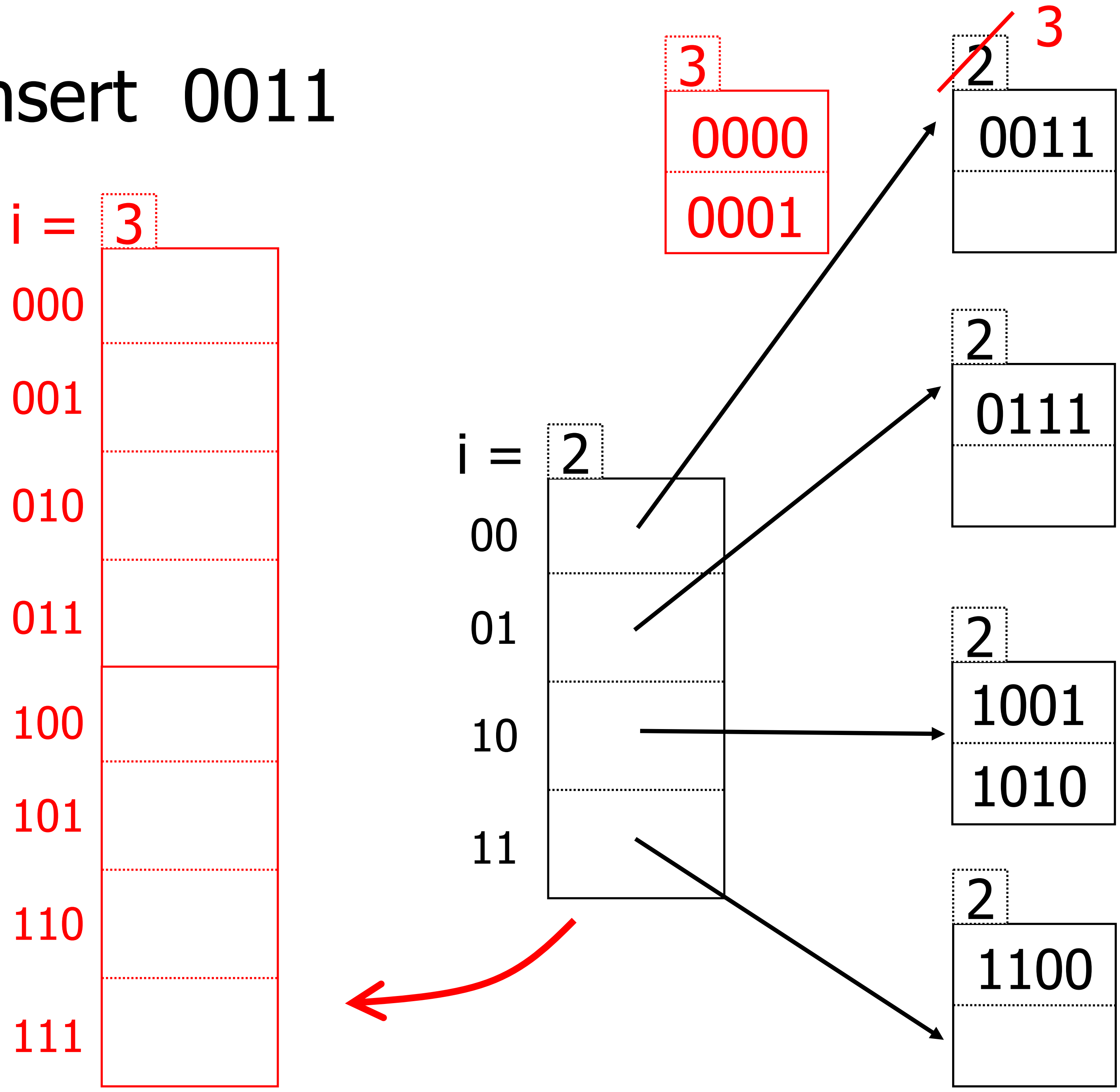
Insert 0011



Update ptr in dir

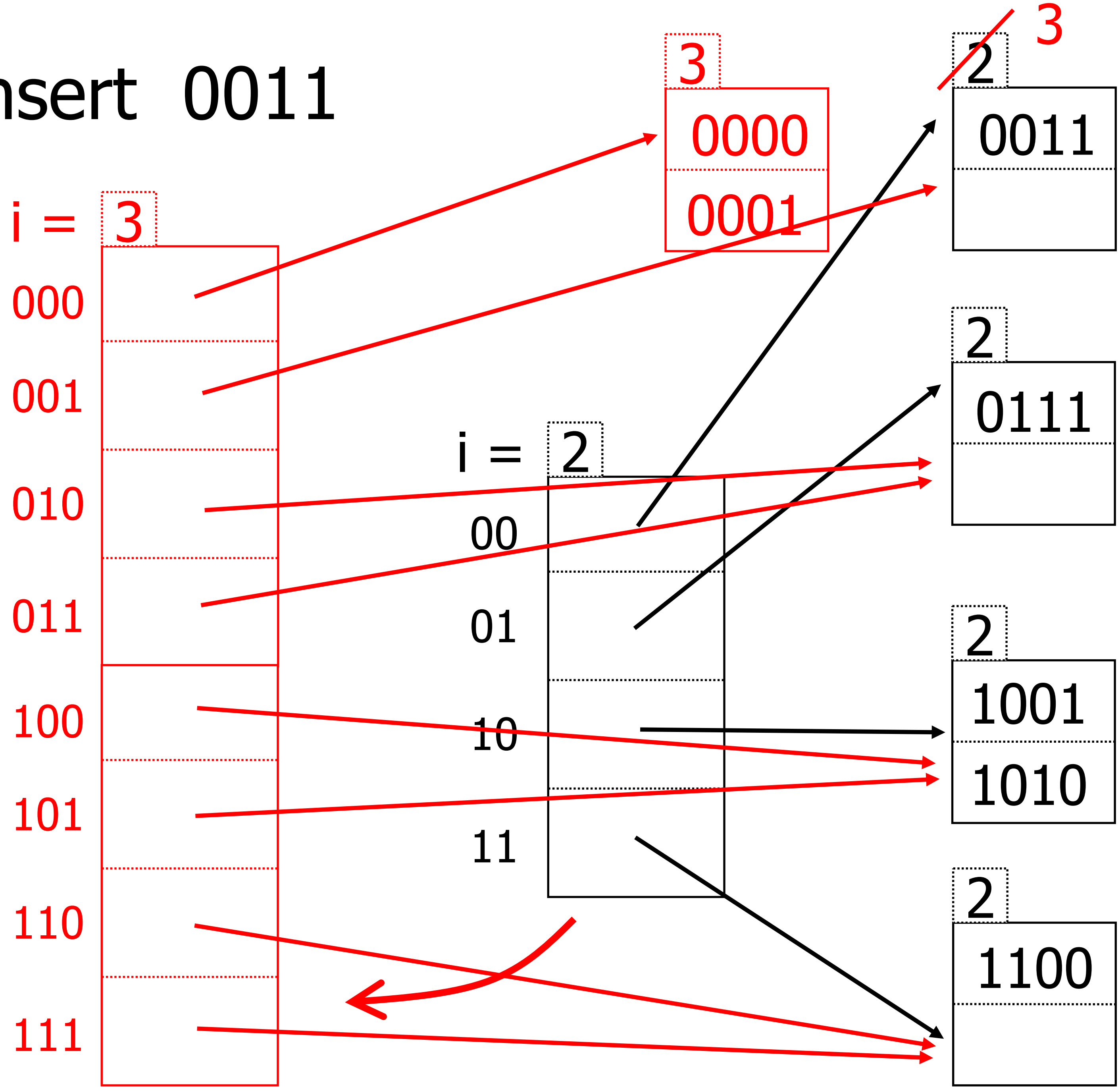
If no space, double directory

Insert 0011



i = 3
000
001
010
011
100
101
110
111
i = 2
00
01
10
11
3
0000
0001
~~2~~ 3
0011
2
0111
2
1001
1010
2
1100

