# Tree Structures (v5)

Juan Manuel Gimeno Illa

2025/2026

## Bibliography

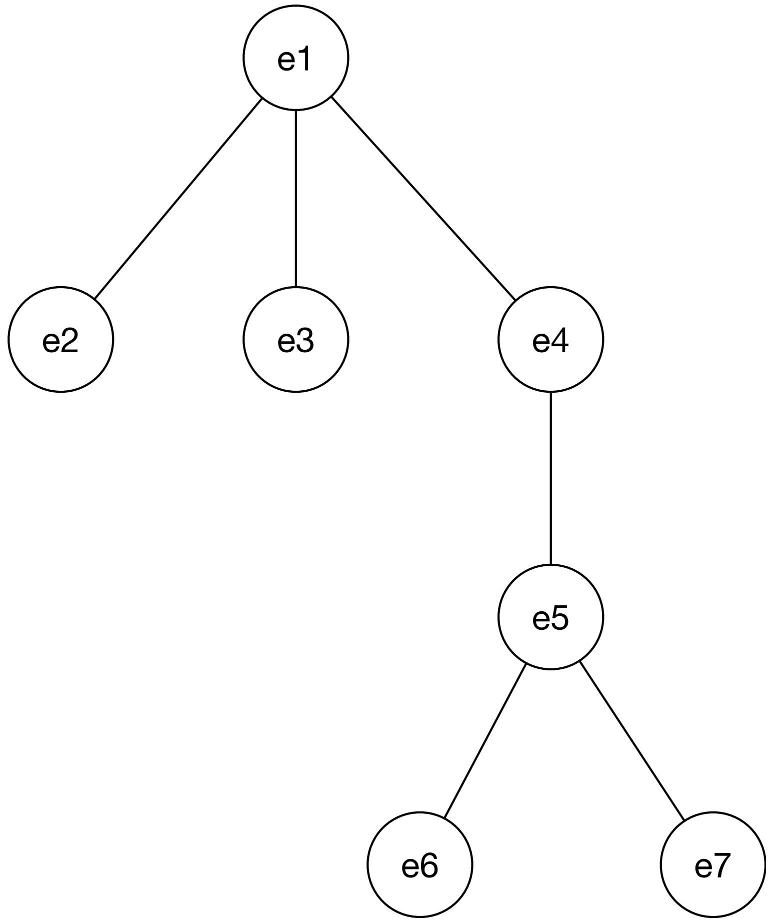
- Robert Sedgewick & Kevon Wayne, Algorithms - Fourth Edition, Pearson Education Inc. (2011)
  - 3.3 (BSTs, 23-Trees and RB-Trees), 6 (B-Trees)
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein, Introduction to Algorithms – Fourth Edition, Massachusetts Institute of Technology (2022)
  - 6.1,10.3, 18 (a variant of B-trees that splits descending), B.5
- Josep Maria Ribó, Apropament a les estructures de dades des del programari lliure. Edicions de la Universitat de Lleida. 2018.
  - Chapter 3, 6 (specially B-trees)

## Importance of tree structures

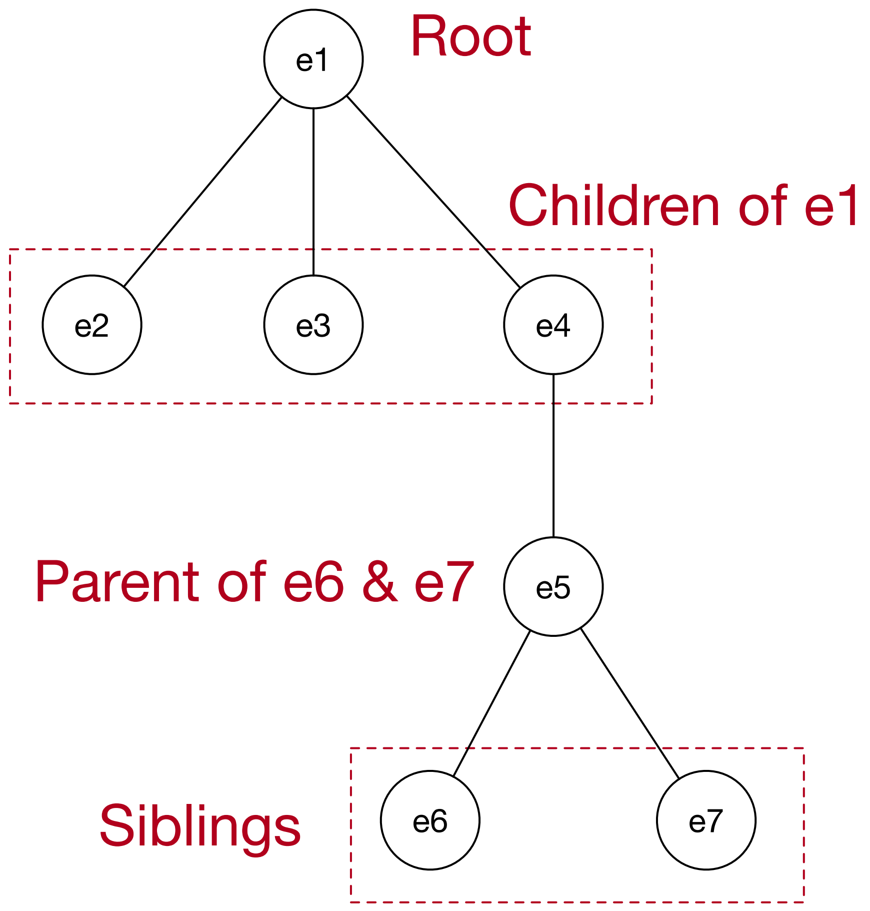
- In computer science, and in data structures in particular, trees are one of the most important structures
- Even that, in most data libraries, we don’t find an interface, or a direct implementation, of a data structure named tree
- Why?
  - Because the tree is used as the **representation** (implementation device) used by other data structures to be **efficient**.
  - For example, PriorityQueue uses a heap, which is a type of tree implemented with an array

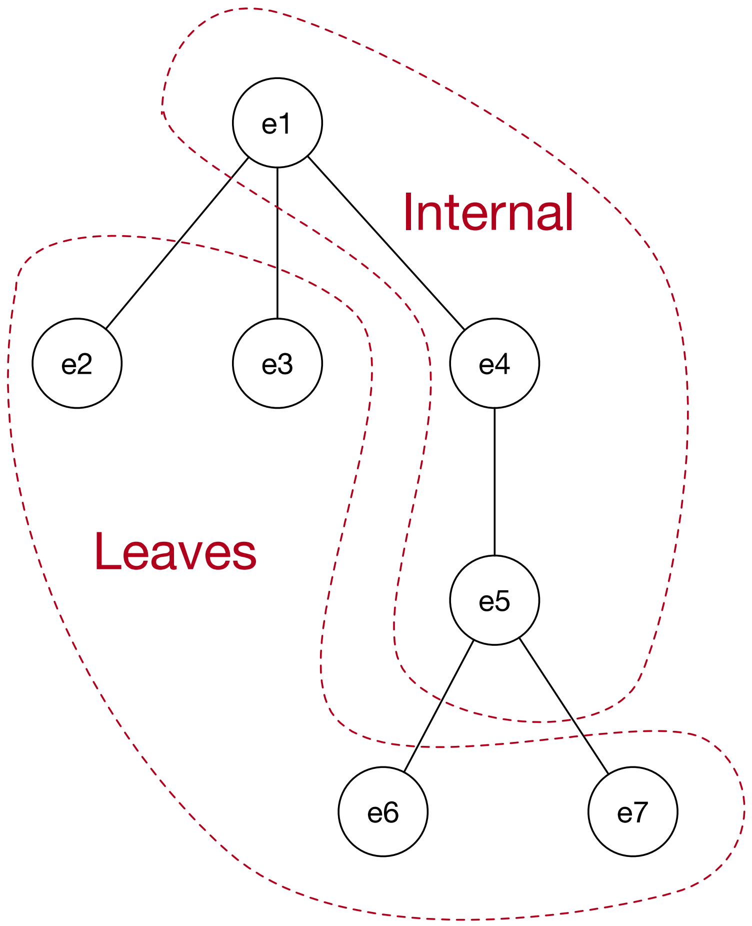
# A definition of Trees

- A **tree**, as a data structure, is a **container of elements** of the same type
  - It can be **empty** (with no elements)
  - Or be a composition of
    - An element (the **root** of the tree)
    - Zero or more **disjoint** (sub)trees named its **children**
- **NOTE:** Usually, empty subtrees are not shown.
- **NOTE:** Sometimes we conflate the nodes of the tree with its elements



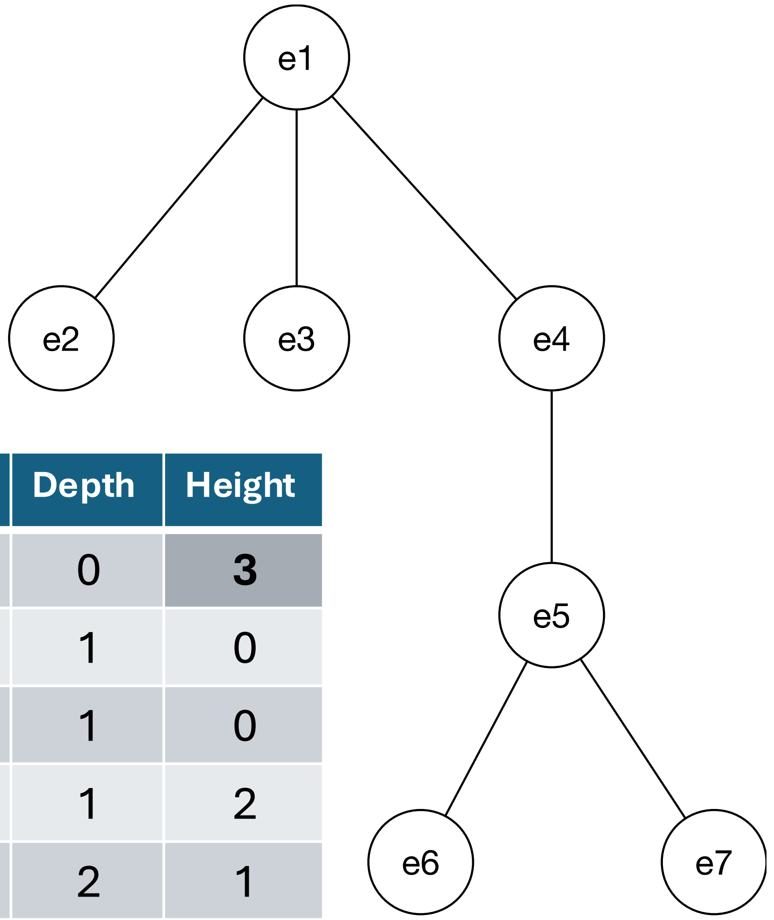
# Names, names and more names





# Names, names and more names

- The **degree of a node** is the number of non-empty children
- The **depth of a node** is the number of edges from the root to the node.
    - **Level k** is the set of nodes at **depth k**
- The **height of a node** is the number of edges from the node to the deepest leaf.
- The **height of a tree** is the **height of its root.**
    - It's equal to the **max depth** of any node
    - An **empty tree** has height **-1**
- The **size of a tree** is the number of nodes



| Level | Node | Degree | Depth | Height |
|---|---|---|---|---|
| 0 | e1 | 3 | 0 | **3** |
| 1 | e2 | 0 | 1 | 0 |
|  | e3 | 0 | 1 | 0 |
|  | e4 | 1 | 1 | 2 |
| 2 | e5 | 2 | 2 | 1 |
| 3 | e6 | 0 | 3 | 0 |
|  | e7 | 0 | 3 | 0 |

## Trees and recursion

- As the very definition of tree is **recursive**, most of the time, the natural and simplest way to program methods over trees is **recursion**
  1. We must define a **base** case, i.e. a case that can be solved **without recursive calls**.
     Usually, the base case is the empty tree.
  2. In the **recursive** case we call the same functions over **"smaller"** trees.
     Usually, the recursive calls are made on the children.
  3. To be **efficient**, and this can be difficult sometimes, we should **avoid duplicities** (i.e. calling the same function more than once on the same tree).

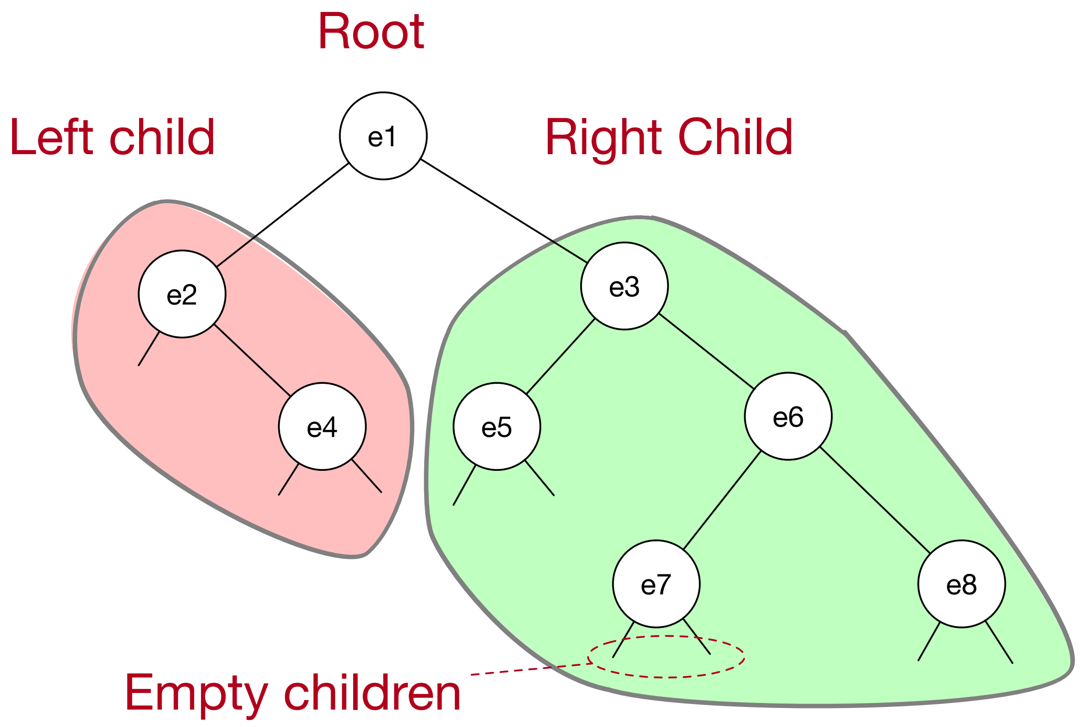
# Types of trees

- **Rooted** trees
  - There is node which is the root of the tree
  - Each node has associated a list of its children (can be empty for leaves)
  - The "position" of each (non-empty) children in the list is not relevant
- **Ordered** trees are rooted trees in which
  - Children have an order: first child, second child, ...
  - But if the second disappears, the third occupies its place and so on
- **Positional** trees are ordered trees in which
  - Each child has a definite position
  - So, if the second disappears, its occupied by an empty child
- **N-ary** trees are positional trees in which
  - Any node has at-most N children (some of them can be empty)
- The most important trees are **binary trees** which are **2-ary trees**

## Binary Trees

# Binary tree

- A **binary tree** is a container of elements of **type E** that can be:
  - An **empty** tree, that is, without elements
  - A **non-empty** tree composed by
    - An element of type E named its **root**
    - Two disjoint binary subtrees (which can be empty) named **left child** and **right child**



**NOTE**: Usually, empty children are not shown

## Properties of binary trees

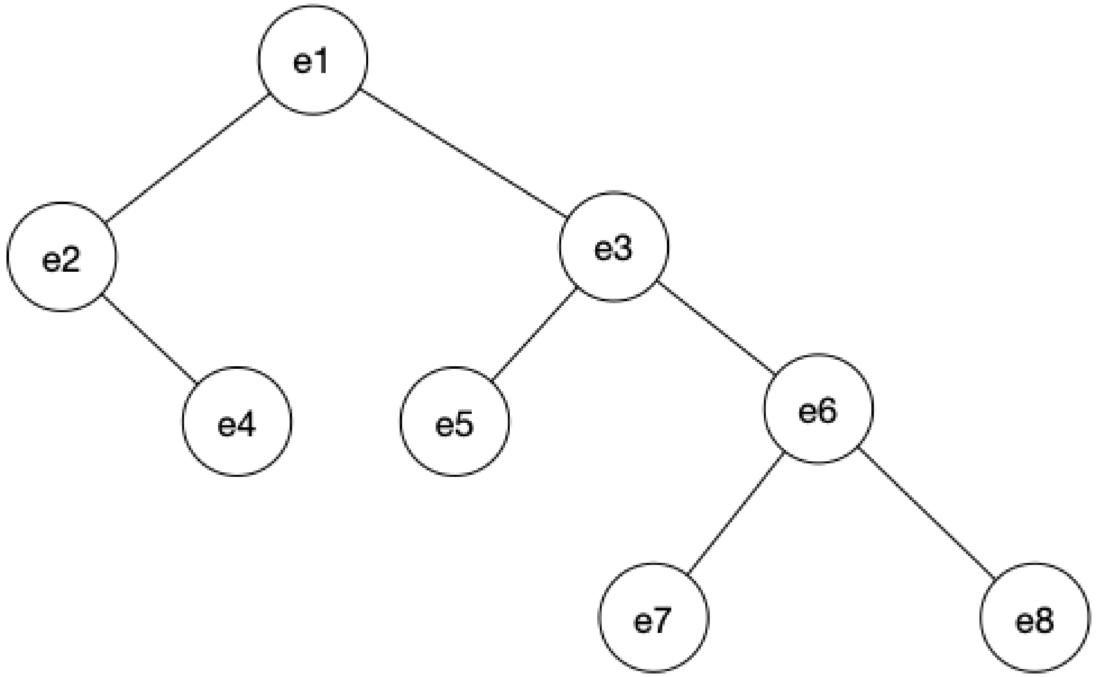
- Binary trees have some **properties** that can be easily **proved by induction**.
- *The maximum number of elements at level $k$ is $2^k$*
  - **Base case**:
    - the root, which is at level 0 has a maximum number of elements of $2^0 = 1$, which is true
  - **Inductive case**:
    - the maximum at level $k+1$ is achieved by adding two children to each of the nodes at level $k$ which, by induction hypotheses, has $2^k$ maximum nodes
    - so, the maximum number at level $k+1$ is $2 * 2^k = 2^{k+1}$
    - QED

## Properties of binary trees

- *The maximum number of elements in a tree of height $h$ is $2^{h+1} - 1$*
  - **Base case**:
    - An empty tree has heigh -1 and has $2^{-1+1} - 1 = 2^0 - 1 = 1 - 1 = 0$ elements
  - **Inductive case**:
    - The maximum number of elements in a tree of heigh $h + 1$ is accomplished by combining a root and two trees of maximum elements of height $h$
    - This tree has $1 + 2 * \left(2^{h+1} - 1\right) = 1 + 2^{h+2} - 2 = 2^{h+2} - 1$ elements
    - QED

## Properties of binary trees

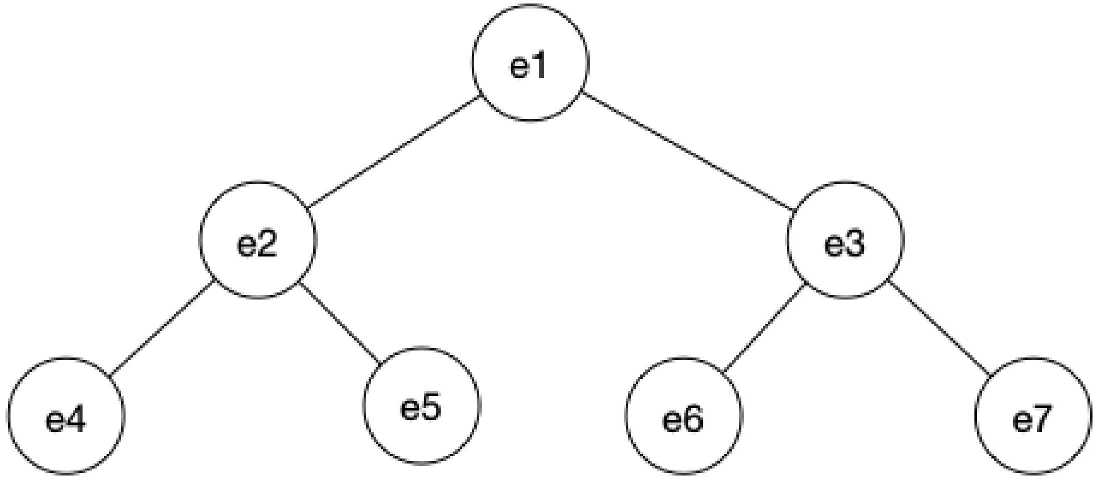
- *The number of leaves (degree 0) in a non-empty binary tree is one more than the nodes of degree 2 (i.e. with two children)*



**Leaves**: {e4, e5, e7, e8}
**Degree 2**: {e1, e3, e6}

## Binary trees

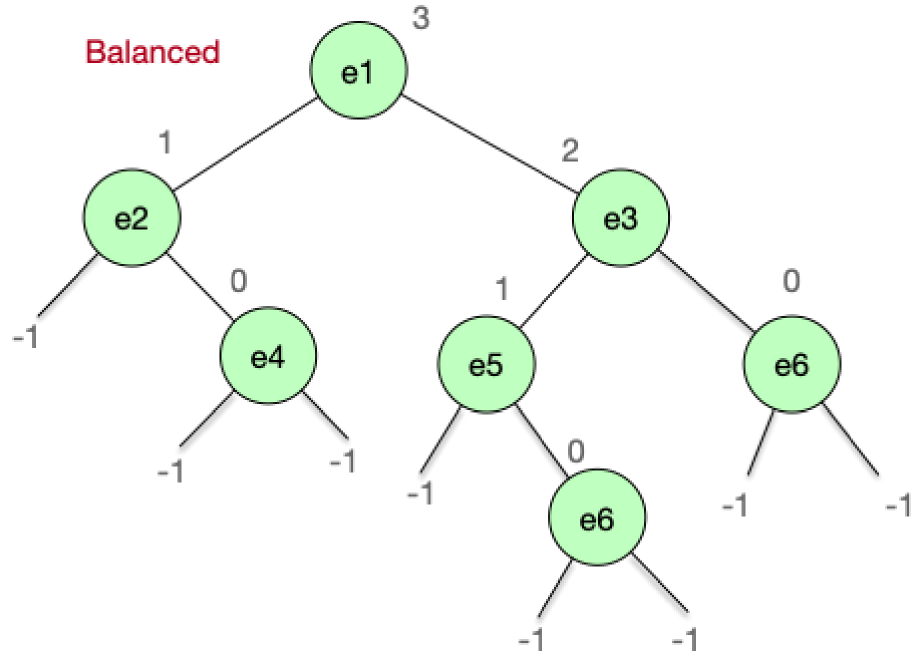
- A **binary tree** is said to be **complete** if
  - It has the maximum number of nodes for its height
- In a **complete binary tree**
  - There are $2^h$ leaves
  - **The height of a complete tree of size $\boldsymbol{n}$ is $\boldsymbol{\Theta(log_2\, n)}$**
  - There are $2^h - 1$ internal nodes
- **NOTES**:
  - This can be generalized easily to k-ary trees
  - Some authors consider that the last level may not be complete, but with all nodes to the left

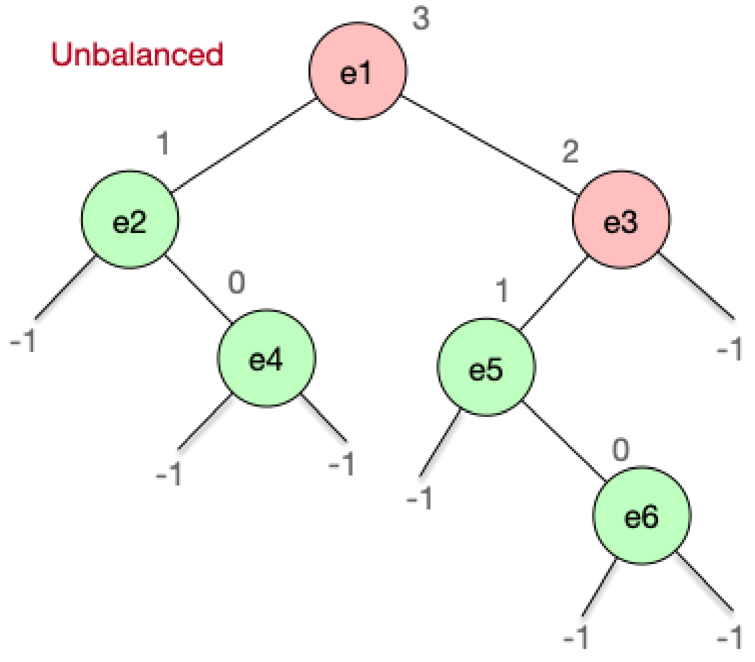


**Height** = 2
**Leaves** = $2^2 = 4$
**Internal** = $2^2 - 1 = 4 - 1 = 3$

# Binary trees

- A **binary tree** is said to be **balanced** when, for each node the heights of its left and right child differ in at most 1 and both children are balanced
- The **height of a balanced binary tree of size $\boldsymbol{n}$ is $\boldsymbol{\Theta(\log_2 n)}$**
- **NOTE**: This property also generalizes to k-ary trees.



