CSE221

Lecture 17:
Multiway Search Trees

Hyungon Moon

Acknowledgment: The content of this file is based on the slides of
the textbook as well as the slides provided by Prof. Won-Ki Jeong.

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

Outline

- m-way search trees
- B-trees
- B^+-trees

Outline

- m-way search trees
- B-trees
- B^+-trees

Memory Hierarchy

- Von Neumann model limitation
  - Memory is bottleneck
- Memory hierarchy
  - Register – cache – memory – disk
- Overall performance is closely related to reducing the access to slow memory

Reduce Memory Access

- Number of memory accesses is closely tied to the ***height*** of the search tree
- Height-balanced binary search tree has $\log_2 n$ height
- Can we break $\log_2 n$ barrier?

→Allow a node to have more than 2, up to m children.

m-way Search Trees

- Root has at least two & at most m subtrees
- Node structure (n<m)

n	A_0	E_1	A_1	E_2	…	E_n	A_n

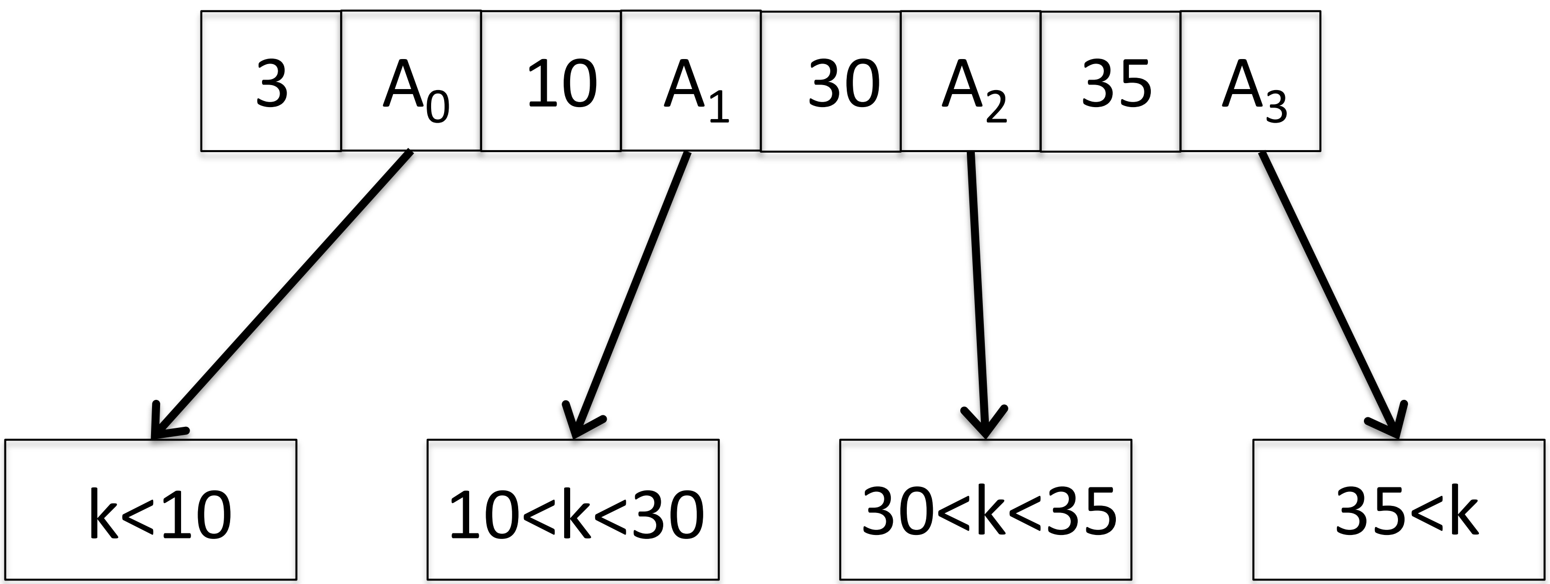
# of element pointer to subtree element

- $E_i.K < E_{i+1}.K$ (key)
- $E_i.K <$ all keys in $A_i < E_{i+1}.K$

Tree is ordered!

- Subtrees A_i are also m-way search trees (recursive definition)

Example: 4-way Search Tree



m-way Search Trees

- Maximum # of nodes happens when all internal nodes are m-nodes (having m subtrees)
  - A full tree with degree m.
- Max # of nodes in a tree of degree m and height h
  - $1 + m + m^2 + \ldots + m^h = \frac{m^h - 1}{m - 1}$
- Each node has $m - 1$ elements
- So, max # of elements: $m^h - 1$

Searching

```
// Search m-way search tree for an element with key x
E0.k=-MAXKEY;
for(p=root; p; p=Ai)
{
    Let p is a node (n, A0, E1, A1, .. , En, An)
    En+1.k = MAXKEY
    Determine i such that Ei.K <= x < Ei+1.K;
    if(x == Ei.K) return Ei; // x is found
}
// x is not found
return NULL;
```

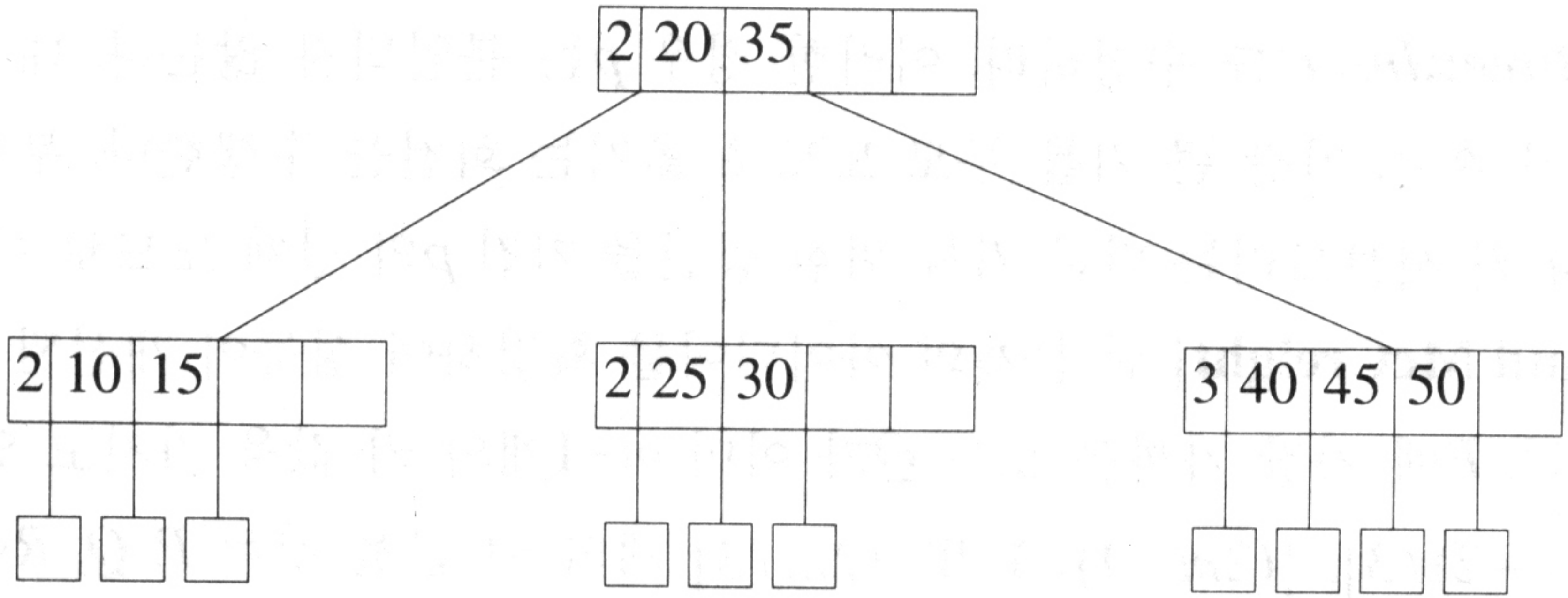
Outline

- m-way search trees
- B-trees
- B^+-trees

B-trees

- Extended m-way search trees by addition of external nodes
  - Replace a NULL pointer to an external node
- Definition
  - If not empty, root node has at least two children
  - All internal nodes (except root) have at least $\left\lceil \frac{m}{2} \right\rceil$ children.
  - All external nodes are at the same level
- *Balanced* m-way search tree

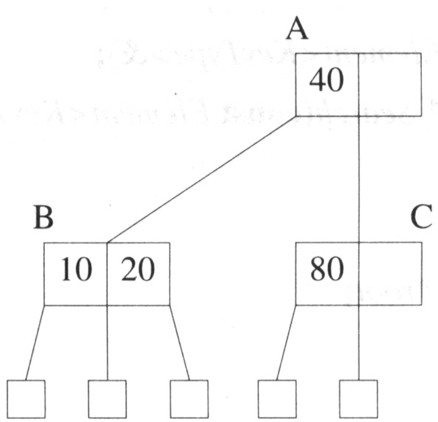
Example



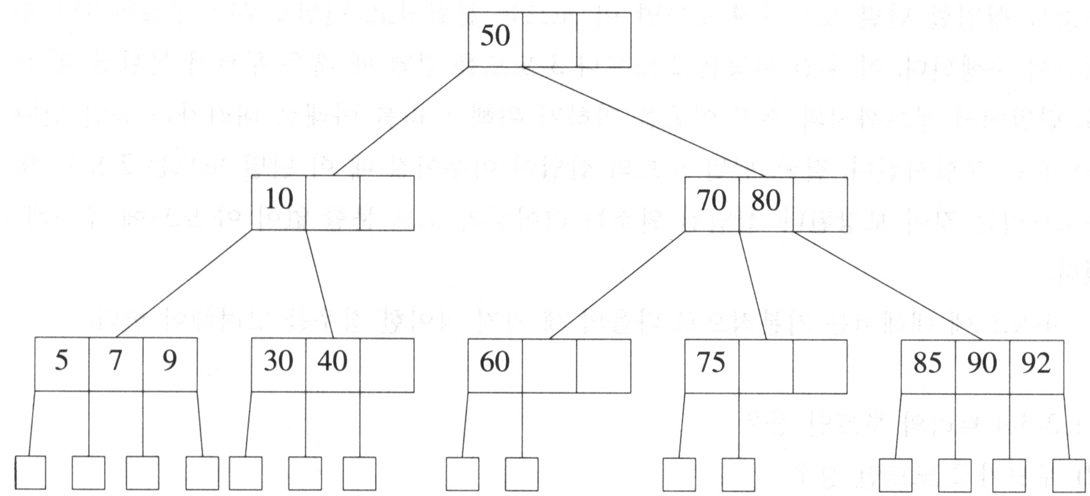
5-way B-tree example, $\left\lceil \frac{5}{2} \right\rceil = 3$

2-3 and 2-3-4 Trees

- 2-3 tree is B-tree of order 3
- 2-3-4 tree is B-tree of order 4
    - Also called (2,4) tree or 2-4 tree



2-3 tree



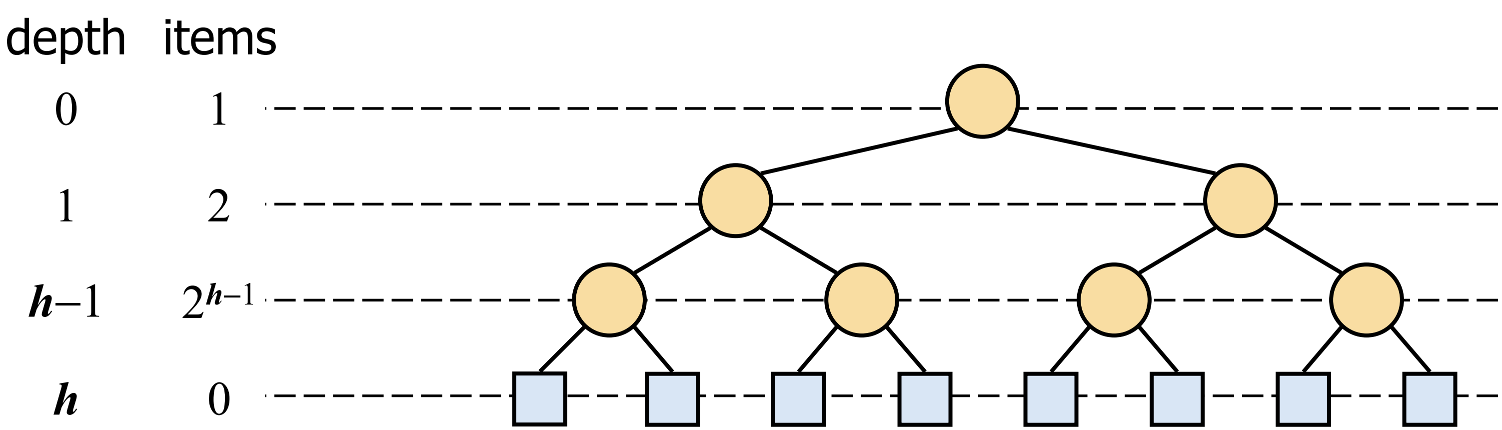
2-3-4 tree

Height of a (2,4) Tree

- Theorem: A (2,4) tree storing $\boldsymbol{n}$ items has height $\mathbf{O}(\log \boldsymbol{n})$

  Proof:
  - Let $\boldsymbol{h}$ be the height of a (2,4) tree with $\boldsymbol{n}$ items
  - Since there are at least $2^{\boldsymbol{i}}$ items at depth $\boldsymbol{i} = 0, \ldots, \boldsymbol{h} - 1$ and no items at depth $\boldsymbol{h}$, we have

    $$\boldsymbol{n} \geq 1 + 2 + 4 + \ldots + 2^{\boldsymbol{h}-1} = 2^{\boldsymbol{h}} - 1$$
  - Thus, $\boldsymbol{h} \leq \log(\boldsymbol{n} + 1)$
- Searching in a (2,4) tree with $\boldsymbol{n}$ items takes $\mathbf{O}(\log \boldsymbol{n})$ time



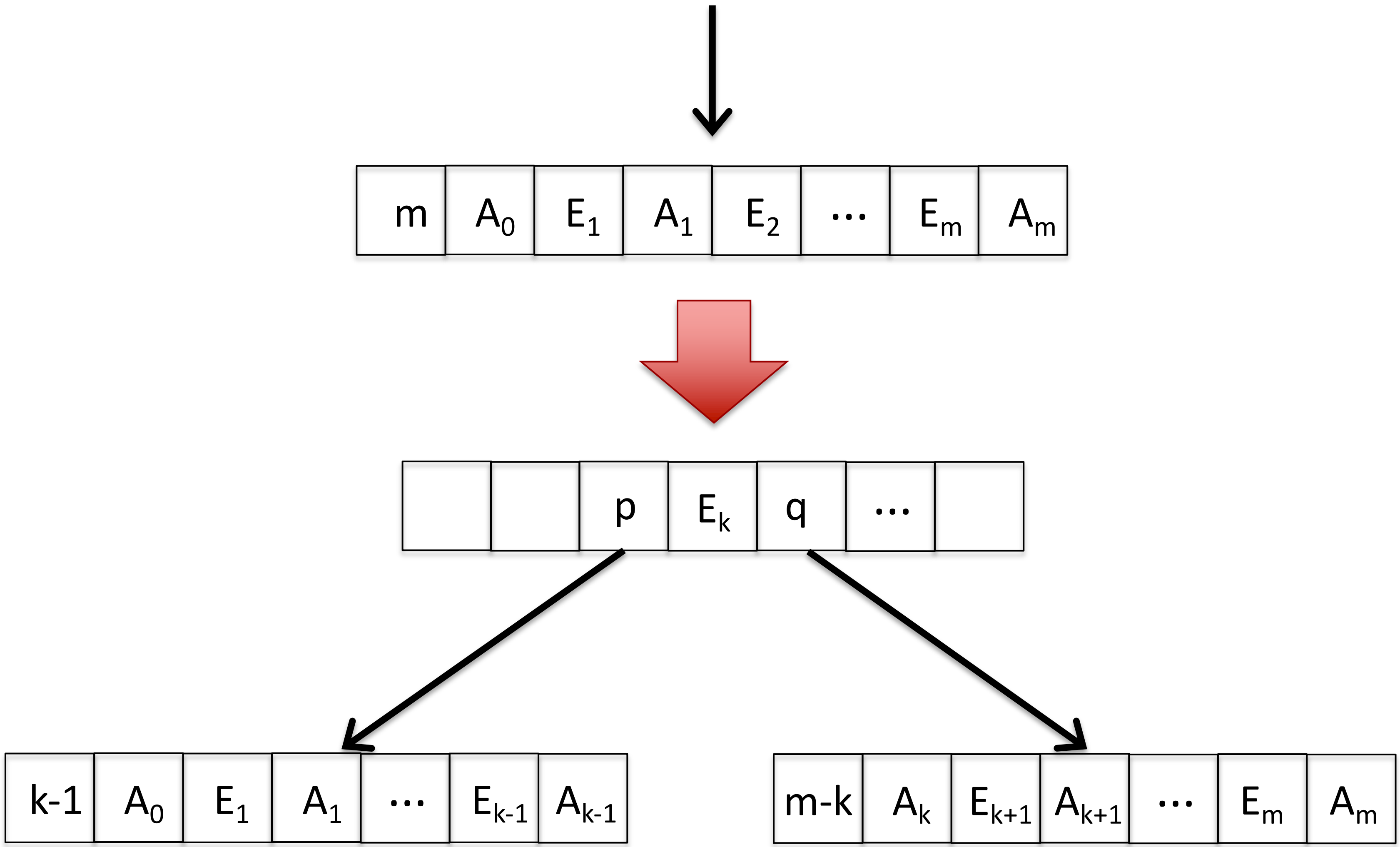
Choice of m

- Worst-case search time
  - (time to fetch a node + time to search node) * height
- Search time increases if m is too small or too large
- Pick m so that single node fits to a single memory access
  - Size of a cache line or disk block

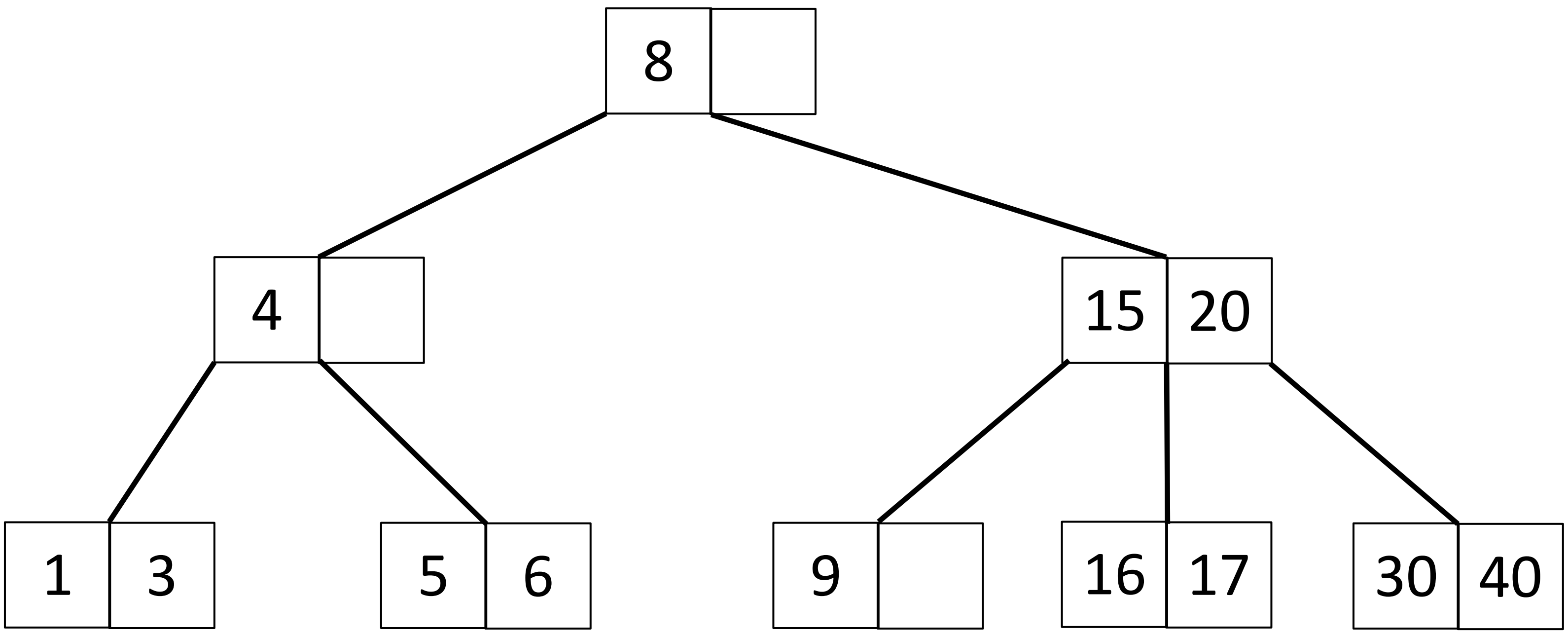
Insert

- If insertion results in m keys for m-way B-tree (overflow), split node
- Let node p have the format
  - $m, A_0, (E_1, A_1), \ldots, (E_m, A_m)$
- p is split into two nodes p and q
  - Let $k = \left\lceil \frac{m}{2} \right\rceil$
  - node p: $k-1, A_0, (E_1, A_1), \ldots, (E_{k-1}, A_{k-1})$
  - node q: $m-k, A_k, (E_{k+1}, A_{k+1}), \ldots, (E_m, A_m)$
  - (E_k, q) is inserted into the parent of p
- Splitting can propagate up to the root

