[CSED233-01] Data Structure

2-3 Tree, B-Tree

B

Jaesik Park

POSTECH

Balanced Search Trees

- Memory-based search trees
  - Balanced BST
    - AVL (Adelson-Velskii & Landis) trees
    - Red-black trees
    - Splay trees, …
  - Balanced multi-way search tree
    - 2-3, 2-3-4 trees (B-trees)

- Disk-based search trees
  - Balanced multi-way search trees
    - B-trees (B+, B*)
    - Prefix B-trees

2-3 Tree: Shape Property

- A tree with the following *shape properties*:
  - Each internal node has 2 or 3 children
  - All leaves are at the same level (→ Perfectly height-balanced)

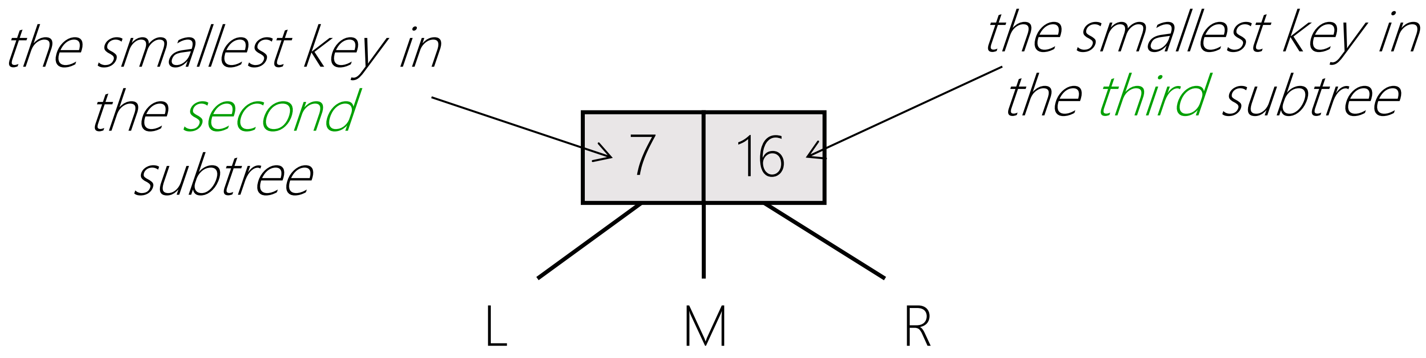
- Internal nodes
  - Store one or two key values
  - Placeholders to guide the search

- Leaf nodes
  - All *actual records* - (*key*, *info*) pairs are stored at the leaves

2-3 Tree: Search Tree Property

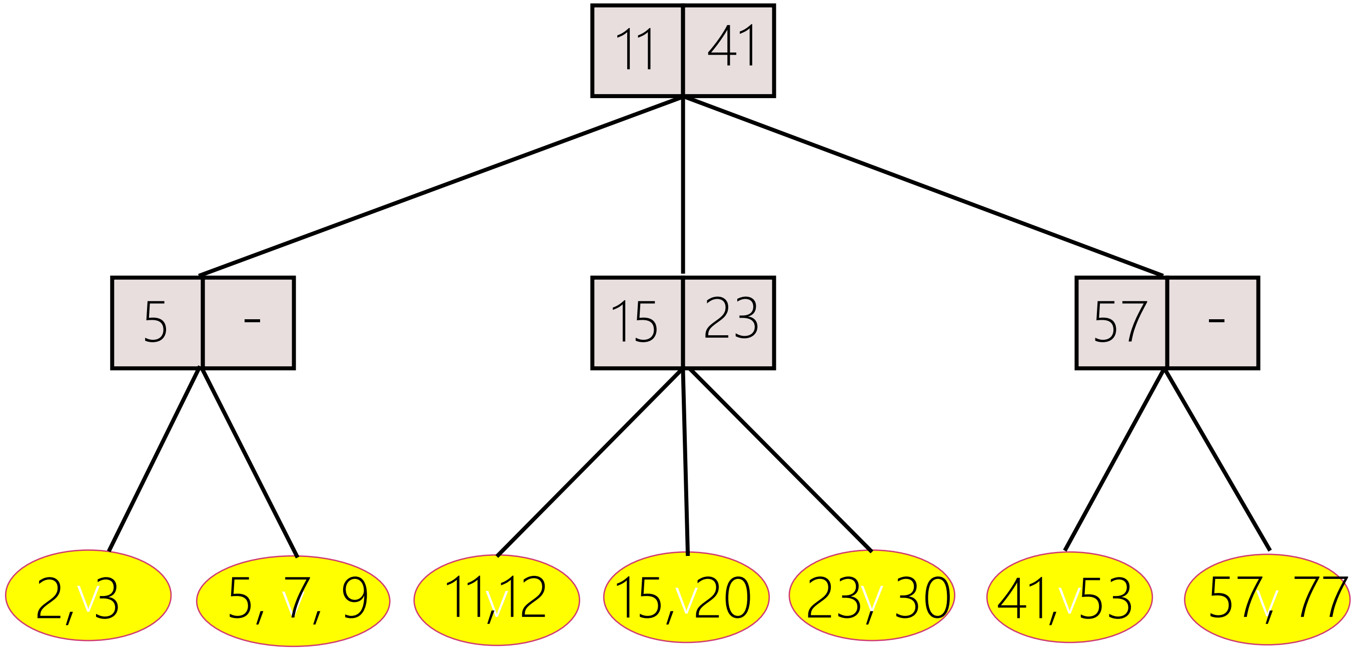
- The 2-3 tree has a *search tree property*:

> - All keys in left subtree are smaller than first key
> - All keys in middle subtree are greater than or equal to first key, and smaller than second key
> - All keys in right subtree are greater than or equal to second key



2-3 Tree: Example

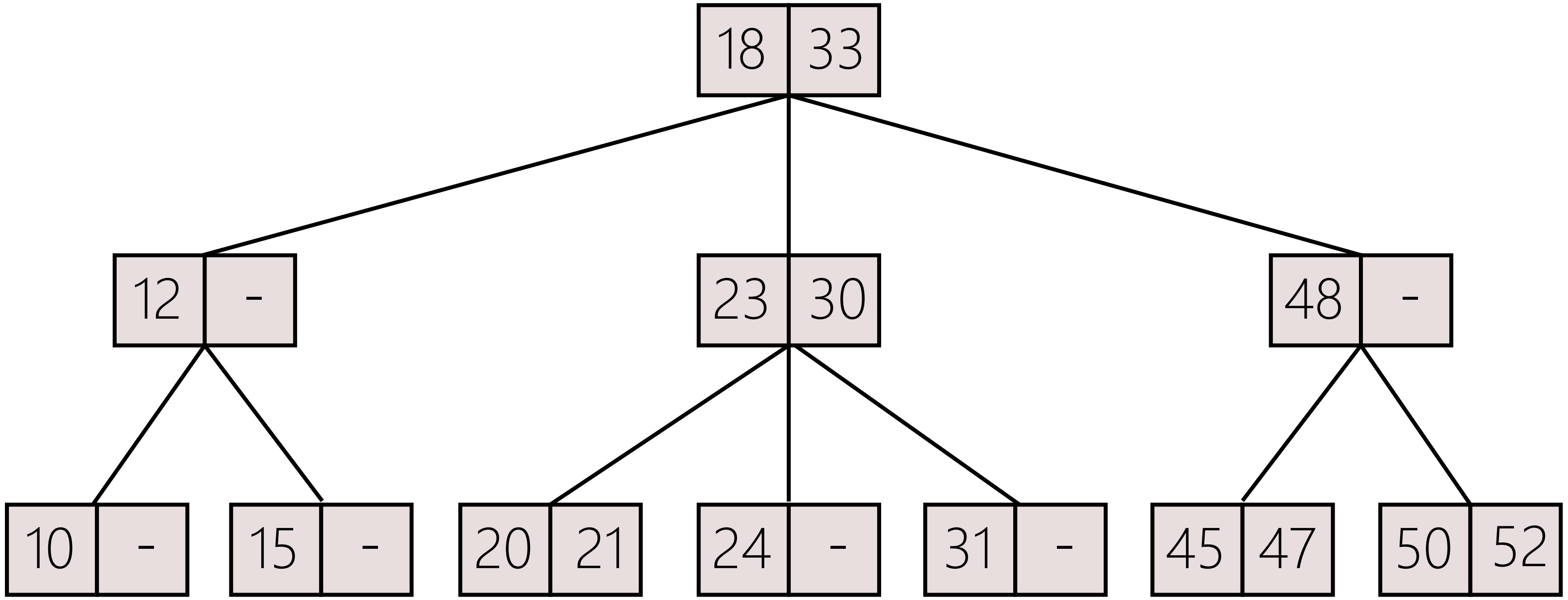
- Actual records are in the *leaves*
  - B^+-tree of order 3