Insert 0011

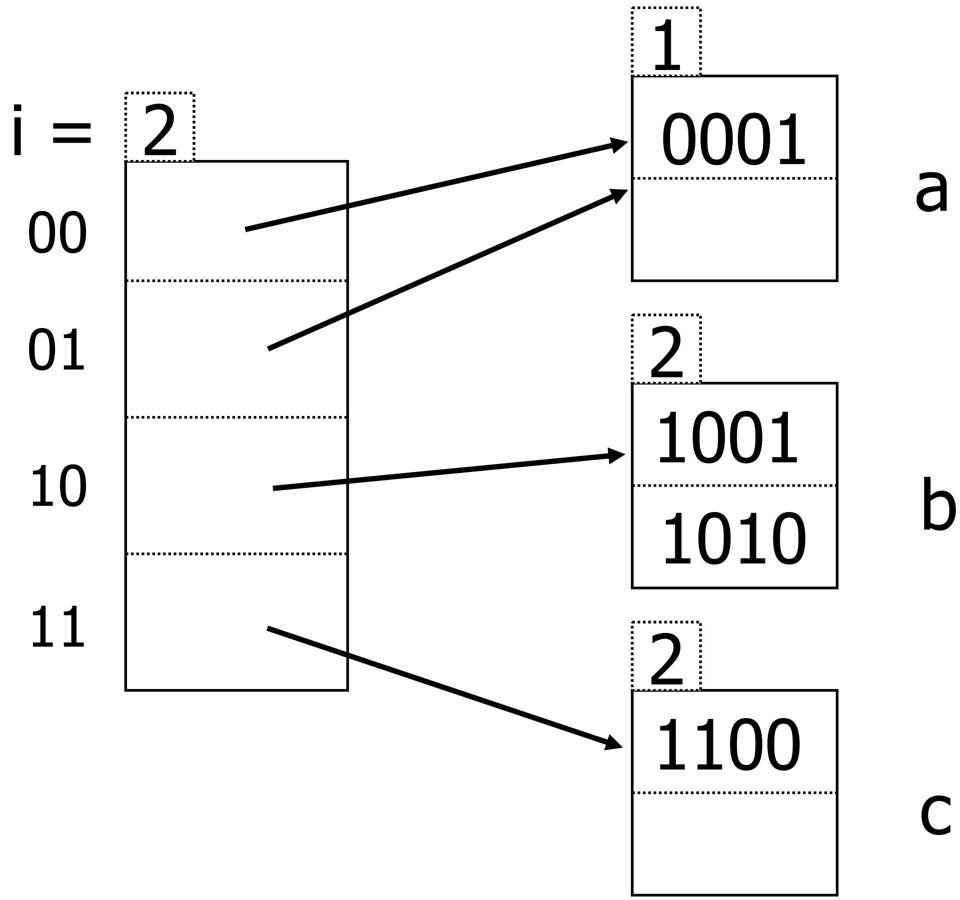


i = 3
000
001
010
011
100
101
110
111
3
0000
0001
~~2~~ 3
0011
i = 2
00
01
10
11
2
0111
2
1001
1010
2
1100