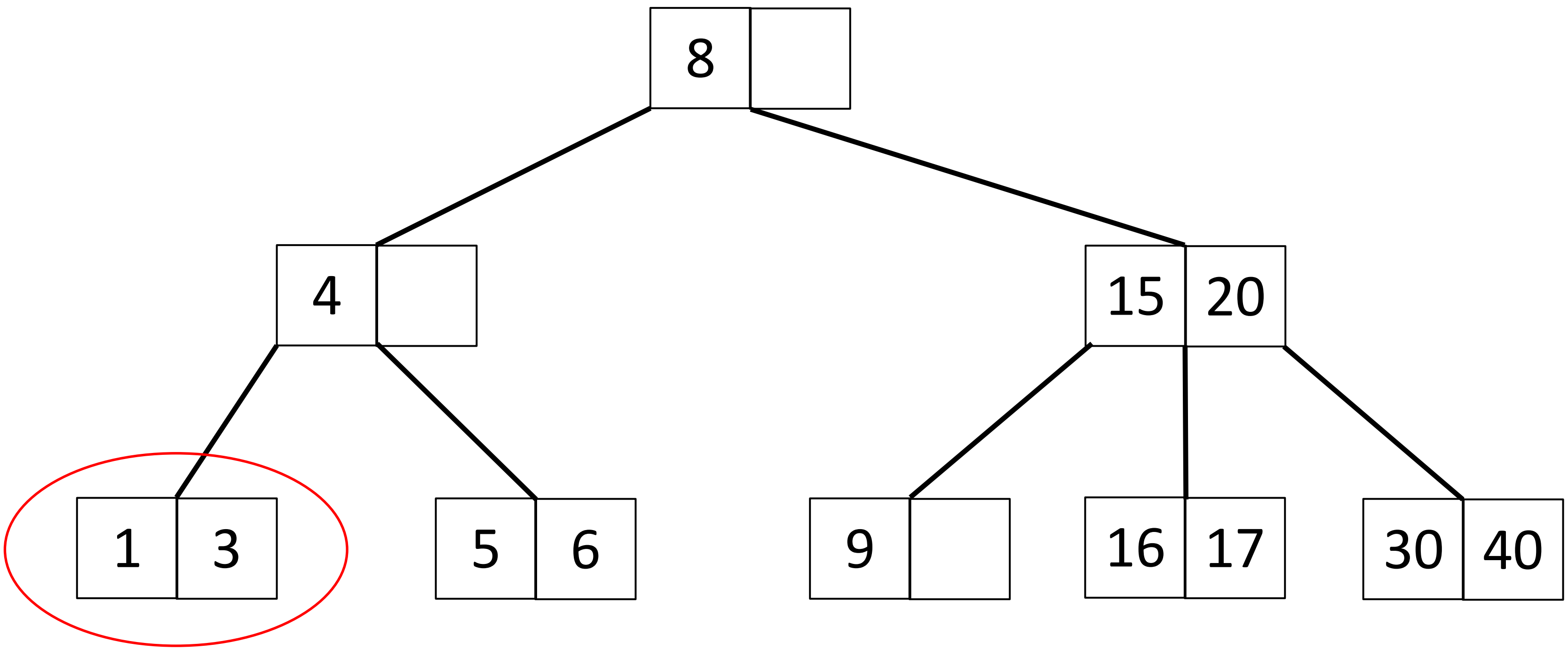
Split Node



Insert (3-way B-tree)

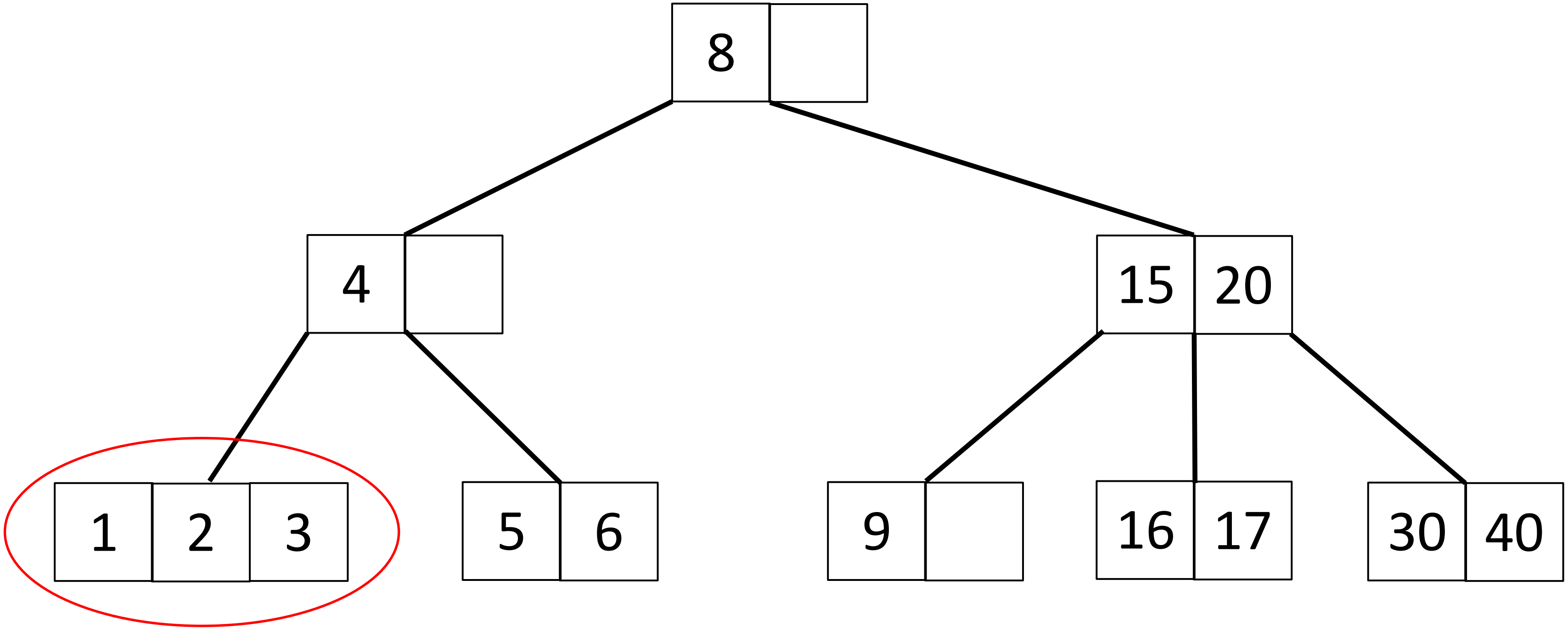


Insert (3-way B-tree)



Insert 2

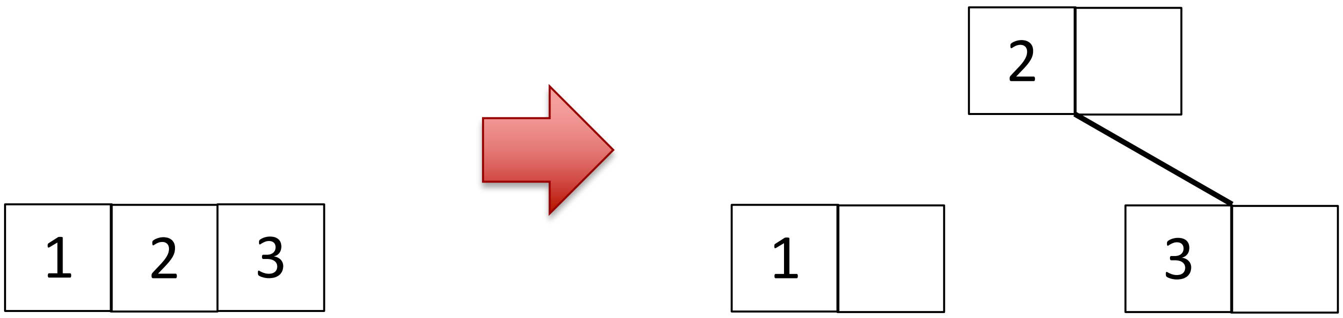
Insert (3-way B-tree)



need split!

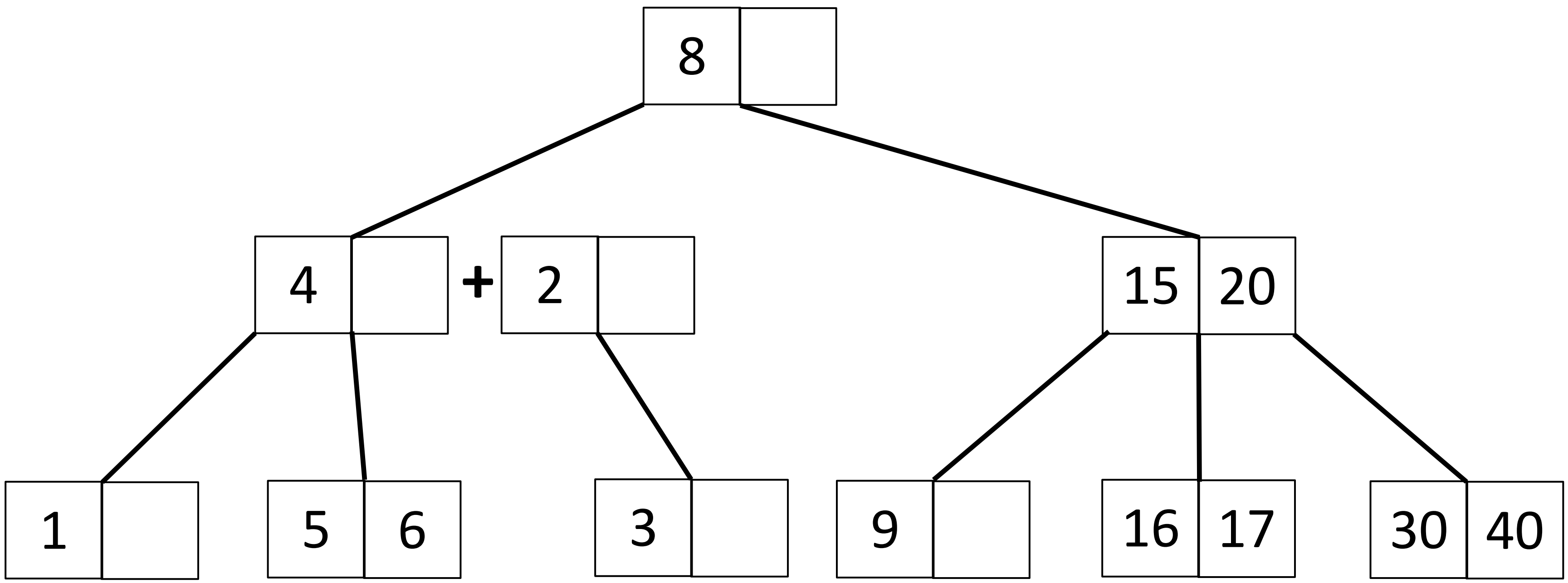
Insert (3-way B-tree)

- Split overflowed node around middle key

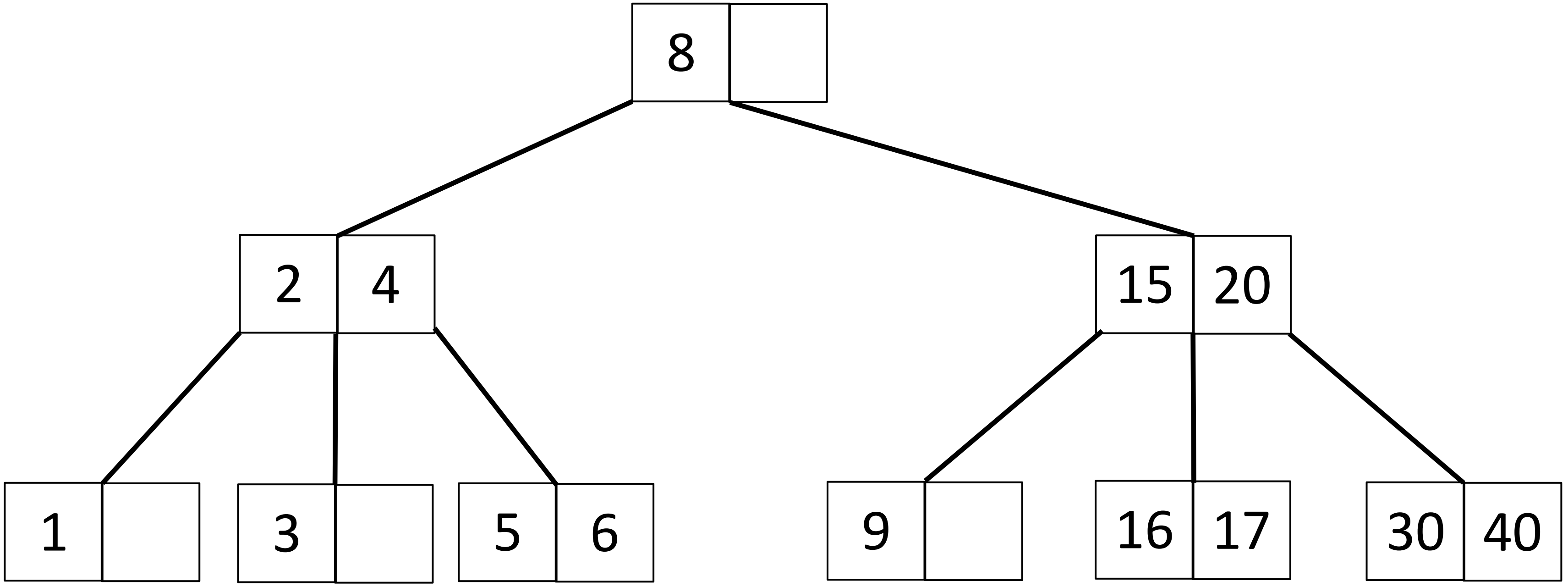


- Insert middle key to its parent

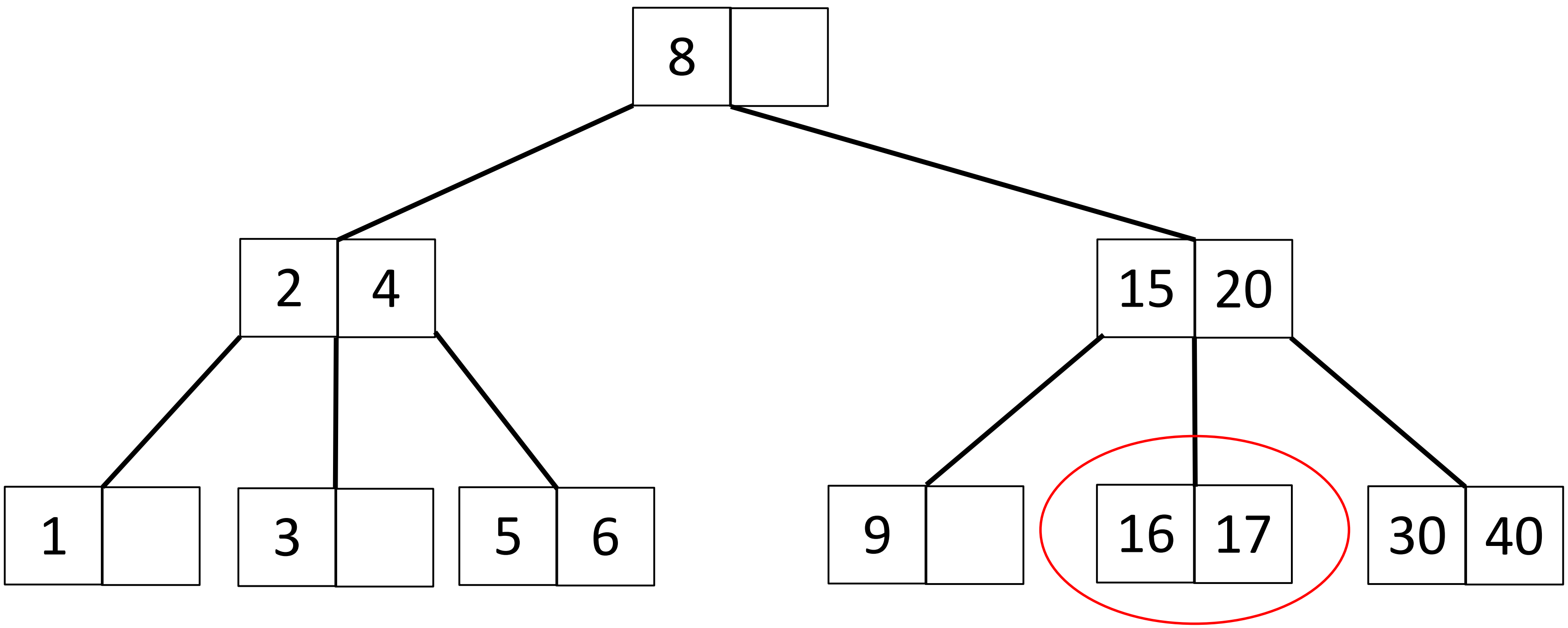
Insert (3-way B-tree)



Insert (3-way B-tree)

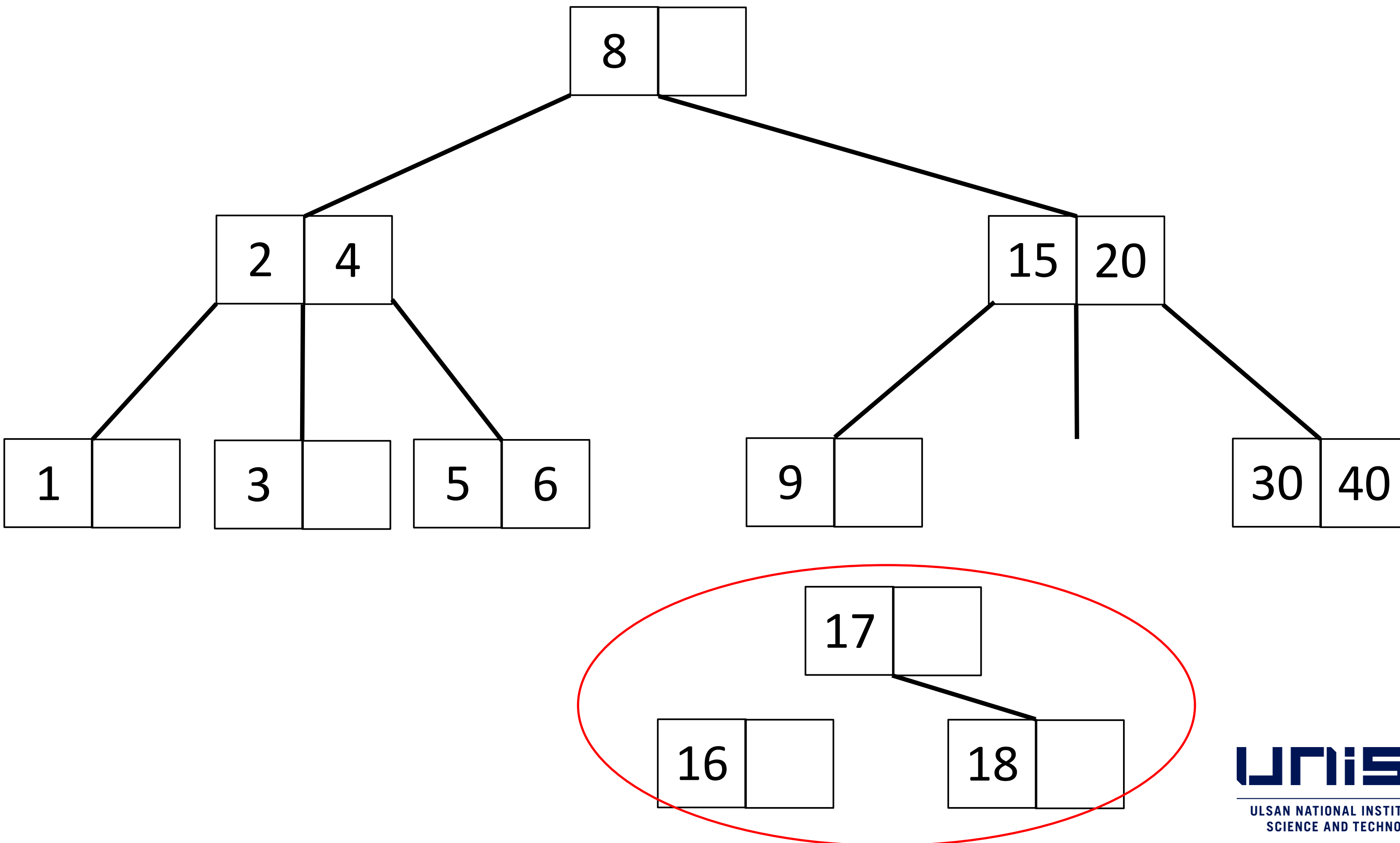


Insert (3-way B-tree)

