Lecture 10

Indexing

COMP3278A

Introduction to Database Management Systems

Dr. Ping Luo

Email : pluo@cs.hku.hk

Department of Computer Science, The University of Hong Kong

Acknowledgement: Dr Chui Chun Kit

In this chapter…

- Outcome 1. **Information Modeling**
  - Able to understand the modeling of real life information in a database system.
- Outcome 2. **Query Languages**
  - Able to understand and use the languages designed for data access.
- Outcome 3. **System Design**
  - Able to understand the design of an **efficient** and reliable database system.
- Outcome 4. **Application Development**
  - Able to implement a practical application on a real database.

We are going to learn…

- Basic concepts
- B^+ -tree

Section 1

Basic Concepts

Basic concepts

- **Index is used to speed up access to desired data.**
  - E.g., Author catalog in library, phone directory index, etc.
- **Search key**
  - An **attribute** or a **set of attributes** used to look up records in a file.
- **Indices are typically much smaller than the original file.**

Primary v.s. secondary

- **Primary index - An index whose search key also defines the sequential order of the file.**
  - E.g., Access staff records through **staffID** (**primary search key**).
- **However, the data file can be sorted in one order only.**
- **How about accessing data with a different search key?**
  - E.g., Access staff records through **roomID** (**a secondary search key, need a secondary index!**).

staffID	roomID	faculty
10101	49	C.S.
12121	42	Finance
15151	35	Music
22222	10	Physics
32343	15	History
33456	18	C.S.
45565	20	E.E.E.
58583	3	Biology
76543	31	Finance
76766	5	Finance
83821	2	C.S.
98345	24	C.S.

Index evaluation factors

- **Each indexing technique must be evaluated on the basis of these factors**
  - **Access types** – The types of access that are supported efficiently (e.g., equality search or range search? Single attribute search or multi-attribute search?)
  - **Access time** – The time it takes to find a particular data item, or a set of items.
  - **Insertion / deletion time**
  - **Space overhead**

No one indexing technique is the best. Rather, each technique is best suited to particular database applications.

Section 2

B^+-tree

Properties of B$^+$-tree

- B$^+$-tree index structure is one of the most widely used index structure in DBMS.
- All paths from root to leaf are of the same length (i.e., balanced)
- Can support efficient processing of the following queries (Assume that the B$^+$-tree is built on attribute A of the relation R):

```
SELECT * FROM R WHERE R.A = 3
```

```
SELECT * FROM R WHERE R.A >= 3 AND R.A < 22
```

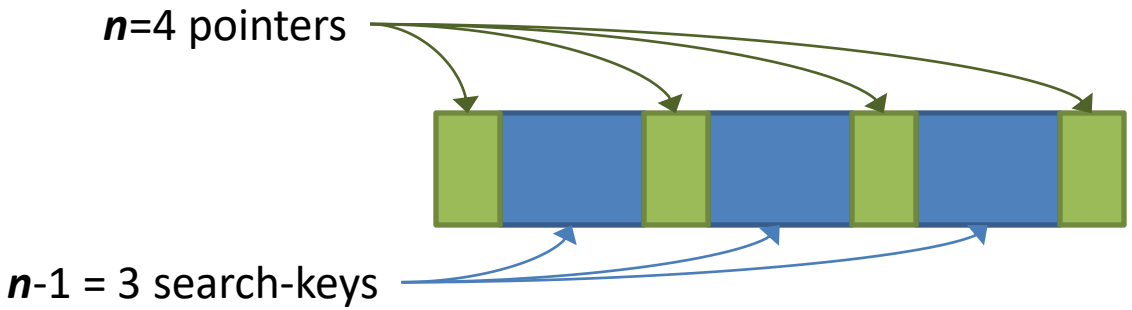
Why not binary search tree?

Balanced Binary Search tree minimizes the number of key comparisons for finding a search key. Why don't we use balanced binary search tree in Database?

- **We need a tree which is**
  - **Node size = 1 block**
    (A node can contain more than one keys)
  - Low in height
  - Balanced

Because we want to **minimize the number of block retrieval** in answering a query (i.e., number of tree nodes to be accessed) rather than the number of key comparisons.

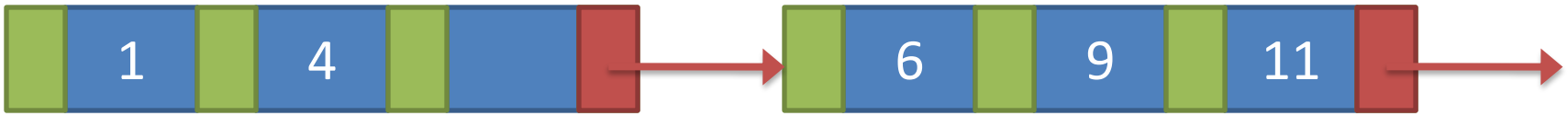
A node in B$^+$-tree



- A node contains up to ***n***-1 **search-key values**, and ***n*** **pointers**.

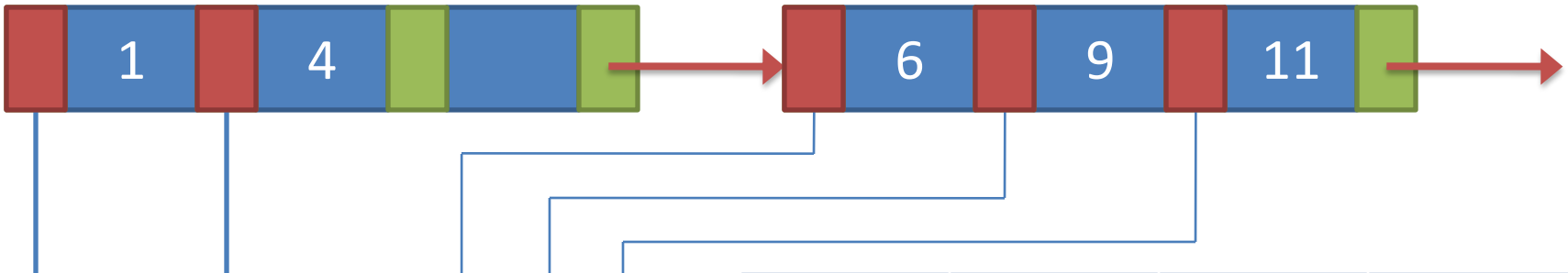
- The search-key values within a node are kept in **sorted order**.

1. Leaf node



- A leaf node has **at least $\lceil (n-1)/2 \rceil$** and at most (n – 1) **values, where n is the number of pointers**.
  - E.g., with n = 4, a leaf node must contain **at least 2 values**, and at most 3 values.
- The **last pointer** is used to **chain together the leaf nodes in search-key order**.

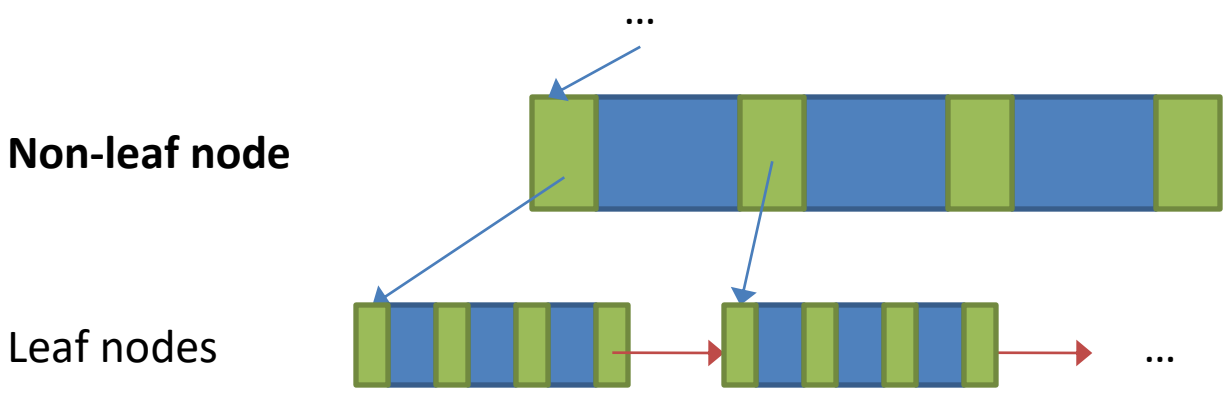
1. Leaf node



1001	**4**	Ben	100
1003	**1**	David	150
1005	**11**	Kit	40
1006	**9**	Anthony	90
1011	**6**	Kenneth	110

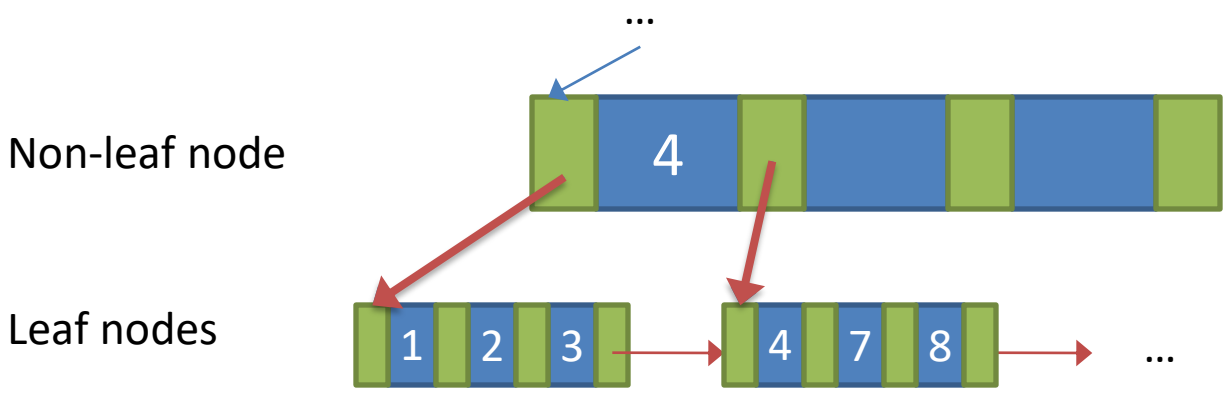
- The pointer **before** a search-key value points to the record that contains the search-key.

2. Non-leaf node



- Non-leaf nodes must hold **at least** $\lceil n/2 \rceil$, and at most ***n*** **pointers**.
  - E.g., with ***n*** = 4, a non-leaf node contains **at least 2 pointers**, and at most 4 pointers.

2. Non-leaf node



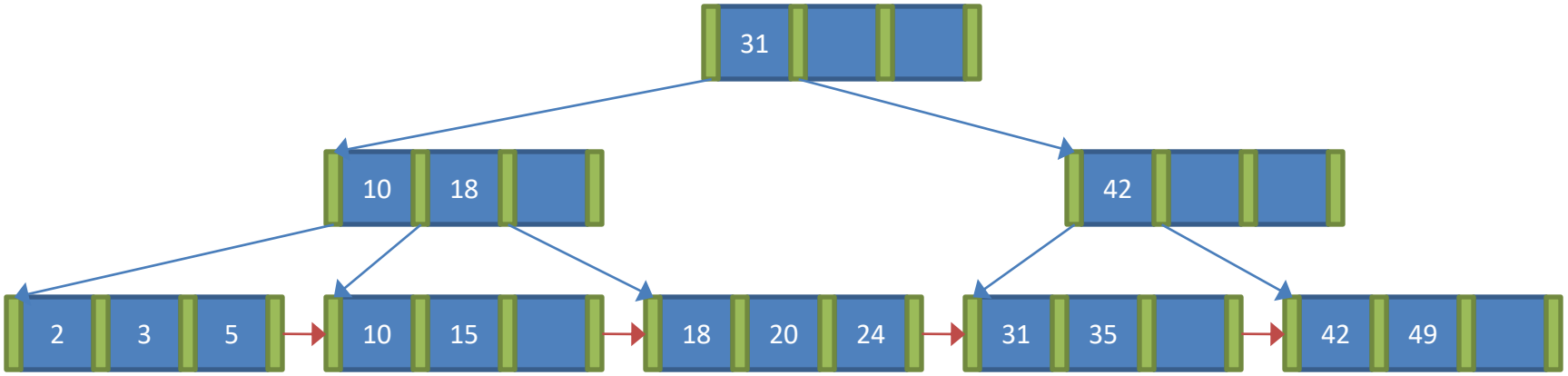
- The pointer on the left of a key ***K*** points to the part of the subtree that contains those key values **less than *K*.**
- The pointer on the right of a key ***K*** points to the part of the subtree that contains those key values **larger than or equal *K*.**

Example B$^+$-tree

In the file, records are ordered according to the 1st attribute, we would like to **build a B$^+$-tree index** (**secondary index**) to speed up the searching on the 2nd attribute.

10101	**49**	C.S.
12121	**42**	Finance
15151	**35**	Music
22222	**10**	Physics
32343	**15**	History
33456	**18**	C.S.
45565	**20**	E.E.E.
58583	**3**	Biology
76543	**31**	Finance
76766	**5**	Finance
83821	**2**	C.S.
98345	**24**	C.S.

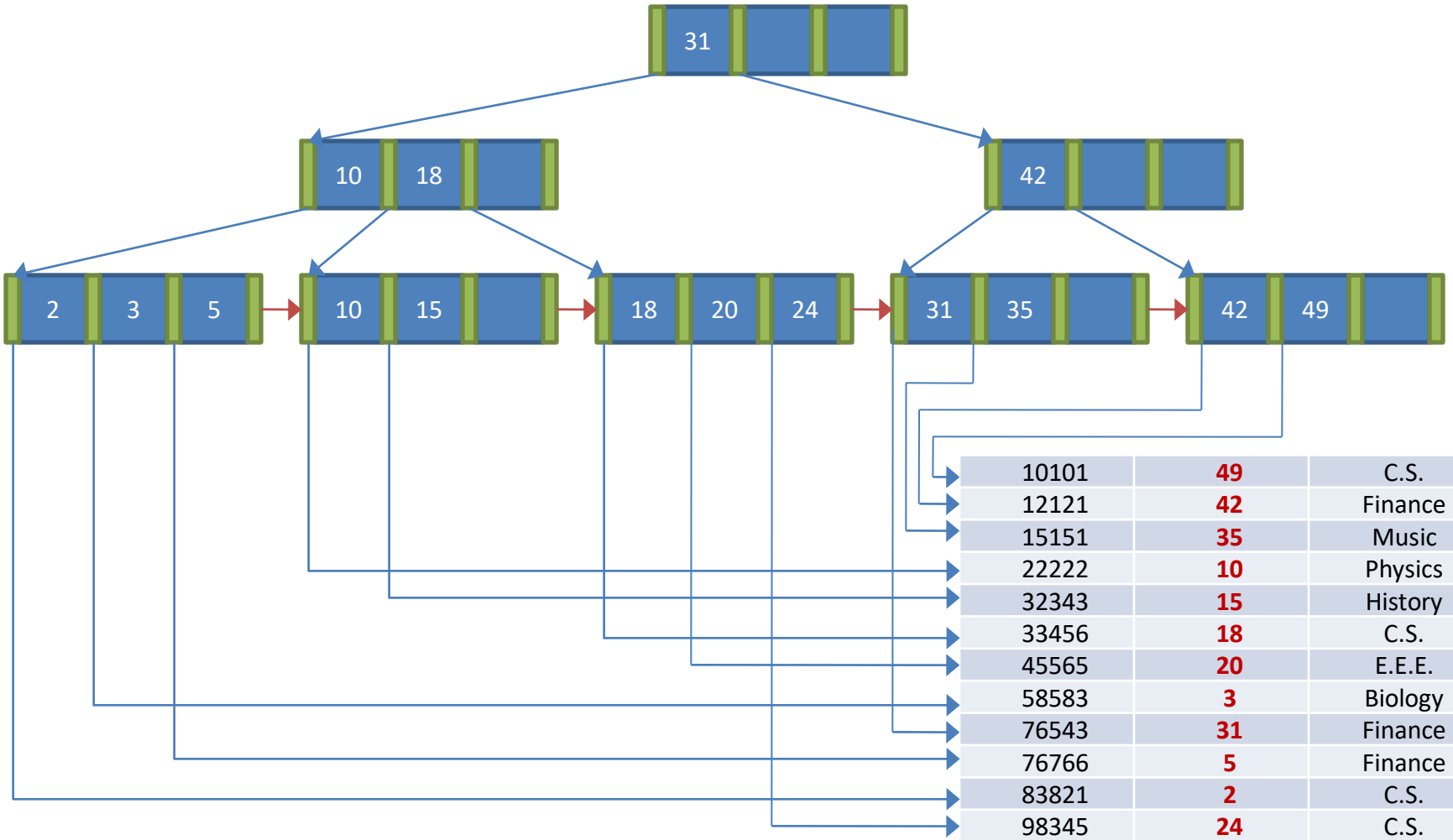
Example B$^+$-tree



- With ***n*** = 4, a leaf node must contain **at least 2 values**, and at most 3 values.
- With ***n*** = 4, a non-leaf node must contain **at least 2 pointers**, and at most 4 pointers.