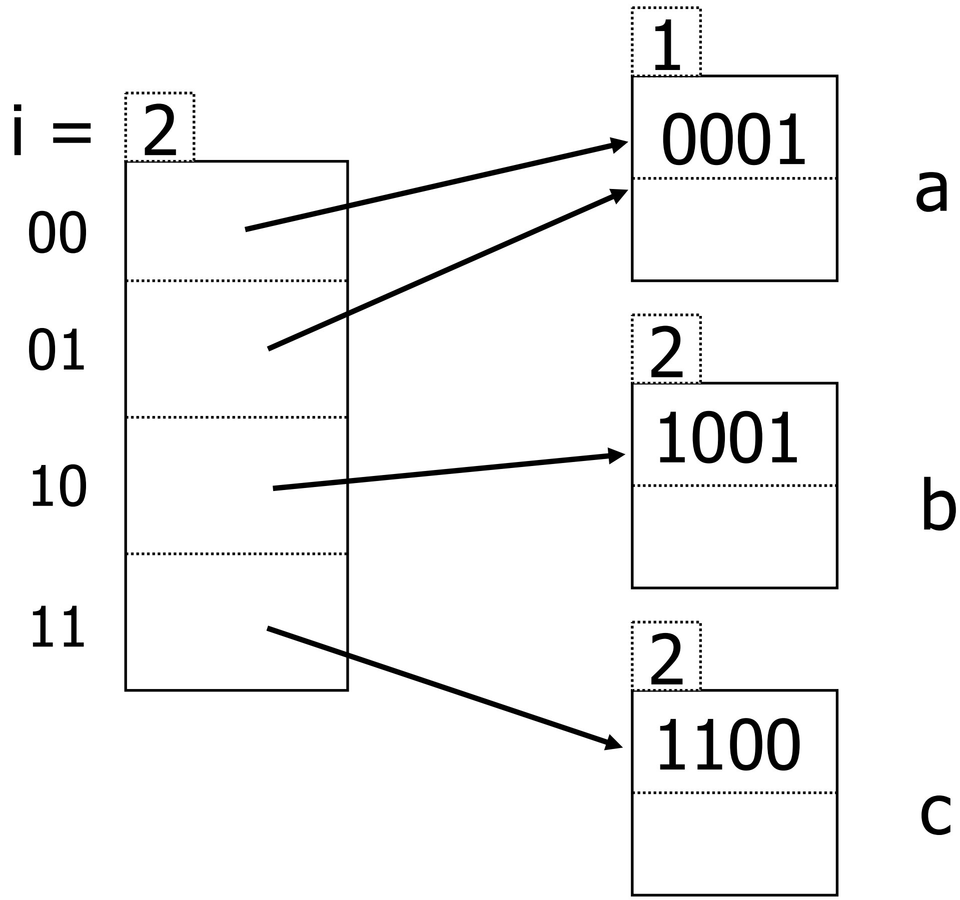
Extendible Hashing: Deletion

- Two options
  a) No merging of buckets
  b) Merge buckets and shrink directory if possible

Delete 1010

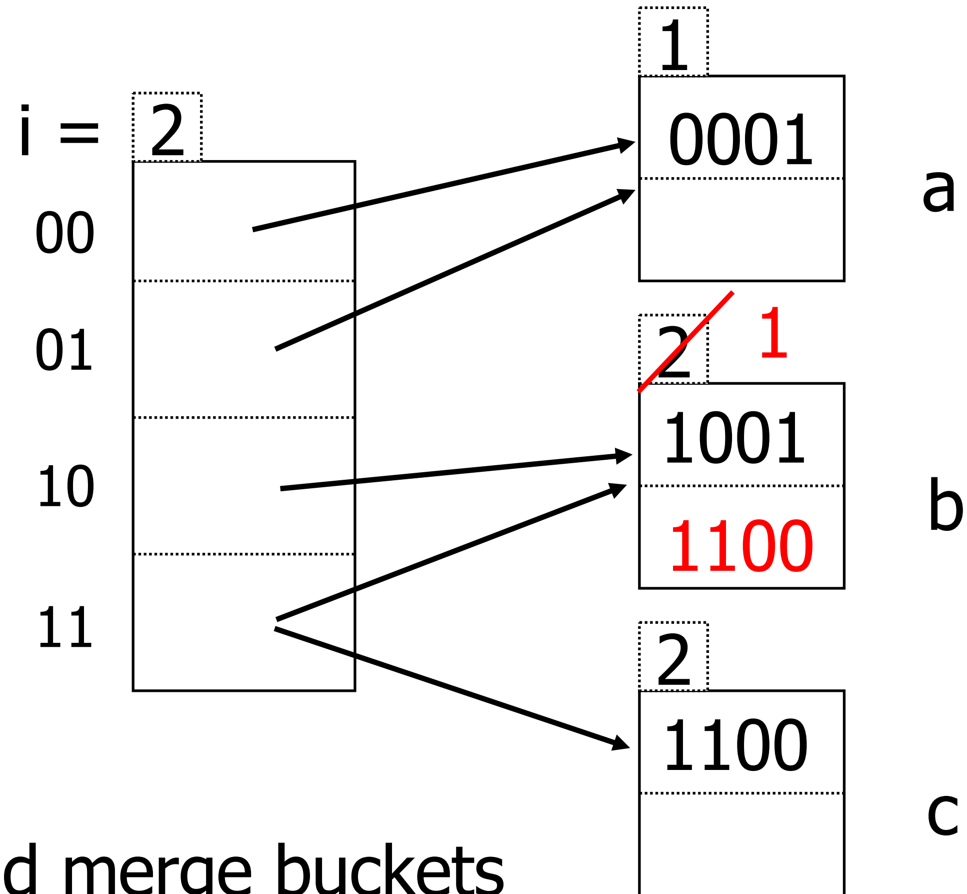


Delete 1010



- Can we merge a and b? b and c?