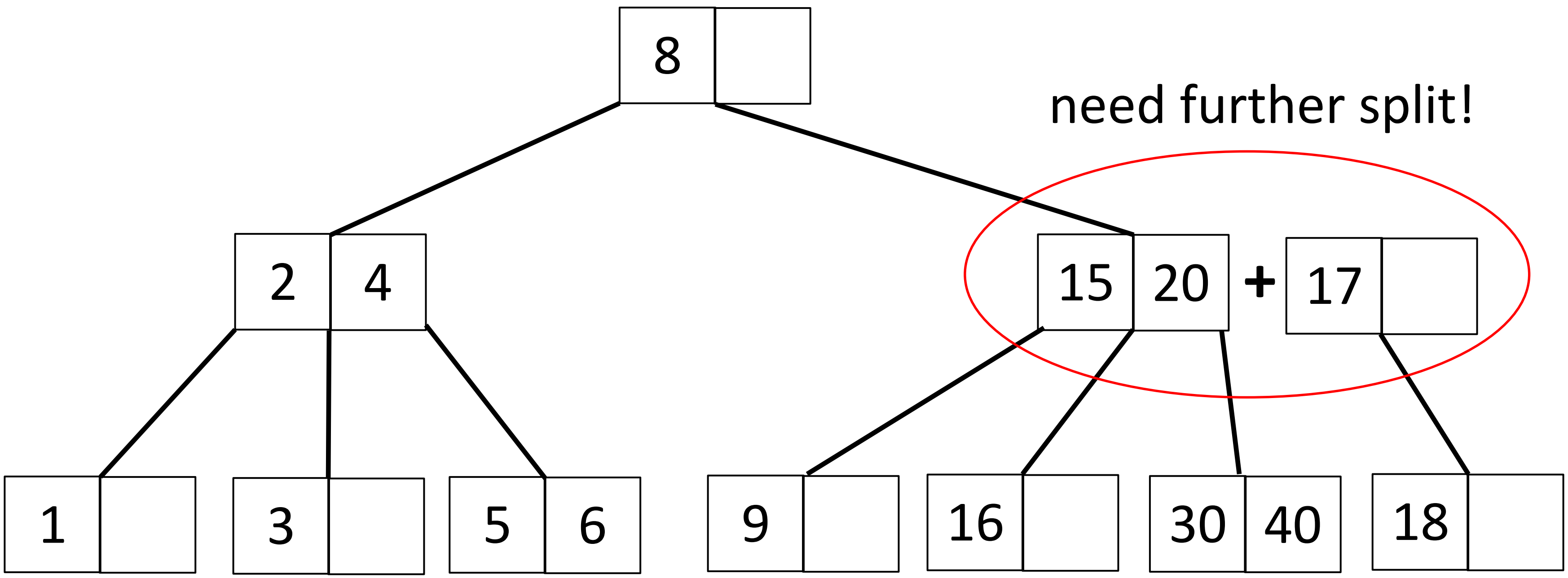


Insert 18

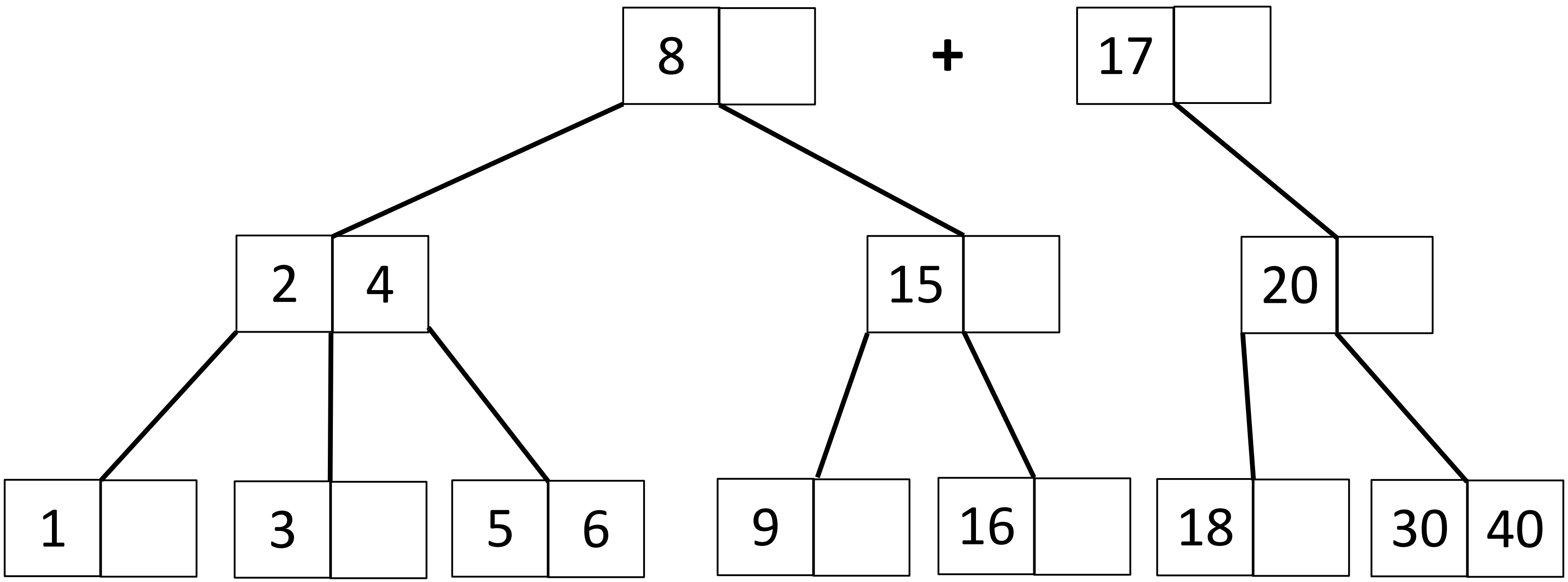
Insert (3-way B-tree)



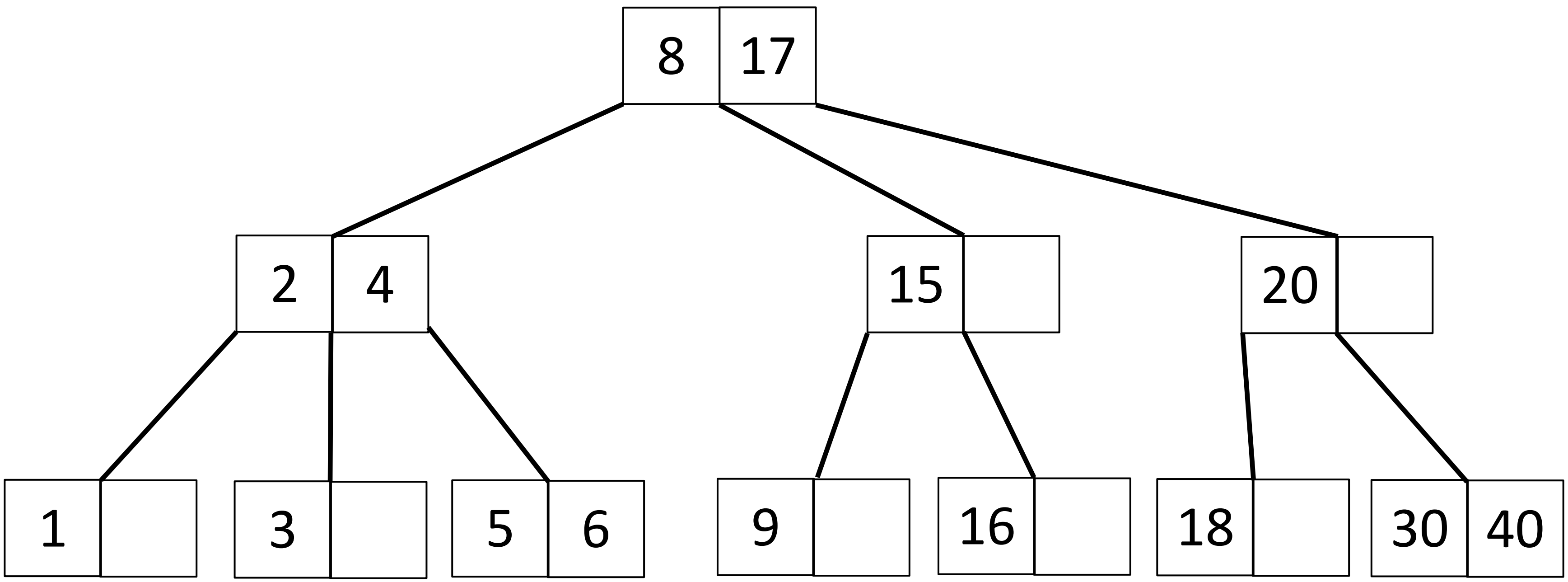
Insert (3-way B-tree)



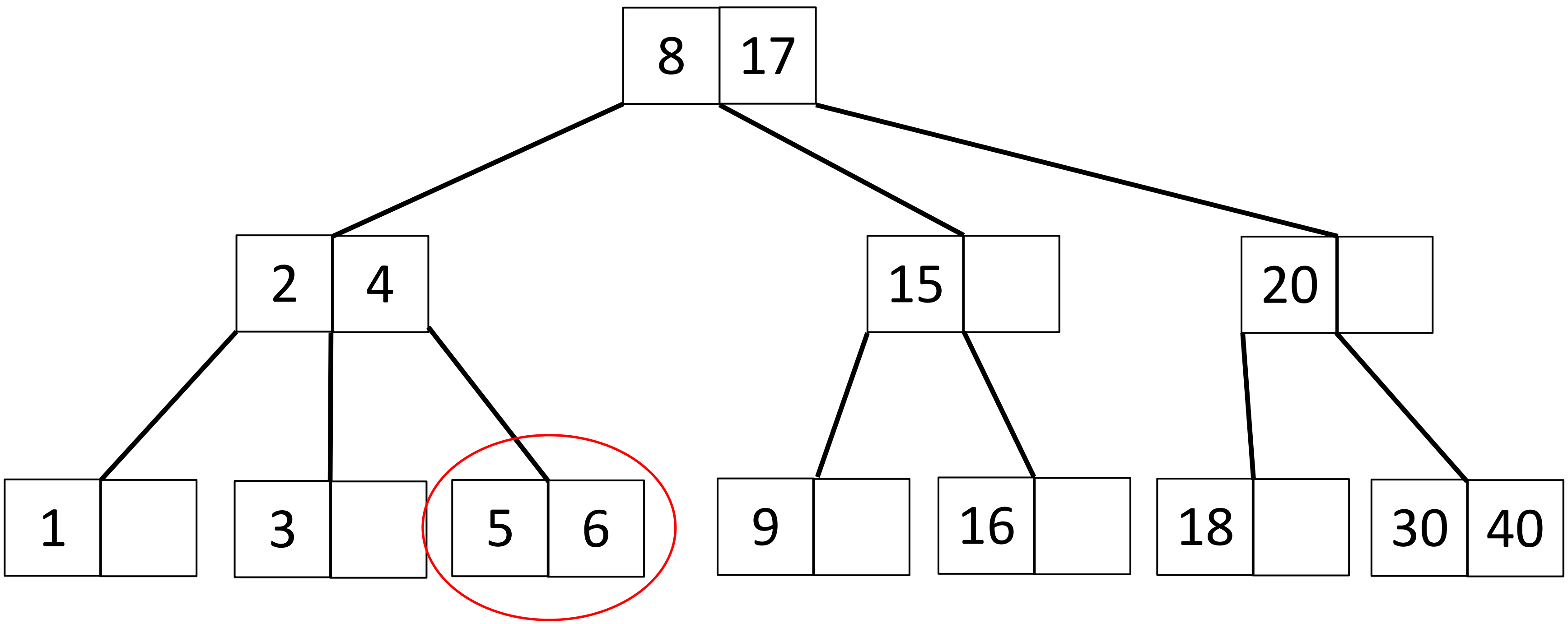
Insert (3-way B-tree)



Insert (3-way B-tree)

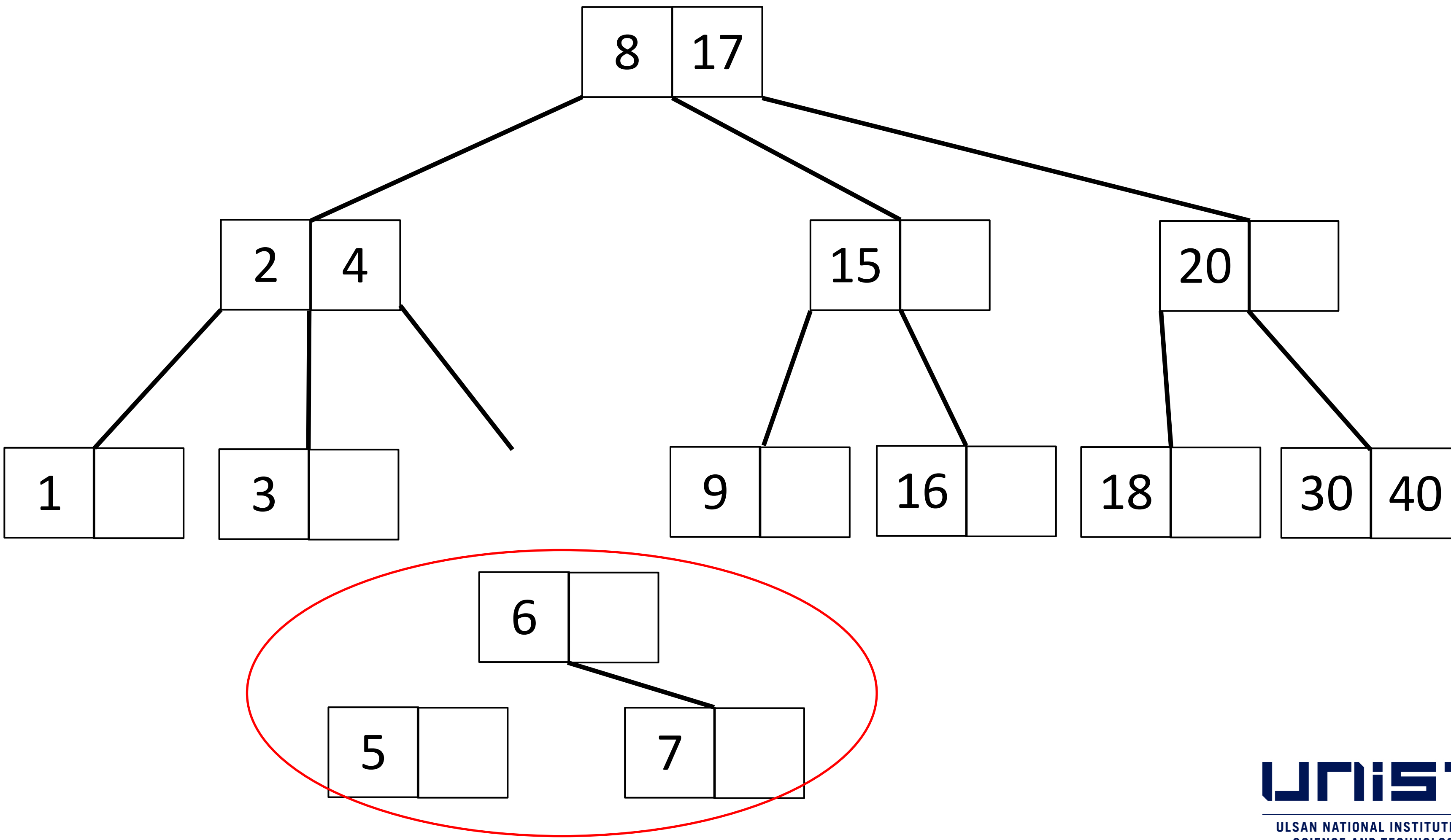


Insert (3-way B-tree)

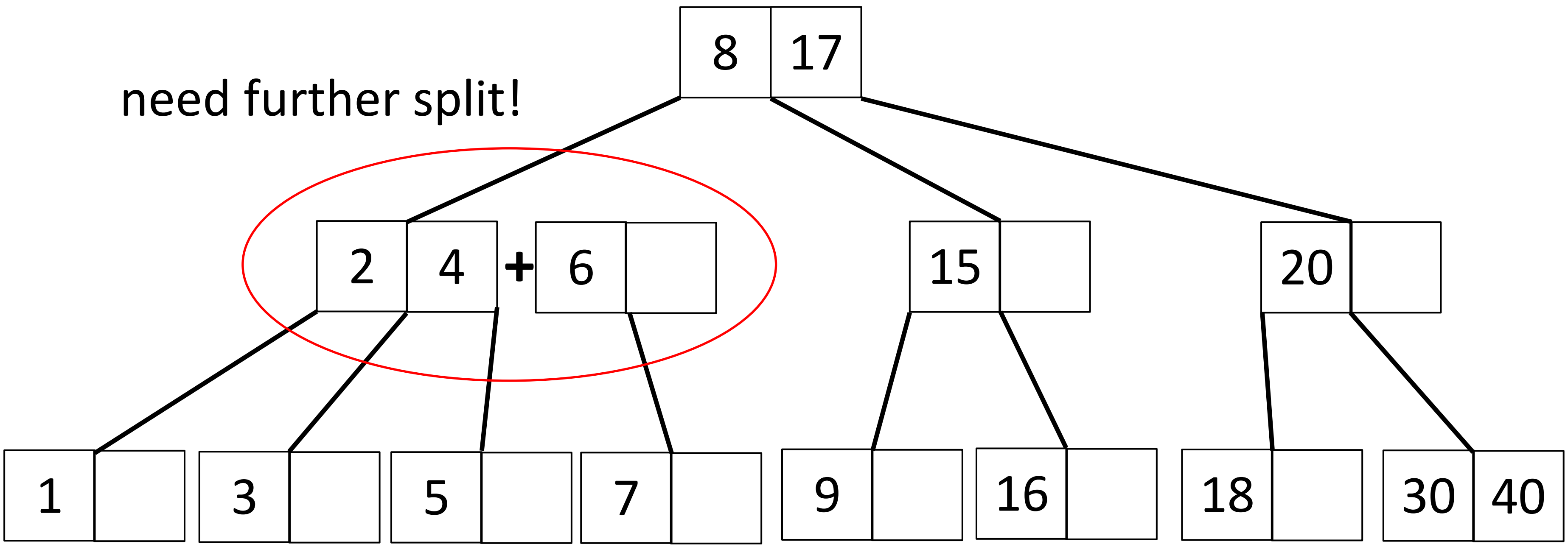


Insert 7

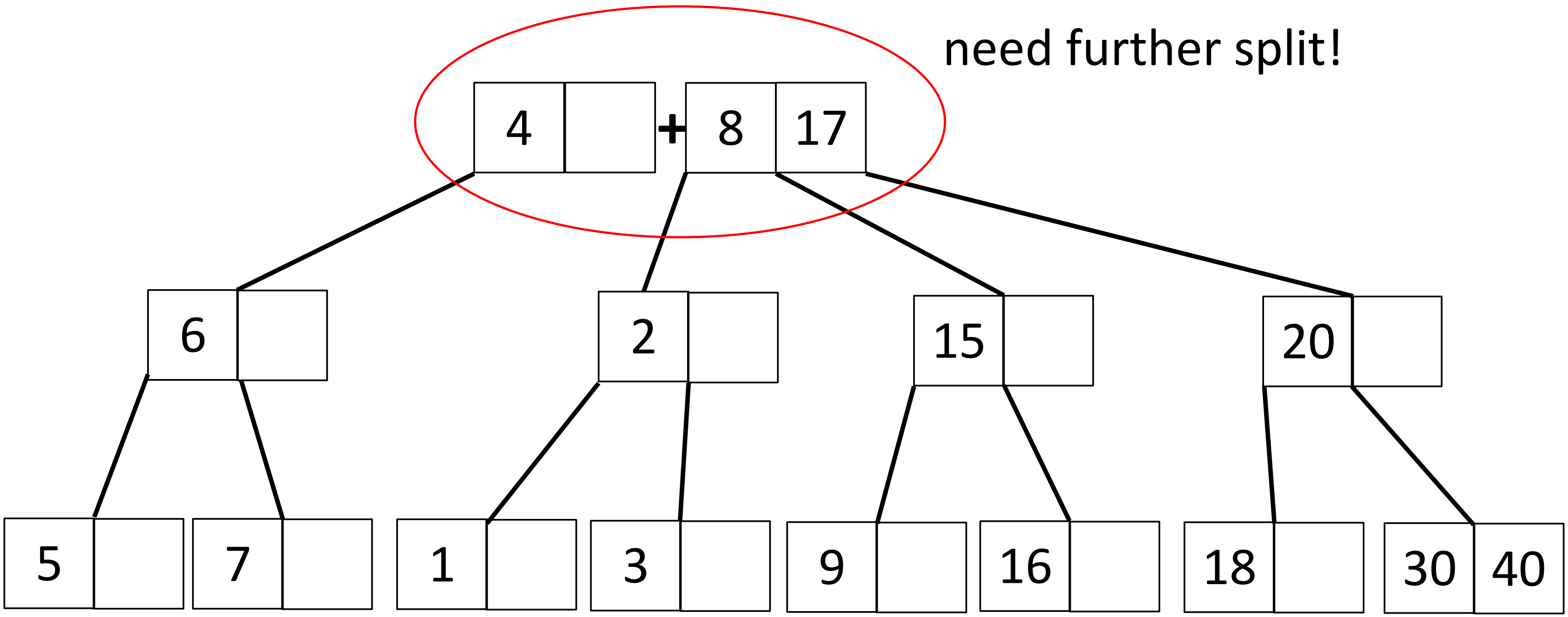
Insert (3-way B-tree)



