Lecture 11: Searching algorithm (Part. 2)

Algorithm

Jeong-Hun Kim

Table of Contents

- Part 1
  - Red-black tree
- Part 2
  - B-tree
- Part 3
  - Multidimensional search tree
    - KD-tree
    - KDB-tree
    - R-tree
    - Grid file

Part 1

RED-BLACK TREE

Red-Black Tree

- Limitations of the binary search tree
  - Average time complexities for storage and search is $\boldsymbol{\Theta(\log n)}$
  - In the worst case, the tree becomes unbalanced
  - When unbalanced, the time complexity is close to $\boldsymbol{\Theta(n)}$

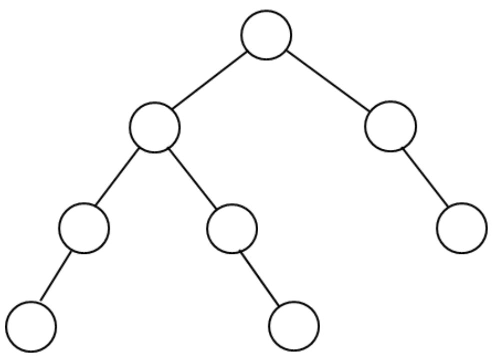
- Balanced tree
  - This always remains balanced
    - e.g., Red-Black tree, B-tree, AVL tree, etc.

Red-Black Tree

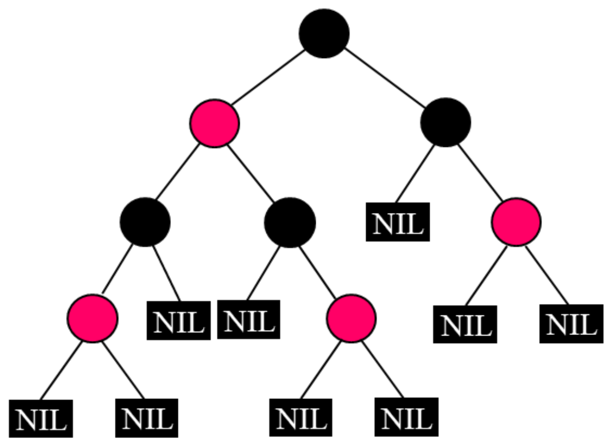
- Properties of a Red-Black tree
  - Every node has either a Red or Black color
    ① Every node is either Red or Black
    ② Root node is Black
    ③ All leaf nodes are Black
    ④ If a node is Red, its children must be Black (No double Red)
    ⑤ The number of Black nodes encountered on any path from the root node to a leaf node is the same
  - Here, the leaf node of Red-Black tree is different from a traditional leaf node
    - Assume all NIL pointers point to a leaf node called NIL

Red-Black Tree

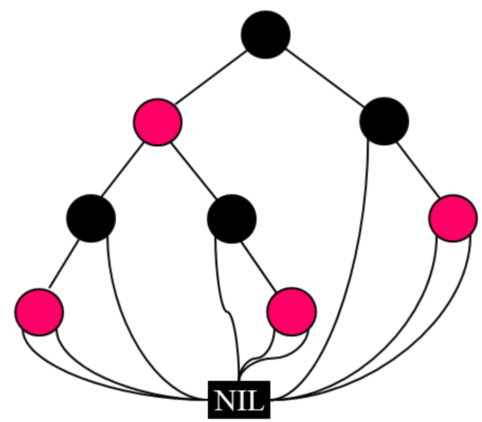
❖ Example of Red-Black tree



Binary search tree



Red-Black tree

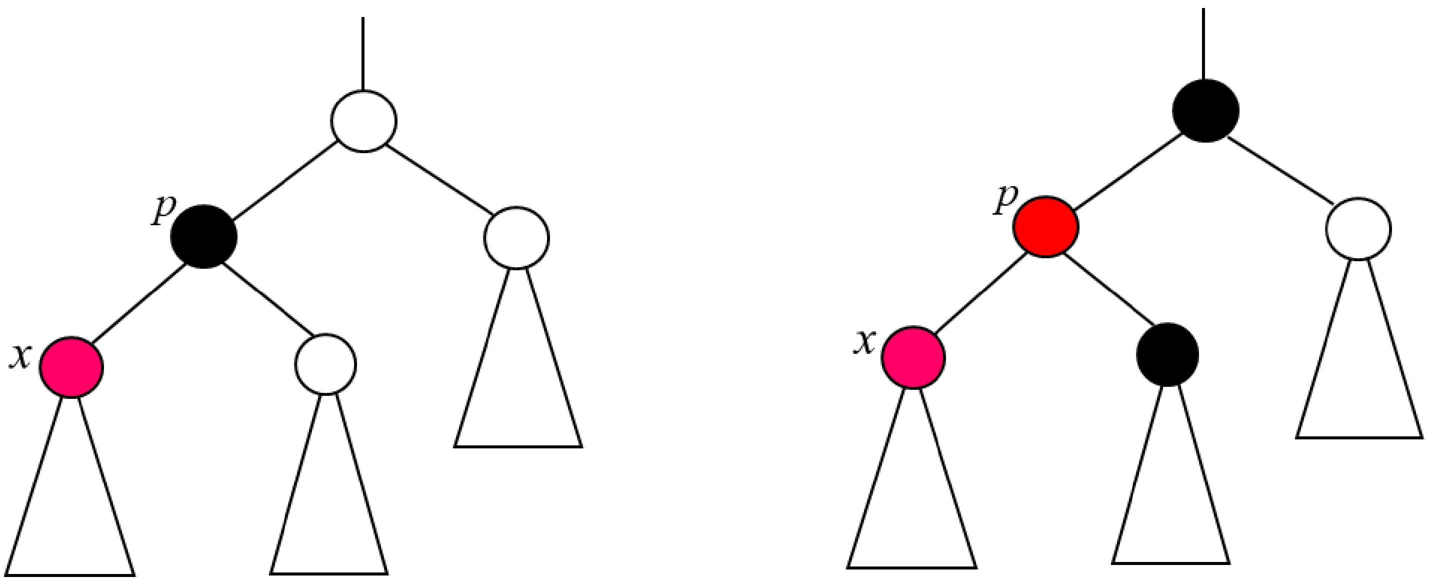


Implemented Red-Black tree

① Root node is Black
② All leaf nodes are Black
③ If a node is Red, its children must be Black
④ The number of Black nodes encountered on any path from the root node to a leaf node is the same

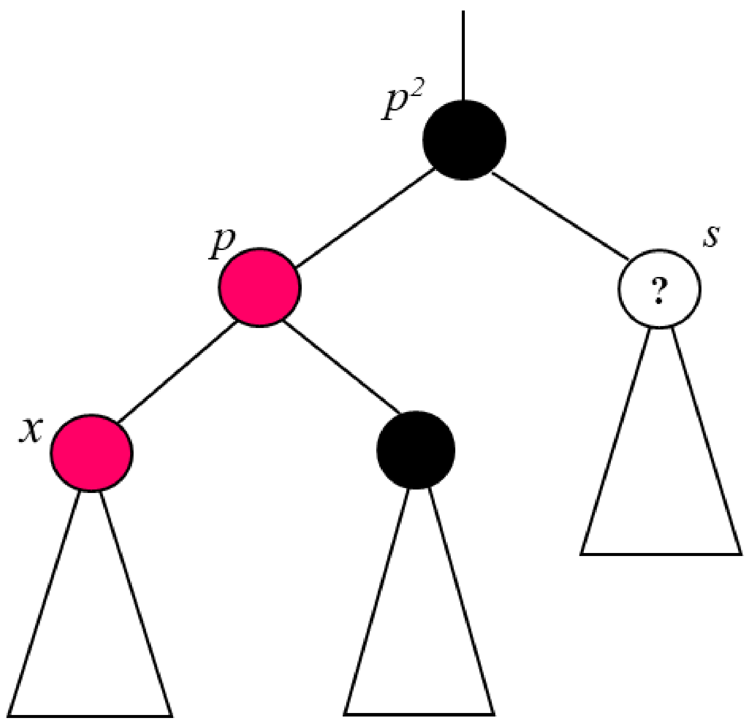
Red-Black Tree

- ❖ Insertion in a Red-Black tree
  - ▪ Fundamentally the same as a binary search tree
    - Mark the inserted node (node x) as Red after insertion
      - – If the color of node x's parent node p is
        - » Black: no problem
        - » Red: property ③ of the Red-Black tree is not satisfied



Red-Black Tree

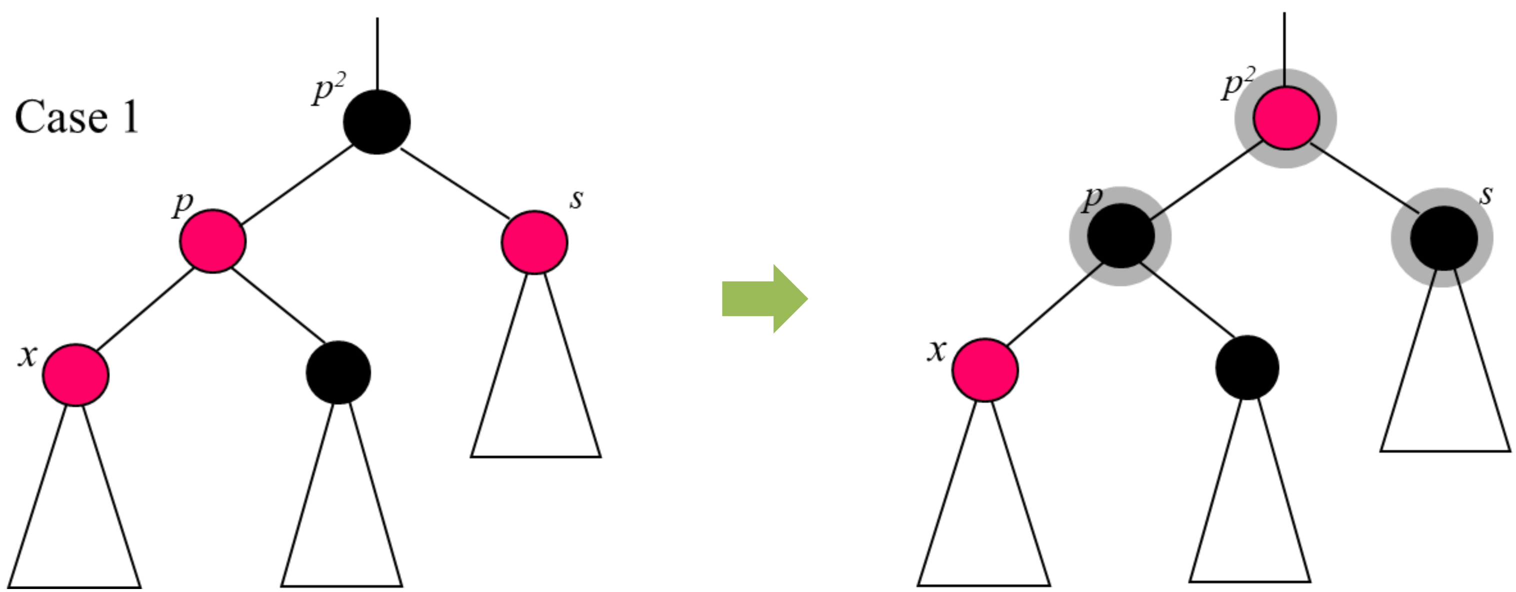
- Insertion in a Red-Black tree (cont'd)
  - The node p^2 and the sibling node of x must be Black
  - Divided into two cases depending on the color of node s
    - Case 1: node s is Red
    - Case 2: node s is Black



Red-Black Tree

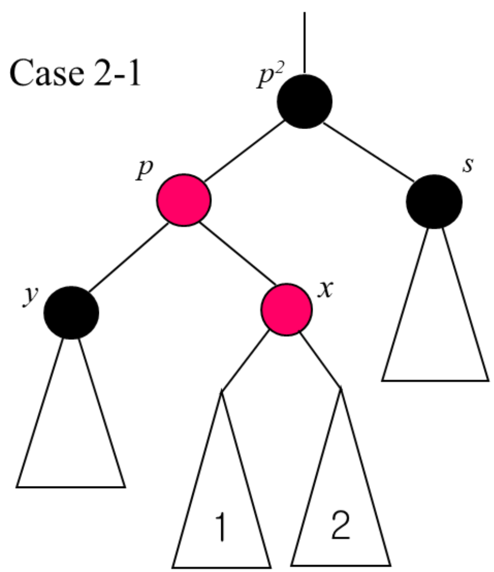
- Insertion in a Red-Black tree (cont'd)
  - Case 1: node s is Red
    - Change node p and its sibling node s to Black
    - However, the same issue may occur at node p^2
      - Solve recursively

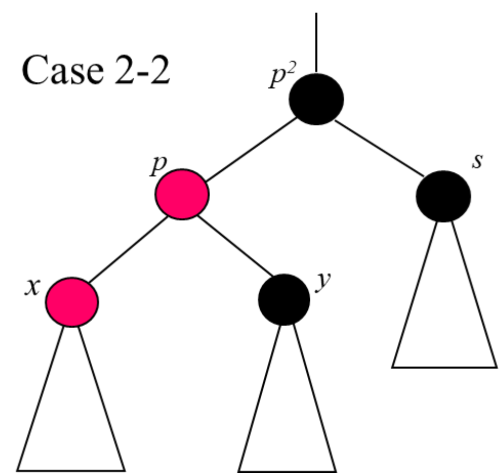
: nodes with changed colors



Red-Black Tree

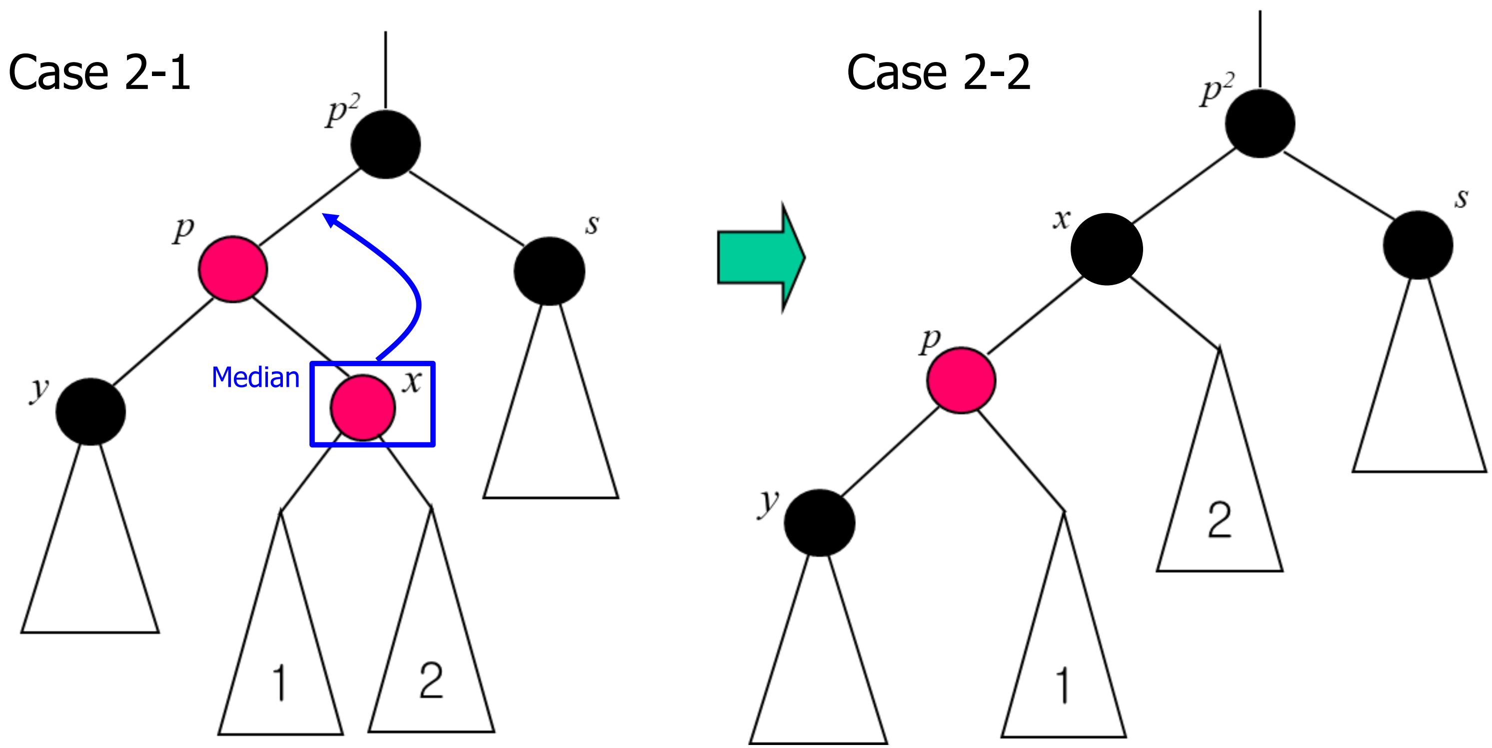
- Insertion in a Red-Black tree (cont'd)
  - Case 2: node s is Black
    - Case 2-1: node s is Black, and node x is the **right child** of parent node p
    - Case 2-2: node s is Black, and node x is the **left child** of parent node p





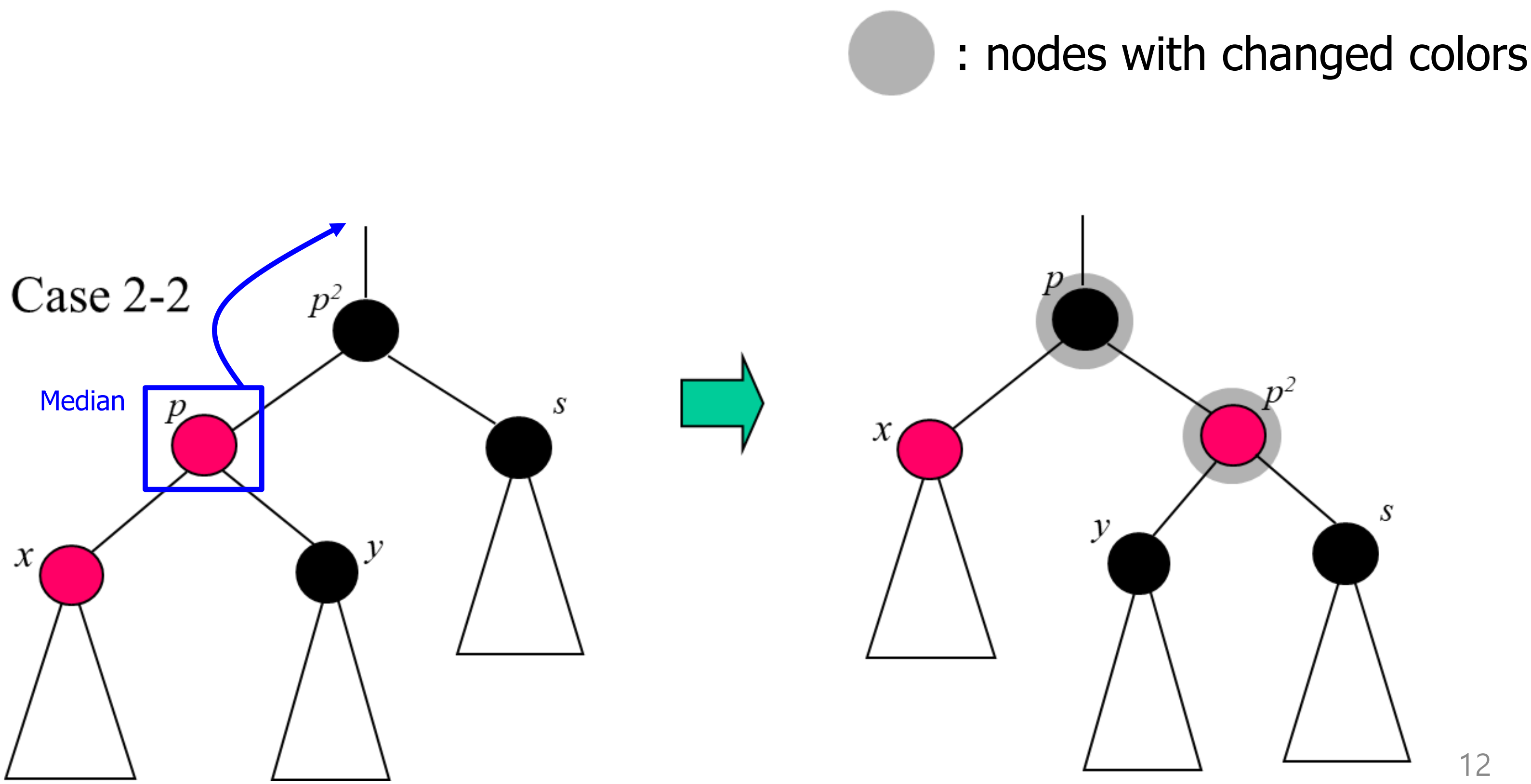
Red-Black Tree

- Insertion in a Red-Black tree (cont'd)
  - Case 2-1: node s is Black, and x is the **right child** of parent node p
    - Rotate left around node p
      - Violates property ③, transition to Case 2-2
    - New parent is Black, and the children are Red