10101	**49**	C.S.
12121	**42**	Finance
15151	**35**	Music
22222	**10**	Physics
32343	**15**	History
33456	**18**	C.S.
45565	**20**	E.E.E.
58583	**3**	Biology
76543	**31**	Finance
76766	**5**	Finance
83821	**2**	C.S.
98345	**24**	C.S.

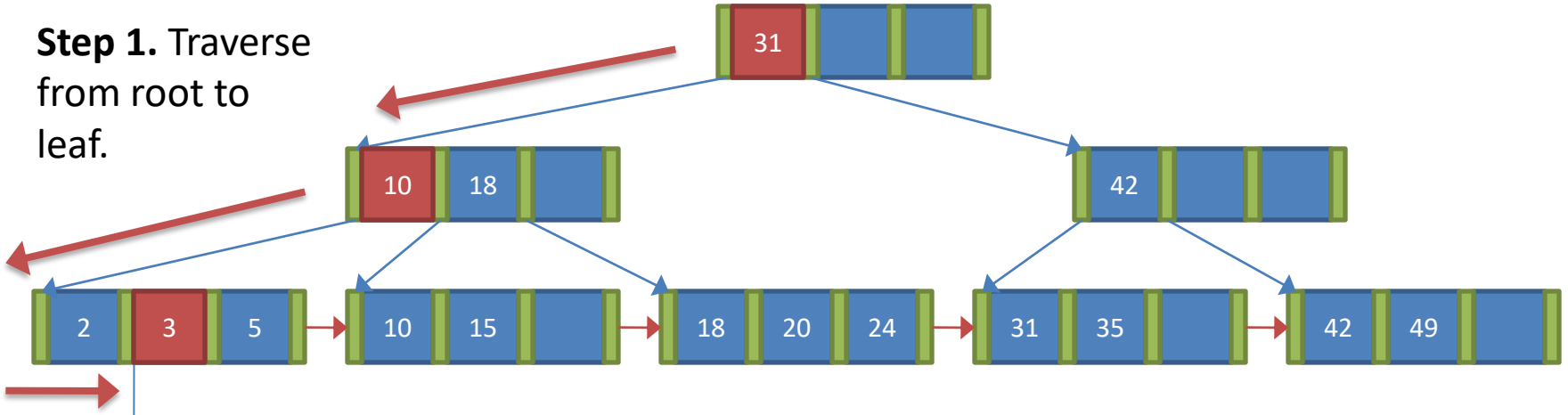
Example B+-tree



10101	49	C.S.
12121	42	Finance
15151	35	Music
22222	10	Physics
32343	15	History
33456	18	C.S.
45565	20	E.E.E.
58583	3	Biology
76543	31	Finance
76766	5	Finance
83821	2	C.S.
98345	24	C.S.

Searching

Step 1. Traverse from root to leaf.



Step 2. Search in the leaf node.

Point query

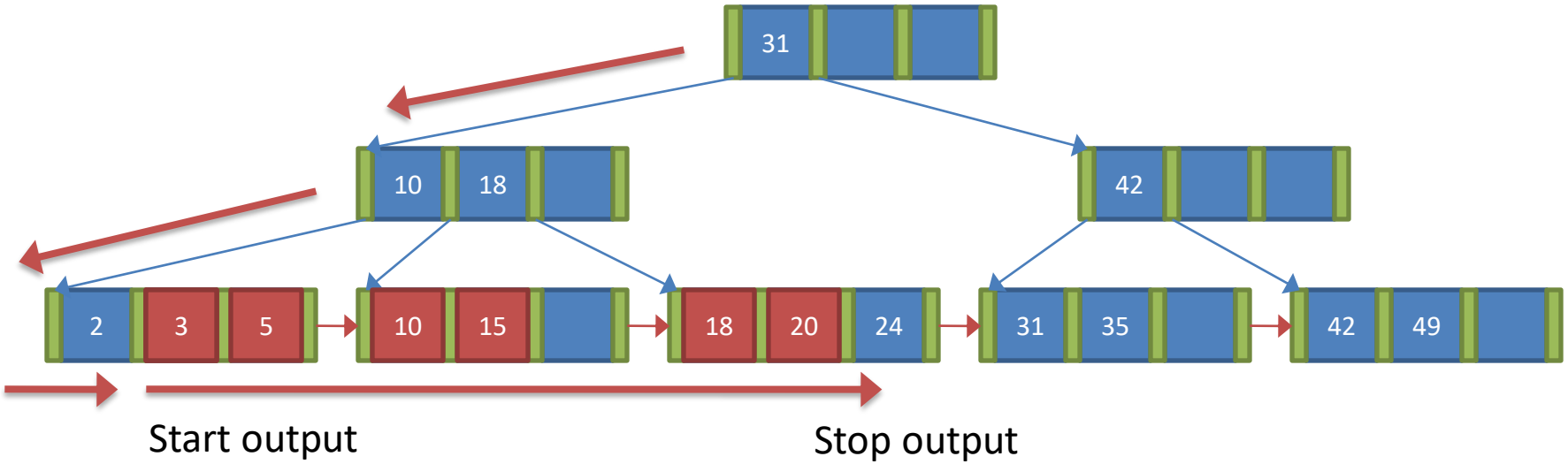
```
SELECT * FROM R WHERE R.B = 3
```

Step 3. Follow the pointer in the leaf node to retrieve the record.

10101	49	C.S.
12121	42	Finance
15151	35	Music
22222	10	Physics
32343	15	History
33456	18	C.S.
45565	20	E.E.E.
58583	3	Biology
76543	31	Finance
76766	5	Finance
83821	2	C.S.
98345	24	C.S.

With this B$^+$-tree, how many disk block accesses to answer this query?

Searching



Range query

```
SELECT * FROM R WHERE R.B >= 3 AND R.B < 22
```

B⁺-tree can also handle **range search** very well. Search for the left border of the range and **traverse the leaf chain** until a record with search-key larger than the right border is encountered.

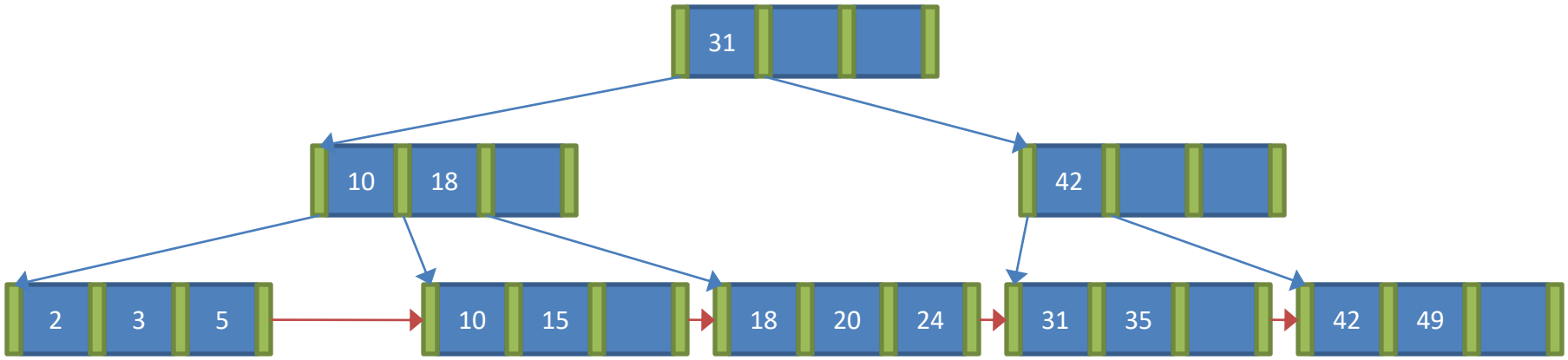
10101	49	C.S.
12121	42	Finance
15151	35	Music
22222	10	Physics
32343	15	History
33456	18	C.S.
45565	20	E.E.E.
58583	3	Biology
76543	31	Finance
76766	5	Finance
83821	2	C.S.
98345	24	C.S.

Insertion

- Assume no duplicate entries are inserted, insertion is simply **searching** + **insert entry**.
- If a leaf node is full, **node splitting** has to be performed.
  - **Step 1.** Create one more node and distribute the first $\lceil n/2 \rceil$ records to one node and the remaining to the other node.
  - **Step 2.** **Parent nodes (non-leaf nodes) have to be updated accordingly.**

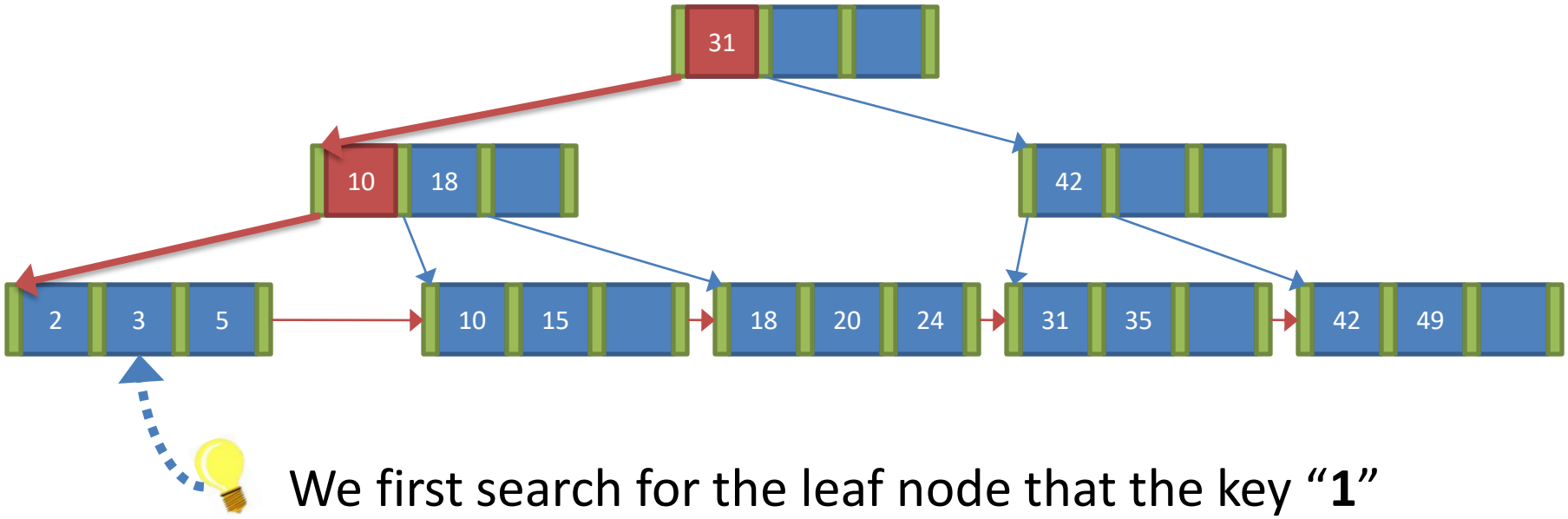
Insert key "1"

1. Node splitting (leaf node)



Let's learn how **node splitting** is implemented on leaf node by considering inserting key "**1**" in the above B$^+$-tree.

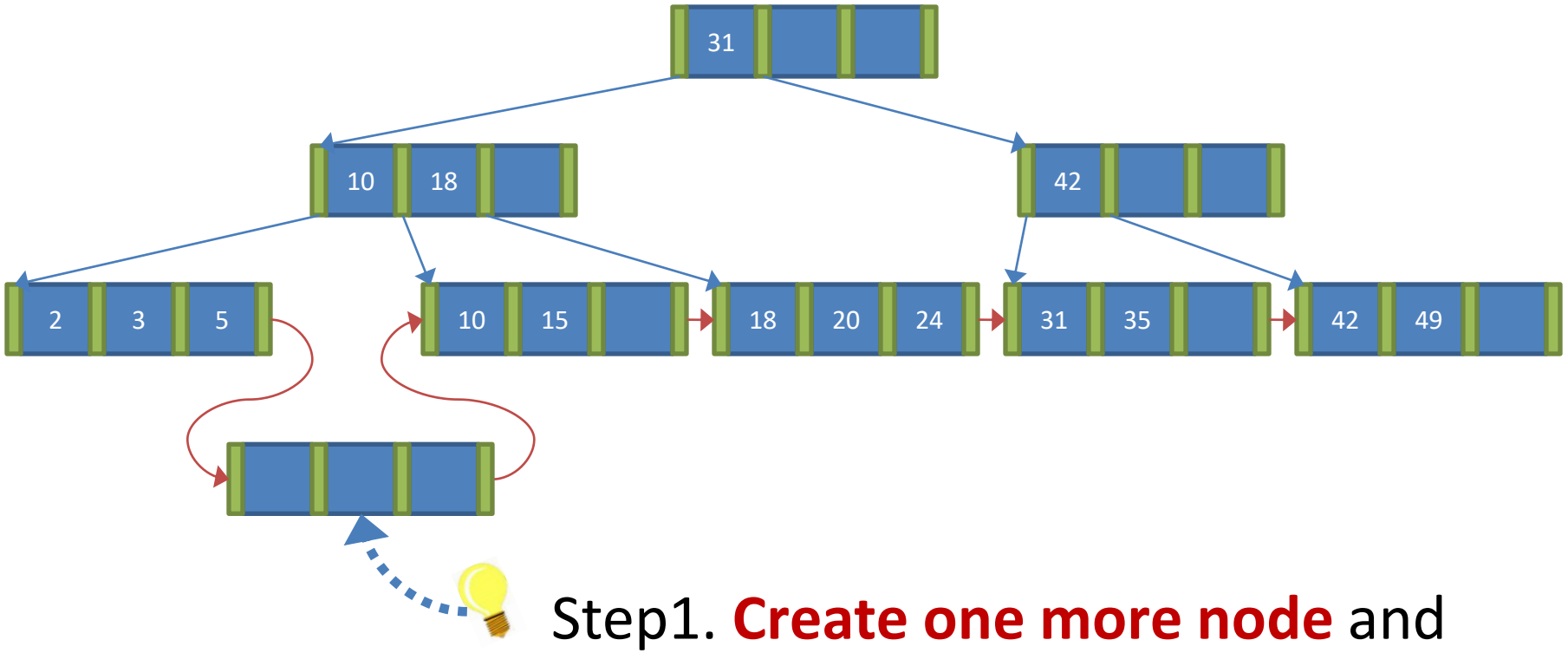
1. Node splitting (leaf node)



We first search for the leaf node that the key "**1**" should be inserted.