- The keys are in the leaves
  - linearly ordered from left to right

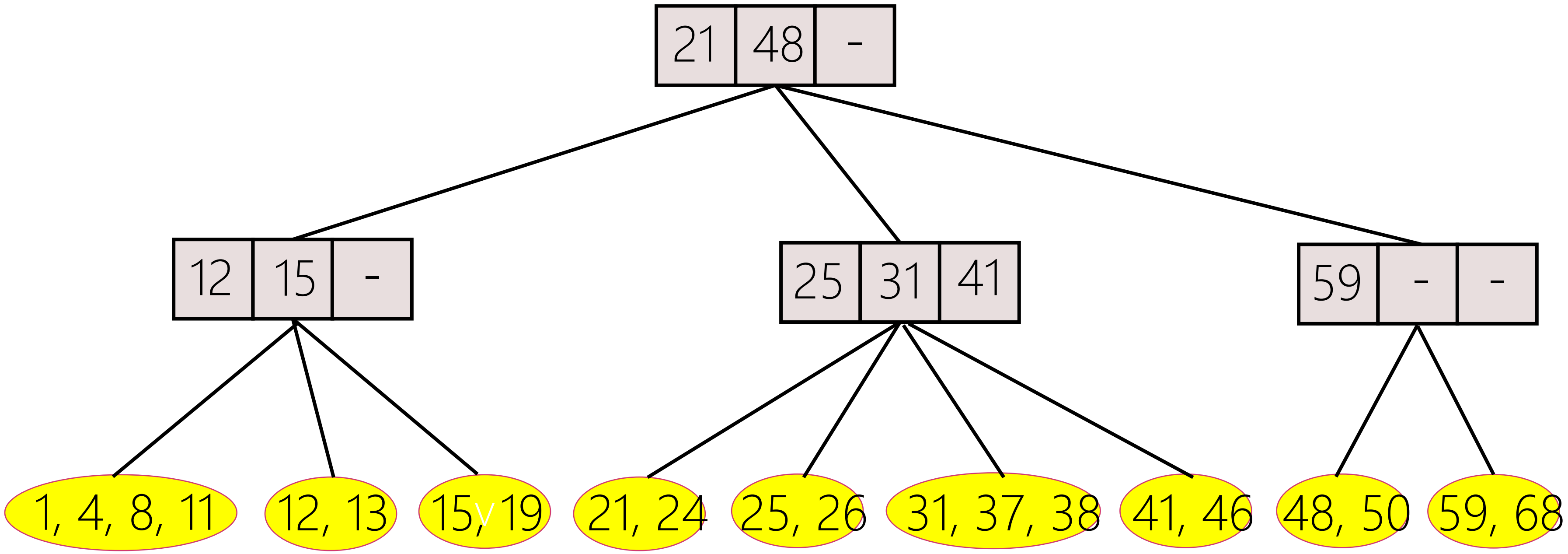
2-3 Tree: Alternative Structure

- Actual records are stored in both *leaves* and *internal nodes*
  - B-tree of order 3



2-3-4 Tree: Example

- B^+-tree of order 4



- A leaf node in a B^+-tree of order *m*
  - may store more or less than *m* (generally, between $\lceil m/2 \rceil$ *and m)* actual records

2-3 Tree: Height Bound

- 2-3 tree of h levels (of h-1 height) has
  - At least 2^{h-1} leaves (since each internal node has at least 2 children)
  - At most 3^{h-1} leaves (since each internal node has at most 3 children)

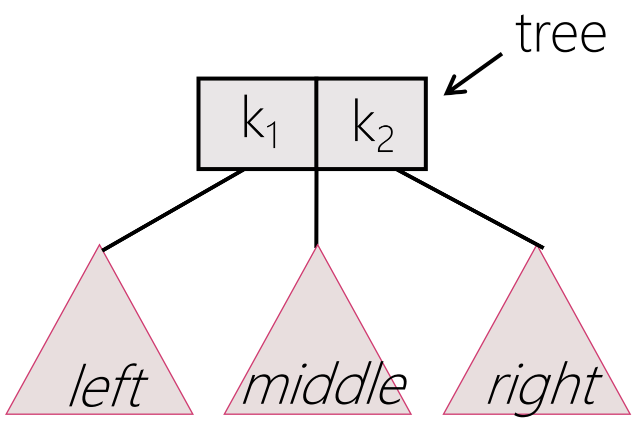
- Let n: the # of leaves in a 2-3 tree
  - $2^{h-1} \leq n \leq 3^{h-1}$
  - $(\log_3 n + 1) \leq h \leq (\log_2 n + 1)$

O(log n)

➔ Path length (height) = ?

2-3 Tree: Search

- *search(x, tree)*
  - If $x < k_1$, *search(x, left_subtree)*
  - If $k_1 < x < k_2$, *search(x, middle_subtree)*
  - If $x > k_2$, *search(x, right_subtree)*`
  - If $x = k_1$ or k_2
    - then the search key x is in the tree



- Search by simply moving down the tree

O(log n)

- Time complexity = ?

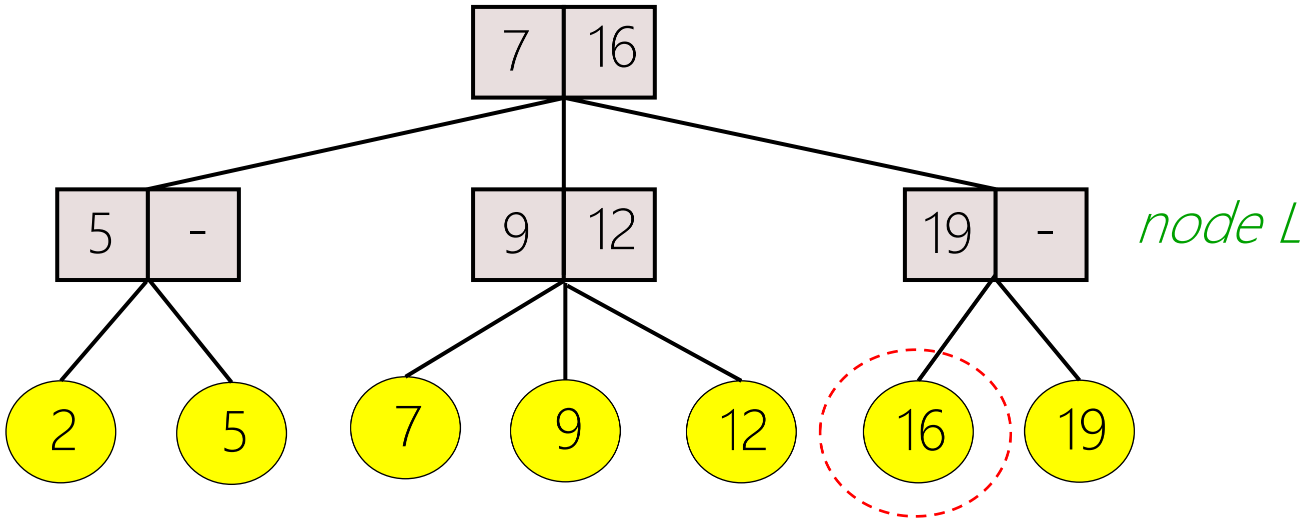
2-3 Tree: Insert

- Find a node L
  - One of whose children would contain the key if it were in the tree
- If L is not full
  - Then the new key is added
- If L is already full
  - Split L into two nodes L and L' (dividing the keys evenly among the two nodes)
  - Promote a copy of the *least-valued key* in L'

- Promotion may cause the parent to split in turn
  - Perhaps eventually leading to splitting the root (causing the 2-3 tree to gain a new level)

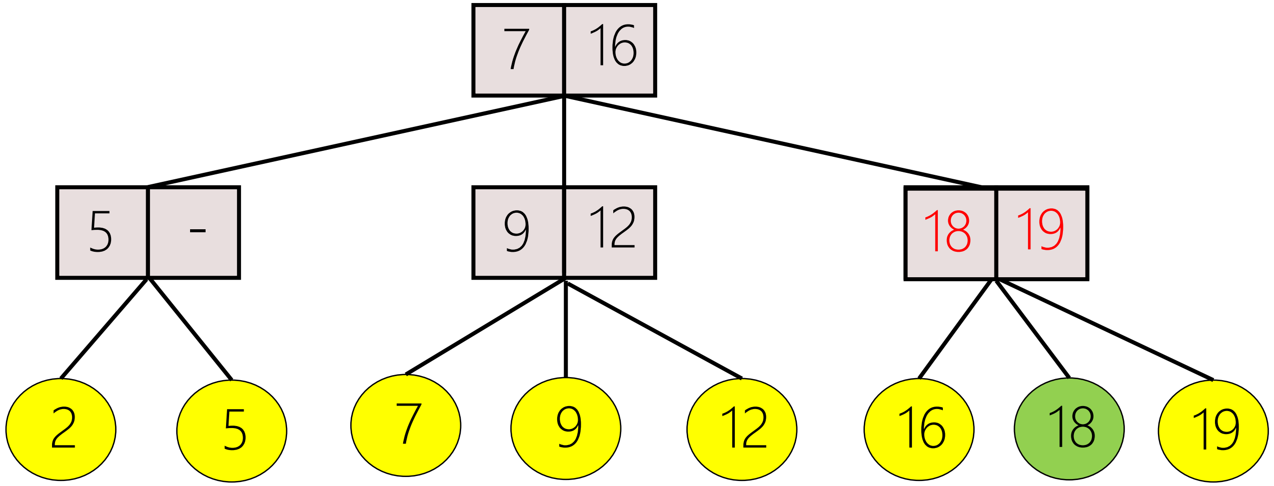
2-3 Tree: Insert - Example

- Insert a record with key value 18
  - Find the node *L*, one of whose children would contain 18 if it were in the tree
  - Since the parent *L* is not full, the key 18 can be added with no further modification to the tree