Red-Black Tree

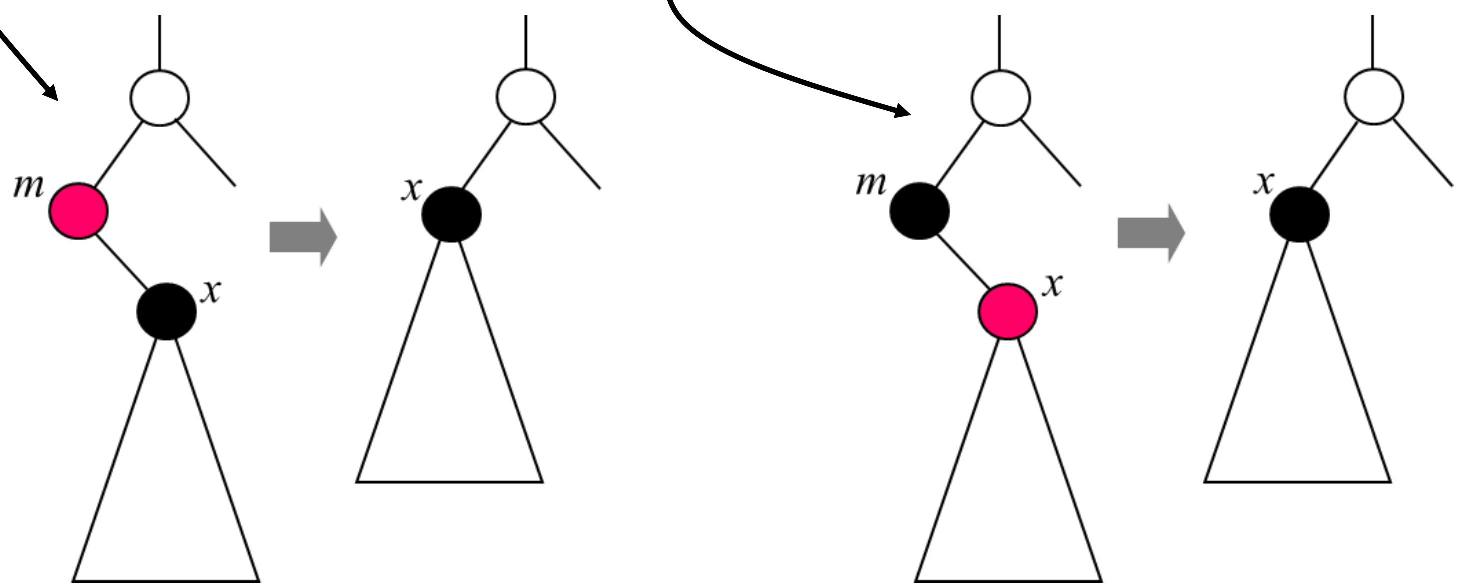
- Insertion in a Red-Black tree (cont'd)
  - Case 2-2: node s is Black, and x is the **left child** of parent node p
    - Rotate right around node p^2, and swap the colors of p and p^2



Red-Black Tree

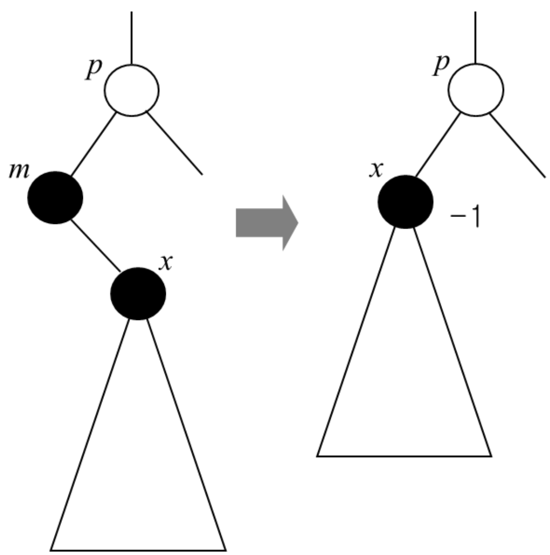
❖ Deletion in a Red-Black tree

- Deletion is straightforward if property ④ is satisfied
  - If the target is Red, there is no problem
  - Even if the target is Black, there is no problem if the only child is Red

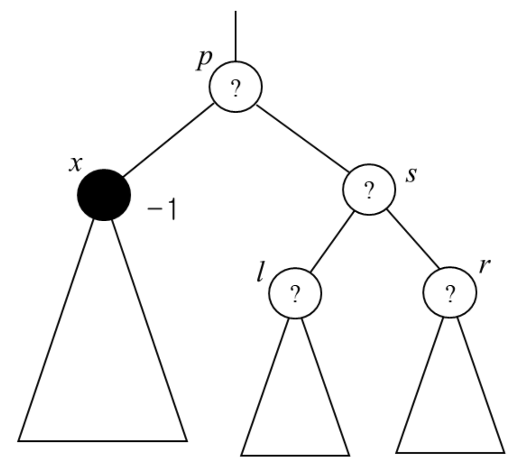


Red-Black Tree

- Deletion in a Red-Black tree (cont'd)
  - Deletion becomes complex if property ④ is violated
    - '-1' next to node x indicates that the number of Black nodes on the path from the root to the leaf through x is one less than required
    - Handled by dividing into a total of five cases



Problem occurs after deleting node m
(Property ④ is violated)



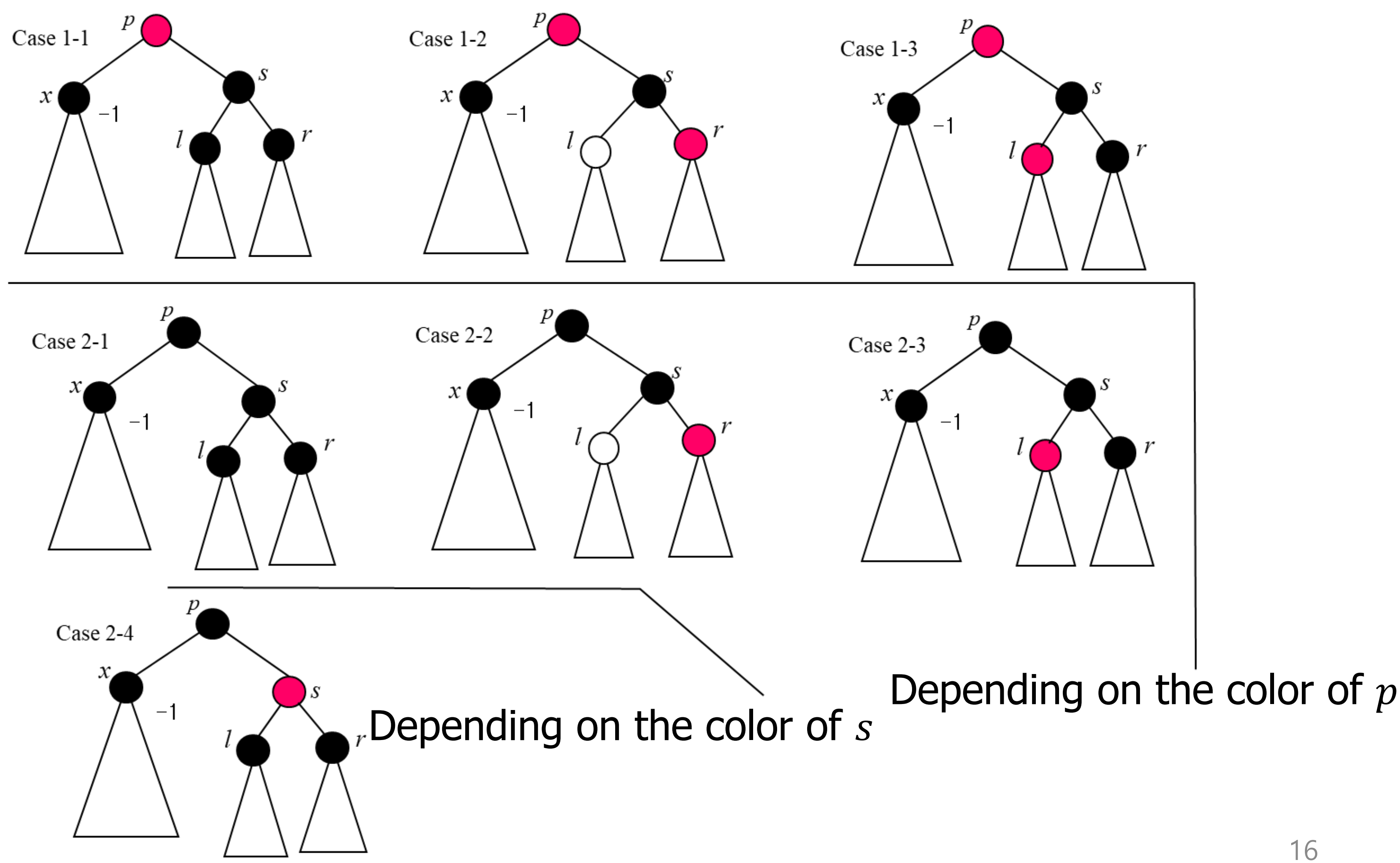
Handling method varies depending on x's surrounding conditions

Red-Black Tree

- Deletion in a Red-Black tree (cont'd)
  - Case 1: if node p is **Red** (node s must be Black) and depending on the <colors of l and r>
    - Case 1-1: <Black, Black>
    - Case 1-2: <*, Red>
    - Case 1-3: <Red, Black>
  - Case 2: if node p is **Black** and depending on the <colors of s, l, and r>
    - Case 2-1: <Black, Black, Black>
    - Case 2-2: <Black, *, Red>
    - Case 2-3: <Black, Red, Black>
    - Case 2-4: <Red, Black, Black>
      - If s is Red, then l and r must be Black

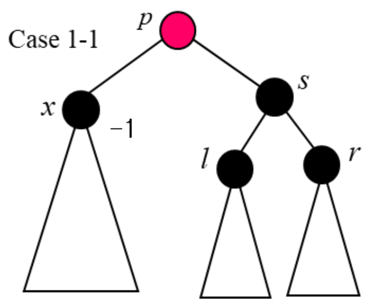
Red-Black Tree

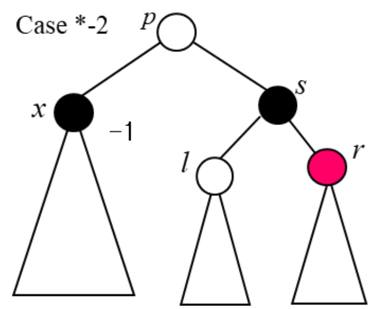
❖ Deletion in a Red-Black tree (cont'd)

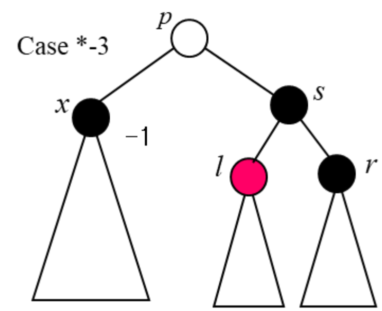


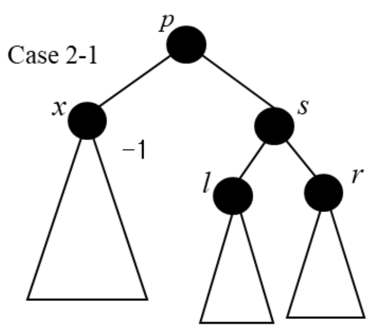
Red-Black Tree

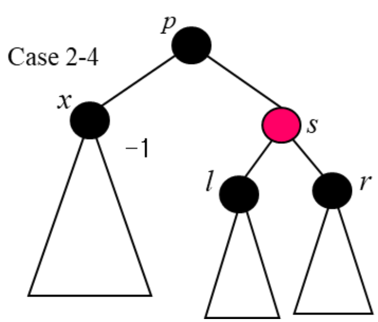
❖ Deletion in a Red-Black tree (cont'd)









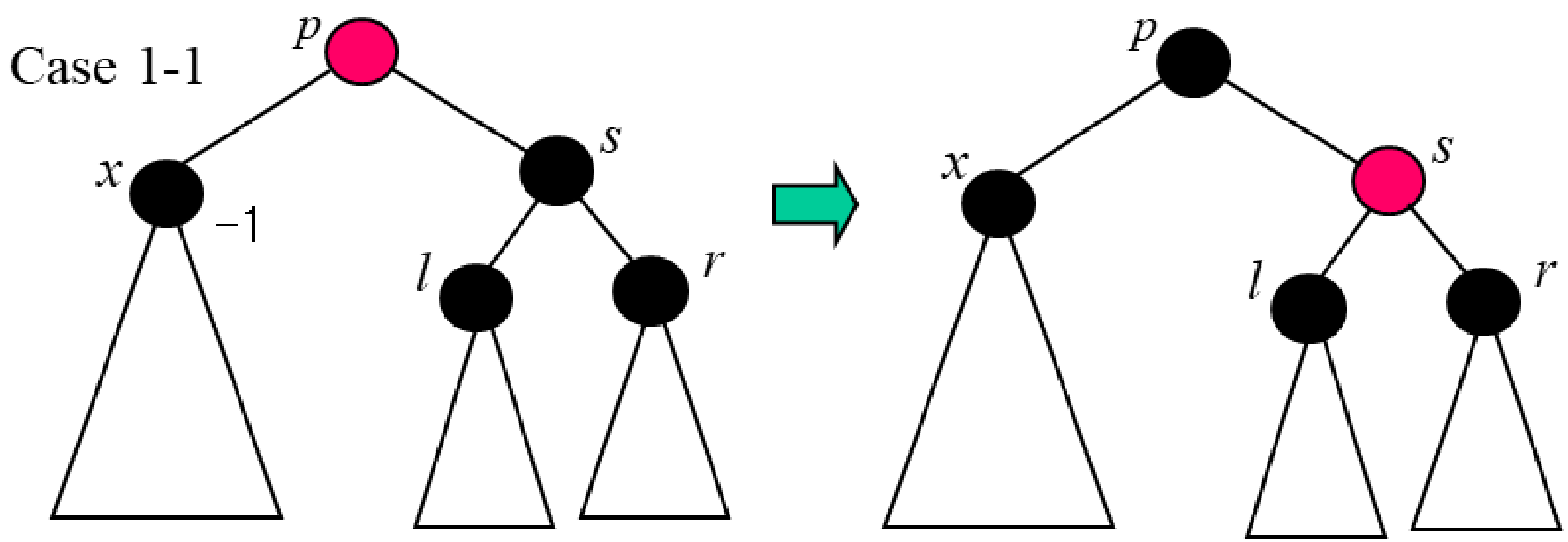


Finally divided into five cases

Red-Black Tree

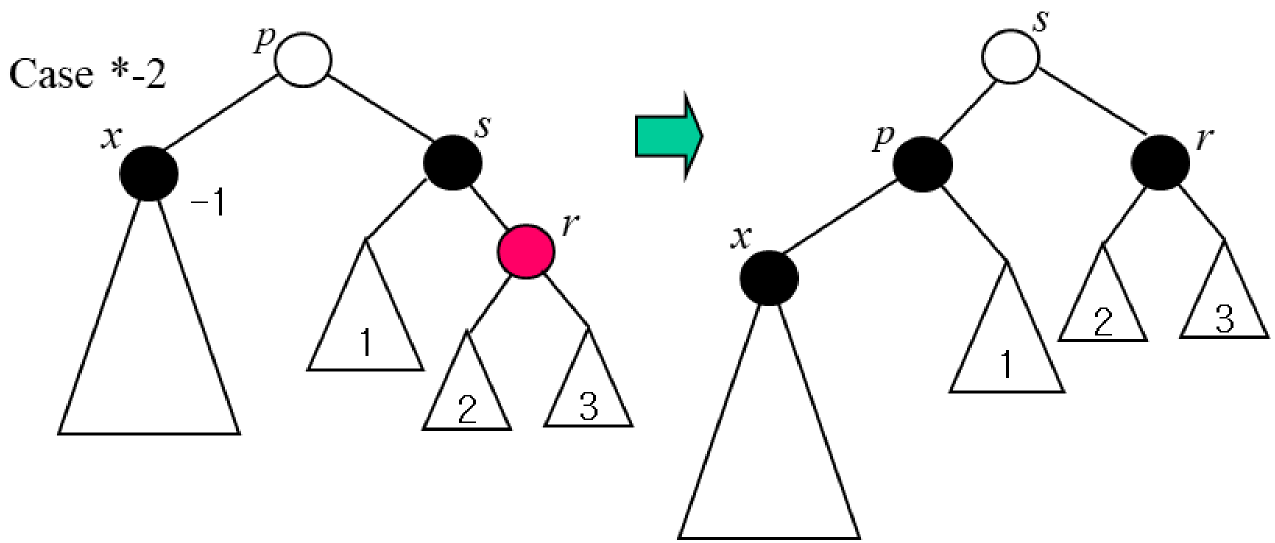
❖ Deletion in a Red-Black tree (cont'd)

- Case 1-1: swap the colors of nodes p and s
  - Number of Black nodes on the path to x increases
  - Number of Black nodes on the path through s remains unchanged



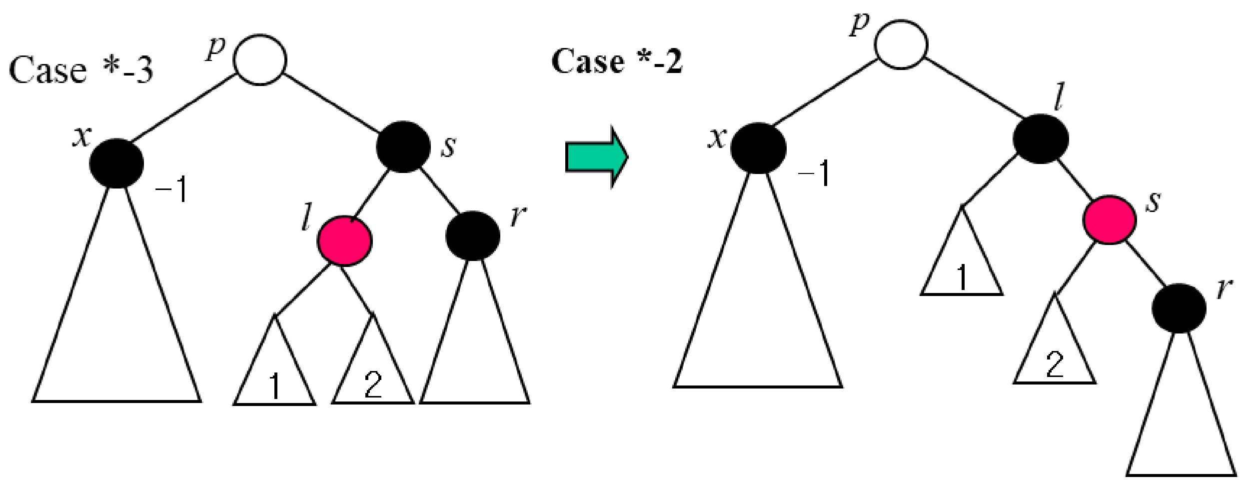
Red-Black Tree

- Deletion in a Red-Black tree (cont'd)
  - Case *-2:
    - ① Perform a left rotation around p
    - ② Swap the colors of p and s
    - ③ Change the color of r from Red to Black
    - Number of Black nodes on the path to x increases
    - Number of Black nodes on the path through s remains unchanged



Red-Black Tree

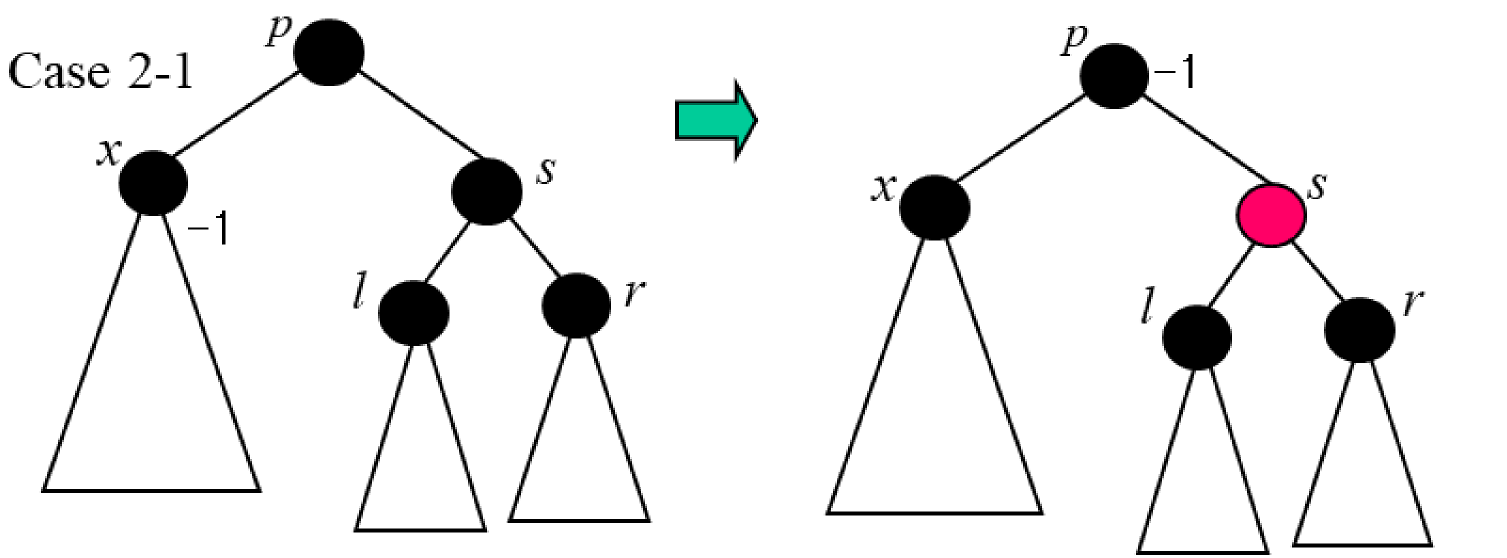
- Deletion in a Red-Black tree (cont'd)
  - Case *-3:
    - ① Perform a right rotation around s
    - ② Swap the colors of l and r
    - ③ Transition to Case *-2
    - Number of Black nodes on the path to x increases
    - Number of Black nodes on the path through s remains unchanged