Delete 1010

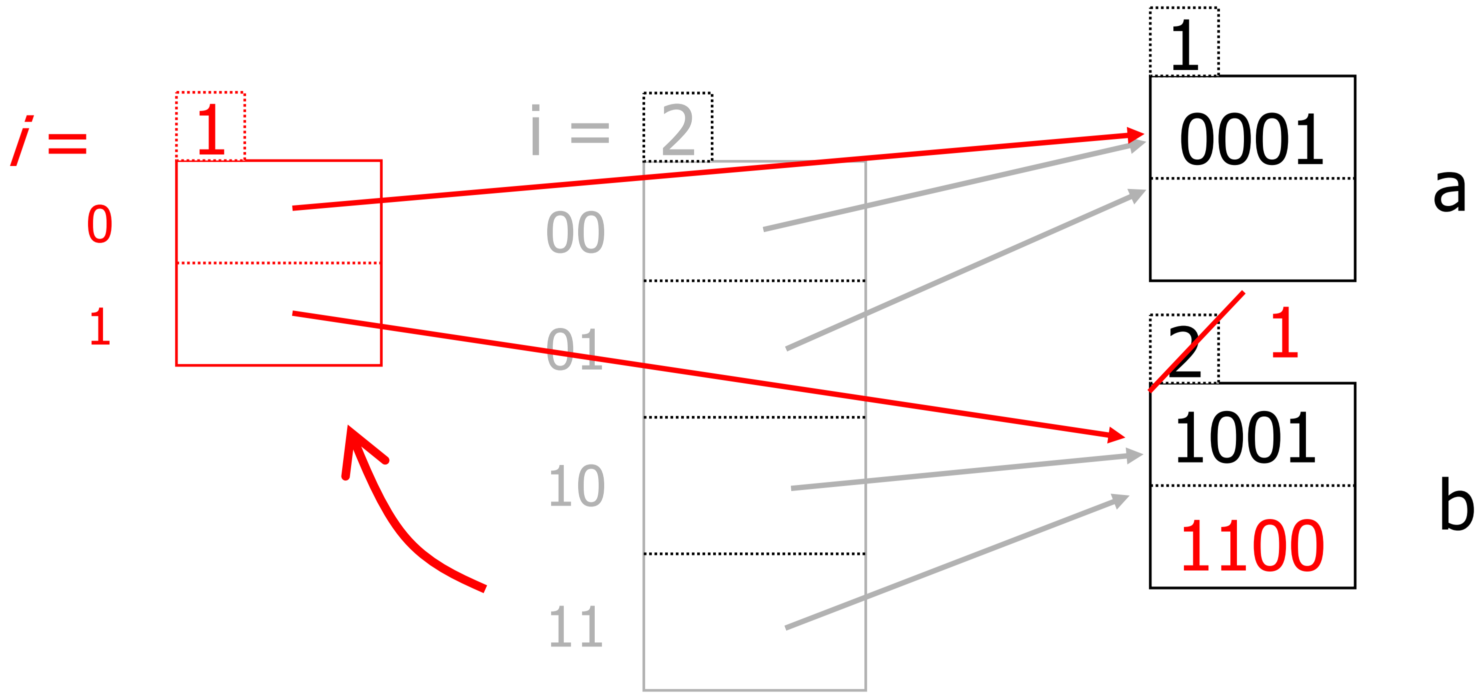


Decrease i and merge buckets

Update ptr in directory

Q: Can we shrink directory?