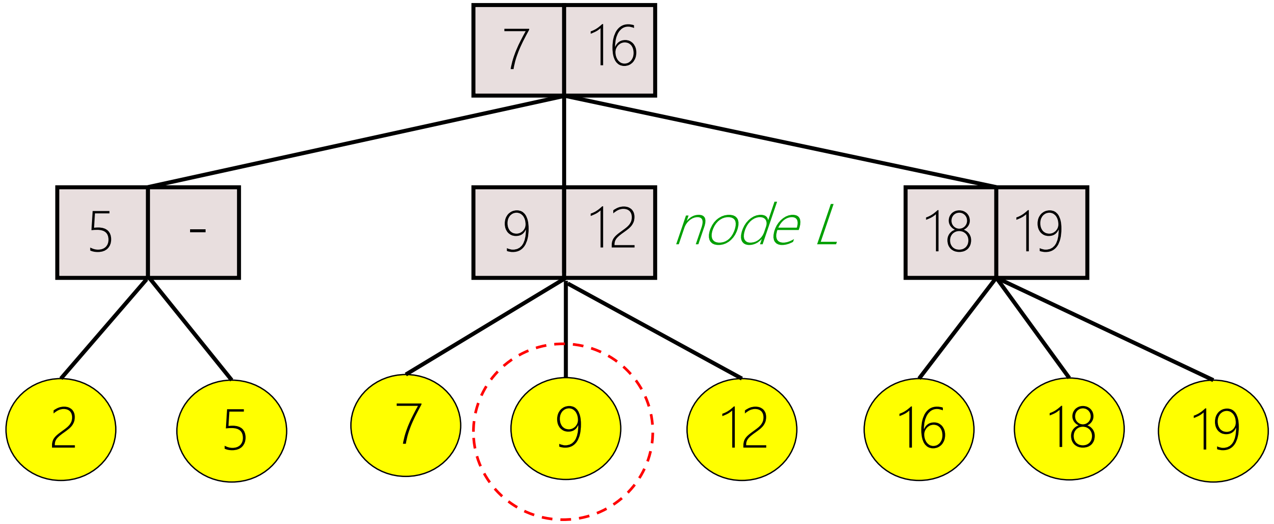
2-3 Tree: Insert - Example

- Insert a record with key value 18
  - Find the node *L*, one of whose children would contain 18 if it were in the tree
  - Since the parent *L* is not full, the key 18 can be added with no further modification to the tree
  - Update the key values in the parents



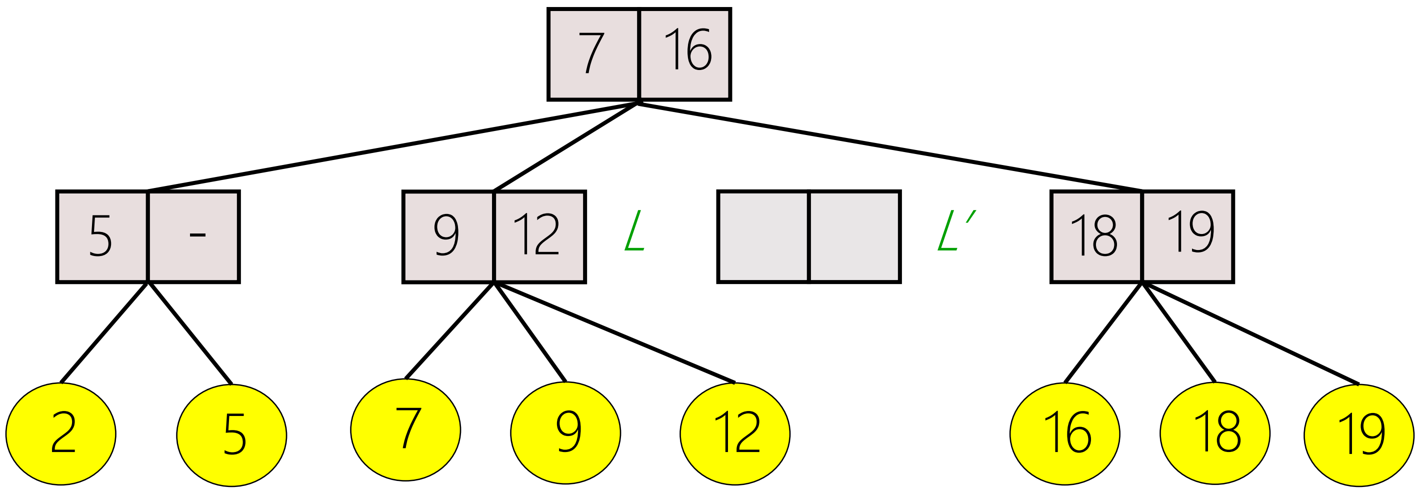
2-3 Tree: Insert - Example

- Insert a record with key value 10
  - Find the node *L*, one of whose children would contain 10 if it were in the tree
  - Since the parent *L* is already full, it has to be split (into *L* & *L'*)



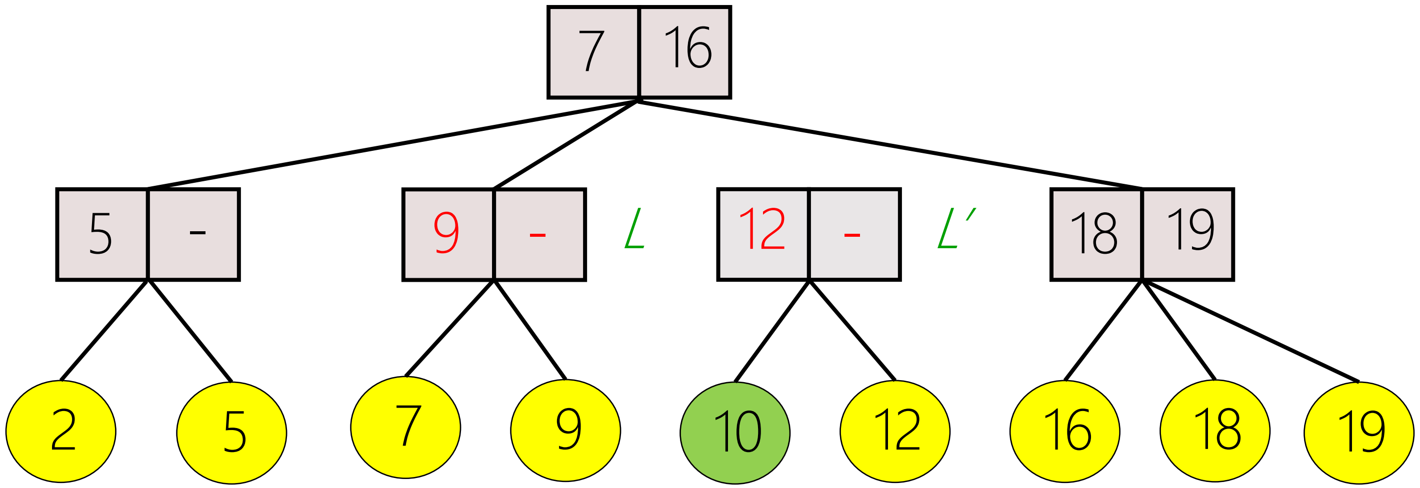
2-3 Tree: Insert - Example

- Insert a record with key value 10
  - Find the node *L*, one of whose children would contain 10 if it were in the tree
  - Since the parent *L* is already full, it has to be split (into *L* & *L'*)
    - Next rearrange the keys evenly among the two nodes
    - Update the key values in the parents



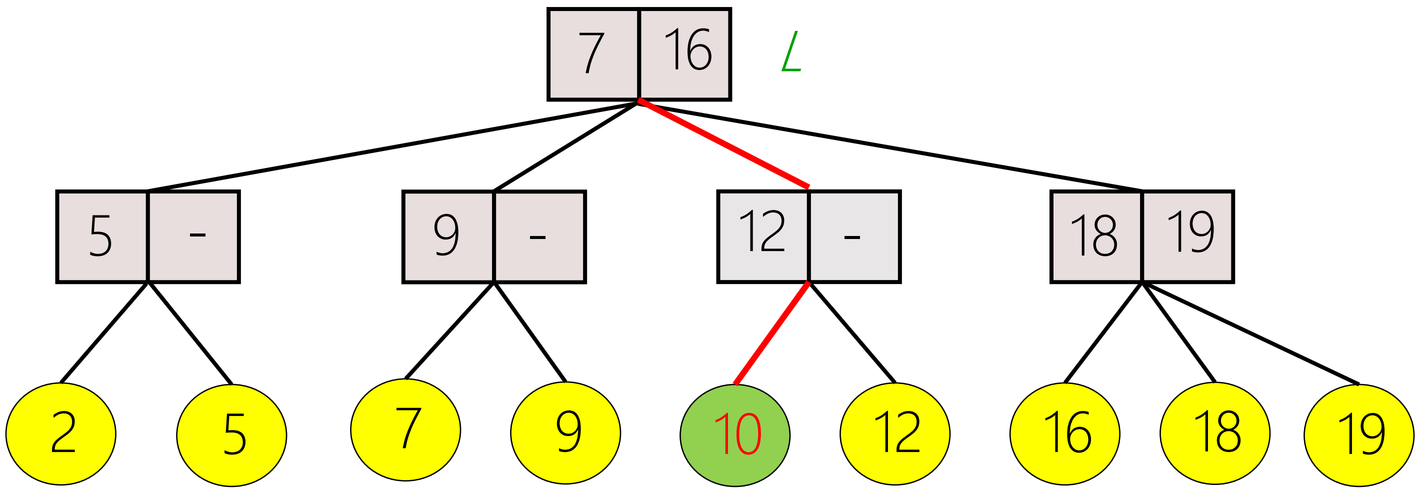
2-3 Tree: Insert - Example

- Insert a record with key value 10
  - Find the node *L*, one of whose children would contain 10 if it were in the tree
  - Since the parent *L* is already full, it has to be split (into *L* & *L'*)
    - Next rearrange the keys evenly among the two nodes
    - Update the key values in the parents
  - Promote a copy of the *least-valued key 10* in *L'*