Red-Black Tree

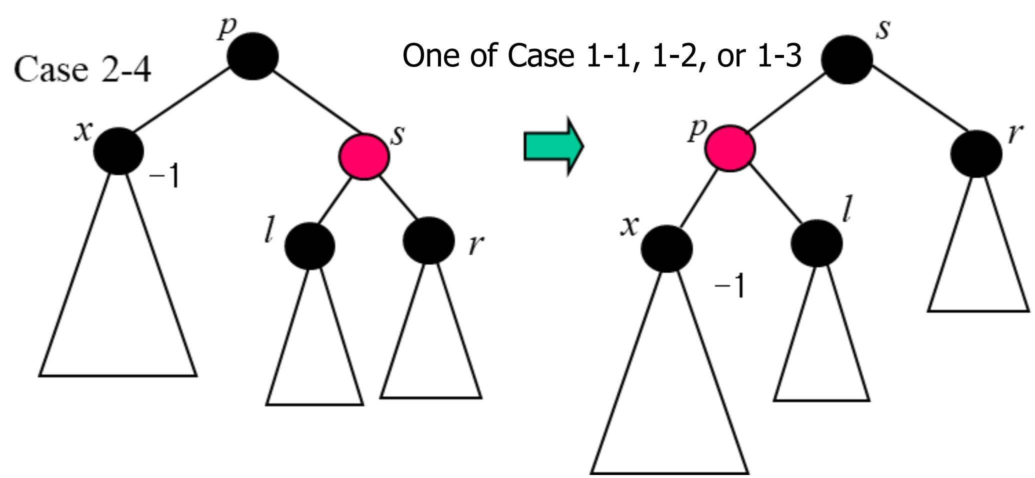
❖ Deletion in a Red-Black tree (cont'd)

- Case 2-1:
  ① Change the color of s from Black to Red
    – Path through s becomes deficient in Black nodes
    – Entire path through p becomes deficient in Black nodes
  ② Set p as the target node and solve recursively



Red-Black Tree

- Deletion in a Red-Black tree (cont'd)
  - Case 2-4:
    ① Perform a left rotation around p
    ② Swap the colors of p and s
      – Paths through l and r have no issues
      – However, the color of x's parent node changes from Black to Red
    ③ Consider the color combinations and transition to one of Case 1-1, 1-2, or 1-3



Red-Black Tree

- Analysis of the Red-Black tree
  - Possible maximum depth of a Red-Black tree with a total of n keys:
    - $\mathrm{O}(\log n)$
      - If the number of keys is n, then there are n internal nodes
      - The depth of the tree is $\lfloor \log n \rfloor + 1$
      - No matter how well constructed, the Black nodes on any path from the root node to an arbitrary leaf cannot exceed $\lfloor \log n \rfloor + 1$
      - Since Red nodes cannot exist consecutively, there are fewer Red nodes than Black nodes
      - The total length of any leaf cannot exceed $2(\lfloor \log n \rfloor + 1)$

Part 2

B-TREE

B-Tree

- Background
  - Utilize resources used for disk access
    - Accessing a disk takes significantly more time compared to accessing memory
      - Accessing data on a disk requires block-level access for reading and writing
      - Even if you want to write just one byte, the entire block must be read first
      - Blocks are usually 8KB or 16KB
      - At the software level, a block is referred to as a page
      - Reading one block requires 200,000 clock cycles (based on a 2GHz CPU)
        - Processing a machine instruction takes less than 2 clock cycles

B-Tree

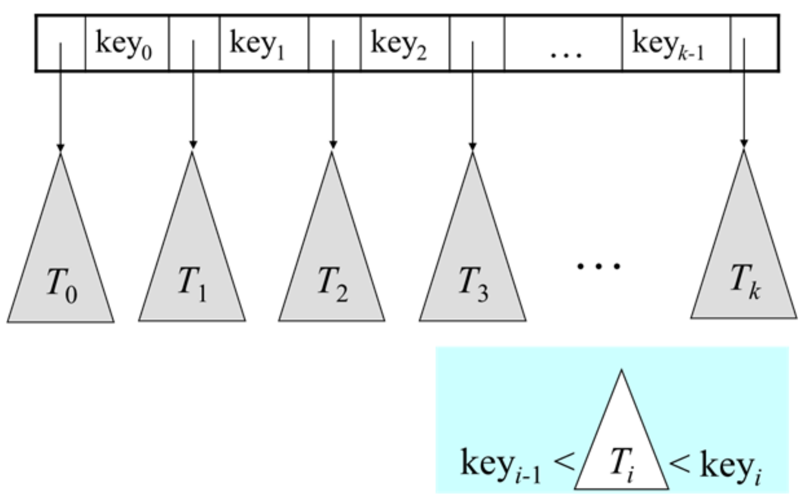
- Background
  - What if the search tree is too large?
    - Search tree cannot be loaded into memory for use
      - The search tree is placed on the disk for processing
      - Number of disk accesses has a greater impact on performance than CPU clock cycles
      - Increasing the branching factor of the search tree can reduce its expected depth
        - What if handling around one billion keys?
        - In the case of a binary tree, the depth is 30
        - If there are 256 branches, the depth is 5
        - If data access requests are frequent, the number of disk accsses is reduced to 1/6
        - A branch refers to the number of children a parent can have

B-Tree

- Background
  - External search tree
    - Search tree on the disk
  - B-tree
    - A search tree with more than two branches
    - Suitable for use in a disk environment
    - Maintains balance to reduce the number of disk accesses

B-Tree

- B-tree
  - Properties
    - All nodes except the root have $\lfloor k/2 \rfloor \sim k-1$ keys
    - All leaf nodes have the same depth
  - Maximize the number of branches while ensuring each node has at least half of its maximum allowable keys



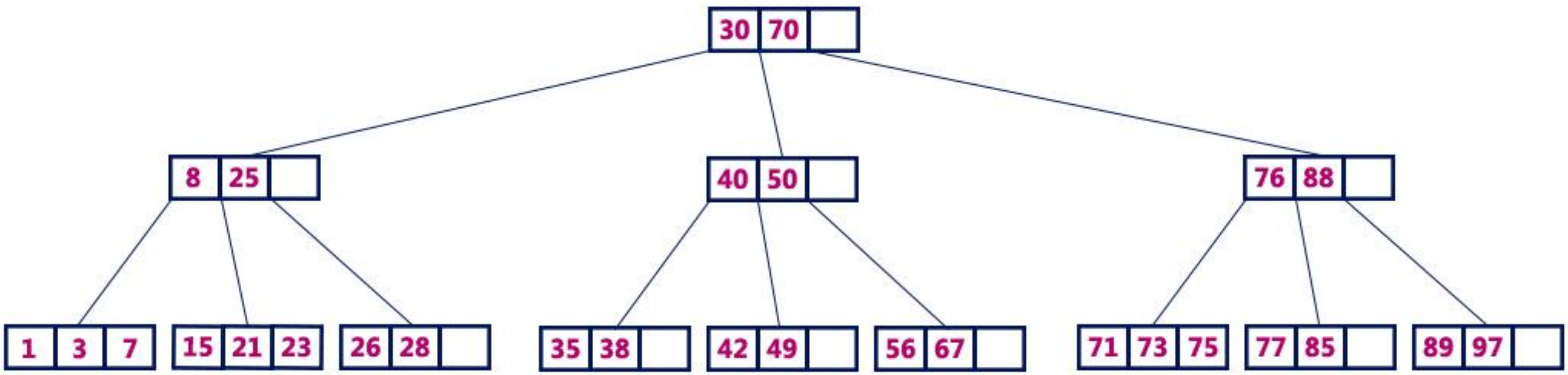
B-Tree

- B-Tree of Order k has the following properties
  ① All the leaf nodes must be at the same level
  ② All nodes except root must have at least $\lfloor k/2 \rfloor$ keys and maximum of $k-1$ keys
  ③ All internal nodes except root must have at least $\lfloor k/2 \rfloor$ children
  ④ If the root node is not leaf node, then it must have at least 2 children
  ⑤ An internal node with $n-1$ keys must have n number of children
  ⑥ All keys within a node must be in ascending (descending) order

B-Tree

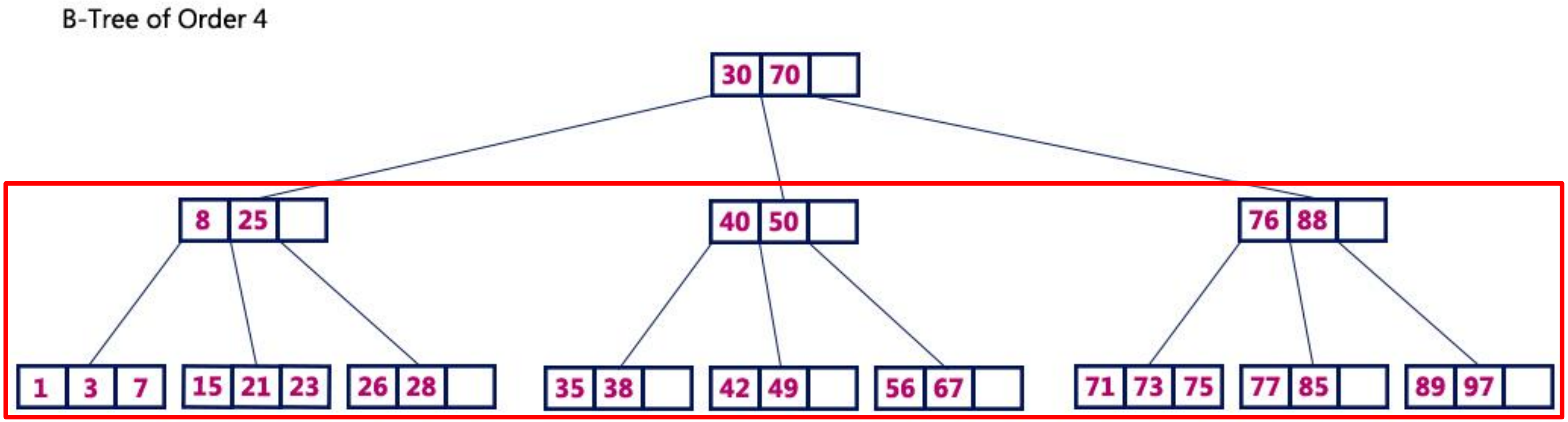
- B-Tree of Order k has the following properties
  - Property ①: all the leaf nodes must be at the same level

B-Tree of Order 4



B-Tree

- B-Tree of Order k has the following properties
  - Property ②: all nodes except root must have at least $\lceil k/2 \rceil$ keys and maximum of $k-1$ keys

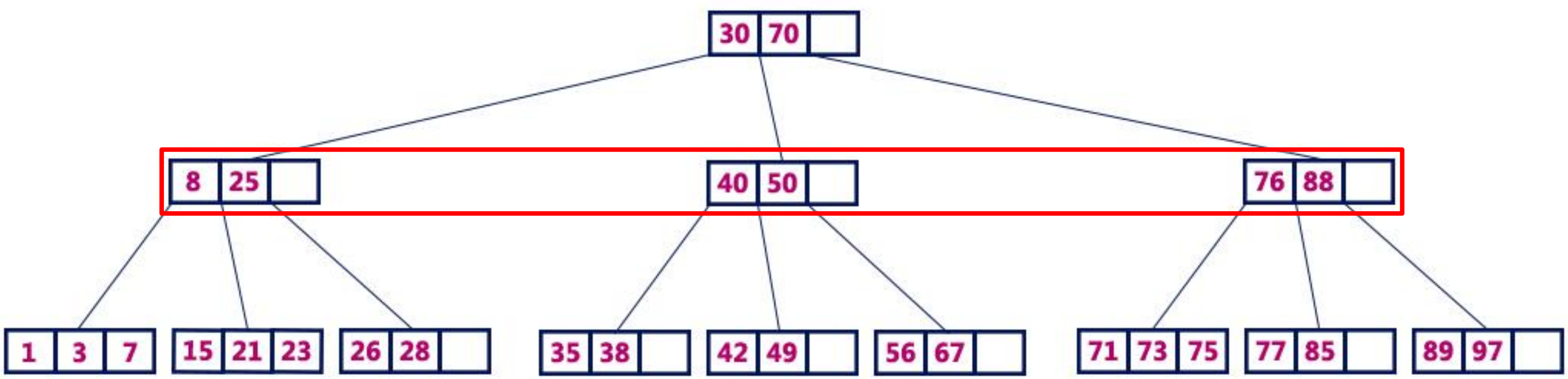


B-Tree

❖ B-Tree of Order k has the following properties

- Property ③: all internal nodes except root must have at least $\lceil k/2 \rceil$ children

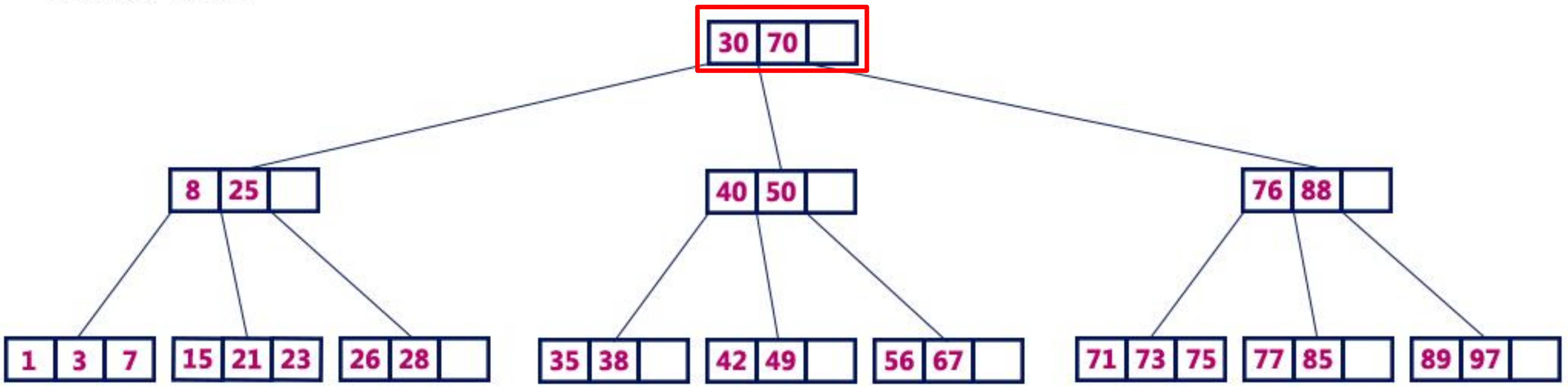
B-Tree of Order 4



B-Tree

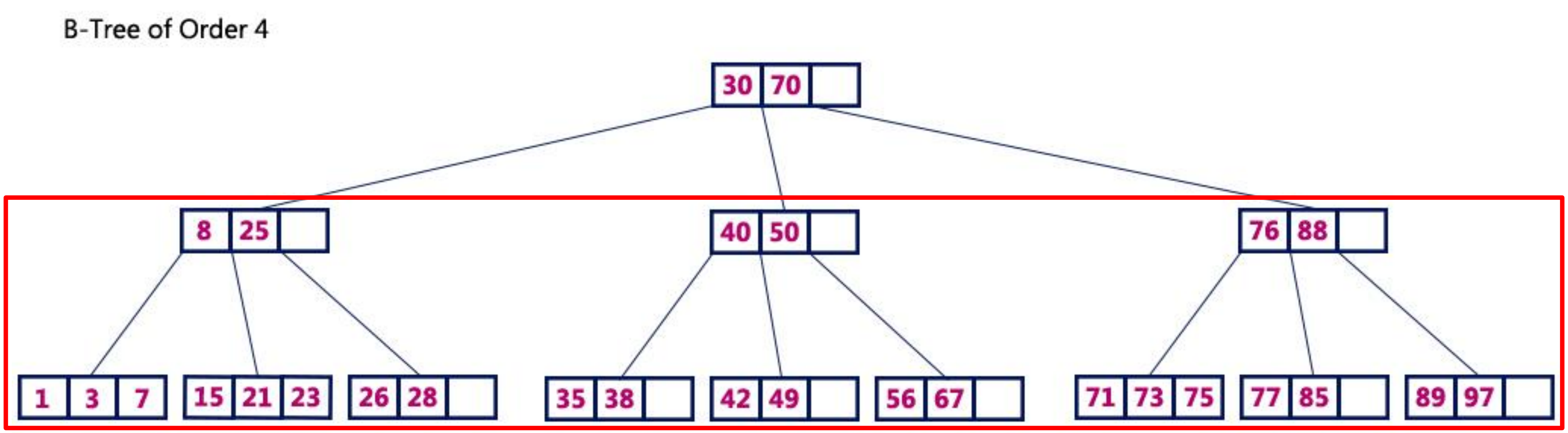
- B-Tree of Order k has the following properties
  - Property ④: if the root node is not leaf, then it must have at least 2 children

B-Tree of Order 4



B-Tree

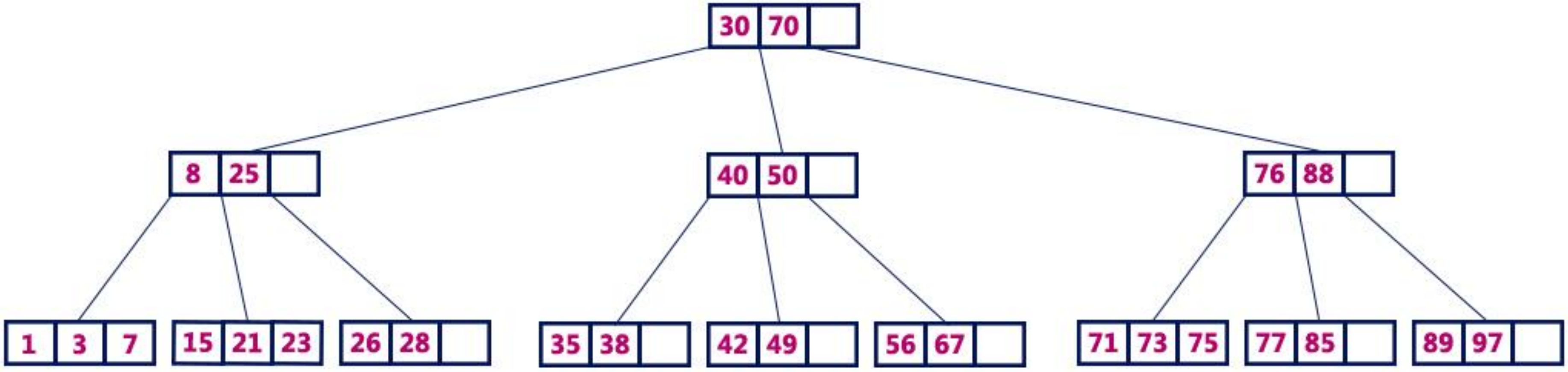
- ❖ B-Tree of Order k has the following properties
  - Property ⑤: an internal node with $n-1$ keys must have n number of children



B-Tree

- B-Tree of Order k has the following properties
  - Property ⑥: all keys within a node must be in ascending (descending) order

B-Tree of Order 4



B-Tree

- In a B-tree, new element must be added only at leaf node
  1. If tree is **empty**, then create a new node with new key and insert into the tree as a root node
  2. If tree is **not empty**, then find a leaf node to which the new key can be added using binary search tree logic
  3. If the leaf node has an empty space, then add the new key to the leaf node by maintaining **order of keys** within the node
  4. If the leaf node is already full, then split the node by sending middle value to its parent node. Repeat that same until sending value is fixed into a node
  5. If the splitting is occurring to the root node, then the middle value becomes **new root node** for the tree and the height of the tree is increased by one

B-Tree

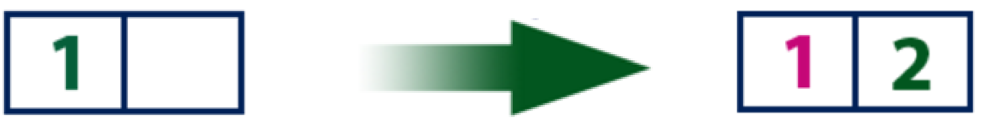
❖ Example

- Construct a B-tree of Order 3 (k=3) by inserting numbers from 1 to 10

- Insert '1'
  - Rule ①:
    - If tree is empty, then create a new node with new key and insert into the tree as a root node

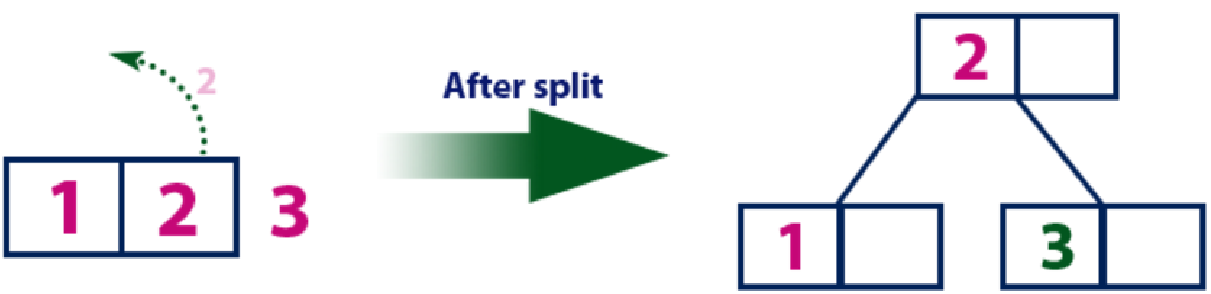
B-Tree

- Example
  - Insert '2'
    - Rule ②:
      - If tree is not empty, then find a leaf node to which the new key can be added using binary search tree logic
    - Rule ③:
      - If the leaf node has an empty space, then add the new key to the leaf node by maintaining order of keys within the node



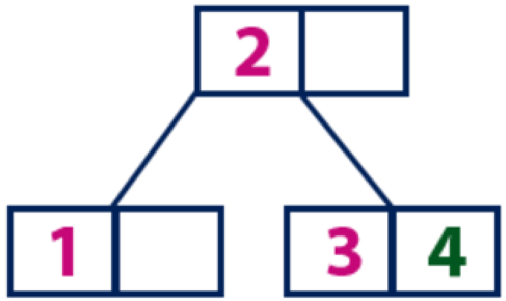
B-Tree

- ❖ Example
  - ▪ Insert '3'
    - Rule ④:
      - – If the leaf node is already full, then split the node by sending middle value to its parent node
    - Rule ⑤:
      - – If the splitting is occurring to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one

