B^+-tree

Younghoon Kim
(nongaussian@hanyang.ac.kr)

Dictionary data structures

- Two main choices:
  - Hashtables (e.g., dynamic(extendible) hashtable)
  - Trees (e.g., B-tree, B^+-tree)
- Some IR systems use hashtables, some trees

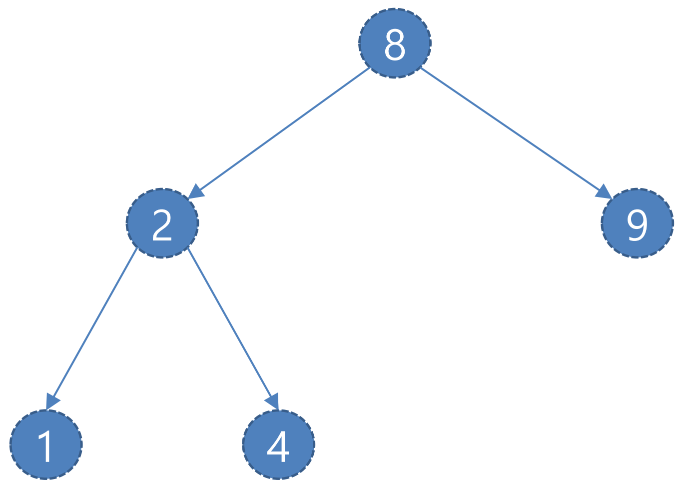
Hashtables

- Each vocabulary term is hashed to an integer
  - (We assume you've seen hashtables before)
- Pros:
  - Lookup is faster than for a tree: O(1)
- Cons:
  - No easy way to find minor variants:
    - judgment/judgement
  - No prefix search [i.e., tolerant retrieval (X)]
  - If vocabulary keeps growing, need to occasionally do the expensive operation of rehashing *everything*
  - *Waste memory space!*
  - *In the worst case, it performs terribly!*
  - *Irregular search time!*

Trees

- Simplest: binary tree
- More usual: B^+-trees
- Trees require a standard ordering of characters and hence strings … but we typically have one
- Pros:
  - Solves the **prefix search problem** (e.g., terms starting with *hany*)
  - Optimized for disk-based retrieval
- Cons:
  - Slower: O(log M) [and this requires balanced tree]
    - But it always guarantees a regular search time for every query
  - Rebalancing binary trees is expensive
    - But B-trees mitigate the rebalancing problem

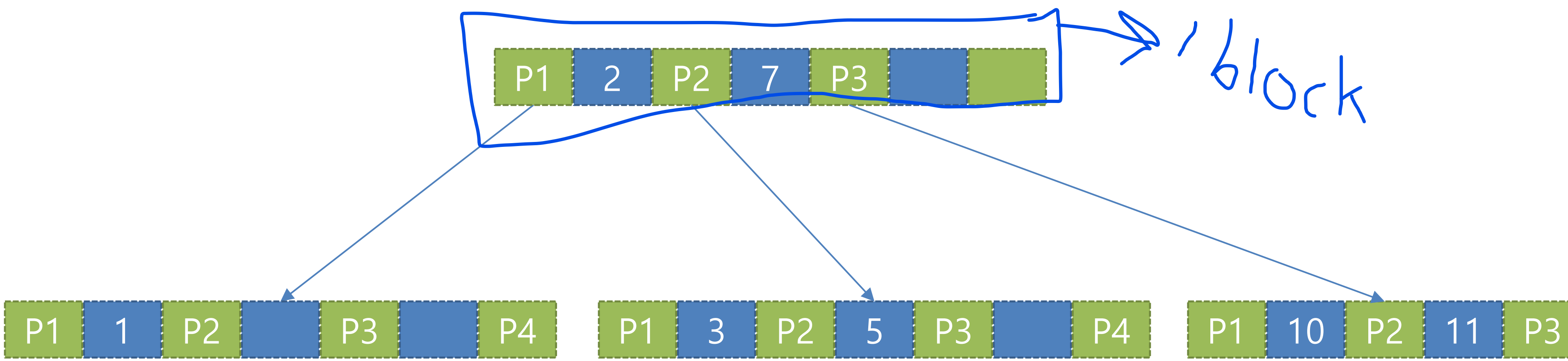
Trees



Binary trees

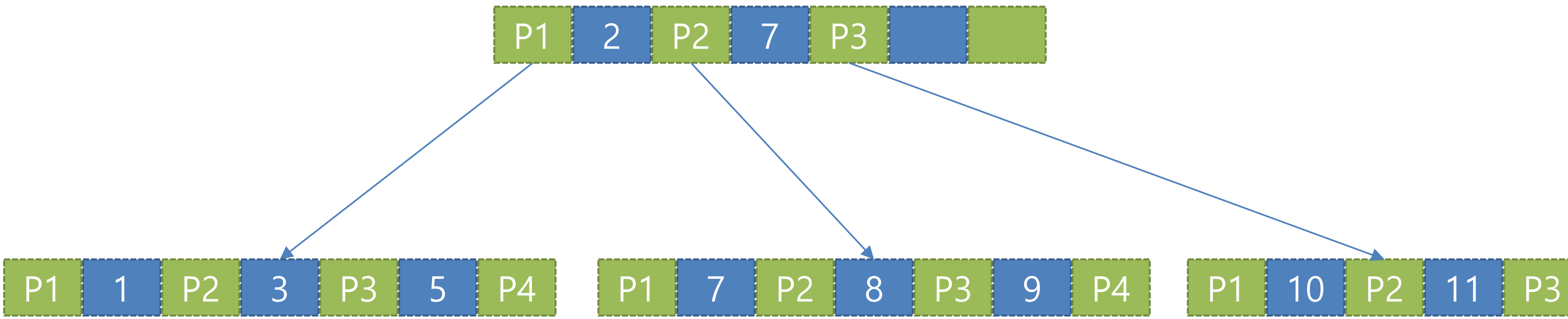
- in-memory index
- 2 children
- Balancing: AVL, red-black trees

Trees



B-tree
- Block-based I/O
- Multiple children
- Balanced
- Keys are stored in both leaf and non-leaf nodes

Trees



B⁺-tree

- Block-based I/O
- Multiple children
- Balanced
- All keys are stored in leaf nodes only

B^+-tree

- Basic concepts
- Ordered indices
- Building and searching a B^+-tree
  - Basic operations
    - Insert
    - Delete
    - Search

Interface for Module 3

```
public interface BPlusTree {

    /**
     * Opening and initializing the directory
     *
     * @param metafile A meta-file with configurations for the dictionar
     * @param filepath Directory or path for opening the dictionary
     * @param blocksize Available blocksize in the main memory of the cu
     * @param nblocks Available block numbers in the main memory of the 
     * @throws IOException Exception while opening B+ tree
     */
    void open(String metafile, String filepath,
              int blocksize, int nblocks) throws IOException;

    /**
     * Searching for a key
     *
     * @param keyThe integer key of index term to search
     * @returnStatus code
     * @throws IOExceptionException while accessing B+ tree
     */
```

```java
    /**
     * Searching for a key
     *
     * @param keyThe integer key of index term to search
     * @returnStatus code
     * @throws IOExceptionException while accessing B+ tree
     */
    int search(int key) throws IOException;

    /**
     * Inserting a key and the bound value
     *
     * @param key Key
     * @param val Value
     * @throws IOExceptionException while accessing B+ tree
     */
    void insert(int key, int val) throws IOException;


    /**
     * Closing the dictionary
     *
     * @throws IOExceptionException while closing B+ tree
     */
    void close() throws IOException;
}
```

BASICS OF INDEX

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer

- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

Index Evaluation Metrics

- Access types
  - E.g., sequential access in a sorted order
- Access time
- Insertion time
- Deletion time
- Space overhead

Ordered Indices

- In an **ordered index**, index entries are stored sorted on the search key value.
  - E.g., author catalog in library
- **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index

Dense Index Files

- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *student* relation

INDEX(ID)

DIR.
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011

ID	Name	Major	Birth
1000	Suji	CS	2001
1001	Wuwan	Finance	2002
1002	Minhee	Physics	1999
1003	Fujiko	Physics	1996
1004	Ehwa	History	2000
1005	Giljun	Physics	2000
1006	Kang	CS	2001
1007	Canna	History	1999
1008	Sinji	Finance	1995
1009	Choi	Biology	2001
1010	Bok	CS	2000
1011	Kim	EE	1999

Dense Index Files (Cont.)

- Dense index on *major*, with *student* file sorted on *major*

INDEX(Major)

Major
Biology
CS
EE
Finance
History
Physics

ID	Name	Major	Birth
1009	Choi	Biology	2001
1000	Suji	CS	2001
1006	Kang	CS	2001
1010	Bok	CS	2000
1011	Kim	EE	1999
1001	Wuwan	Finance	2002
1008	Sinji	Finance	1995
1004	Ehwa	History	2000
1007	Canna	History	1999
1002	Minhee	Physics	1999
1003	Fujiko	Physics	1996
1005	Giljun	Physics	2000

Sparse Index Files

- **Sparse Index**: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key

1002 ?

INDEX(ID)

DIR.
1000
1004
1008

ID	Name	Major	Birth
1000	Suji	CS	2001
1001	Wuwan	Finance	2002
1002	Minhee	Physics	1999
1003	Fujiko	Physics	1996
1004	Ehwa	History	2000
1005	Giljun	Physics	2000
1006	Kang	CS	2001
1007	Canna	History	1999
1008	Sinji	Finance	1995
1009	Choi	Biology	2001

1006 ?

To locate a record with search-key value *K* :
- Find index record with largest search-key value < *K*
- Search file sequentially starting at the record to which the index record points

Secondary Indices Example

- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.

INDEX(Birth)

Birth
1995
1996
1999
2000
2001
2002

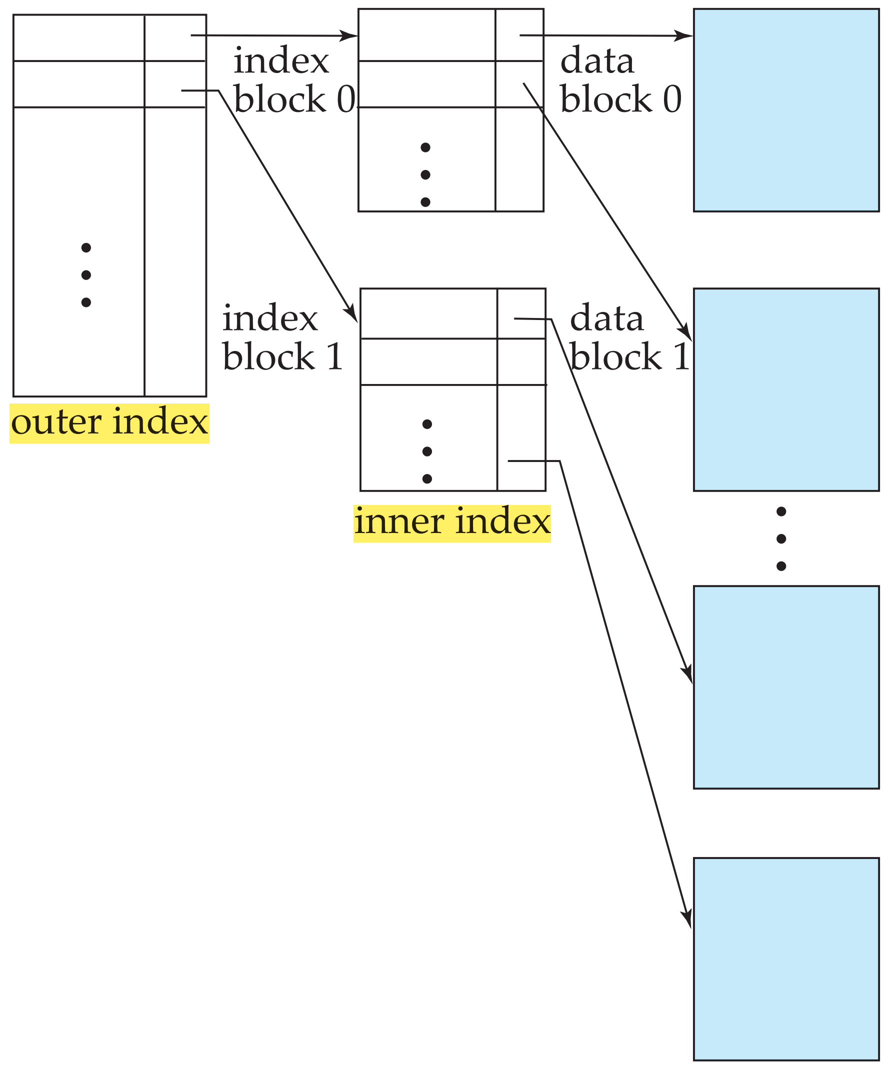
SEC	ID	Name	Major	Birth
	1000	Suji	CS	2001
	1001	Wuwan	Finance	2002
	1002	Minhee	Physics	1999
	1003	Fujiko	Physics	1996
	1004	Ehwa	History	2000
	1005	Giljun	Physics	2000
	1006	Kang	CS	2001
	1007	Canna	History	1999
	1008	Sinji	Finance	1995
	1009	Choi	Biology	2001
	1010	Bok	CS	2000
	1011	Kim	EE	1999

Secondary index on *birth* field of *student*

Multilevel Index

- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

Multilevel Index (Cont.)



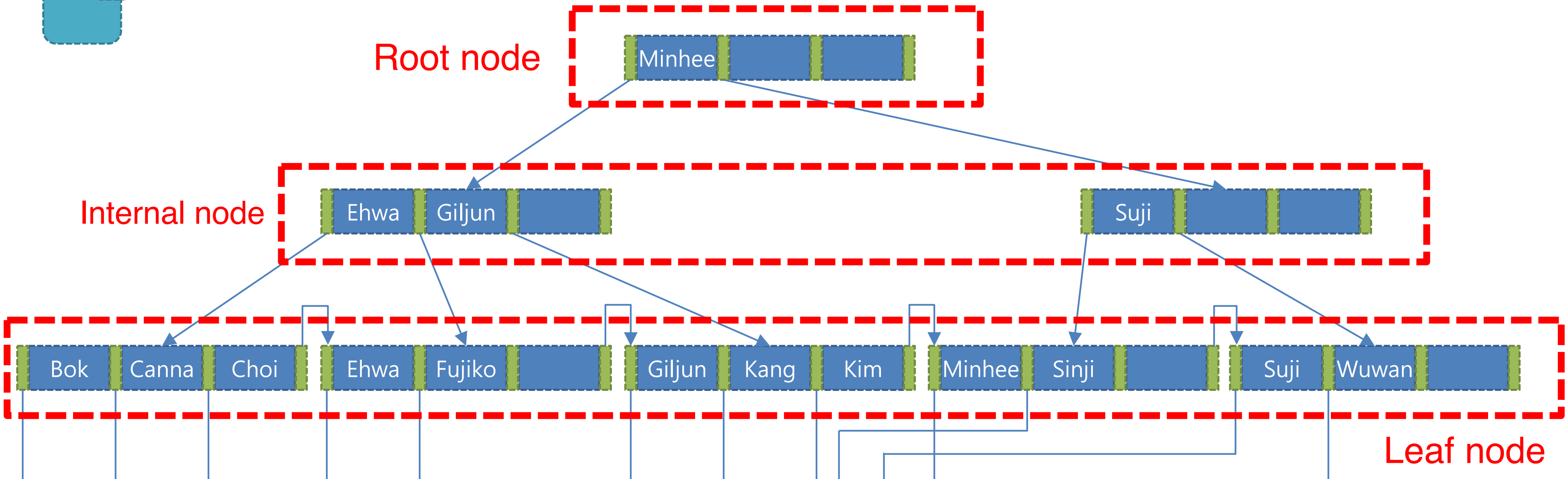
B^+-TREE

B^+-Tree Index Files

B^+-tree indices are an alternative to indexed-sequential files.