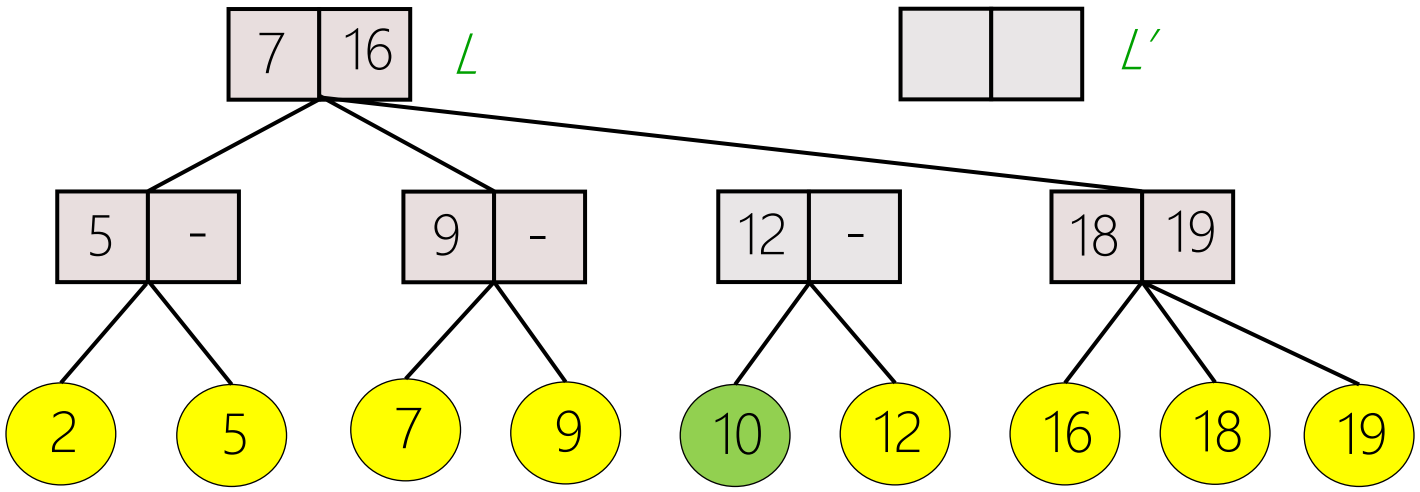
2-3 Tree: Insert - Example

- Insert a record with key value 10
  - Promote a copy of the *least-valued key 10* in *L′*
  - Grandparent is already full
    - So the *promoted key 10* cannot be inserted
  - Split-and-promote process again
    - Split *L* into *L* & *L′*



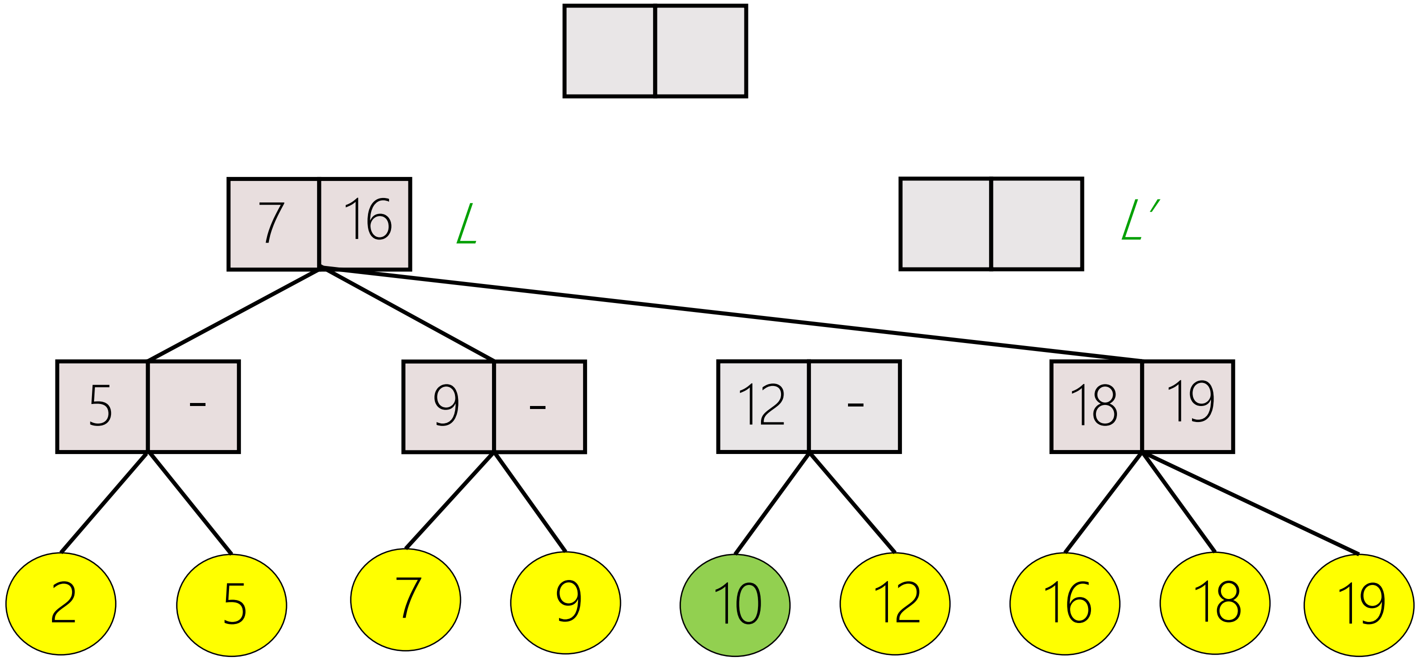
2-3 Tree: Insert - Example

- Insert a record with key value 10
  - Split-and-promote process again
    - Since L is the root, a new root has to be created as well



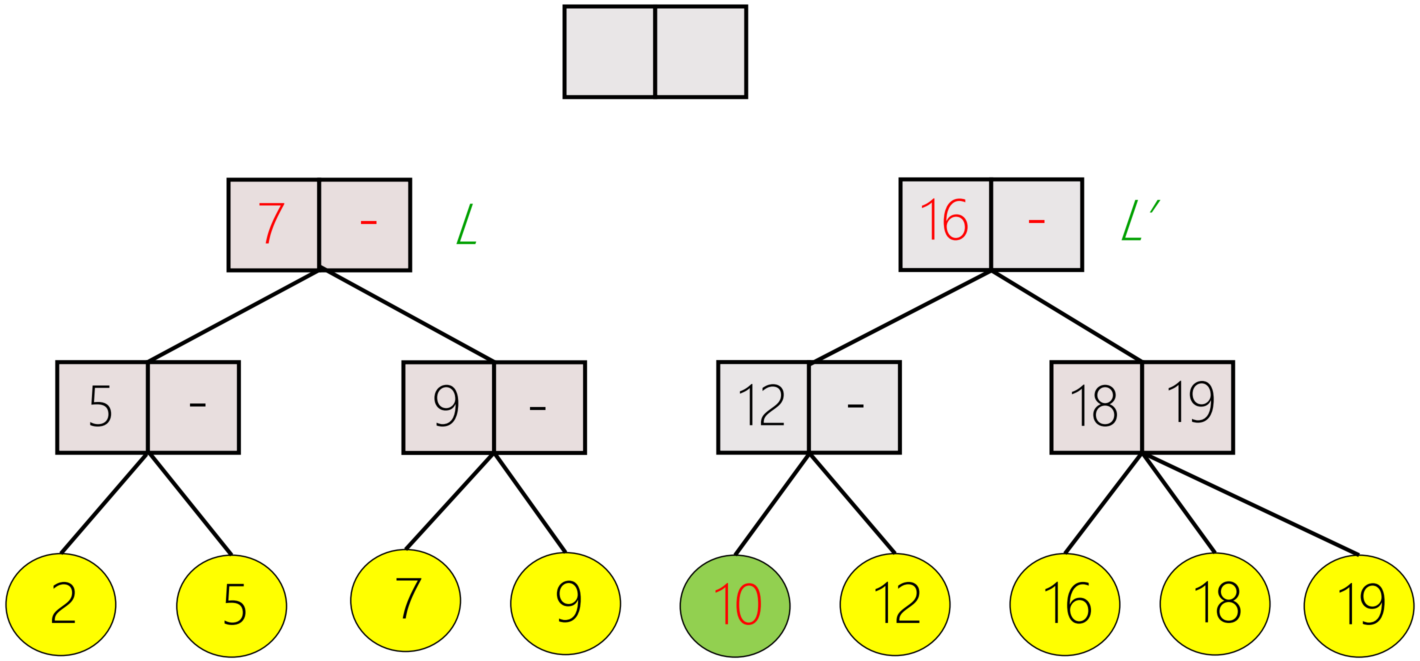
2-3 Tree: Insert - Example

- Insert a record with key value 10
  - Split-and-promote process again
    - Next rearrange the keys evenly among the two nodes
    - Update the key values in the parents