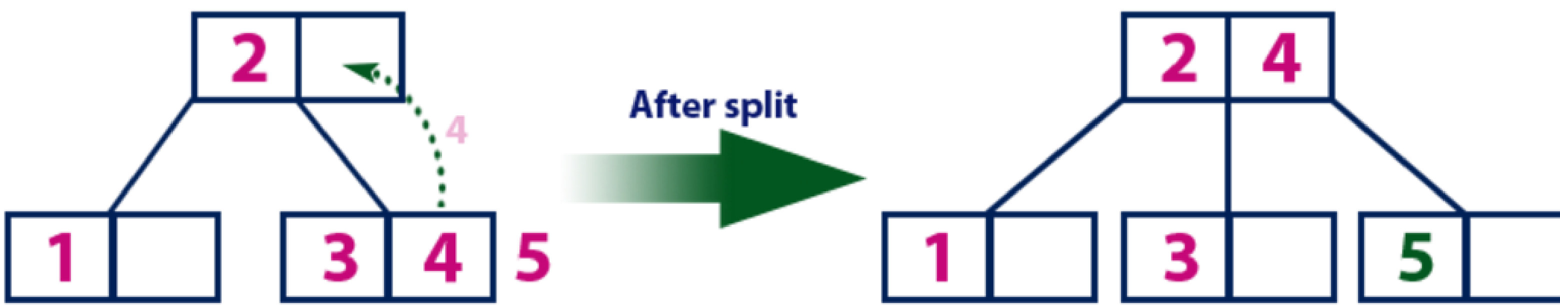


B-Tree

- Example
  - Insert '4'

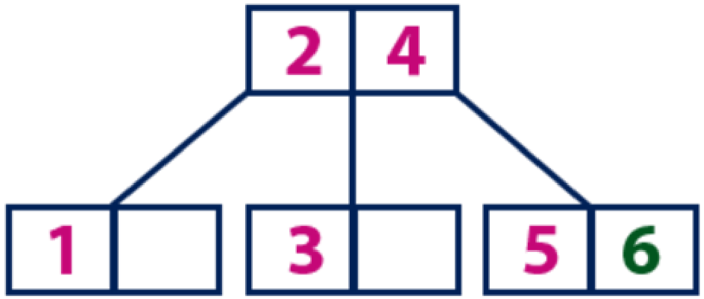


  - Insert '5'

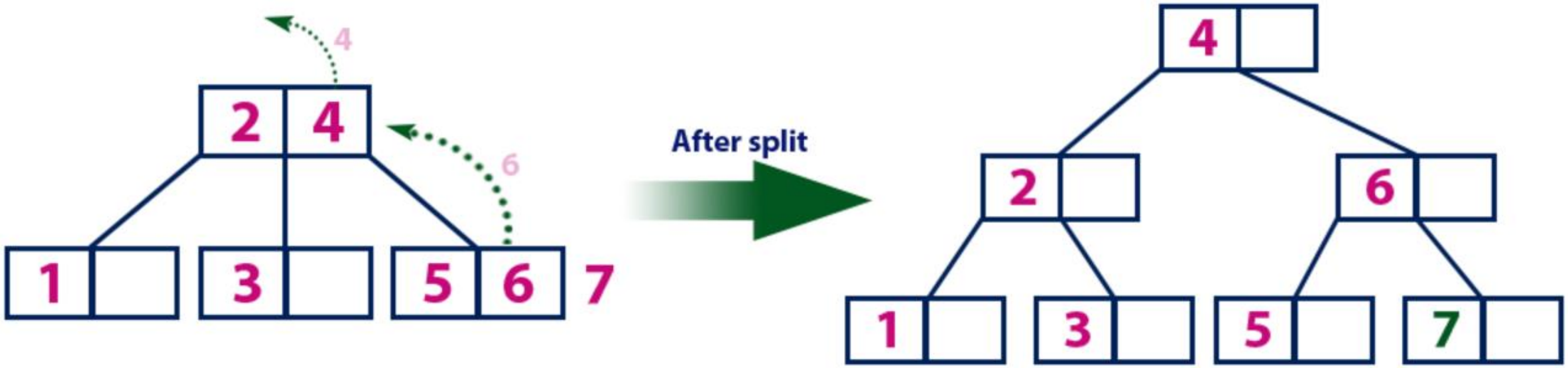


B-Tree

- Example
  - Insert ‘6’

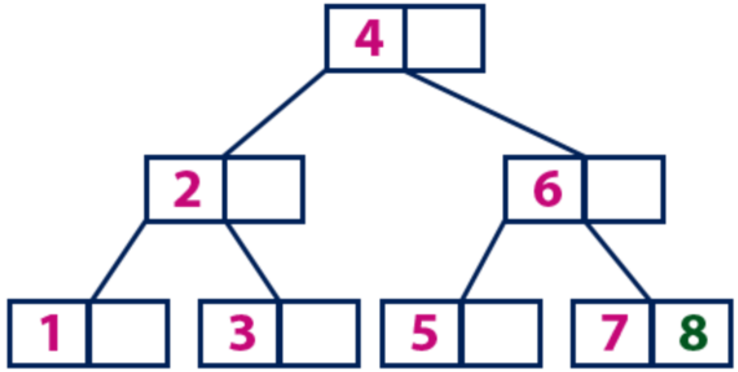


  - Insert ‘7’

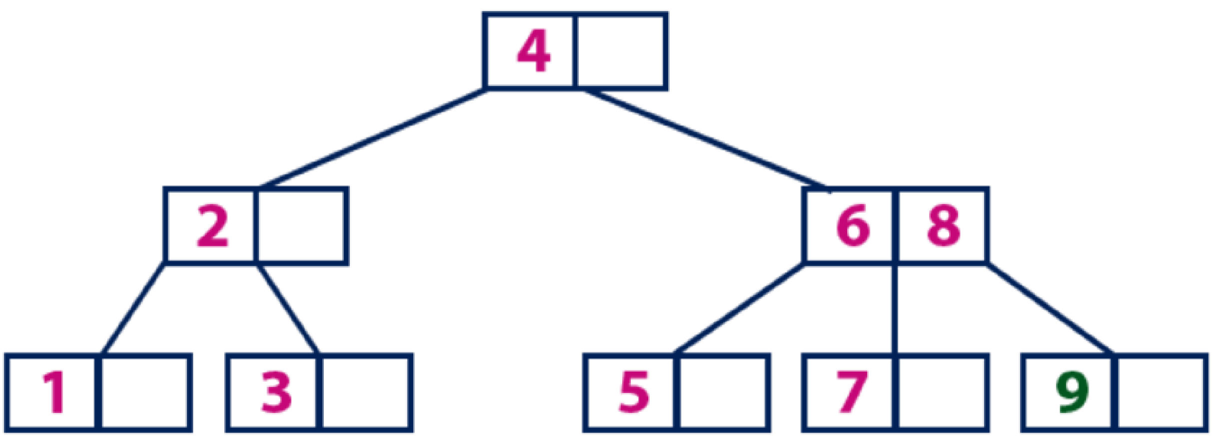


B-Tree

- Example
  - Insert '8'

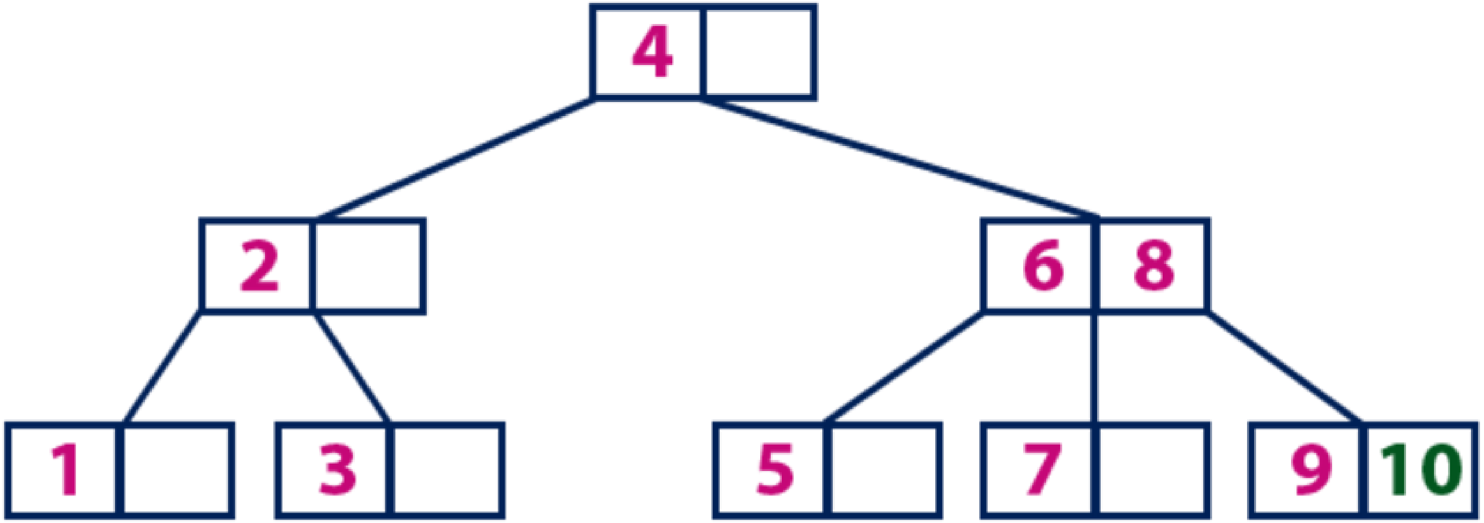


  - Insert '9'



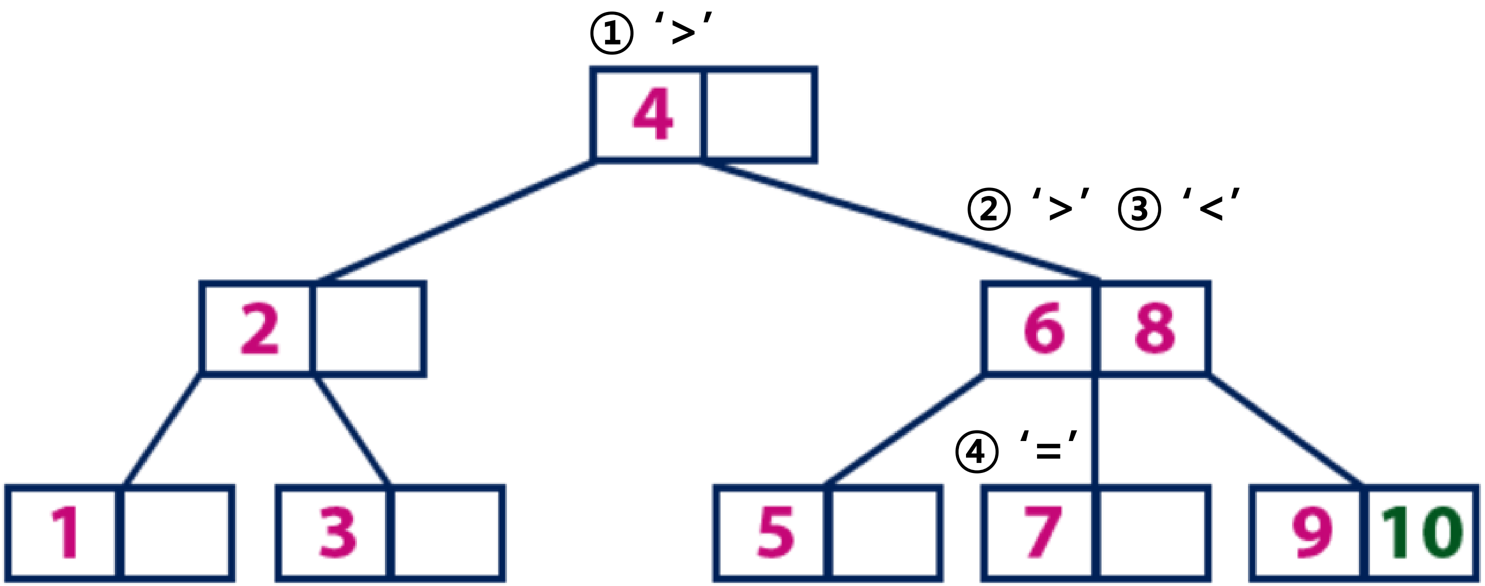
B-Tree

- Example
  - Insert ‘10’



B-Tree

- How to search a key using B-tree
  - B-tree searches for the key top-down from the root node to the leaf
  - Search '7'



B-Tree

❖ Deletion in a B-tree

- Process:

  ① Find the node with key x

  ② If it is not a leaf node, find the leaf node r with key y, where y is the element immediately following x

  ③ Delete x from the leaf node

  ④ Resolve any underflow in the node after deleting x

B-Tree

- Deletion in a B-tree
  - Pseudo-code

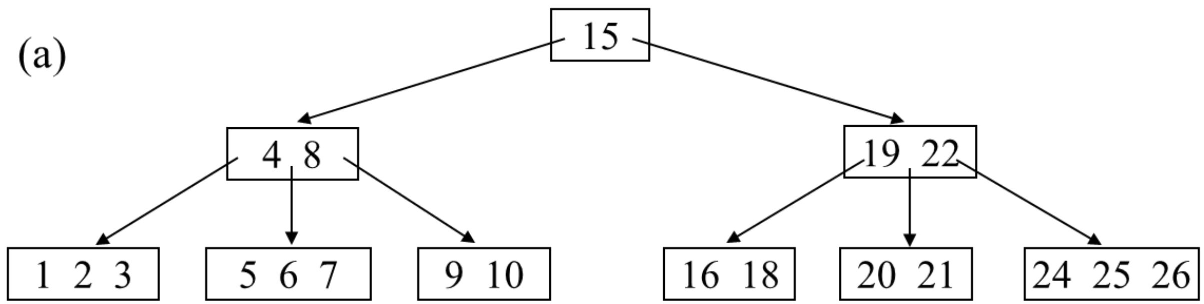
```
Deletion(t, x, v){
   if (v is not leaf) then{
      find a leaf node with a key y
      swap x and y
   }
   delete x in the leaf node r
   if (underflow occurs in r) then{
      clearUnderflow(r)
   }
}
```

```
clearUnderflow(r){
   if (r's sibling nodes has extra keys) then
      send the extra keys to r
   else {
      merge r with its sibling nodes
      if (underflow occurs in the paranet p)
      then
         clearUnderflow(p)
   }
}
```

B-Tree

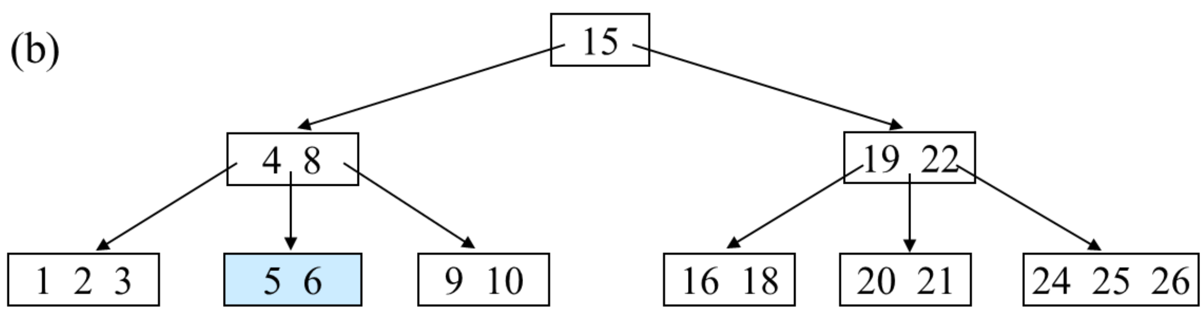
- Deletion in a B-tree
  - Example ($k = 6$)

(a)



Delete '7'

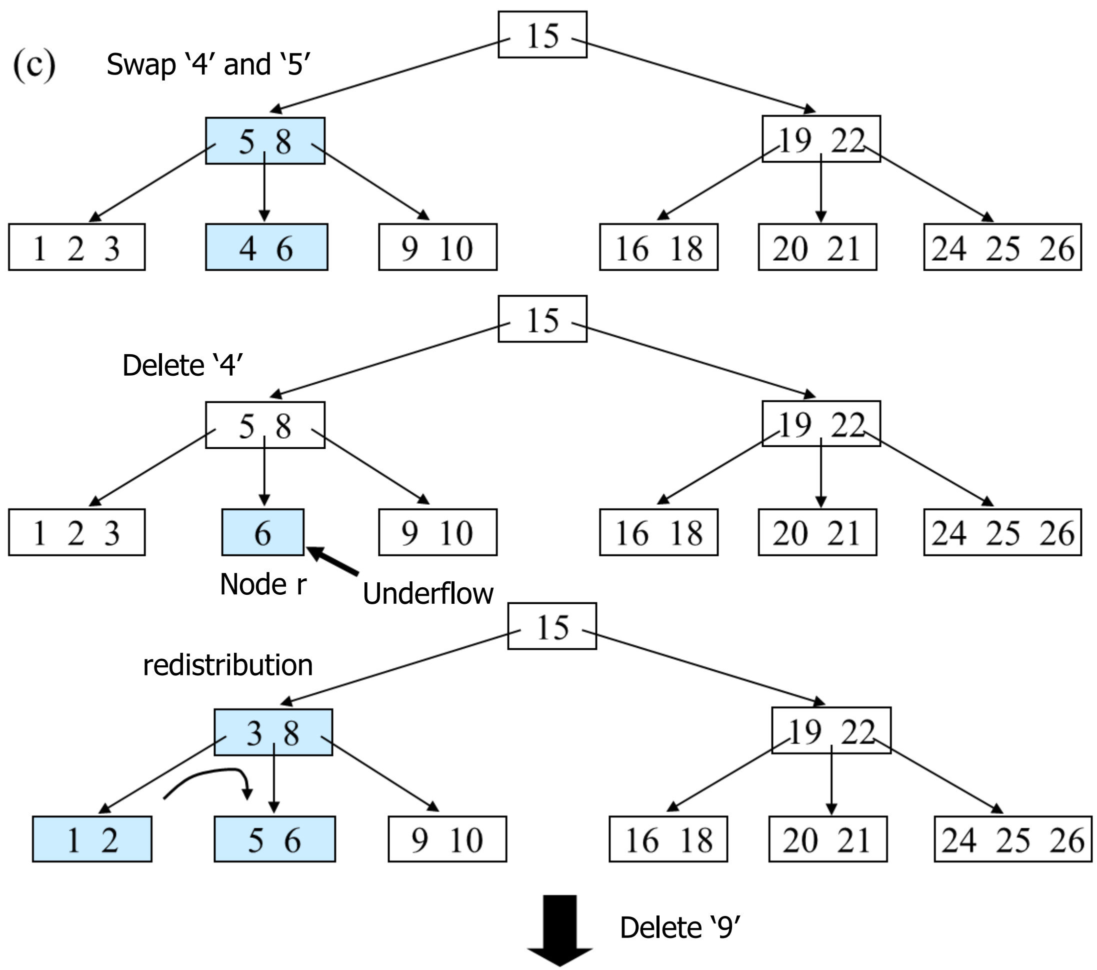
(b)



Delete '4'

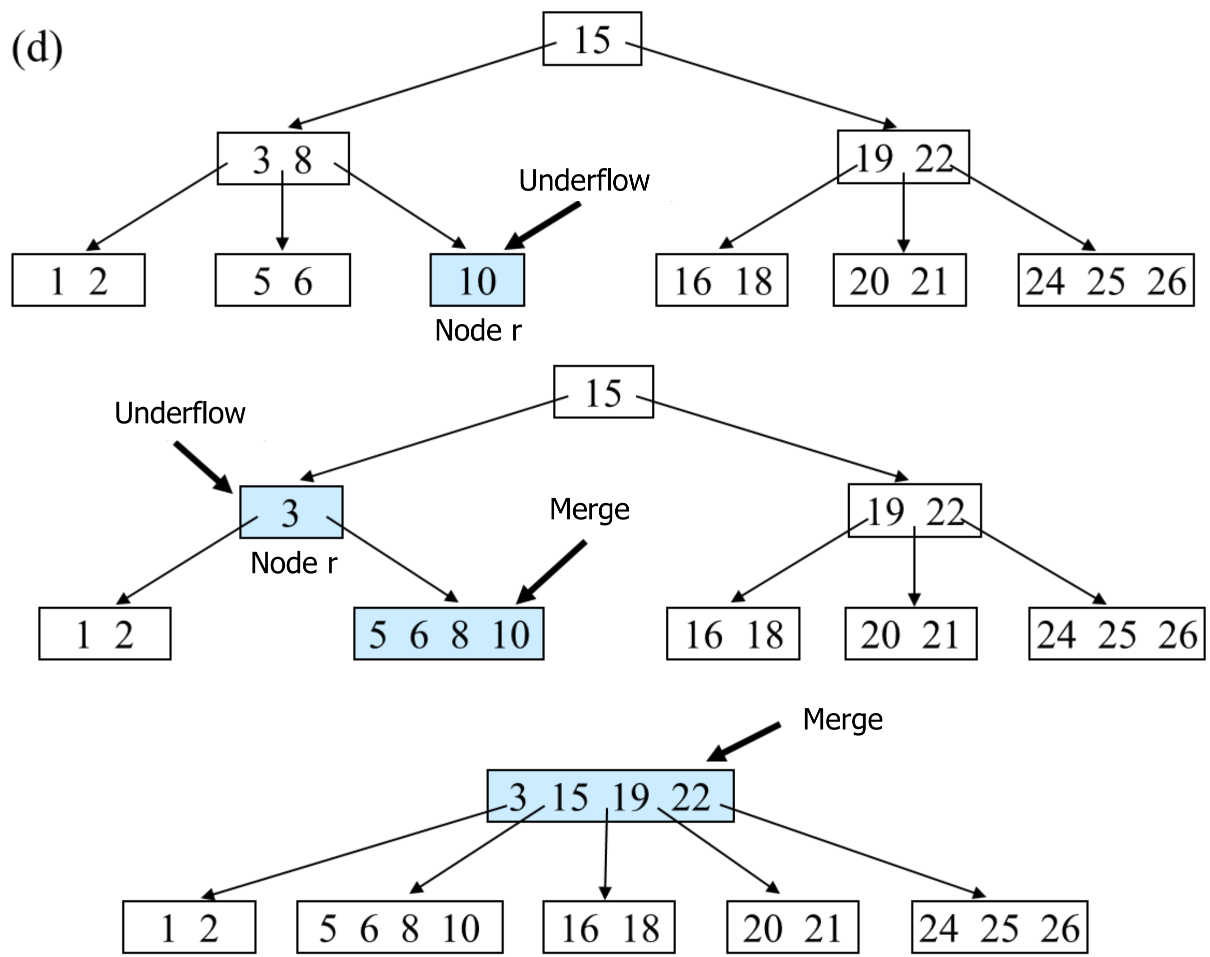
B-Tree

- Deletion in a B-tree
  - Example ($k = 5$)