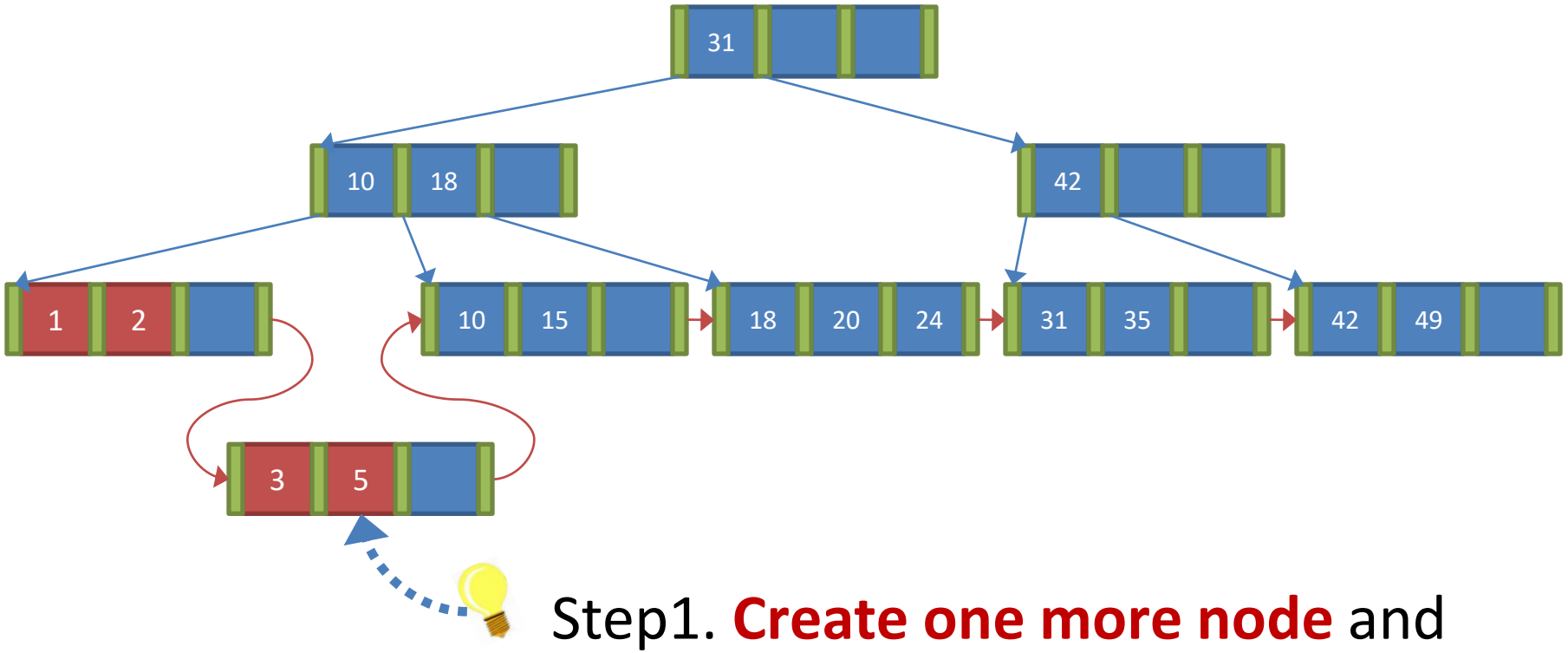
Since this node is **full**, inserting "**1**" requires **SPLITTING** this leaf node.

1. Node splitting (leaf node)



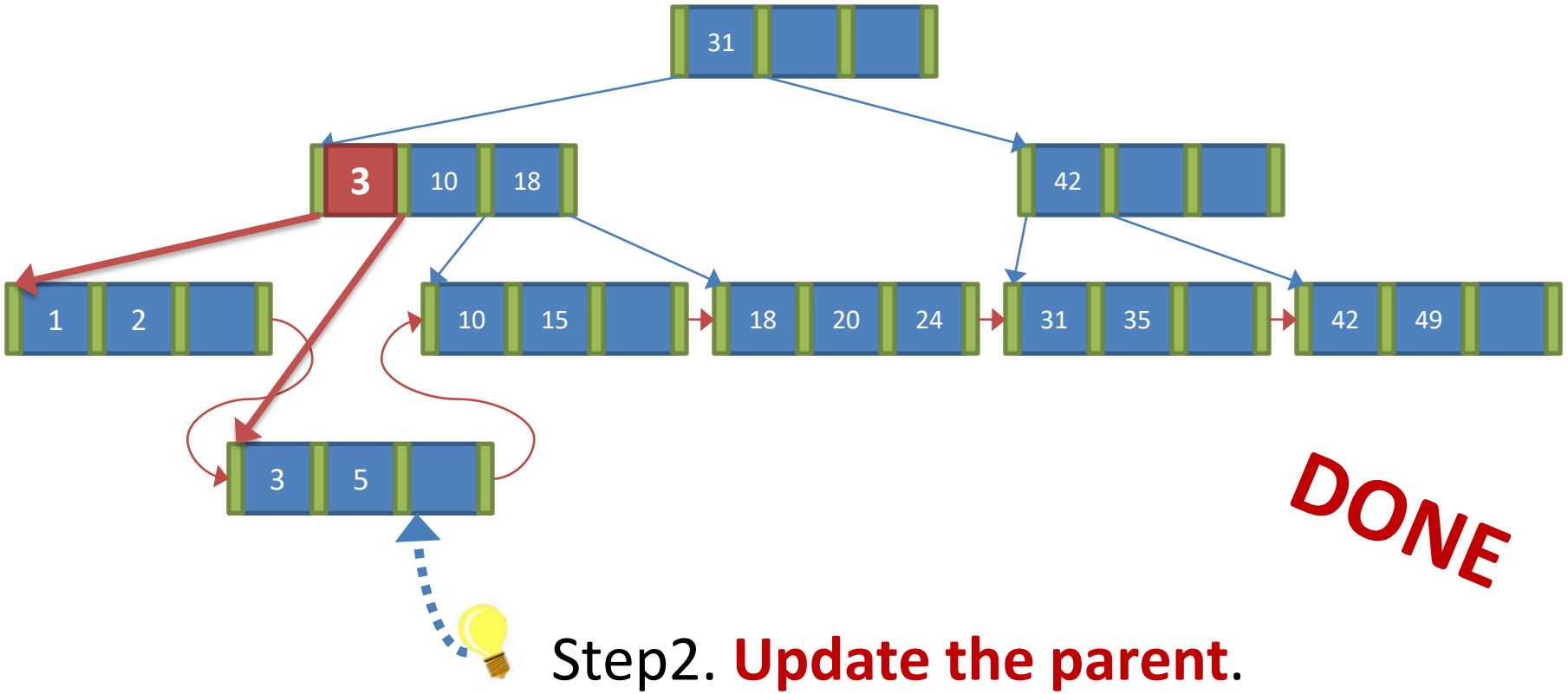
Step1. **Create one more node** and **distribute the entries**.

1. Node splitting (leaf node)



Step1. **Create one more node** and **distribute the entries**.

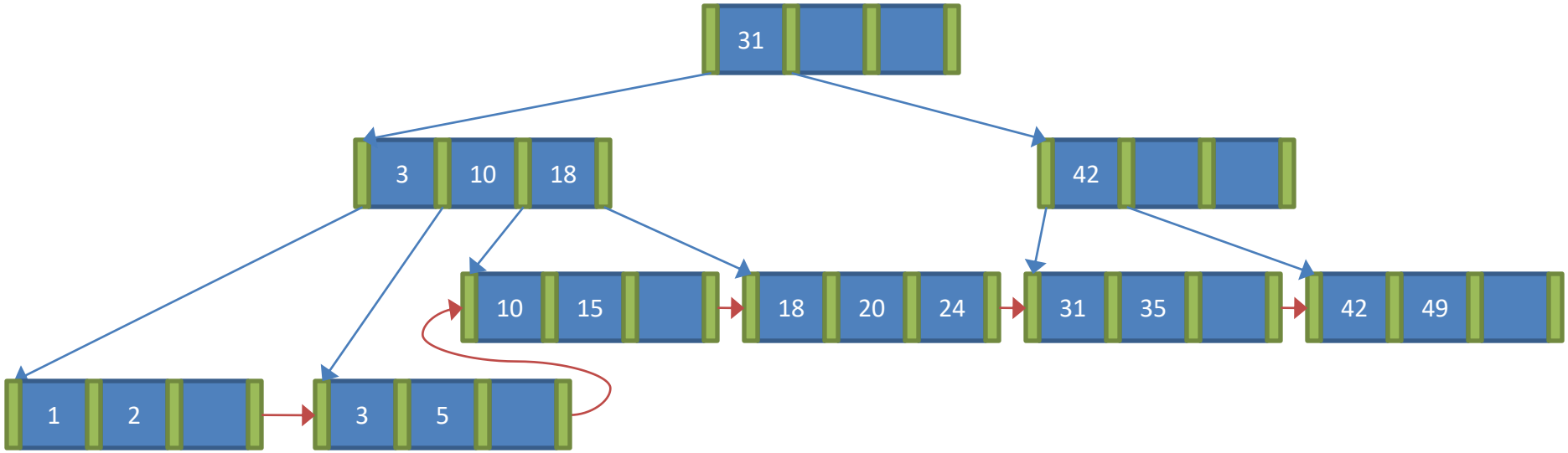
1. Node splitting (leaf node)



Step2. **Update the parent**.

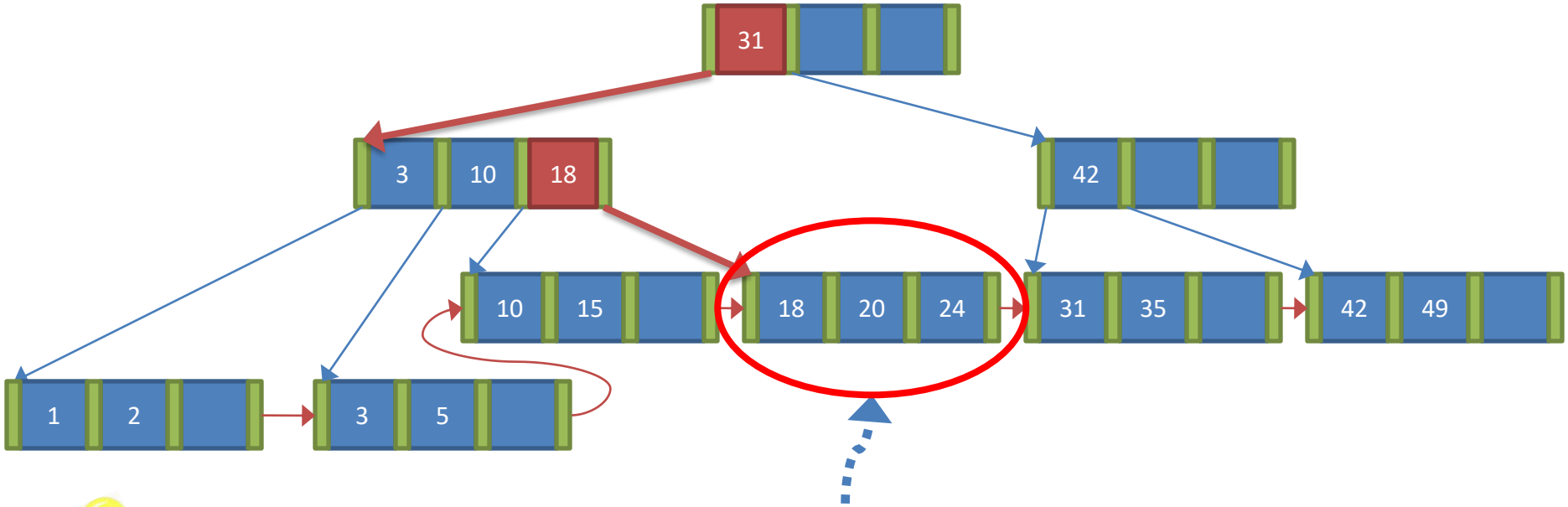
2. Node splitting (non-leaf node)

- Splitting of a **non-leaf node** is a little different from splitting of a leaf node.



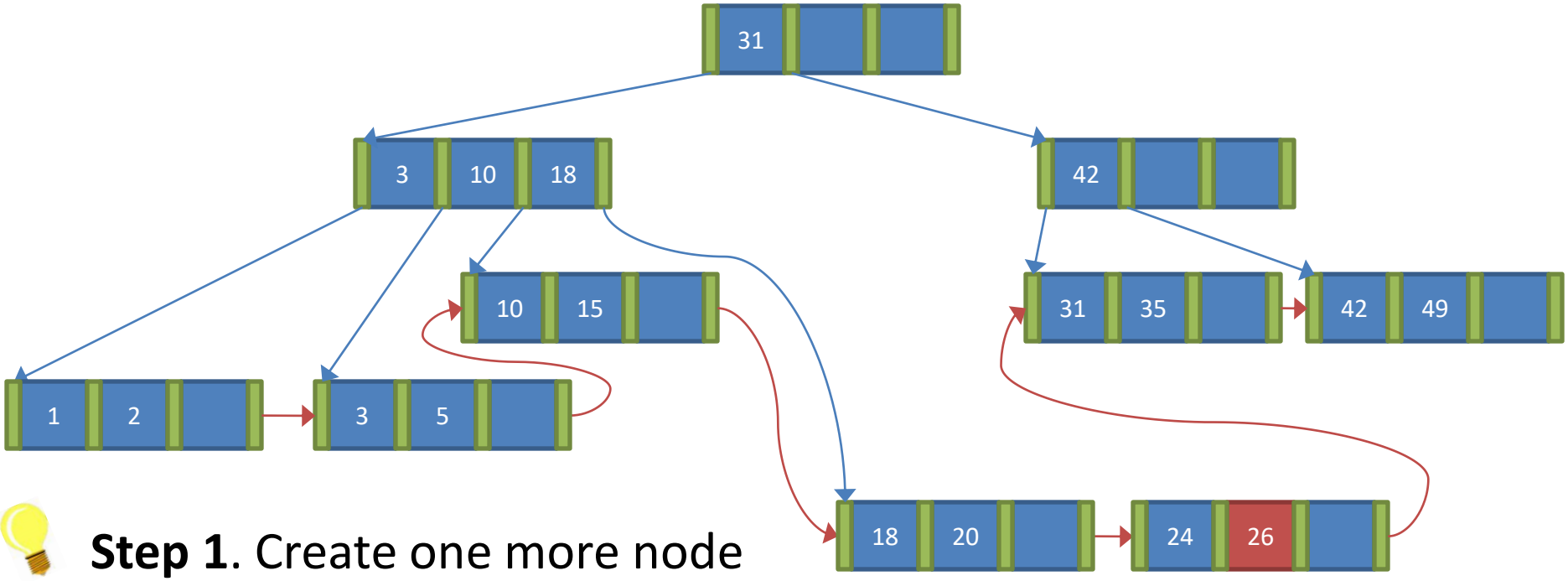
Let's learn how node splitting is implemented on non-leaf node by considering inserting key "**26**" in the above B$^+$-tree.