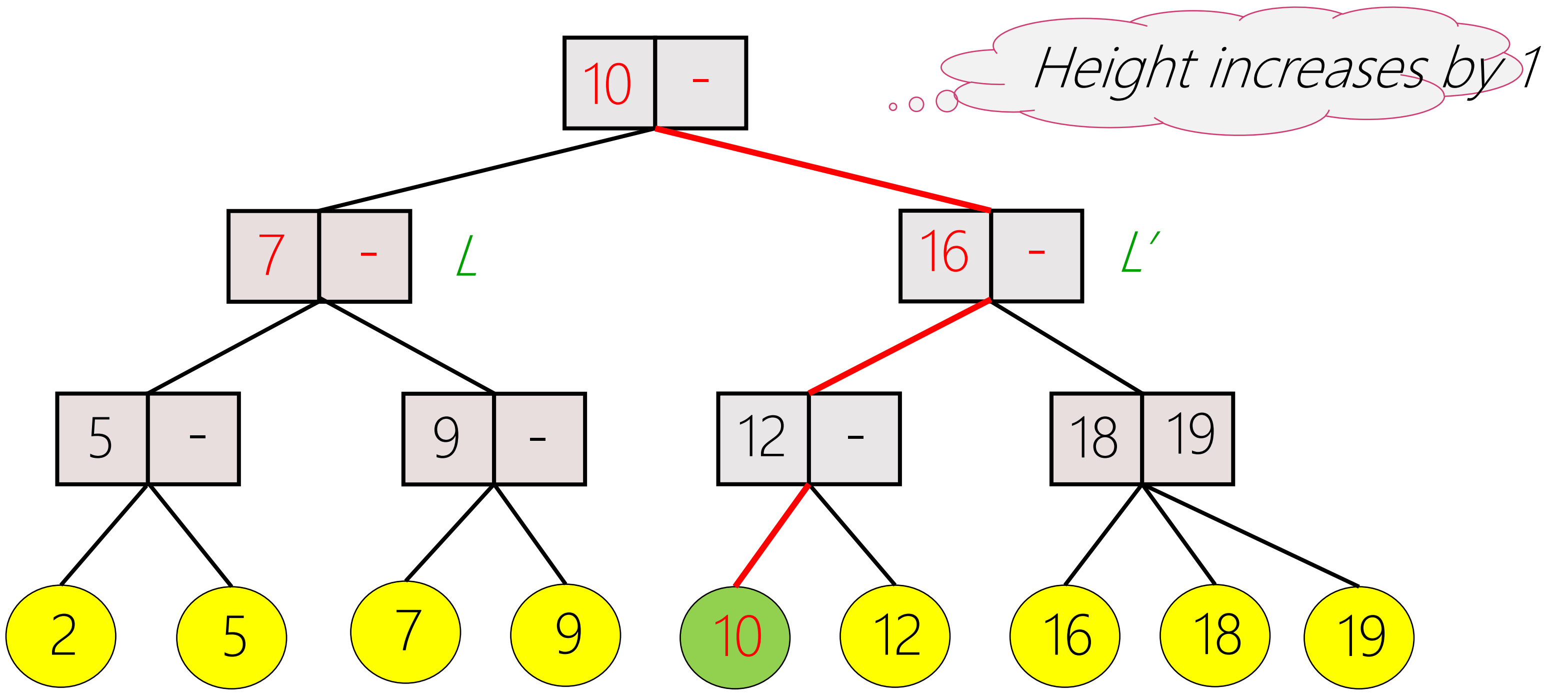
2-3 Tree: Insert - Example

- Insert a record with key value 10
  - Split-and-promote process again
    - Promote a copy of the least-valued key 10 in L'



2-3 Tree: Insert - Example

- Insert a record with key value 10
  - Split-and-promote process again
    - The insertion is complete



2-3 Tree: Delete

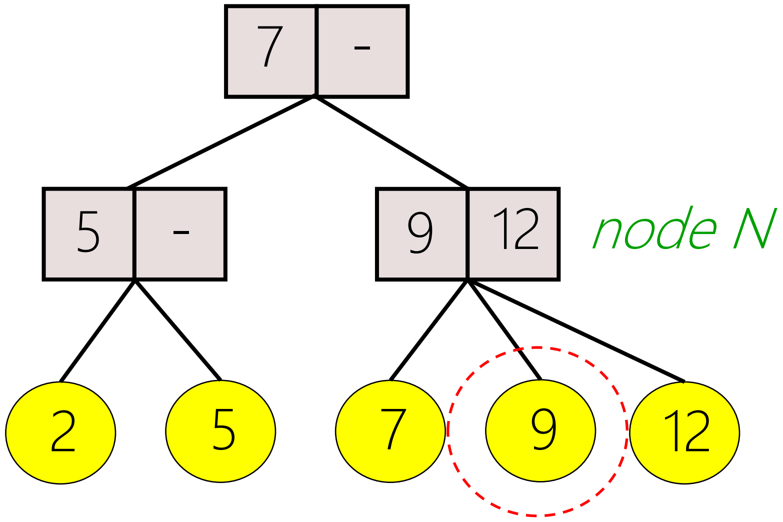
- Locate *node N*, one of whose children is *key K* to be deleted

- Three cases to consider:
  - If *N* has 3 children
    - Just remove *K*

  - If *N* has 2 children (Underflow)
    - Look at *N*'s adjacent siblings to determine if they have a spare key that can be used to fill the gap
    - If not (i.e., neither sibling can lend a key to the underflow node *N*)
      - *N* must give its key to a sibling and be removed from the tree

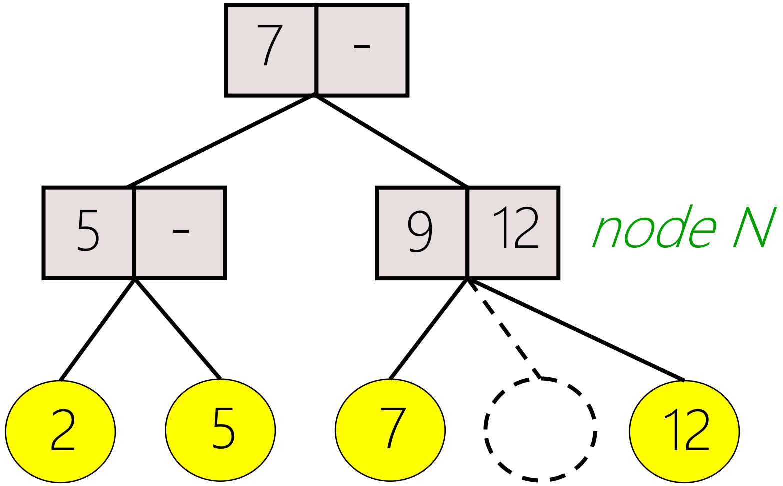
2-3 Tree: Delete (Case-1: Three Children)

- Delete a record with key value 9
  - Find the *node N*, one of whose children is *key 9* to be deleted
  - Remove the record



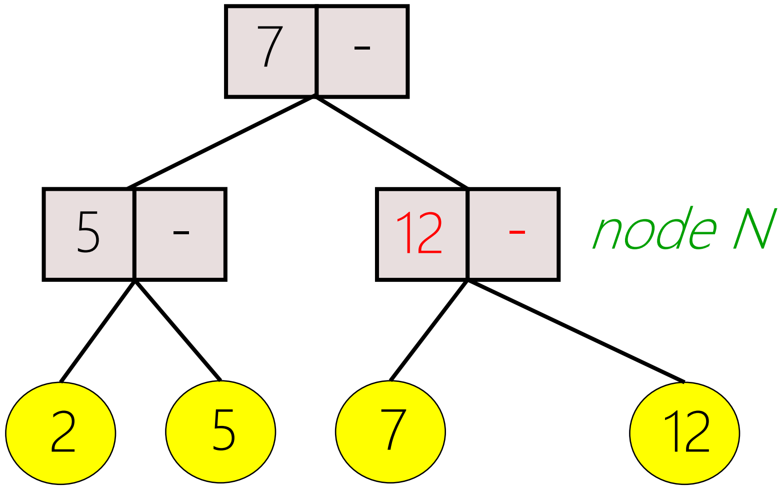
2-3 Tree: Delete (Case-1: Three Children)

- Delete a record with key value 9
  - Update the key value of the parent node *N*



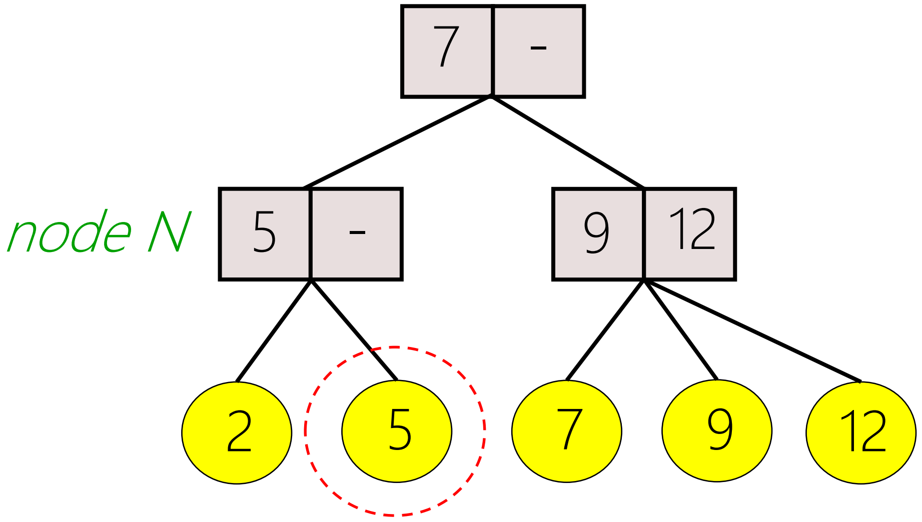
2-3 Tree: Delete (Case-1: Three Children)

- Delete a record with key value 9
  - Update the key value of the parent node *N*
  - The deletion is complete