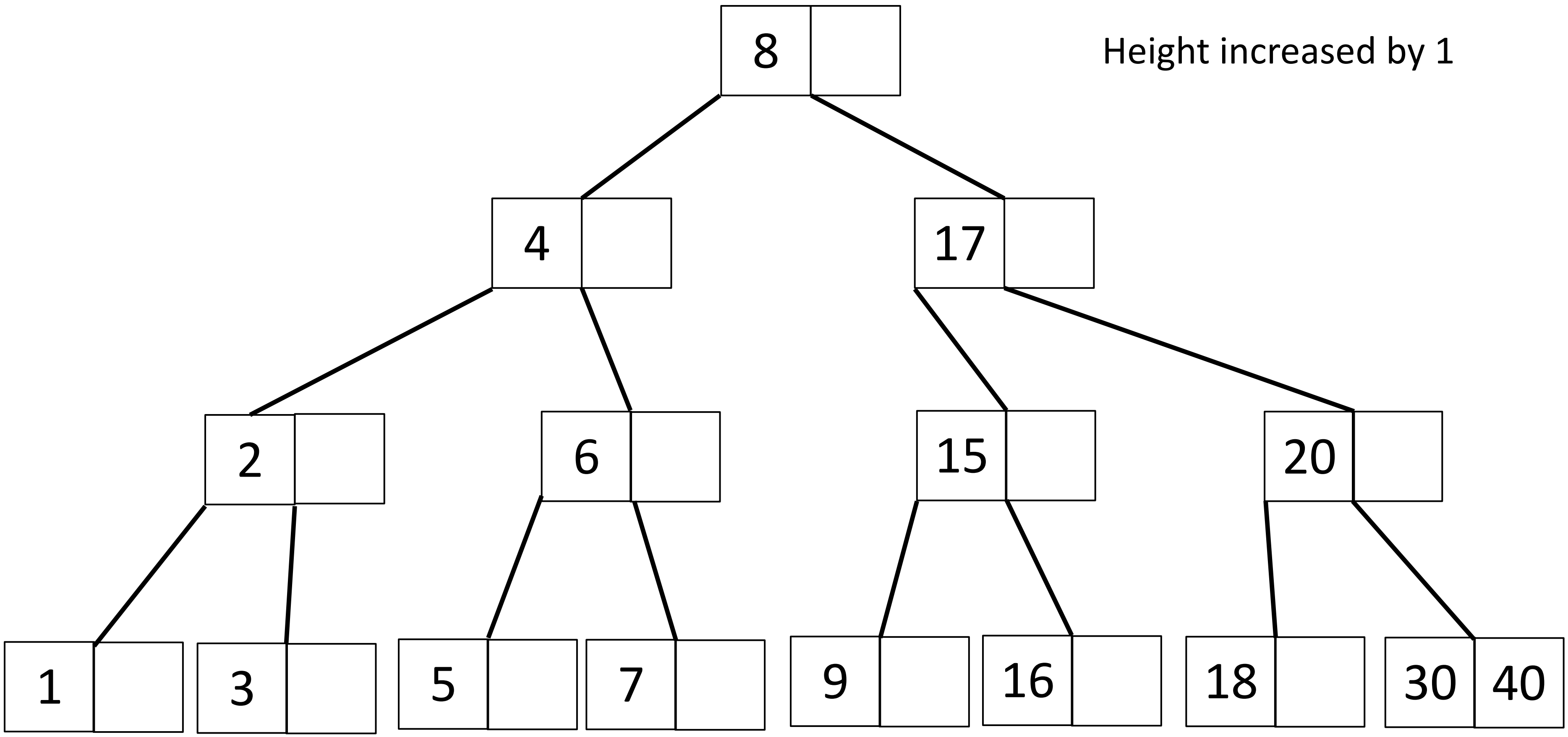
Insert (3-way B-tree)



Insert (3-way B-tree)



Insert (3-way B-tree)



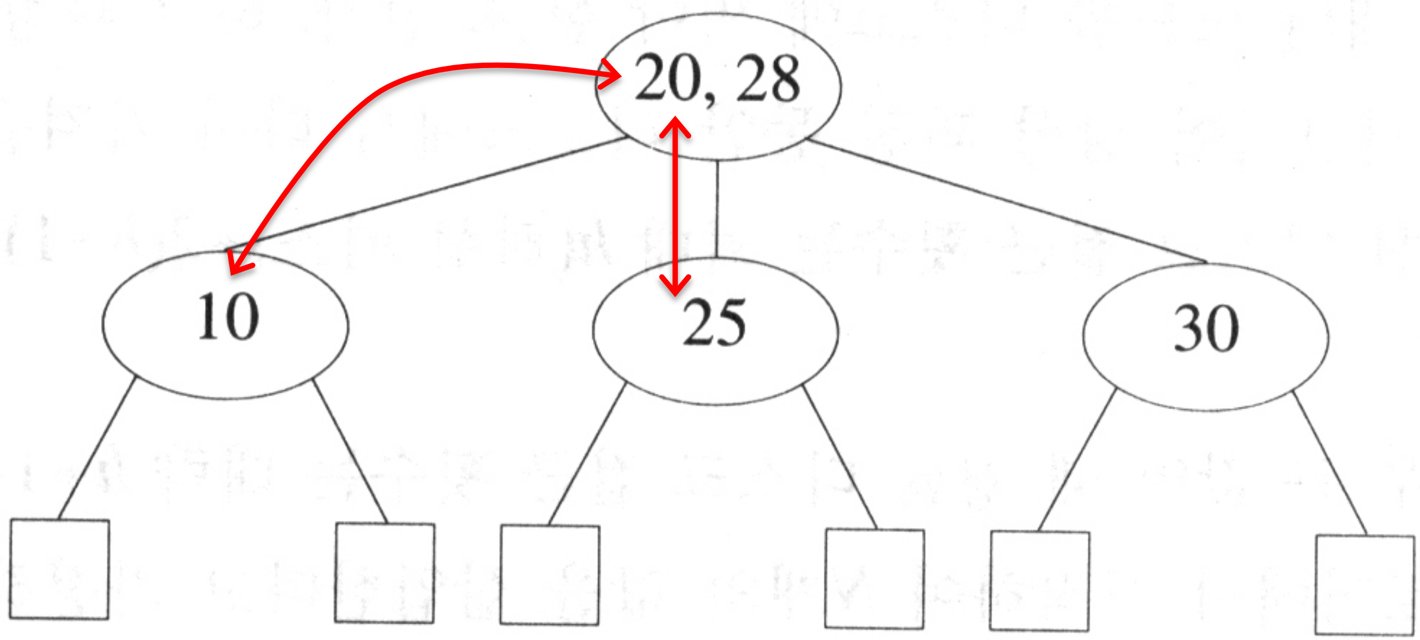
Height increased by 1

Deletion

- Delete from interior node can be done by replacing with the largest in left subtree or the smallest in right subtree
  - Similar to binary search tree
  - Smallest/largest is in the leaf node
    - Deletion from an interior node is transformed into a deletion from a leaf node
  - If deletion results in less than $\left\lceil \frac{m}{2} \right\rceil$ elements, rotation or combine must be done

Example

- Delete 20
  - Replace with 10 or 25



Deletion

- Four cases when deleting an element from a leaf node p
  - p is root: nothing to do.
  - p is not the root:
    - The number of elements in p

$$\begin{cases} \geq \left\lceil \frac{m}{2} \right\rceil - 1: \text{ nothing to do} \\ = \left\lceil \frac{m}{2} \right\rceil - 2: \begin{cases} \text{can bring from the sibling} \\ \text{cannot bring from the sibling} \end{cases} \\ < \left\lceil \frac{m}{2} \right\rceil - 2: \text{ not happening} \end{cases}$$

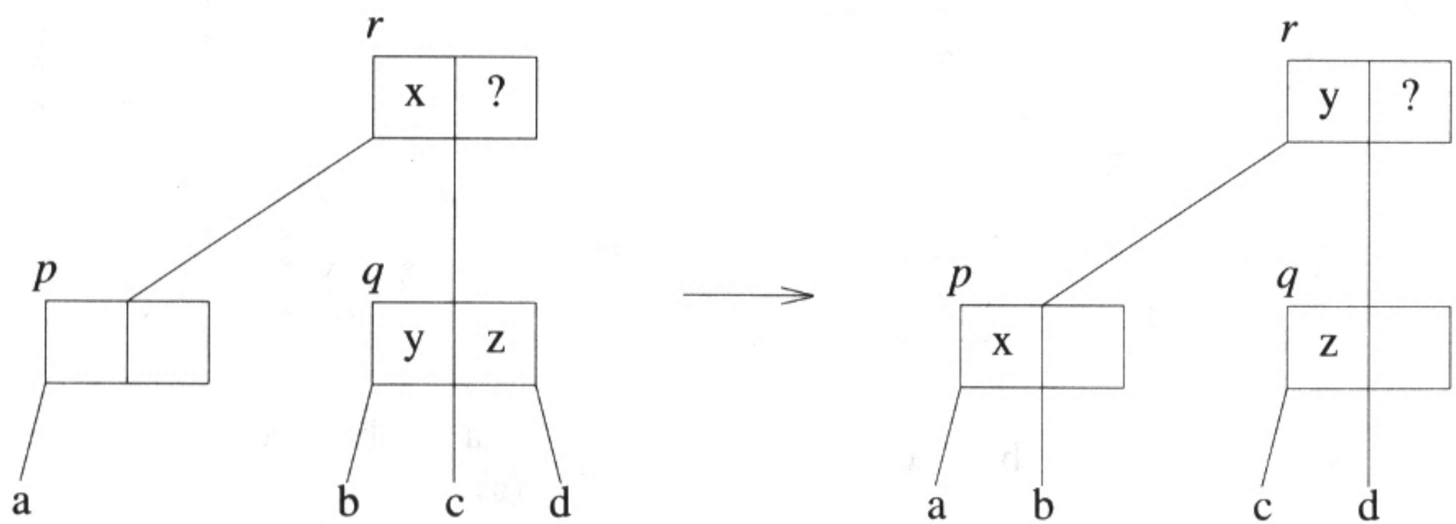
Deletion

- Four cases when deleting an element from a leaf node p

1. p is root and left with at least one element after delete
   - OK: root is not empty
2. p is internal and left with at least $\left\lceil \frac{m}{2} \right\rceil - 1$ elements after delete
   - OK: $\left\lceil \frac{m}{2} \right\rceil - 1$ elements = $\left\lceil \frac{m}{2} \right\rceil$ children

Deletion

3. p has $\left\lceil \frac{m}{2} \right\rceil - 2$ elements and its sibling q has at least $\left\lceil \frac{m}{2} \right\rceil$ elements
   - Rotation, p++, q--

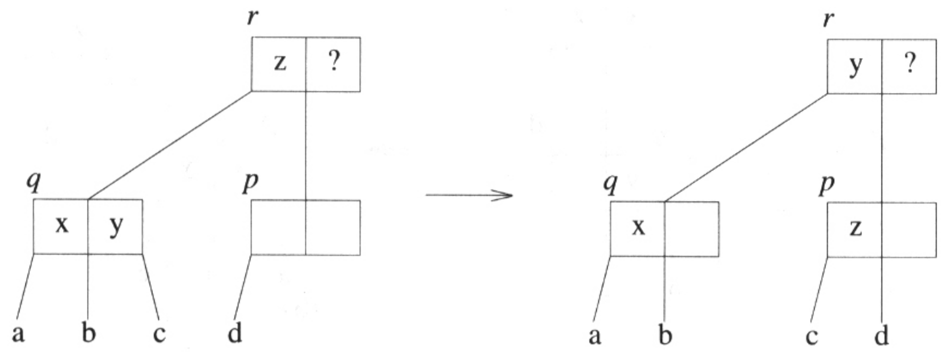


p is left child of r

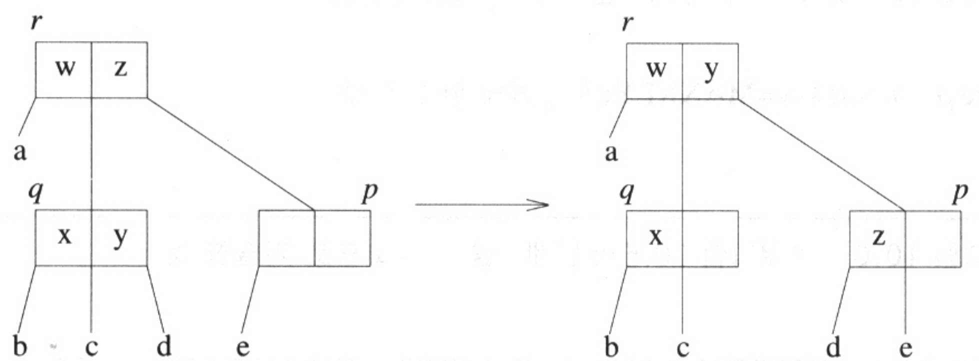
2-3 tree example

Deletion

3. More rotation examples



p is middle child of r



p is right child of r

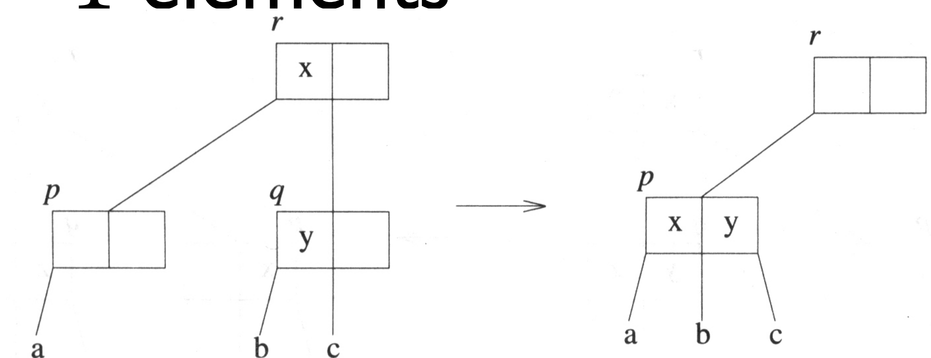
Deletion

4. p has $\left\lceil \frac{m}{2} \right\rceil - 2$ elements and its sibling q has $\left\lceil \frac{m}{2} \right\rceil - 1$ elements
   - p is deficient and q has the minimum number of elements
   - Cannot rotate: cannot reduce q's element
   - p, q, and in-between element E_i in the parent r are combined, reduce the number of element in r by one
   - If r has $\left\lceil \frac{m}{2} \right\rceil - 2$ elements, rotation and combine is applied upward to the root

Deletion

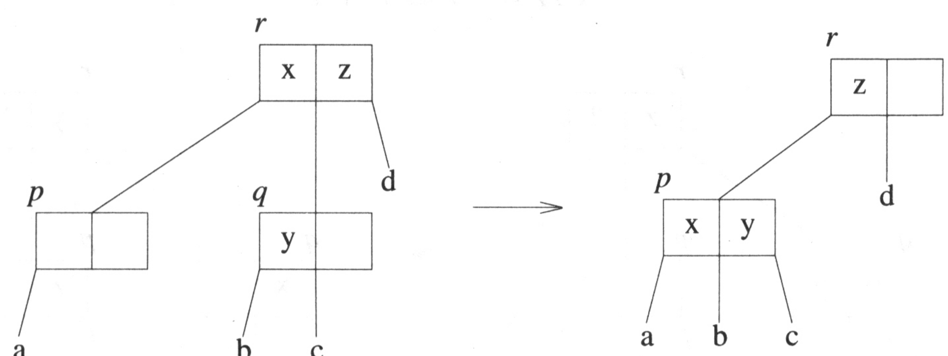
4. p has $\left\lceil \frac{m}{2} \right\rceil - 2$ elements and its sibling q has $\left\lceil \frac{m}{2} \right\rceil - 1$ elements

r has insufficient element, combine is applied upward



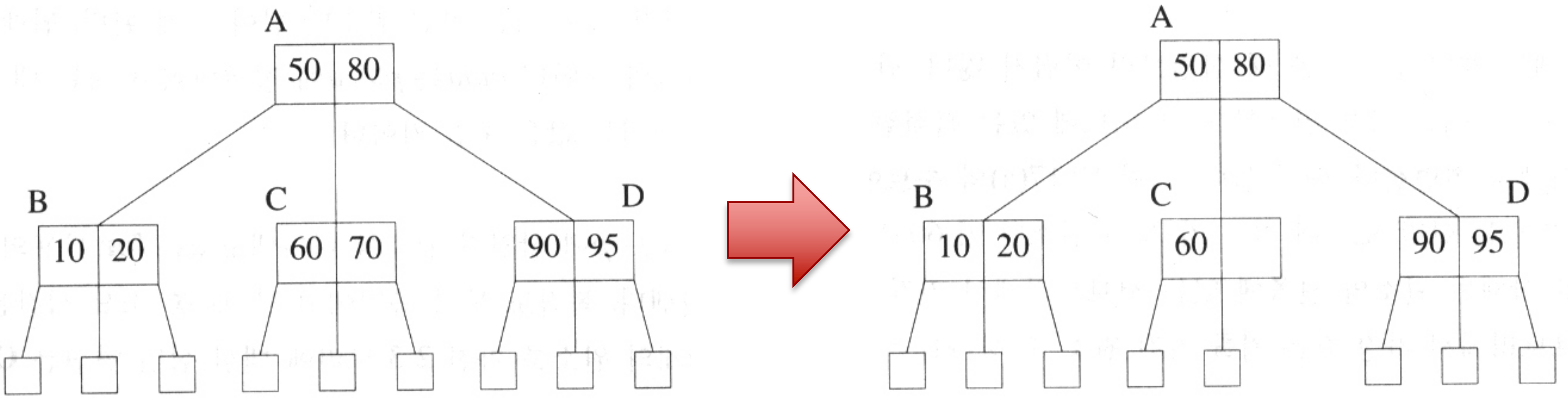
(a)

p is left child of r



(b)