2. Node splitting (non-leaf node)



"**26**" should be inserted into this leaf node.
Since this node is **full**, node **SPLITTING** need to be performed.

2. Node splitting (non-leaf node)

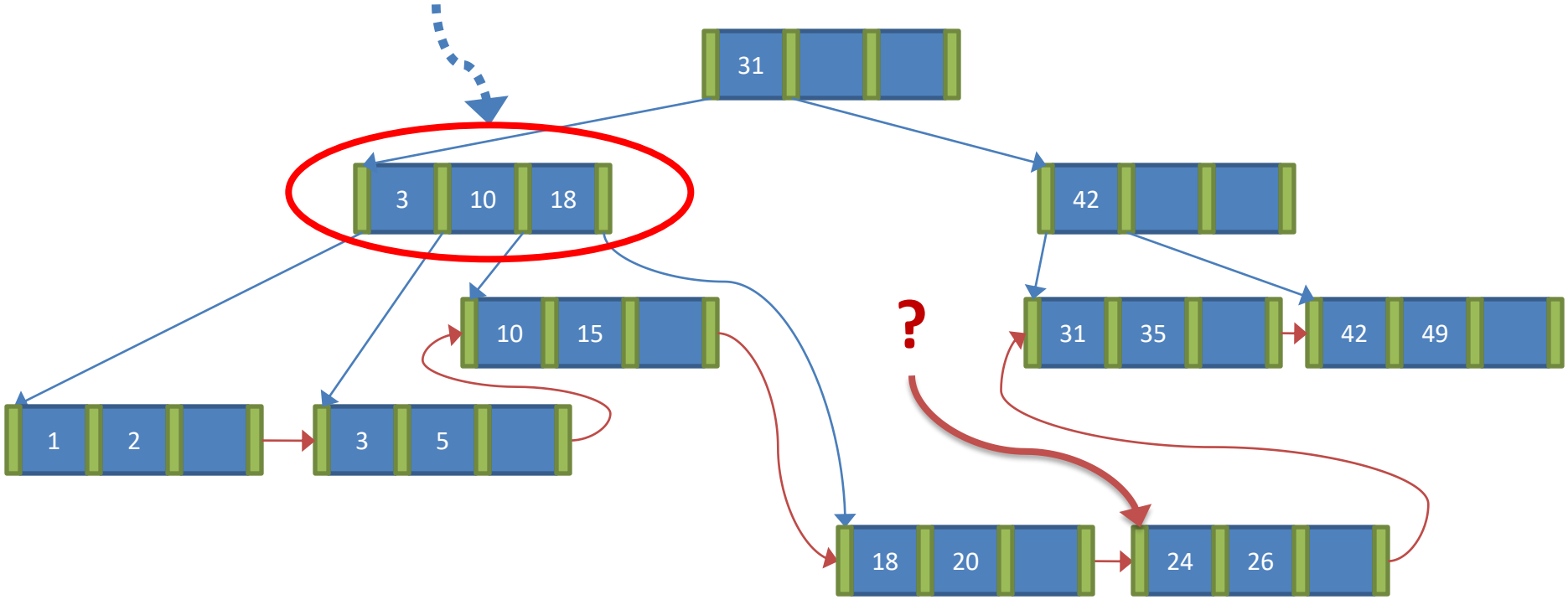


Step 1. Create one more node and distribute the entries.

2. Node splitting (non-leaf node)

Step 2. Update the parent (Parent node is full!)

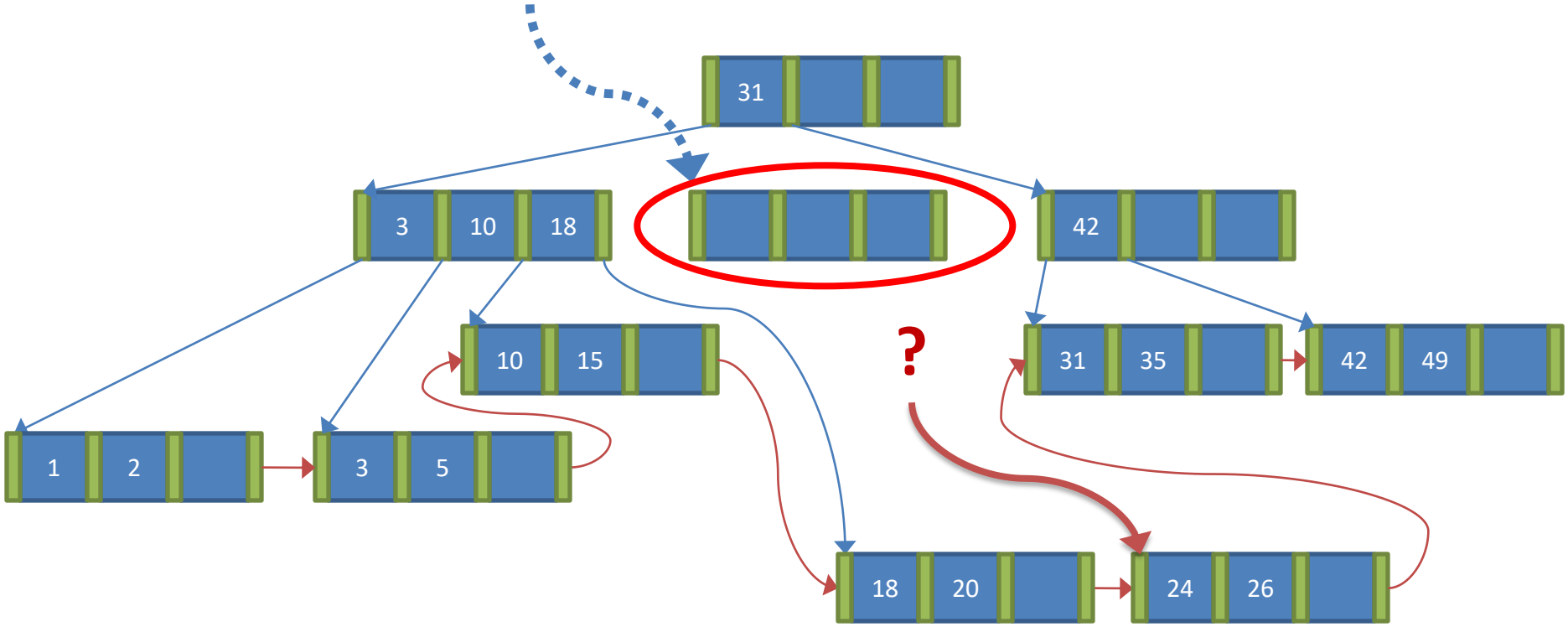
As we cannot have 5 pointers stored in a non-leaf node, **we need to split this non-leaf node (Recursively).**



2. Node splitting (non-leaf node)

Splitting non-leaf node (Recursive)

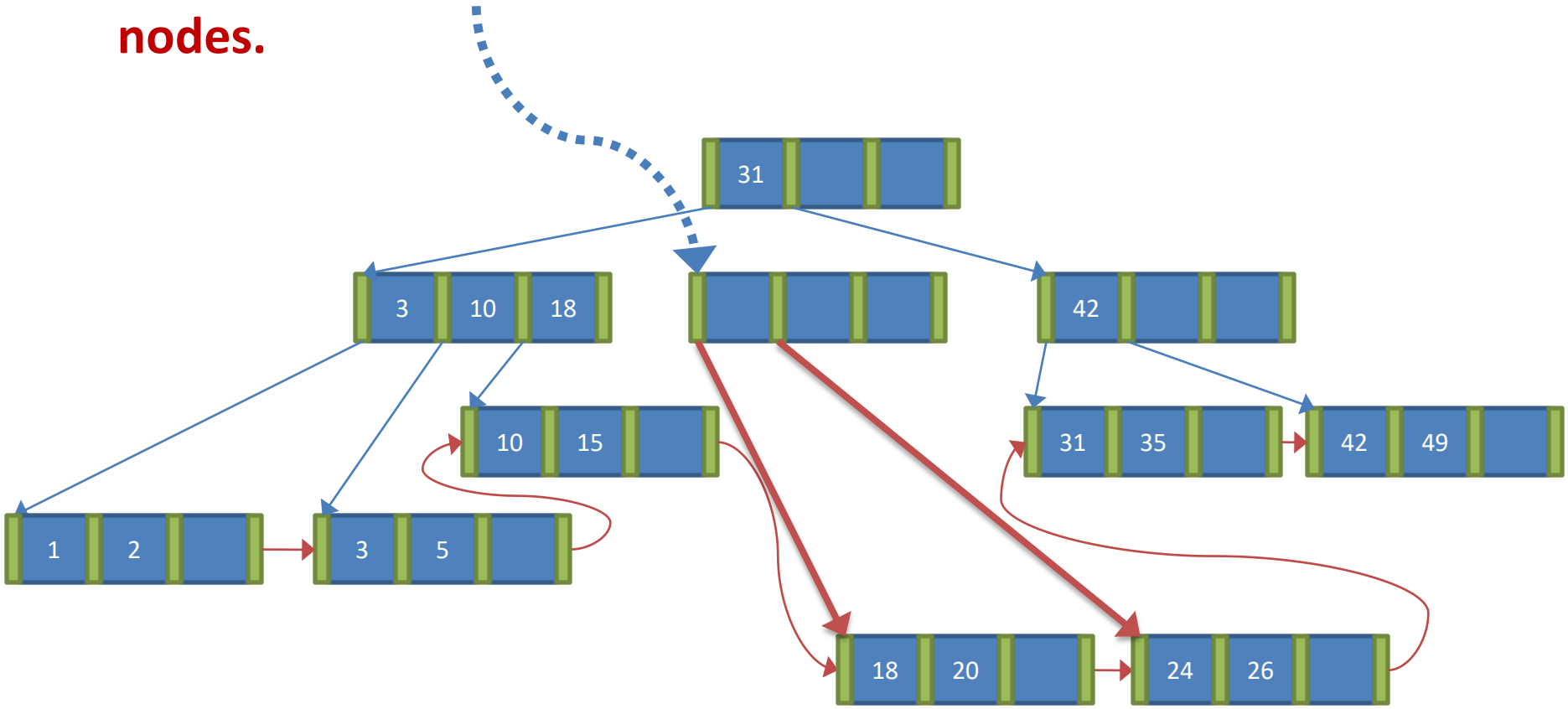
Step 1. We first create a new node to accommodate the new **pointers (the 5 pointers, one for each leaf node)**.



2. Node splitting (non-leaf node)

Splitting non-leaf node

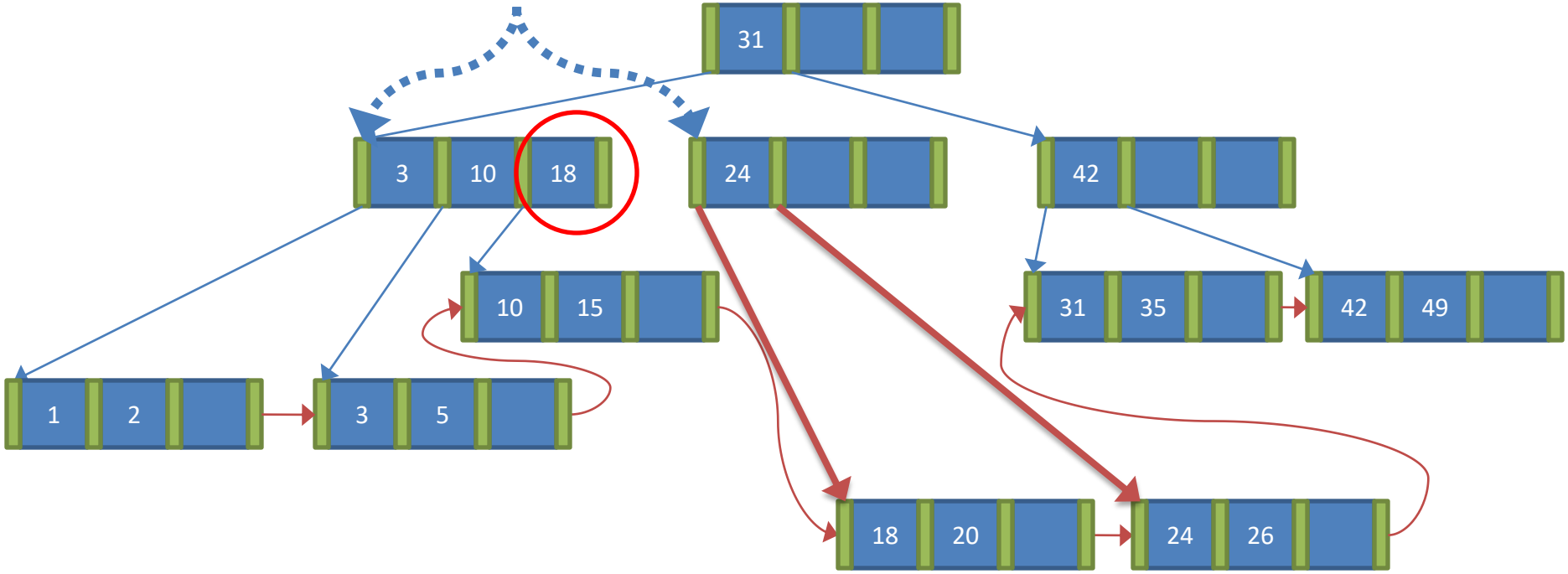
Step 2. We distribute the **pointers among the two nodes.**