- Disadvantage of indexed-sequential files
  - performance degrades as file grows, since many overflow blocks get created.
  - Periodic reorganization of entire file is required.
- Advantage of B^+-tree index files:
  - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B^+-trees:
  - extra insertion and deletion overhead, space overhead.
- Advantages of B^+-trees outweigh disadvantages
  - B^+-trees are used extensively

Example of B^+-Tree



ID	Name	Major	Birth
1000	Suji	CS	2001
1001	Wuwan	Finance	2002
1002	Minhee	Physics	1999
1003	Fujiko	Physics	1996
1004	Ehwa	History	2000
1005	Giljun	Physics	2000
1006	Kang	CS	2001
1007	Canna	History	1999
1008	Sinji	Finance	1995
1009	Choi	Biology	2001
1010	Bok	CS	2000
1011	Kim	EE	1999

B+-Tree Index Files (Cont.)

A B^+-tree is a rooted tree satisfying the following properties:

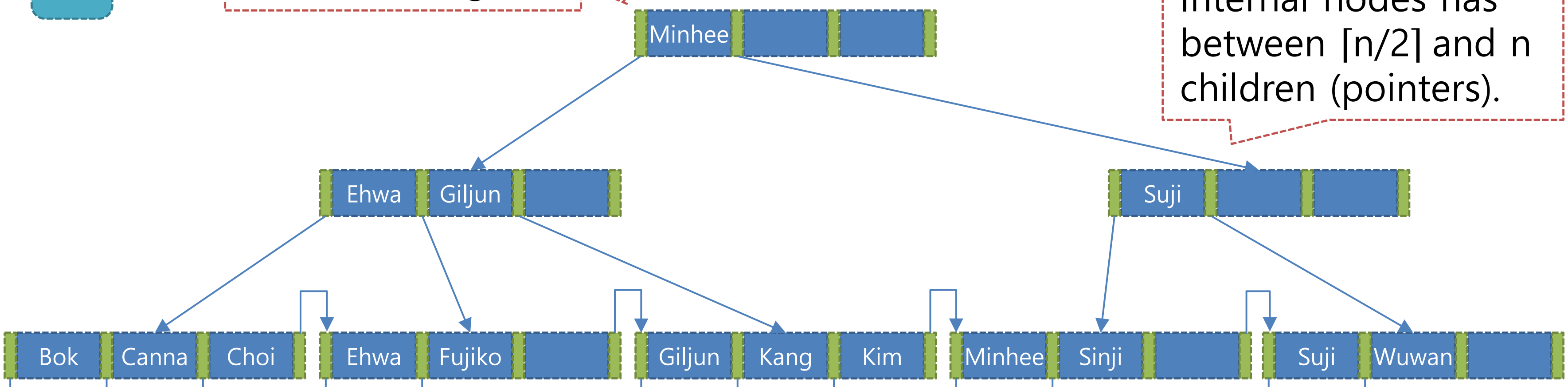
*Fanout **n** of a node*: the number of pointers out of the node

- All paths from root to leaf are of the same length (balanced)
- **Internal nodes (that is not a root or a leaf)** has between ⌈n/2⌉ and n children.
- **A leaf node** has between ⌈(n–1)/2⌉ and n–1 values
- **A root** has at least 2 children.
- Special cases:
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and (n–1) values.

Example of B+-Tree

All paths from root to leaf are of the same length

Internal nodes has between ⌈n/2⌉ and n children (pointers).



A leaf node has between ⌈(n–1)/2⌉ and n–1 values

ID	Name	Major	Birth
1000	Suji	CS	2001
1001	Wuwan	Finance	2002
1002	Minhee	Physics	1999
1003	Fujiko	Physics	1996
1004	Ehwa	History	2000
1005	Giljun	Physics	2000
1006	Kang	CS	2001
1007	Canna	History	1999
1008	Sinji	Finance	1995
1009	Choi	Biology	2001
1010	Bok	CS	2000
1011	Kim	EE	1999

B$^+$-Tree Node Structure

- Typical node

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

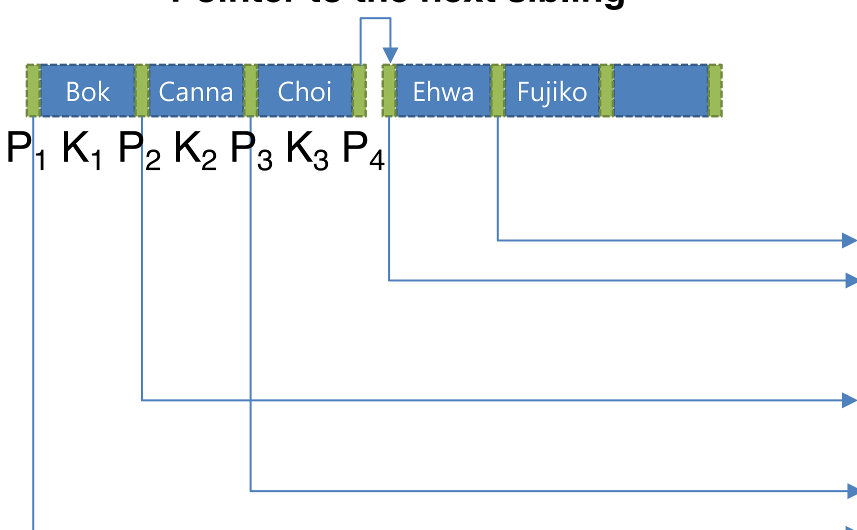
(Initially assume no duplicate keys, address duplicates later)

Leaf Nodes in B^+-Trees

Properties of a leaf node:

- For $i = 1, 2, . . ., n–1$, pointer P_i points to a file record with search-key value K_i,
- If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than or equal to L_j's search-key values (i.e., increasing or decreasing order)
- P_n points to next leaf node in search-key order

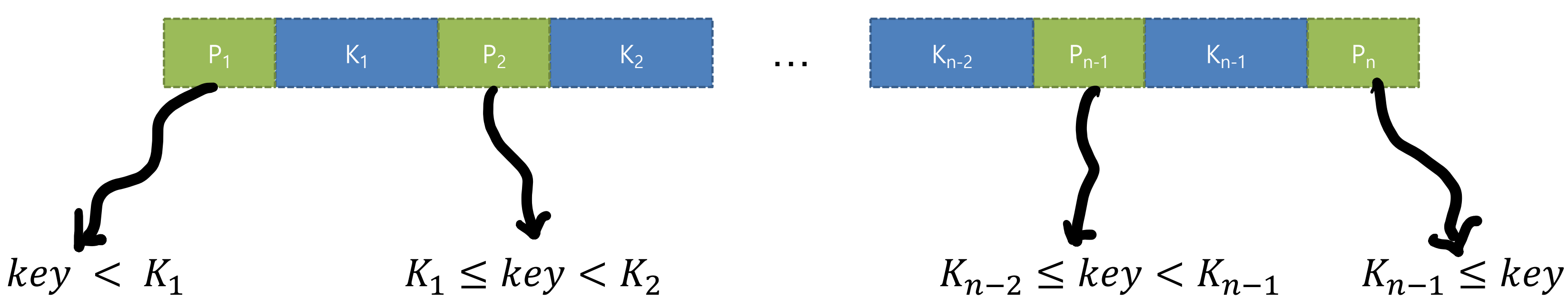
Pointer to the next sibling



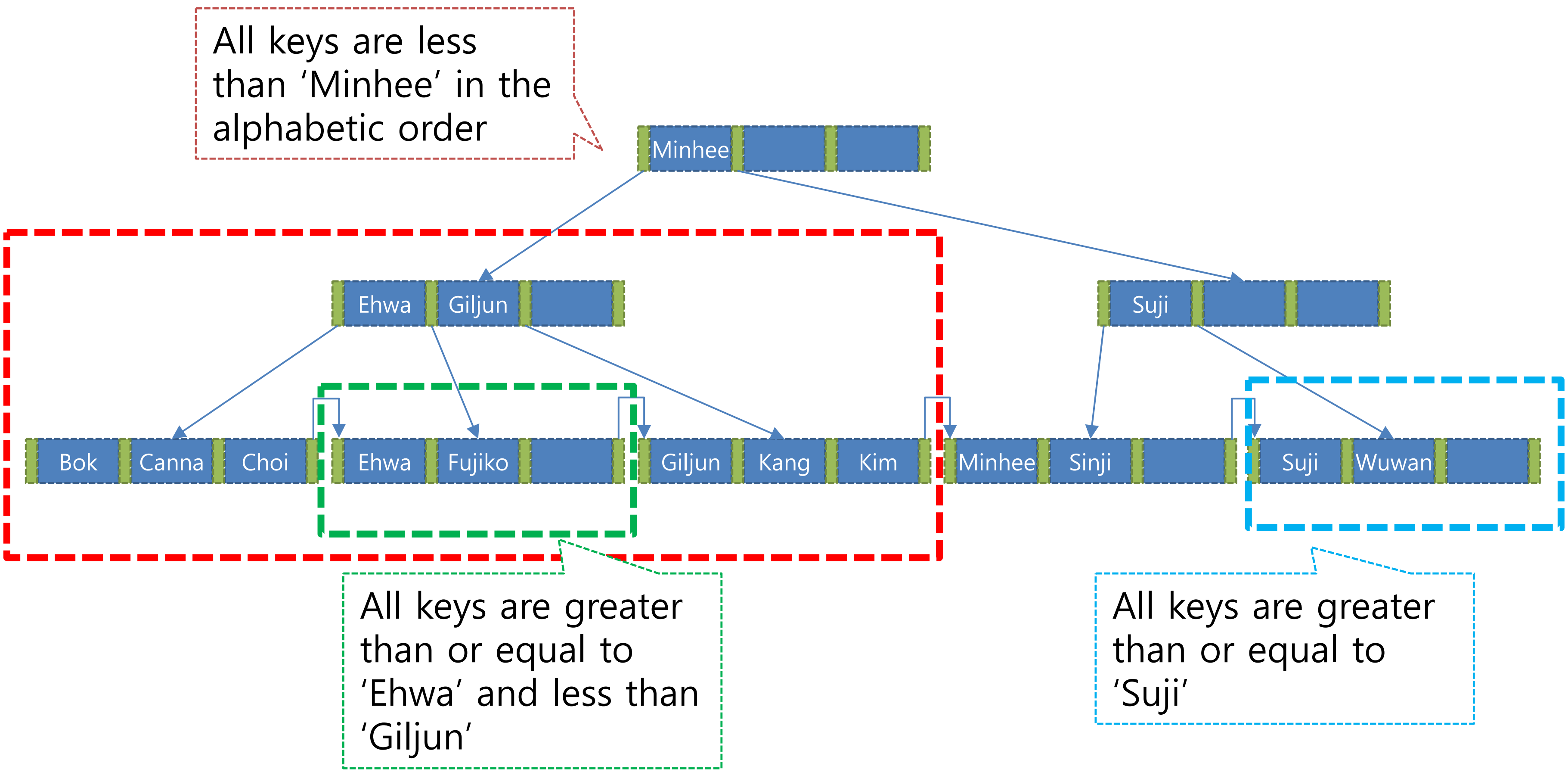
ID	Name	Major	Birth
1000	Suji	CS	2001
1001	Wuwan	Finance	2002
1002	Minhee	Physics	1999
1003	Fujiko	Physics	1996
1004	Ehwa	History	2000
1005	Giljun	Physics	2000
1006	Kang	CS	2001
1007	Canna	History	1999
1008	Sinji	Finance	1995
1009	Choi	Biology	2001
1010	Bok	CS	2000
1011	Kim	EE	1999

Non-Leaf Nodes in B⁺-Trees

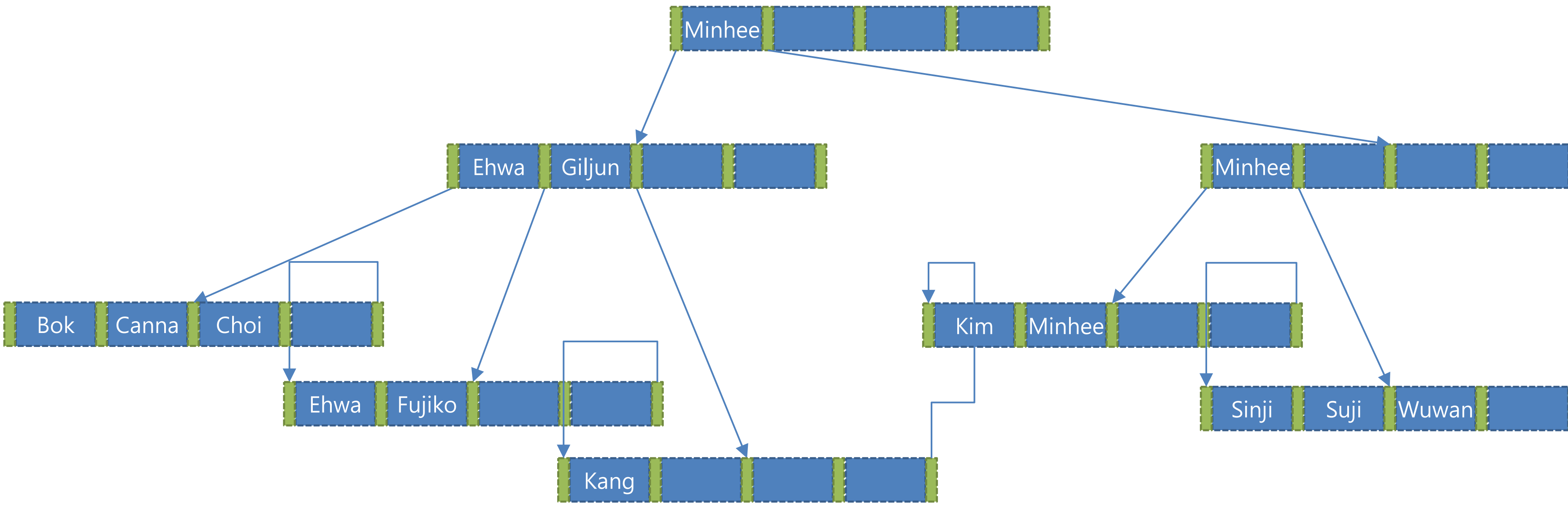
- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with *n* pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
  - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}