B-Tree

- Deletion in a B-tree
  - Example ($k = 5$)



B-Tree

- Analysis of the B-tree
  - If a binary search tree is well balanced, its height approaches $\log_2 n$
  - If a d-branches tree is well balanced, its height approaches $\log_d n$
  - In a B-tree, if $k = \mathrm{d}$, every internal node except the root must have at least $\left\lfloor \frac{d}{2} \right\rfloor$ children
    - In worst case, the height of B-tree is $\log_{\left\lfloor \frac{d}{2} \right\rfloor} n$

B-Tree

- Analysis of the B-tree (cont'd)
  - Searching time: $O(\log n)$
  - Insertion time: $O(\log n)$
  - Deletion time: $O(\log n)$

Part 3

MULTIDIMENSIONAL SEARCH TREE

Multidimensional Search Tree

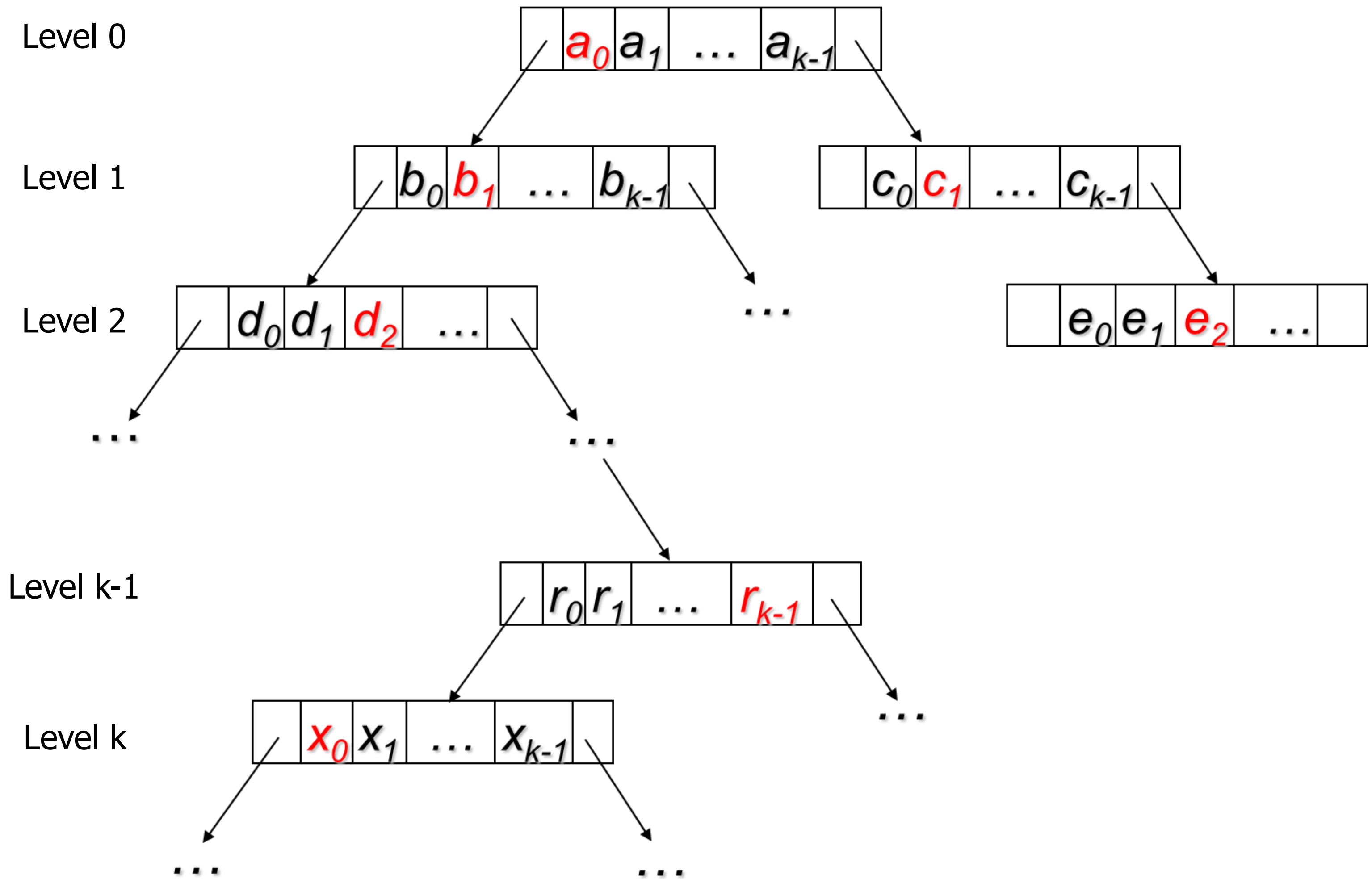
- What is Multidimensional Search Tree?
  - Single dimensional search tree
    - Record key consists of a single field
  - Multidimensional search tree
    - Record key consists of multiple fields
    - Indexing for spatial data
  - Representative trees
    - KD-tree
    - KDB-tree
    - R-tree
    - Grid file

Multidimensional Search Tree

- KD-tree
  - An extension of the binary search tree
  - Fields composing the keys are used alternately for searching at each level
    - Root node branches using only the first field
    - The next level branches using only the second field
    - (…)
    - The i-th level branches using only the i mod k field

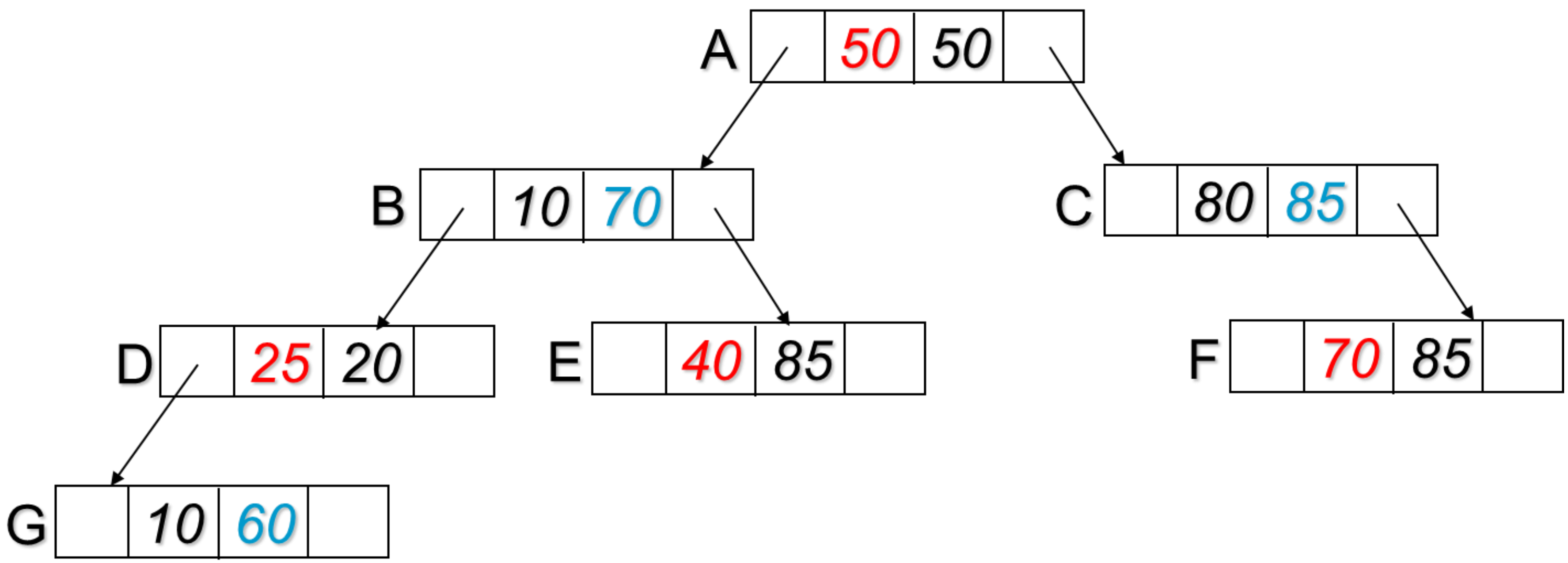
Multidimensional Search Tree

❖ Example of KD-tree



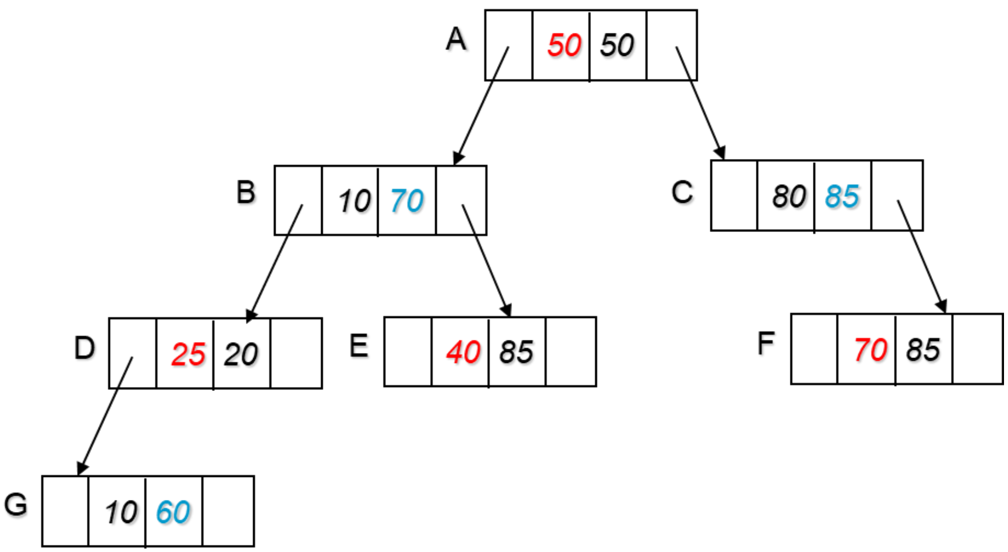
Multidimensional Search Tree

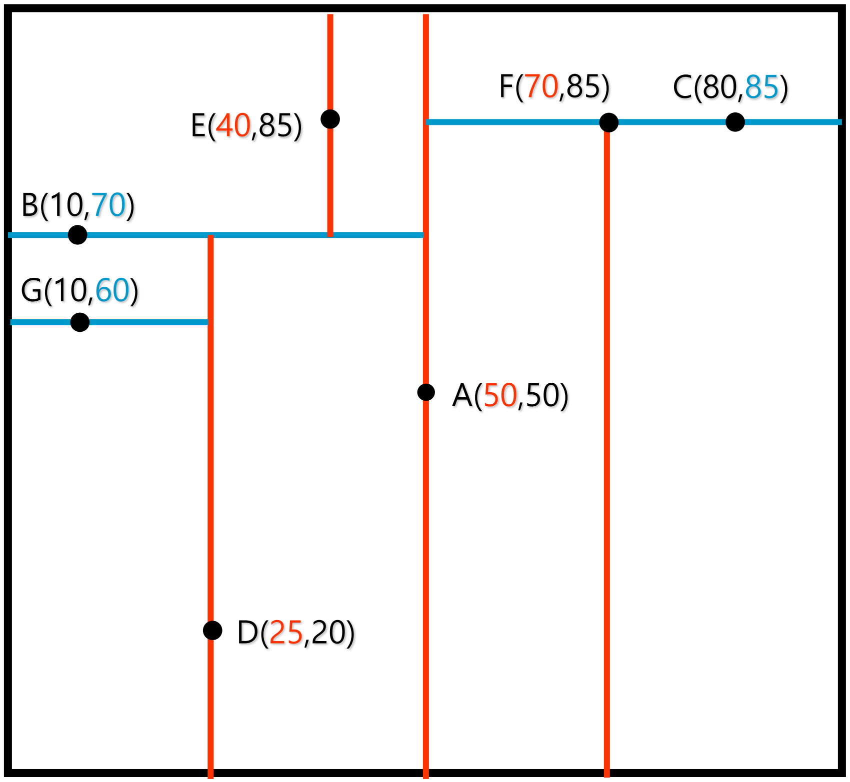
- Example of KD-tree
  - Two-dimensional KD-tree represented by (x, y)
    - Input sequence: A(50, 50) → B(10, 70) → C(80, 85) → D(25, 20) → E(40, 85) → F(70, 85) → G(10, 60)



Multidimensional Search Tree

- KD-tree and multidimensional space
  - KD-tree is a process of partitioning space
  - KD-tree performs data searches by gradually narrowing down the space





Multidimensional Search Tree

- ❖ Searching in a KD-tree
  ① Input an arbitrary key
  ② Use each field in sequence for the search

- ❖ Insertion in a KD-tree
  ① Input the new key to insert
  ② Use each field in sequence for the search
  ③ Traverse the tree, and upon reaching a leaf node, insert on the left or right

Searching and inserting are simple extensions of a binary search tree

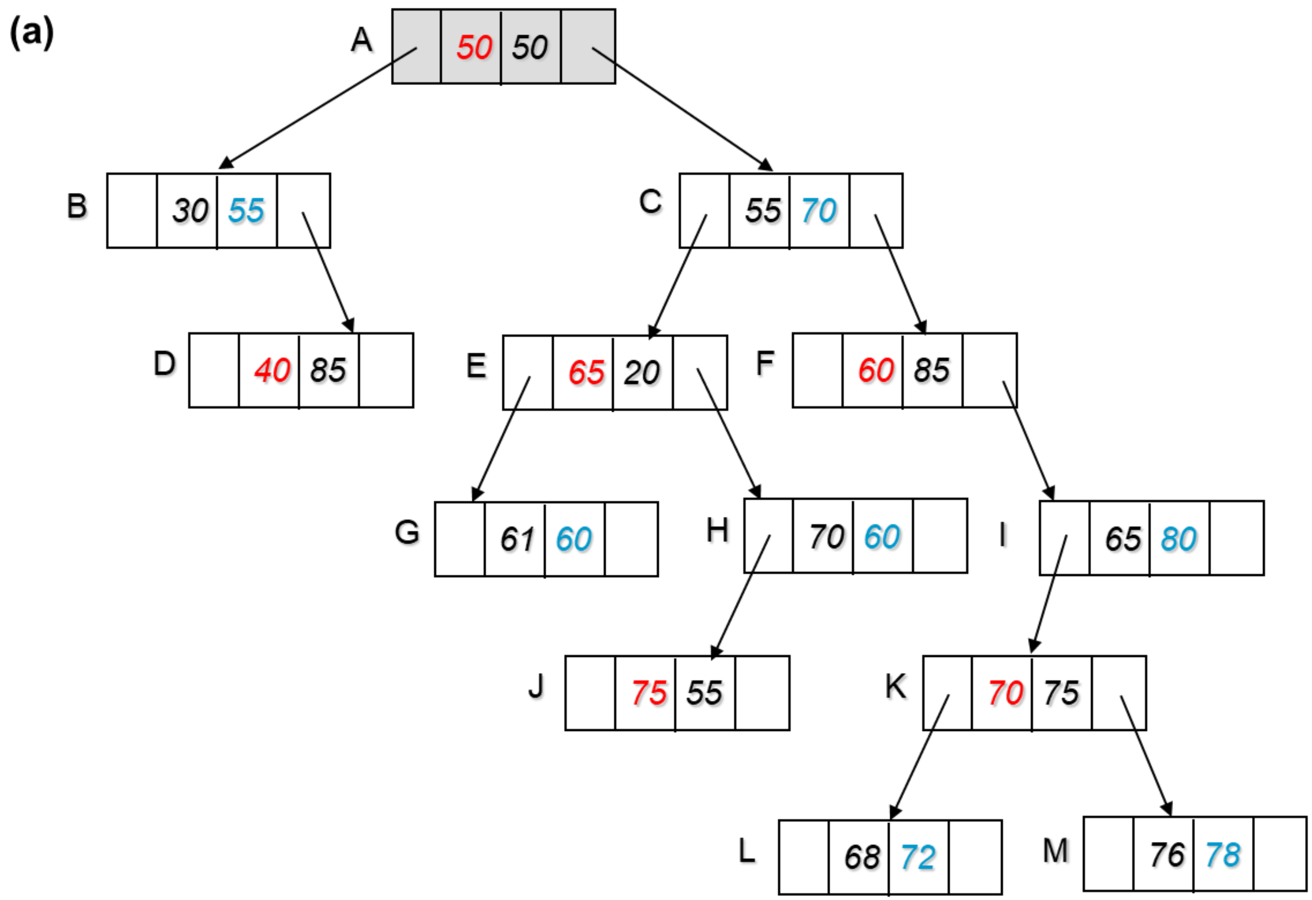
Multidimensional Search Tree

- Deletion in a KD-tree
  - Binary search tree
    - Handle by categorizing into three cases based on the number of children
  - KD-tree
    - Case 1: no children
      - Remove only the target node r
    - Case 2: one children
      - Directly connect the parent of the target node r to r’s child node
      - However, the properties of the KD-tree are not preserved
      - Therefore, delete it using the same method as when there are two children
    - Case 3: two children
      - Move the node with the smallest branching field value in the right subtree to the position of node r
      - If there is only a left child, move the node with the largest branching field value in the left subtree to the position of node r