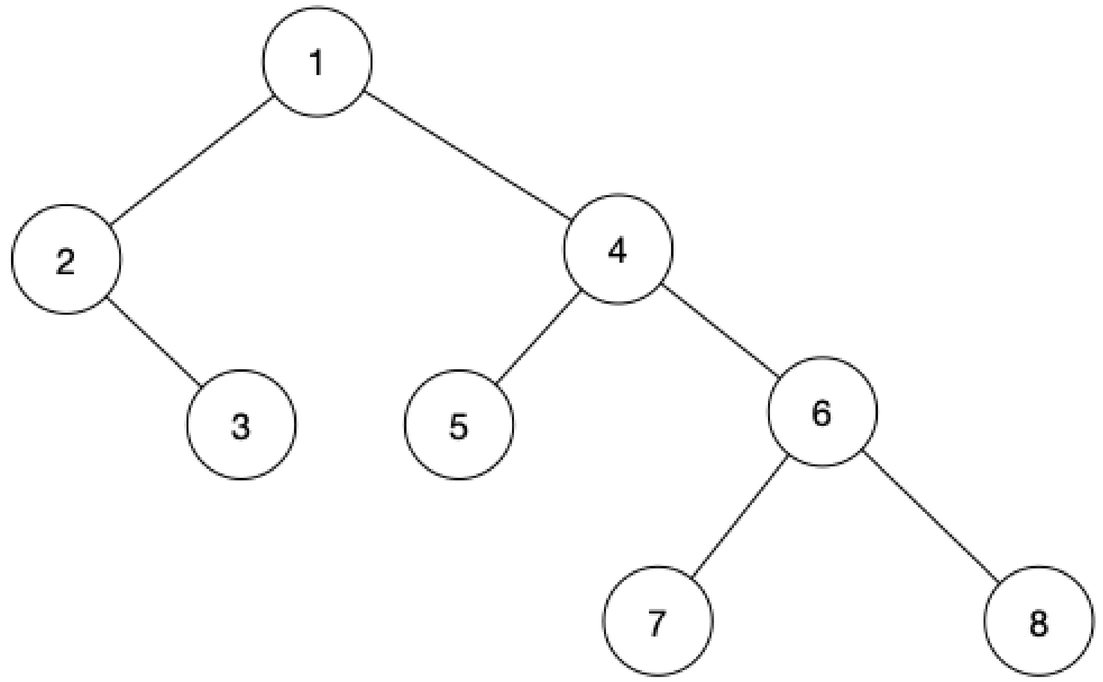


## Binary tree traversals

- One of the things we can do with a tree is **traverse** it
  - That is, obtaining a sequence (usually represented as a List) of all the elements in the tree
- But, in which order?
  - Unlike with sequential data structures, here each element has no definite position (i.e. the first element, the second element, …)
- But there are four "natural" orders
  - Three *recursive* (or in-depth) traversals named: **pre**-order, **in**-order and **post**-order, which are defined recursively and differ in when the **root** node is visited
  - A fourth *non-recursive* traversal named **level**-order

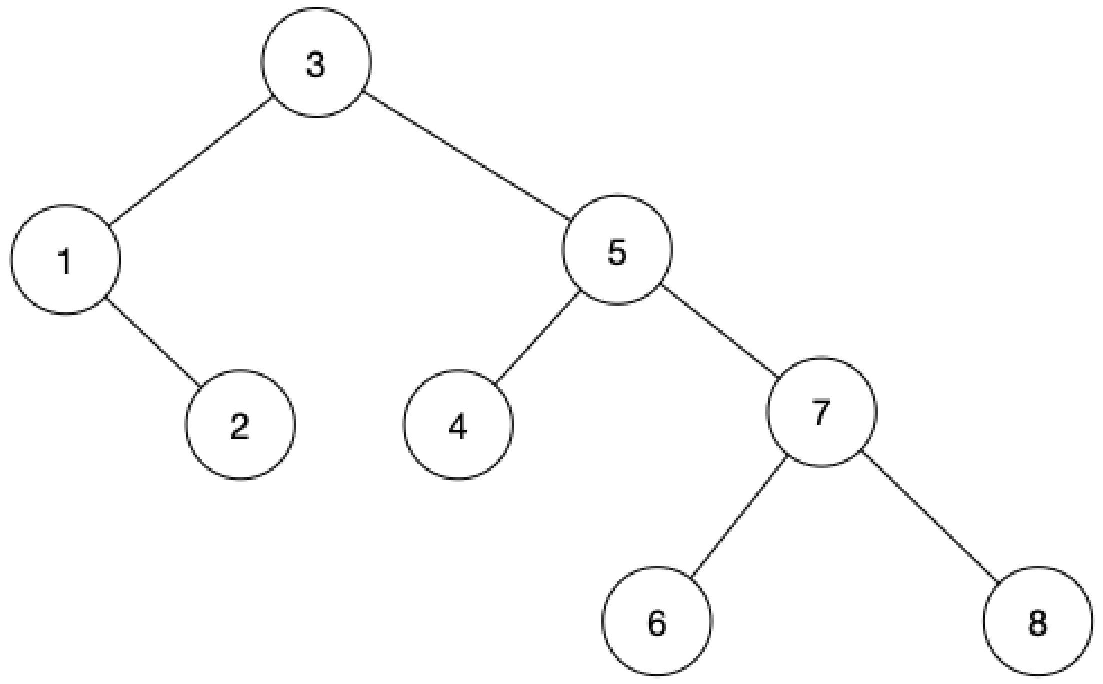
# Binary tree traversals

- **Pre-order**:
  1. the **root** of the tree
  2. the left child in pre-order
  3. the right child in pre-order



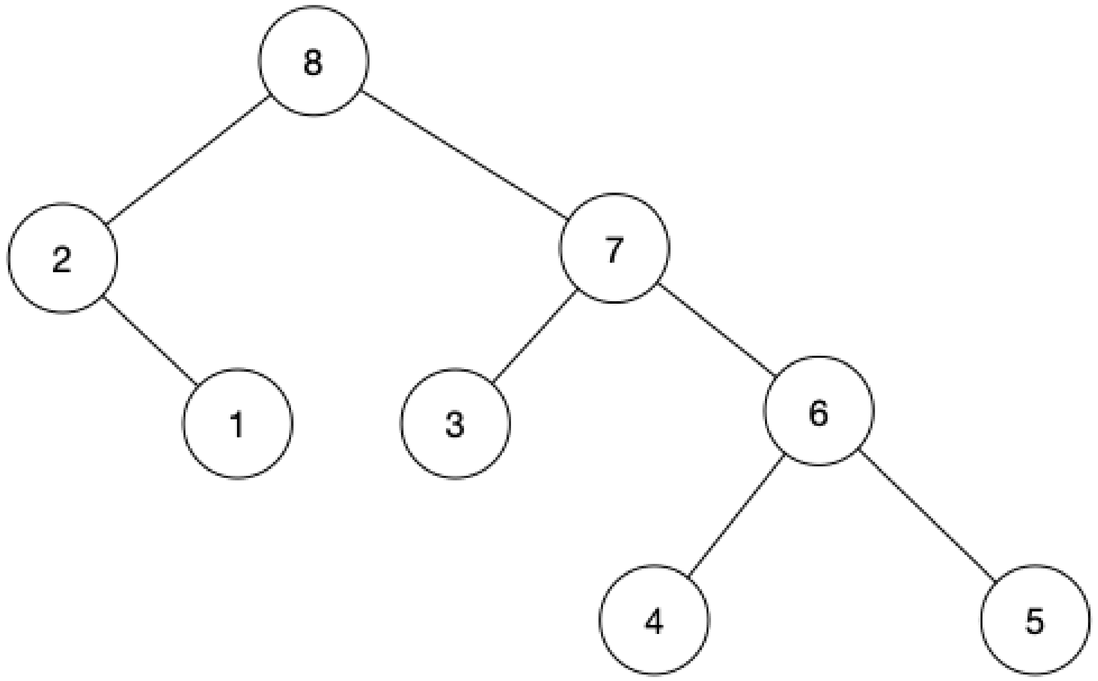
## Binary tree traversals

- **In-order:**
  1. the left child in in-order
  2. the **root** of the tree
  3. the right child in in-order



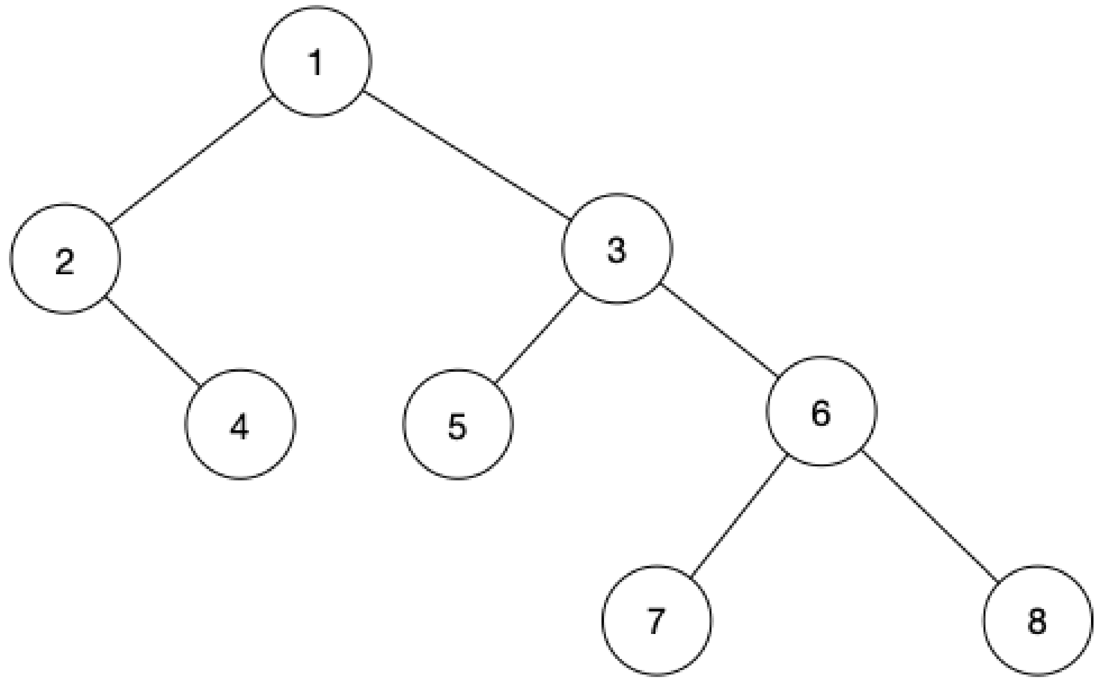
## Binary tree traversals

- **Post-order:**
  - The left child in post-order
  - The right child in post-order
  - The **root** of the tree



## Binary tree traversals

- **Level-order:**
  - First level 0
  - Then level 1
  - Then level 2
  - …
  - (each level from left to right)

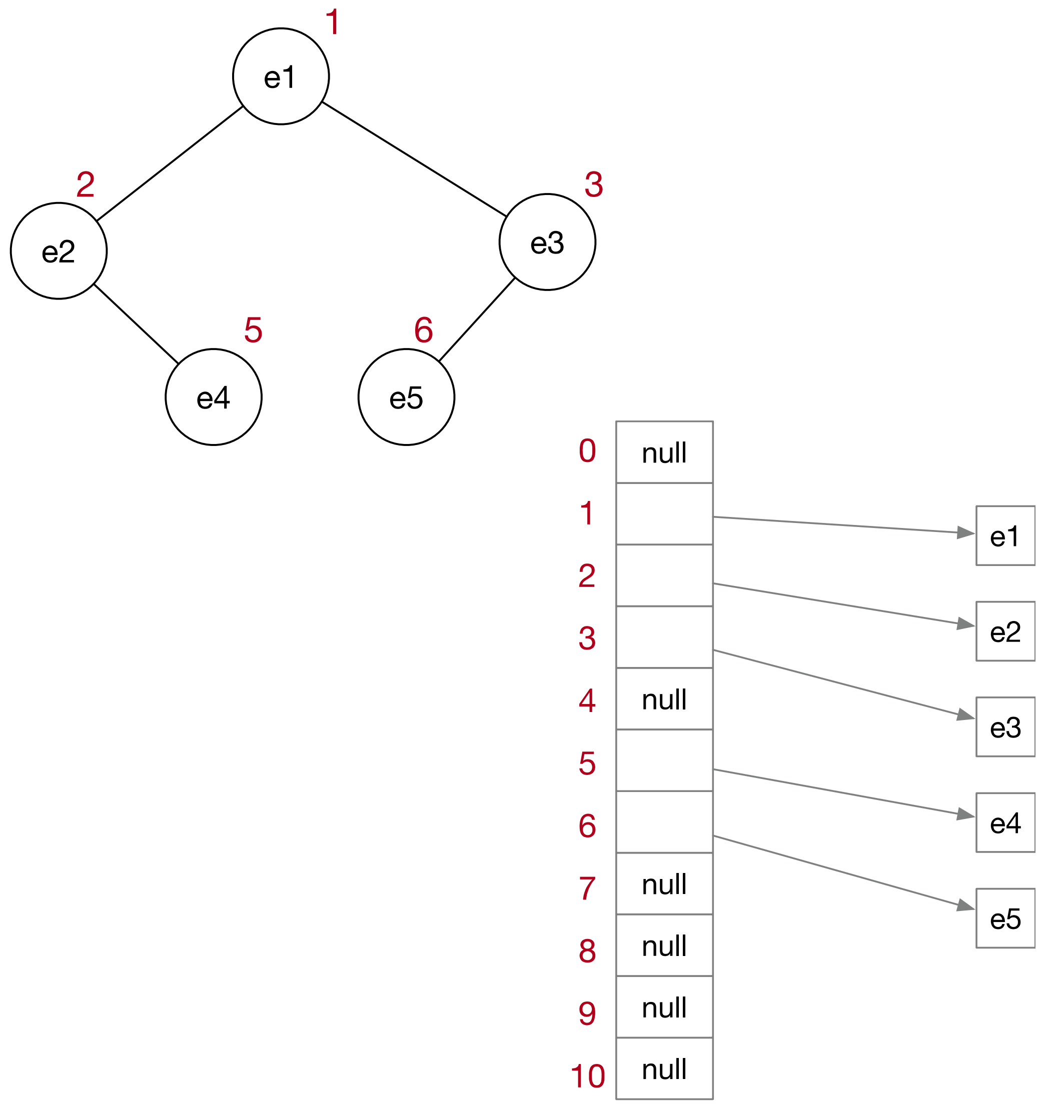


## Binary trees

- The first representation we will consider, simply using an **array**
  - But, as we will see, it is **only reasonable in a limited number of cases**
  - Mainly to be used in the implementation of **heaps** and in the **heap-sort** algorithm and in the implementation of **priority queue**
- **In some cases**, as in the examples we will show, the **0-index** position in the array is **not used** (to be able to implement faster arithmetic)
  - For example, in priority queues some implementations ignore the 0-index position
  - But, once you’ve understood the principles behind the design, moving from one implementation to the other is easy

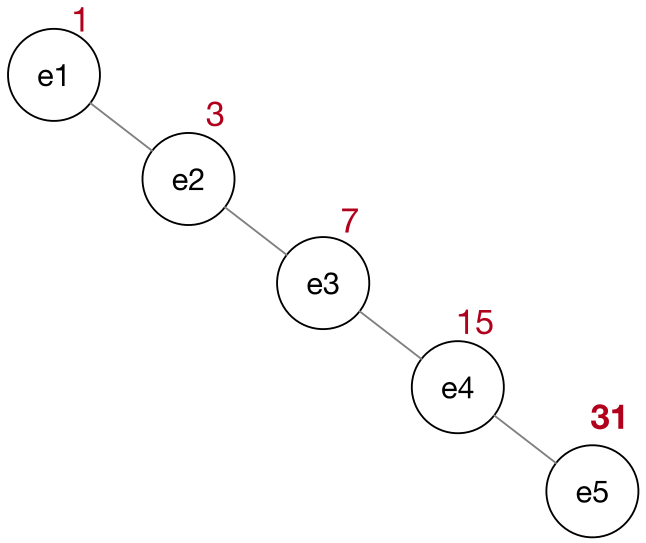
# Binary Trees

- **Root** is at index **1**
- For a ***node*** at index $\boldsymbol{i}$
    - **Left** child: at $\mathbf{2} * \boldsymbol{i}$
    - **Right** child: at $\mathbf{2} * \boldsymbol{i} + \mathbf{1}$
- How do we know if the **child exists?**
    - Because the **index** is **out** of the **array**; or the **value** at its index is ***null***
    - So, in this implementation, **null** values are **not allowed** for the **elements**



# Binary Trees

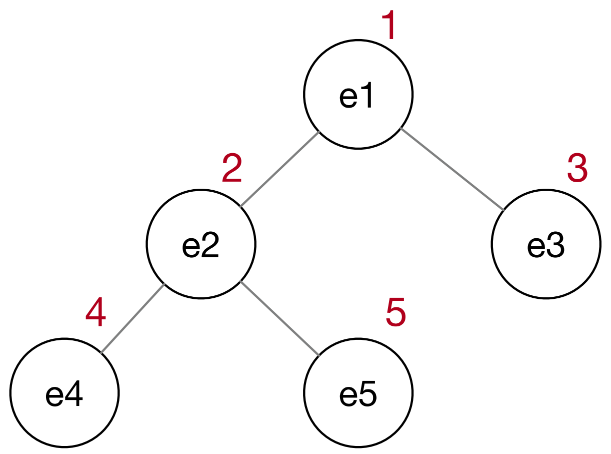
- Why is this implementation not valid in the general case?
  - Because for tree with $n$ nodes, it may need an array of size $2^n$



You need an array of $2^5 = 32$ positions !!
- and only 5 of them won't be null

## Binary Trees

- But, for trees that are complete (even allowing for an incomplete last level of nodes aligned to the left), it is a very compact and efficient implementation
  - Amortizing resizing operations on the array

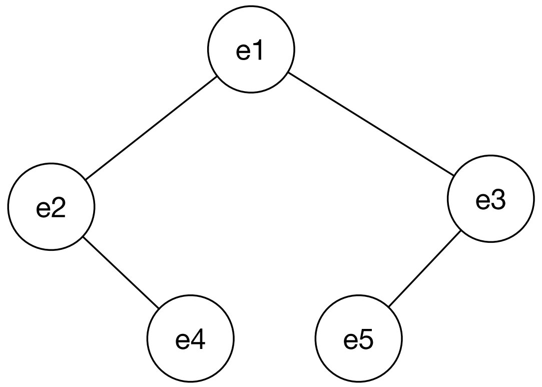


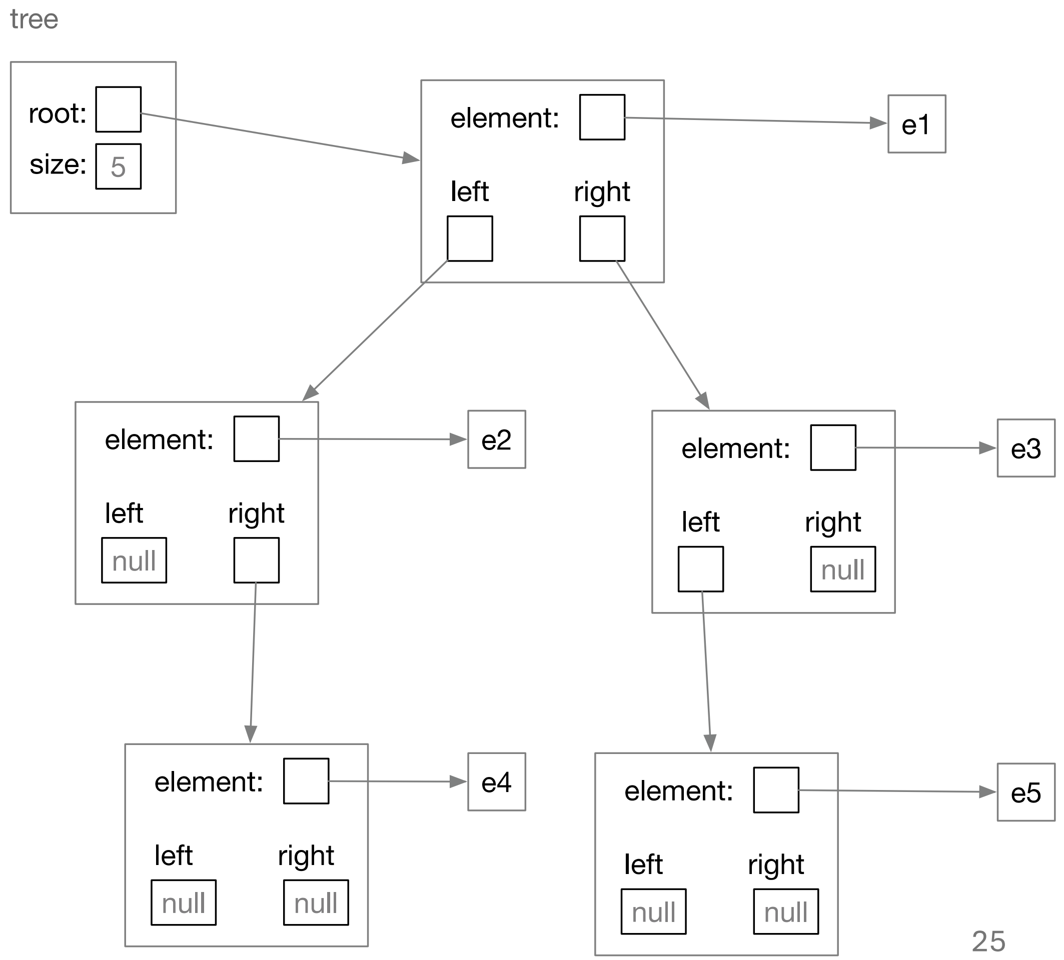
You need an array of only $5 + 1 = 6$ positions !!