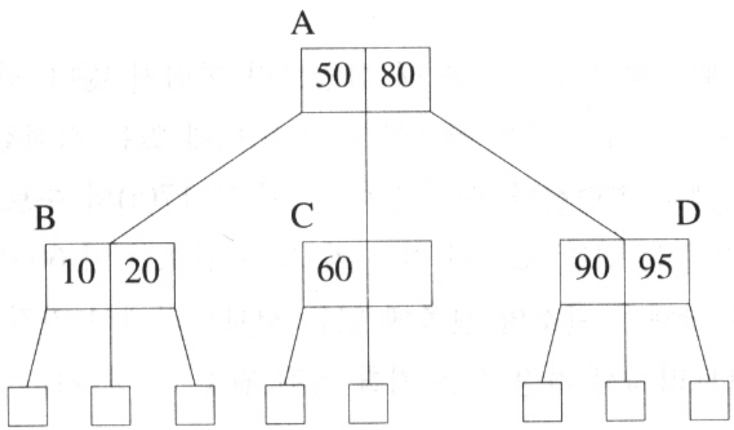
Example



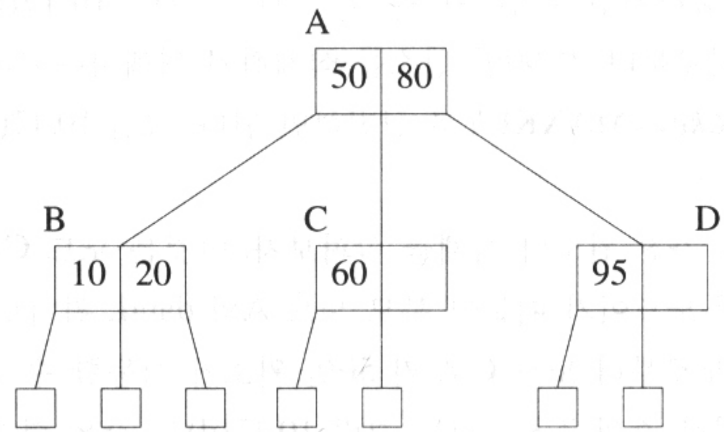
(a) Initial 2-3 tree

(b) 70 deleted

Example

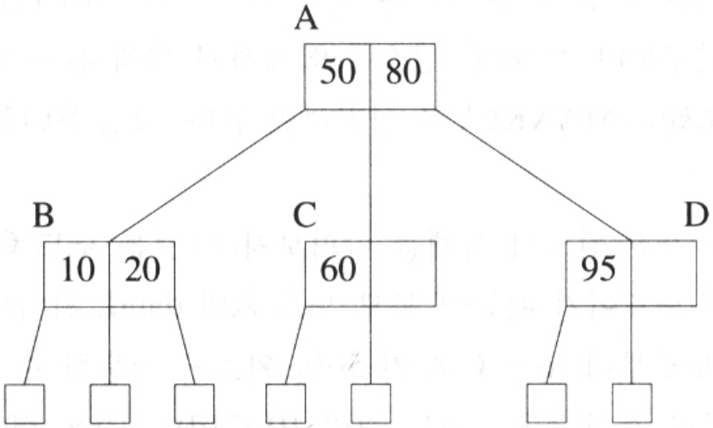


(b) 70 deleted

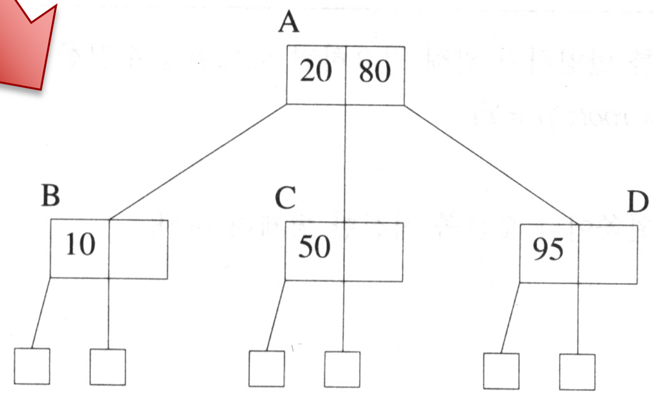


(c) 90 deleted

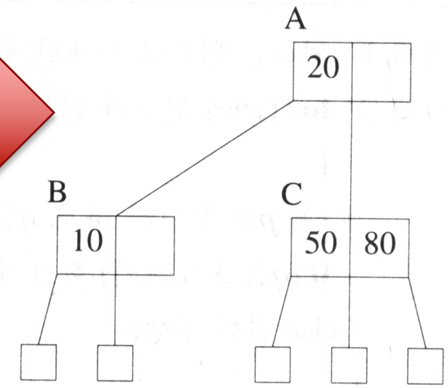
Example



(c) 90 deleted

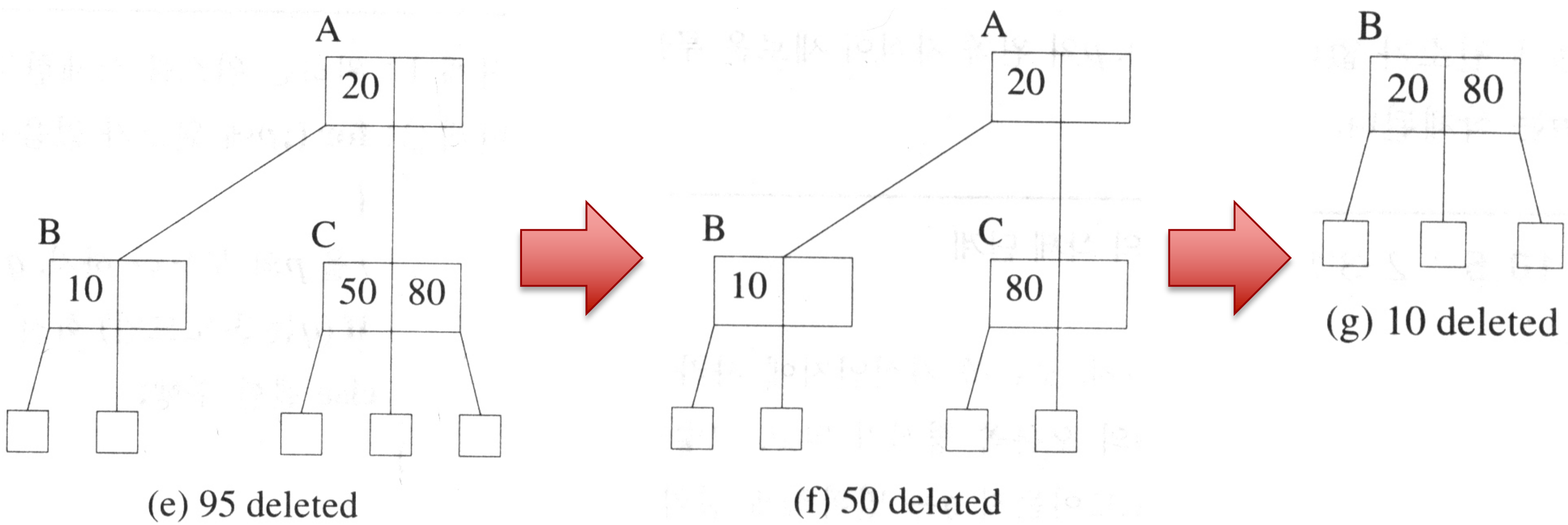


(d) 60 deleted



(e) 95 deleted

Example



(e) 95 deleted (f) 50 deleted (g) 10 deleted

Analysis

	Find	Put	Erase	Notes
Hash Table	1 expected	1 expected	1 expected	○ no ordered map methods ○ simple to implement
Skip List	log ***n*** high prob.	log ***n*** high prob.	log ***n*** high prob.	○ randomized insertion ○ simple to implement
AVL and (2,4) Tree	log ***n*** worst-case	log ***n*** worst-case	log ***n*** worst-case	○ complex to implement

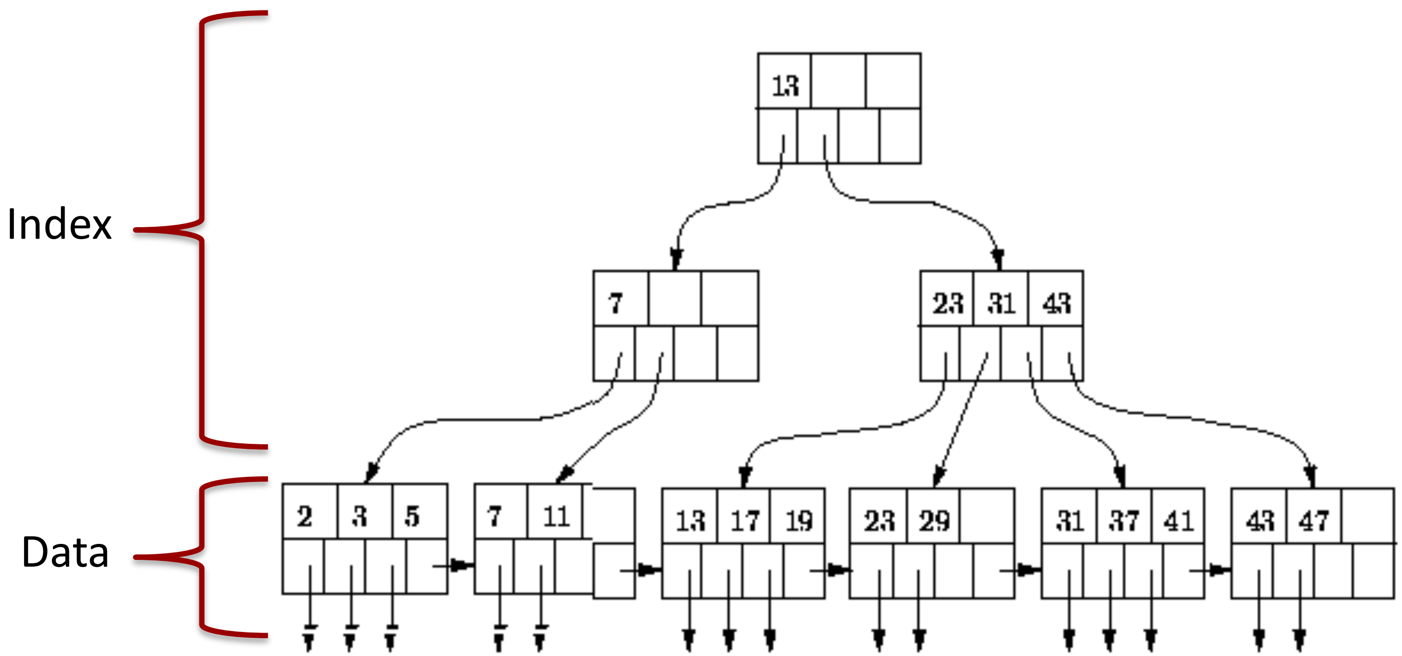
Outline

- m-way search trees
- B-trees
- B^+-trees

B+-Trees

- Interior node : index (key)
- Leaf node : data
- Data nodes are linked using linked list

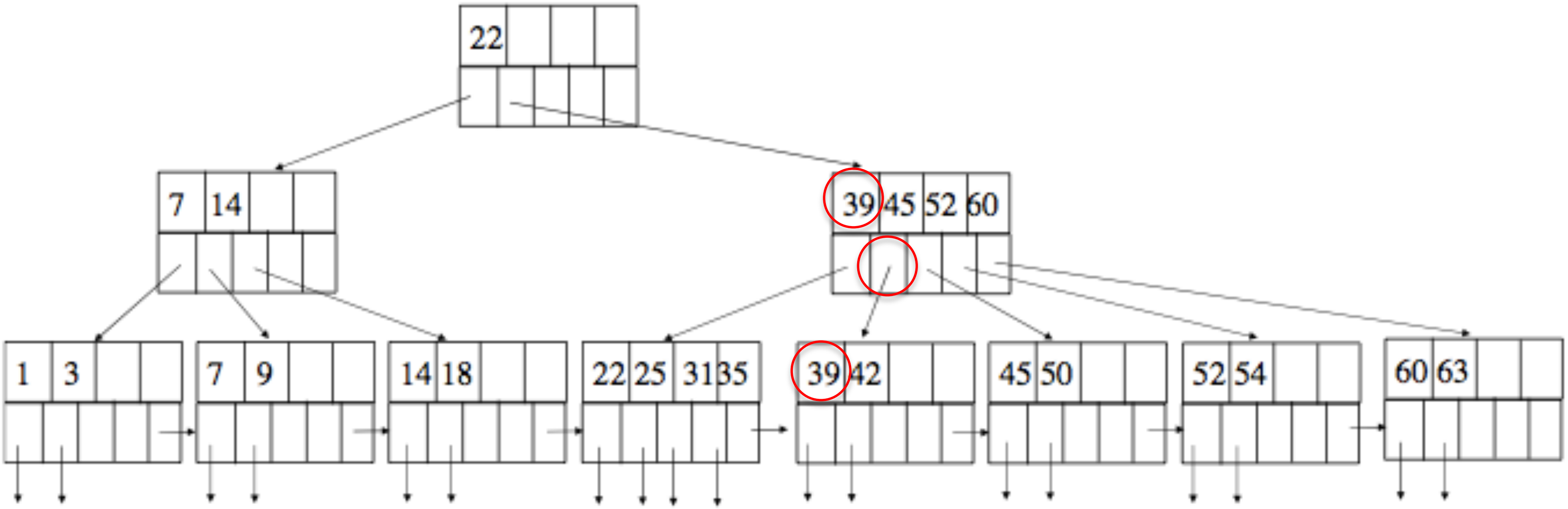
A B+ Tree



B^+-Trees

- All data nodes are at the same level and are leaves
  - Data node contains all the keys
- The index nodes define a B-tree of order m
- Let index node p have the format
  - $m, A_0, (K_1, A_1), \dots, (K_n, A_n), n<m$
  - $K_i \leq$ all elements in $A_i < K_{i+1}$
- Efficient for both direct and sequential access

B+-Trees

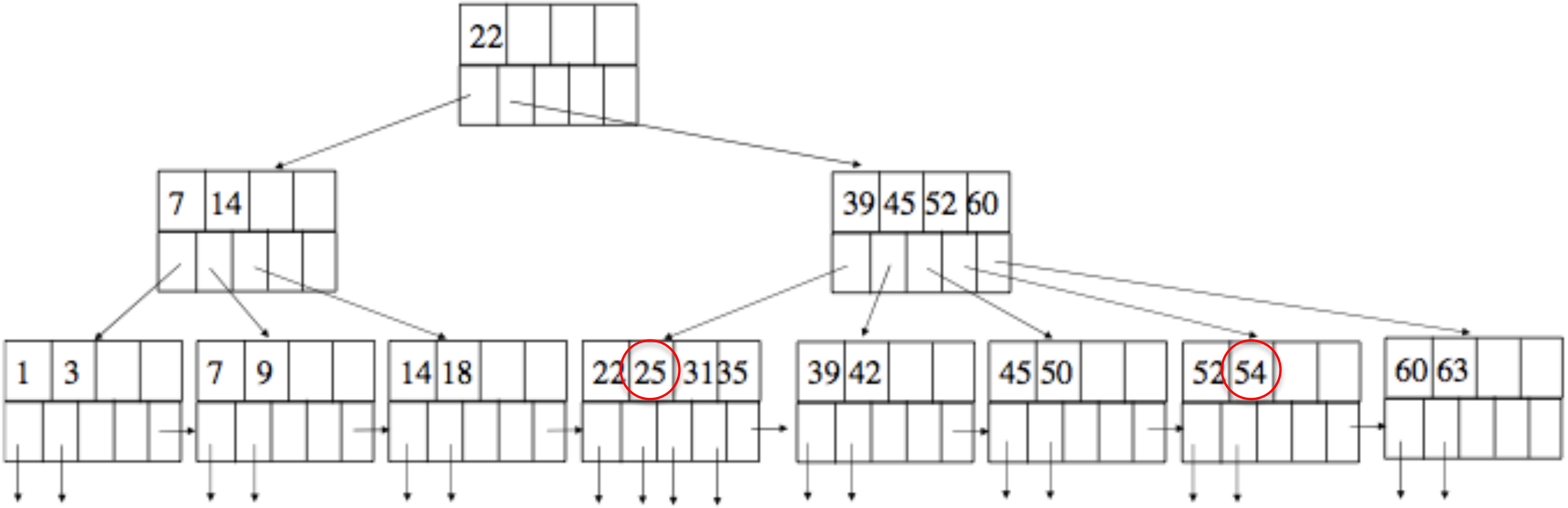


B^+-Trees Search

- Exact match
  - Search to leaf node, return exact match
- Range search [A,B]
  - Search to leaf node for A
  - Start from that node, linear search in the data node that exceed B
  - Collect all the elements between them

Range Search

- [23,55]

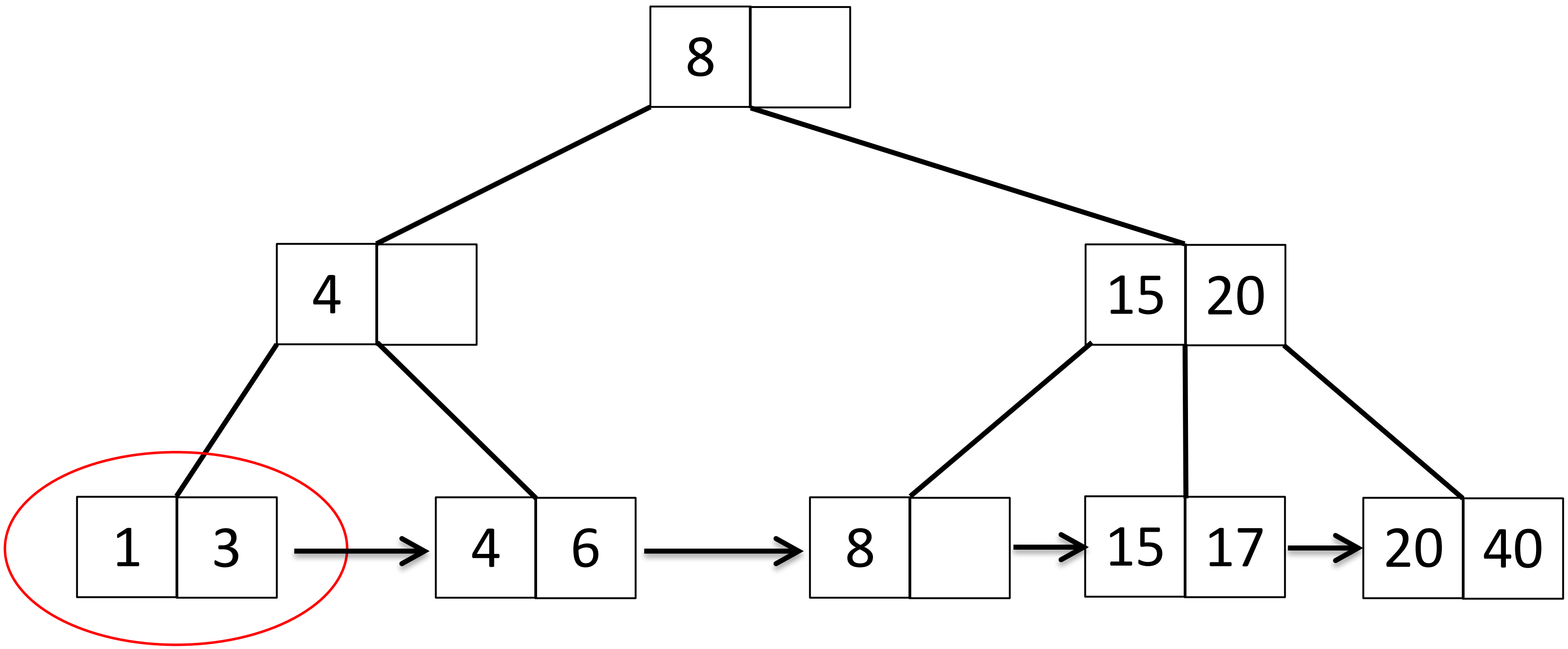


B^+-Trees Insert

- Similar to B-tree insert
- Split leaf (data) node if overfull
- Smallest key of the newly created data node is inserted to the parent index node
  - That key exists in both leaf and its parent