Example of B⁺-Tree



Example of B^+-tree

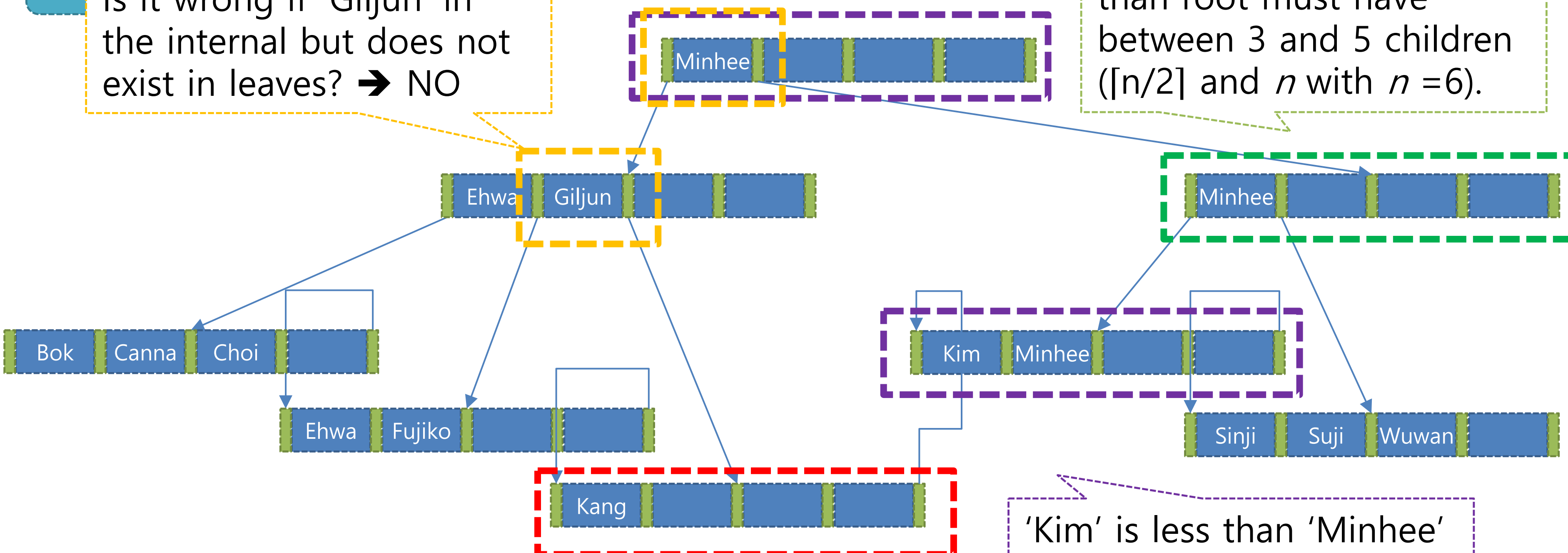


Exam ree

It is okey for the root to have only 2 children as an exception against half-rule

Is it wrong if `Giljun' in the internal but does not exist in leaves? ➔ NO

Non-leaf nodes other than root must have between 3 and 5 children (⌈n/2⌉ and *n* with *n* =6).



Leaf nodes must have between 2 and 4 values (⌈(n–1)/2⌉ and *n* –1, with *n* = 5).

'Kim' is less than 'Minhee' but in the subtree of right pointer of the root

Observations about B⁺-trees

- The non-leaf levels of the B⁺-tree form a hierarchy of sparse indices.
- The B⁺-tree contains a relatively small number of levels ($\lceil n/2 \rceil \geq 2$)
  - Level below root has at least 2* $\lceil n/2 \rceil$ pointers
  - Next level has at least 2* $\lceil n/2 \rceil$ * $\lceil n/2 \rceil$ pointers
  - …
  - Final level has at least 2* $\lceil n/2 \rceil^{H-2} \lceil (n-1)/2 \rceil$ pairs of key and pointer

$1 > \frac{m-1}{m} \geq \frac{1}{2}$ with $m \geq 2$

$$K \geq 2\left\lceil\frac{n}{2}\right\rceil^{H-2}\left\lceil\frac{n-1}{2}\right\rceil \geq 2\left\lceil\frac{n}{2}\right\rceil^{H-2}\left(\left\lceil\frac{n}{2}\right\rceil - 1\right)$$

$$\log K \geq (H-1)\log\left\lceil\frac{n}{2}\right\rceil + \log 2 + \log\left(\frac{\left\lceil\frac{n}{2}\right\rceil - 1}{\left\lceil\frac{n}{2}\right\rceil}\right) \geq (H-1)\log\left\lceil\frac{n}{2}\right\rceil$$

$$\log_{\left\lceil\frac{n}{2}\right\rceil} K + 1 \geq H$$

$$\therefore \left\lceil \log_{\left\lceil\frac{n}{2}\right\rceil} K \right\rceil \geq H$$

If there are K search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil} K \rceil$, thus searches can be conducted efficiently.

Maximum Depth of A B^+-Tree

https://cs.stackexchange.com/questions/82015/maximum-depth-of-a-b-tree

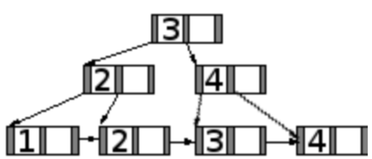
Maximum depth of a B+ tree

Asked 4 years, 7 months ago Modified 4 years, 7 months ago Viewed 9k times

1

Given K... `# key values`, n... `# pointers in a node`.

I read somewhere, that the **maximum** depth is defined as $\lceil \log_{\lceil \frac{n}{2} \rceil}(K) \rceil$. However, it is not correct, as I can come up with a counterexample. When the tree is minimum filled, it won't work. E.g.:



This is a valid B^+-tree, the root has at least two childs, each inner node has at least $\lceil n/2 \rceil$ childs and each leaf has at least $\lceil \frac{n-1}{2} \rceil$ record. So, $n = 3$ and $K = 4$, then $\log_2(4) = 2$. Now, when you fill up the leafs: [1,1,2,2,3,3,4,4], then it is again a valid tree and $K = 8$, hence $\log_2(8) = 3$, but same depth.

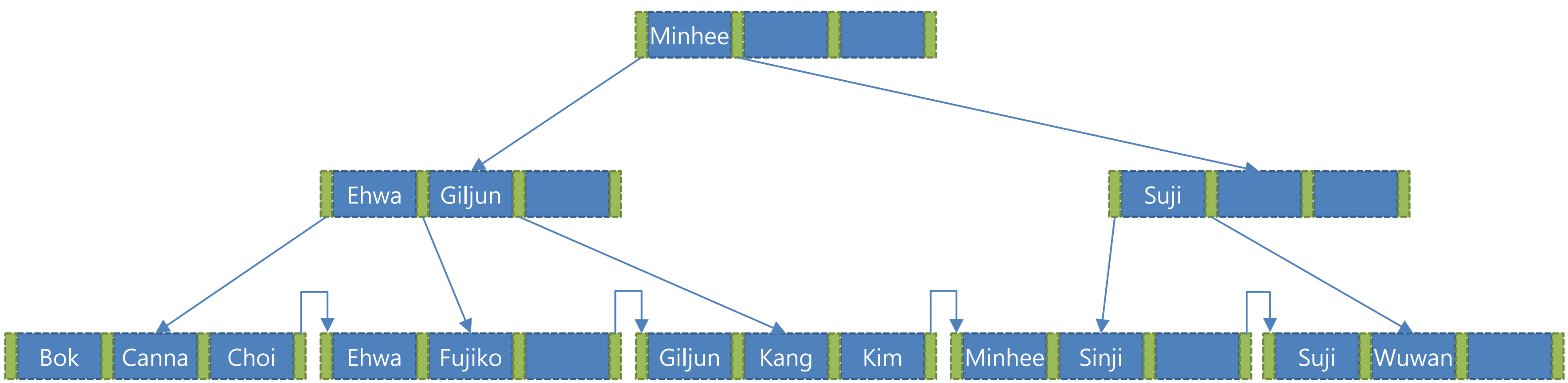
Notice: I am looking for a formula or explanation but for a B^+-tree **not** a B-tree. A source would be nice.

Queries on B$^+$-Trees

Find record with search-key query *V*.

1. *C=root*
2. While C is not a leaf node {
   1. Let *i* be least value s.t. $V \leq K_i$.
   2. If no such exists, set *C* = *last non-null pointer in C*
   3. Else { if (V = K_i) Set C = P_{i+1} else set $C = P_i$}

   }
3. Let *i* be least value s.t. $K_i = V$
4. If there is such a value *i*, follow pointer P_i to the desired record.
5. Else no record with search-key value *k* exists.



Queries on B+-Trees (Cont.)

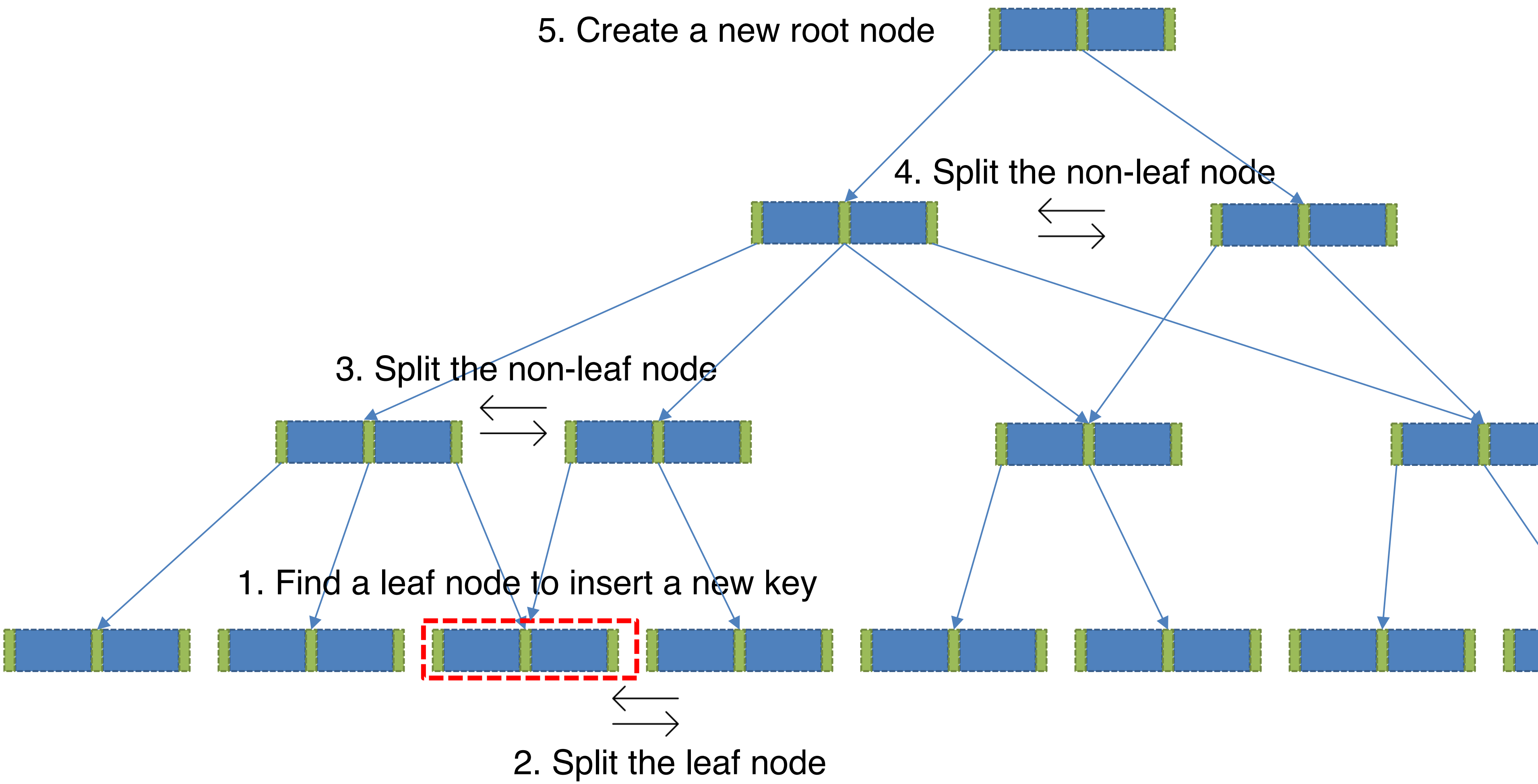
- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil} K \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and n is typically around 200 (16 bytes per index entry).
- With 1 million search key values and n = 100
  - at most $\lceil \log_{100} 1M \rceil$ = 3 nodes are accessed in a lookup.
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

INSERTION

Updates on B^+-Trees: Insertion

1. Find the leaf node in which the search-key value would appear
2. If the search-key value is already present in the leaf node
   1. Act properly depending on application
3. If the search-key value is not present, then
   1. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
   2. Otherwise, split the node (along with the new (key-value, pointer) entry) as discussed in the next slide.