- and only 1 of them is null

# Binary Trees

- The most used representation of trees uses a ***linked structure of nodes***
- A tree has a reference to the node that represents its root
    - **null** if the tree is **empty**
- Each node, apart from the **element** it stores, has two references:
    - one to the **left child**
    - one to the **right child**
- Extra information, such as tree **size**, can be **stored** in the **tree**





## Binary Trees

- As we have said before, trees are used mostly to get efficient implementation of other data structures
  - So, for instance, the JCF does not contain a data structure that presents an interface about trees
- We will add such an interface, and outline a possible implementation that uses this linked representation
- And, when designing this implementation, we will consider important topic such as
  - using more space to not waste time recomputing things
  - mutability and the necessity of copies
  - the java Clonable interface to make copies

## Binary Trees

- It’s difficult to decide the methods in a BinaryTree interface
  - As we have said, most trees are used as implementation devices for other types
- We have selected some methods that will allow us to comment some trade-offs in the implementation

```
public interface BinaryTree<E> {

   E root();
   BinaryTree<E> left();
   BinaryTree<E> right();

   default boolean isEmpty() {
       return size() == 0;
   }


   int size();
   int height();

   E  replaceRoot(E e);
   void removeLeft();
   void removeRight();

   List<E> preOrder();
   List<E> inOrder();
   List<E> postOrder();
   List<E> levelOrder();
}
```

## Binary Trees

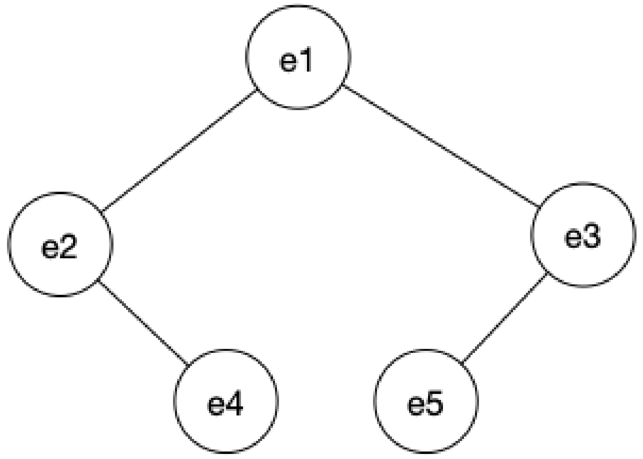
- The first important decision we have made is to define the trees as **modifiable**
  - Once a tree is created, we can in-place replace its root or delete (made empty) any of its children
- In the implementation of this type
  - We want the implementations of the **constructors** and **accessors** to be **efficient**, that is, O(1)
  - And for the two properties
    - **Size**: needs to be **efficient**
    - **Height**: **no** need of efficiency

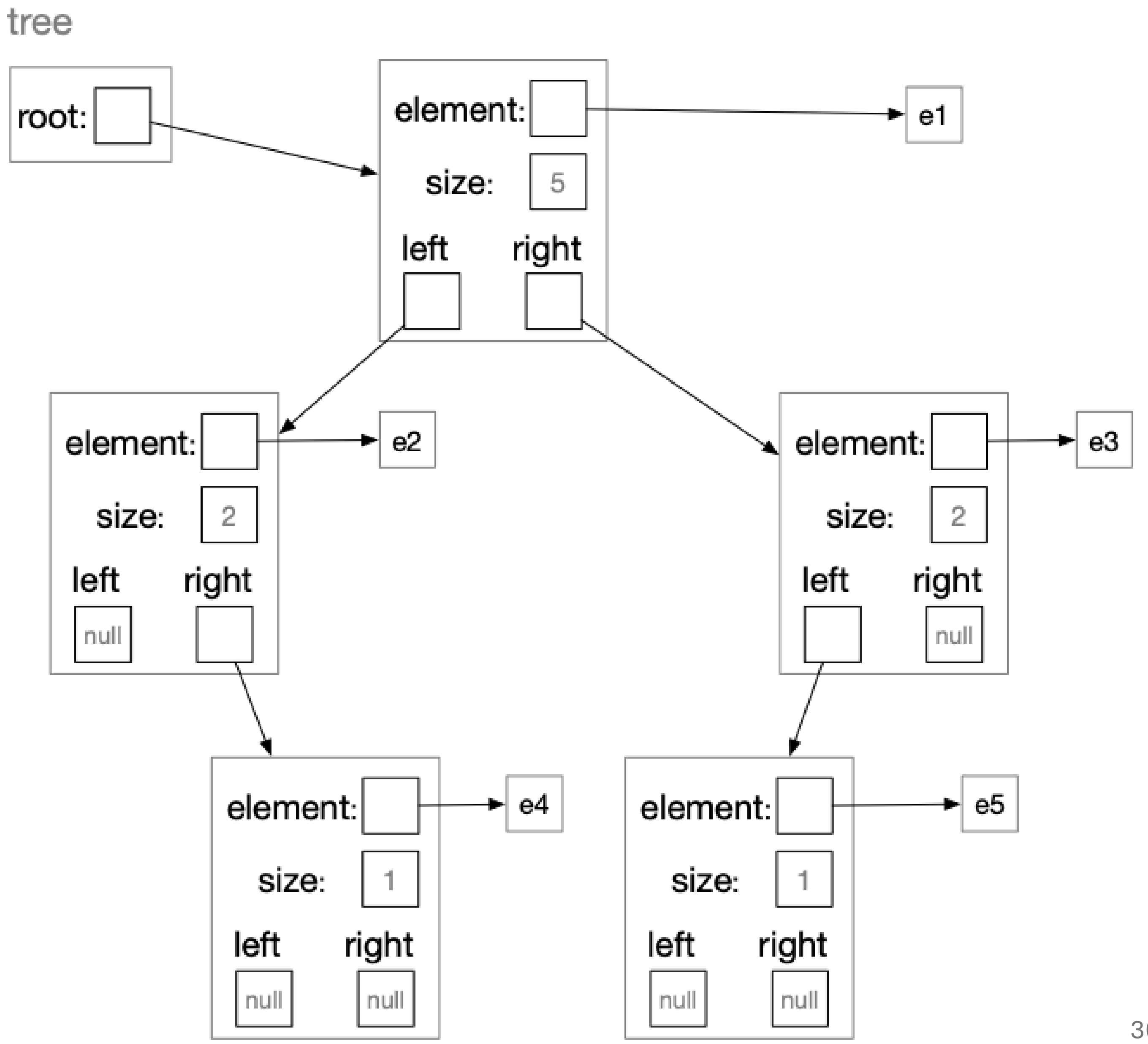
```
public interface BinaryTree<E> {

   E root();
   BinaryTree<E> left();
   BinaryTree<E> right();

   default boolean isEmpty() {
      return size() == 0;
   }


   int size();
   int height();

   E  replaceRoot(E e);
   void removeLeft();
   void removeRight();

   List<E> preOrder();
   List<E> inOrder();
   List<E> postOrder();
   List<E> levelOrder();
}
```

## Binary Trees

- The **representation** shown before can be **analysed** given the **efficiency restrictions** we have outlined
- **PROS**:
  - Having the size precomputed in the tree, gives us O(1) for the size property
- **CONS**:
  - But, e.g. returning the left child is O(n), where n is the size of the left child, because the size of the subtree must be computed by traversing it
- So, the first implementation decision is to make the **size** to be **cached** at the **nodes**
- NOTE: It’s very important to be able to **reason** about this kind of things **before even writing a single line of code**

## Binary Trees





```java
public class LinkedBinaryTree<E> implements BinaryTree<E > {

  private Node<E> root;

  private static class Node<E> {
    Node<E> left;
    E element;
    Node<E> right;
    int size;

    Node(Node<E> left, E element, Node<E> right) {
      this.left = left;
      this.element = element;
      this.right = right;
      this.size = 1 + Node.size(left) + Node.size(right);
    }

    static int size(Node<?> node) {
      return node == null ? 0 : node.size;
    }
  }
```

```java
// Constructors

public LinkedBinaryTree() { root = null; }

public LinkedBinaryTree(
    LinkedBinaryTree<E> left,
    E elem,
    LinkedBinaryTree<E> right) {

  Node<E> leftChild = left == null ? null : left.root;
  Node<E> rightChild = right == null ? null : right.root;
  root = new Node<>(leftChild, elem, rightChild);
}

private LinkedBinaryTree(Node<E> root) {
  this.root = root;
}

// ...
// Accessors

@Override
public E root() {
   if (root == null)
      throw new NoSuchElementException("root of empty tree");
   return root.element;
}

@Override
public LinkedBinaryTree<E> left() {
   if (root == null)
      throw new NoSuchElementException("left child of empty tree");
   return new LinkedBinaryTree<>(root.left);
}

@Override
public LinkedBinaryTree<E> right() { … }
```

```
// Properties

@Override
public boolean isEmpty() {
   return root == null;
}

@Override
public int size() { return Node.size(root); }

@Override
public int height() { return Node.height(root); }
```

```
// Node.height implementation
static int height(Node<?> node) {
   if (node == null)
      return -1;
   else
      return 1 + Math.max(height(node.left), height(node.right));
}
```

```java
// Modifiers

@Override
public E replaceRoot(E newElement) {
   if (root == null)
      throw new NoSuchElementException("the empty tree has no root to replace");
   E oldElement = root.element;
   root.element = newElement;
   return oldElement;
}

@Override
public void removeLeft() {
   if (root == null)
      throw new NoSuchElementException("Empty tree");
   root.size -= Node.size(root.left);
   root.left = null;
}

@Override
public void removeRight() { ... }
```

```
// Traversals

@Override
public List<E> preOrder() {
   List<E> result = new ArrayList<>(size());
   if (root != null)
      root.preOrder(result);
   return result;
}

// Node.preOrder
void preOrder(List<E> result) {
   result.add(element);
   if (left != null)
      left.preOrder(result);
   if (right != null)
      right.preOrder(result);
}
```

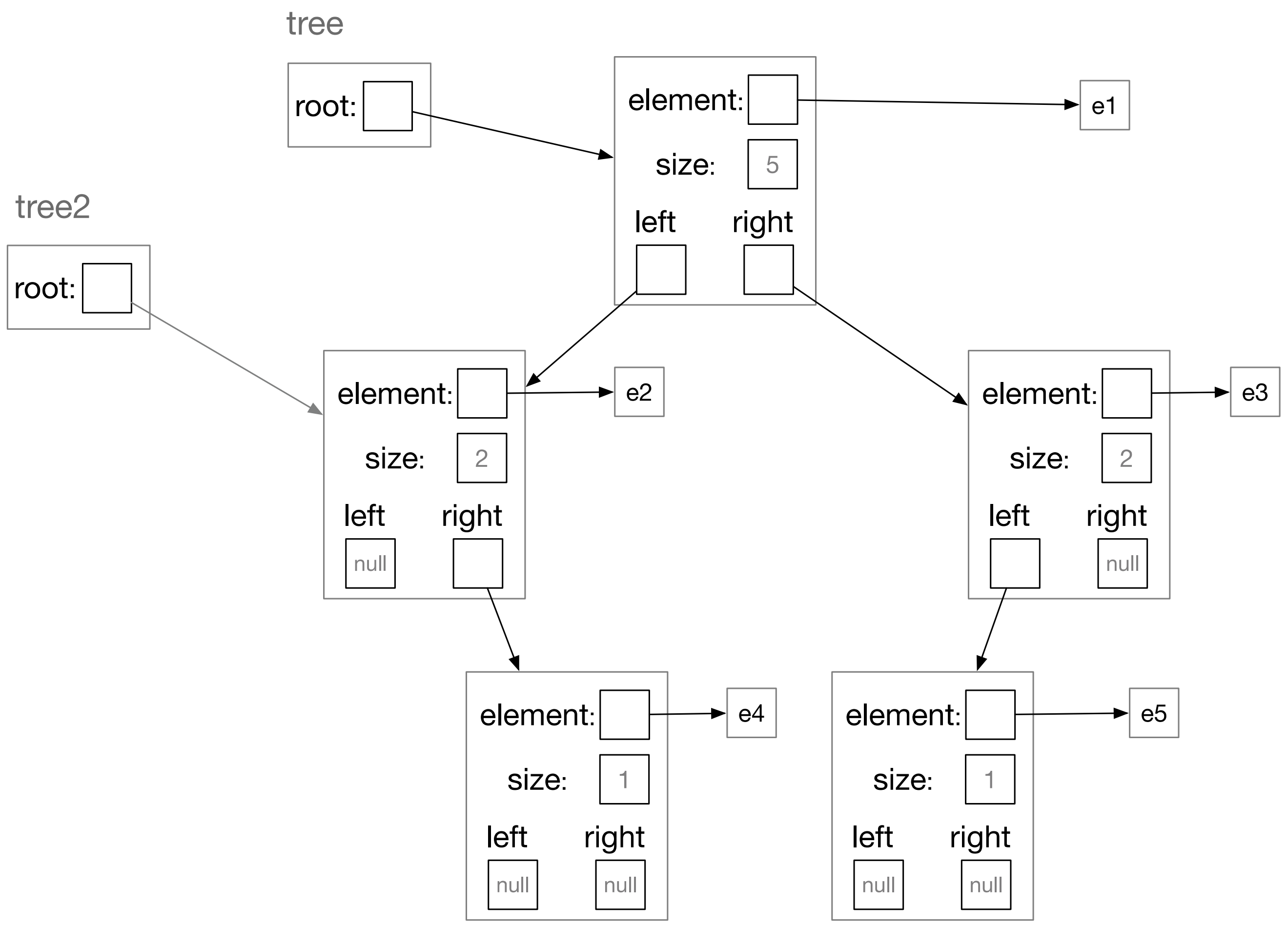
```
// Methods overridden from Object

@Override
public boolean equals(Object o) {
   if (this == o)
      return true;
   if (!(o instanceof LinkedBinaryTree<?> bt))
      return false;

   return Node.equals(root, bt.root);
}

// Node.equals
static boolean equals(Node<?> node1, Node<?> node2) {
   if (node1 == null || node2 == null)
      return node1 == node2;
   else
      return node1.size == node2.size
            && Objects.equals(node1.element, node2.element)
            && equals(node1.left, node2.left)
            && equals(node1.right, node2.right);
}
```

## Binary Trees

- Some comments on the implementation:
  - As we have said we consider that height is not called as often as size, so we do not need to cache it in each node
  - The E in LinkedBinaryTree<E> is not the same E in Node<E> because the later is static, but it's customary to use the same letter
  - The auxiliary methods in class Node are mostly static cause they deal with null references
    - If they were non-static the client code must deal will the nulls
  - The traversal method creates the list for the result, and this list is shared by all the recursive calls
    - If each recursive called returned a list, most of the execution time would be dedicated to copying lists

## Binary Trees

- To make the left() operation O(1), the resulting tree and the original one share nodes
- This wouldn't be a problem if the trees and the elements were **unmodifiable**
- This is not the case here:
    - **due to operations on the tree**
    - or to operations on the elements (we cannot control them, because we do not know what E is)



## Binary Trees

- If we want to protect the users from the problems derived from sharing, we could
    - make a copy of the whole left tree and return a new tree pointing to it
    - but this would be O(n), where n is the size of the left tree
    - and this could be not needed in all cases !!
- What can we do?
    - add a copy mechanism that will allow a client of the class to get a copy of a tree that doesn't share nodes with the original one
- NOTE:
    - this won't solve the problems with modifiable elements
    - avoiding this kind of problems, by forbidding modifications, is one of the big advantages of functional programming

## Binary Trees

- One possibility is to add a method named copy, with signature:

```
public LinkedBinaryTree<E> copy() { … }
```

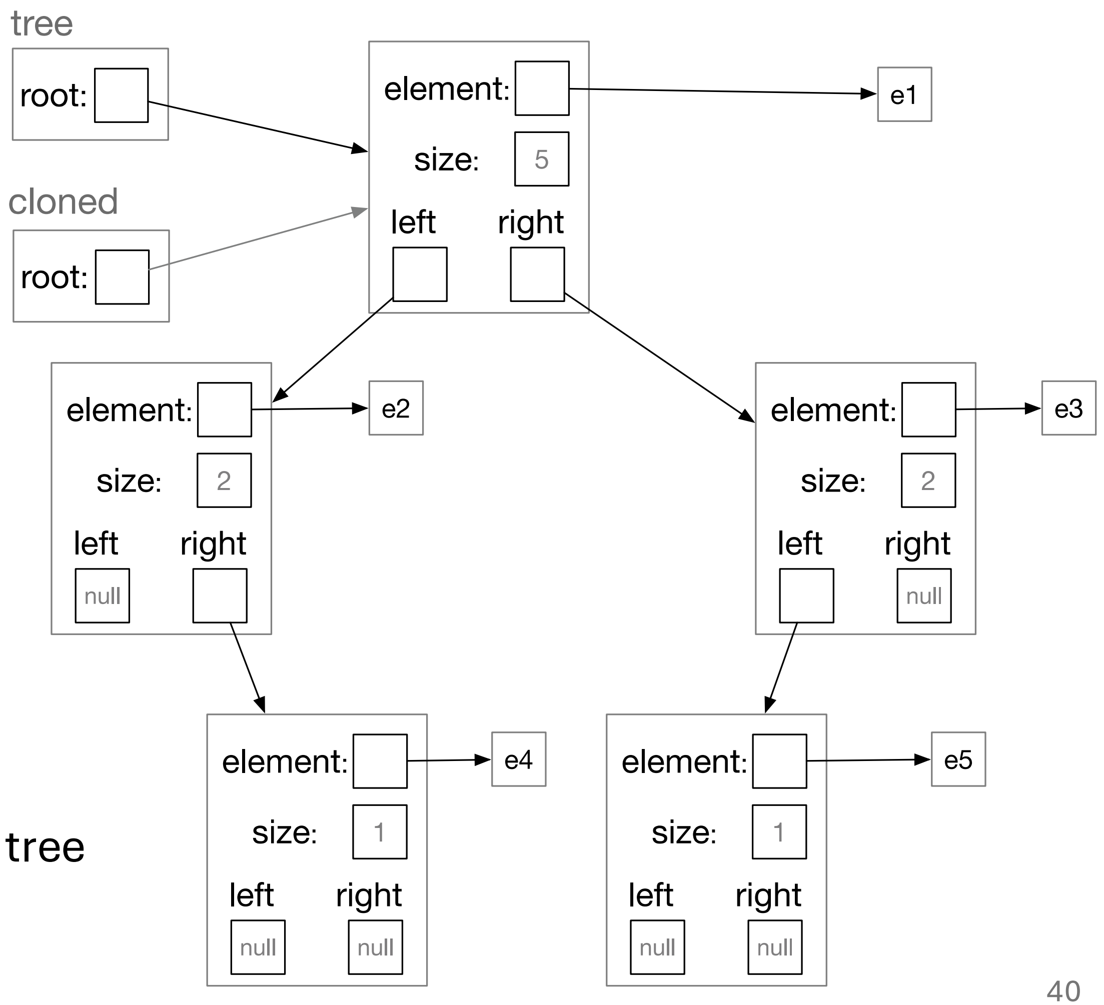
- But in Java there exists a construction specifically designed to do that: the marker interface **Cloneable**
  - it's a marker interface, i.e., it defines no methods
  - we have seen it before on the implementations of lists
  - uses extra-linguistic features, i.e. not implemented with the language, but by special machinery in the virtual machine
  - it makes a field-for-field copy of an object (shallow copy)
  - NOTE: Its use is controversial, and most authors recommend to use a method such as copy, but as Java programmers we must know how it works.