2. Node splitting (non-leaf node)

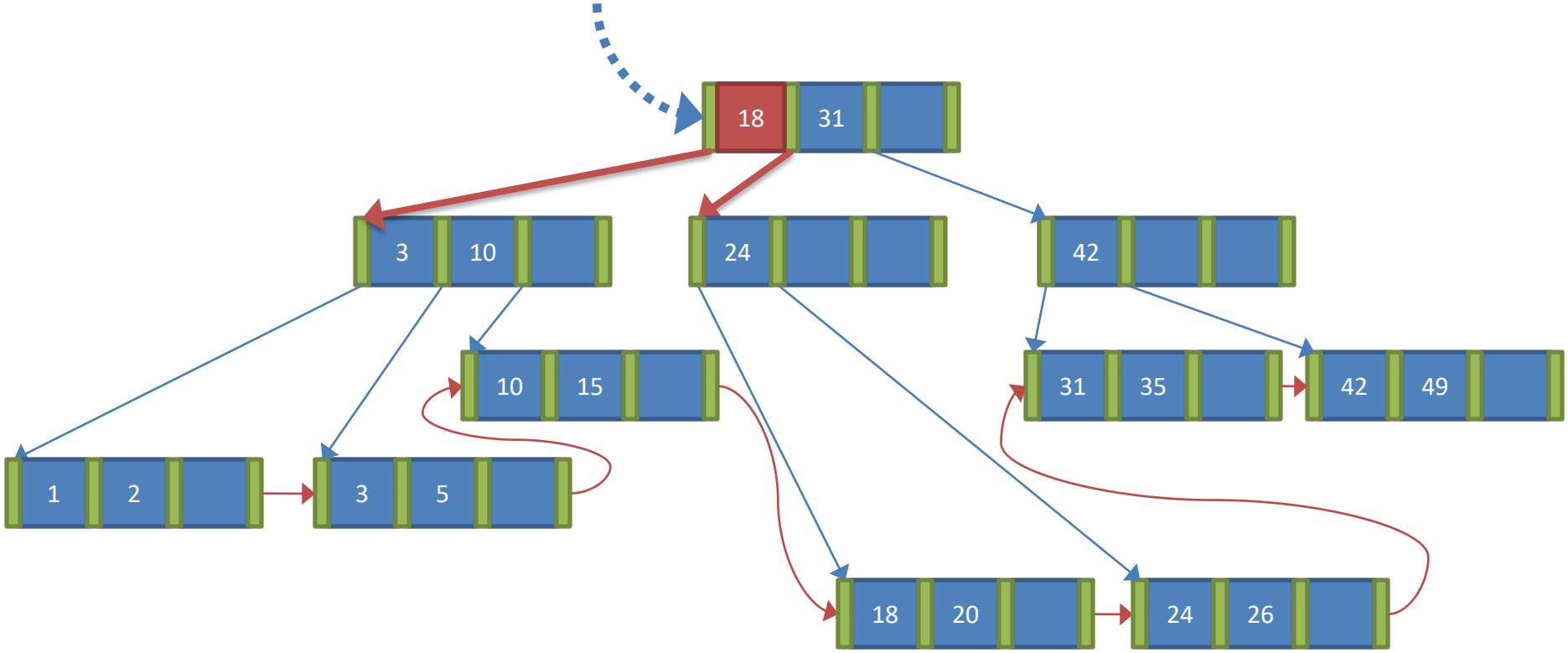
Splitting non-leaf node

Step 3. Then consider the keys that are required in each slot among the two nodes.



2. Node splitting (non-leaf node)

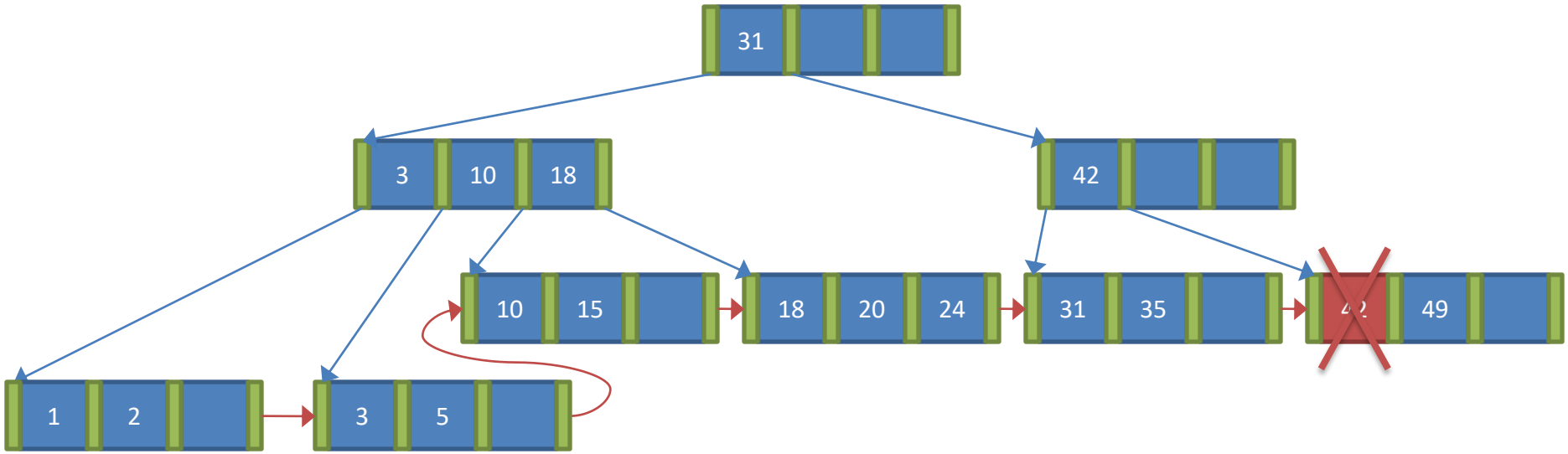
"18" is **moved to the parent node to separate the search-keys among the two nodes** (if the parent node is full, split the parent node recursively)



Deletion

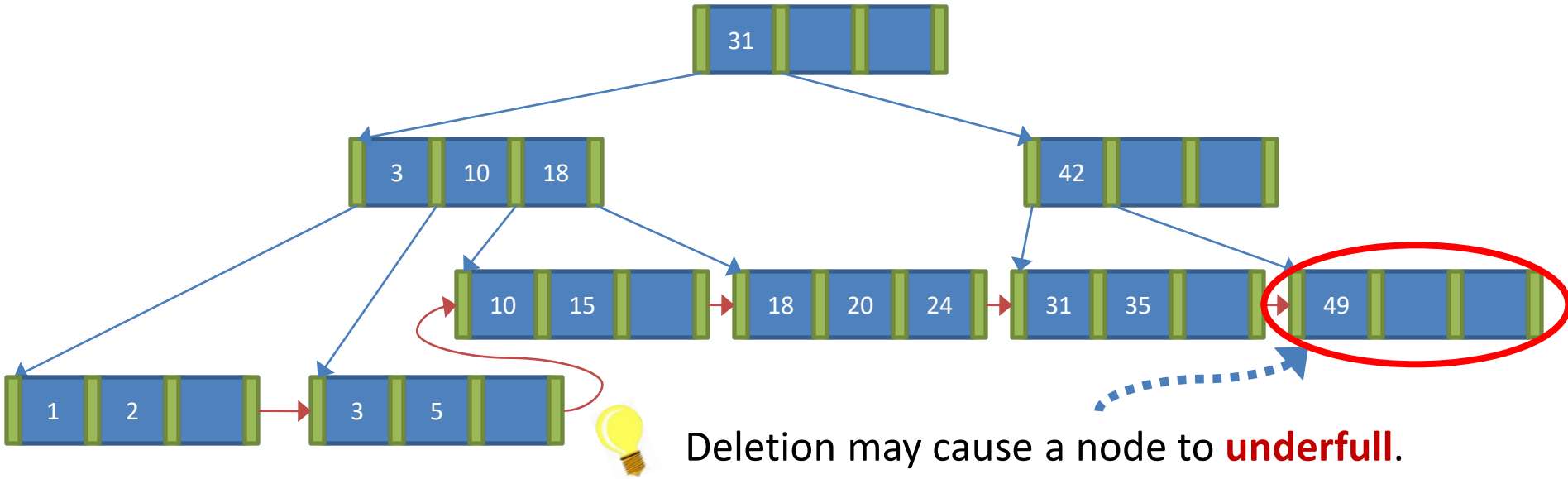
- Find the record to be deleted.
- Remove it from the file and from the leaf node (if present)
- If the leaf node has too few entries due to the removal:
  - Try to **MERGE** the node with its sibling node.
  - Try to **REDISTRIBUTE** the entries if MERGING fails.

1. Merging



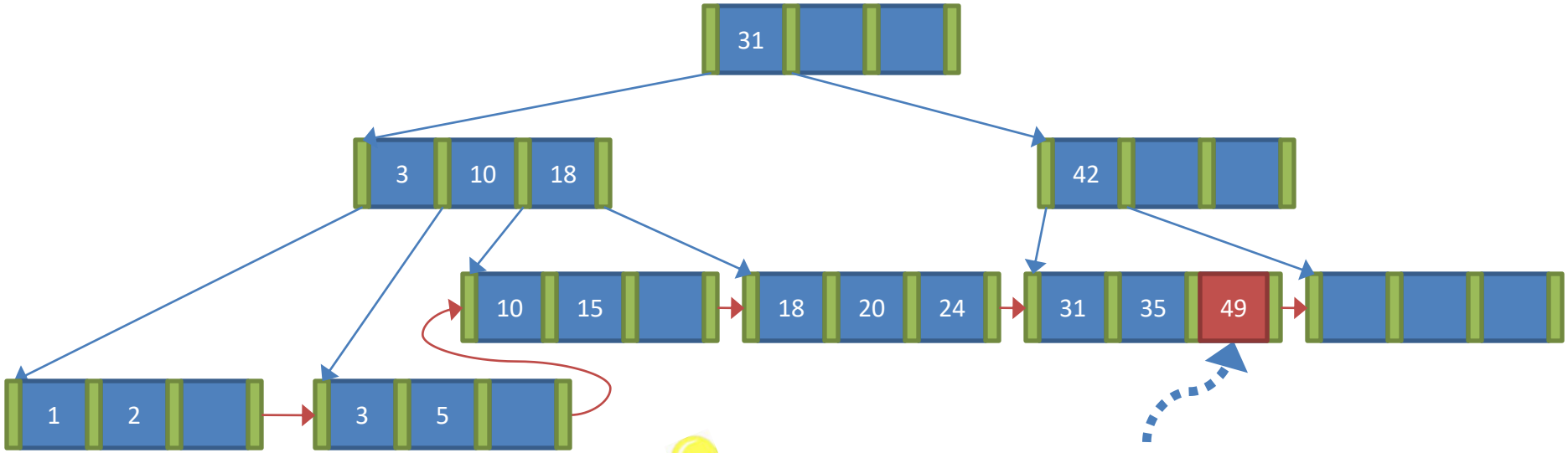
Let's try to remove key "**42**" in the above B$^+$-tree.

1. Merging



Deletion may cause a node to **underfull**.
This node has only 1 value, which violates the requirement that each leaf node must contain at least $\lceil (n-1)/2 \rceil$ values (i.e., 2 in this case).

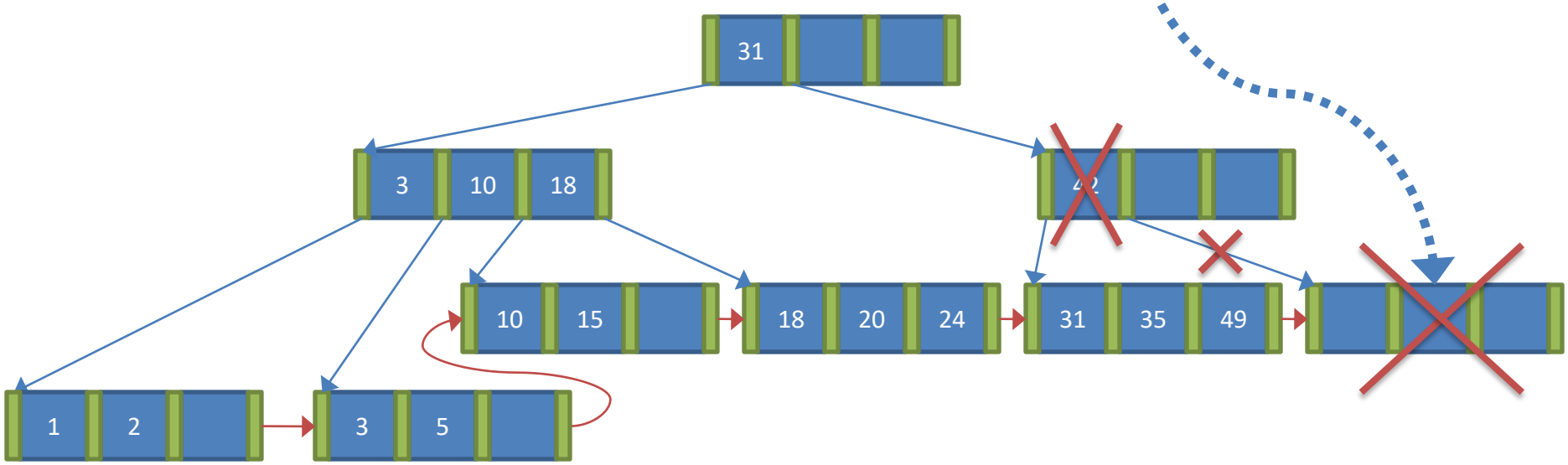
1. Merging



Step 1. Merge with sibling node.

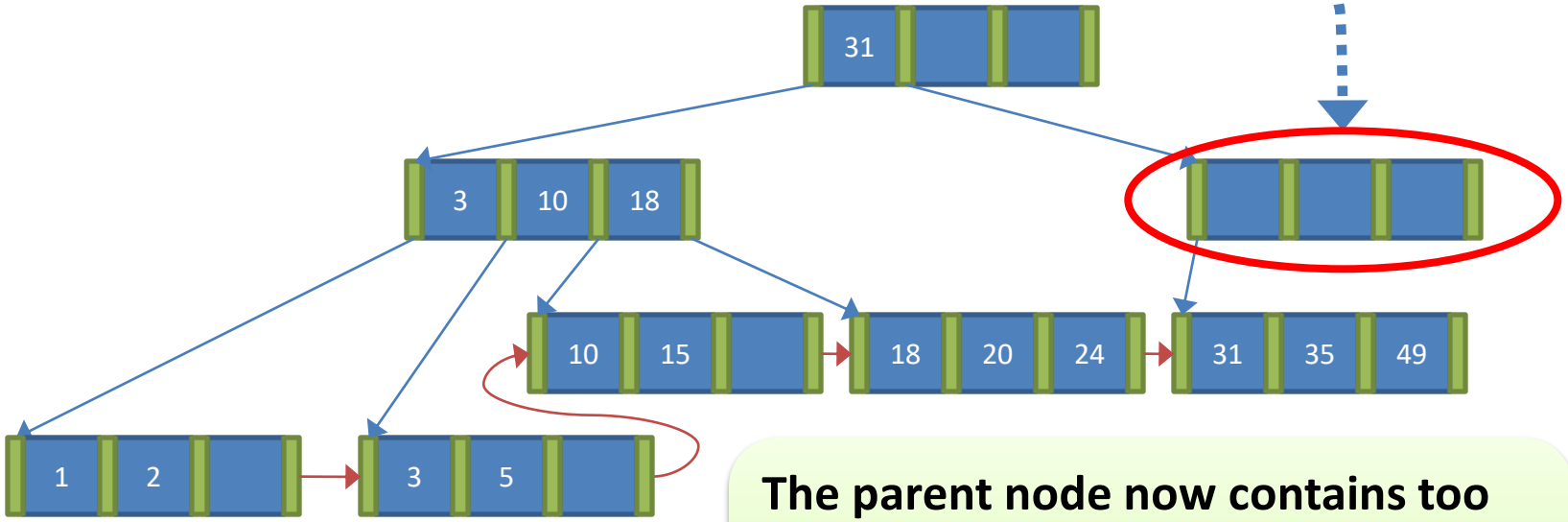
1. Merging

After merging, this leaf node is empty and **no longer used**.