Delete 1010



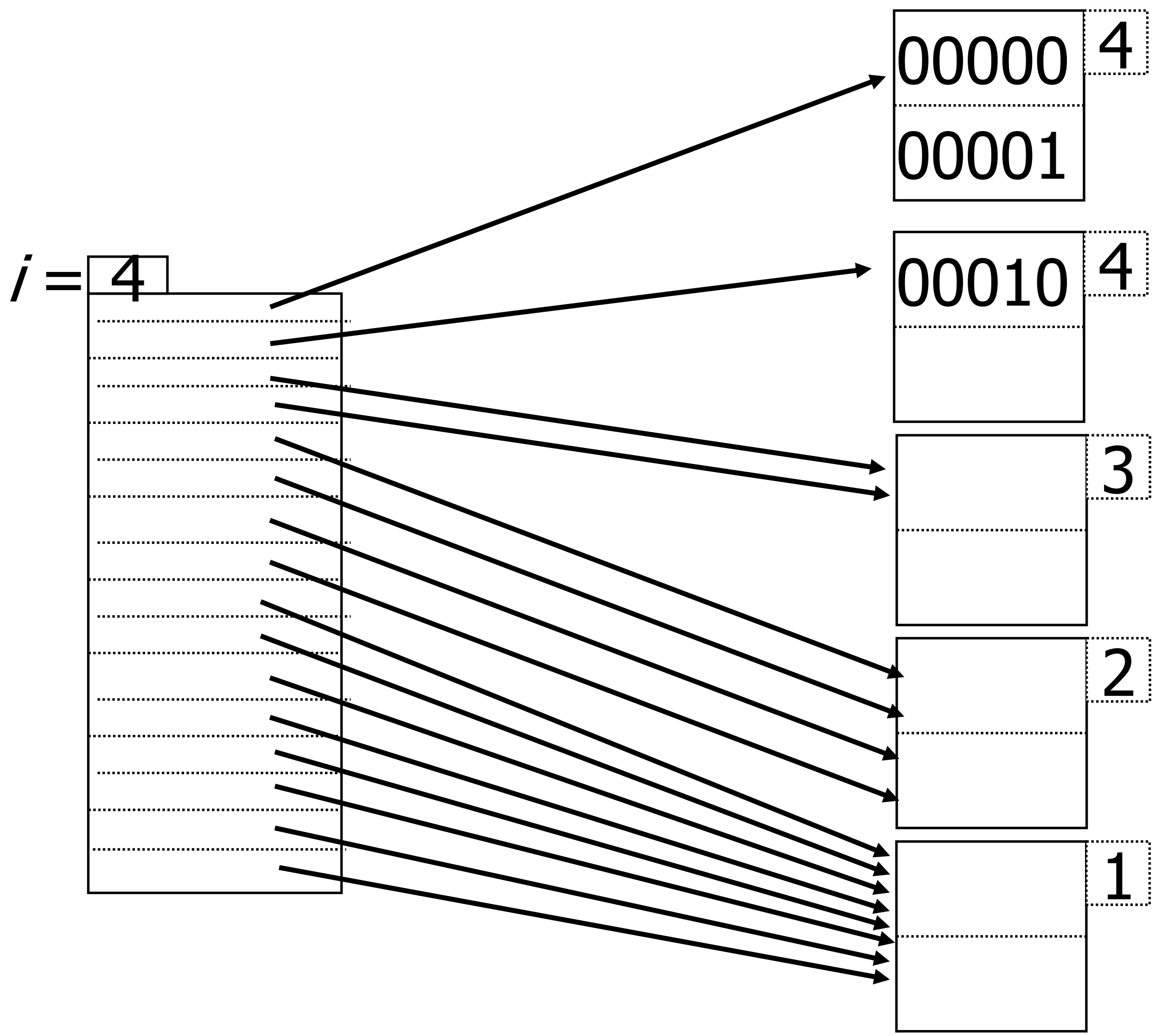
Bucket Merge Condition

- Bucket merge condition
  - Bucket i's are the same
  - First (i-1) bits of the hash key are the same
- Directory shrink condition
  - All bucket i's are smaller than the directory i

Questions on Extendible Hashing

- Can we provide minimum space guarantee?