Multidimensional Search Tree

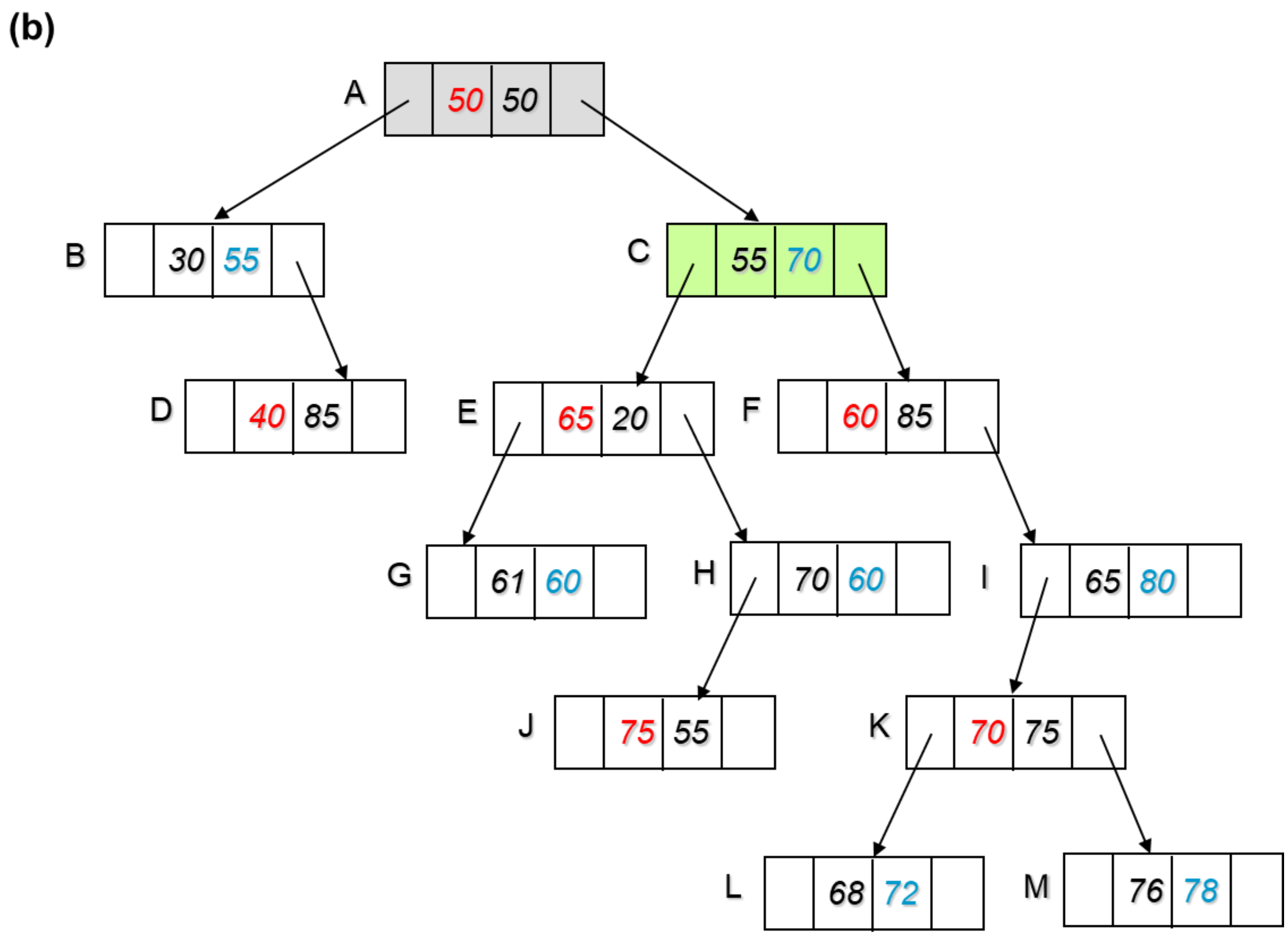
❖ Deletion in a KD-tree

- Example: delete node A



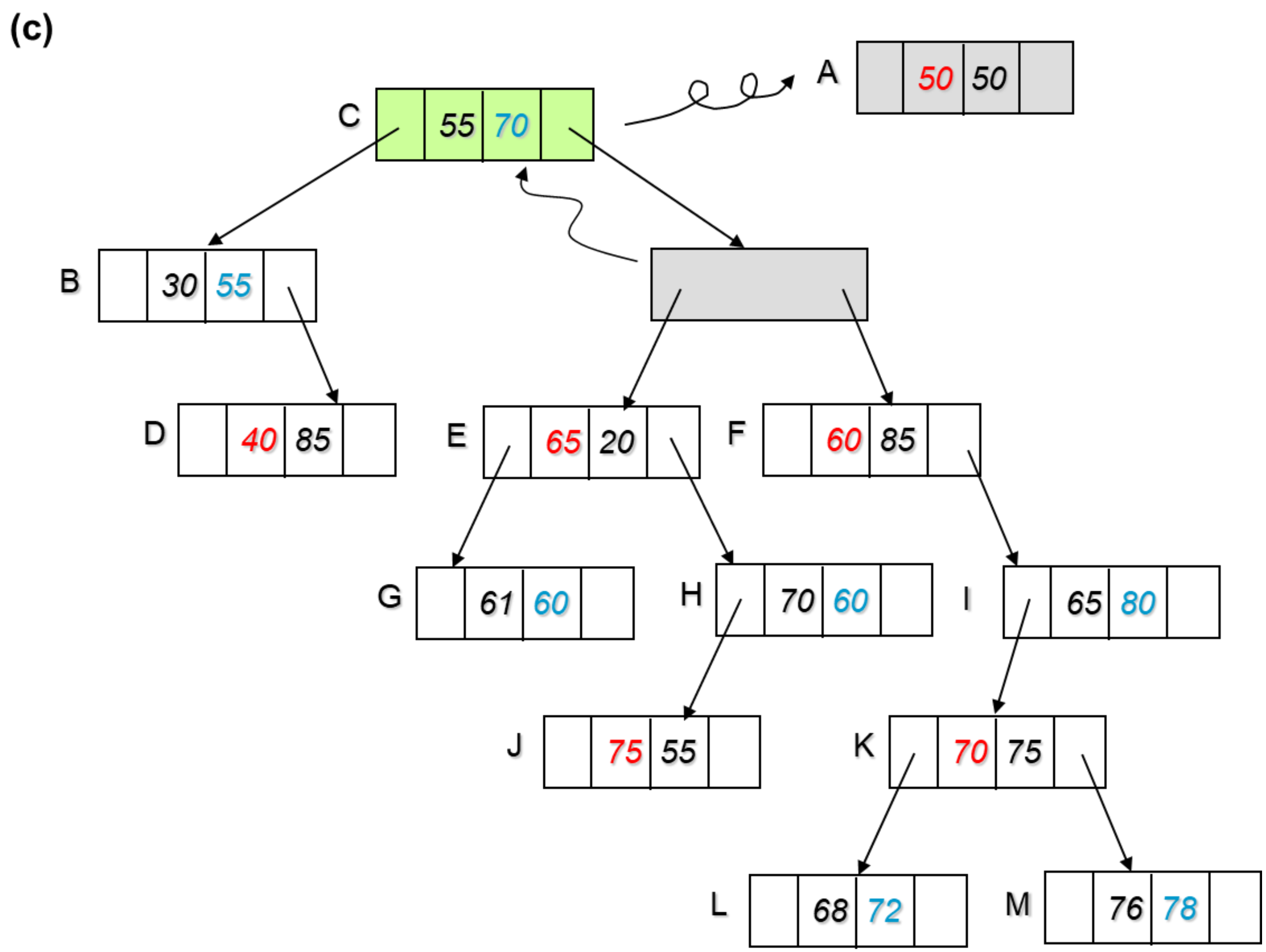
Multidimensional Search Tree

- Deletion in a KD-tree
  - Example: delete node A



Multidimensional Search Tree

- Deletion in a KD-tree
  - Example: delete node A

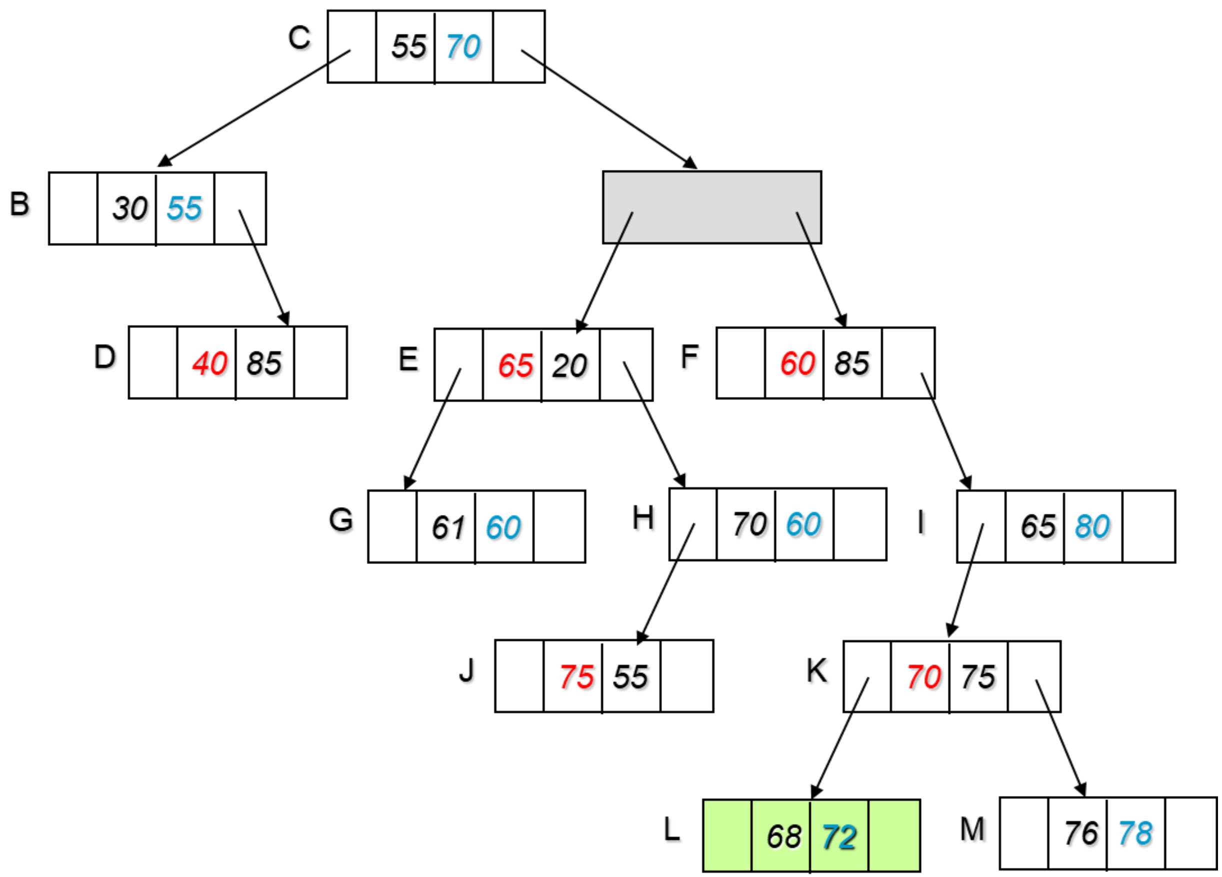


Multidimensional Search Tree

❖ Deletion in a KD-tree

- Example: delete node A

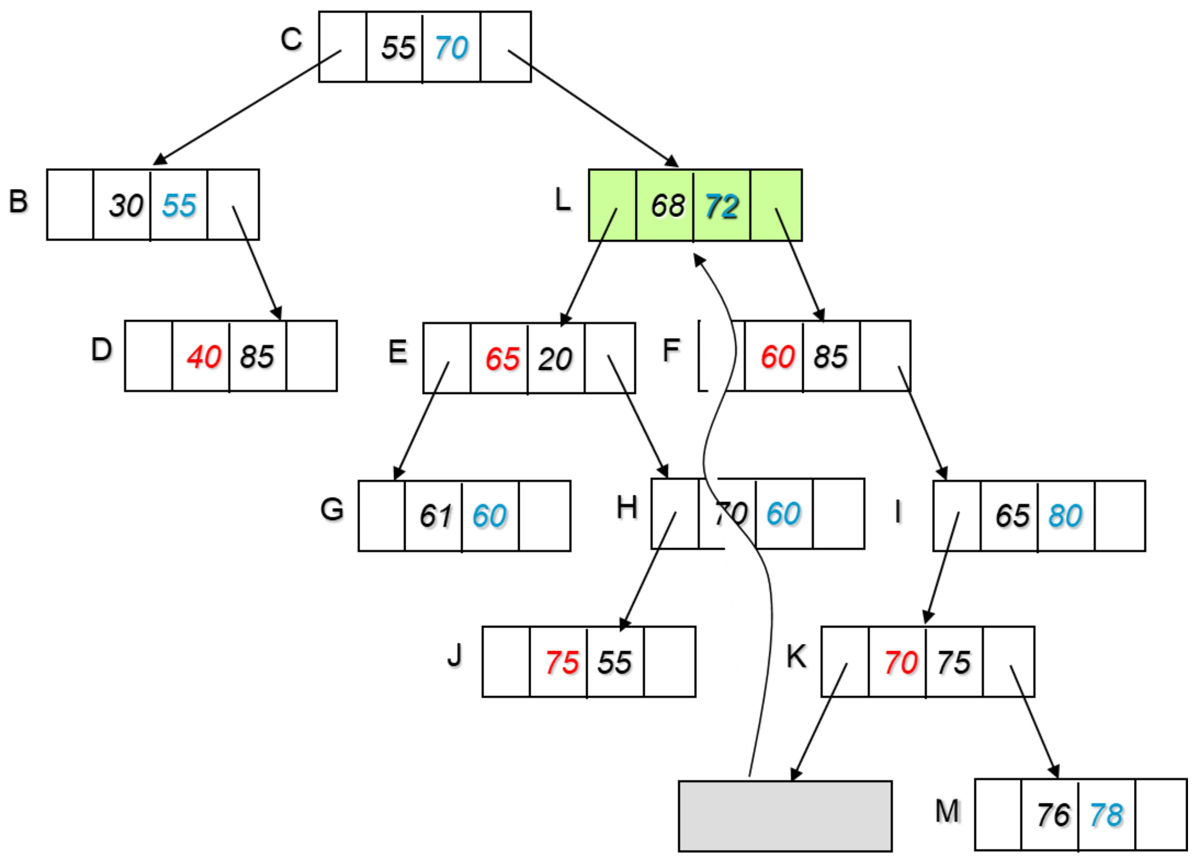
(d)



Multidimensional Search Tree

- Deletion in a KD-tree
  - Example: delete node A

(e)

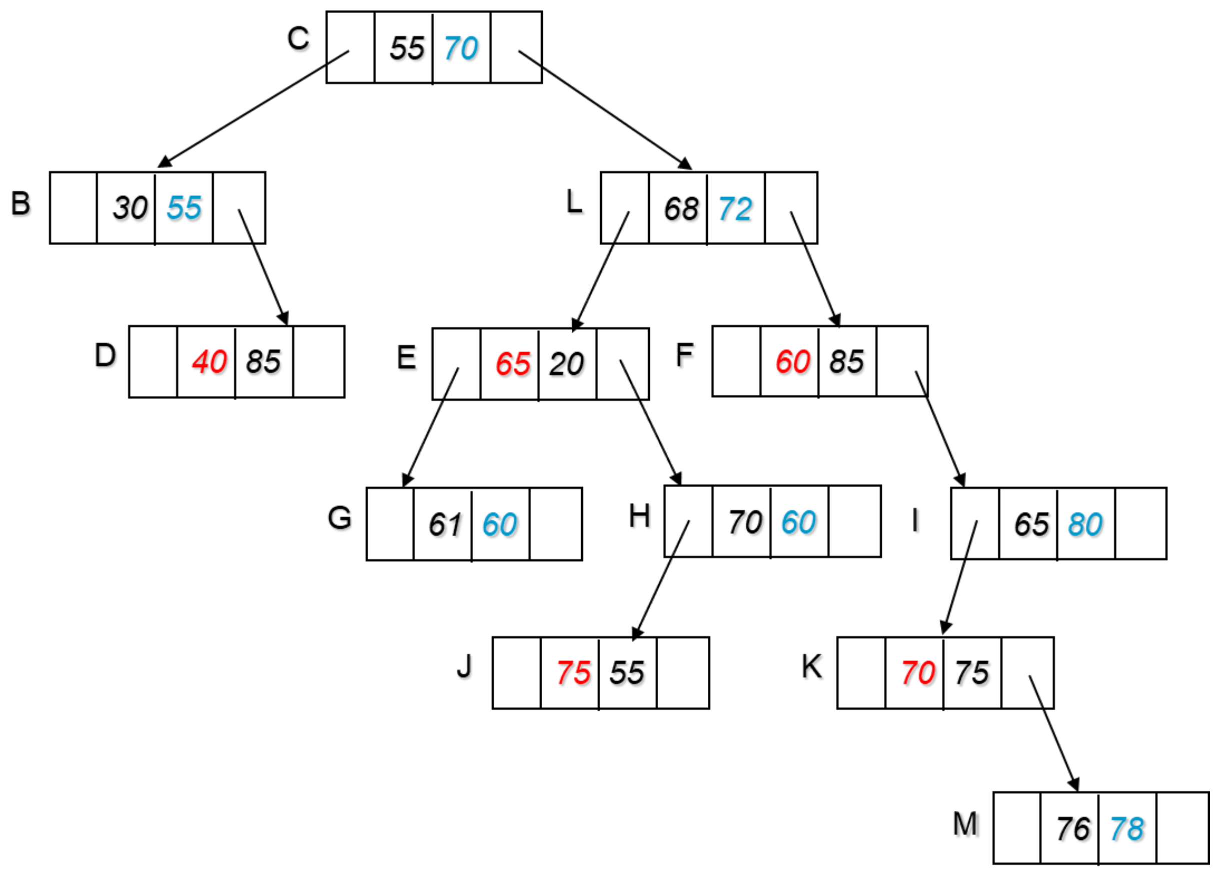


Multidimensional Search Tree

❖ Deletion in a KD-tree

- Example: delete node A

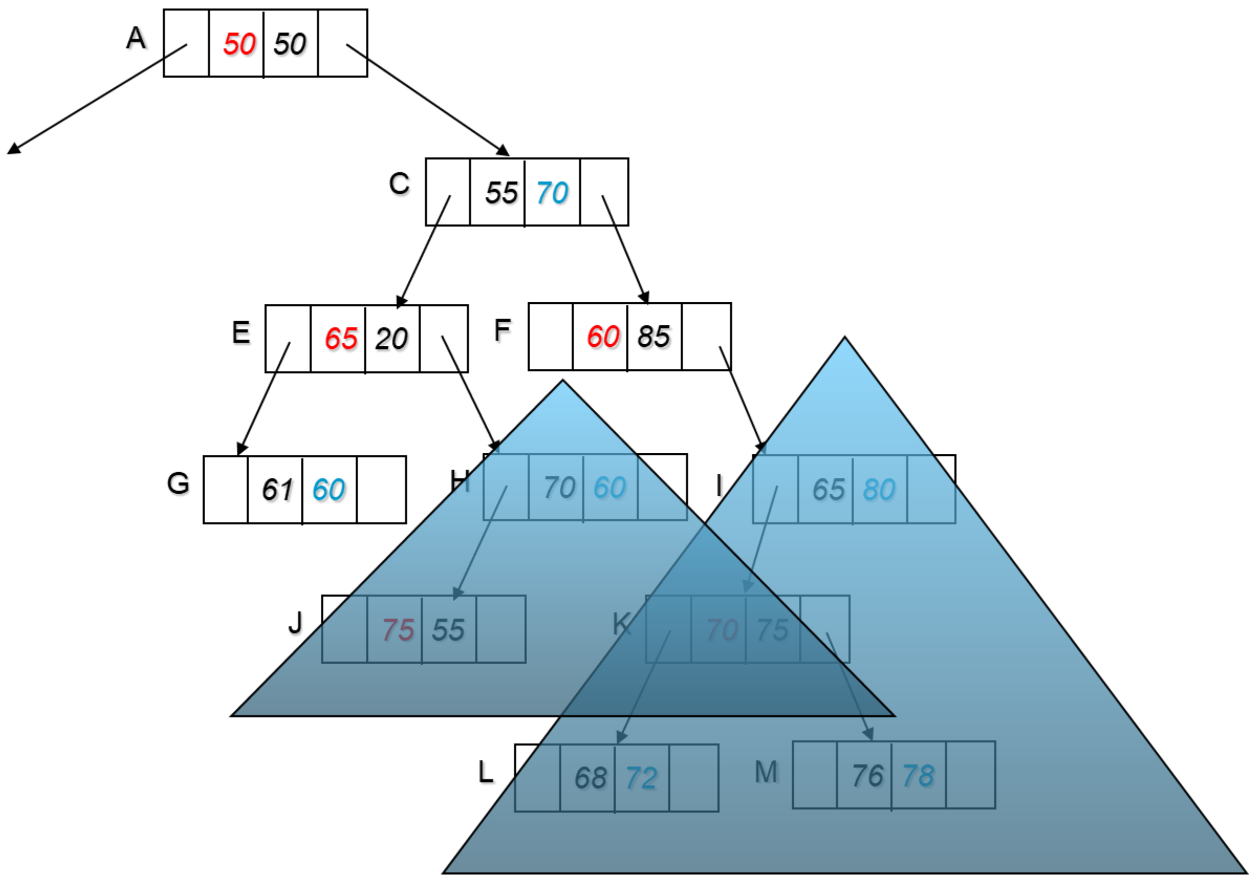
(f)



Multidimensional Search Tree

❖ Deletion in a KD-tree

- The parts of the KD-tree that do not need to be traversed when searching for the smallest value x



Multidimensional Search Tree

- Example uses of a KD-tree
  - Multidimensional k-nearest neighbor search
    - https://www.quora.com/What-is-a-kd-tree-and-what-is-it-used-for

1. Nearest Neighbor Search

Let's say you intend to build a **Social Cop** in your smartphone. Social Cop helps people report crimes to the nearest police station in real-time.

So what seems to be a problem here ?

Yes, you guessed it right. We need to search for the police station nearest to the crime location before attempting to report anything.

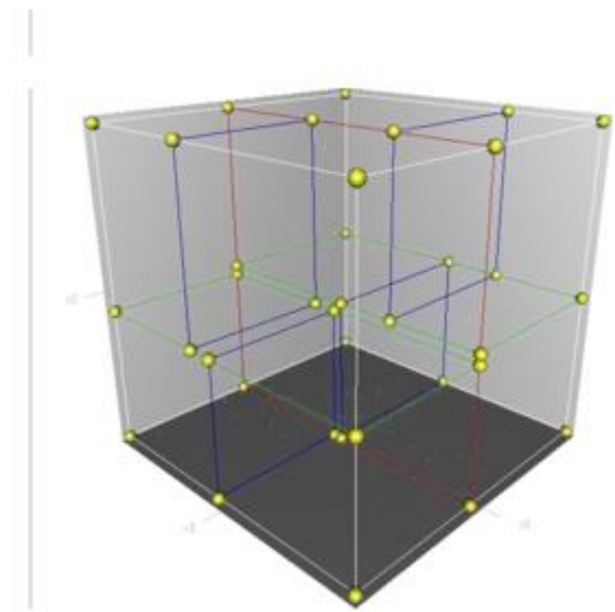
*How could we do it **quickly** ?*

Seems k-d trees can help you find the nearest neighbor to a point on a two dimensional map of your city. All you have to do is construct a 2 dimensional k-d tree from the locations of all the police stations in your city, and then query the k-d tree to find the nearest police station to any given location in the city.

Okay, I get what they can do. But how do they do it ?

If you already know how binary search trees work, understanding how k-d trees work would be nothing new. k-d trees help in partitioning space just as binary search trees help in partitioning the real line. k-d trees recursively partition a region of space, creating a binary space partition at each level of the tree.

This is what a 3 dimensional region of space partitioned by a 3 dimensional k-d tree looks like [1] :



Multidimensional Search Tree

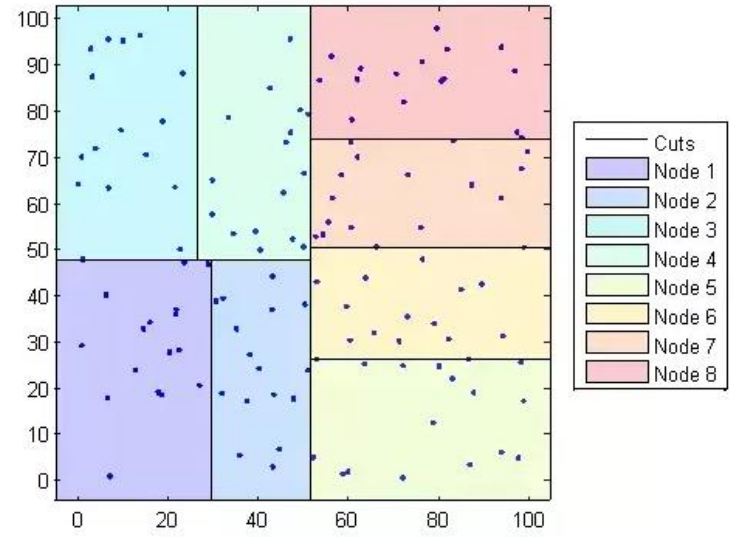
- Example uses of a KD-tree
  - Database query
    - https://www.quora.com/What-is-a-kd-tree-and-what-is-it-used-for

2. Database queries involving a multidimensional search key

A query asking for all the employees in the age-group of (40, 50) and earning a salary in the range of (15000, 20000) per month can be transformed into a geometrical problem where the age is plotted along the x-axis and the salary is plotted along the y-axis [4]

[4] The x-axis denotes the age of the employee in *years*, and the y-axis denotes the monthly salary in *thousand rupees*.