## Binary Trees

- What do we mean by field-for-field copy?
  - primitive types get its value copied
  - reference types get its reference aliased (that is why is called a shallow copy)
- When you implement the interface cloneable
  - you implement a method clone() that will return the copy
  - your implementation will call the one you inherit from your superclass (in our case, Object)
  - and then, we must decide whether we must do a deeper copy
- Let’s see a minimal implementation and its problems

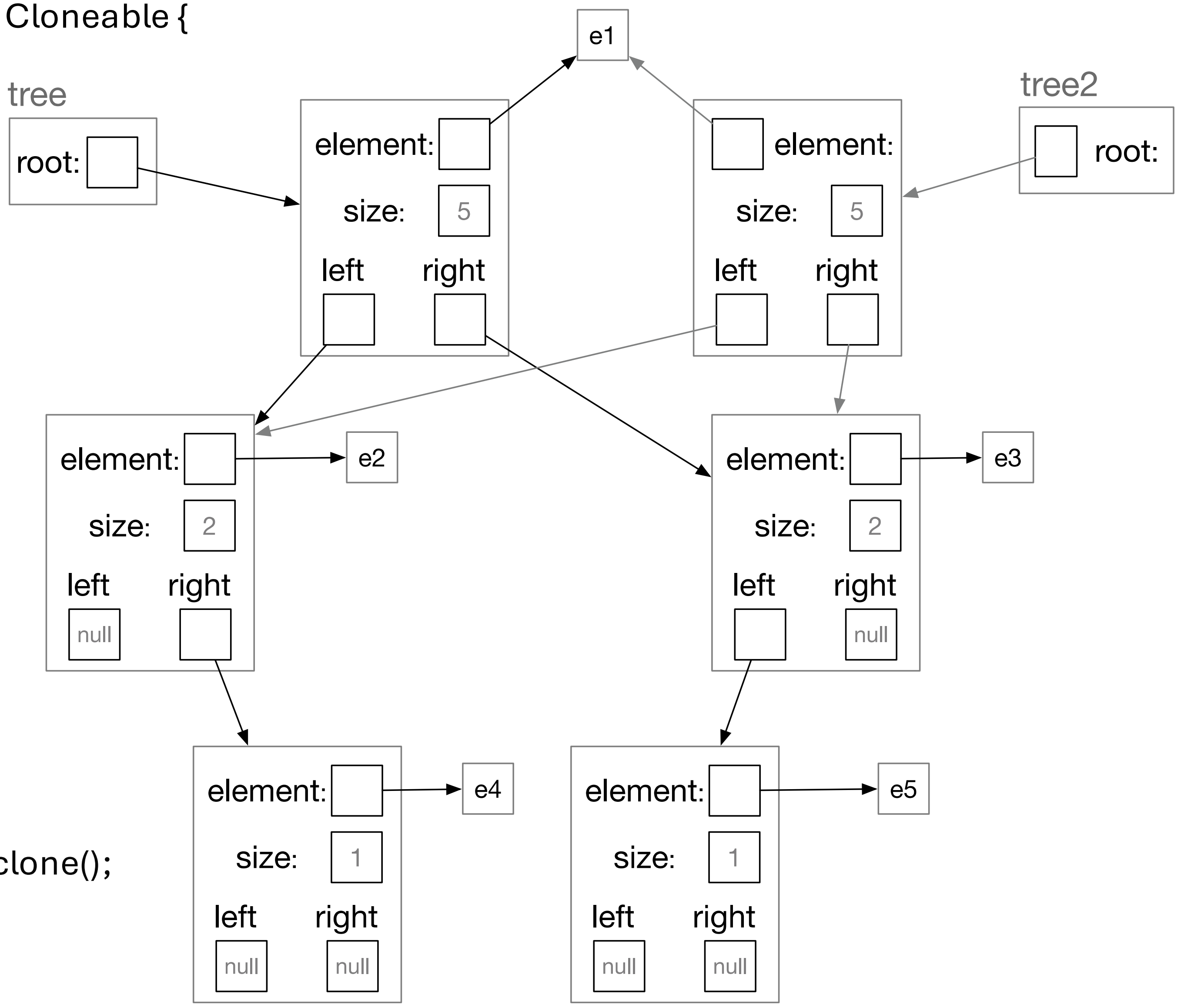
# Binary Trees

```
public class LinkedBinaryTree<E>
   implements BinaryTree<E>, Cloneable {

  @Override
  @SuppressWarnings("unchecked")
  public LinkedBinaryTree<E> clone()  {
     try {
        return (LinkedBinaryTree<E>) super.clone();
     } catch (CloneNotSupportedException e) {
        // this shouldn't happen, since we are Cloneable
        throw new InternalError(e);
     }
  }
}
```



- a new instance of LBT is created
- its root points to the root of the original tree
- what if we also clone the root?

```
public class LinkedBinaryTree<E> implements BinaryTree<E>, Cloneable {

 private static class Node<E> implements Cloneable {
  @Override
  @SuppressWarnings("unchecked")
  public Node<E> clone() {
    try {
      return (Node<E>) super.clone();
    } catch (CloneNotSupportedException e) {
     // this shouldn't happen, since we are Cloneable
      throw new InternalError(e);
    }
  }
 }

 @Override
 @SuppressWarnings("unchecked")
 public LinkedBinaryTree<E> clone()  {
  try {
   LinkedBinaryTree<E> clone = (LinkedBinaryTree<E>) super.clone();
   if (root != null) clone.root = root.clone();
   return clone;
  } catch (CloneNotSupportedException e) {
    // this shouldn't happen, since we are Cloneable
   throw new InternalError(e);
  }
}
```



- You continue sharing most of the tree
- You must clone down the tree !!

## Binary Trees

- To do a **deep-copy** of the structure of the tree, the clone method of the class Node must do a **recursive cloning**
- But this does not solve the problem with sharing the instances of elements !!!
  - We can demand the E to be Cloneable as well
  - But then our implementation won't be usable for some types

```
private static class Node<E> implements Cloneable {
  @Override
  @SuppressWarnings("unchecked")
  public Node<E> clone() {
    try {
      Node<E> clone = (Node<E>) super.clone();
      if (left != null) clone.left = left.clone();
      if (right != null) clone.right = right.clone();
      return clone;
    } catch (CloneNotSupportedException e) {
     // this shouldn't happen, since we are Cloneable
      throw new InternalError(e);
    }
  }
}
```

## Binary Trees

- You can access this implementation at this repository
- The implementation goes a little bit farther than the one described here because, as the empty tree is unmodifiable, it tries to create only one instance for it
- So, all references to the empty tree reference this instance
- As said before, the empty tree is unmodifiable (and immutable), so his is safe to do
  - And it saves some space !!!

## Binary Trees

- But this implementation is not perfect because it allows the creation of **non-tree** things
  - But, at least, no cycles are allowed
- Can be avoid this?
  - Yes, but then the constructor won't be $\mathcal{O}(1)$ but $\mathcal{O}(n)$ because we'll need to clone the tree we receive as parameter
  - As always, if we can live with that, a $\mathcal{O}(1)$ cost is desirable, if not we must pay the price of cloning
- For example, draw the representation of the trees after:

```
var tree1 = new LinkedBinaryTree<Integer>(null, 1, null);
var tree2 = new LinkedBinaryTree<Integer>(tree1, 2, tree1);
```

## Binary Search Trees (BSTs)

## Binary Search Trees

- At the beginning, we’ve said that trees are usually used as an **implementation device** to implement other data structures
- This will be the case in hand, in which we’ll use trees to implement an **associative data structure**
  - In this structure, we’ll **associate keys to values**
  - **Keys are comparable**
- Possible implementations
  - List of pairs key-value
    - all operations linear
  - Sorted list (by key) of pairs key-value
    - search logarithmic
    - insertion/deletion linear

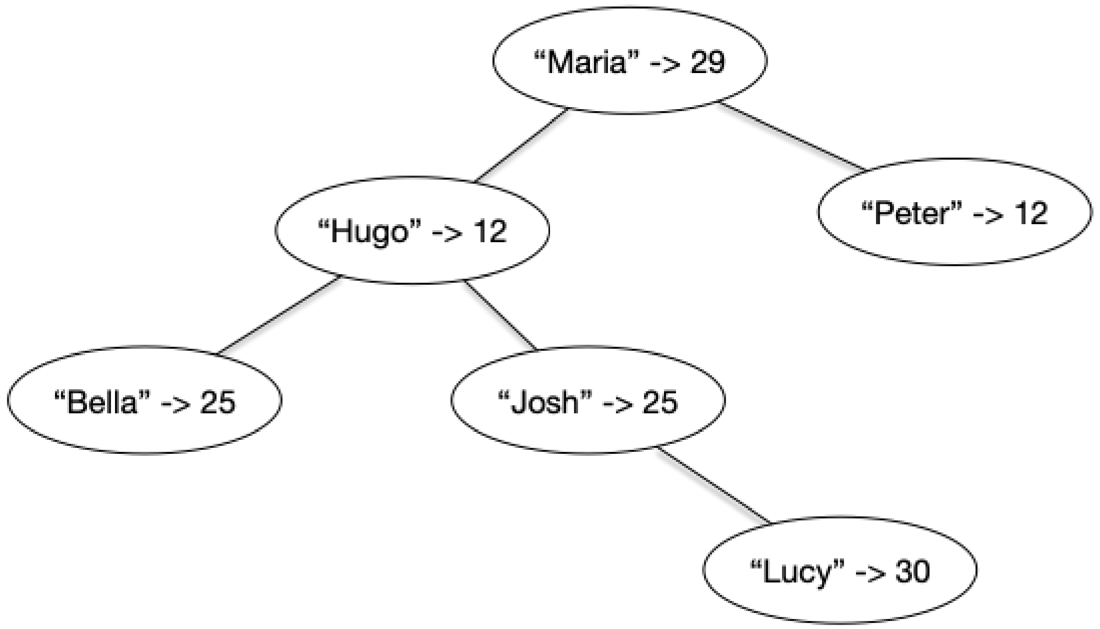
## Binary Search Trees

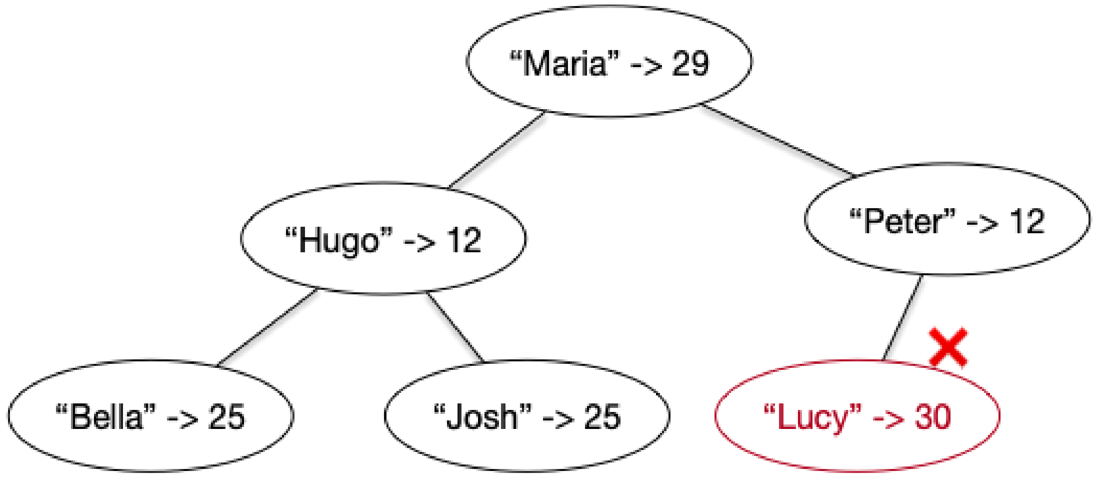
- Binary Search Trees are binary trees which allow to implement these operations in **logarithmic time**
  - Well, **only if the tree is balanced**
- The **structure** of the tree is only **determined by its keys** (the values associated to them are only a payload)
  - So, we'll only show the keys in the diagrams
  - But each node carries a pair key-value
  - Well, when you use a BST to implement a Set, you do not have a value

## Binary Search Trees

- A **binary search tree** is either
    - An **empty** binary tree
    - A non-empty binary tree in which
        - The **key** in the **root** is **greater** than all the **keys** in the **left** subtree
        - The **key** in the **root** is **lower** than all the **keys** in the **right** subtree
        - Both **left** and **right** subtrees are **binary search trees**
- So,
    - There are **no duplicated** keys
    - **Keys** must be **comparable**
    - If the BST implements an associative data structure, the nodes contains both a key and a value
        - Values can be duplicated in the tree and do not need to be comparable

# Binary Search Trees

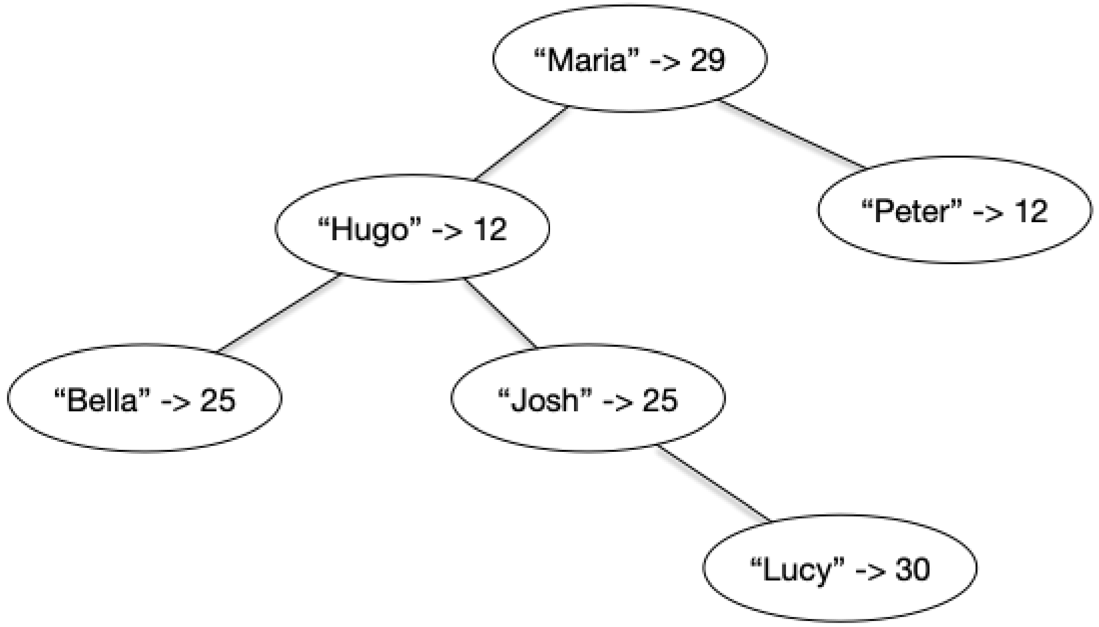




## Binary Search Trees

**Search for the value associated to a key $k$**

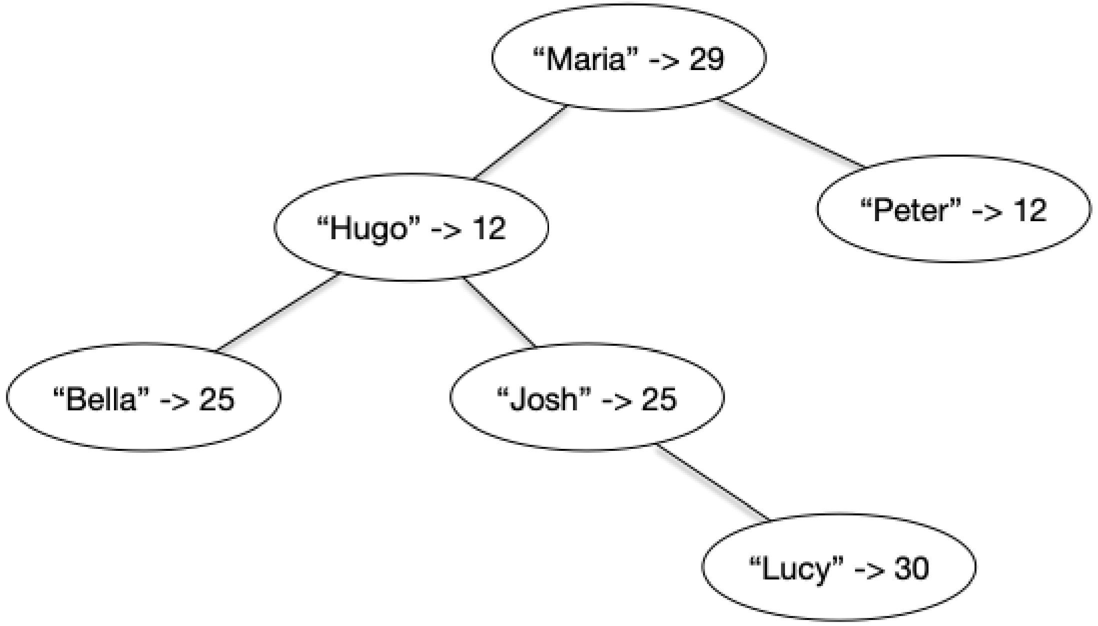
- If at, any point, the tree is empty, we know that the key $k$ does not belong to the tree and we're finished.
- Beginning at the root node of the tree
    - If the key in the node is equal to $k$, we've found it, and we're finished
    - If the key in the node is greater than $k$, continue the search on the left child
    - If the key in the node is lower than $k$, continue the search on the right child.



# Binary Search Trees

**Insert (associate) the key $k$ with the value $v$**

- If the tree is empty, create a node to be the new root with the new pair
- If not, search for the key $k$ and
  - if is found, change its associated value to $v$
  - If not, the node where the search failed was a leaf or had only one child
    - Add a new node with the new pair to it

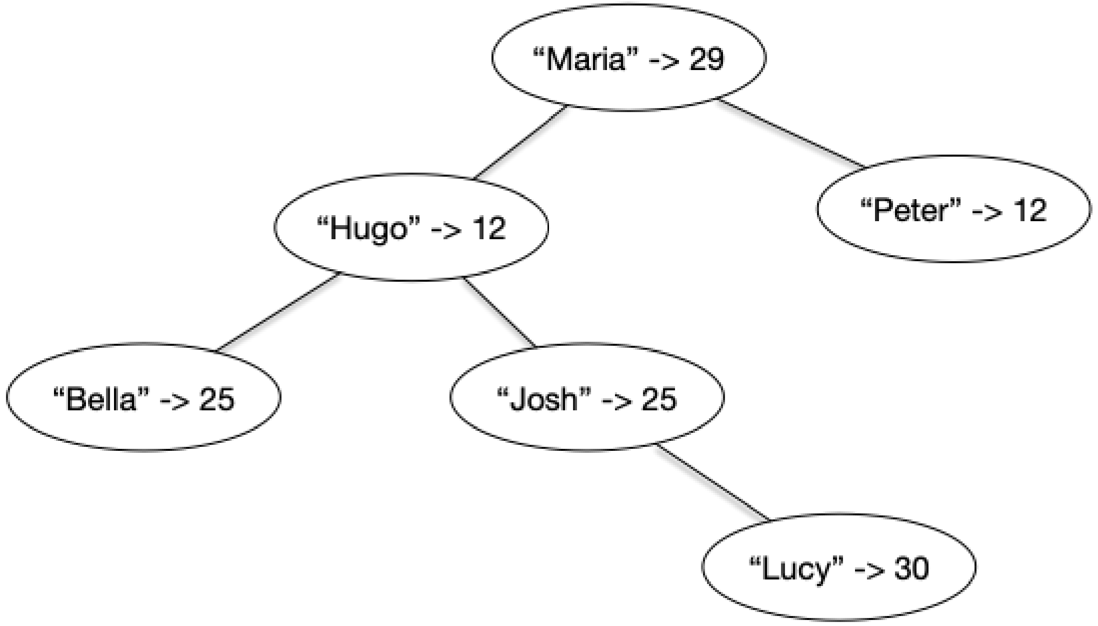


## Binary Search Trees

**Delete key $k$**

There are four cases:

A. $k$ does not exist in the tree
B. It’s in a leaf
C. It’s in a node of degree 1 (with a single child)
D. It’s in a node of degree 2 (with two children)



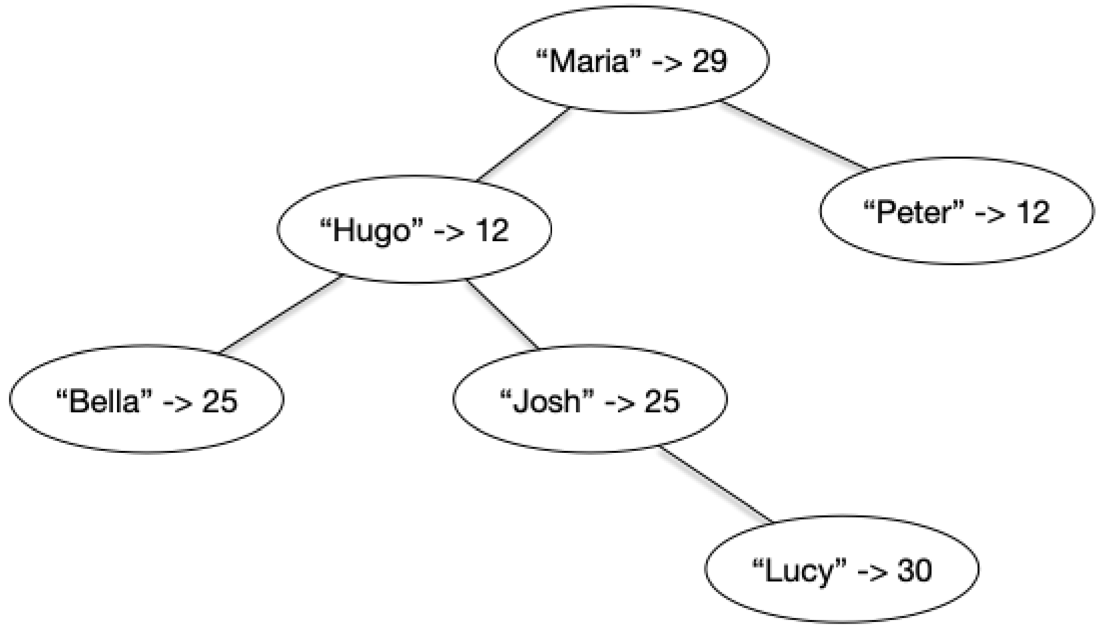
# Binary Search Trees

**Delete key $k$**

There are four cases:

$A.$ $k$ does not exist in the tree

- No further action is needed



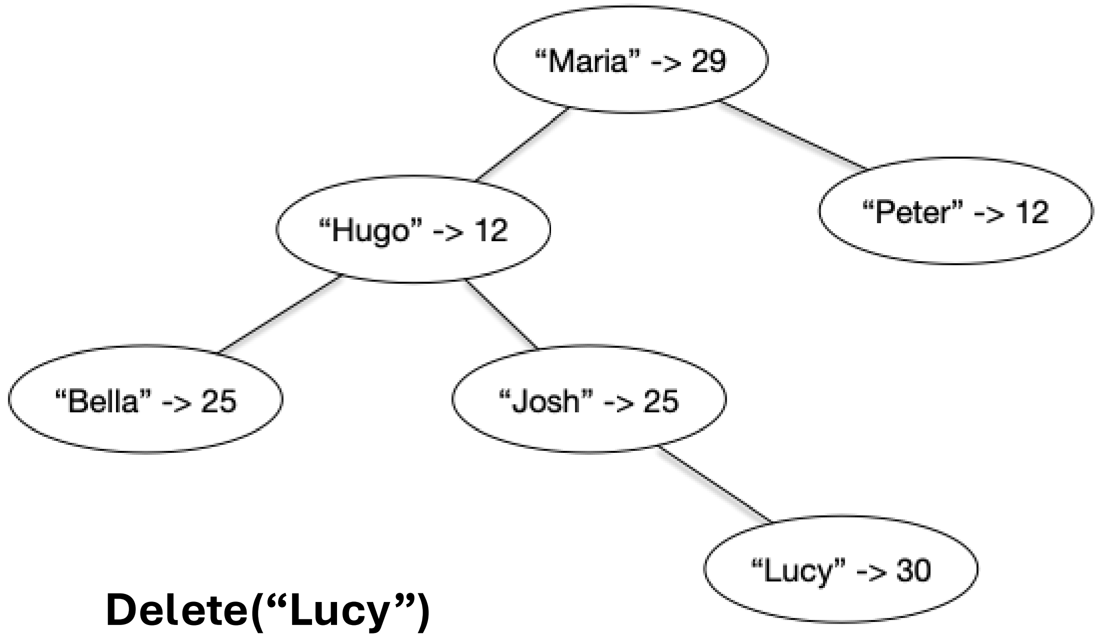
## Binary Search Trees

**Delete key $k$**

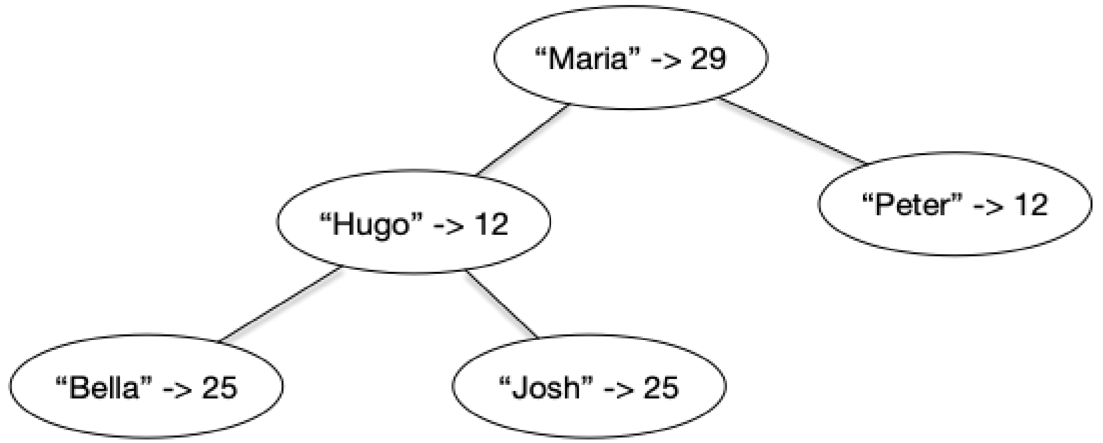
There are four cases:

B. It’s in a leaf

- We delete the leaf



**Delete(“Lucy”)**



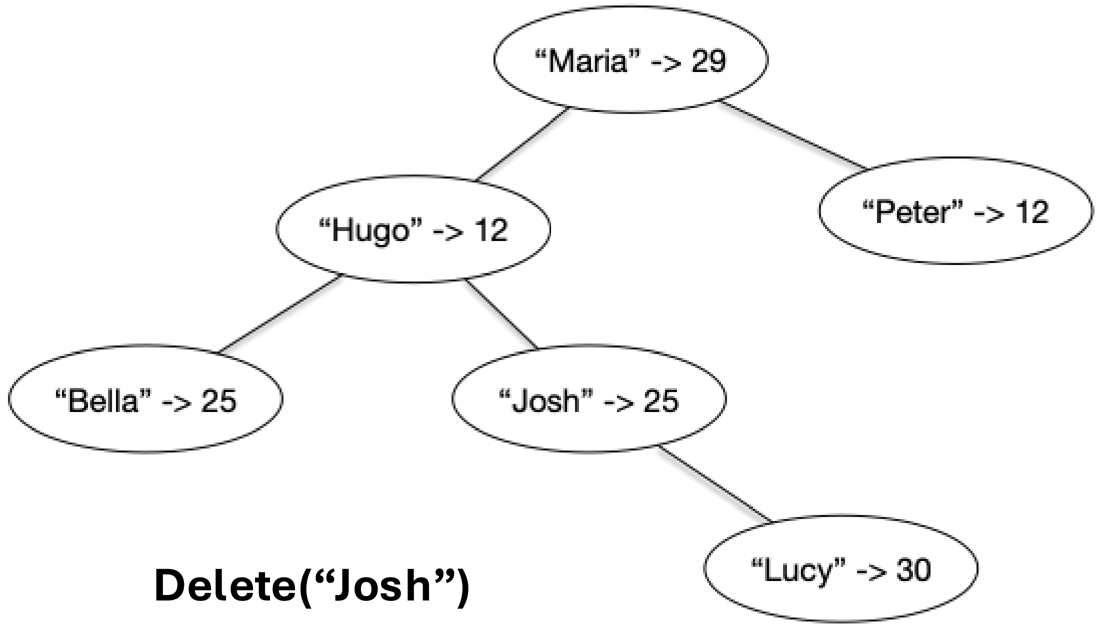
## Binary Search Trees

**Delete key $k$**

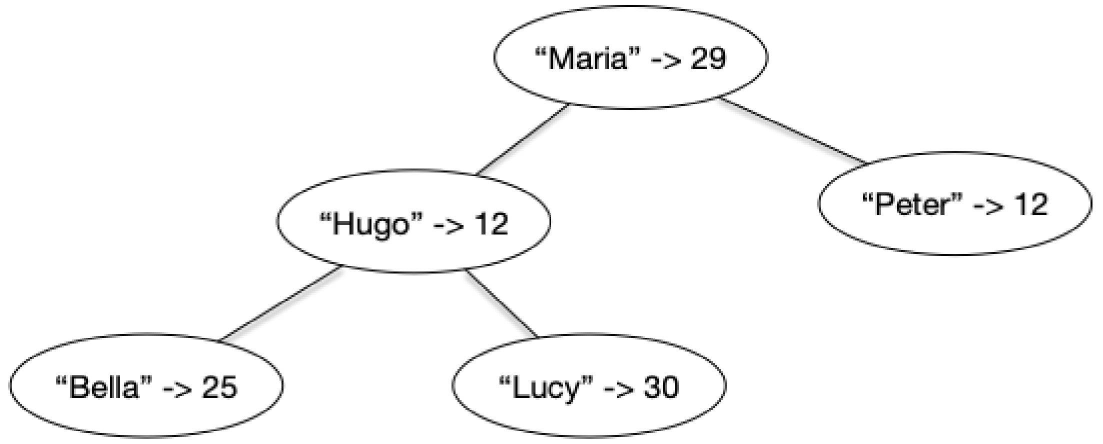
There are four cases:

C. It's in a node of degree 1 (with a single child)

- Its single child substitutes it



**Delete("Josh")**



# Binary Search Trees

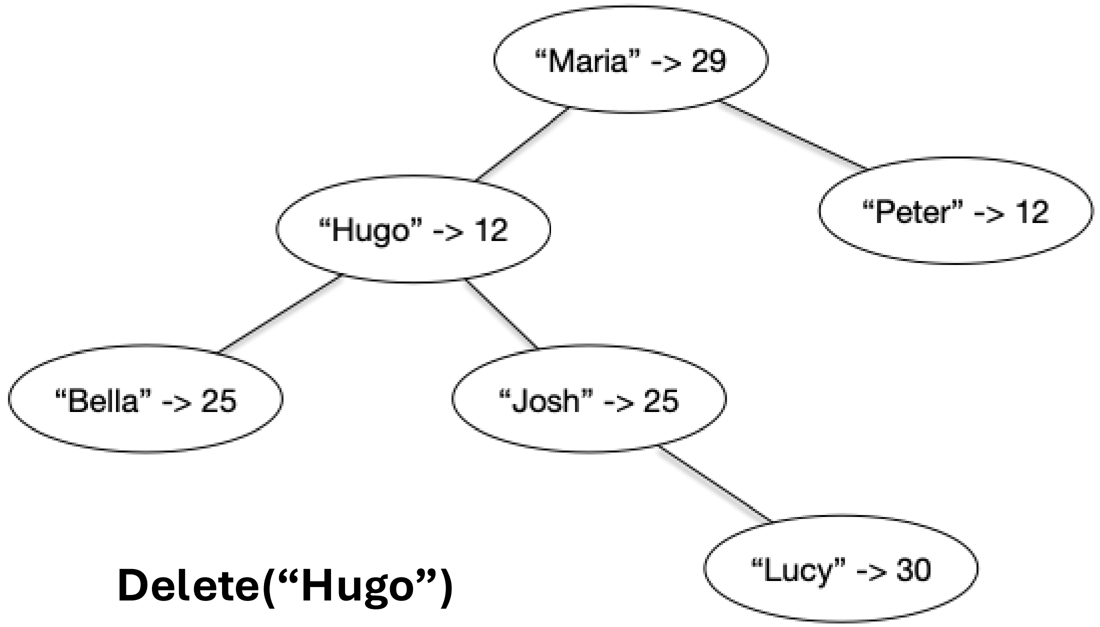
**Delete key $k$**

There are four cases:

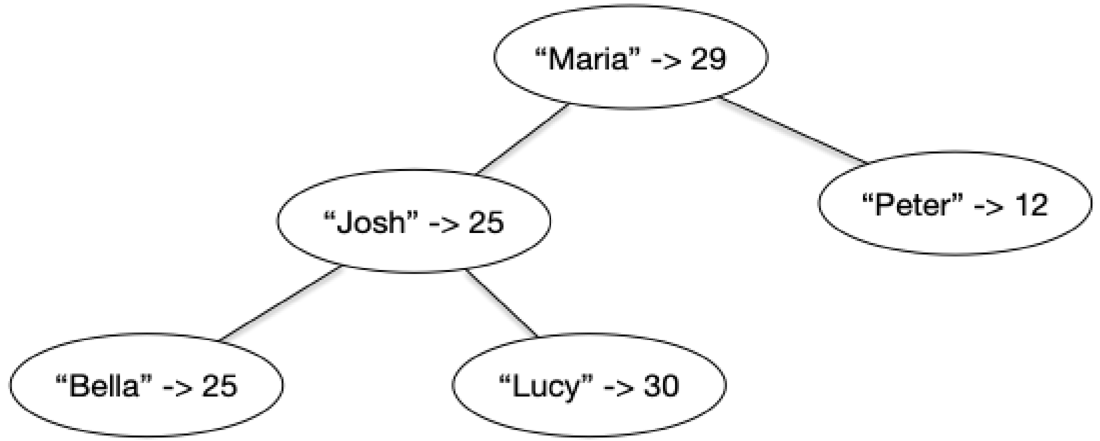
D. It's in a node of degree 2 (with two children)

- We substitute the key pair with that of the lowest key in the right subtree
- And we remove it from the right subtree

- NOTES:
  - The key is the next to $k$ in sorted order
  - We could have used the biggest key in the left subtree (the previous in sorted order)

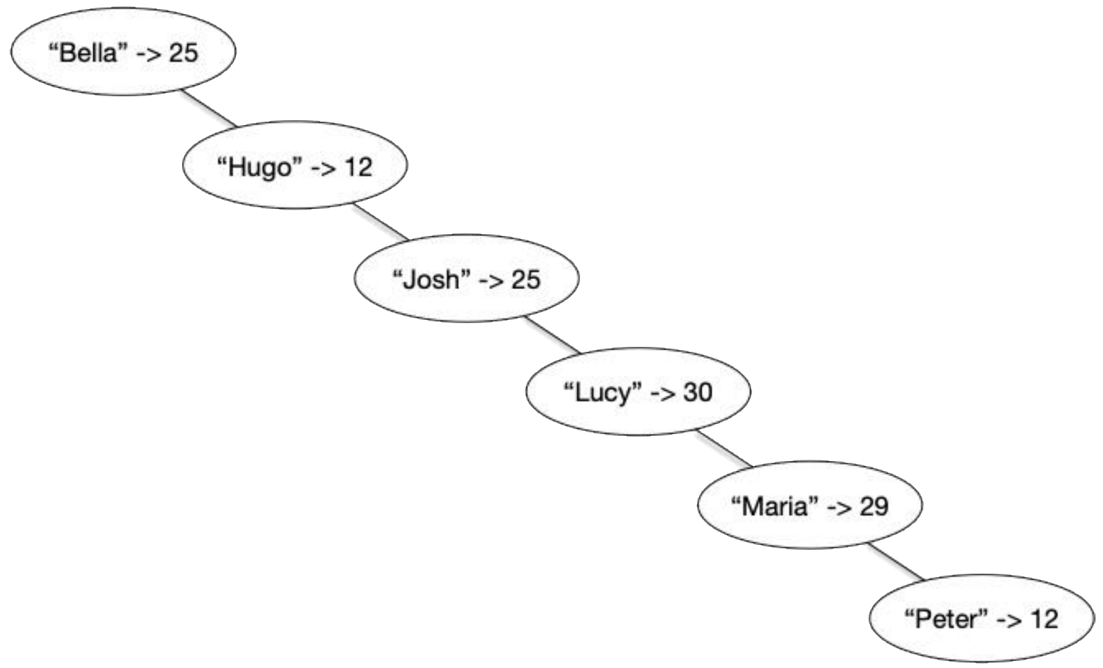


**Delete("Hugo")**



## Binary Search Trees

- All the operations described above have cost that depends on the **height of the tree**
  - But this make the **linear on the size of the tree**
- Why?
  - Because nothing makes the tree to be balanced !!



## Binary Search Trees

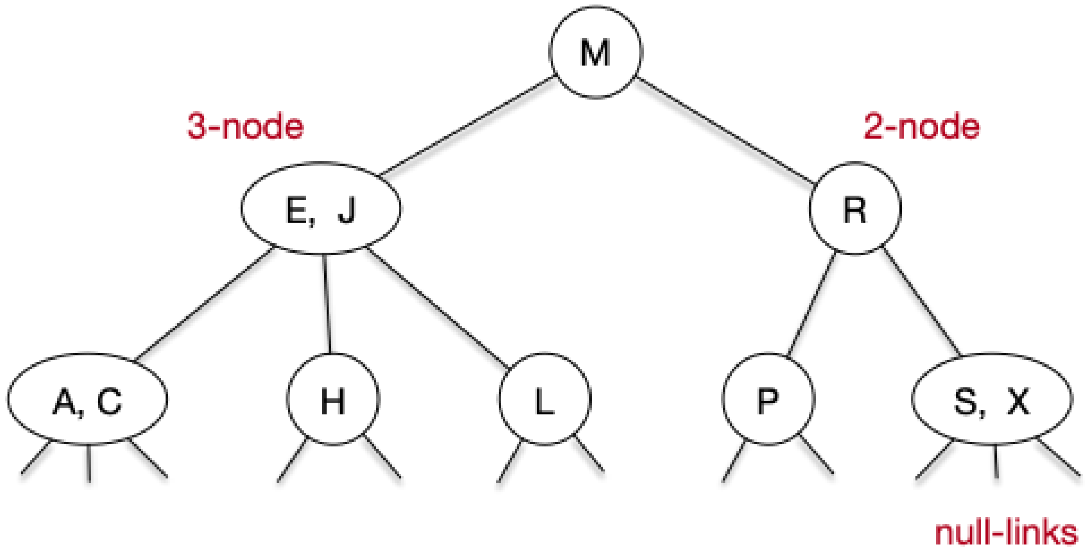
- So, to make the operations in a tree **logarithmic** on its size, one must guarantee that the tree is **balanced** !!
- AVL Trees
  - Invented by Georgii Adelson-Velsky and Yevgeniy Landis in 1962
  - The tree is perfectly balanced all time
  - But operations are complex to implement
- Red-Black Trees
  - Invented by Leonidas Guibas and Robert Sedgewick in 1978, based on previous work by Rudolf Bayer in 1972
  - The tree is not perfectly balanced  but guaranteed logarithmic time
  - Operations are simpler to implement

## 2-3 Trees

- Although the most used implementation is that of Cormen et al., we’ll present Sedgewick’s because it is simpler
- And we’ll only consider the search (trivial) and insertion operations
    - For the rest of the operations, refer to section 3.3 of Sedgewick’s book
    - Or its associated web page
    - Or this paper Left-leaning Red-Black Trees
- But this presentation does not start with Red-Black trees but with another kind of (not binary) search trees: the **2-3-Search Trees**

# 2-3 Trees

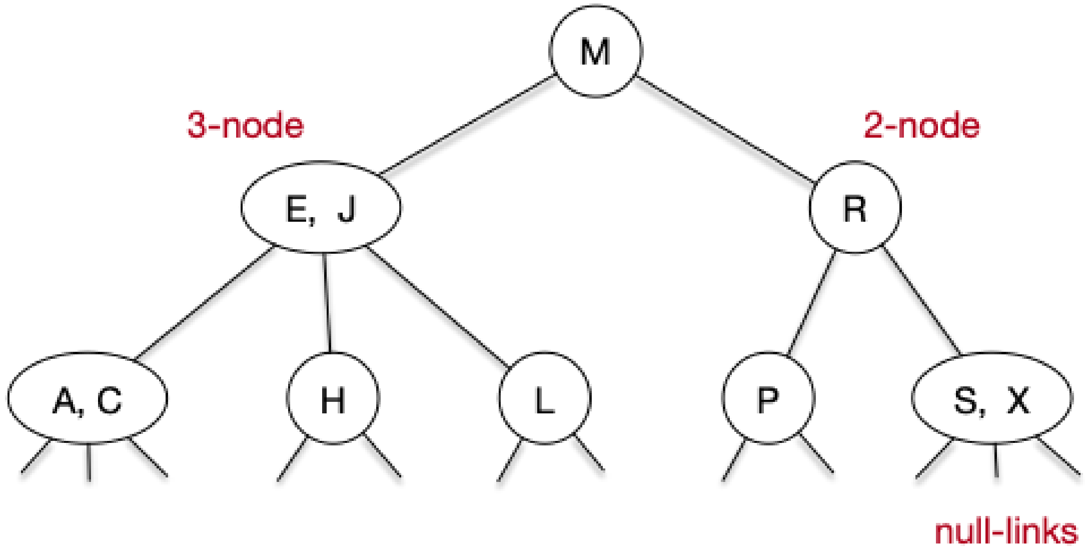
- A **2-3 Search Tree** is a tree that can be
  - **Empty**
  - A **2-Node** with a key (and associated value) and a left 23-subtree with smaller keys and a right 23-subtree with bigger keys
  - A **3-Node** with two keys (and associated values) and a left 23-subtree with smaller keys than those in the node, a middle 23-subtree with keys between those in the node, and a right 23-subtree with bigger keys than those in the node
- A **null-link** is a link to an empty tree.
- A **perfectly balanced 2-3-Search Tree** has all the **null-links** at the same distance from the root
  - We'll only consider those and simply call them **2-3-Trees**



## 2-3 Trees

**Search for a key $k$**

- A simple extension of the algorithm for BSTs
- As we only consider balanced 23-Trees, the cost of the search is logarithmic on the number of nodes
    - The heigh of the tree is between $\log_3 n$ and $\log_2 n$



# 2-3 Trees

**Insertion of a key $k$ (and associated value $v$):**

- We proceed as in a BST
- We have several cases:

A. Insert into a 2-Node

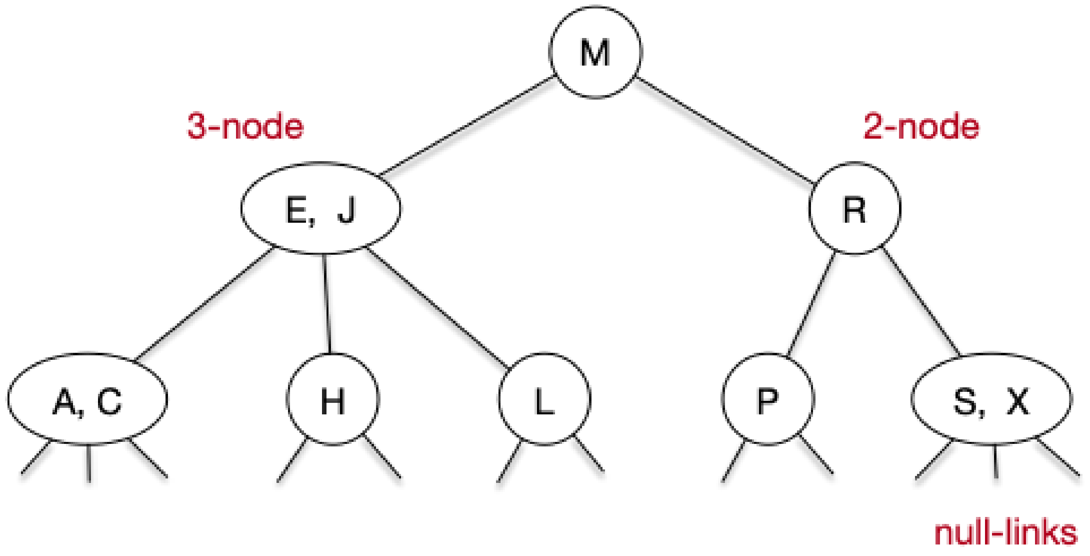
B. Insert into a 3-Node

   i. It's the single node of the tree

   ii. His parent is a 2-Node

   iii. His parent is a 3-Node

   iv. It's the root

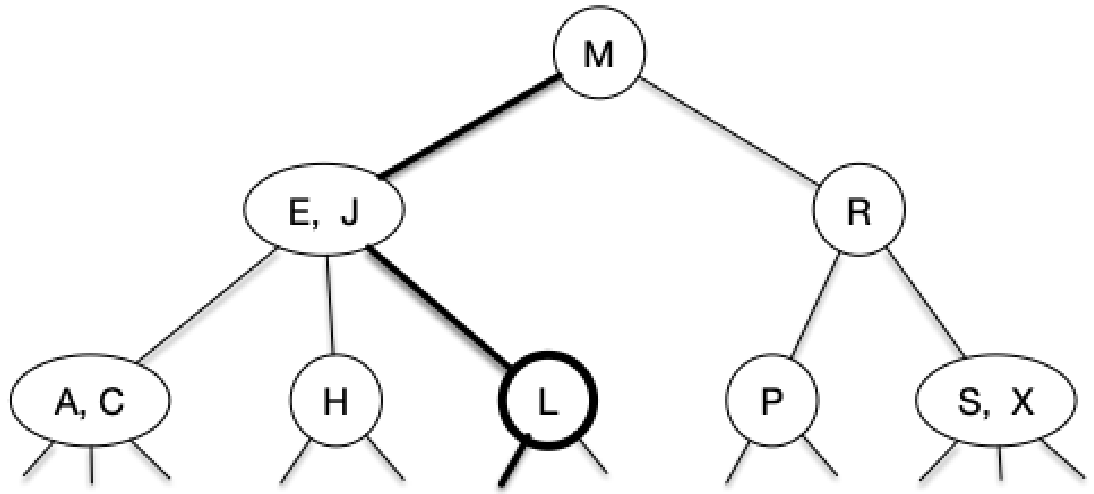


# 2-3 Trees

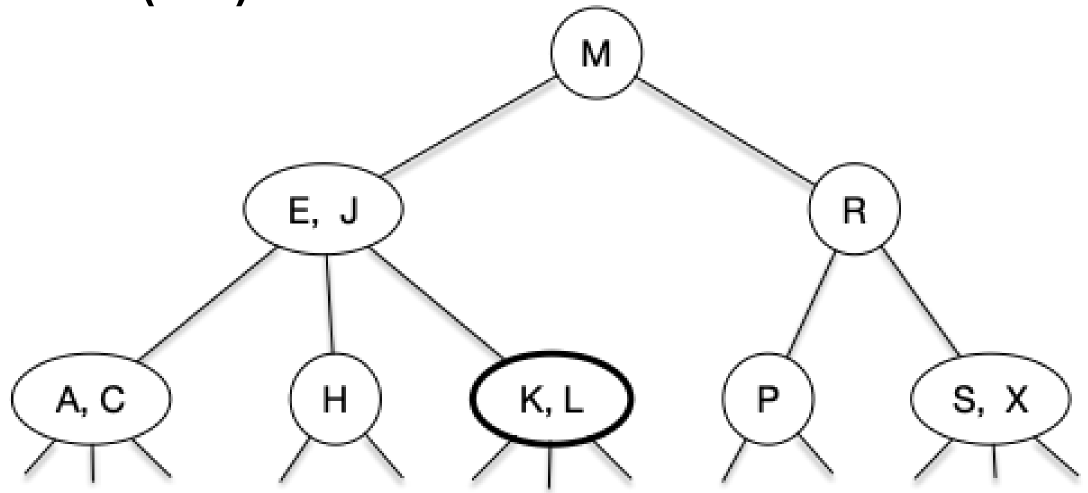
**Insertion of a key $k$ (and associated value $v$):**

A. Insert into a 2-Node

- **Replace** the 2-Node where we fail with a 3-Node



**Insert("K")**



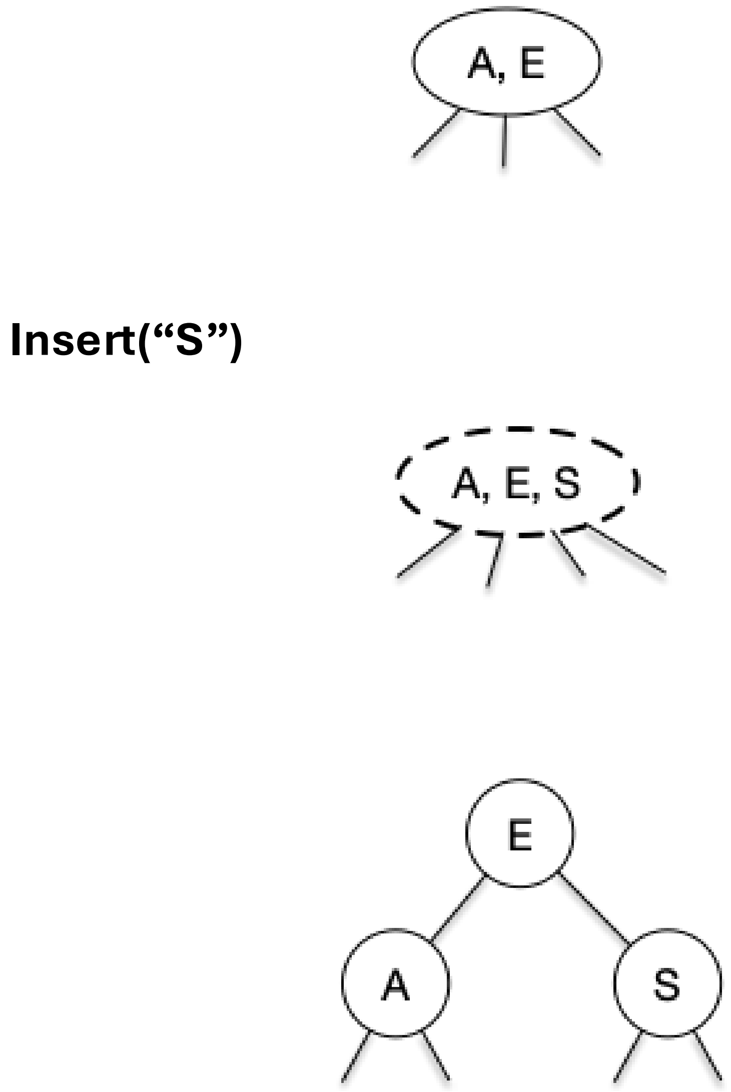
## 2-3 Trees

**Insertion of a key $k$ (and associated value $v$):**

B. Insert into a 3-Node

   i. It’s the single node of the tree

   - **Replace** the 3-Node with a **temporal** 4-Node
   - **Split** it into three 2-Nodes



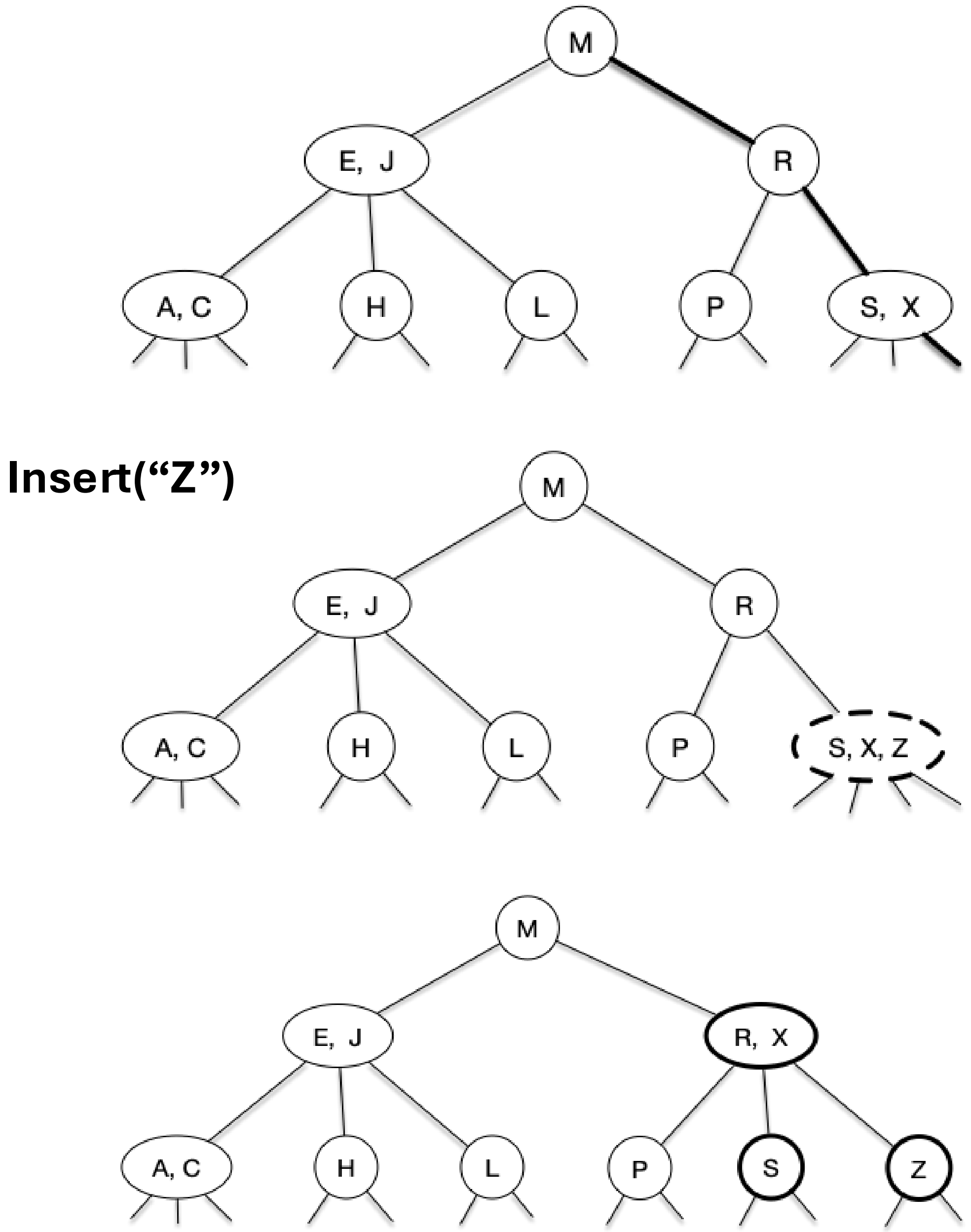
# 2-3 Trees

**Insertion of a key $k$ (and associated value $v$):**

B. Insert into a 3-Node

ii. Its parent is a 2-Node

- **Replace** the 3-Node with a temporal 4-Node
- **Split** the 4-node into two 2-Nodes (for the left and right)
- **Insert** the middle key on the parent



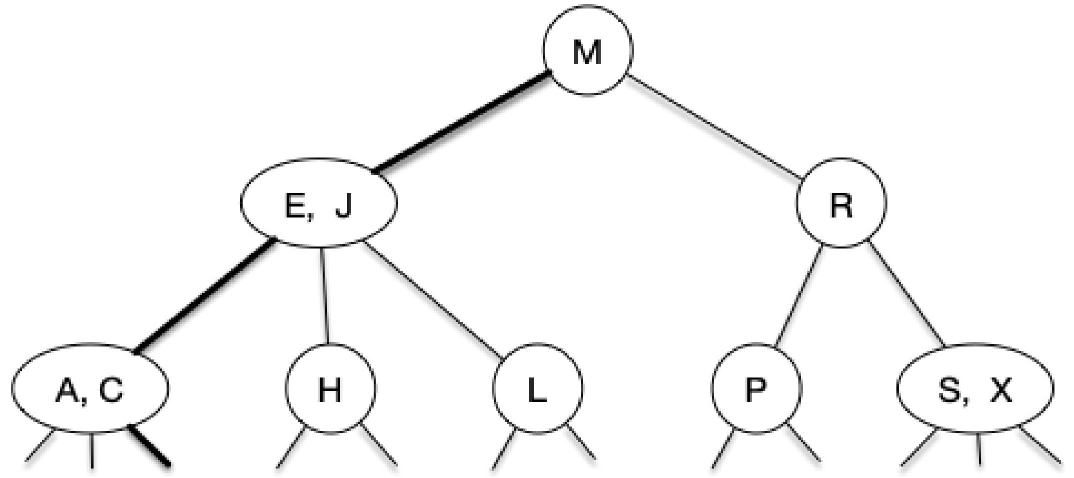
## 2-3 Trees

**Insertion of a key $k$ (and associated value $v$):**

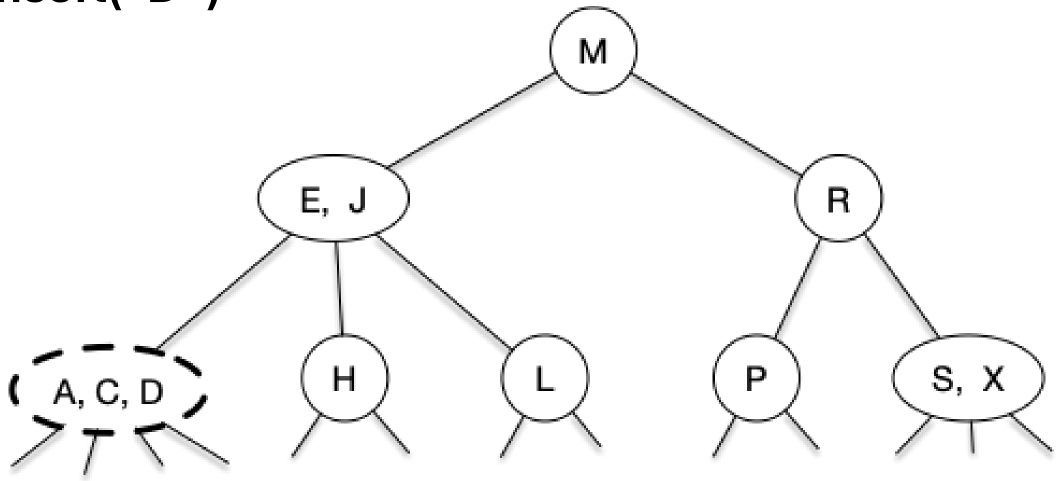
B. Insert into a 3-Node

   iii. Its parent is a 3-Node

   - **Replace** the 3-Node with a temporal 4-Node
   - **Split** the 4-node into two 2-Nodes (for the left and right)
   - **Insert** the middle key on the parent creating a new temporal 4-Node and … (this can arrive up to the root)



**Insert(“D”)**



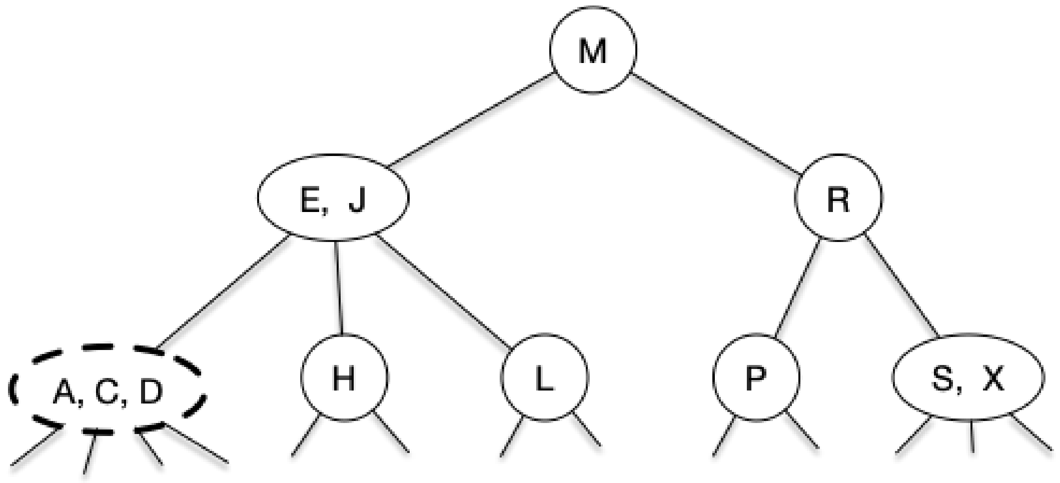
# 2-3 Trees

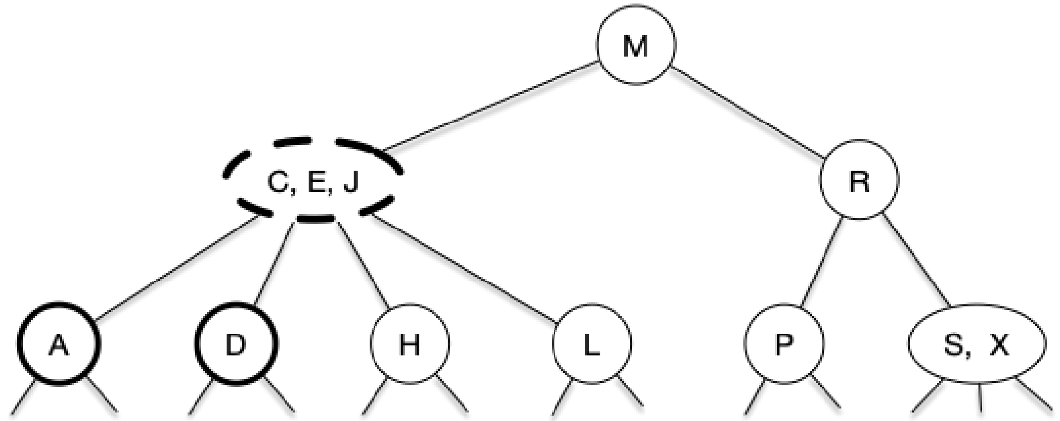
**Insertion of a key $k$ (and associated value $v$):**

B. Insert into a 3-Node

iii. Its parent is a 3-Node

- **Replace** the 3-Node with a temporal 4-Node
- **Split** the 4-node into two 2-Nodes (for the left and right)
- **Insert** the middle key on the parent creating a new temporal 4-Node and … (this can arrive up to the root)





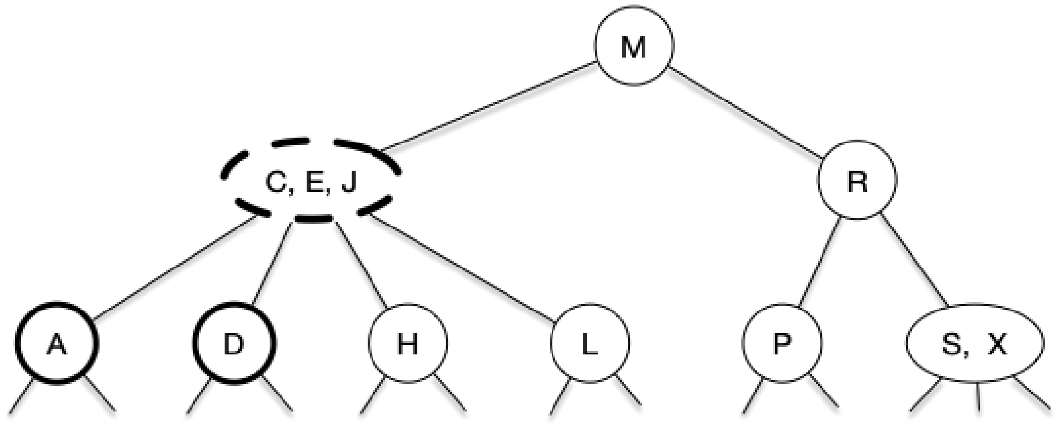
## 2-3 Trees

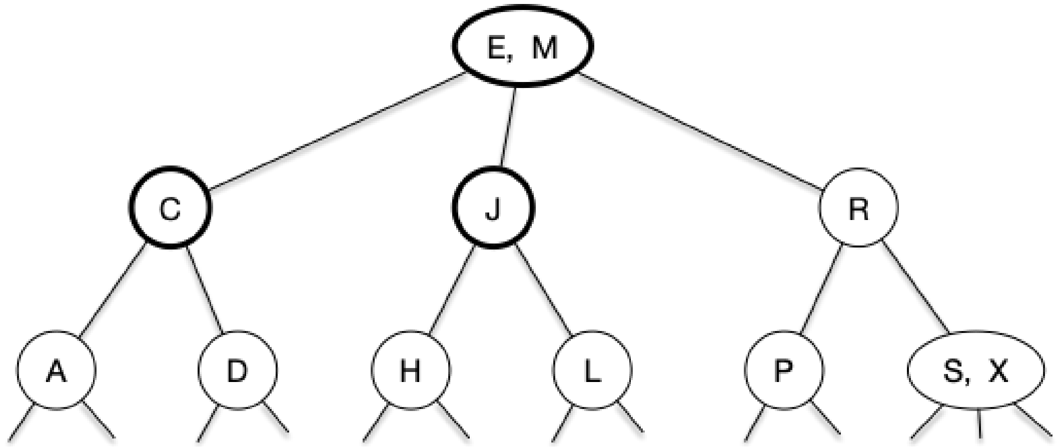
**Insertion of a key $k$ (and associated value $v$):**

B. Insert into a 3-Node

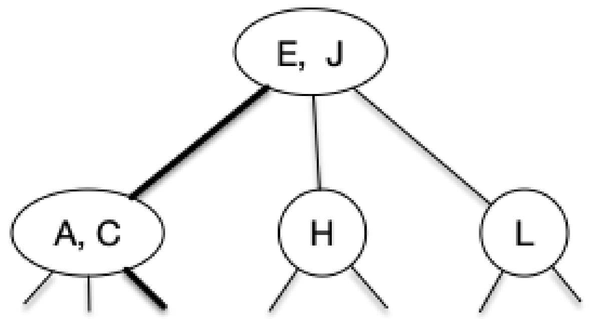
iii. Its parent is a 3-Node

- **Replace** the 3-Node with a temporal 4-Node
- **Split** the 4-node into two 2-Nodes (for the left and right)
- **Insert** the middle key on the parent creating a new temporal 4-Node and ... (this can arrive up to the root)

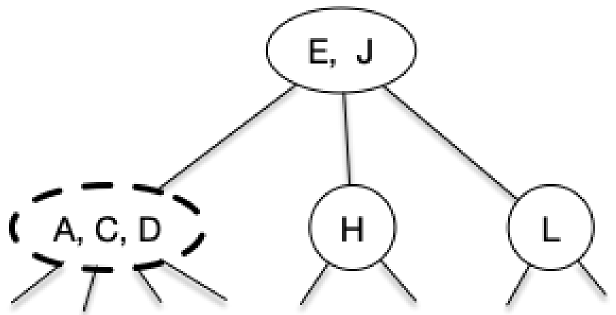




# 2-3 Trees



Insert(“D”)