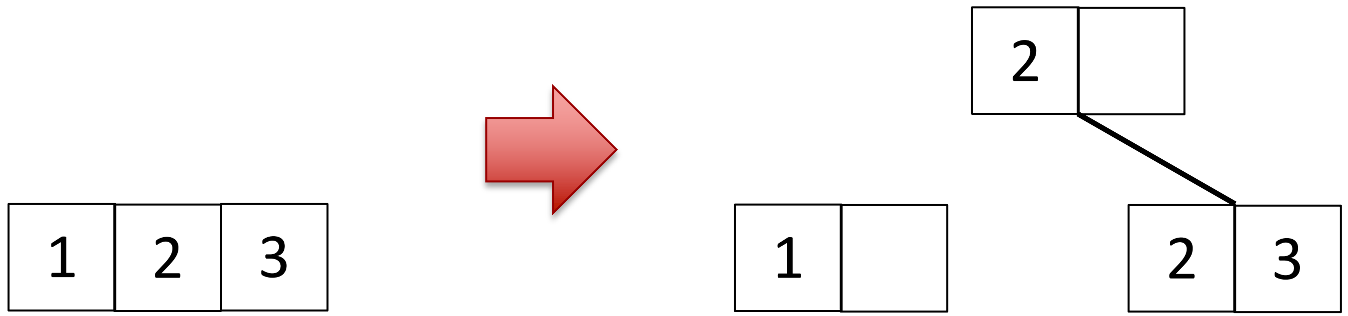
Insert

- Insert key = 2



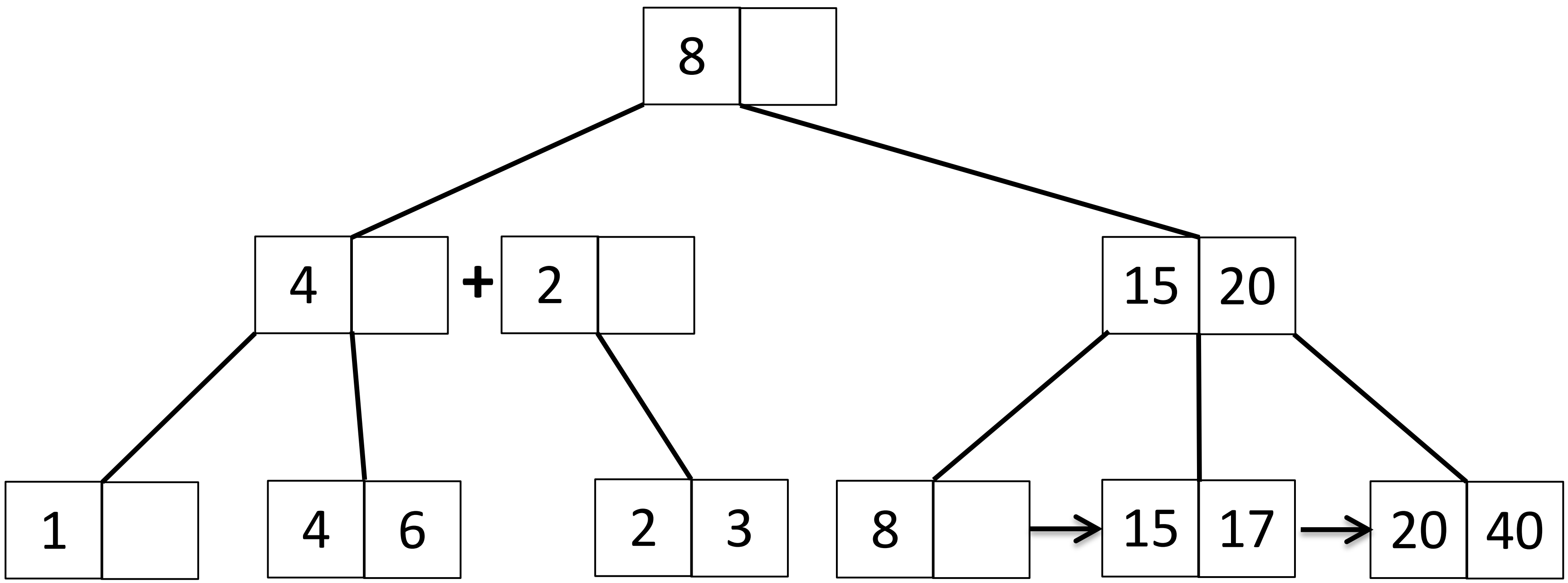
Insert into a Data Node

- Split overflowed node into half

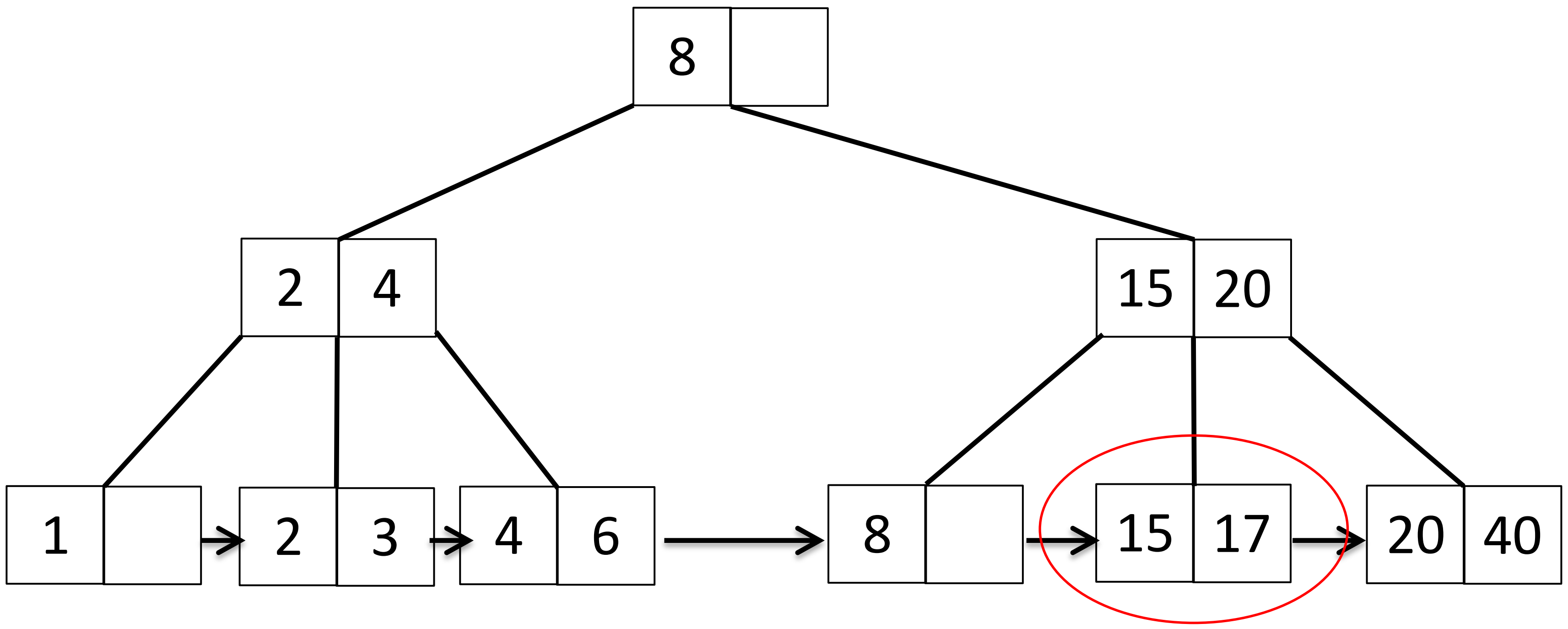


- Insert smallest key of <u>right half</u> to its parent
  - 2 is duplicated in parent and child nodes