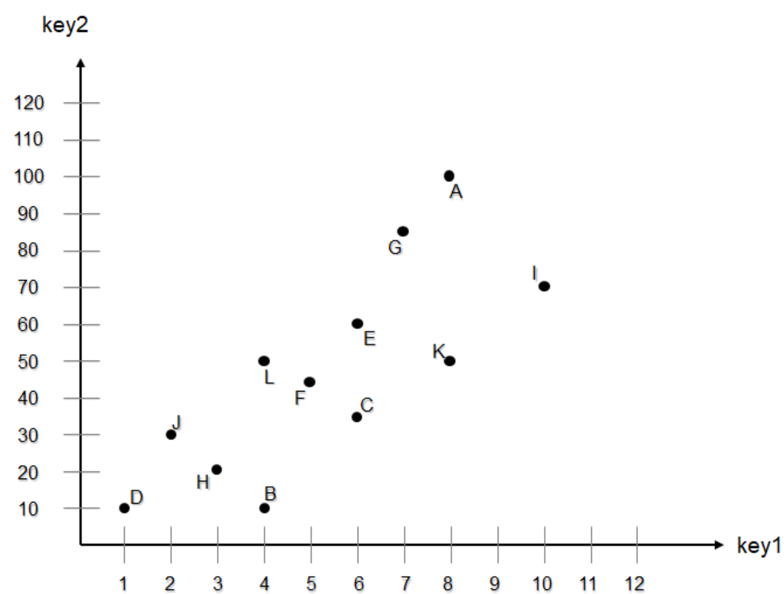
A 2 dimensional k-d tree on the composite index of ***(age, salary)*** could help you efficiently search for all the employees that fall in the rectangular region of space defined by the query described above.



Multidimensional Search Tree

- R-tree
  - Extend B-tree to handle multidimensional searches
  - Balanced tree
  - Nodes in an R-tree represent the minimum bounding space that contains all relevant keys

이름	Key1	Key2
A	8	100
B	4	10
C	6	35
D	1	10
E	6	40
F	5	45
G	7	85
H	3	20
I	10	70
J	2	30
K	8	50
L	4	50

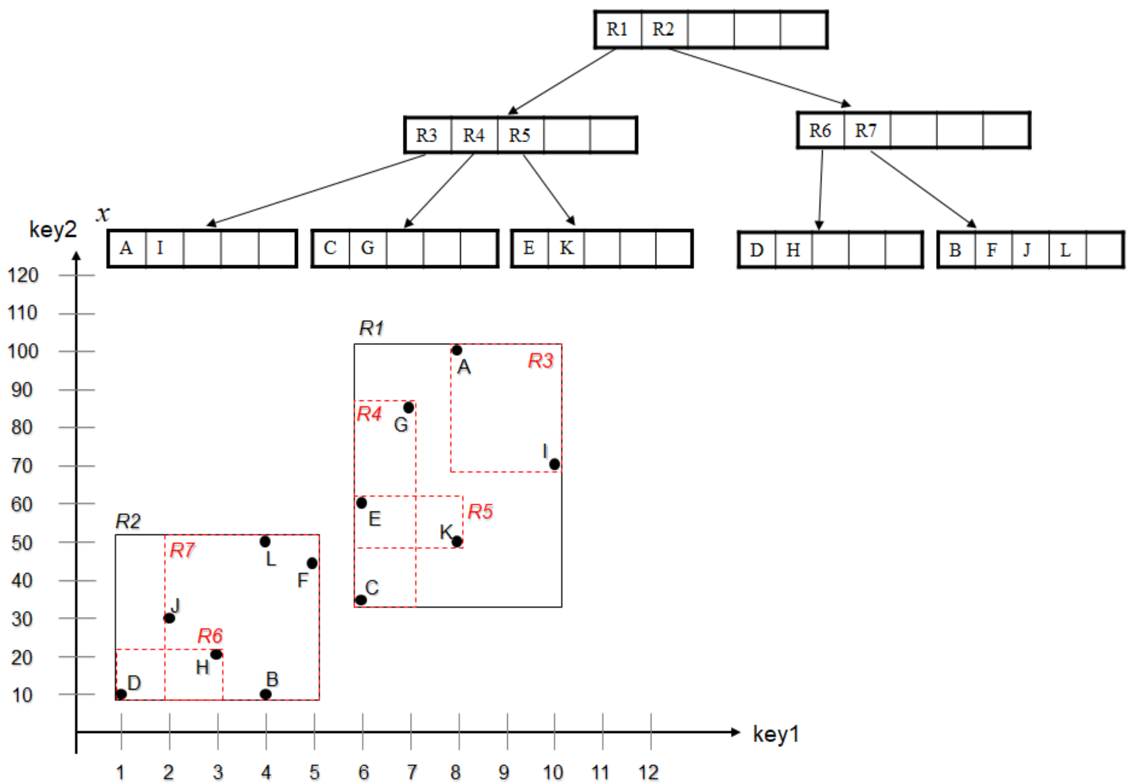


Multidimensional Search Tree

- Nodes in R-tree
  - Region page
    - Composed of multiple (region, page number) pairs
    - All internal nodes are region pages
  - Point page
    - All leaf nodes are point pages
- Upper and lower bounds on the number of regions in an R-tree
  - All internal nodes except the root have $\left\lfloor \frac{k}{2} \right\rfloor \sim k$ regions
  - All leaf nodes have the same depth
  - All records are referenced only in the leaf nodes

Multidimensional Search Tree

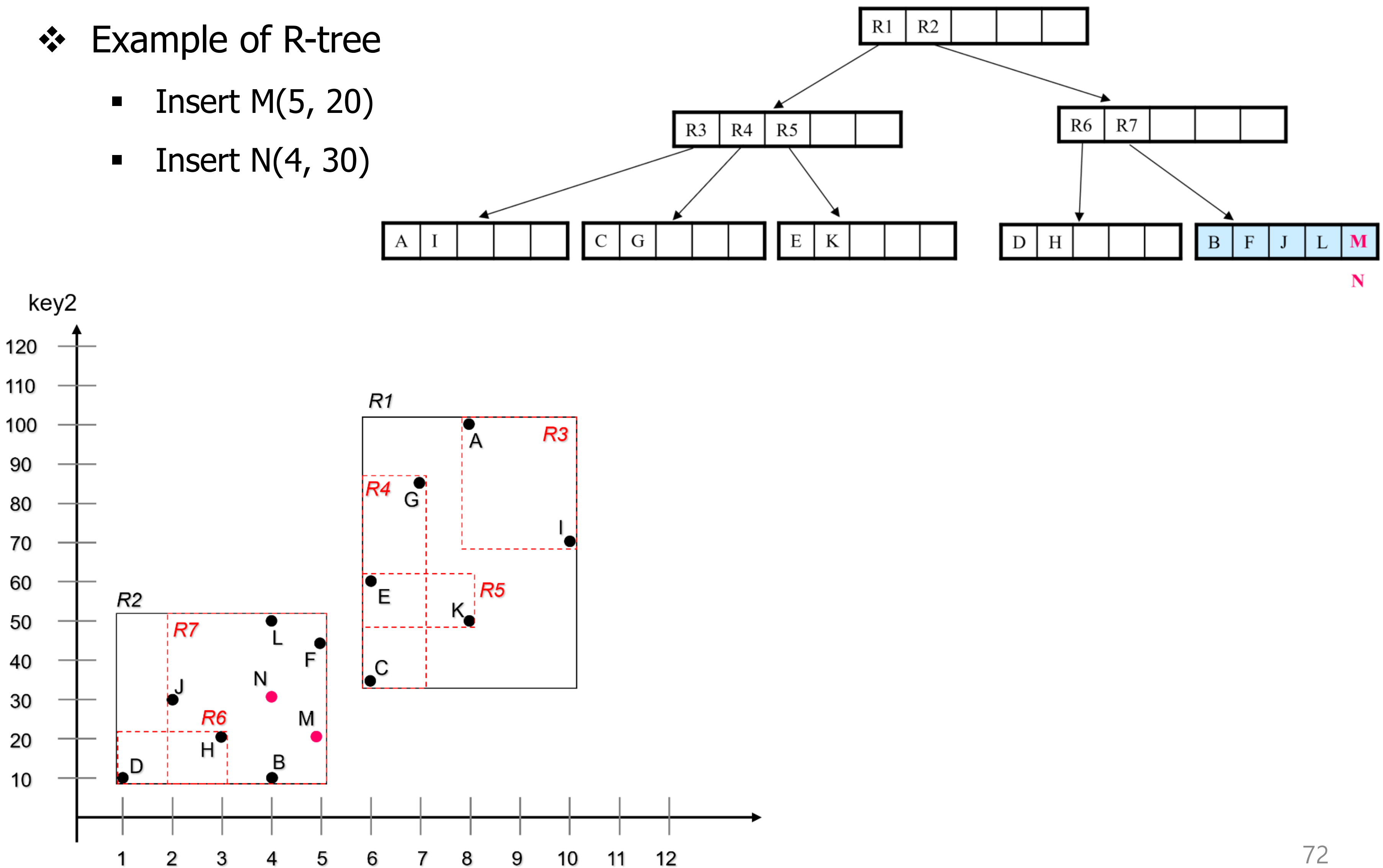
- Example of R-tree
  - If k=5, all nodes except the root have 2 to 5 elements
  - Leaf nodes contain the actual (key, page number) pairs
  - Can be included in overlapping regions



이름	Key1	Key2
A	8	100
B	4	10
C	6	35
D	1	10
E	6	40
F	5	45
G	7	85
H	3	20
I	10	70
J	2	30
K	8	50
L	4	50

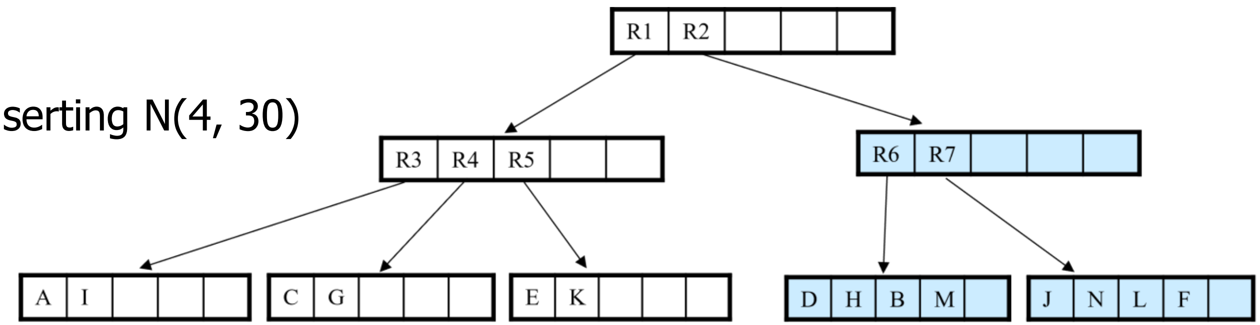
Multidimensional Search Tree

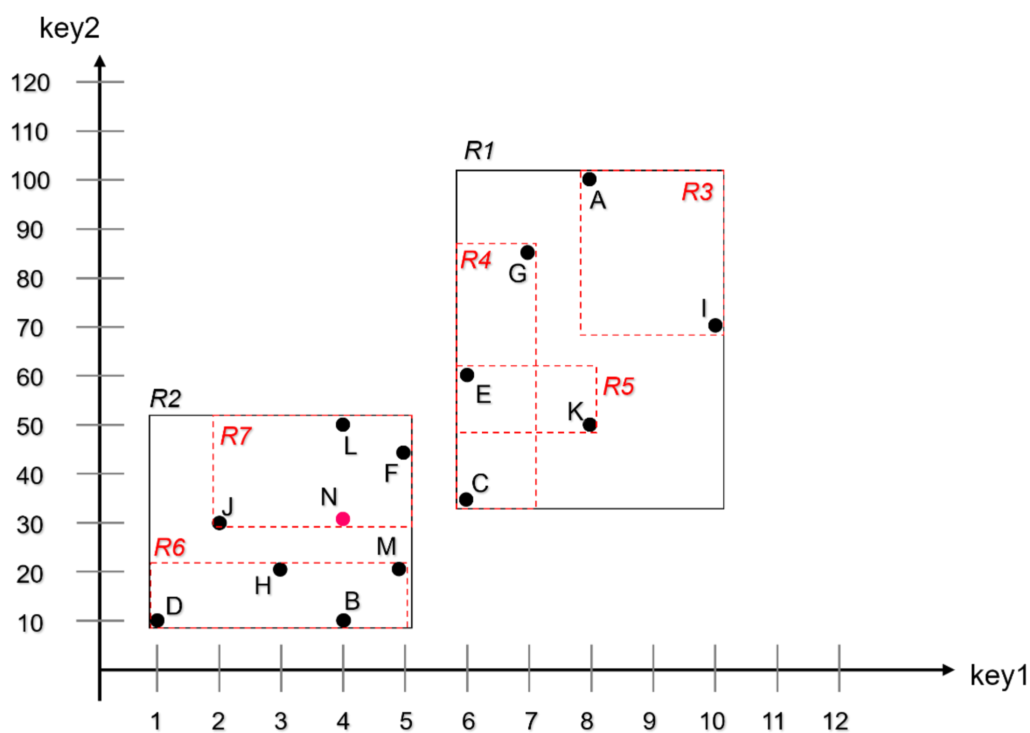
- Example of R-tree
  - Insert M(5, 20)
  - Insert N(4, 30)



Multidimensional Search Tree

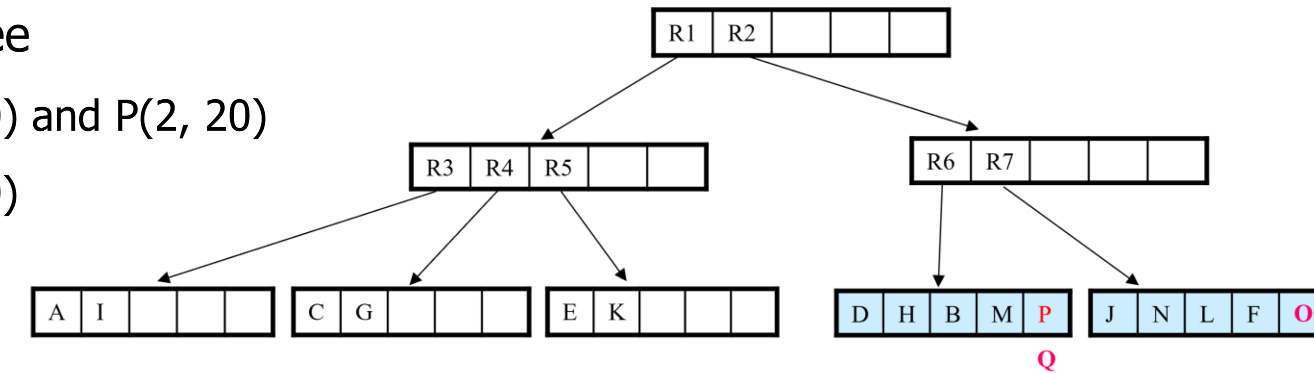
- Example of R-tree
  - Redistribution for inserting N(4, 30)

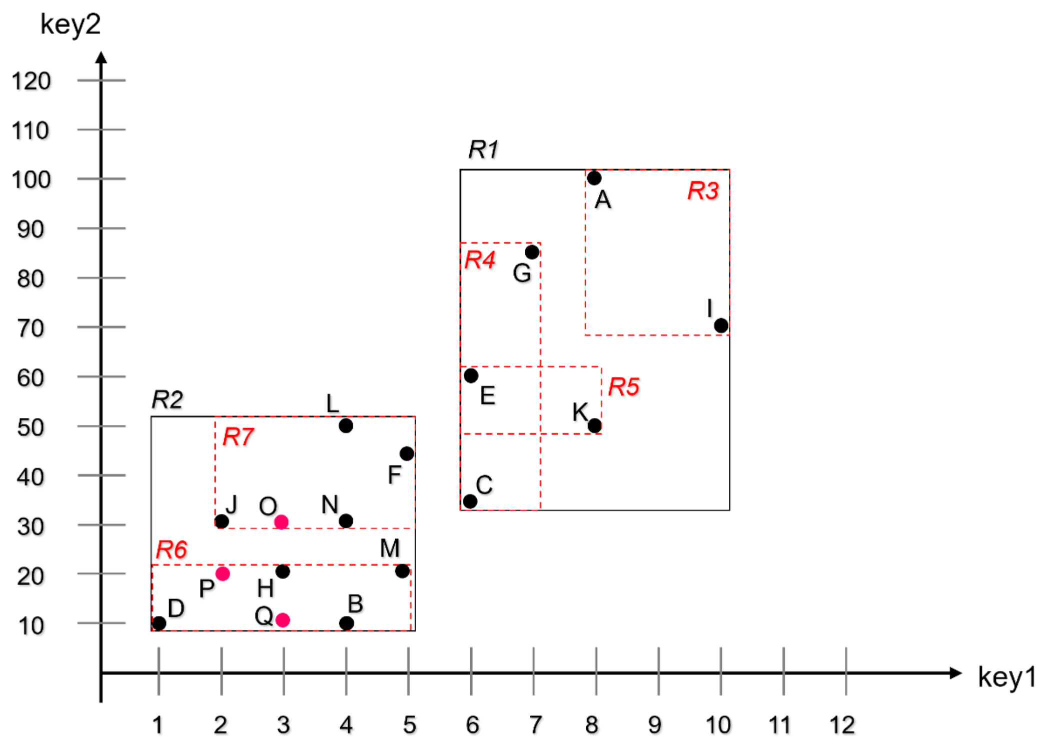




Multidimensional Search Tree

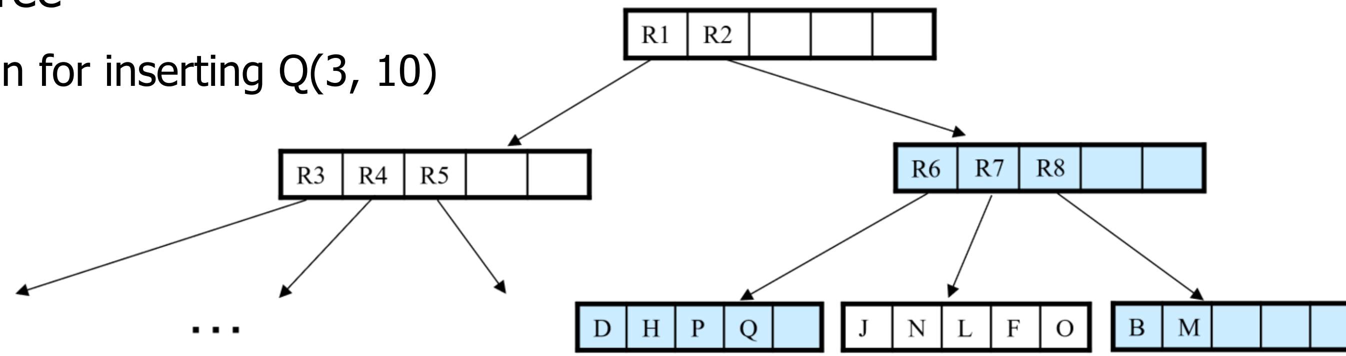
- Example of R-tree
  - Insert O(3, 30) and P(2, 20)
  - Insert Q(3, 10)

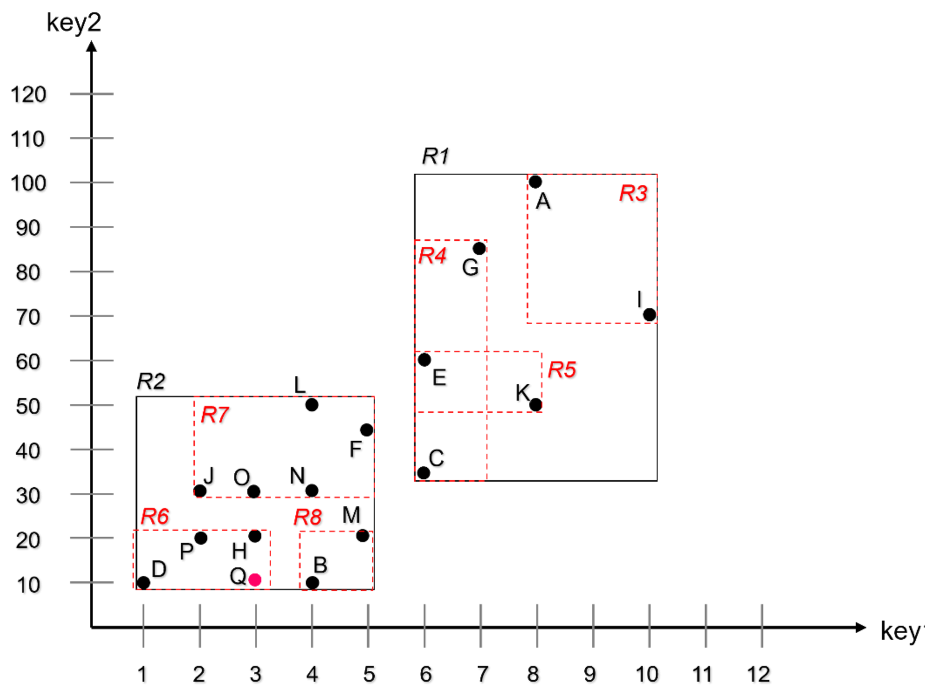




Multidimensional Search Tree

❖ Example of R-tree
  - Redistribution for inserting Q(3, 10)



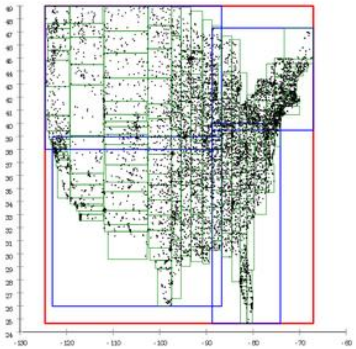


Multidimensional Search Tree

- Insertion and deletion in an R-tree
  - Similar to a B-tree

- In an R-tree, node regions can overlap
  - Allows flexibility in node adjustments
  - However, the path for record search may not be unique
  - This is improved in the R*-tree