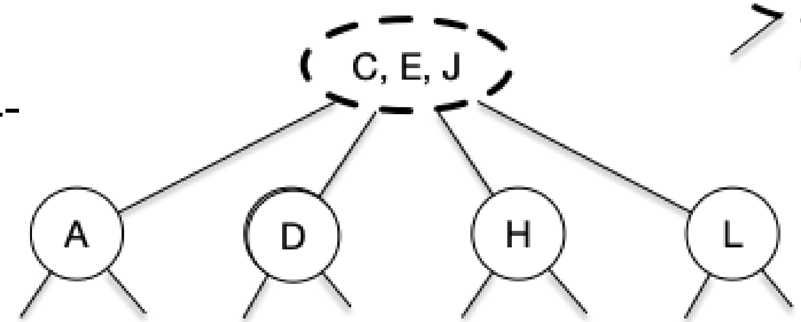


**Insertion of a key $k$ (and associated value $v$):**

B. Insert into a 3-Node

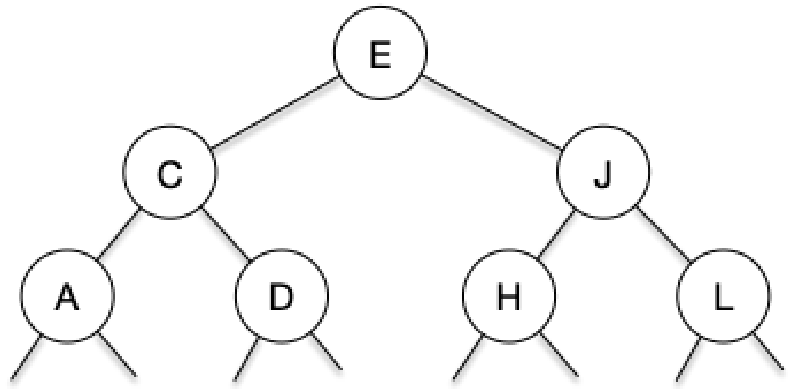
iv. It’s the root

- **Replace** the root with a temporal 4-Node
- **Split** it into three 2-Nodes



NOTES:

- This is when there is a path formed by only 3-Nodes from the point of insertion to the root
- Actually, it’s the same case as B.i

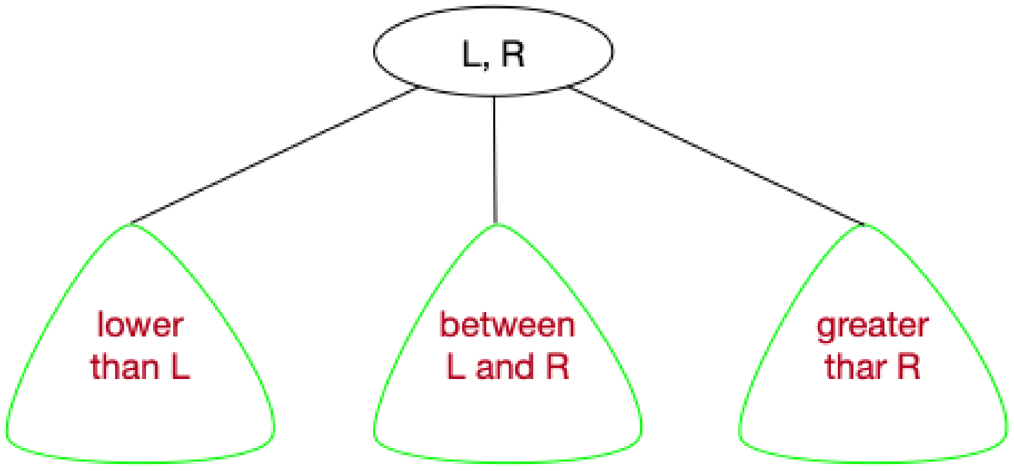


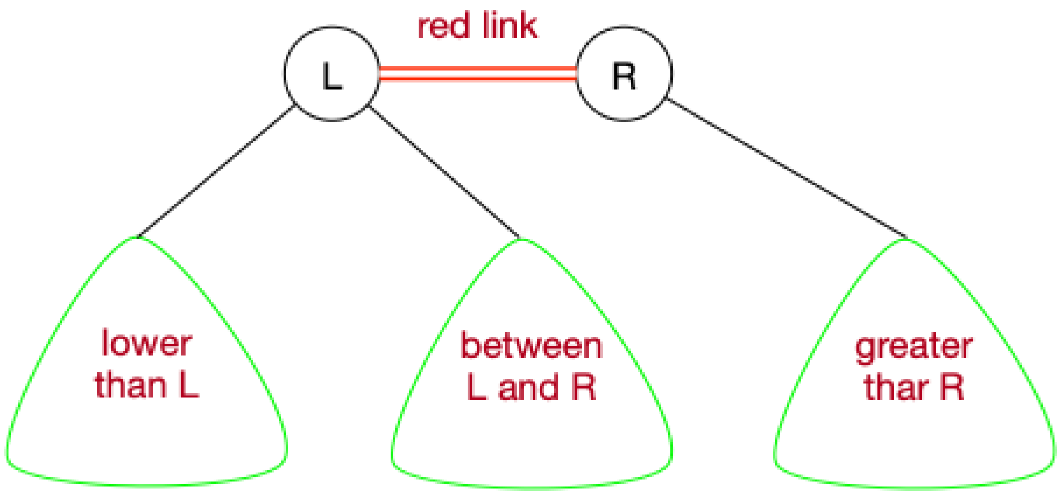
## 2-3 Trees

- **Splitting** a temporary 4-Node is a **local operation** of the tree ($\mathcal{O}(1)$)
  - No other part of the tree must be examined other that the specific nodes and links
  - And, at most, it propagates up to the root (**logarithmic** path)
- Besides, all transformations **preserve** the **order** and **perfect balance** of the tree
- **NOTE**:
  - Unlike BSTs, 23-Trees grow from the bottom to the top
- **Problem**: implementing it with different kinds of nodes is not easy.
- **Solution**: **red-black** trees are a way to **encode 2-3-trees** with only **one kid of node**.

# Red-Black Trees

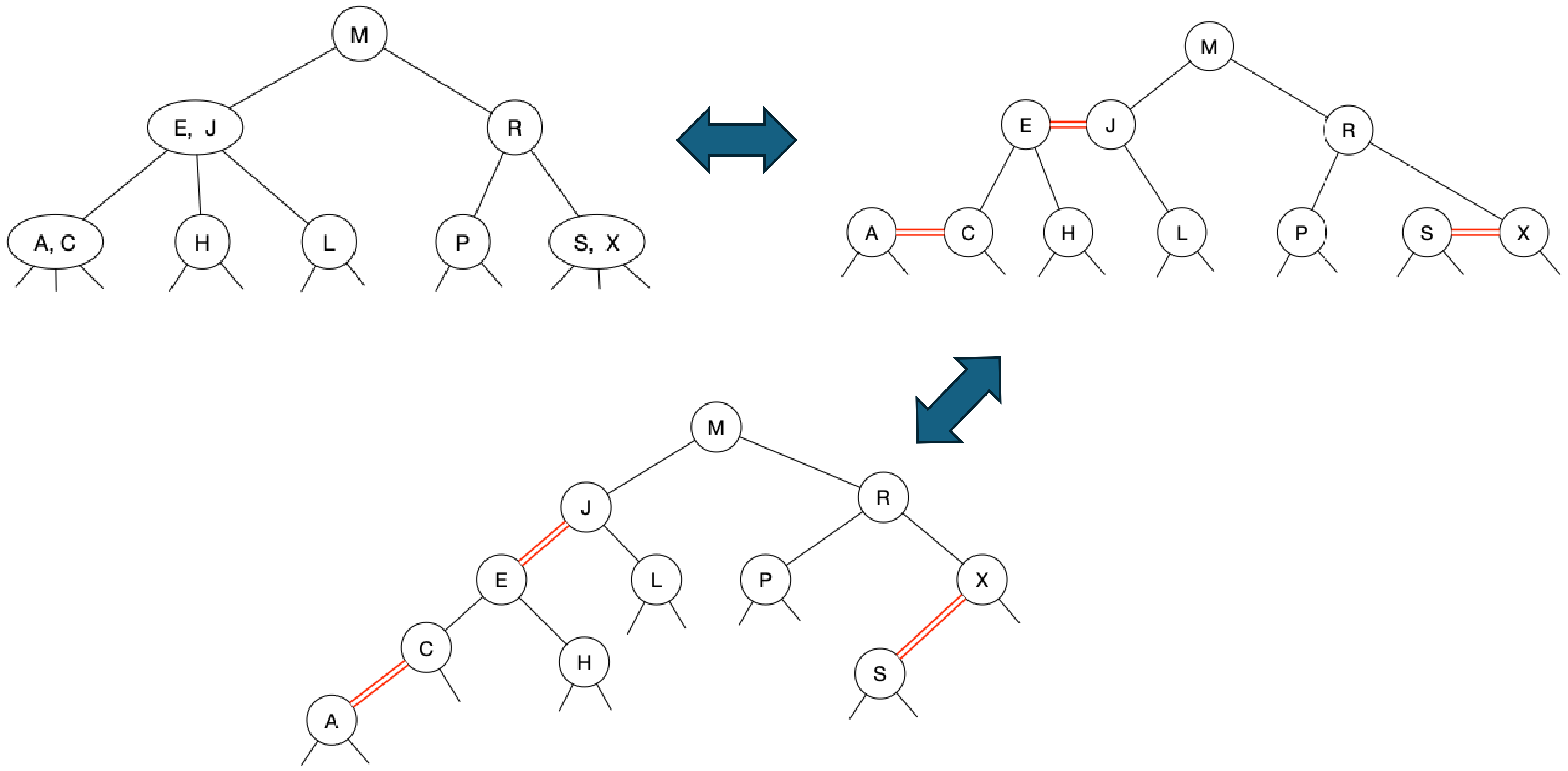
- **NOTE**: There are slightly different versions of red-back trees, and we'll follow Sedgewick's
- The basic idea behind a red-black tree is to **encode 2-3 trees**
  - Starting with simple BSTs to encode 2-nodes
  - Adding additional information to encode 3-nodes
- Two different kind of links:
  - **Red links**: bind two 2-nodes to represent a 3-node
  - **Black links**: bind the 2-3 tree together
  - **Null-links** are always **black**





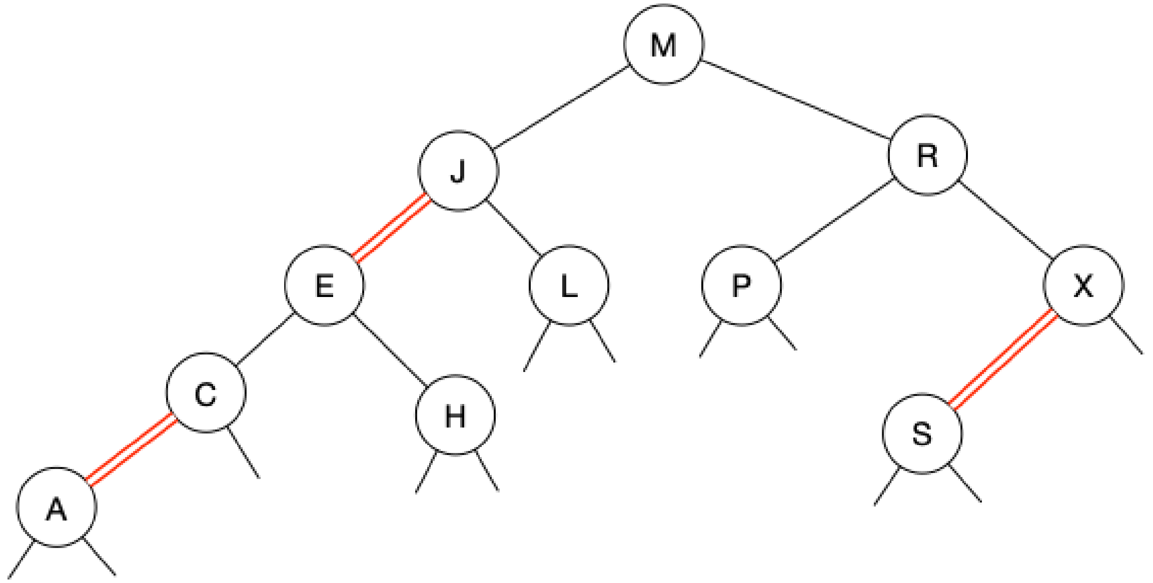
## Red-Black Trees

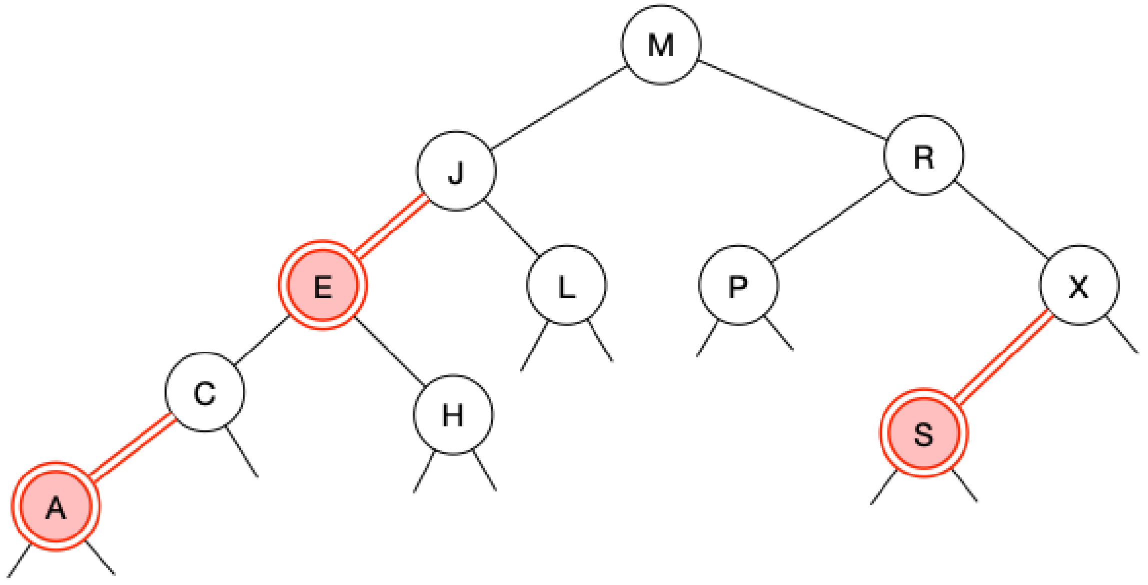
- We can define this representation as BSTs having red and black links and satisfying the following restrictions:
  1. **Red links lean left**
  2. **No node has two red links** connected to it
  3. The tree has **perfect black balance**: every **path** to the **root** to a **null-link** has the **same number of blank links**
- This type of RB Trees are called **Left Leaning Red Black Trees (LLRBTs)**
- There is a 1-to-1 relation (**isomorphism**) between LLRBTs and 2-3 trees


M
E, J
R
A, C
H
L
P
S, X
M
E
J
R
A
C
H
L
P
S
X
M
J
R
E
L
P
X
C
H
S
A

# Red-Black Trees

- As links are references, they cannot encode its colour
- But, as each node has only one link that comes from its parent, **we'll encode its colour in the child.**
- So, we'll have two kinds of nodes:
  - **Black nodes**: those in which their parent link is black
  - **Red nodes**: those in which their parent link is red
- The **root** node will always be **black**.





## Red-Black Trees

- The full code of the implementation in Sedgewick's book can be found at code
- Sometimes the use of Java constructs is simplified:
  - Comparable
  - Non-static class Node
- **NOTE**: We have preferred to maintain the original implementation.

```
public class RedBlackBST<Key extends Comparable<Key>, Value> {

  private static final boolean RED   = true;
  private static final boolean BLACK = false;

  private Node root;    // root of the BST

  // BST helper node data type
  private class Node {
    private Key key;            // key
    private Value val;          // associated data
    private Node left, right;   // links to left and right subtrees
    private boolean color;      // color of parent link
    private int size;           // subtree count

    public Node(Key key, Value val, boolean color, int size) {... }
  }

  private boolean isRed(Node x) {
    if (x == null) return false;
    return x.color == RED;
  }
  ...
}
```

# Red-Black Trees

- The **search** algorithm is the **same** as in **regular BST**
- That is, a top-down search following the ordering imposed by the keys

```
public Value get(Key key) {
   if (key == null)
       throw new IllegalArgumentException(”key is null");
   return get(root, key);
}

// value associated with the given key in subtree rooted at x;
// null if no such key
private Value get(Node x, Key key) {
   while (x != null) {
      int cmp = key.compareTo(x.key);
      if         (cmp < 0) x = x.left;
      else if (cmp > 0) x = x.right;
      else return x.val;
   }
   return null;
}
```

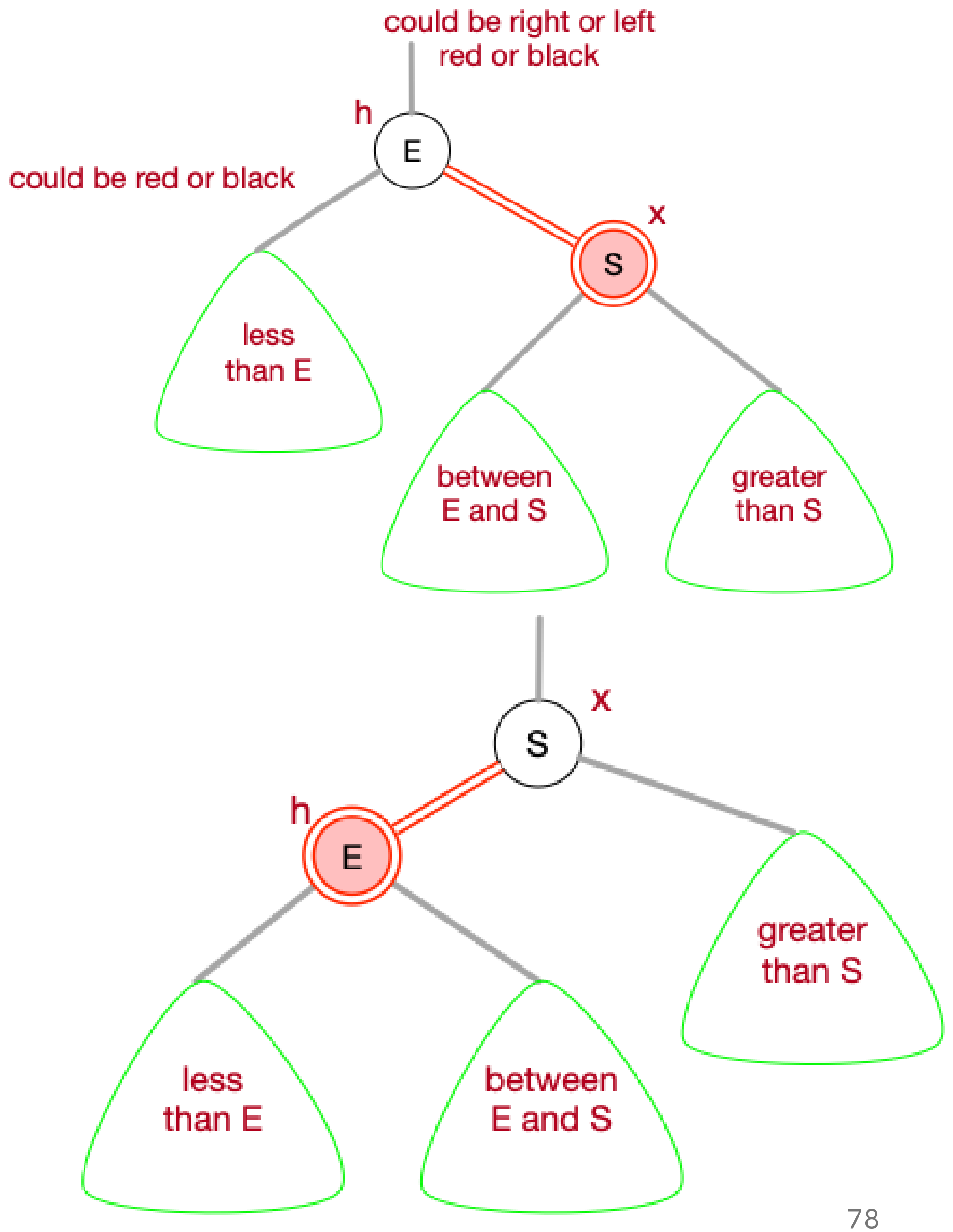
## Red-Black Trees

- All the modifying operations (e.g. insertion) must respect the **invariant** properties of the tree, that is
    1. **Red links lean left**
    2. **No node has two red links** connected to it
    3. **The tree has perfect black balance**: every path to the root to a null-link has the same number of blank links
- But, sometimes, in the middle of them, they will allow
    - Right leaning red links
    - Two red links in a row
    - A node with two red links to both children
- There are two operations that correct this: **rotations** (two versions) and **colour flipping.**
    - All these operations are local ($\mathcal{O}(1)$) and can propagate up the tree, which has **logarithmic height**

# Red-Black Trees

```
// make a right-leaning link lean to the left
private Node rotateLeft(Node h) {
   assert (h != null) && isRed(h.right);
   // assert (h != null) && isRed(h.right) && !isRed(h.left);
   // for insertion only
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   x.size = h.size;
   h.size = size(h.left) + size(h.right) + 1;
   return x;
}
```