Insert

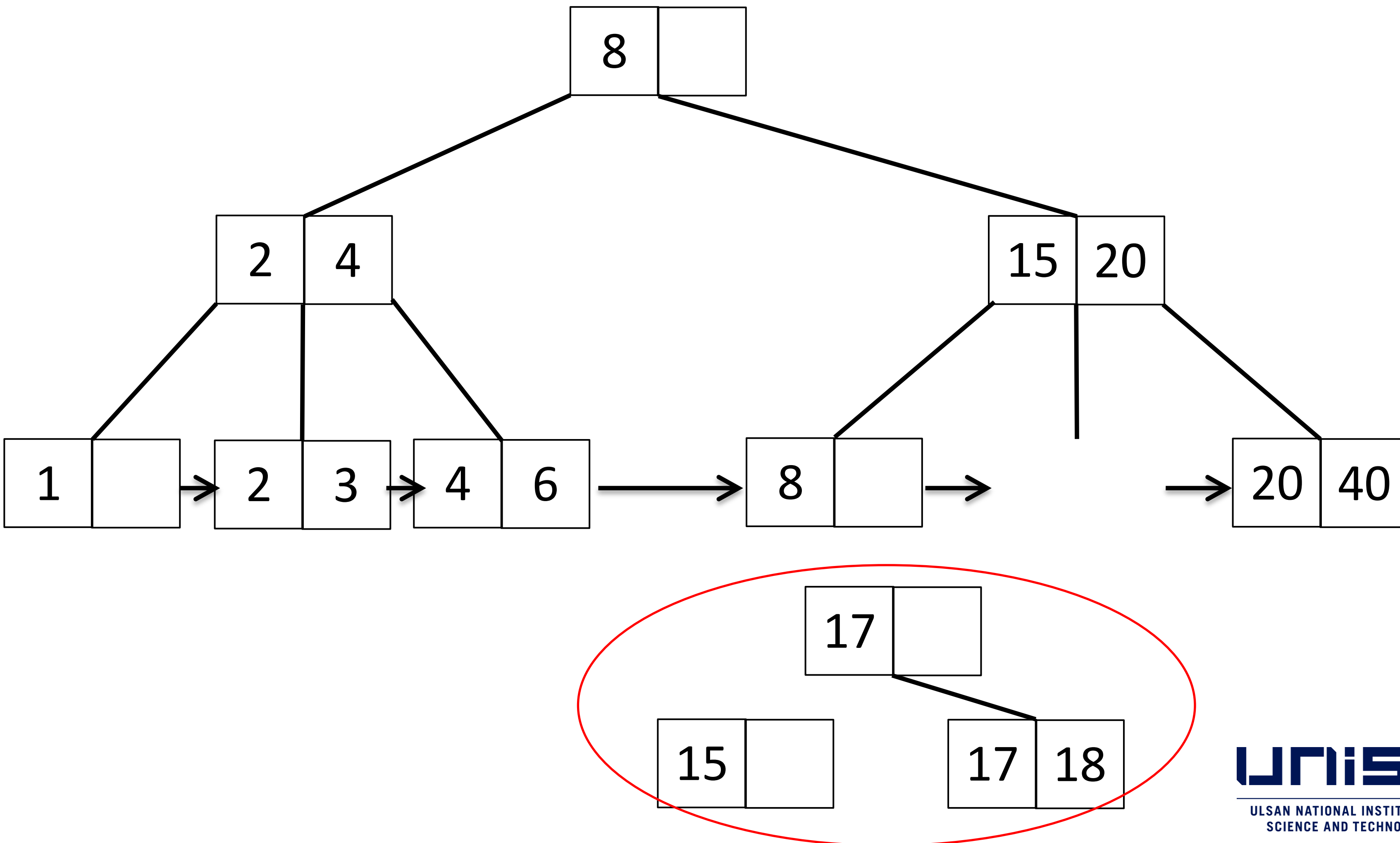


Insert



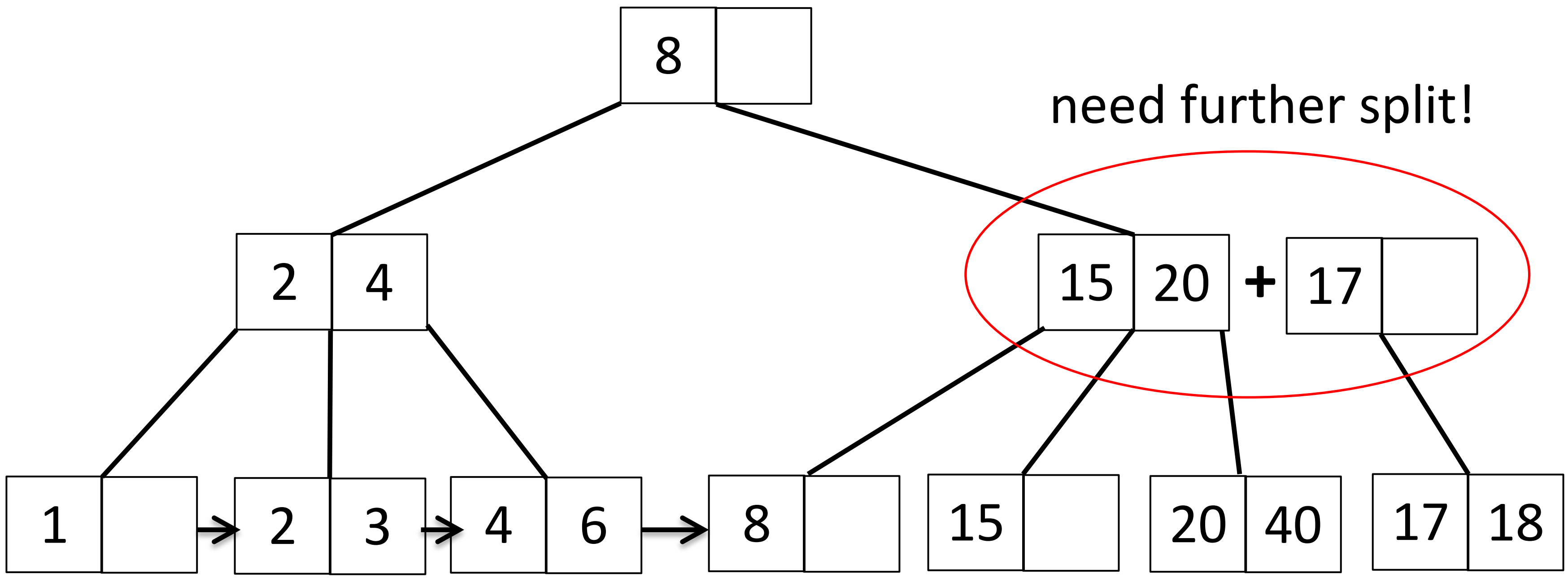
Insert 18

Insert

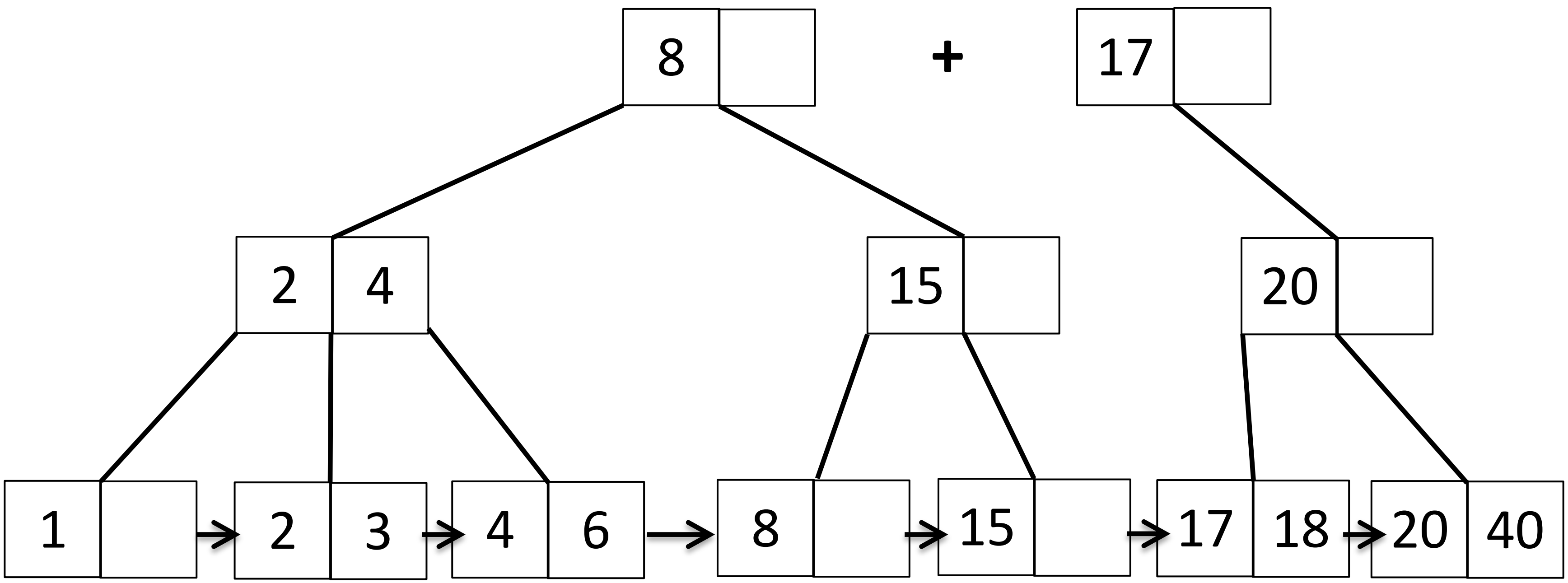


ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

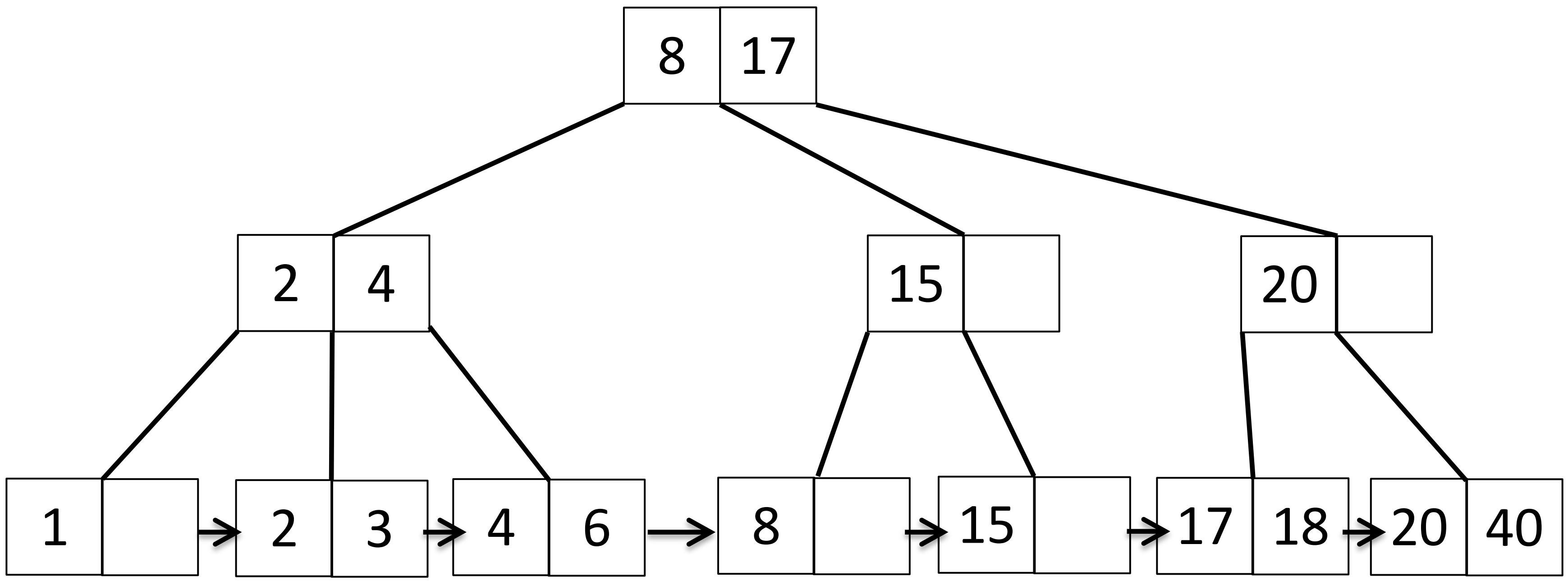
Insert



Insert



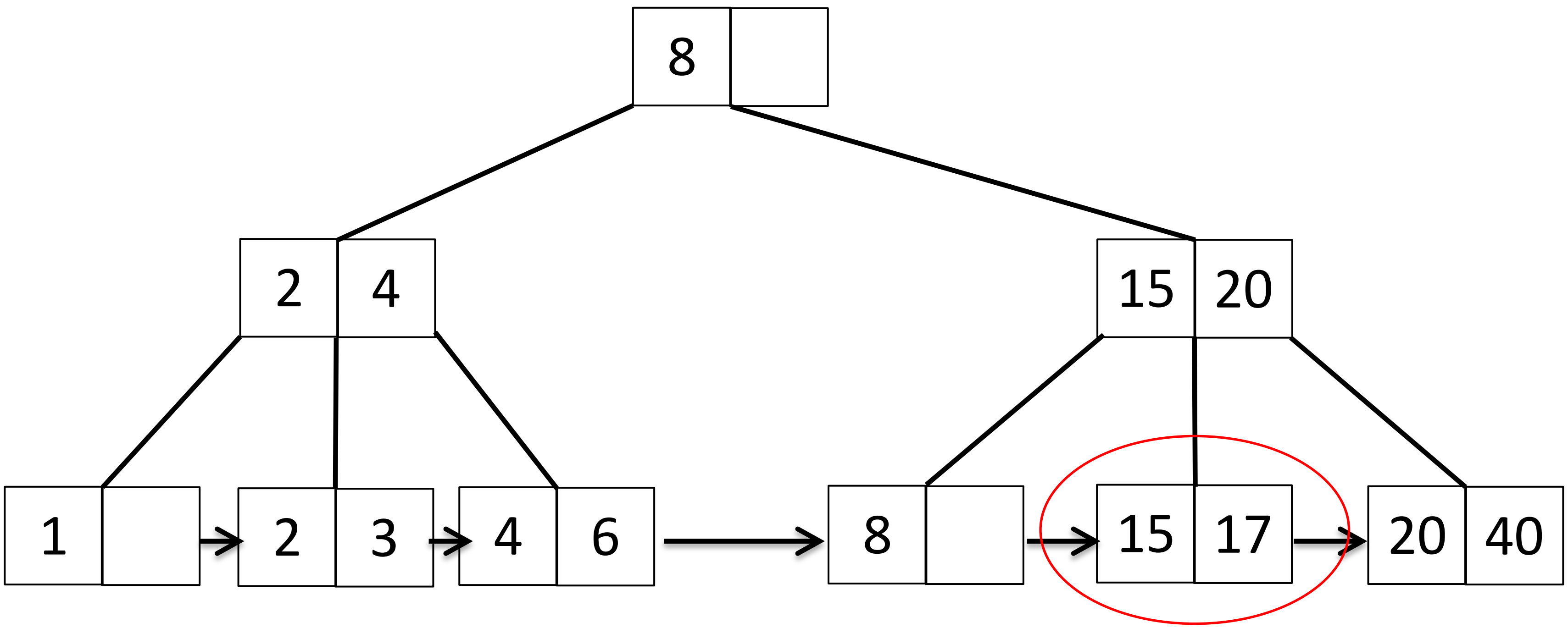
Insert



Delete

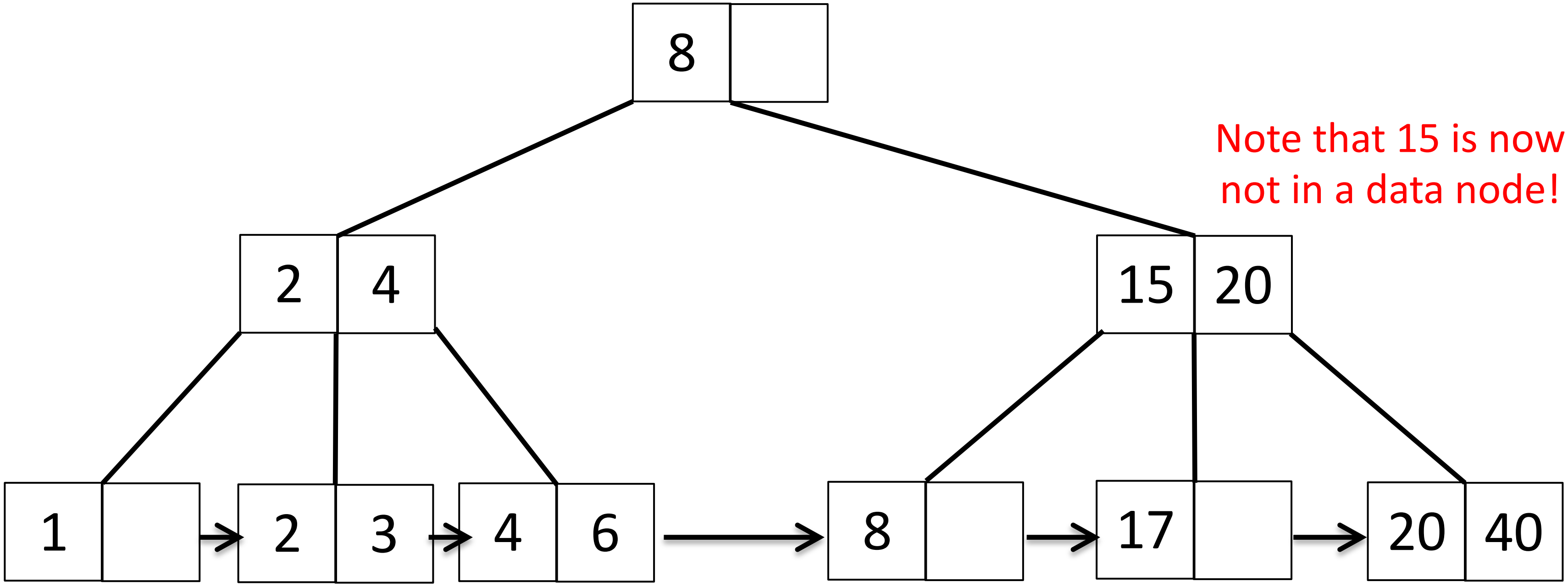
- Delete always occurs on data node
- Data node is deficient if its element is fewer than ceil(c/2), c : capacity of data node
  - Borrow one element from nearest left/right sibling data node and update root index
  - If siblings do not have enough element to borrow, merge two data node and delete index in-between
- If index node is deficient, update as in B-tree

Delete



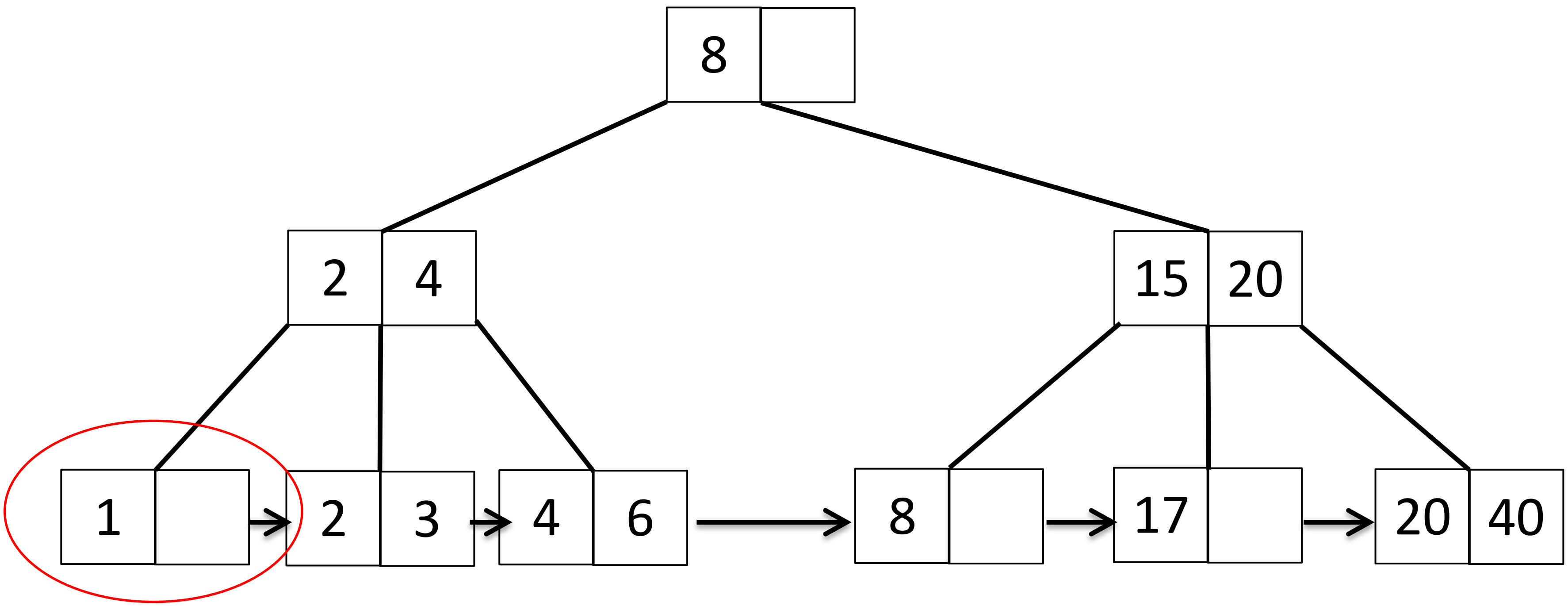
Delete 15

Delete



Note that 15 is now not in a data node!