1. Merging

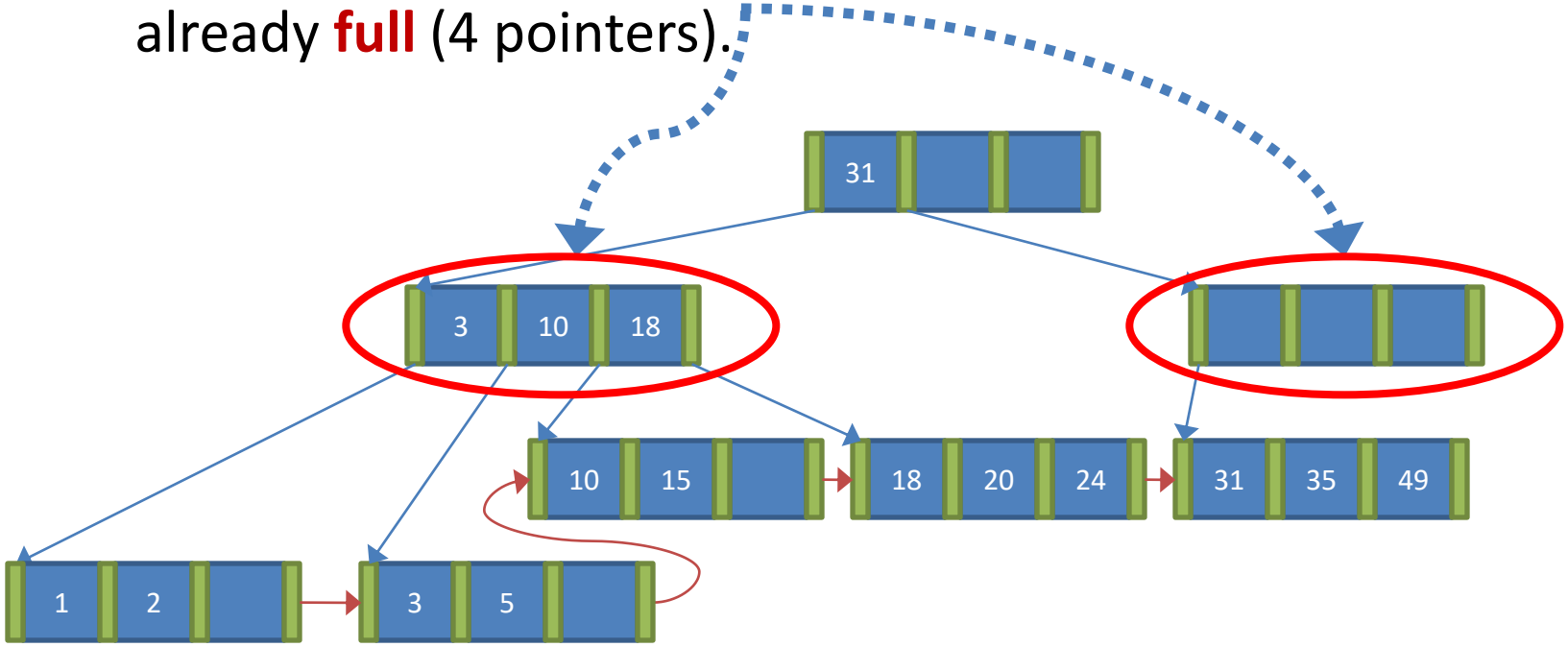
Step 2. Update the parents.



The parent node now contains too few pointers. Remember we require non-leaf node to have at least $\lceil n/2 \rceil$ **pointers.**

1. Merging

Recursively, we try to **MERGE** these 2 nodes.
However, the two nodes **cannot be merged** as the left node is already **full** (4 pointers).

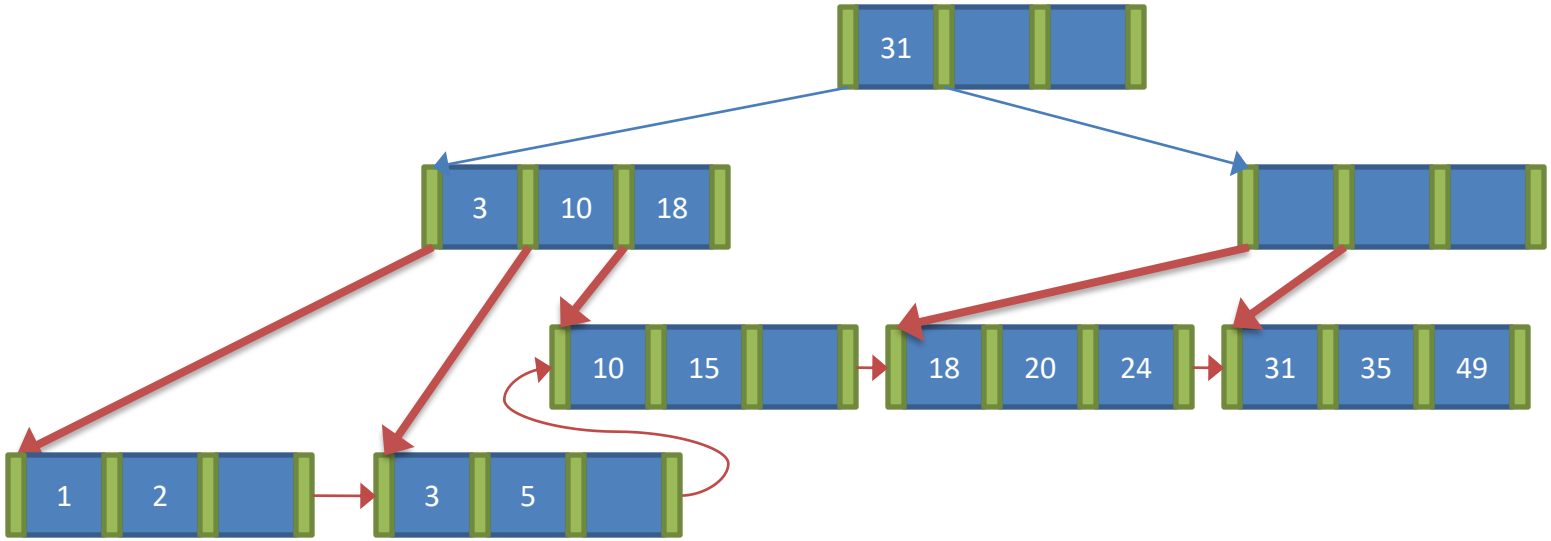


When **MERGE** fails, do **REDISTRIBUTION!**

2. Redistribution

Redistribution

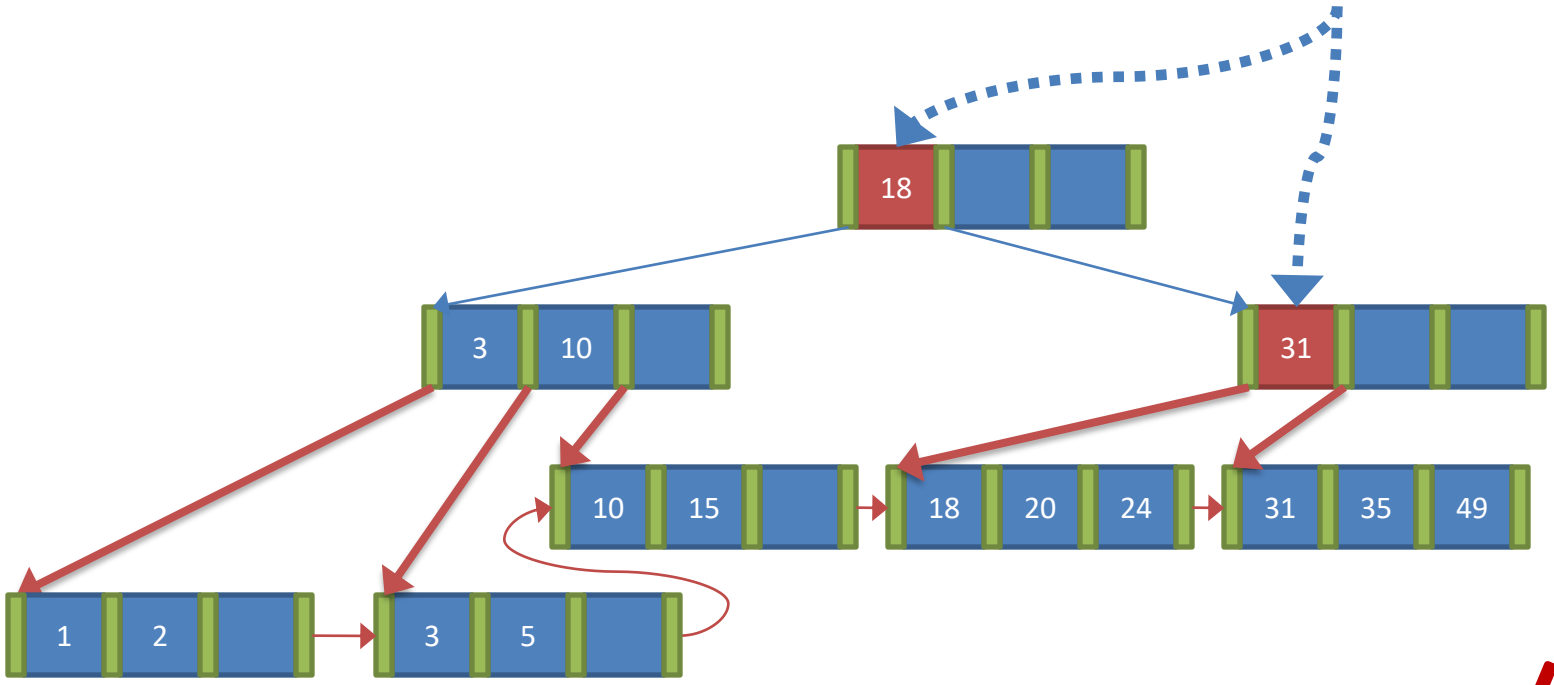
Step1. Redistribute the pointers.



2. Redistribution

Redistribution

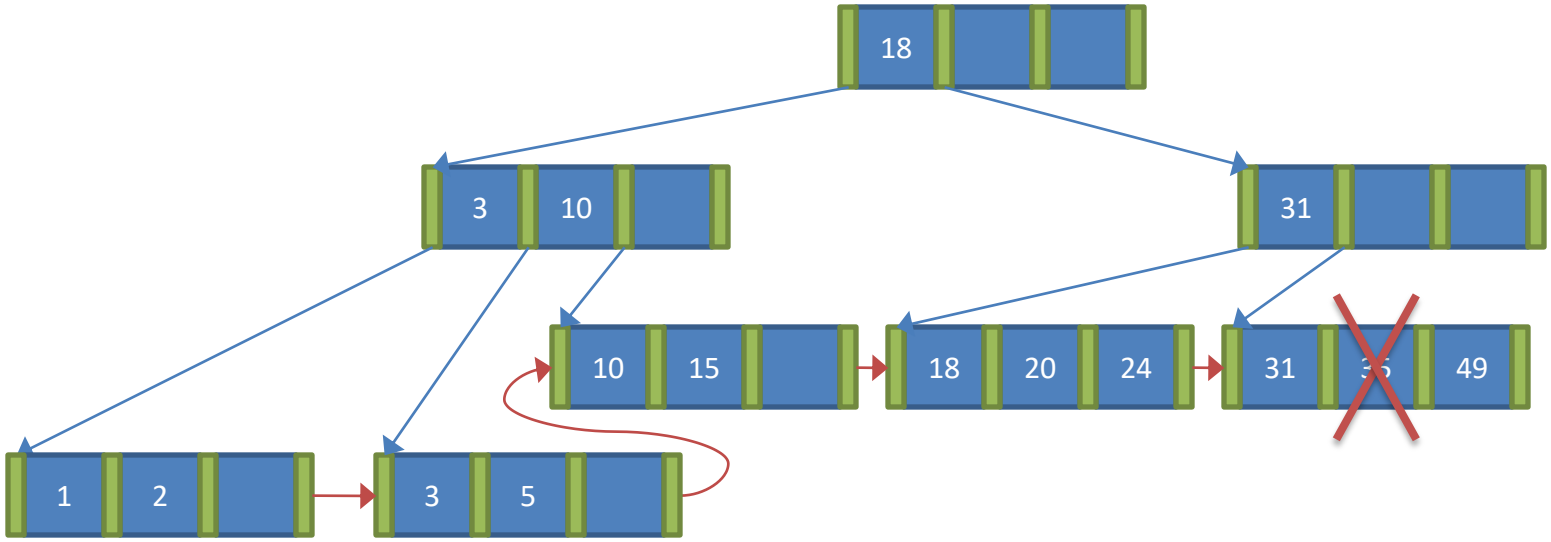
Step2. Update the keys.