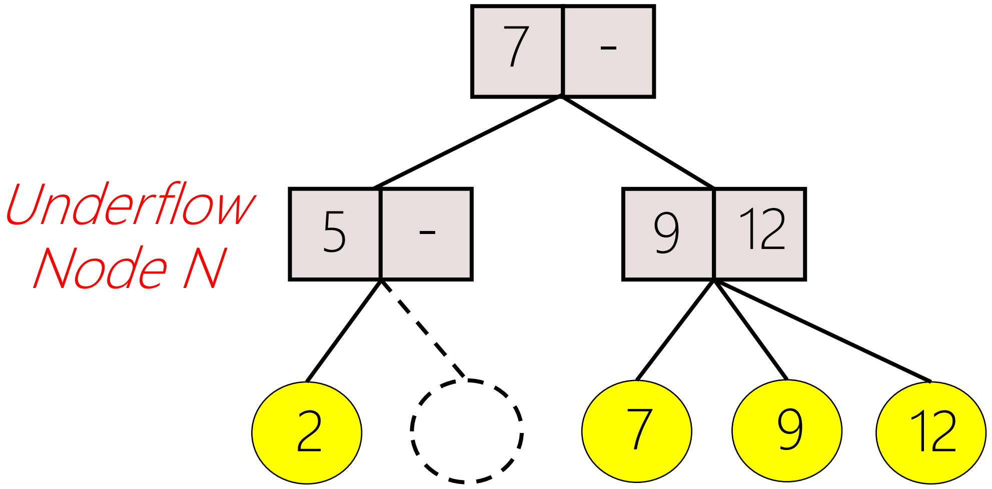
2-3 Tree: Delete (Case-2: Two Children)

- Delete a record with key value 5
    - Find the *node N*, one of whose children is *key 5* to be deleted
    - Remove the record whose key is 5



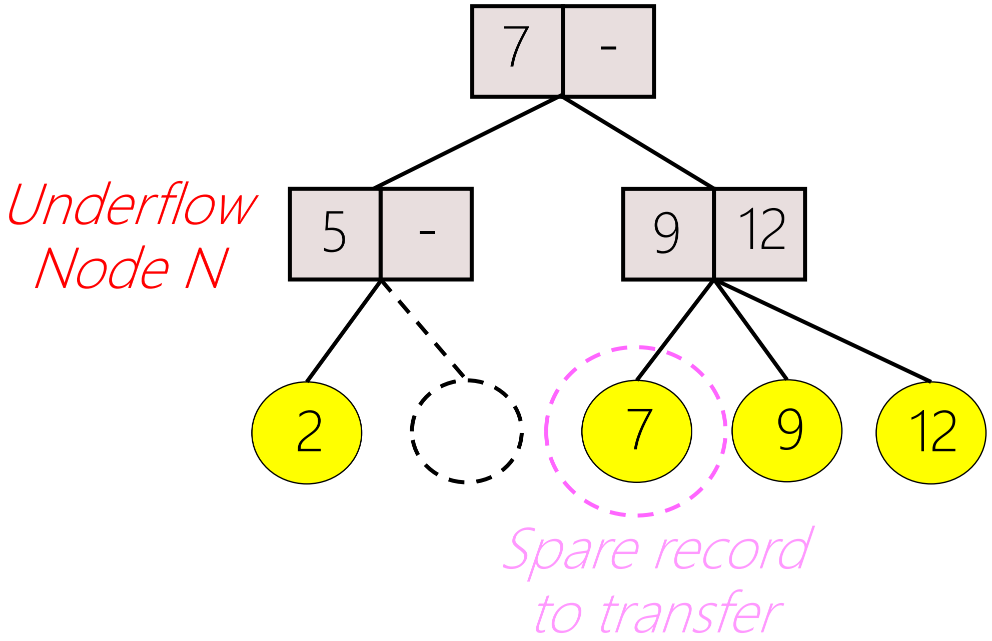
2-3 Tree: Delete (Case-2: Two Children)

- Delete a record with key value 5
  - Deleting a record causes the parent node *N* to underflow
  - Look for *N*s adjacent sibling that can lend a spare key to it



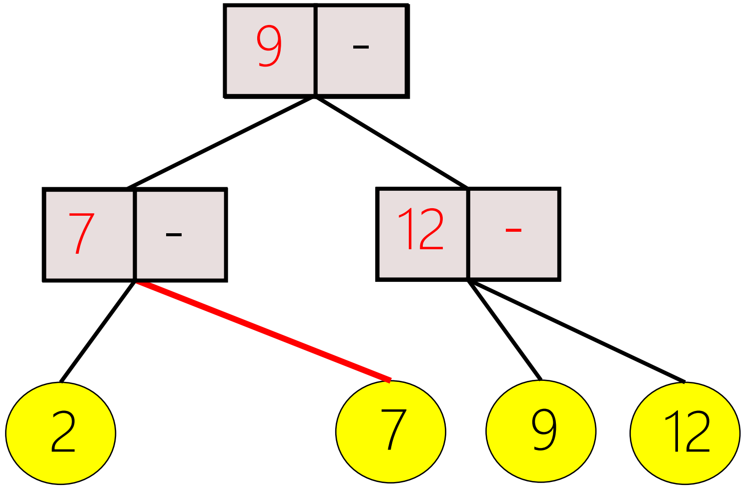
2-3 Tree: Delete (Case-2: Two Children)

- Delete a record with key value 5
  - Transfer the spare record to the underflow node *N*



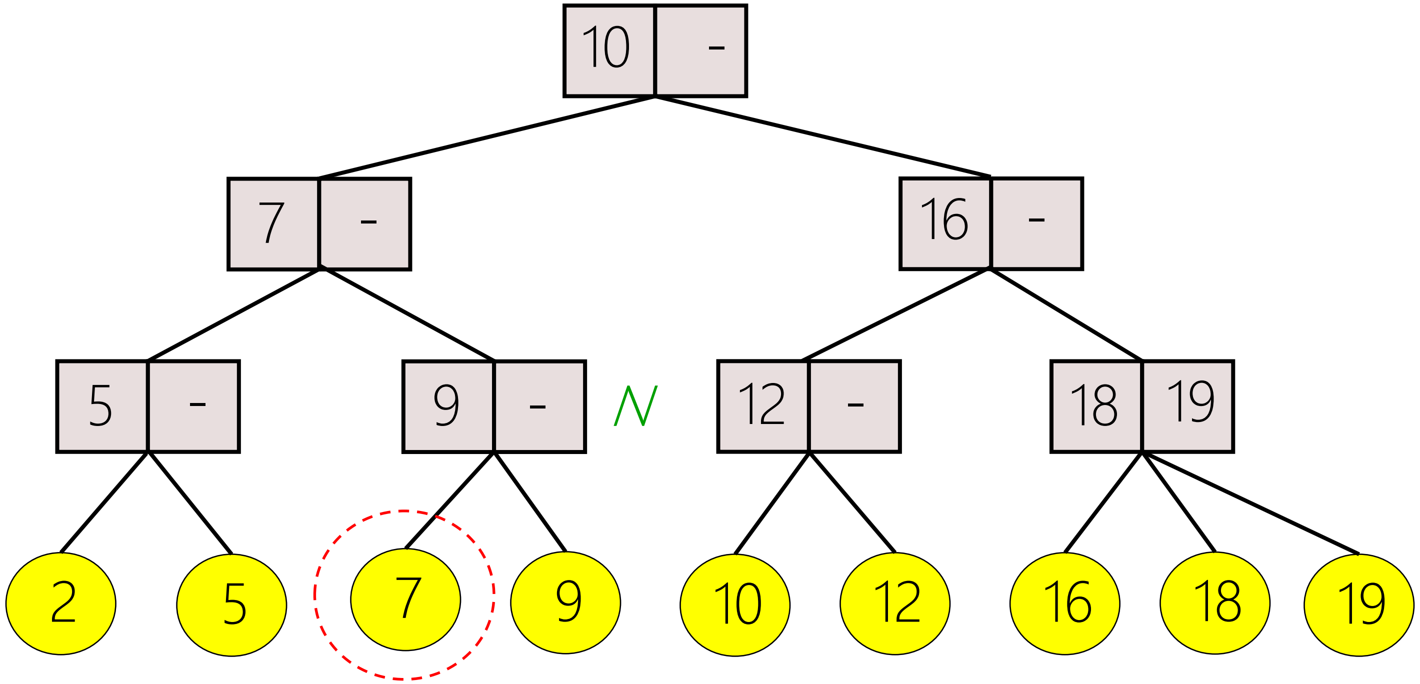
2-3 Tree: Delete (Case-2: Two Children)

- Delete a record with key value 5
  - Transfer the spare record to the underflow node *N*
  - Update the key value of the parent node
  - The deletion is complete