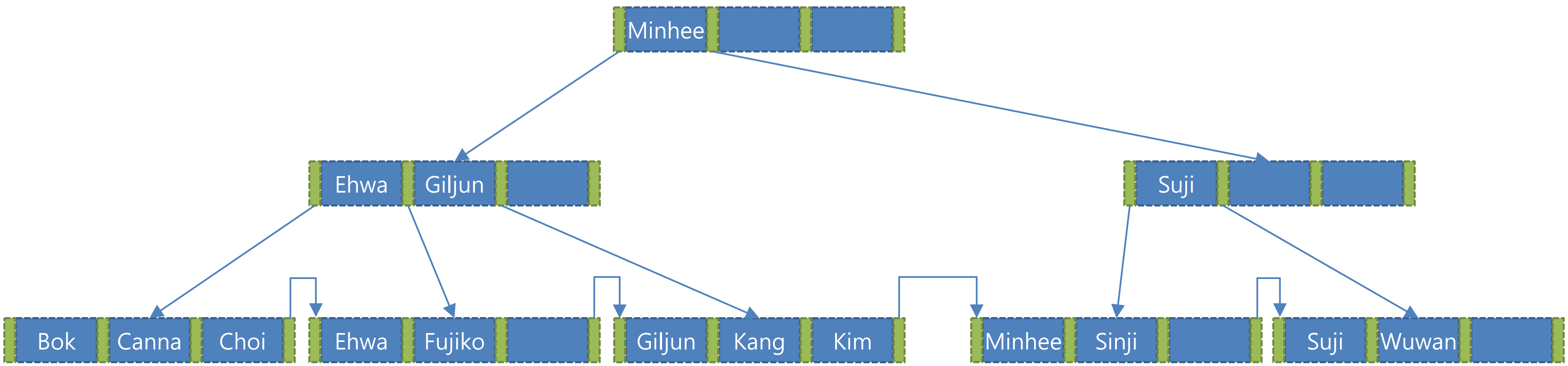
Recursive Propagation of Node Split



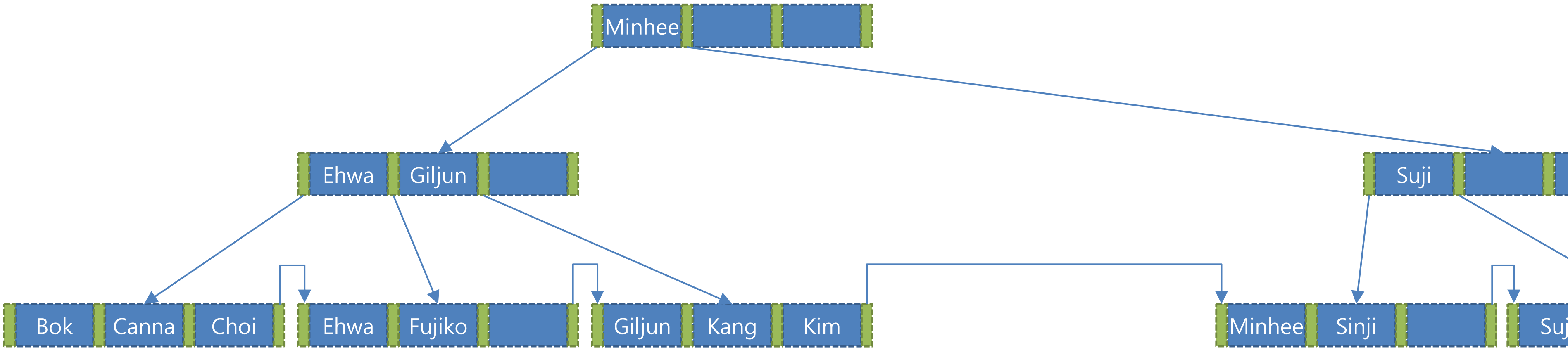
Updates on B^+-Trees: Insertion

- **Splitting a leaf node**:
  - Take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first ⌈n/2⌉ in the original node, and the rest in a new node.
  - Let the new node be *p,* and let *k* be the least(i.e., first) key value in *p.* Let the parent of original node being split be *q.*
  - *Set the last pointer of p to be the original one's last pointer, and the original one's last pointer to q*
  - Call Insert (*q,k,p*) to insert (k, p) into q.
- Splitting of nodes proceeds upwards till a node that is not full is found.
  - If the parent q is full, split it and **propagate** the split further up.
  - In the worst case the root node may be split increasing the height of the tree by 1.

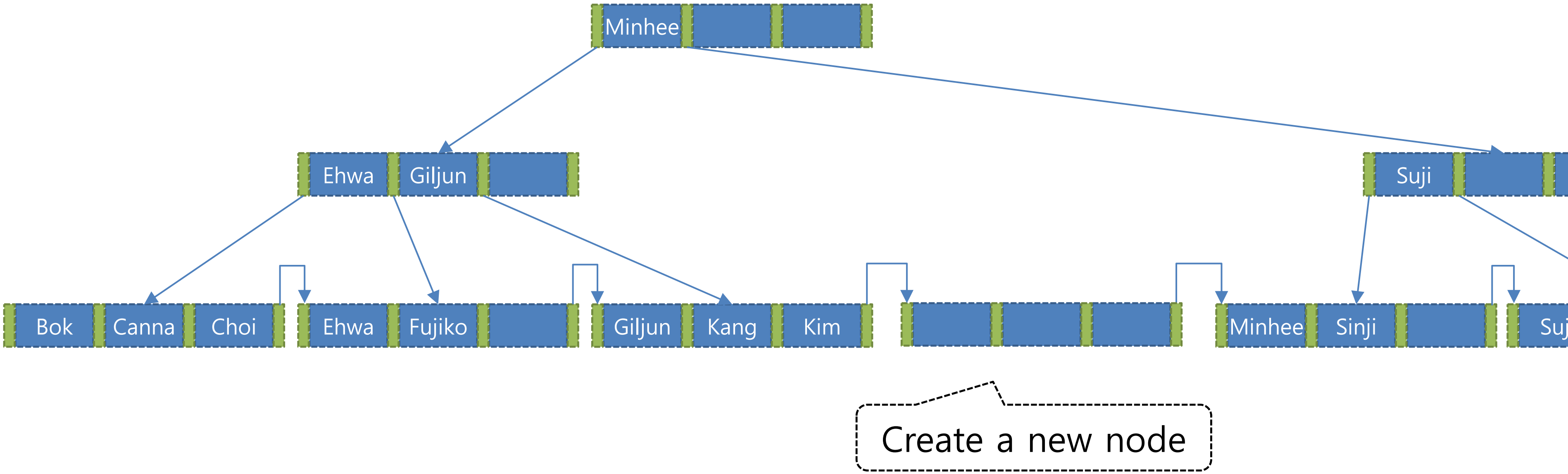
B+-Tree Insertion: (Giyeon, p)