- R-tree can also be used for storing and searching geometric shapes / data

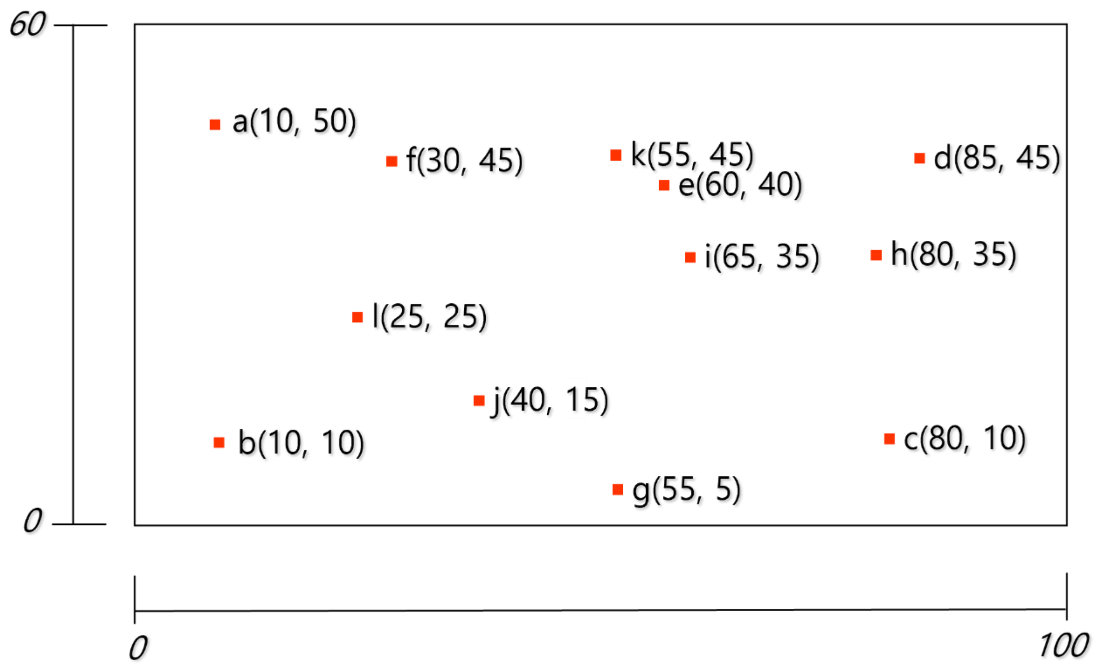


Multidimensional Search Tree

- Grid file
  - Divides the space into mutually exclusive grid regions
  - Stores only the records belonging to that region
  - Allows for fast storage and retrieval of arbitrary records

Multidimensional Search Tree

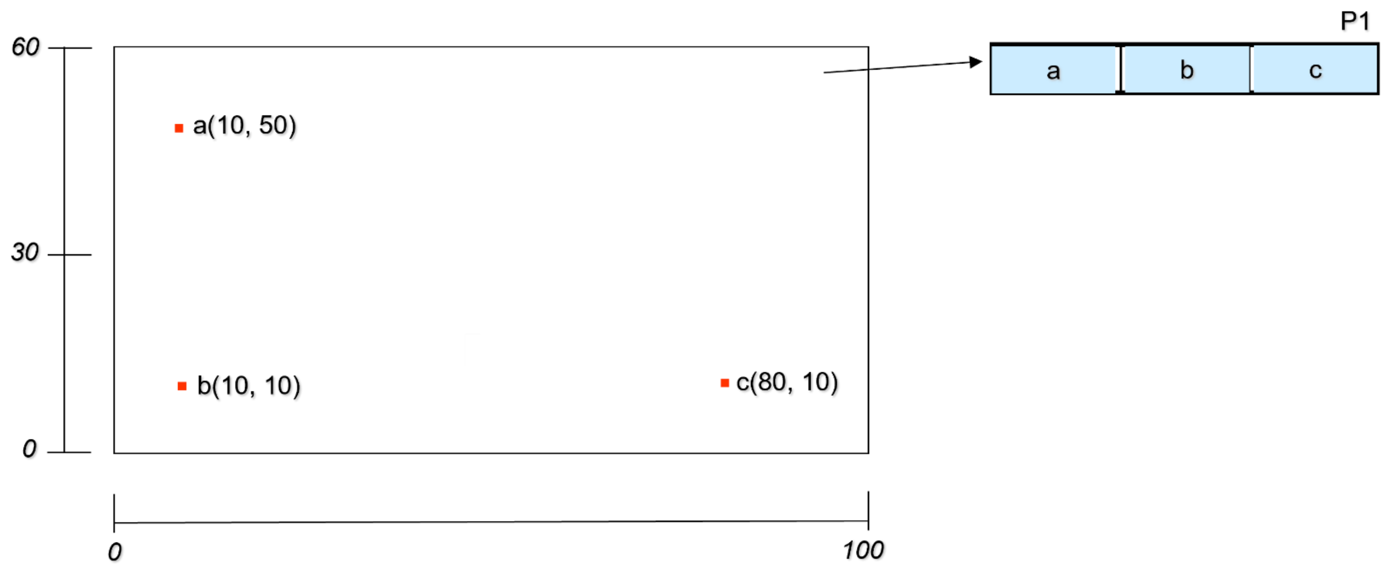
❖ Example of Grid file



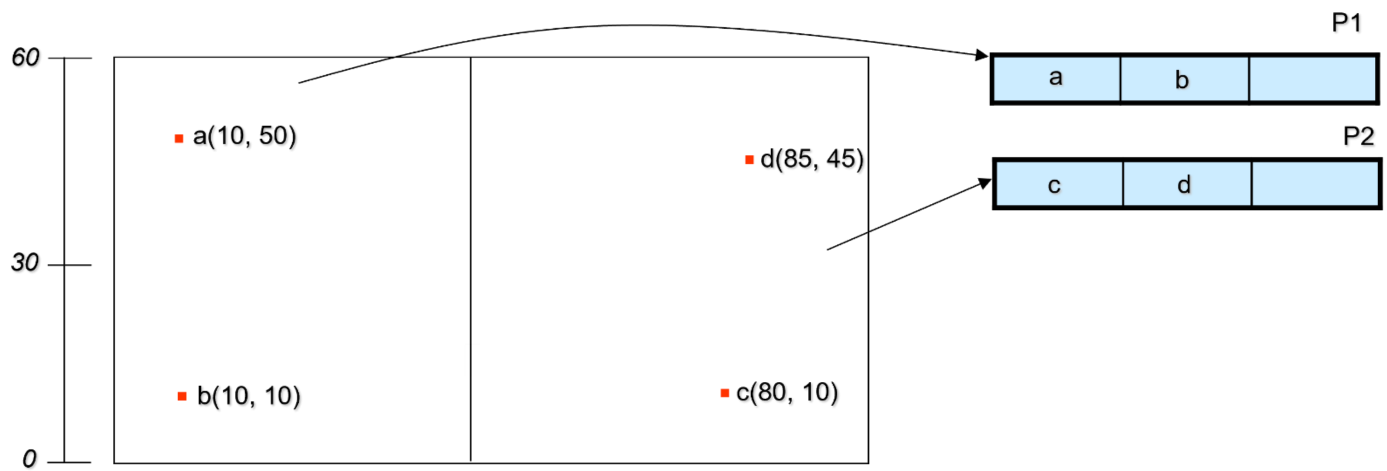
Multidimensional Search Tree

❖ Example of Grid file (cont'd)

(a,b,c)

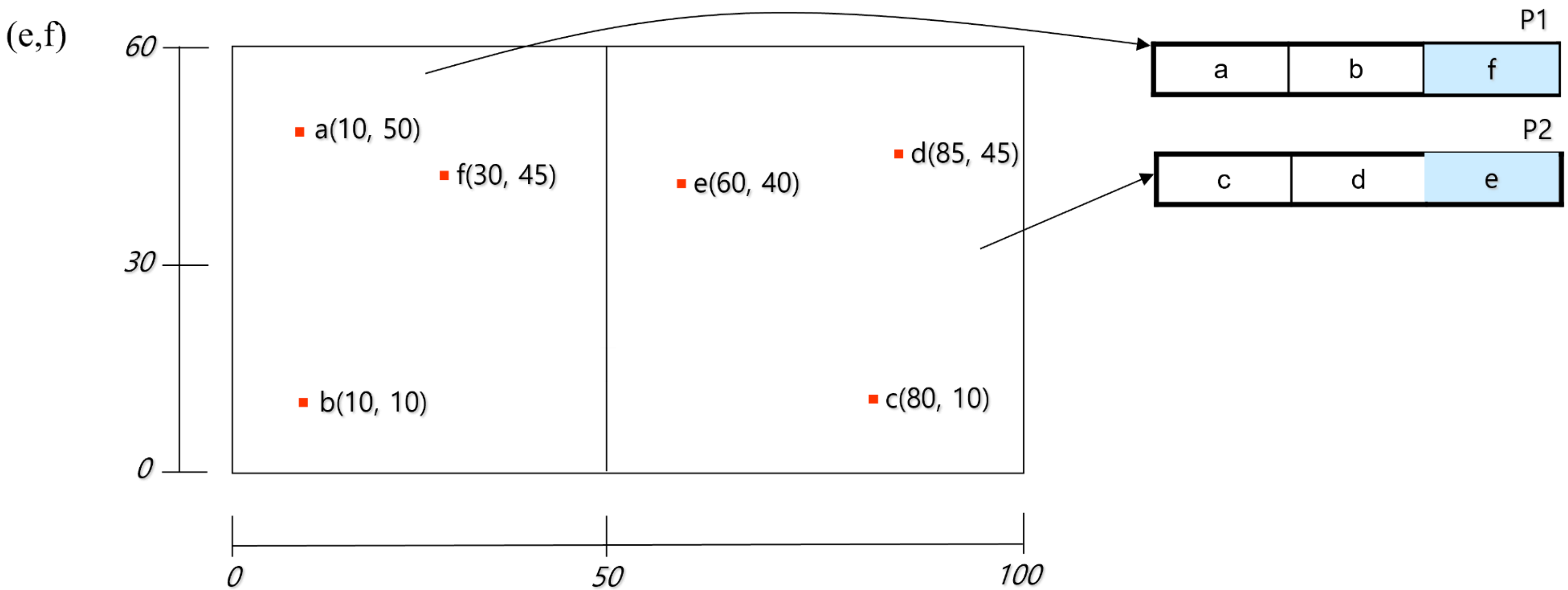


(d)



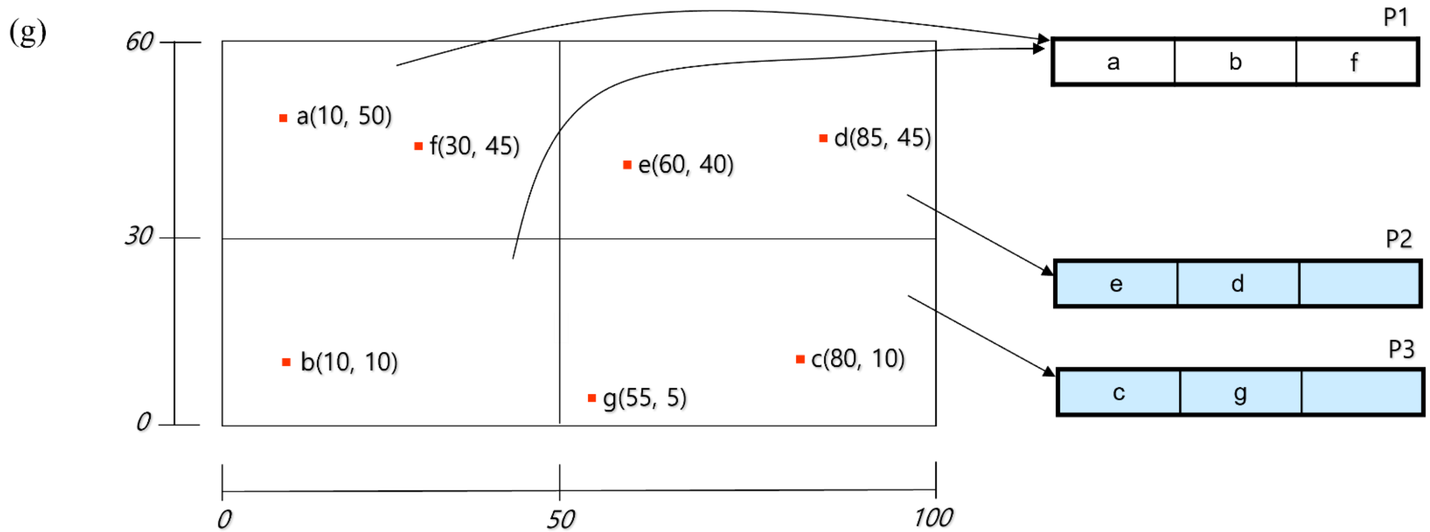
Multidimensional Search Tree

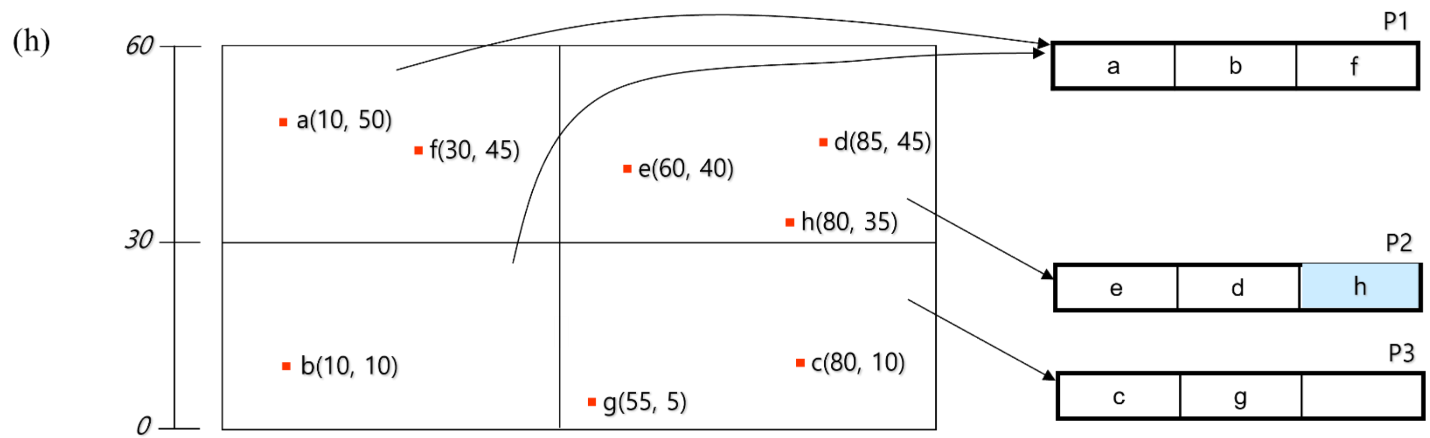
❖ Example of Grid file (cont'd)



Multidimensional Search Tree

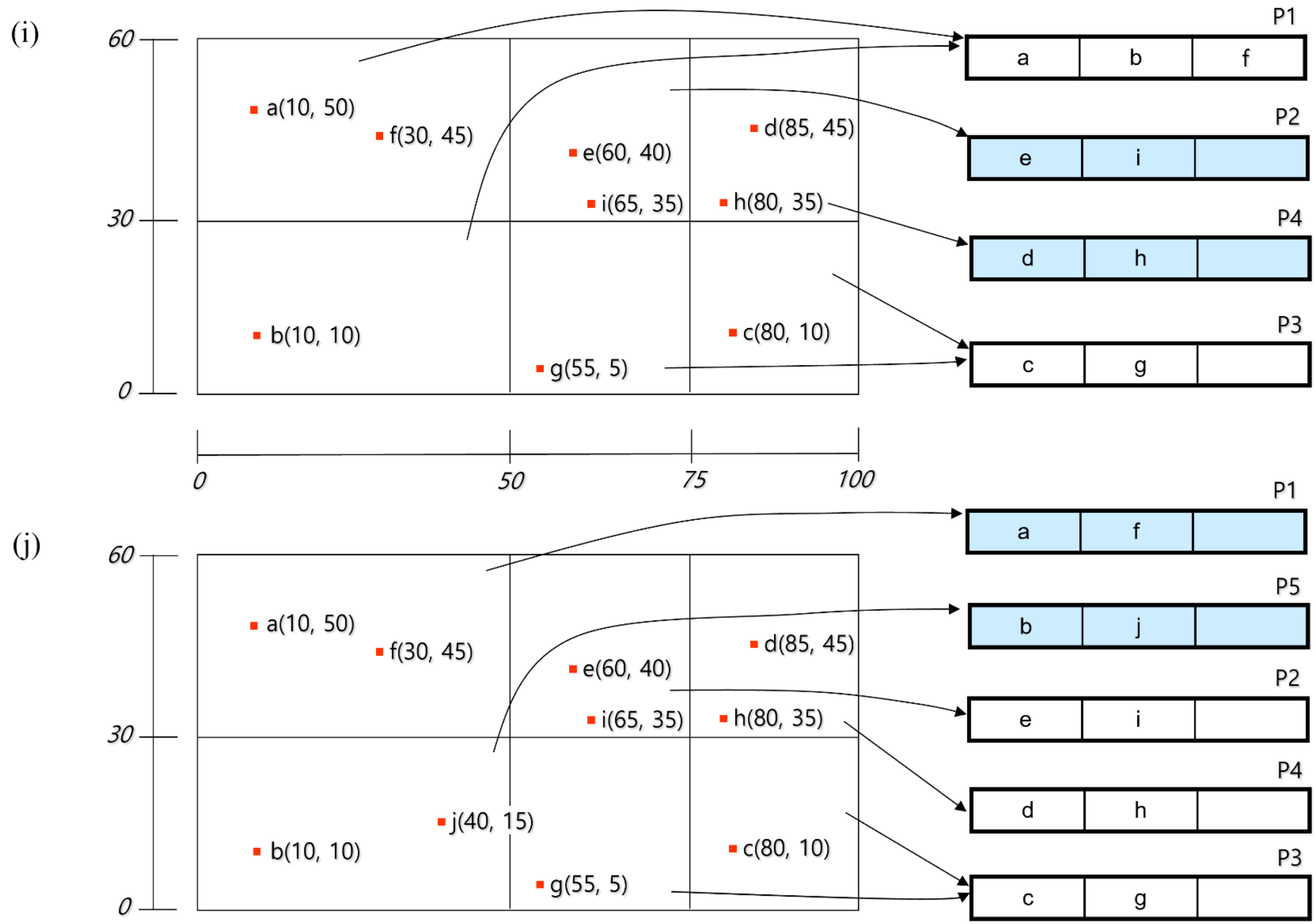
❖ Example of Grid file (cont'd)





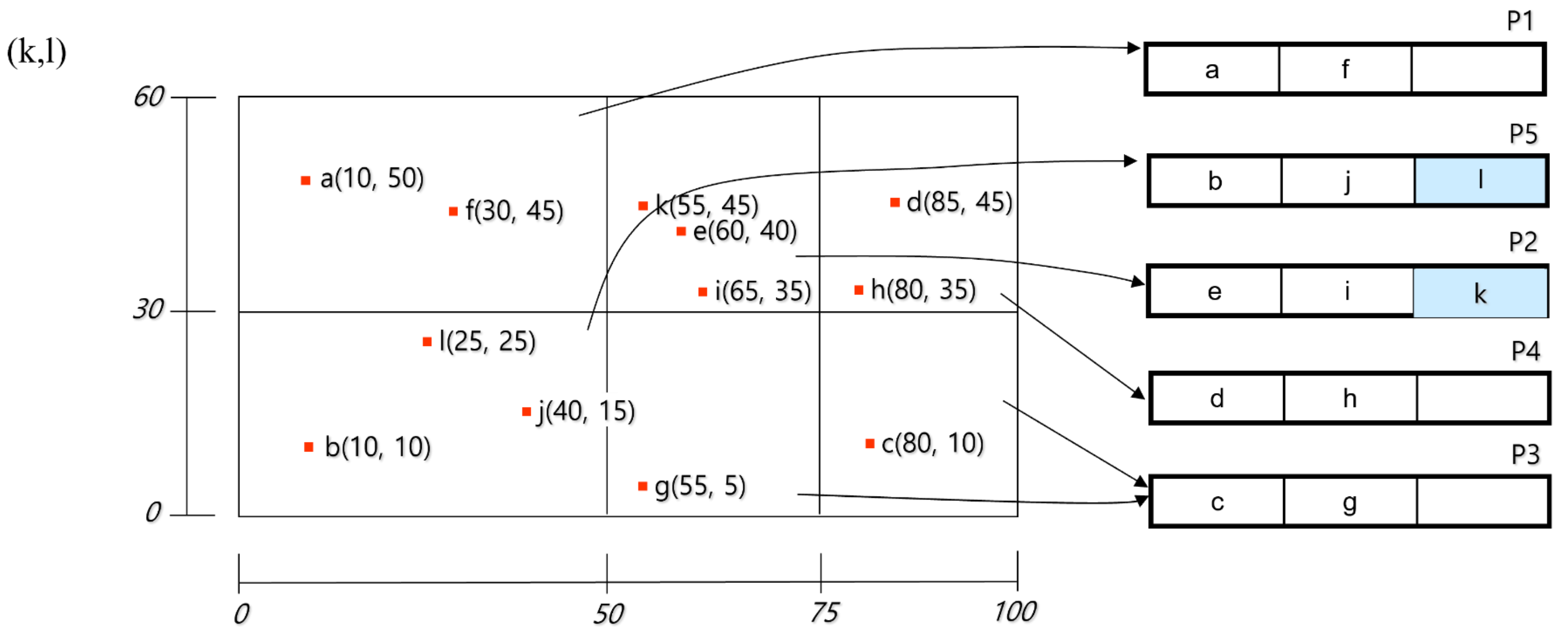
Multidimensional Search Tree

❖ Example of Grid file (cont'd)



Multidimensional Search Tree

❖ Example of Grid file (cont'd)



Multidimensional Search Tree

- Searching in a Grid file
  - If the grid array is not excessively large, it can be loaded into memory for use
    - However, this is not practically feasible
  - If the grid array is not excessively large, it can be loaded into memory for use
    - Determine the page where the target record is stored in the grid array, and then go to that page for use

Summary

- Red-Black tree
  - Balanced tree
  - Properties:
    - Every node is either Red or Black
    - Root node is Black
    - All leaf nodes (NILs) are Black
    - No double Red
    - Same Black depth

- B-tree
  - Multiple branches
  - Useful as an external tree
  - Properties:
    - All nodes except the root have $\lfloor k/2 \rfloor \sim k-1$ keys
    - All leaf nodes have the same depth

- Multidimensional search tree
  - KD-tree
  - KDB-tree
  - R-tree
  - Grid file

Questions?

SEE YOU NEXT TIME!