Space Waste

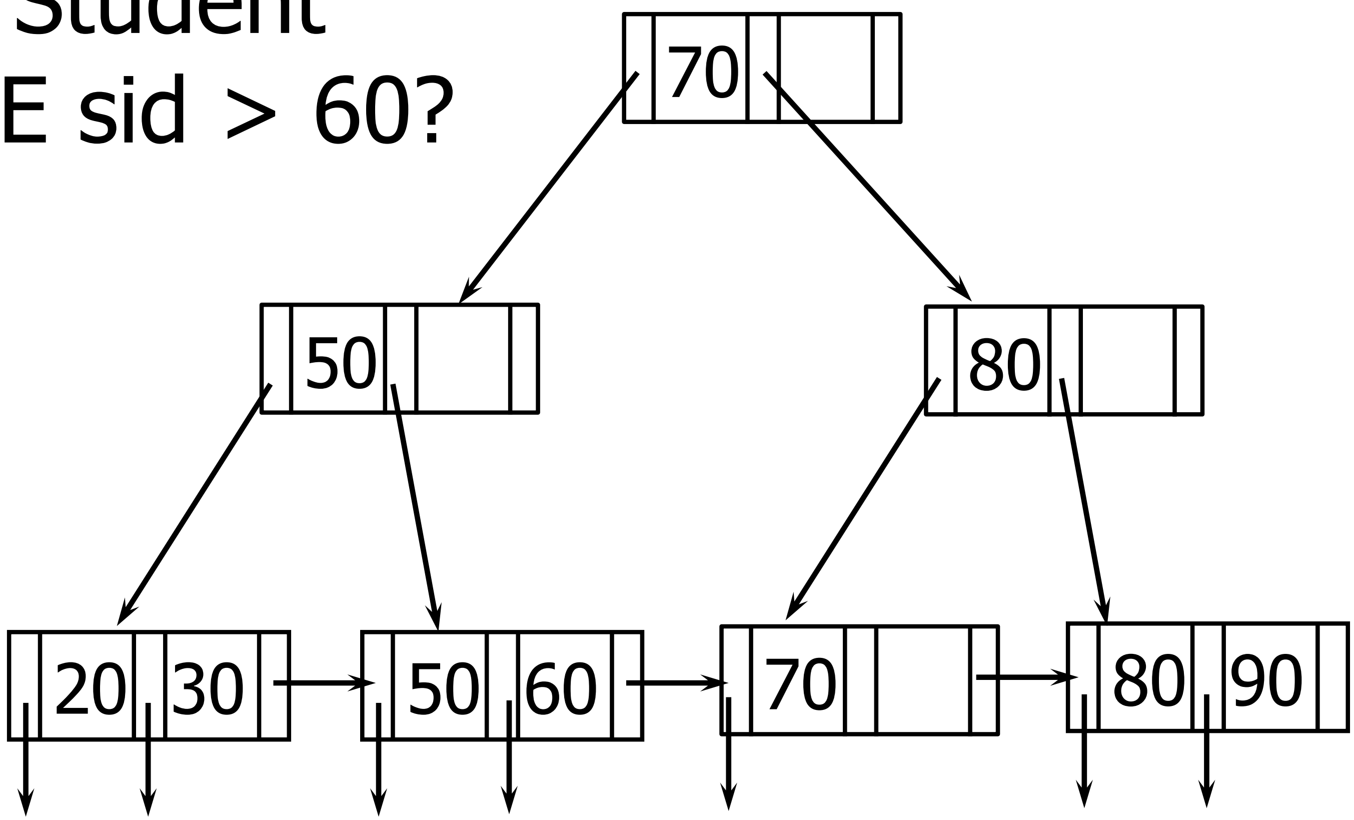


Hash index summary

- Static hashing
  - Overflow and chaining
- Extendible hashing
  - Can handle growing files
    - No periodic reorganizations
  - Indirection
    - Up to 2 disk accesses to access a key
  - Directory doubles in size
    - Not too bad if the data is not too large

Question on B+tree

- SELECT *
  FROM Student
  WHERE sid > 60?



Hashing vs. Tree

- Can an extendible-hash index support?

```
SELECT *
FROM R
WHERE R.A > 5
```

- Which one is better, B+tree or Extendible hashing?

```
SELECT *
FROM R
WHERE R.A = 5
```