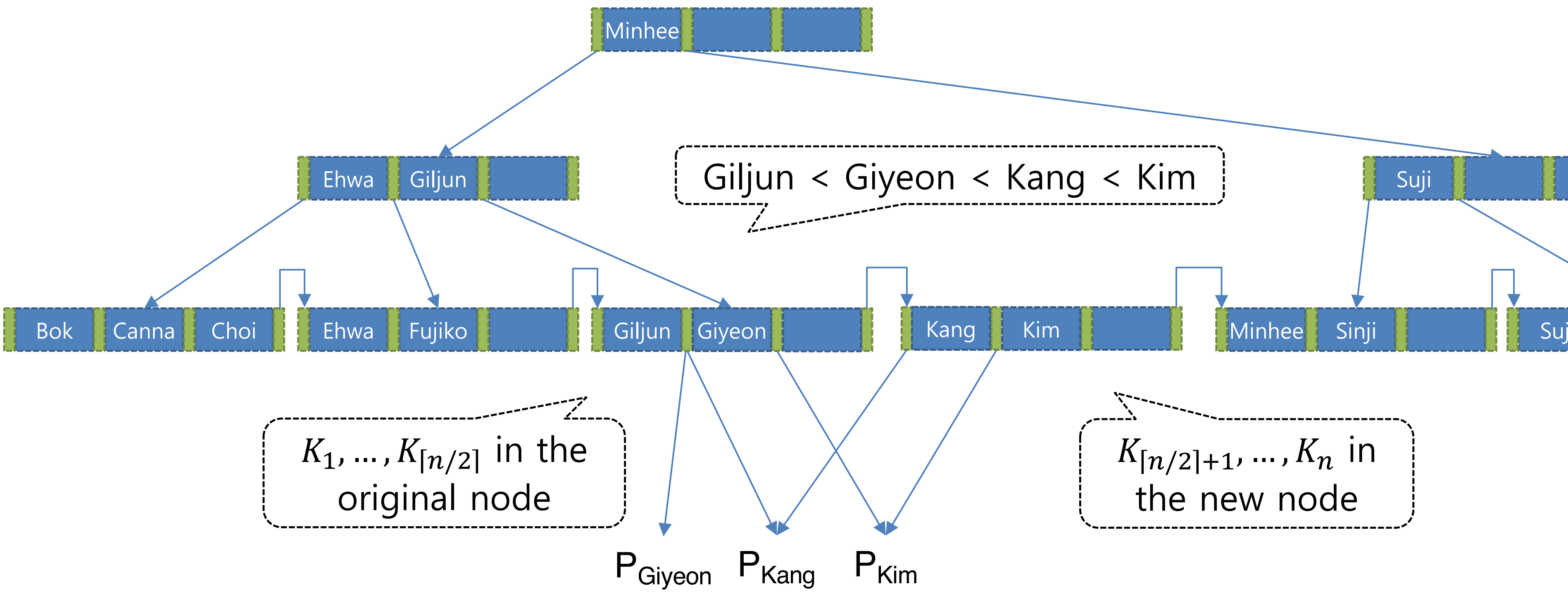
B+-Tree Insertion: Giyeon



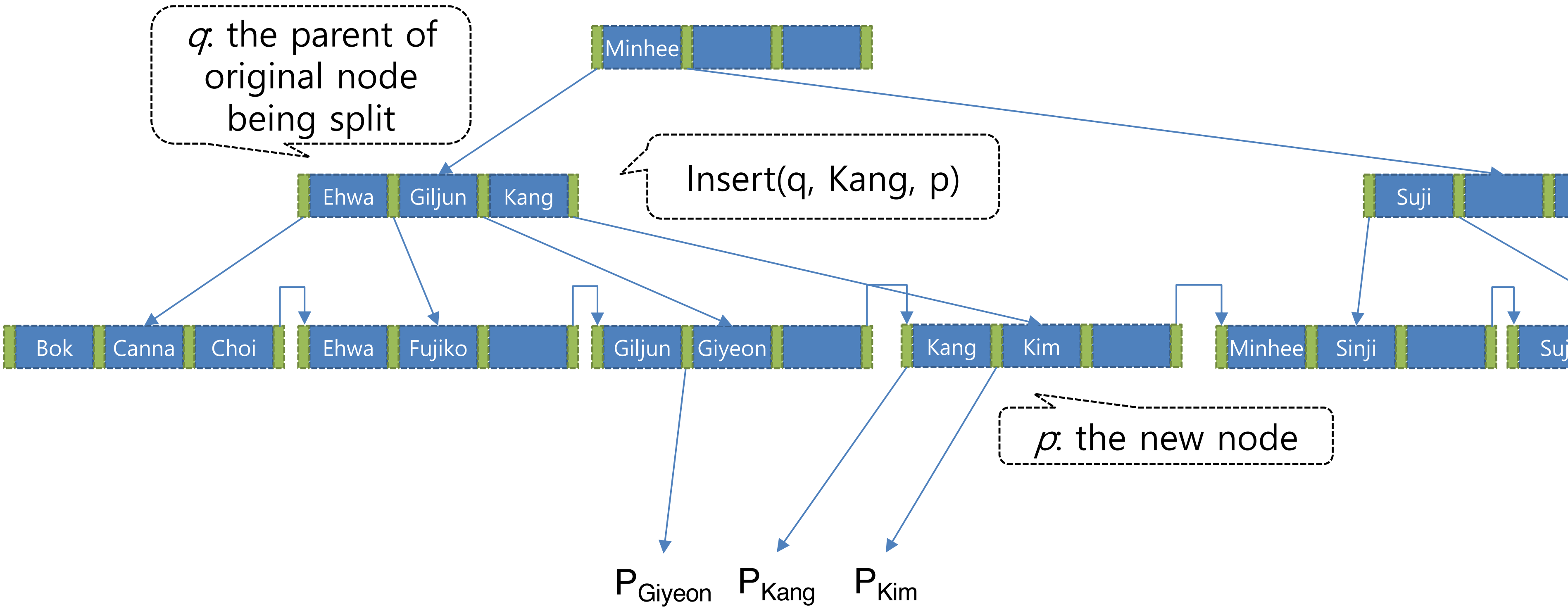
B+-Tree Insertion: Giyeon



B^+-Tree Insertion: Giyeon



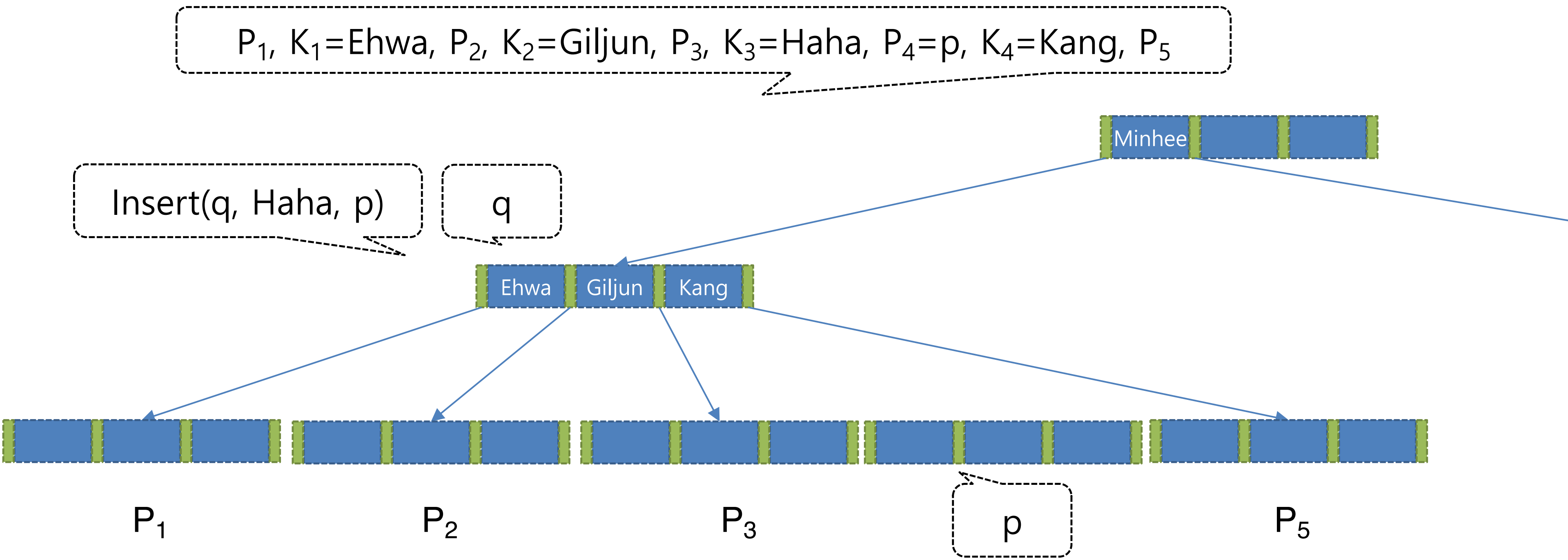
B$^+$-Tree Insertion: Giyeon



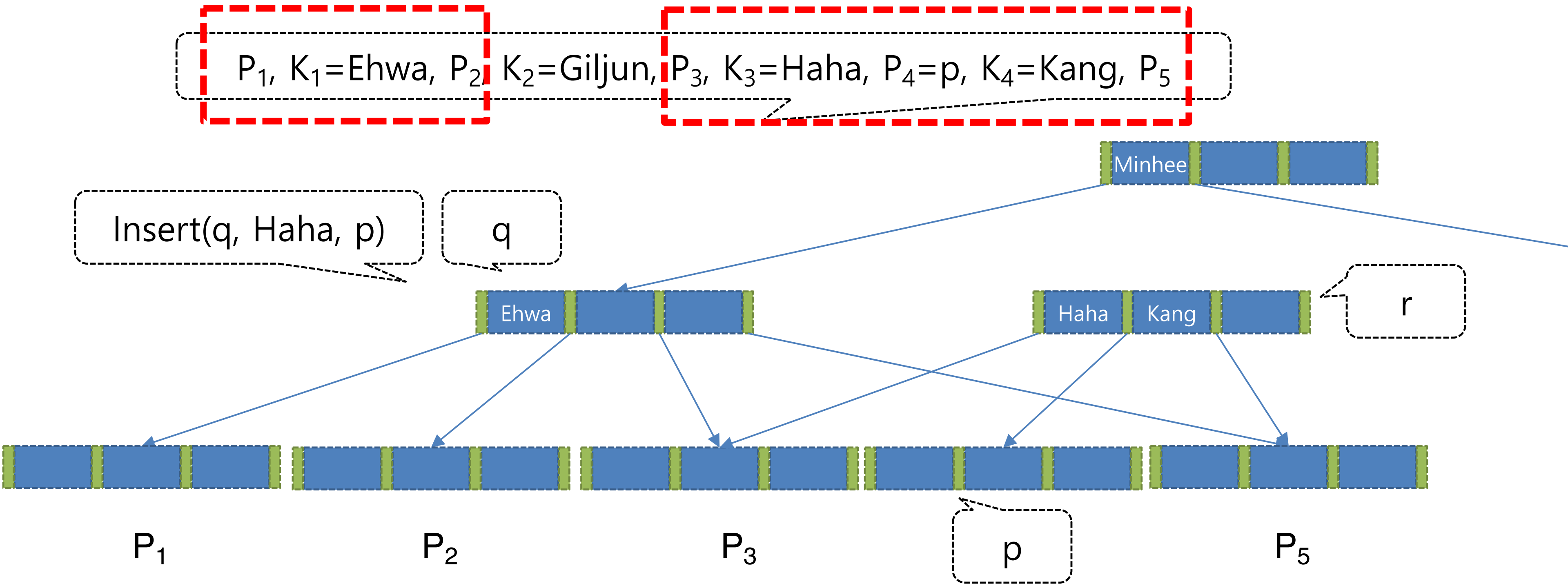
Insertion in B+-Trees (Cont.)

- **Splitting a non-leaf node**: when inserting (q,k,p) into an already full non-leaf node q
  - Let $P_1, K_1, P_2, K_2, \ldots, K_n, P_{n+1}$ be the search-keys and pointers after k and p is inserted into q, where $K_i = k$ and $P_{i+1} = p$ such that $K_{i-1} < K_i = k < K_{i+1}$
  - Copy $P_1, K_1, \ldots, K_{\lceil \frac{n}{2} \rceil - 1}, P_{\lceil \frac{n}{2} \rceil}$ into node q
  - Copy, $P_{\lceil \frac{n}{2} \rceil + 1}, \mathrm{K}_{\lceil \frac{n}{2} \rceil + 1}, \ldots, K_n, P_{n+1}$ into **a newly allocated node r**
  - If the split node is a root node,
    - Create a new root node and set its (P_1, K_1, P_2) to (q, $K_{\lceil \frac{n}{2} \rceil}$, r)
  - Otherwise, call insert(s, $K_{\lceil \frac{n}{2} \rceil}, r$) to insert ($K_{\lceil \frac{n}{2} \rceil}, r$) into s (= the parent of q)

B^+-Tree Insertion into Non-leaf



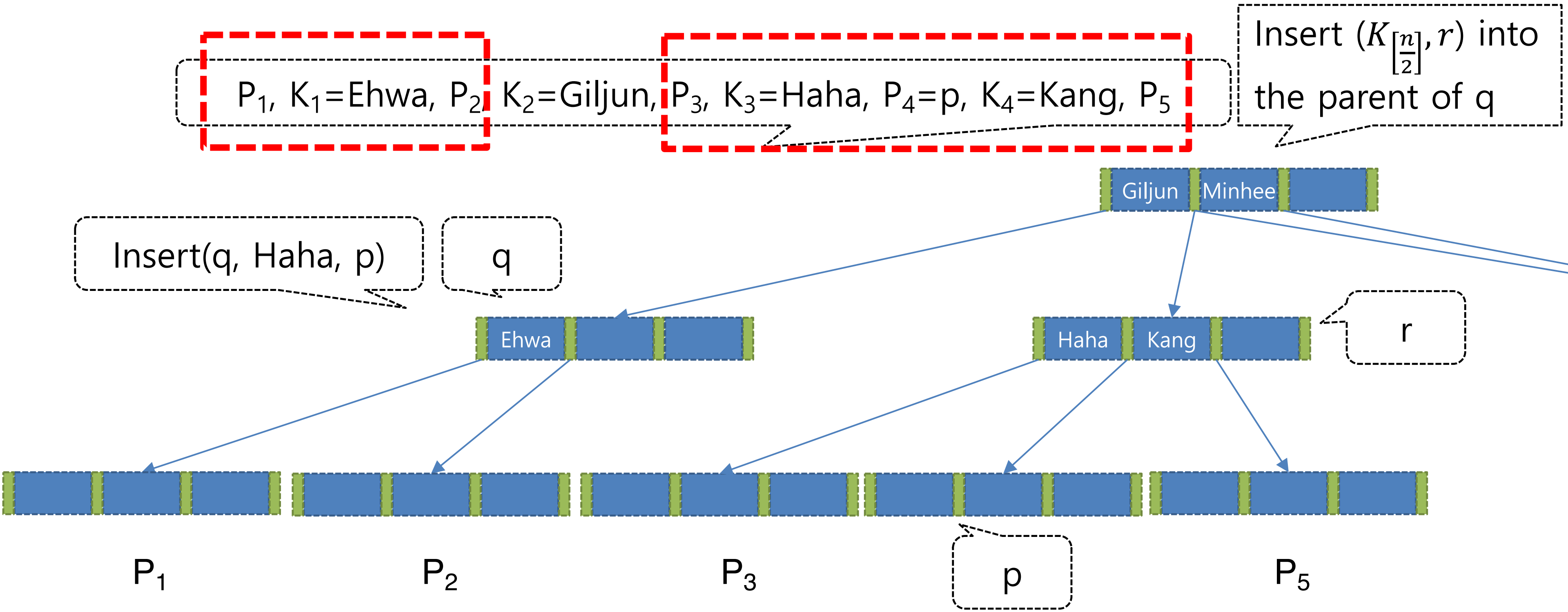
B+-Tree Insertion into Non-leaf



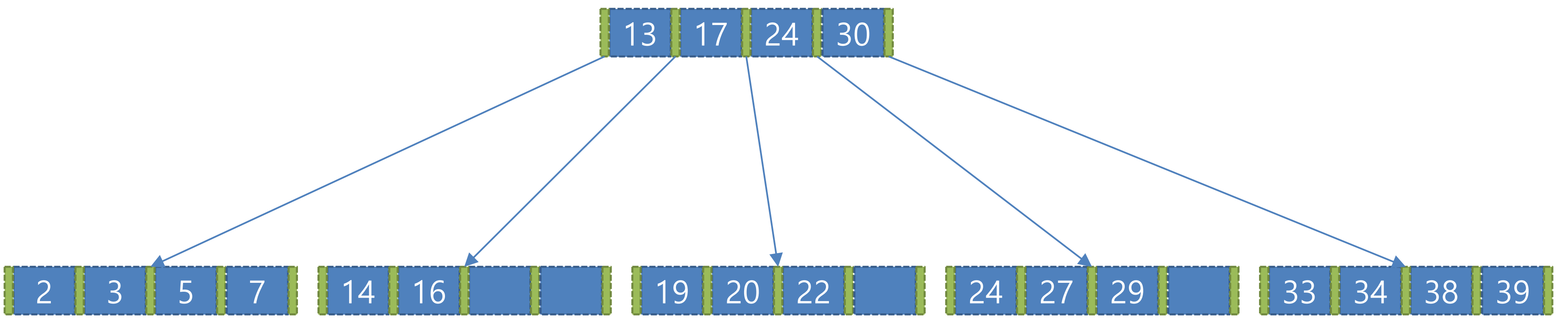
$P_1, K_1, \ldots, K_{\lceil \frac{n}{2} \rceil - 1}, P_{\lceil \frac{n}{2} \rceil}$ into node q

$P_{\lceil \frac{n}{2} \rceil + 1}, K_{\lceil \frac{n}{2} \rceil + 1}, \ldots, K_n, P_{n+1}$ into **a newly allocated node r**