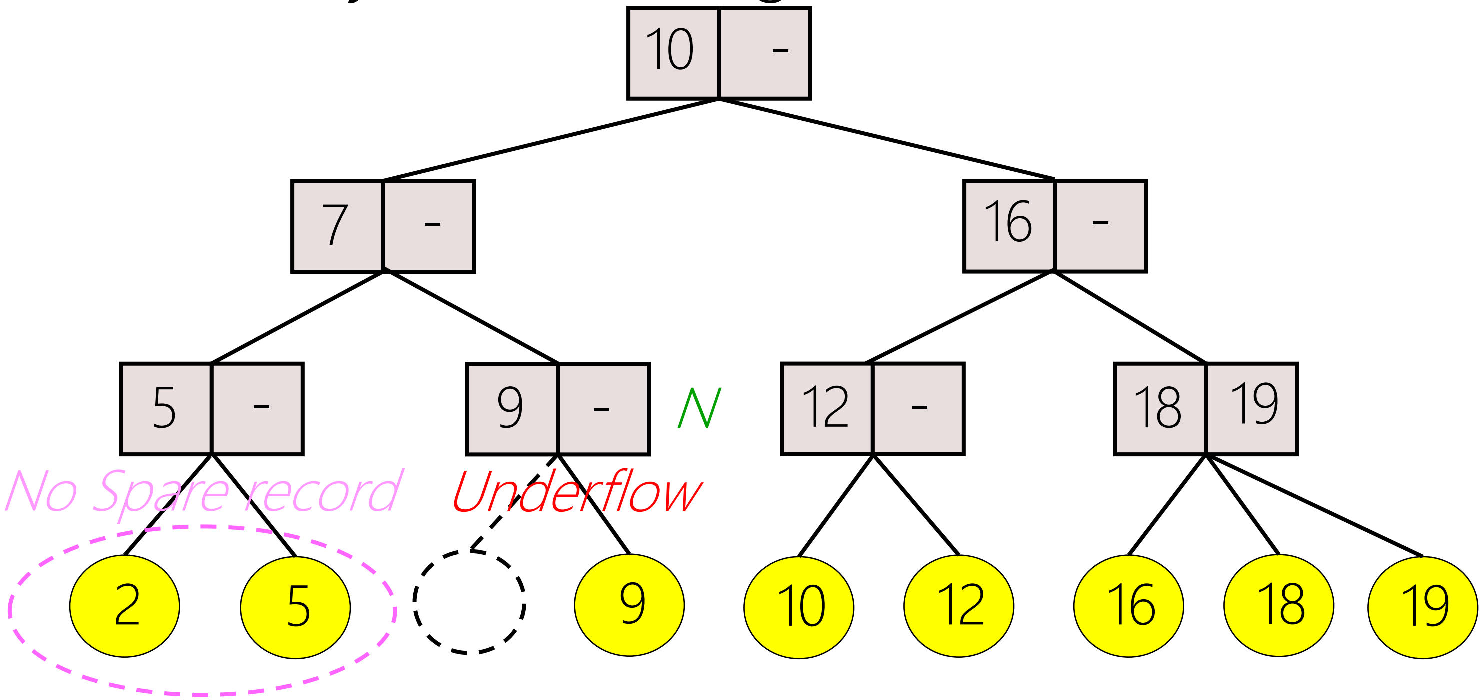
2-3 Tree: Delete (Case-3: No Siblings with a Spare Record)

- Delete a record with key value 7
  - Find the *node N*, one of whose children is *key 7* to be deleted
  - Remove the record whose key is 7



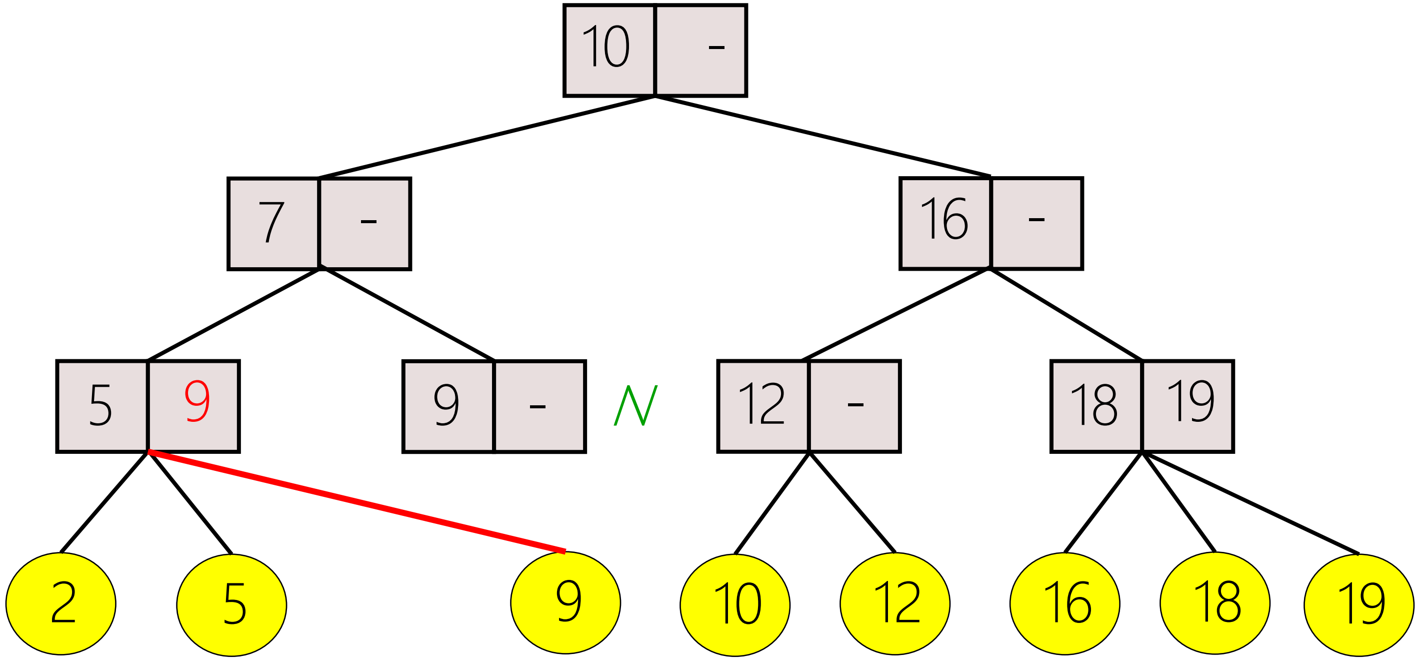
2-3 Tree: Delete (Case-3: No Siblings with a Spare Record)

- Delete a record with key value 7
  - Deleting a record causes the parent node *N* to underflow
  - No adjacent siblings that have a spare record
  - Transfer the record (*key 9*) to a sibling



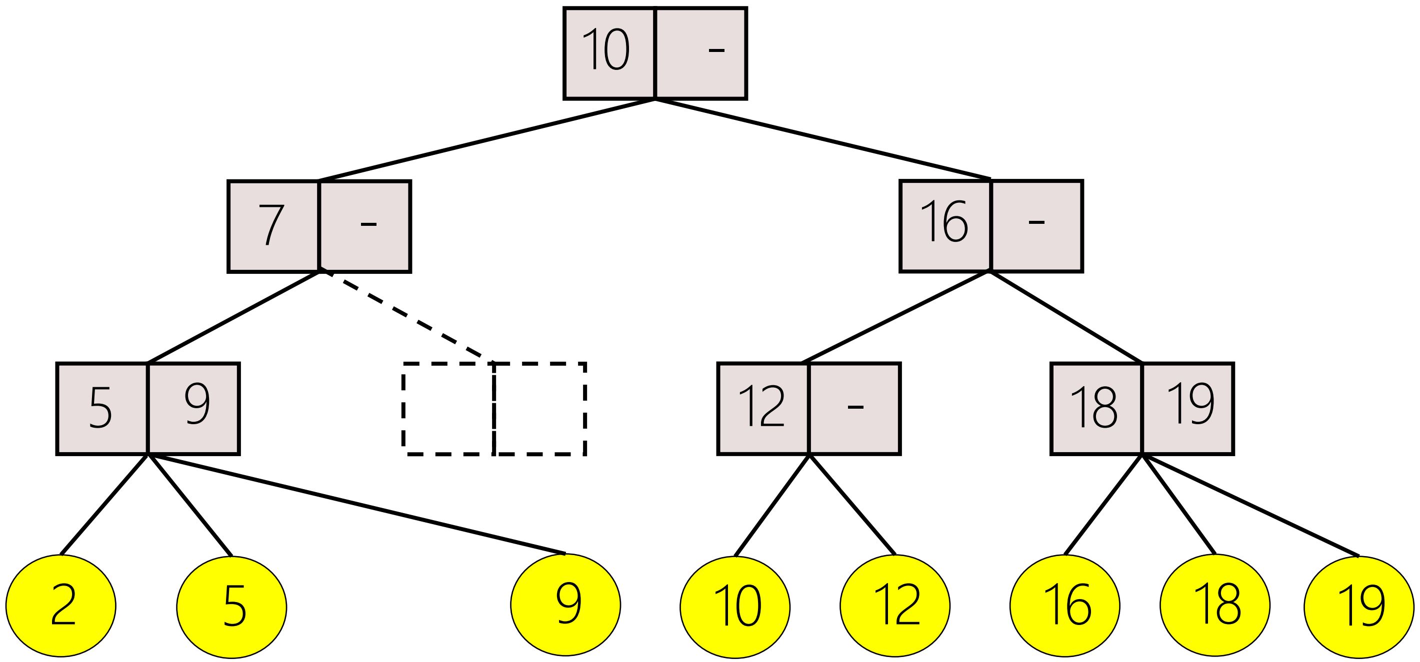
2-3 Tree: Delete (Case-3: No Siblings with a Spare Record)