DONE

Example 1

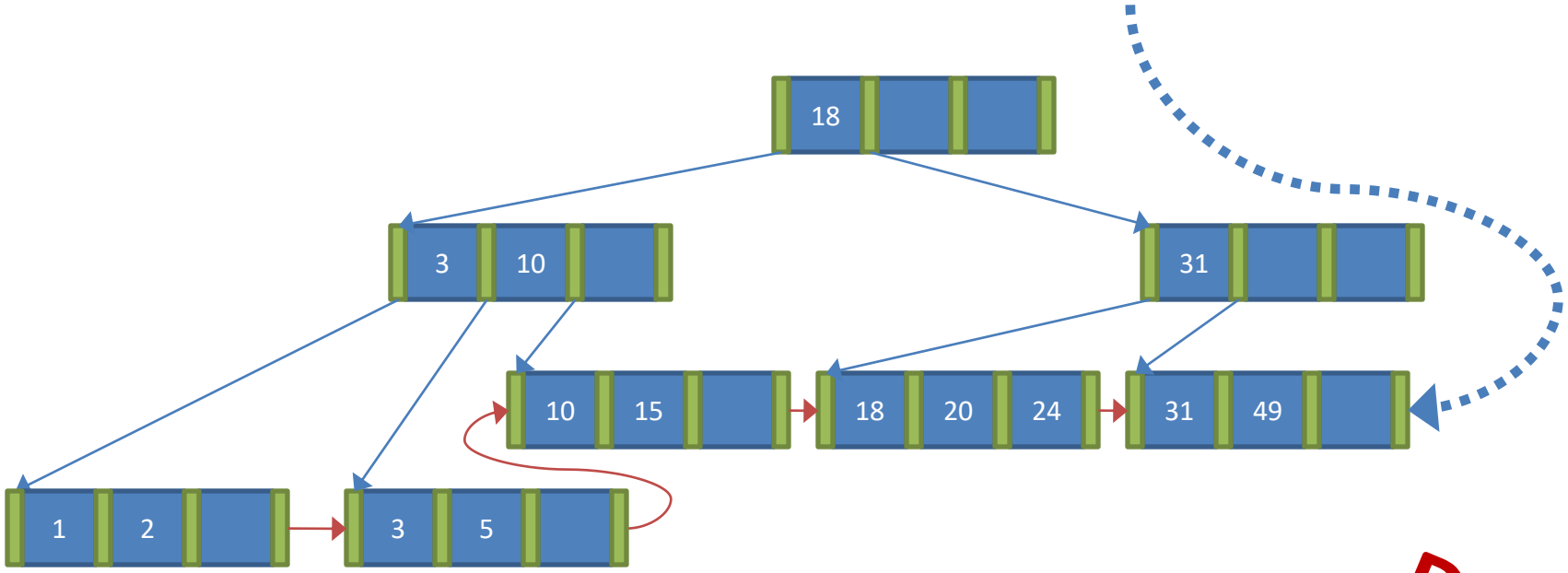
- Delete **35**



Example 1

- Delete **35**

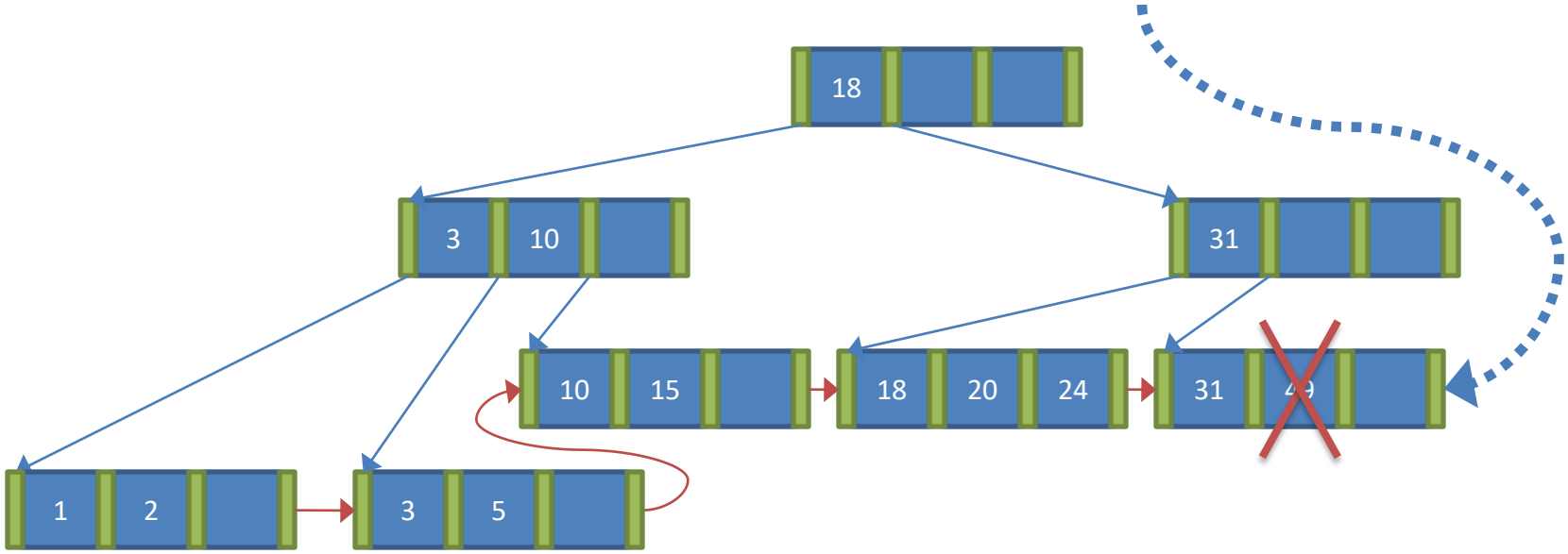
After deletion, this node contains 2 values (VALID). **Remember the keys in a node should be in sorted order**.



Example 2

- Delete **49**

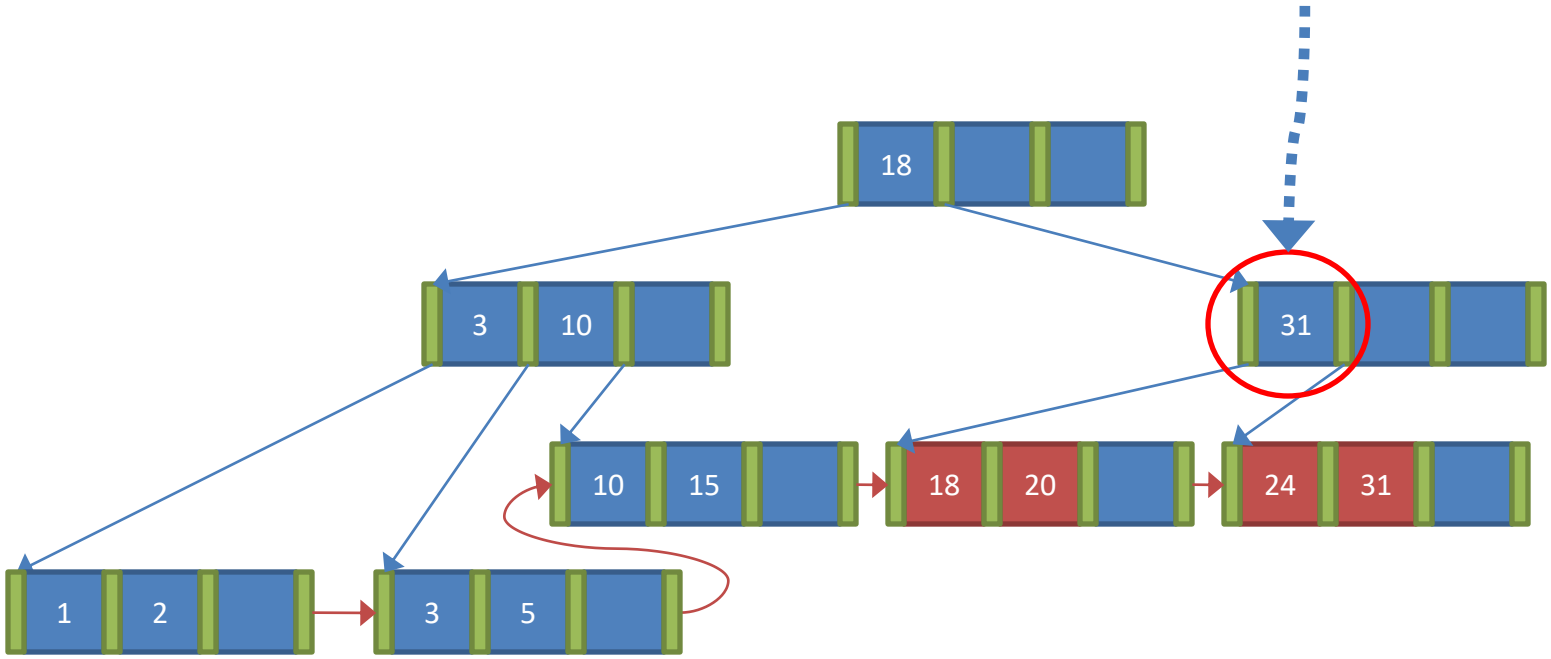
Deletion of "**49**" causes this leaf node to contain only one value, which is **underfull**.
First, try **MERGE** with its sibling node, but the sibling node is **full**, so we need to do **REDISTRIBUTION**.



Example 2

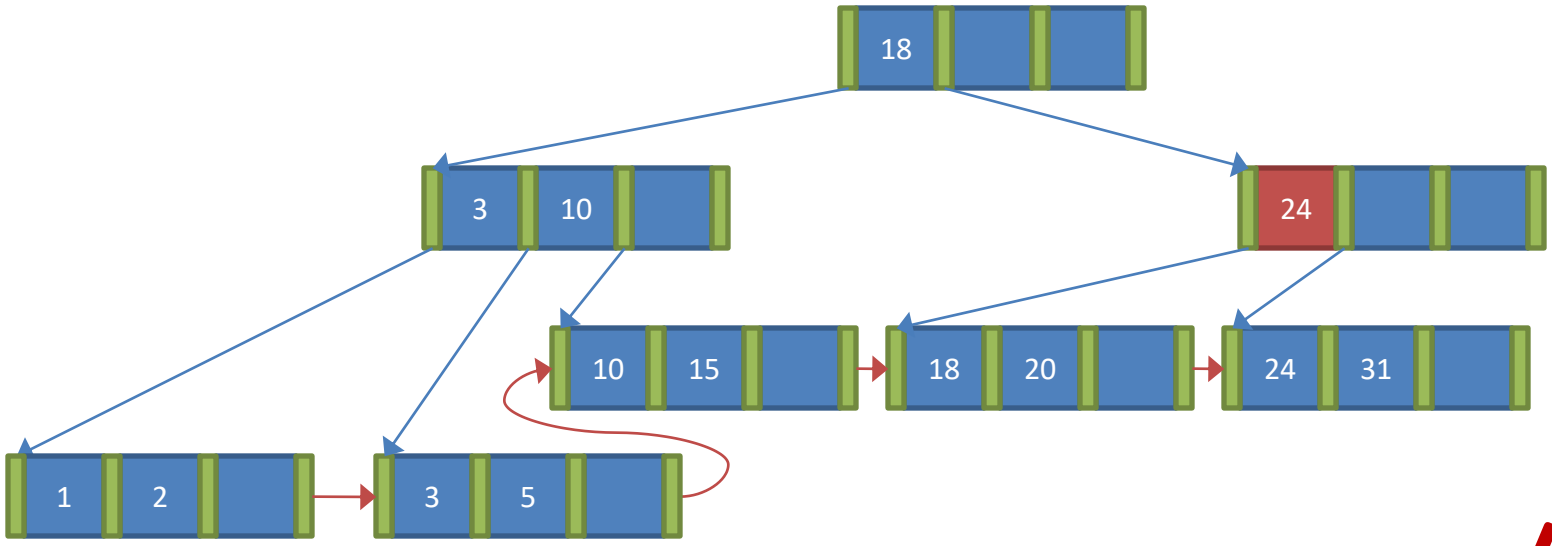
- Delete **49**

After **REDISTRIBUTION**, we need to update the keys.



Example 2

- Delete **49**

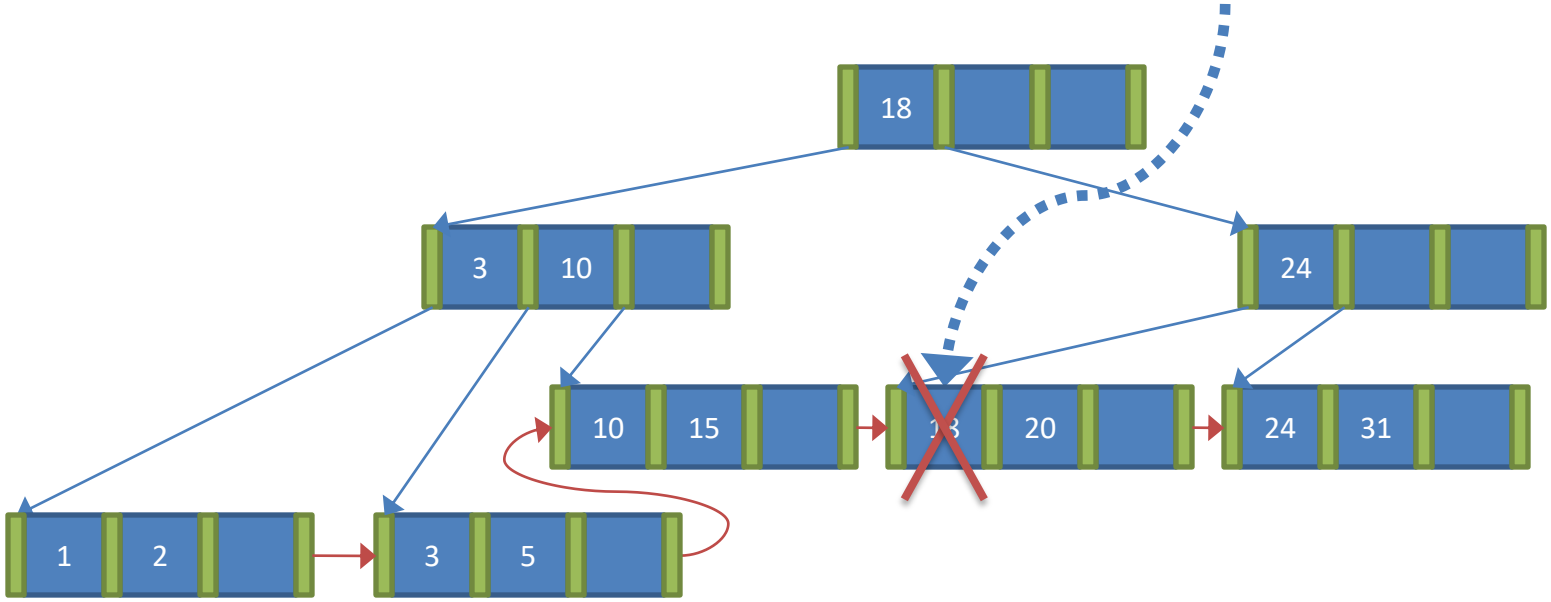


DONE

Example 3

- Delete **18**

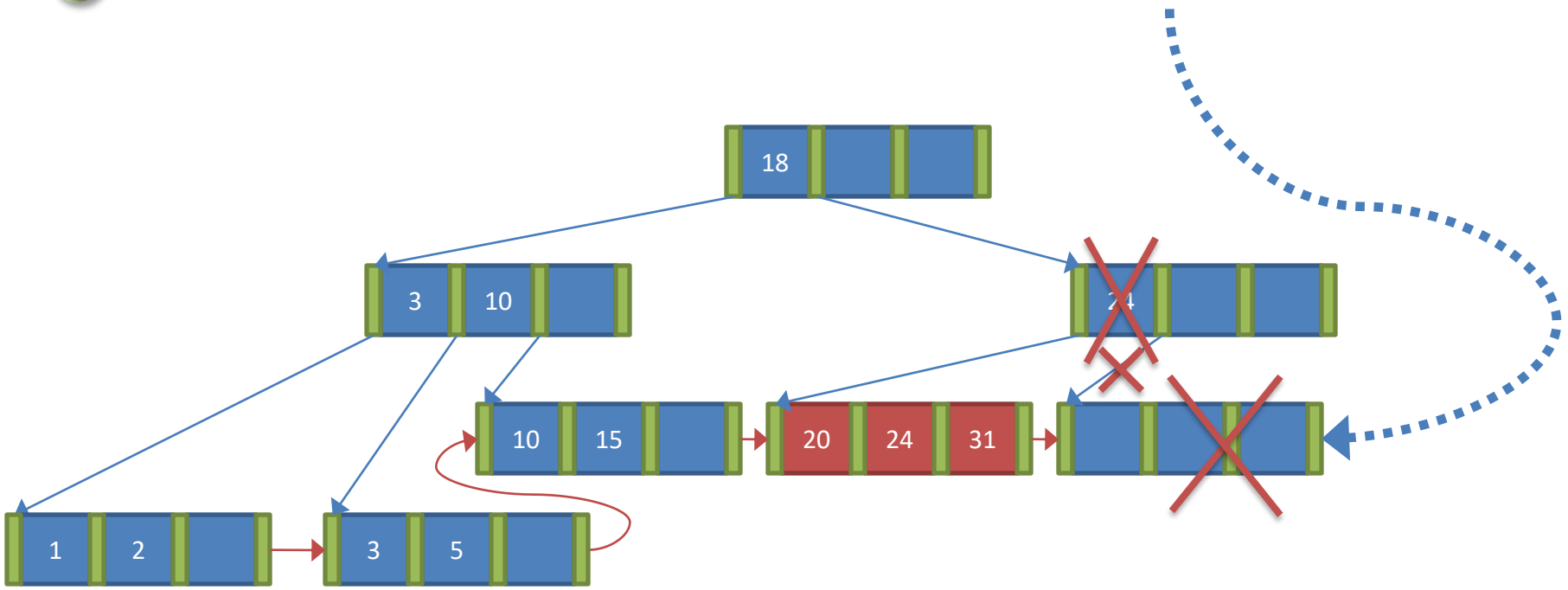
Deletion of "**18**" causes this leaf node to contain only one value, which is **underfull.**
First, try **merge** with its sibling node, **which sibling should be merged?**