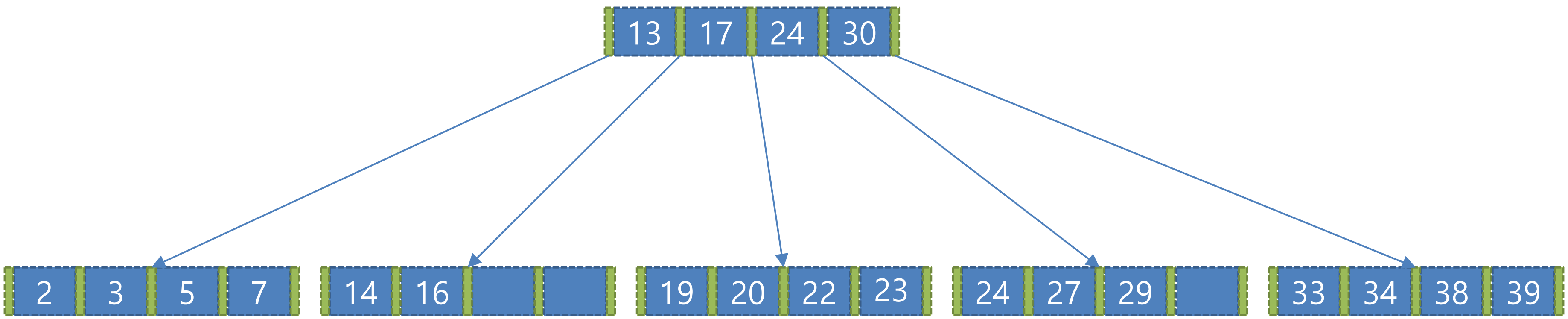
B^+-Tree Insertion into Non-leaf



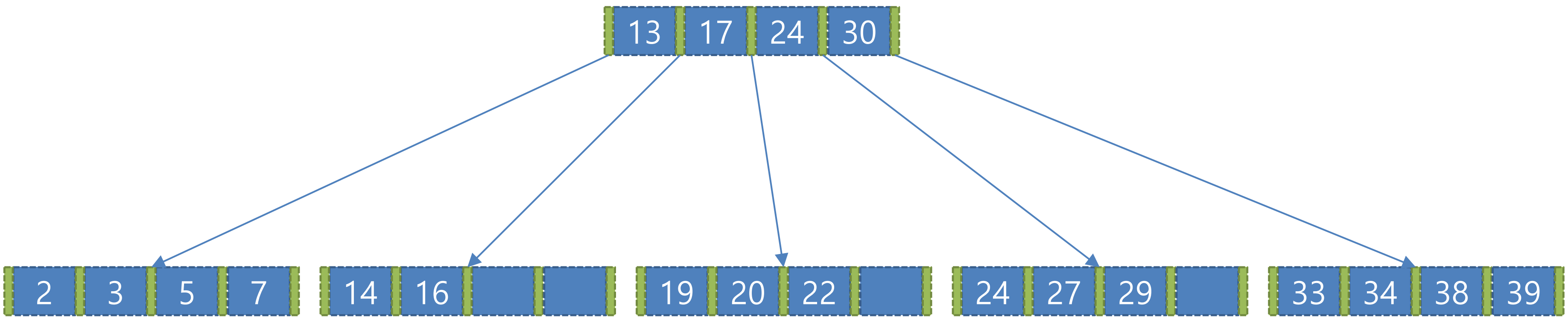
Exercise: Insert 23



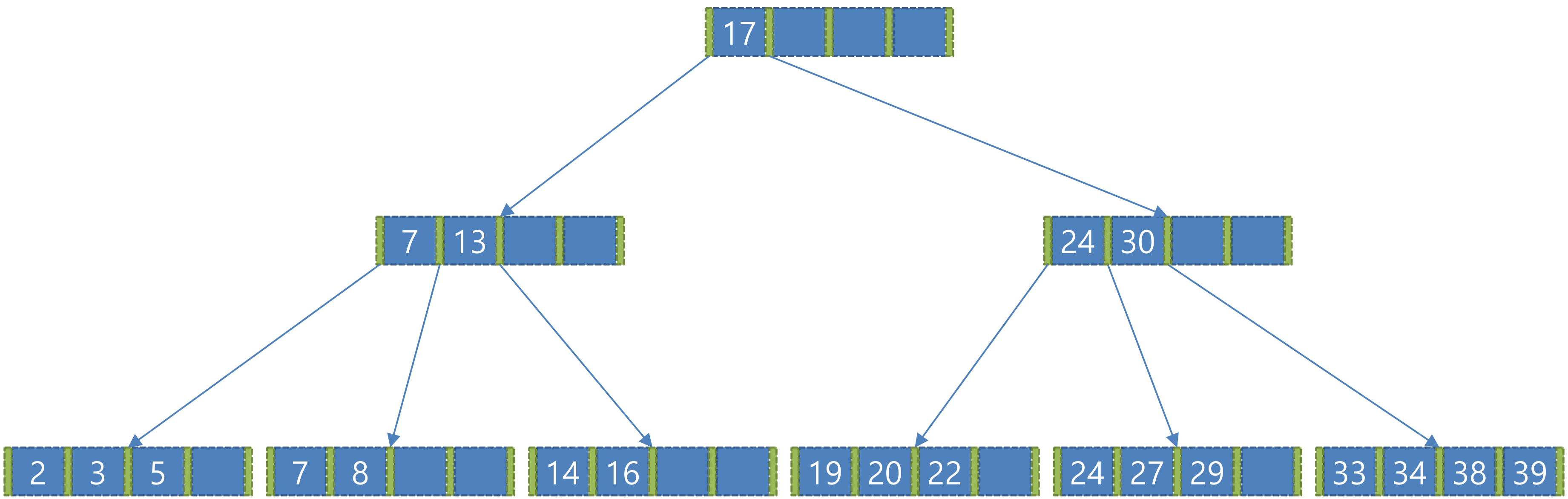
Answer: Insert 23



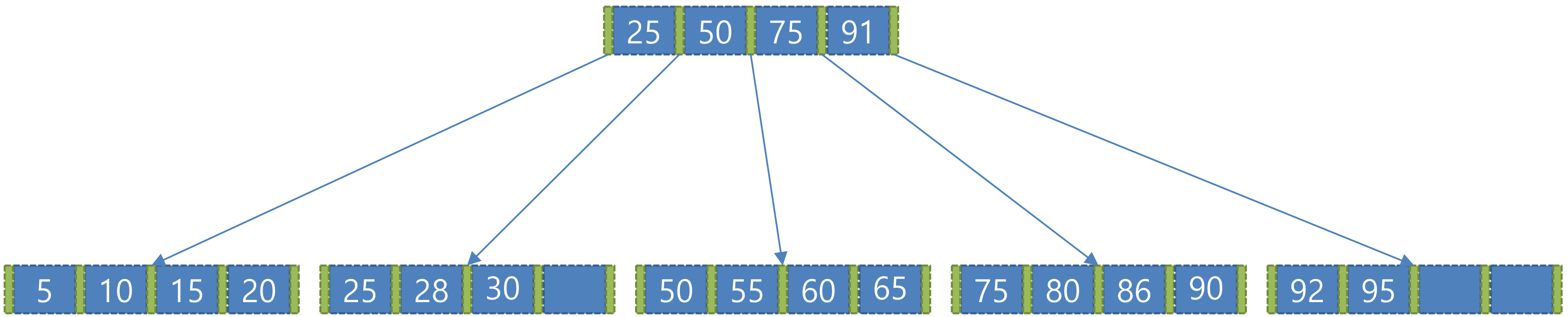
Exercise: Insert 8



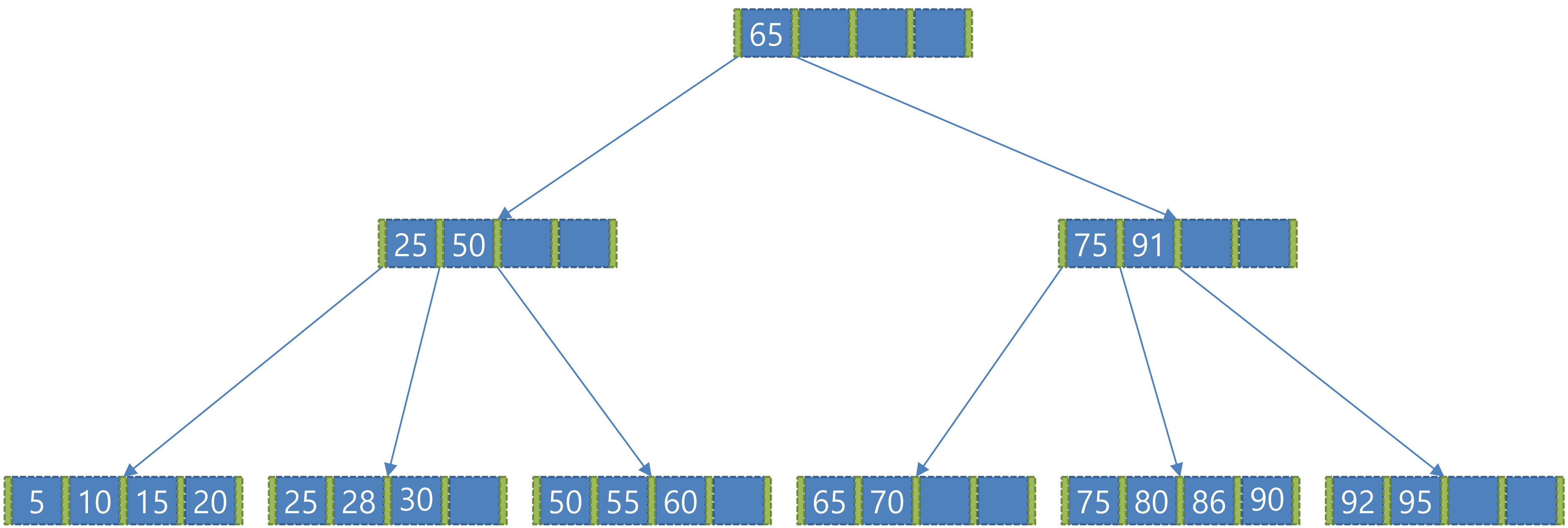
Answer: Insert 8



Exercise: Inserting 70



Answer: Inserting 70

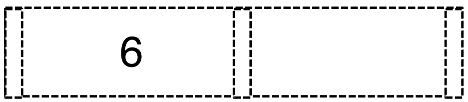


Inserting into a B+ Tree

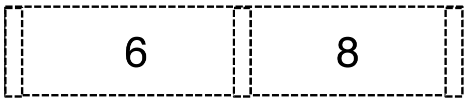
- Example:
  - Suppose we had a B+ tree with n = 3
    - 2 keys max. at each internal node
    - 3 pointers max. at each internal node

Inserting Into B+ Trees (cont.)

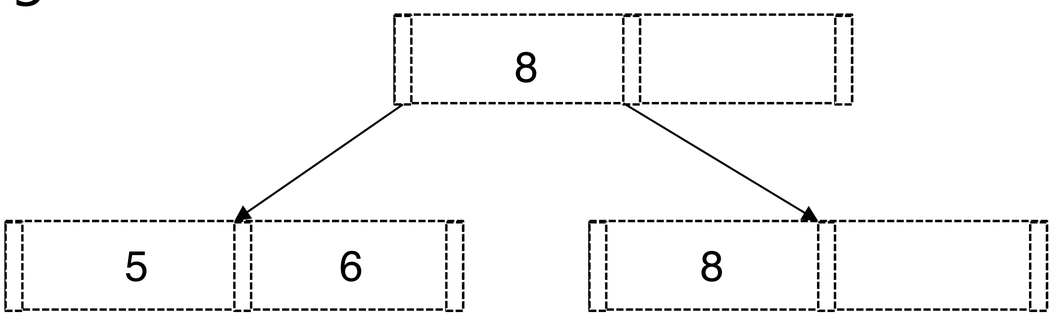
- Case 1: empty root
  - Insert 6



  - Insert 8

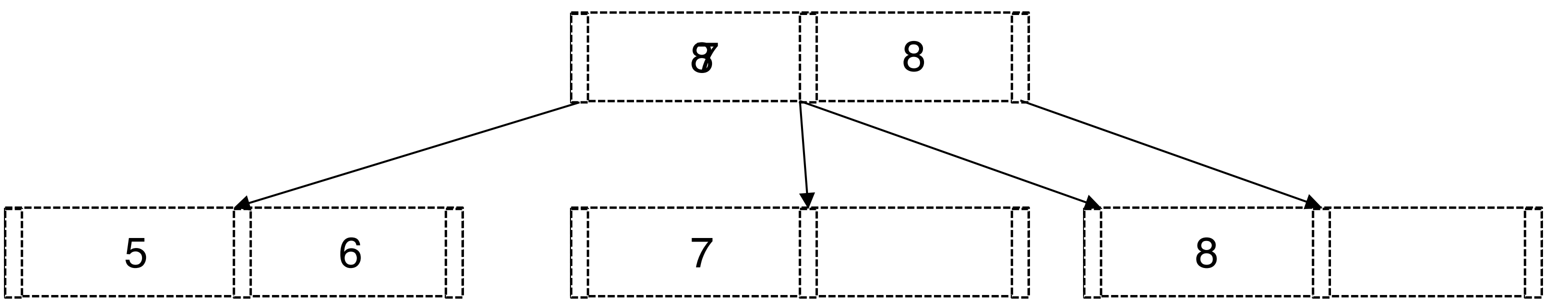


- Case 2: full root
  - Insert 5



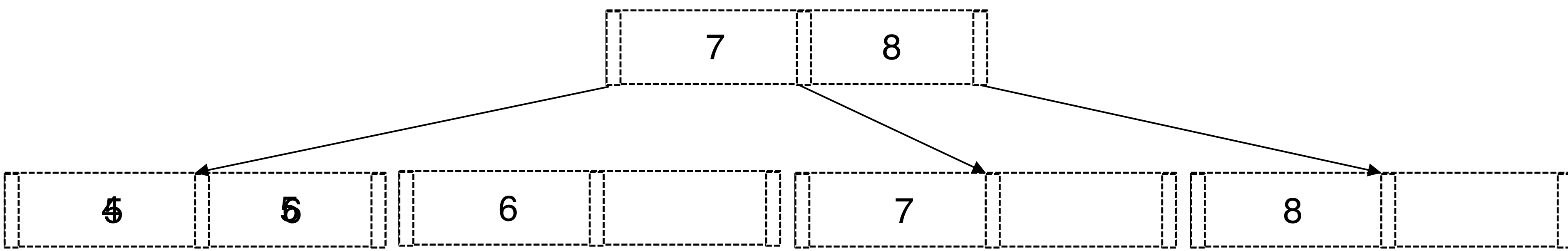
Inserting into B+ Trees (cont.)

- Case 3: Adding to a full node
  - Insert 7 into our tree:



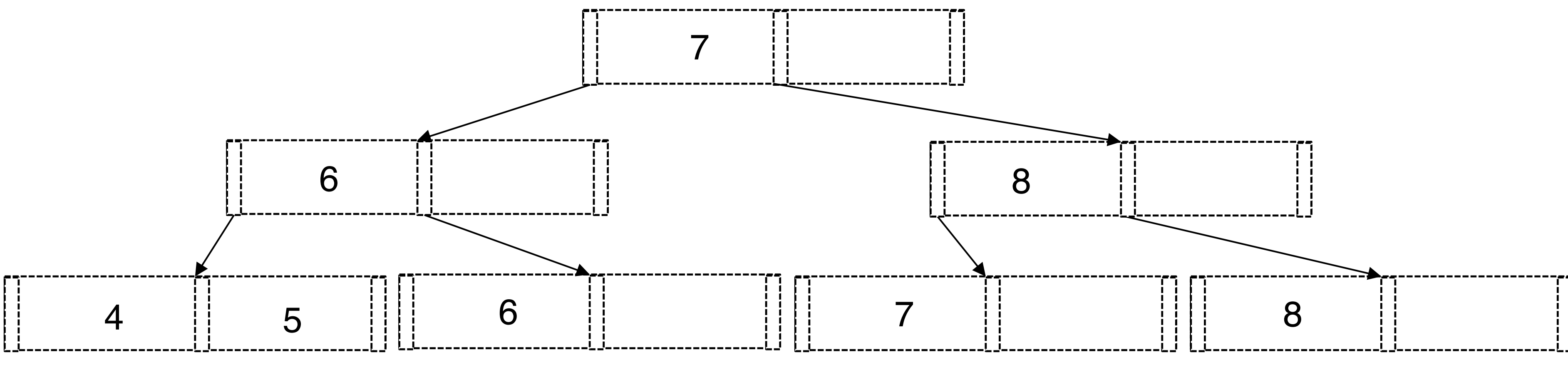
Inserting into B+ Trees (cont.)

- Case 4: Inserting on a full leaf, requiring a split at least 1 level up
  - Insert 4.



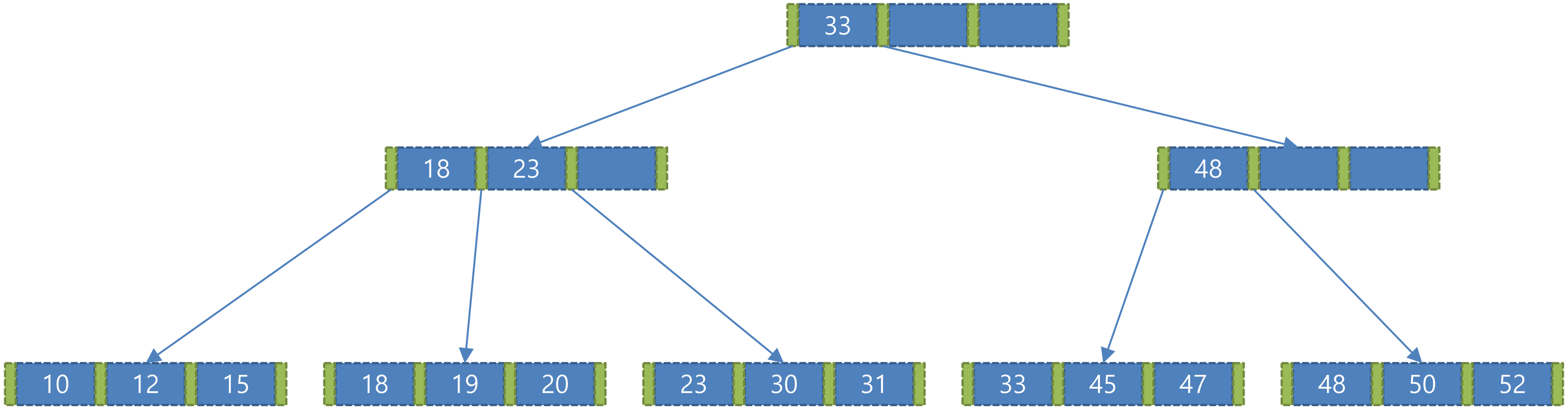
Inserting into B+ Trees (cont.)

- Case 4: Inserting on a full leaf, requiring a split at least 1 level up
  - Insert 4.