- Delete a record with key value 7
  - Remove the parent node *N* recursively



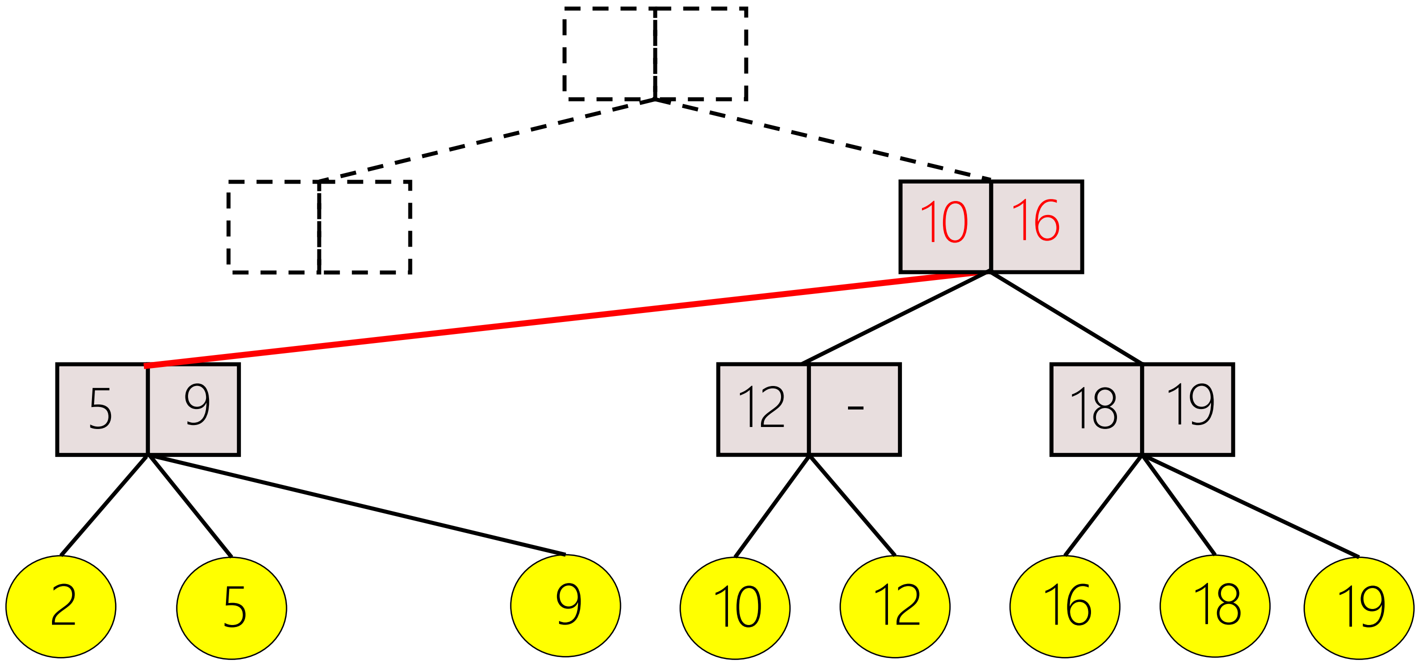
2-3 Tree: Delete (Case-3: No Siblings with a Spare Record)

- Delete a record with key value 7
  - Remove the parent node *N* recursively
    - Node-merge deletion process



2-3 Tree: Delete (Case-3: No Siblings with a Spare Record)

- Delete a record with key value 7
  - Remove the parent node *N* recursively
    - Node-merge deletion process



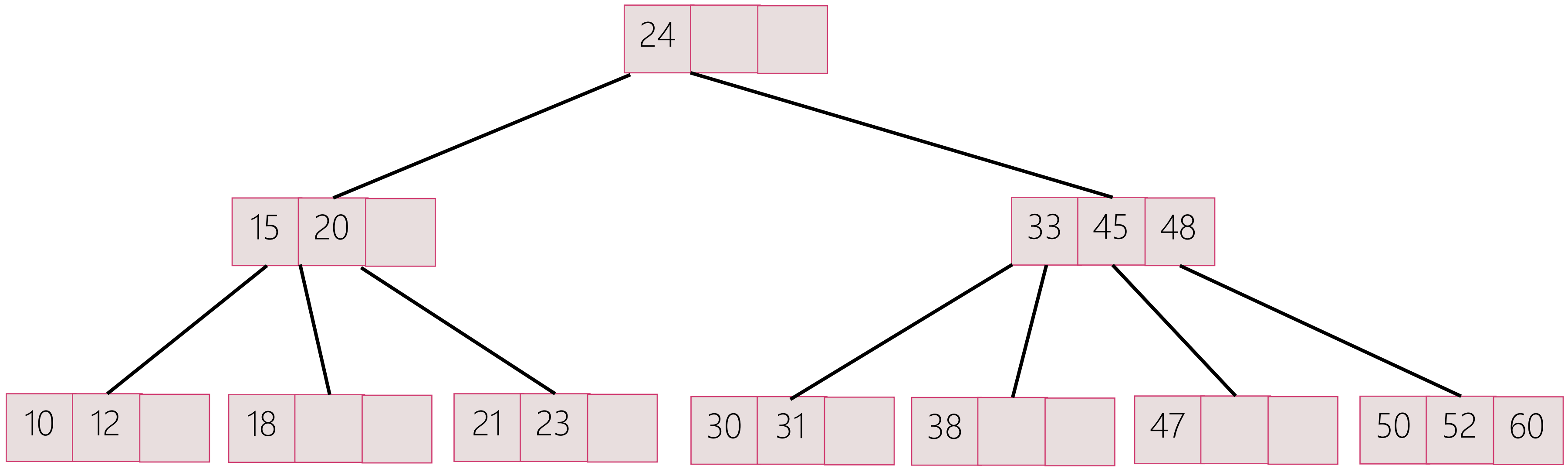
B-tree (of order *m*)

- *m*-way search tree satisfying:

Shape Property

- Root has at least 2 children
- All internal nodes except root have at least $\lceil m/2 \rceil$ children
- All leaf nodes are at the same level

B-tree: Example

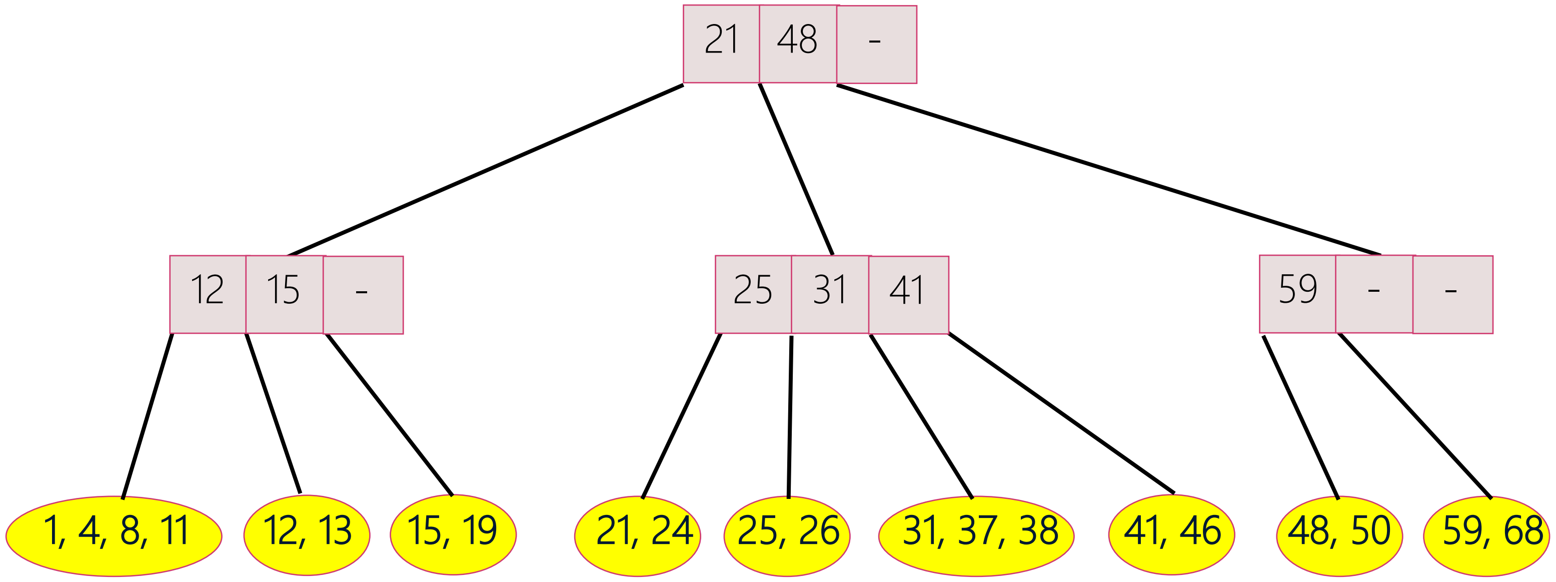


- B-tree of order 4
  - 2-3-4 tree
  - Search tree property + Shape property
- Height Bound = O(log n)

B-tree

- Analysis
  - Insert, Delete, Search: *O(log n)*
    - where the log base is the average branching factor of the tree
- Implementation of B-tree node
  - Normally equivalent to a disk block
  - Typically allows 100 or more children
    - B-tree and its variants are extremely shallow

B^+-tree (of order 4)



- Actual records are stored only at the leaf nodes
  - A leaf node may store more or less than *m* (generally, between $\lceil m/2 \rceil$ & *m*) actual records

B*-tree

- B^+-tree variants
  - Each node is at least 2/3 full

Comparison of B-trees with Sorted Arrays

- Disadvantages
  - More space overhead
  - More complex implementation

- Advantages
  - Updates are much easier
  - Search time is generally faster

References

- Further reading list and references
  - https://en.wikipedia.org/wiki/B-tree
  - https://en.wikipedia.org/wiki/2%E2%80%933_tree

- Slide credit
  - Jaesik Park
  - Seung-Hwan Baek
  - Jong-Hyeok Lee