Example 3

- Delete **18**

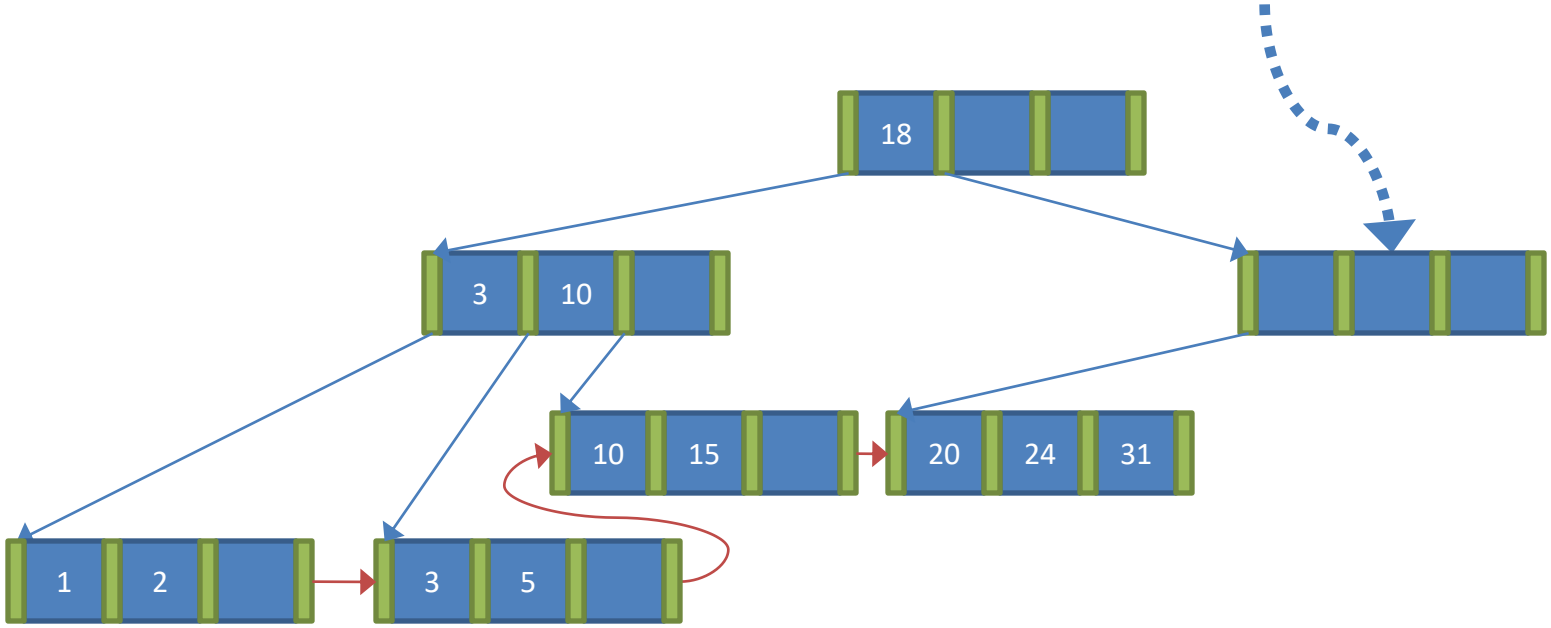
After merging, this leaf node is empty and **no longer used**.



Example 3

- Delete **18**

Now this node has only one pointer, which is **underfull (1 pointer only)**. We try merging it with its sibling.

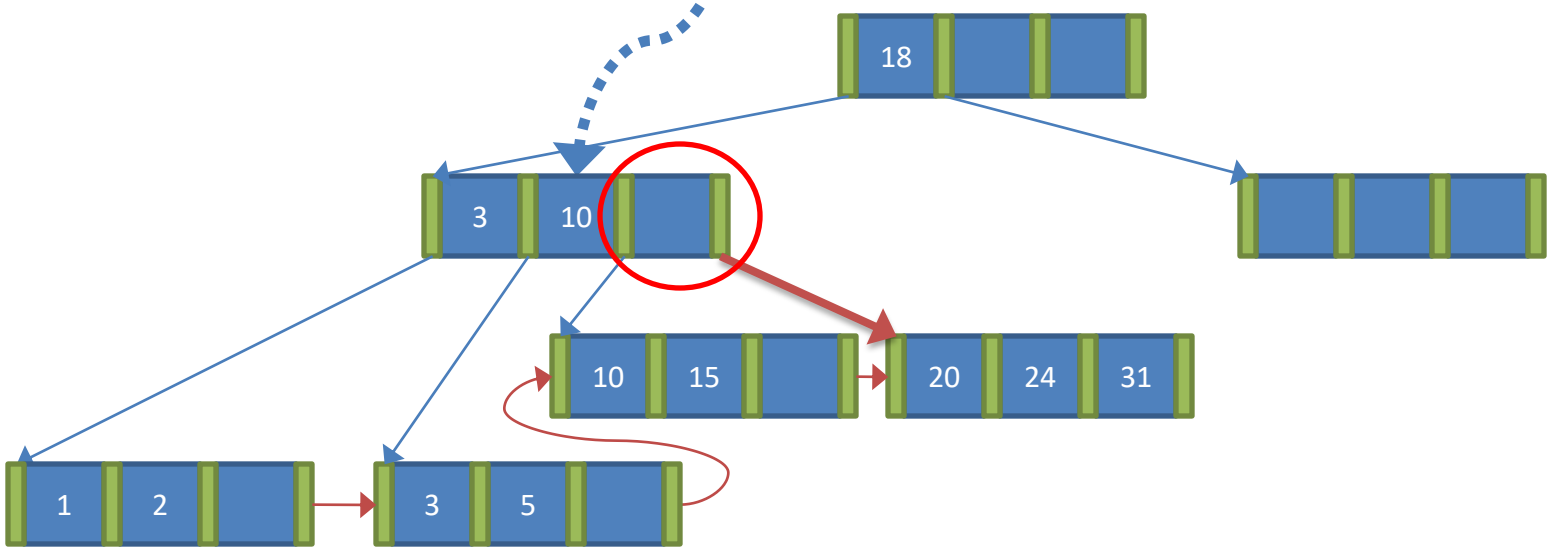


Example 3

- Delete **18**

Merging non-leaf nodes

Step 1. Update the pointers.

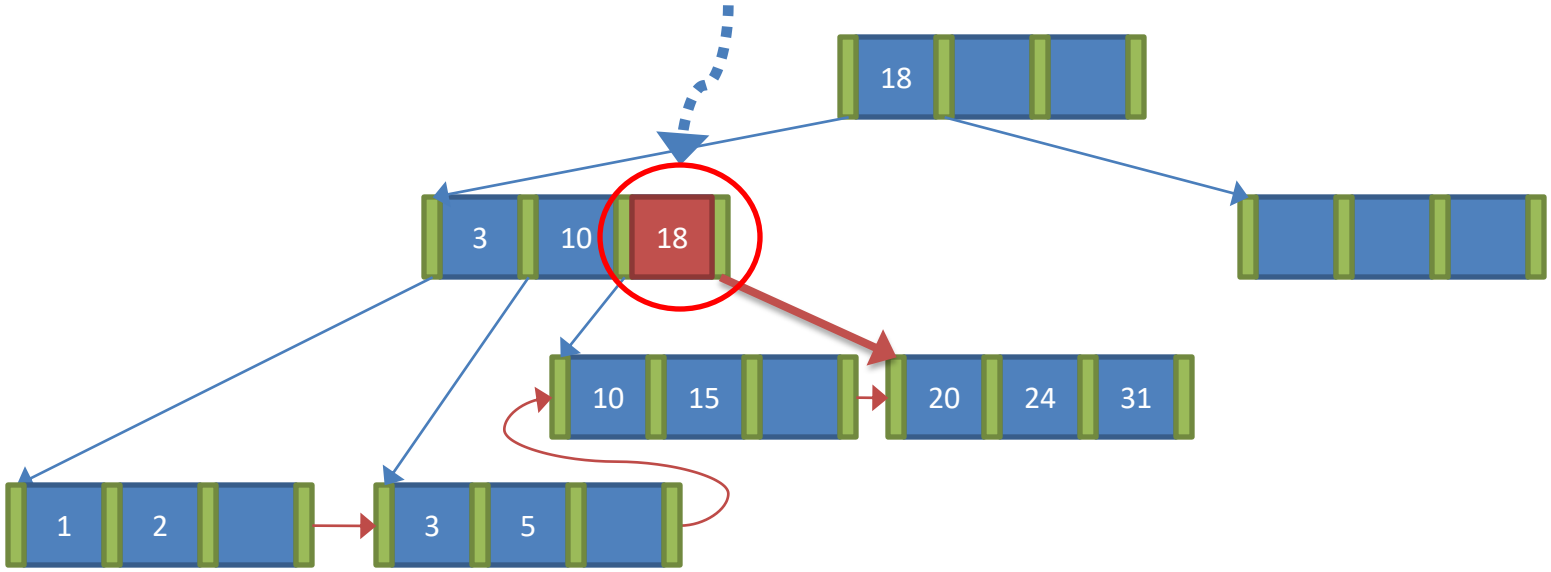


Example 3

- Delete **18**

Merging non-leaf nodes

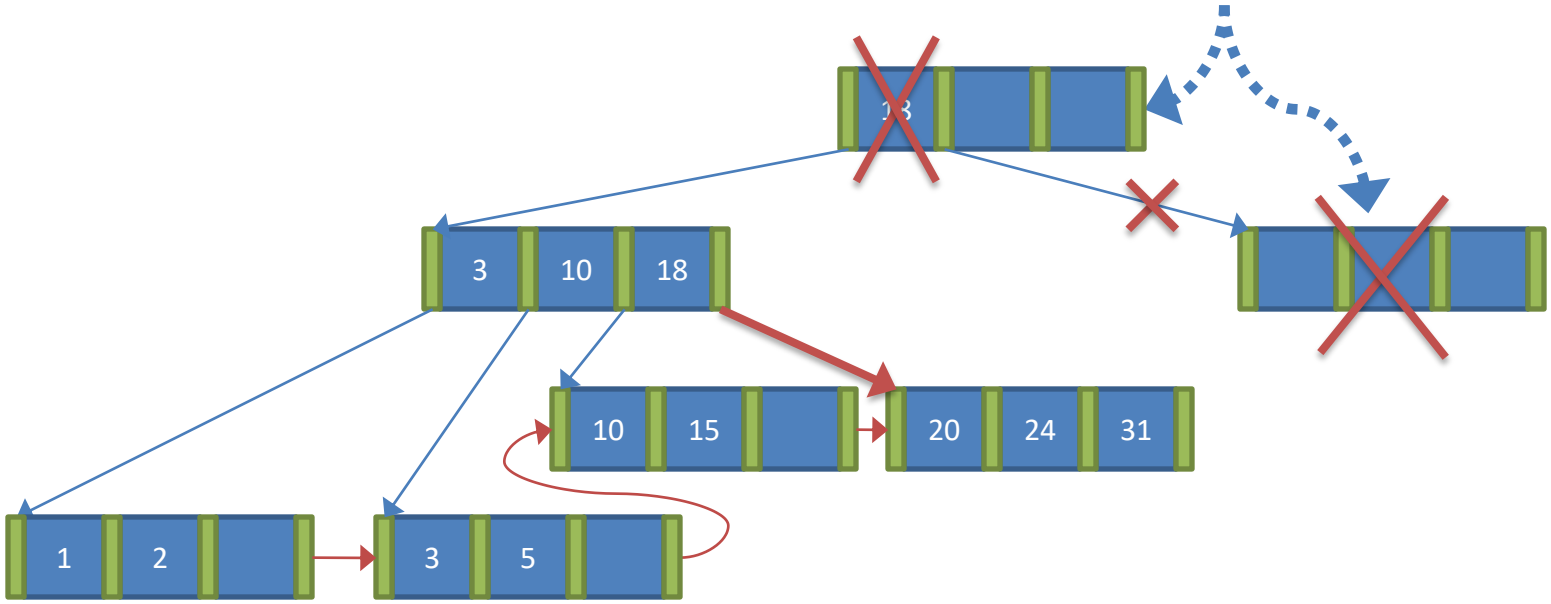
Step 2. Update the keys.
(It is "**18**" as originally it is the key "**18**" in the root node that separate the two pointers.)



Example 3

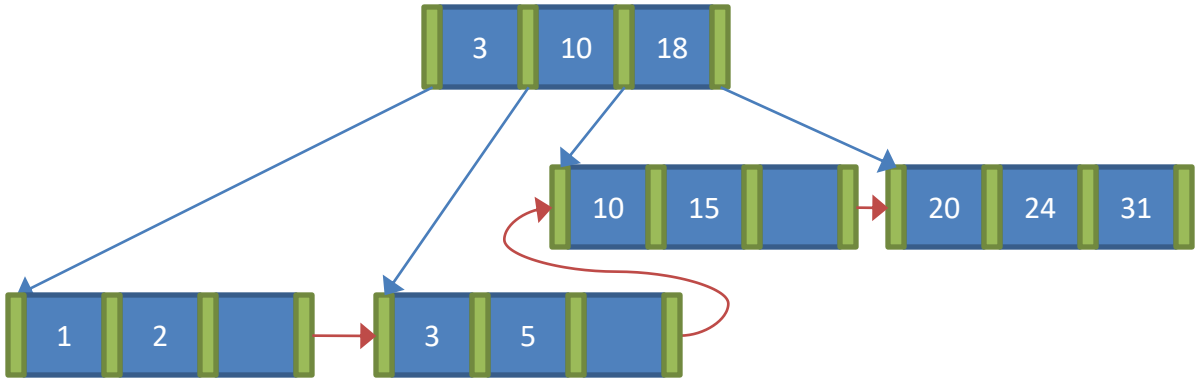
- Delete **18**

Note that since we merged the non-leaf node, some pointers and parent entries can be removed.



Example 3

- Delete **18**



DONE

Defining index in SQL

- To create an index:

  CREATE INDEX <index-name> **ON** <relation-name> (<attribute-list>) [***index_type***]

  - Optional [***index_type***]: **USING** {BTREE | HASH}

- Use **CREATE UNIQUE INDEX** to indirectly specify and enforce the condition that the search-key is a superkey.

- To remove an index **DROP INDEX** <index-name>

Reference: MySQL 13.1.8 CREATE INDEX Syntax http://dev.mysql.com/doc/refman/5.0/en/create-index.html