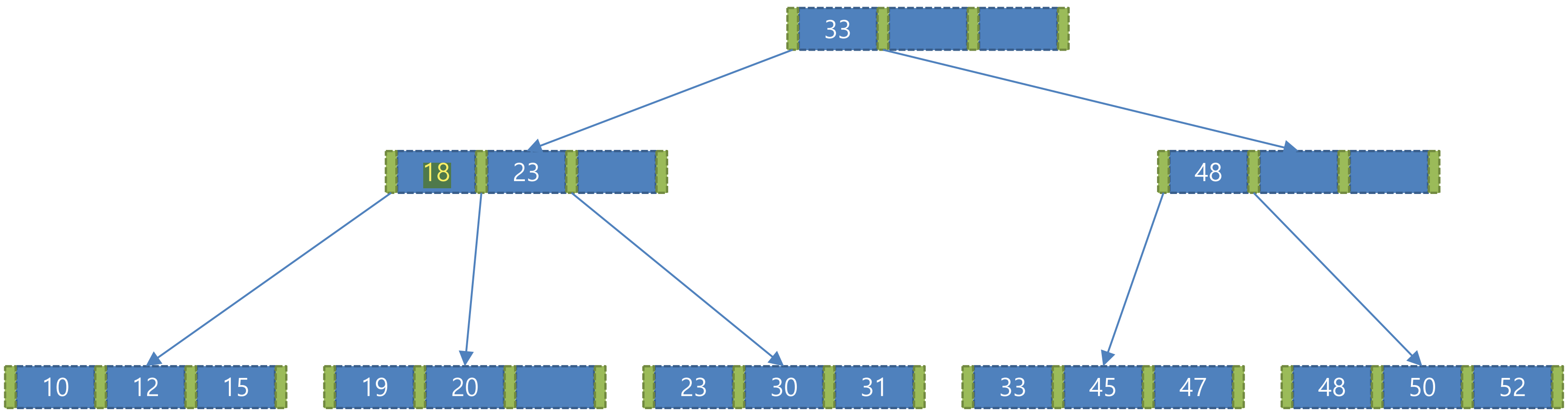


DELETION

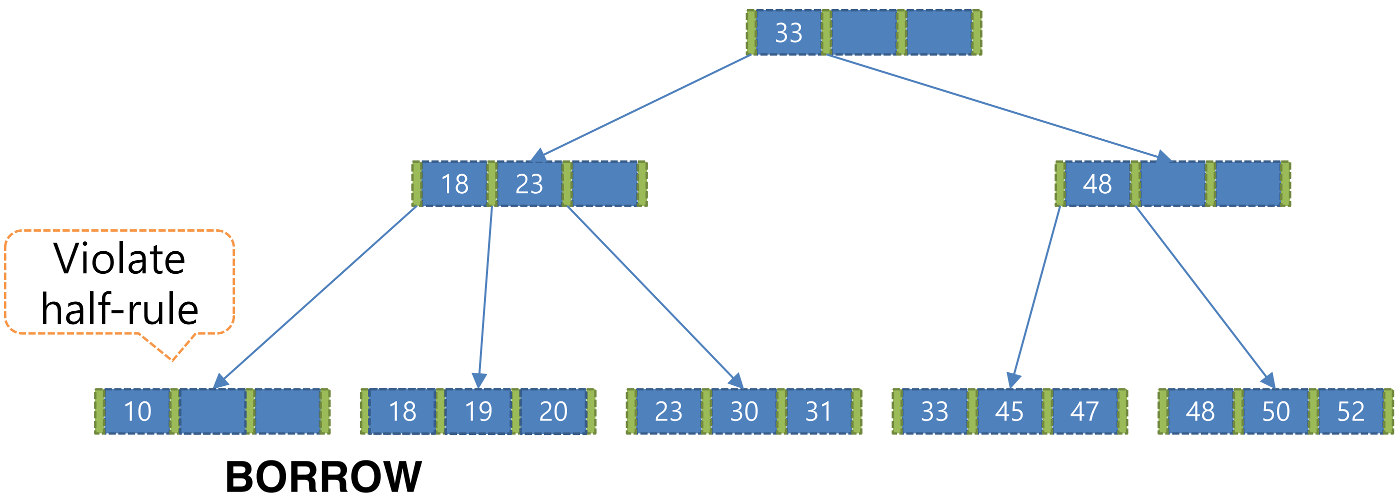
B+-Tree Deletion (1) [Delete 18] :
Leaf node has enough keys



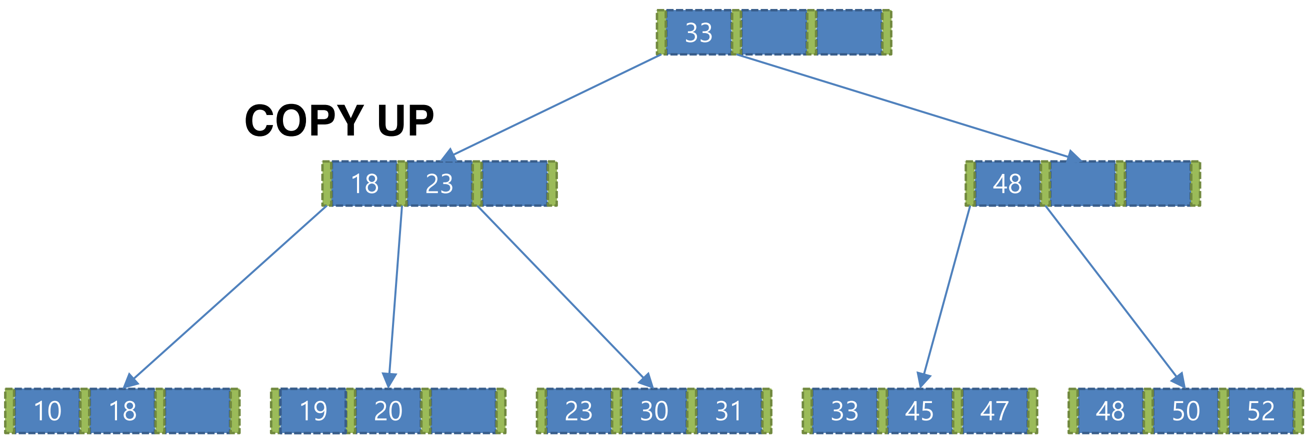
B+-Tree Deletion (1) [Delete 18] : Leaf node has enough keys



B+-Tree Deletion (2) [Delete 12]: Re-distribution in Leaf Nodes

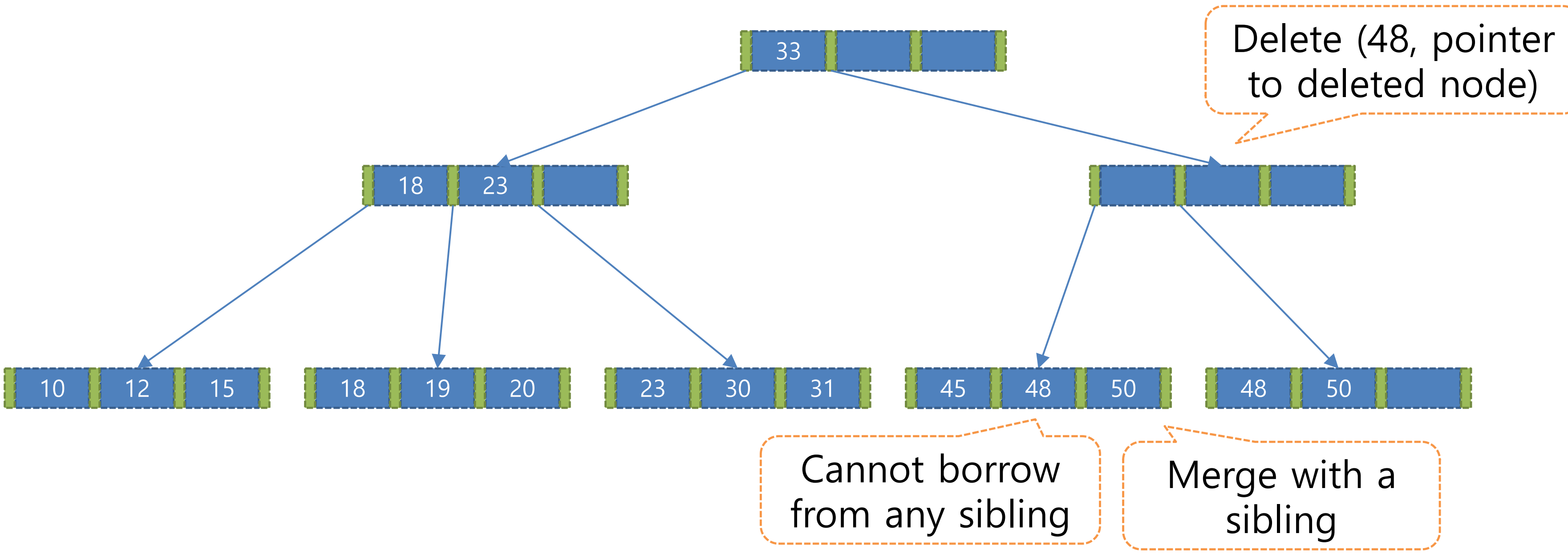


B+-Tree Deletion (2) [Delete 12]: Re-distribution in Leaf Nodes

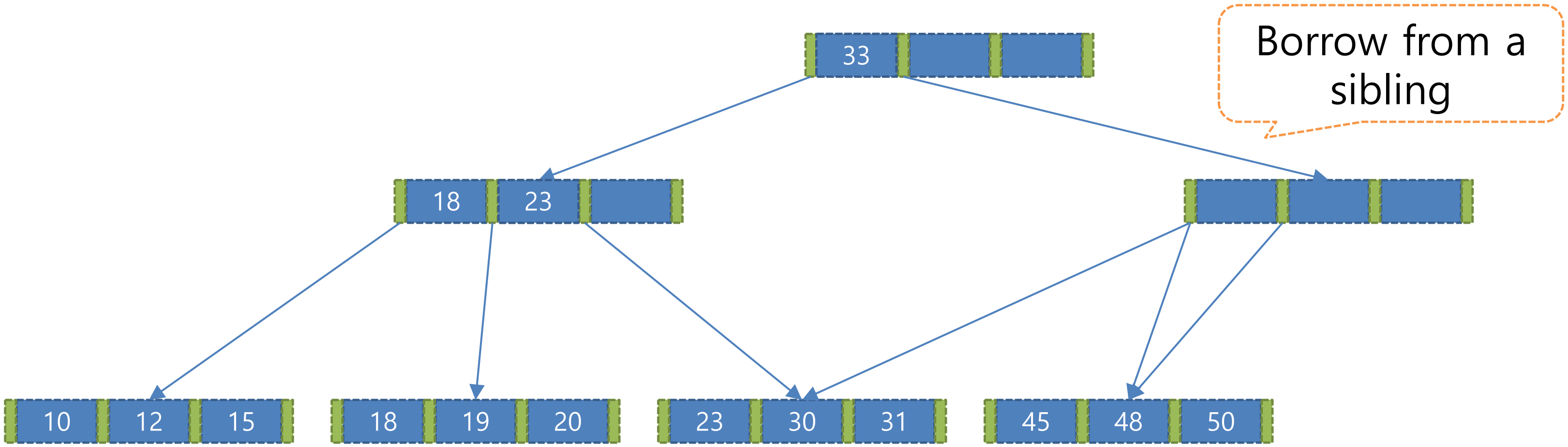


B+-Tree Deletion (3) [Delete 33]:

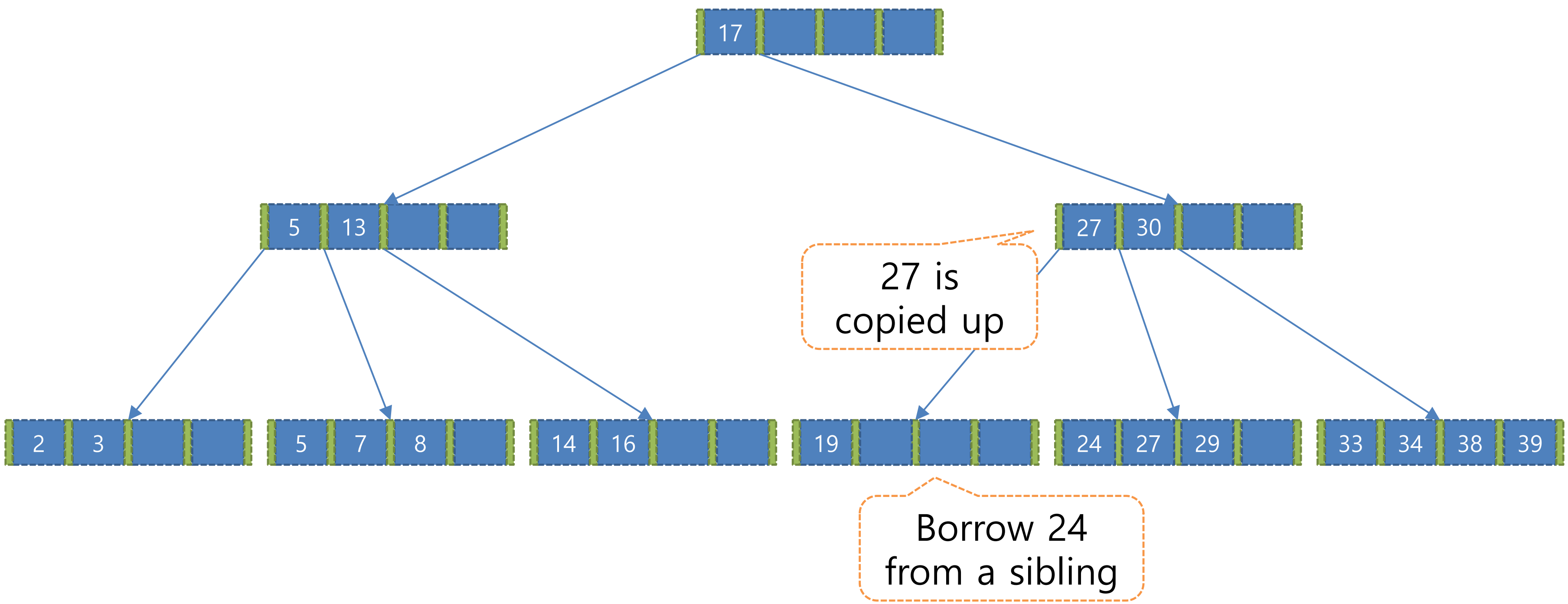
Merge in Leaf Nodes and Re-distribution in Non-leaf Nodes