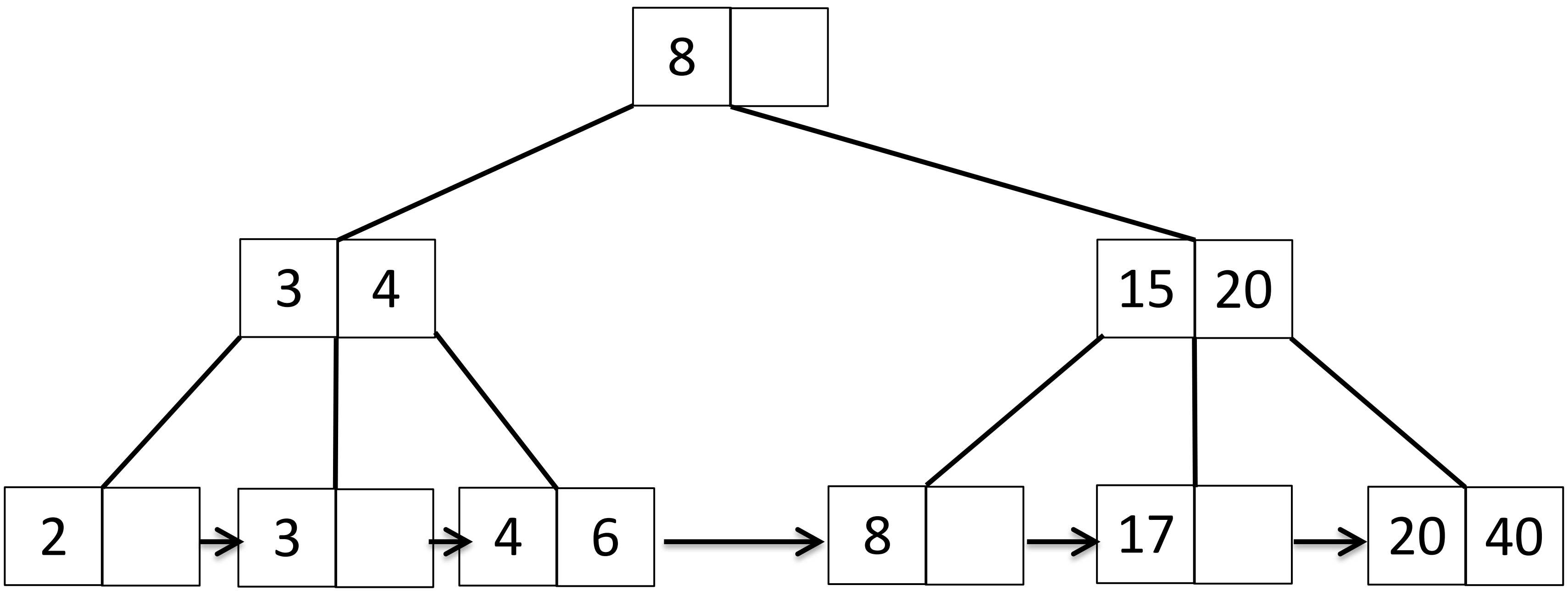
Delete



Delete 1

Get element from sibling and update parent key

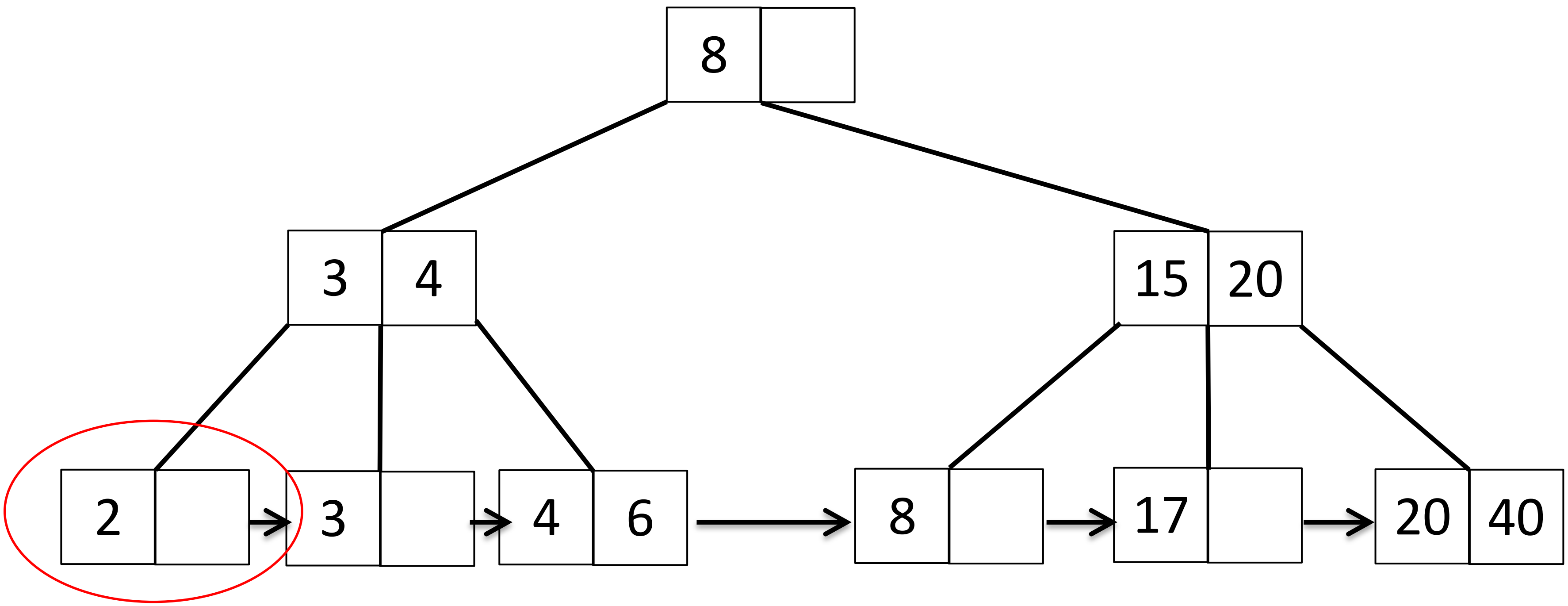
Delete



Delete 1

Get element from sibling and update parent key

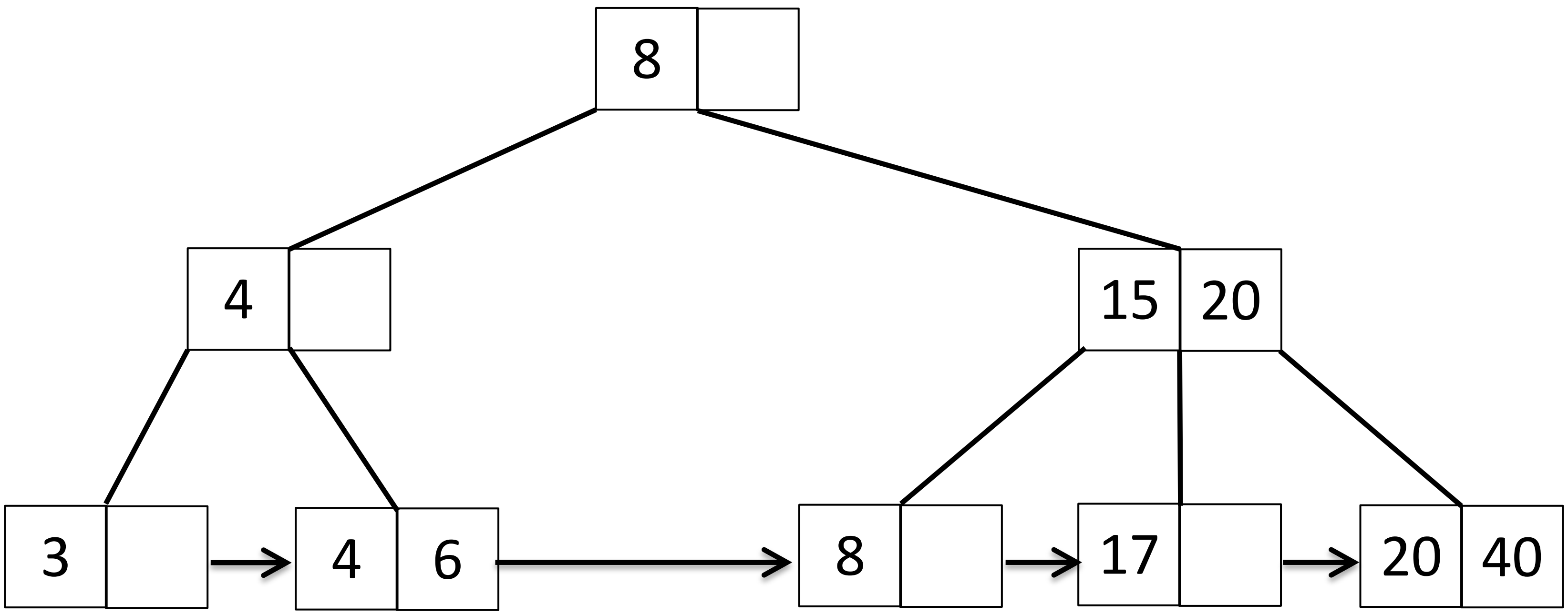
Delete



Delete 2

Merge with sibling, delete in-between key in parent

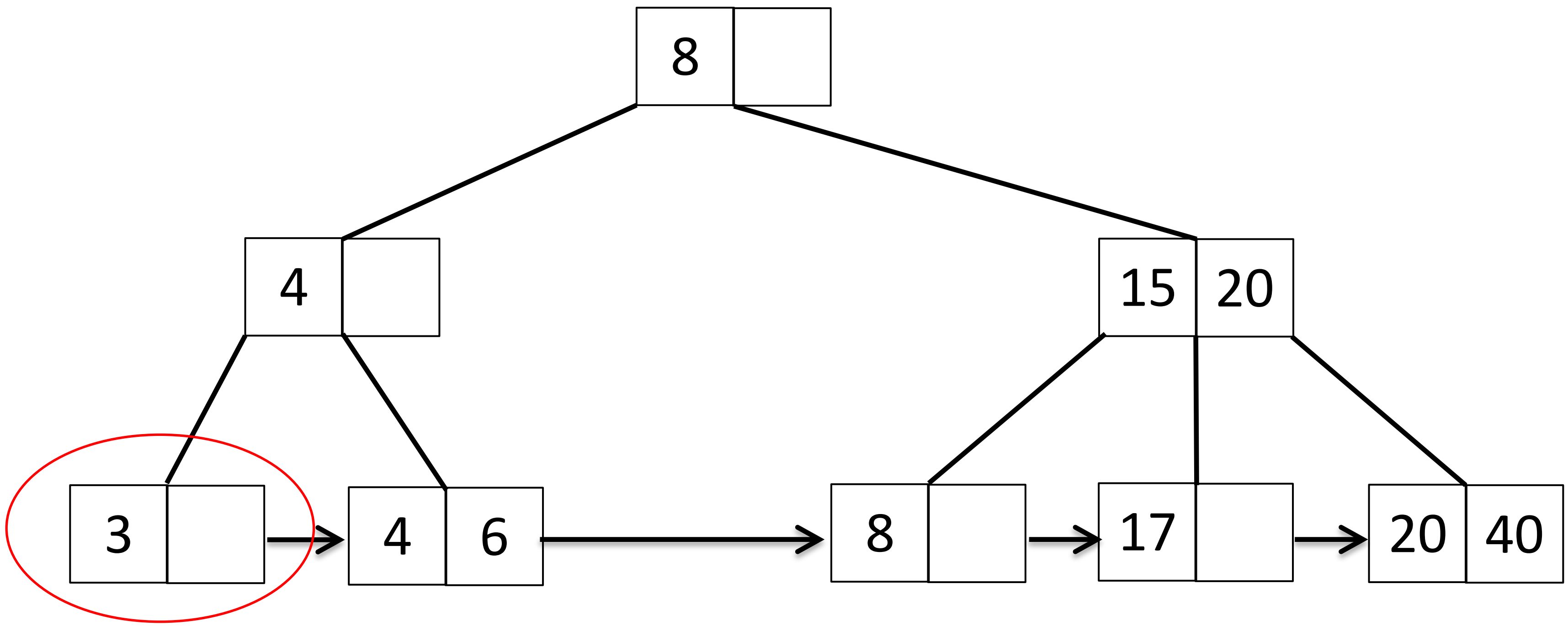
Delete



Delete 2

Merge with sibling, delete in-between key in parent

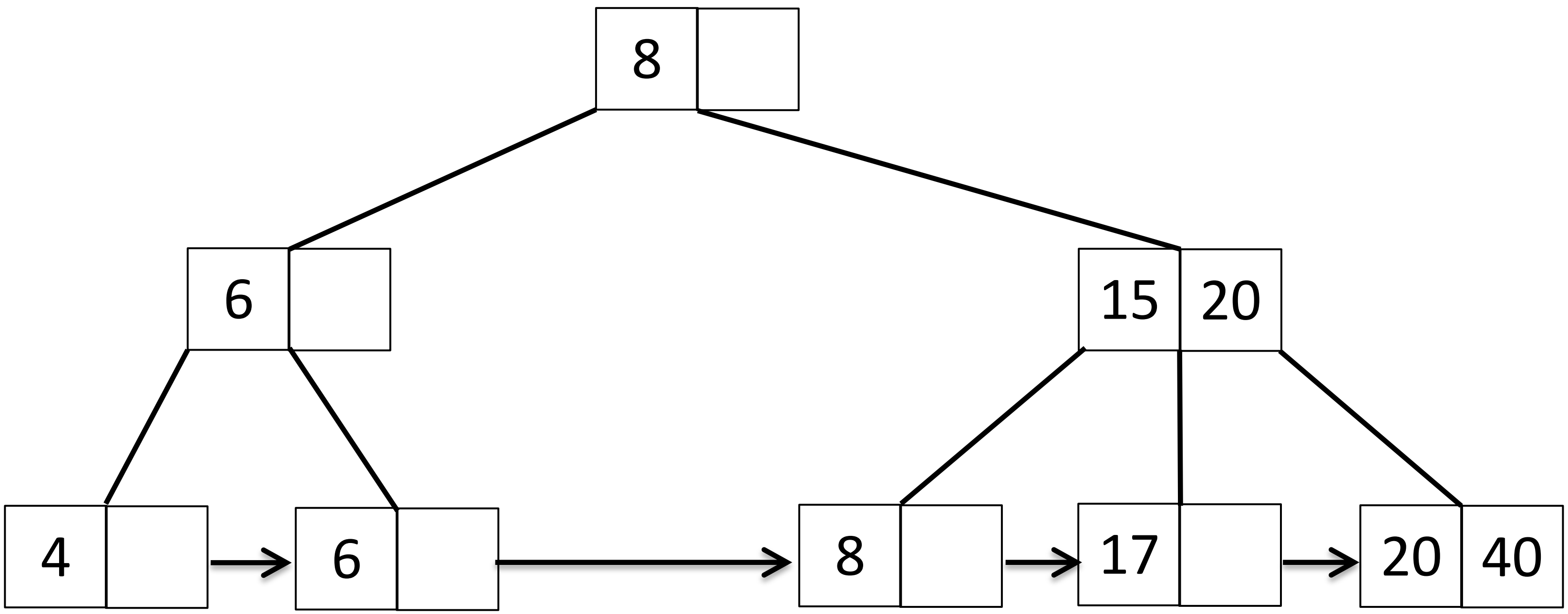
Delete



Delete 3

Get element from sibling and update parent key

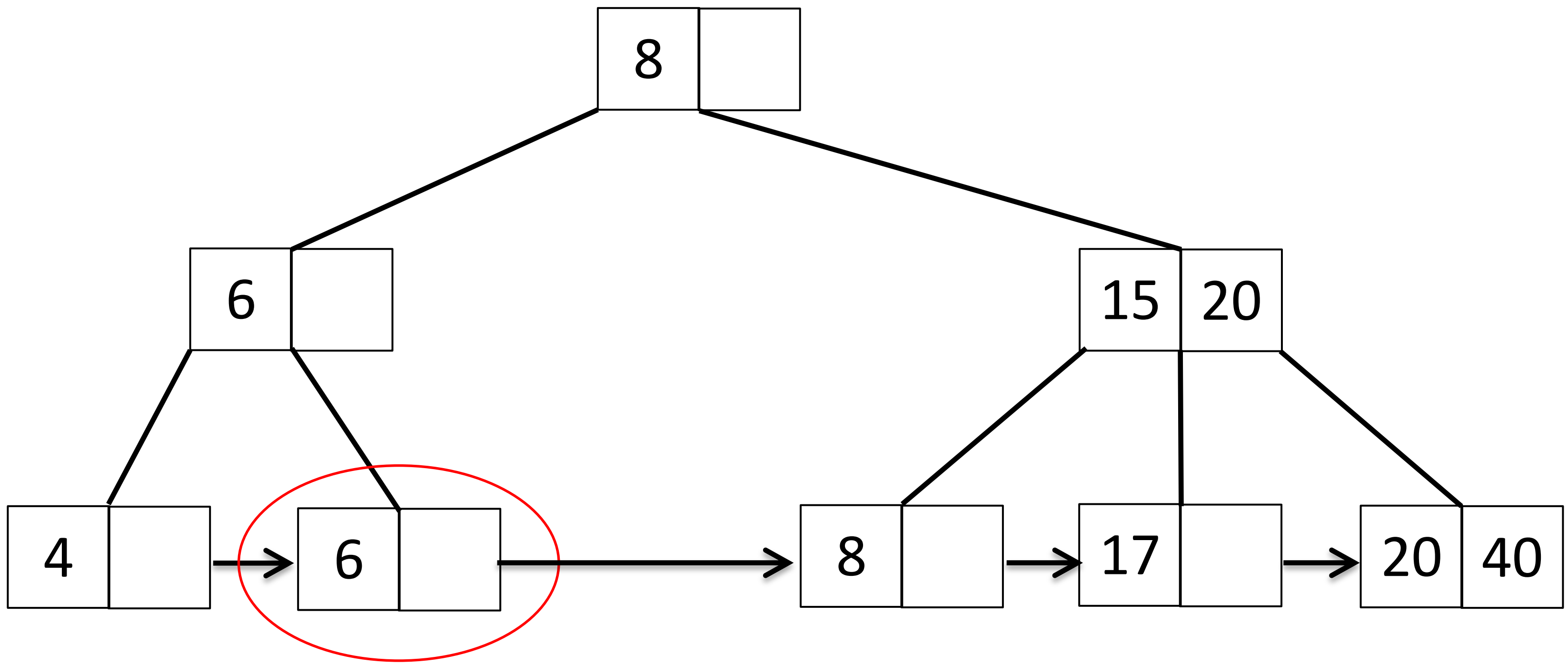
Delete



Delete 3

Get element from sibling and update parent key

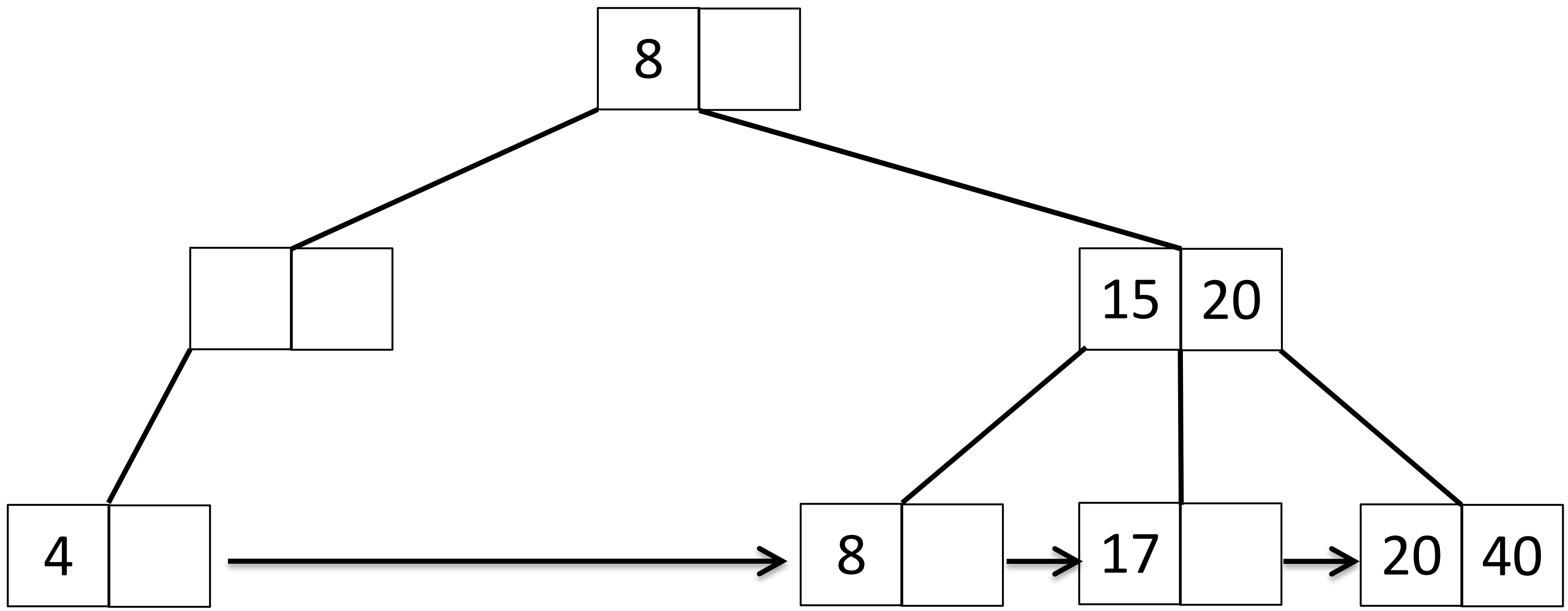
Delete



Delete 6

Merge with sibling, delete in-between key in parent

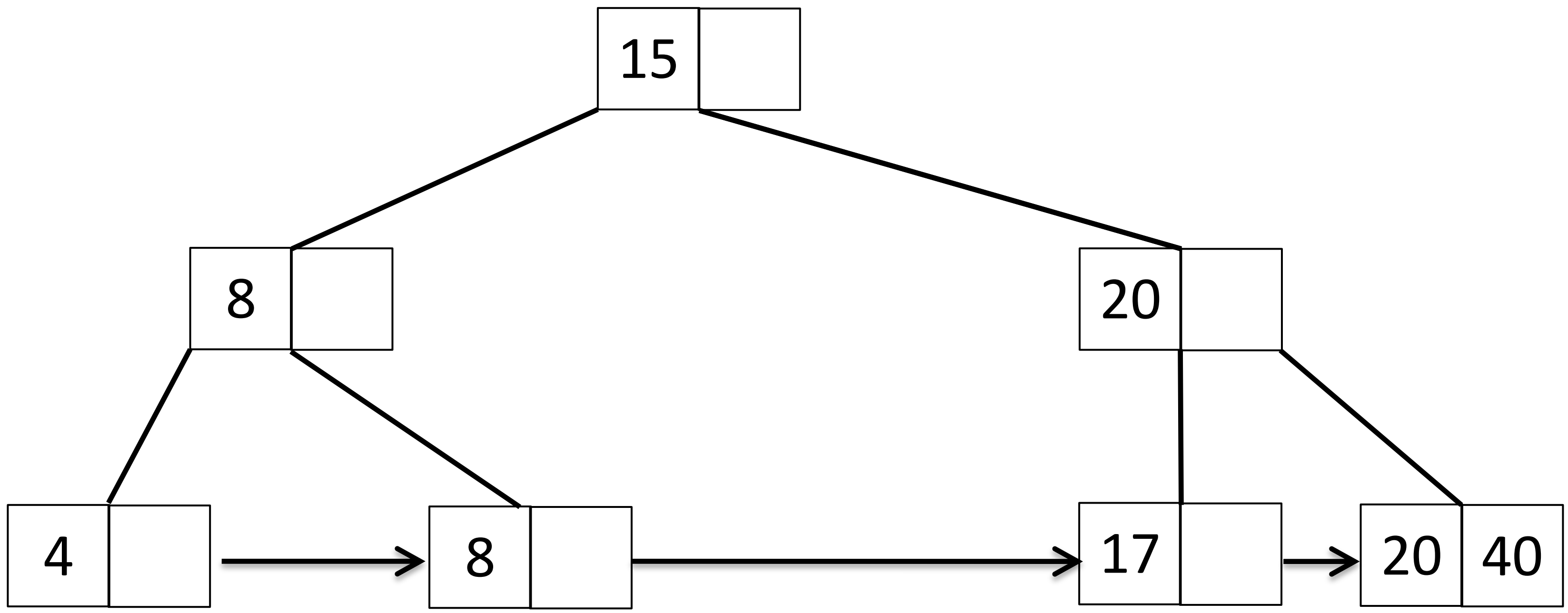
Delete



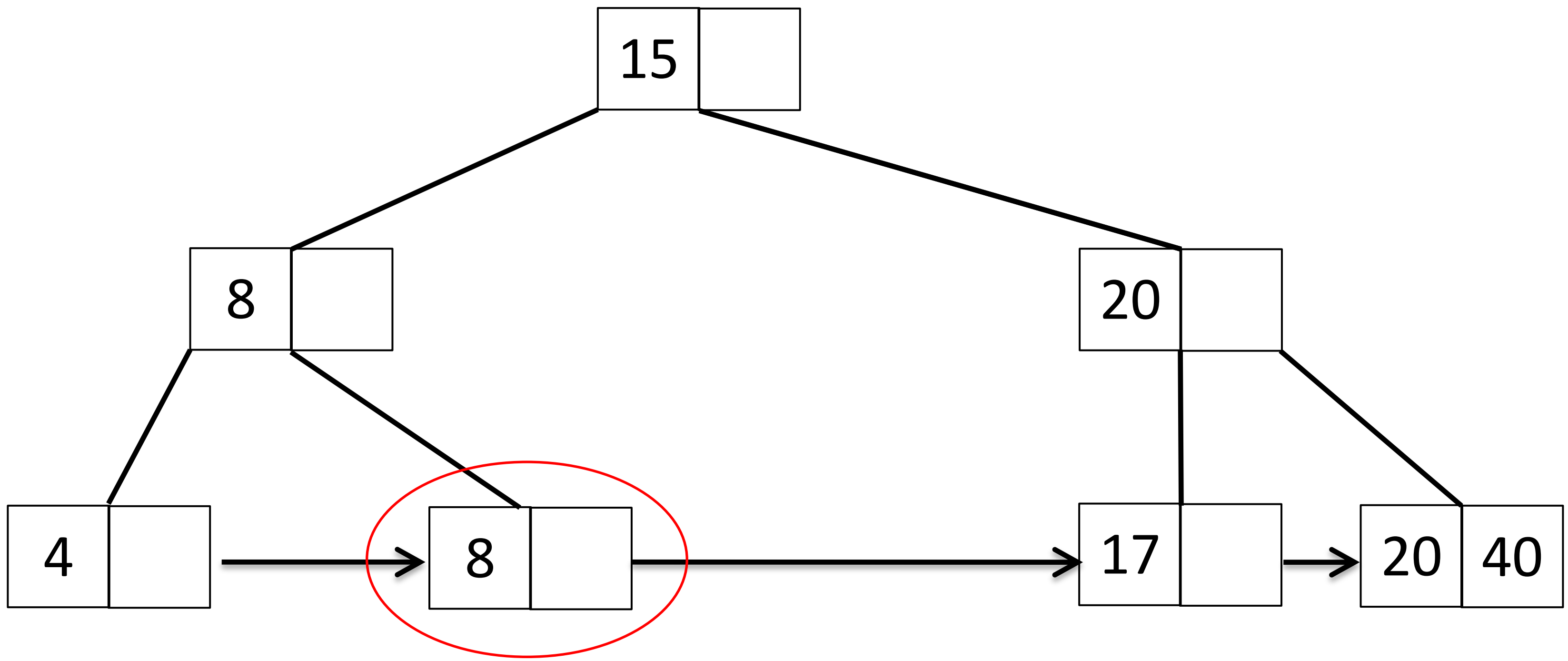
Index node become deficient.

Rotate index node (as in B-tree).

Delete



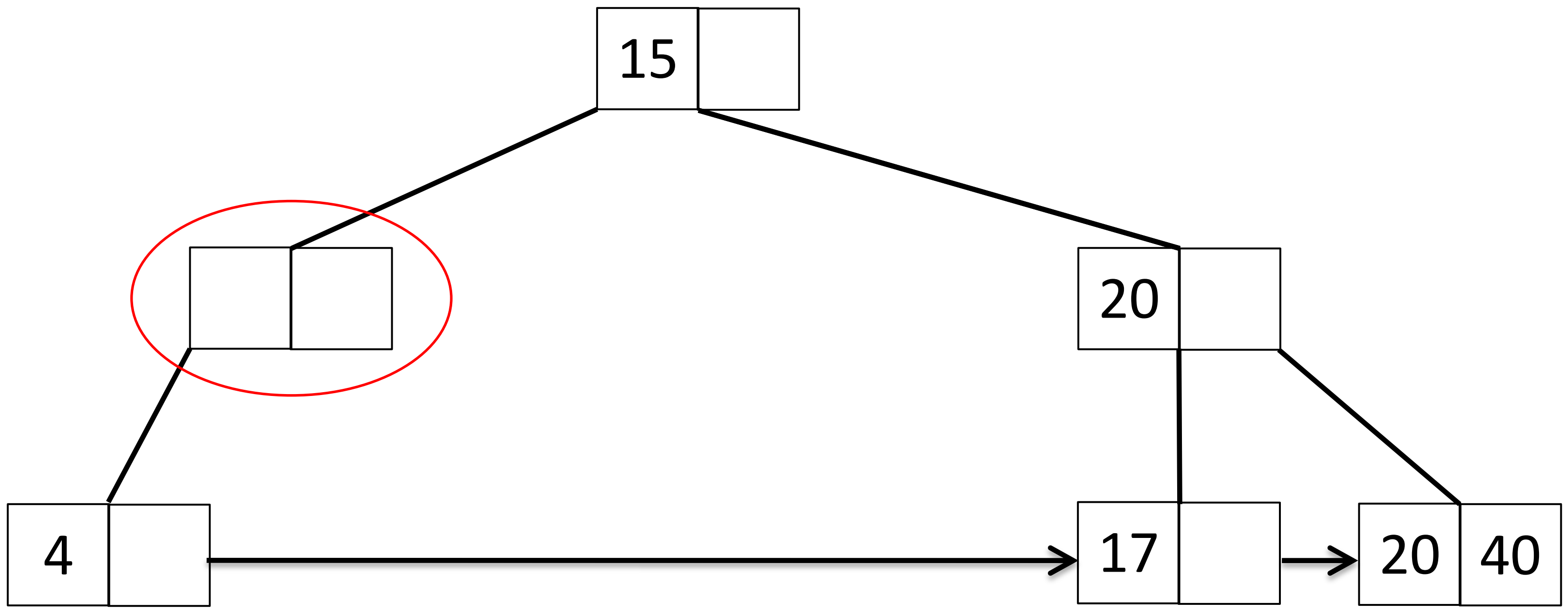
Delete



Delete 8

Merge with sibling, delete in-between key in parent

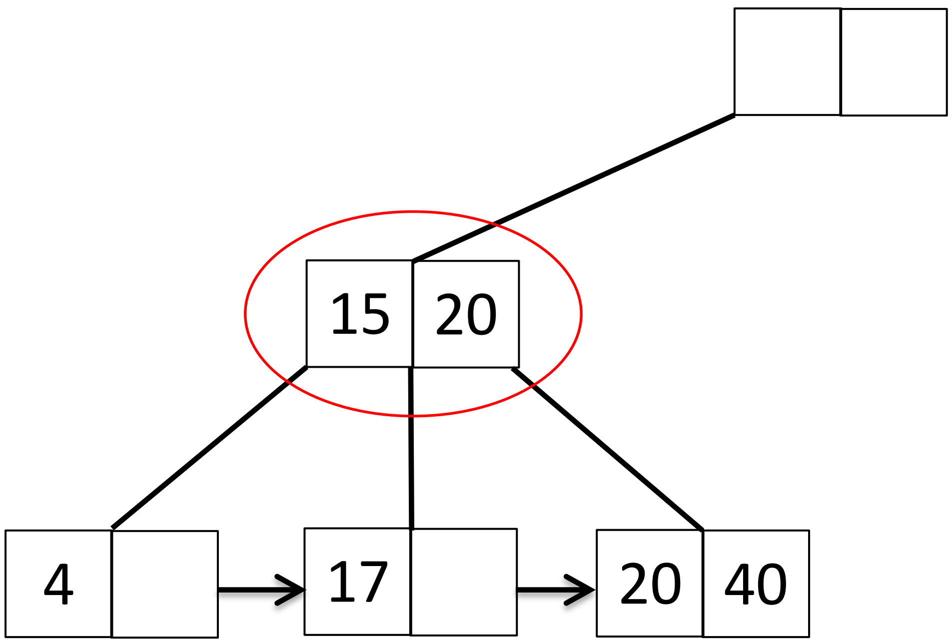
Delete



Index node deficient

Merge with sibling and in-between key in parent

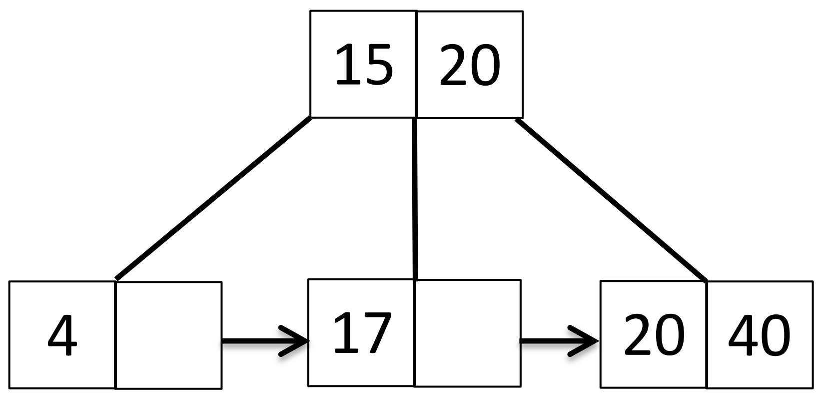
Delete



Index node deficient

It is the root : discard

Delete



Discussion

- B & B+ trees perform similar on direct access
- B+ trees perform better for sequential access
- B+ trees always have to be traversed to leaf for direct access

Questions?