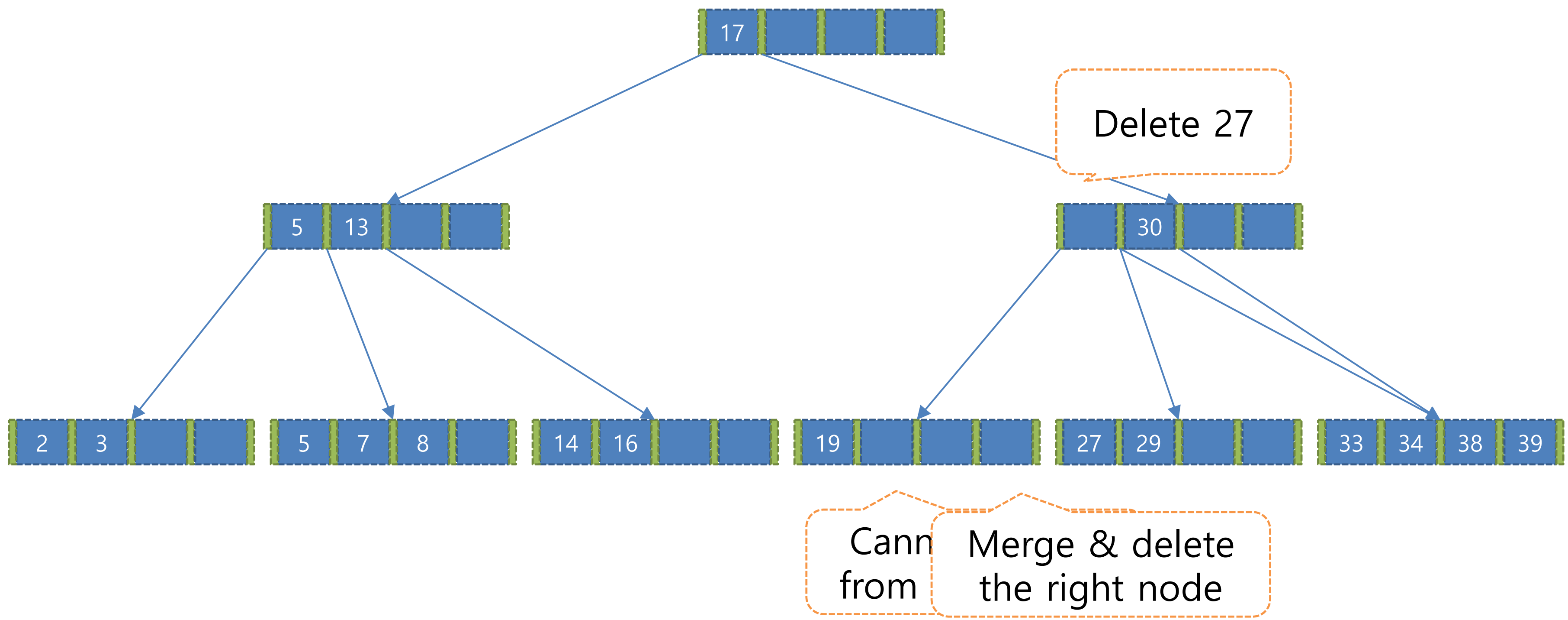
B+-Tree Deletion (3) [Delete 33]:
Merge in Leaf Nodes and Re-distribution in Non-leaf Nodes



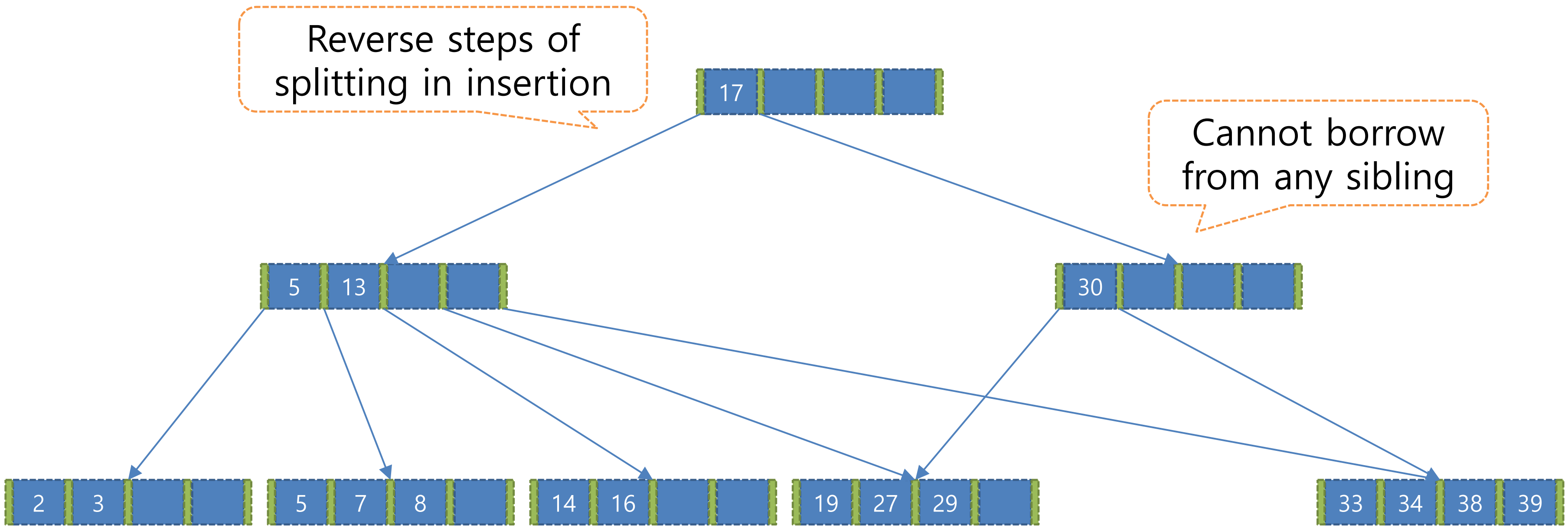
More Examples: Delete 20



More Examples: Delete 24



More Examples: Delete 24

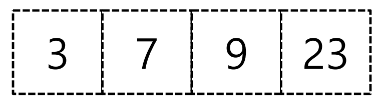


Exercise

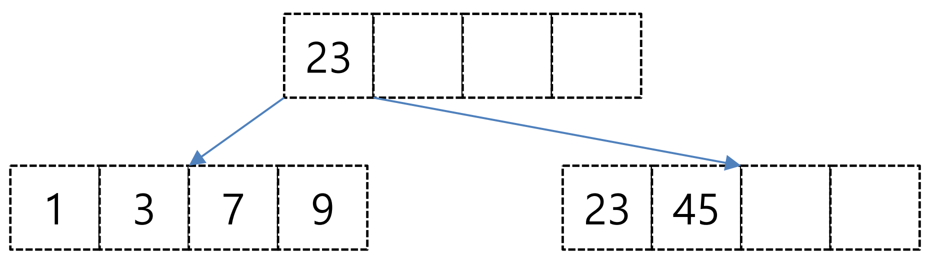
- Build a B^+-tree of fan-out 5 created by these data:
  - 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56
- Add these further keys: 2, 6, 12
- Delete these keys: 4, 5, 7, 3, 14

Solution

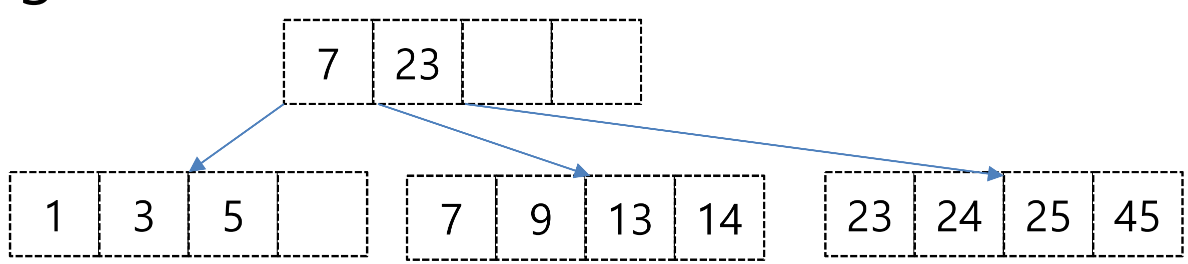
- Adding 3, 7, 9, 23



- Adding 45, 1

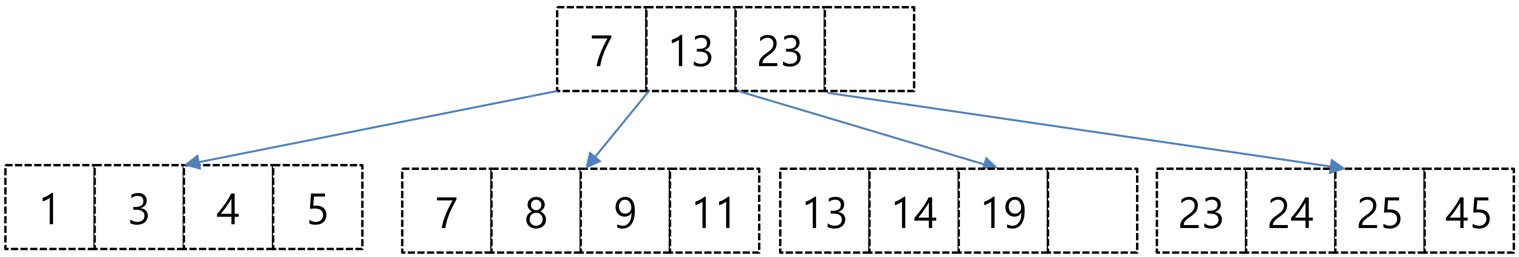


- Adding 5, 14, 25, 24, 13

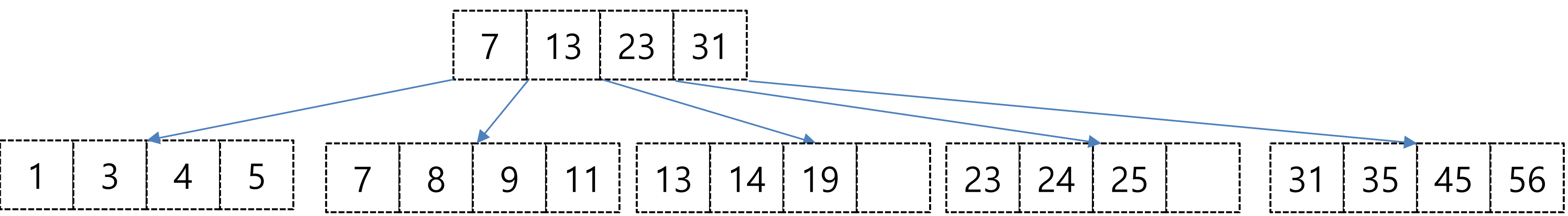


Solution

- Adding 11, 8, 19, 4

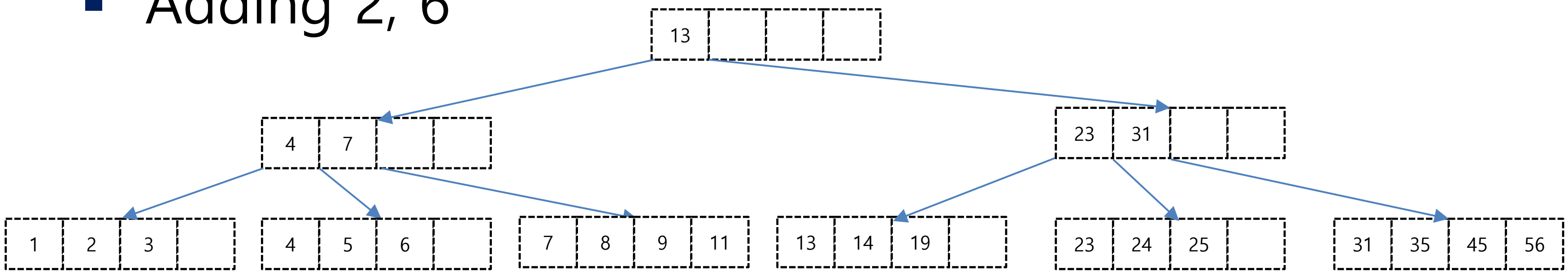


- Adding 31, 35, 56

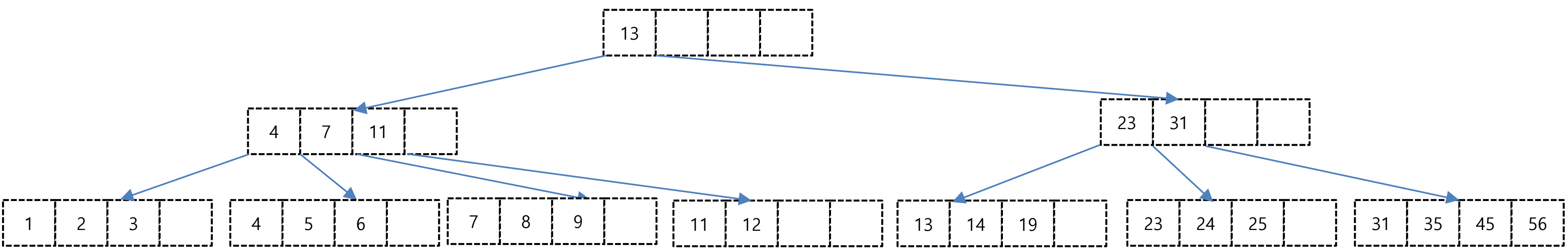


Solution

- Adding 2, 6

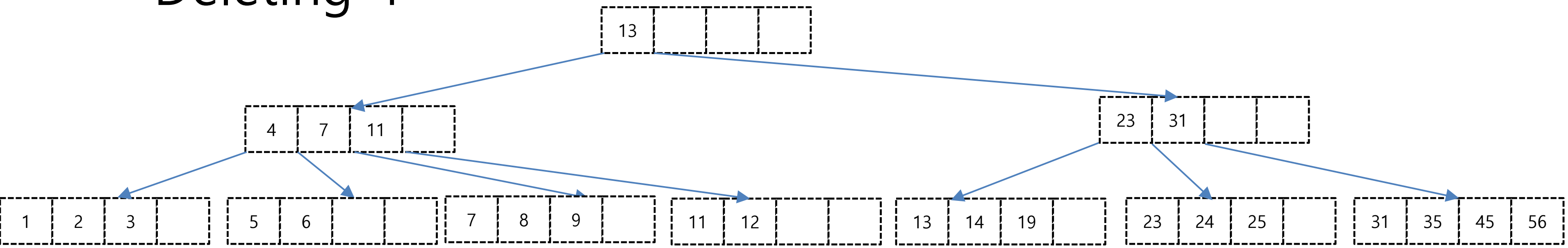


- Adding 12

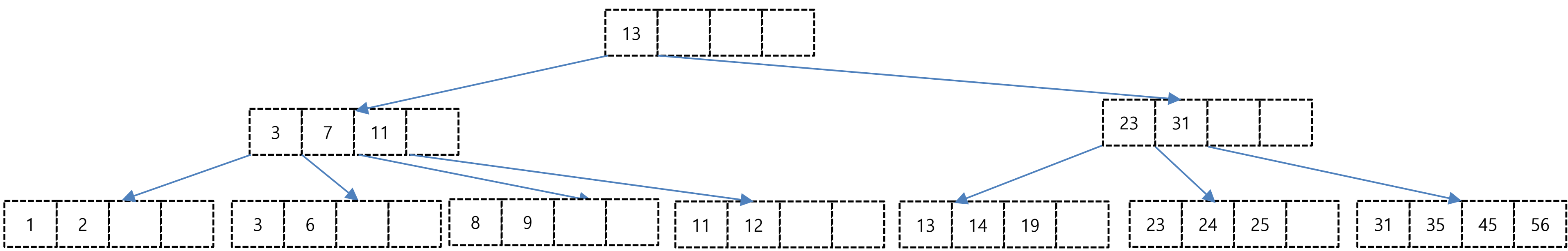


Solution

- Deleting 4



- Deleting 5, 7



Solution

- Deleting 3, 14