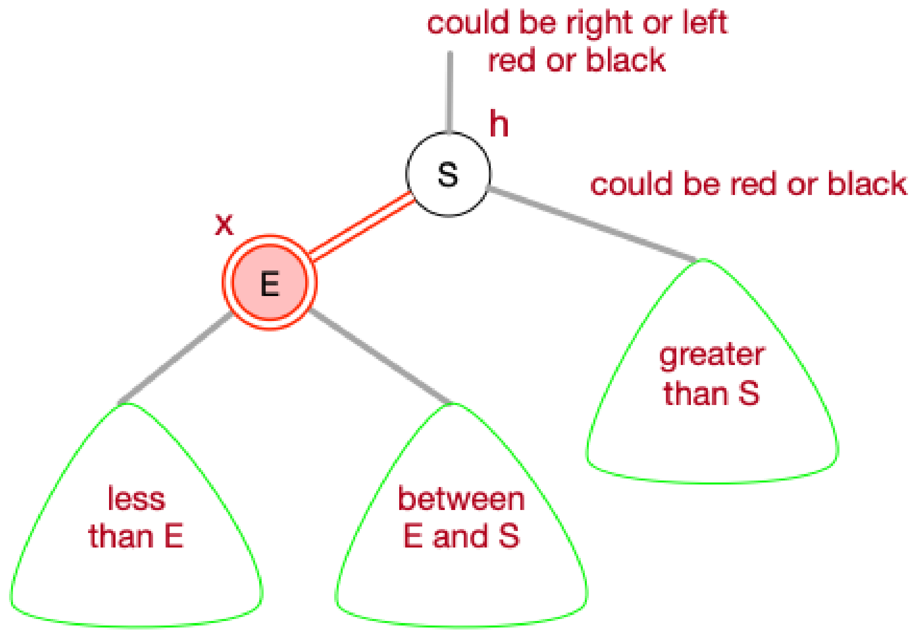
returns the new root

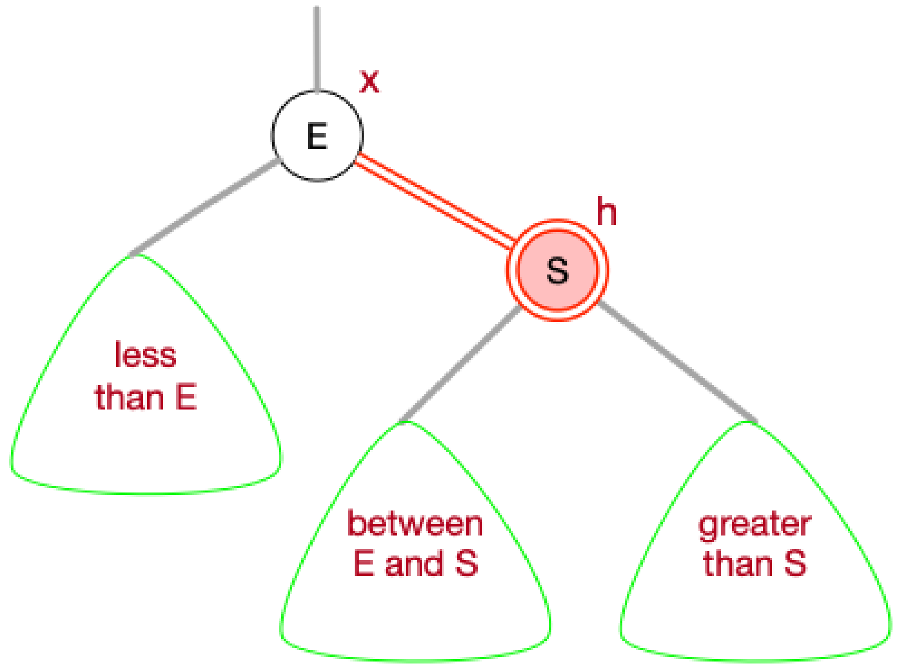


## Red-Black Trees

```
// make a left-leaning link lean to the right
private Node rotateRight(Node h) {
   assert (h != null) && isRed(h.left);
   // assert (h != null) && isRed(h.left) &&  !isRed(h.right);
   // for insertion only
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   x.size = h.size;
   h.size = size(h.left) + size(h.right) + 1;
   return x;
}
```

returns the new root



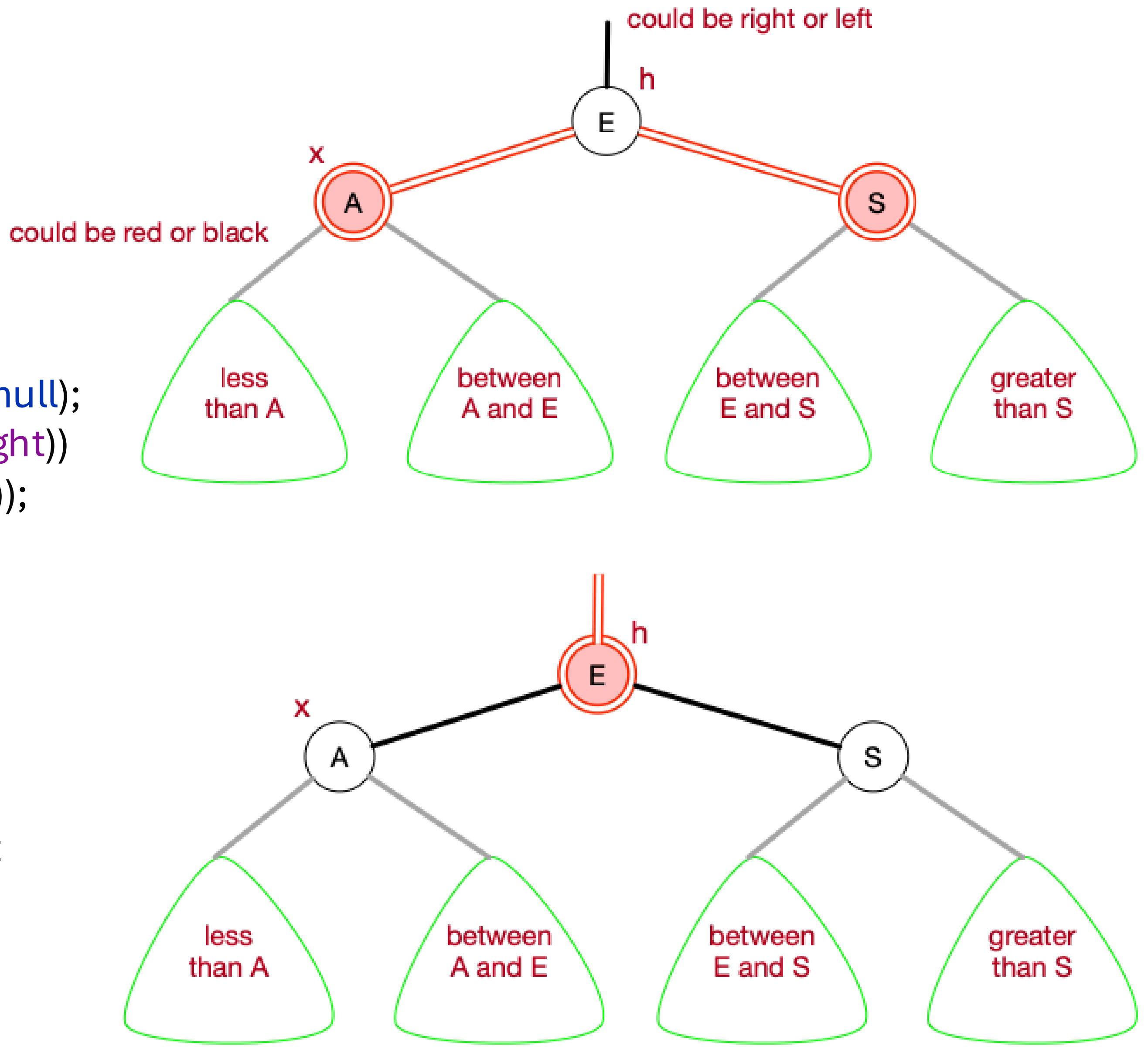


# Red-Black Trees

```
// flip the colors of a node and its two children
private void flipColors(Node h) {
   // h must have opposite color of its two children
   assert (h != null) && (h.left != null) && (h.right != null);
   assert (!isRed(h) &&  isRed(h.left) &&  isRed(h.right))
      || (isRed(h)  && !isRed(h.left) && !isRed(h.right));
   h.color = !h.color;
   h.left.color = !h.left.color;
   h.right.color = !h.right.color;
}
```

We can interpret flip as splitting a 4-node into two 2-nodes and inserting on the parent

- red link attaches middle node to parent
- black links are the resulting left and right 2-nodes after the split



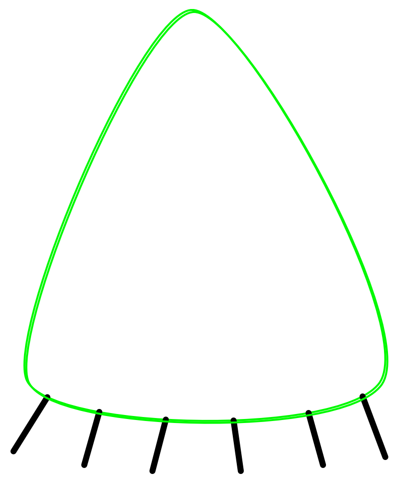
# Red-Black Trees

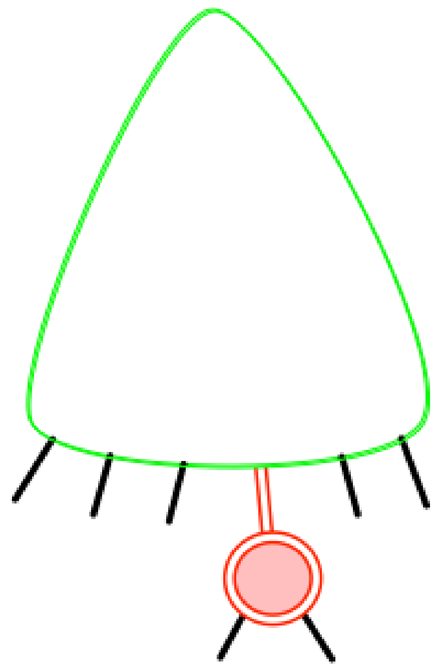
- The insertion algorithm is almost the same as BST’s
  - Doing a **top-down** search for finding the **insertion node**
  - If a new node must be created, it is inserted as a **leaf**
- The difference is that, **bottom-up**, when the recursive calls return, the algorithm will **restore the invariant properties** if they’re violated

```
public void put(Key key, Value val) {
   if (key == null)
         throw new IllegalArgumentException(”key is null");

   root = put(root, key, val);
   root.color = BLACK;
}

// insert the key-value pair in the subtree rooted at h
private Node put(Node h, Key key, Value val) {
   if (h == null) return new Node(key, val, RED, 1);

   int cmp = key.compareTo(h.key);
   if         (cmp < 0) h.left   = put(h.left,  key, val);
   else if (cmp > 0) h.right = put(h.right, key, val);
   else                        h.val    = val;

   // TODO: restore invariant

   return h;
}
```

## Red-Black Trees

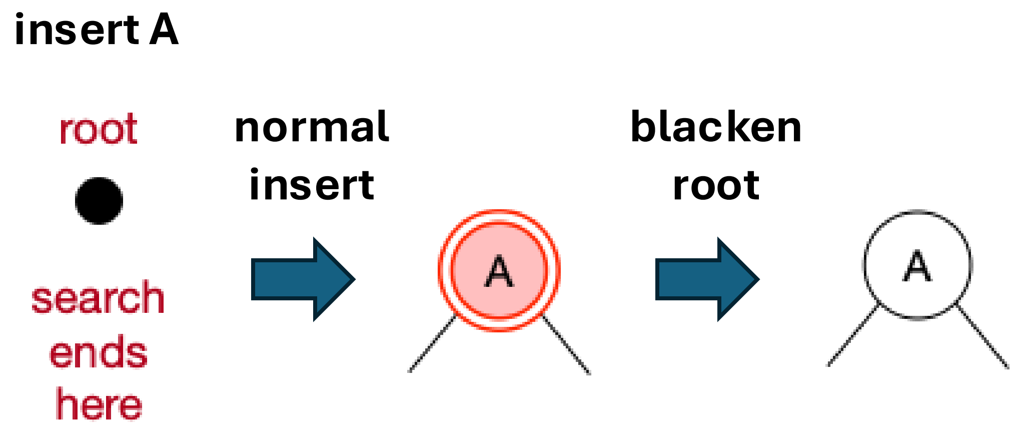
- Inserting the **new node** as **RED**, ensured that the **perfect black balance** property is not violated
  - All null-links continue to be at the same black distance to root
- After the insertion, the **root node** is always coloured as **BLACK**
  - A red root node makes no sense





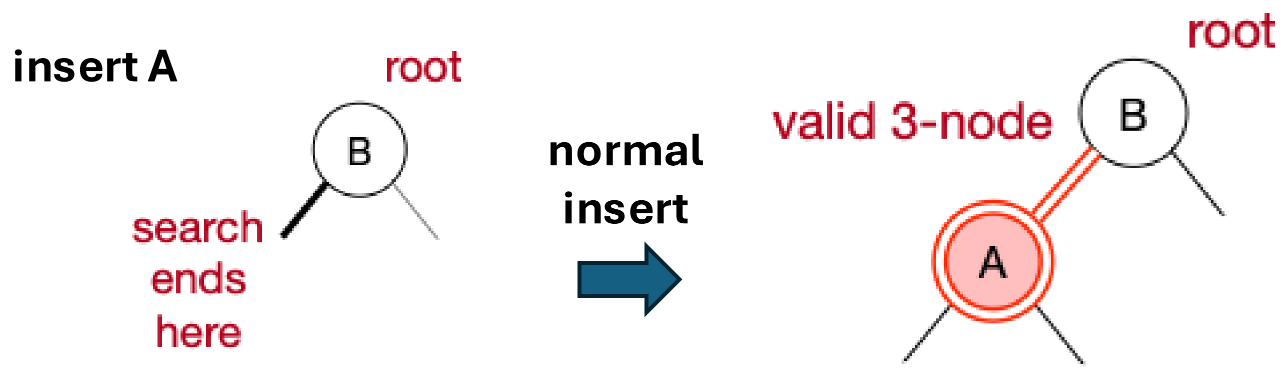
## Red-Black Trees

- Let's begin with three easy cases:
  - Inserting to an **empty tree**:
    - simply create the new root and make it black



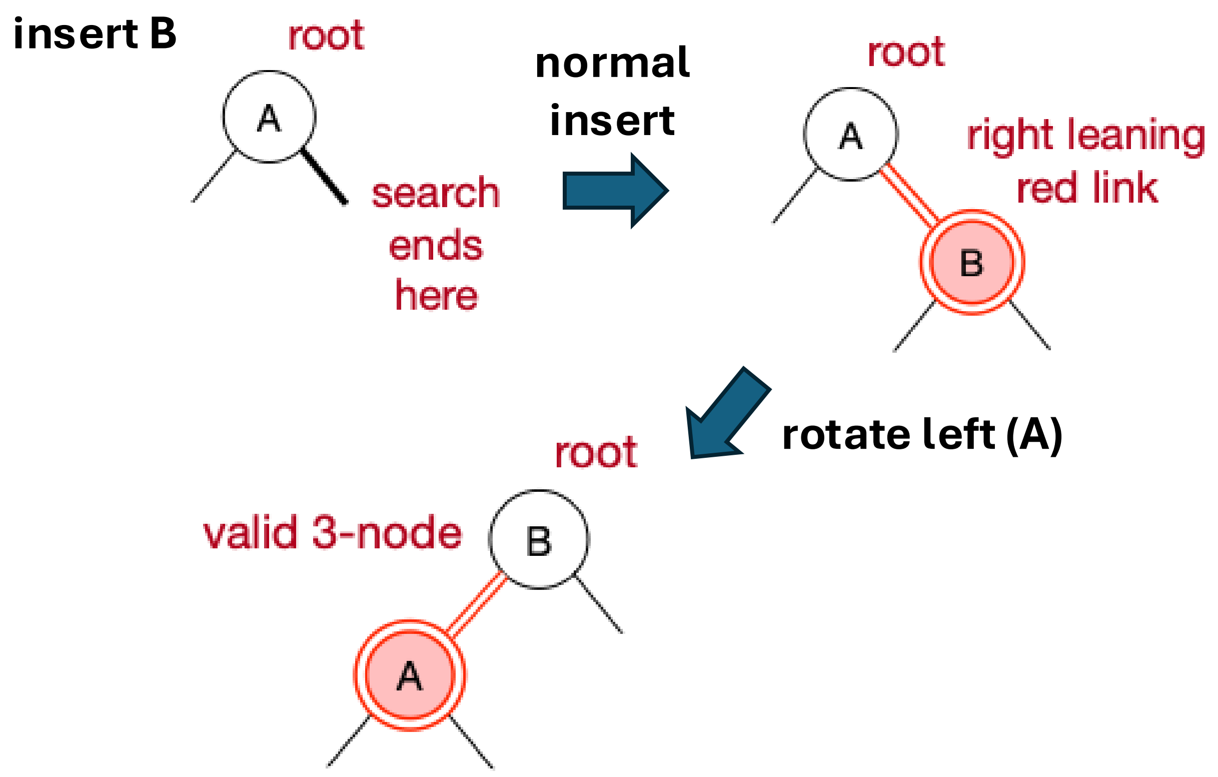
# Red-Black Trees

- Let’s begin with three easy cases:
  - Inserting to an **empty tree**:
    - simply create the new root and make it black
  - Inserting to a tree with a **single 2-node**
    - Insert as **left child**:
      - the new RED node makes the root node a 3-node



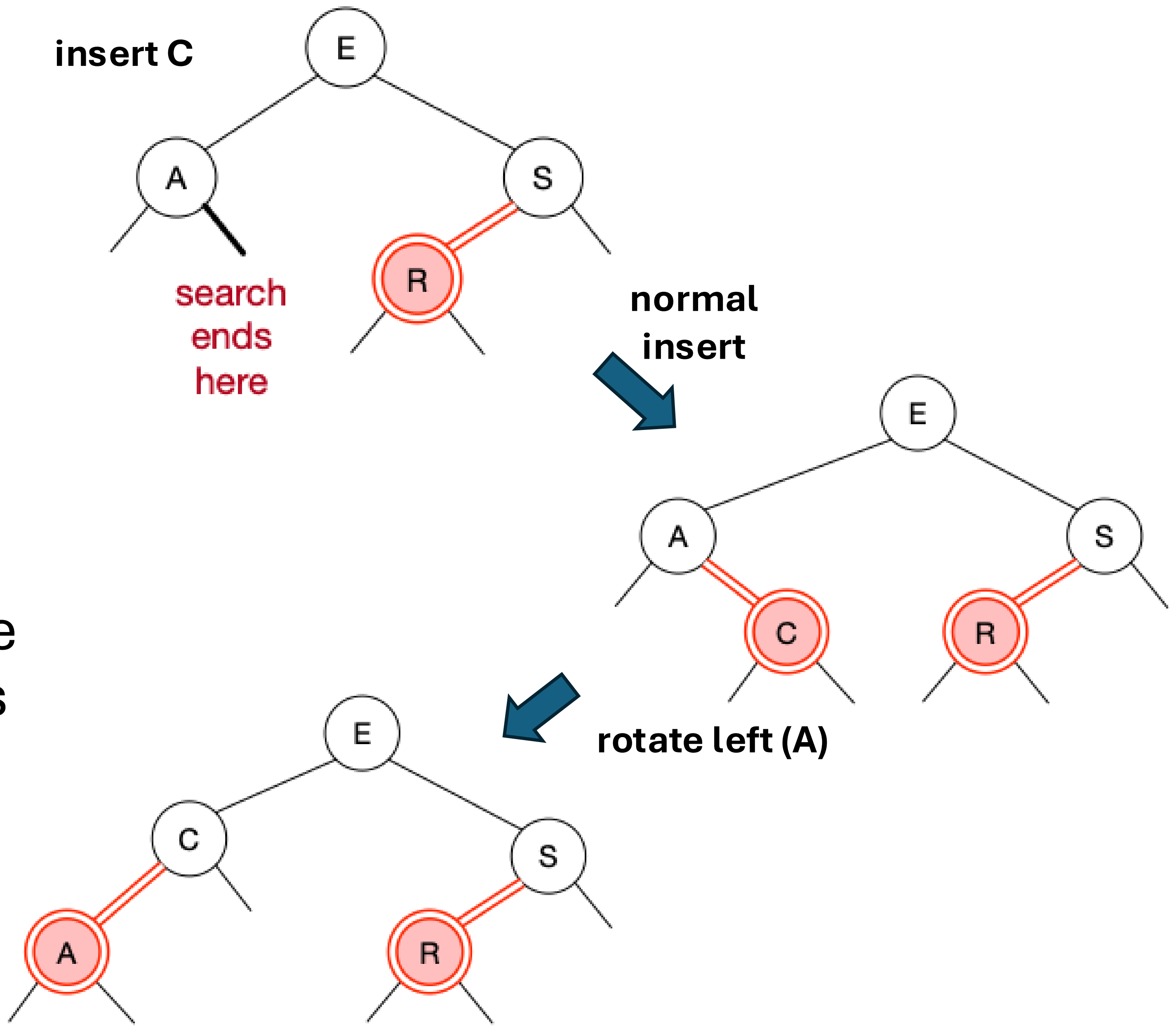
# Red-Black Trees

- Let’s begin with three easy cases:
  - Inserting to an **empty tree**:
    - simply create the new root and make it black
  - Inserting to a tree with a **single 2-node**
    - Insert as **left child**:
      - the new RED node makes the root node a 3-node
    - Insert as **right child**:
      - the new RED node makes a right leaning link
      - so, we need to do a **left rotation** (of the parent of the new node)



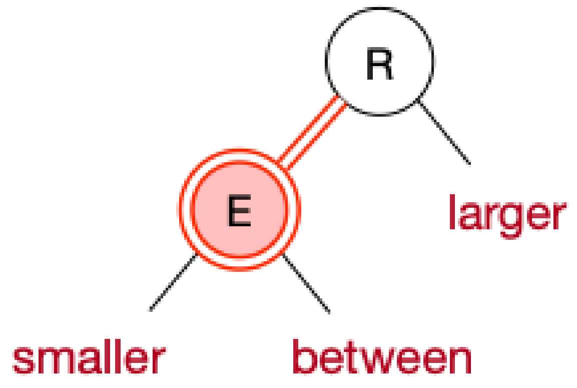
## Red-Black Trees

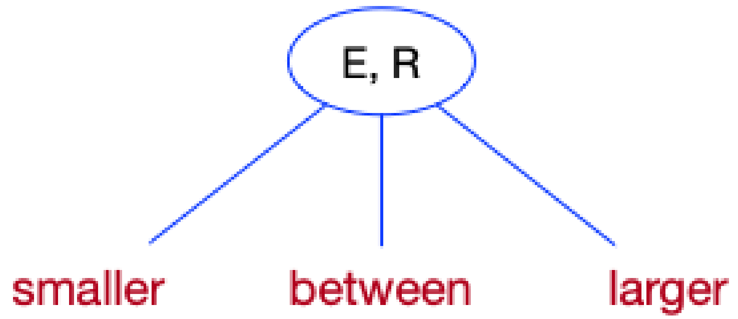
- The same situation happens when adding to 2-nodes at the bottom of the tree.
- Let's show only the case when a left rotation of the parent of the new node is needed



# Red-Black Trees

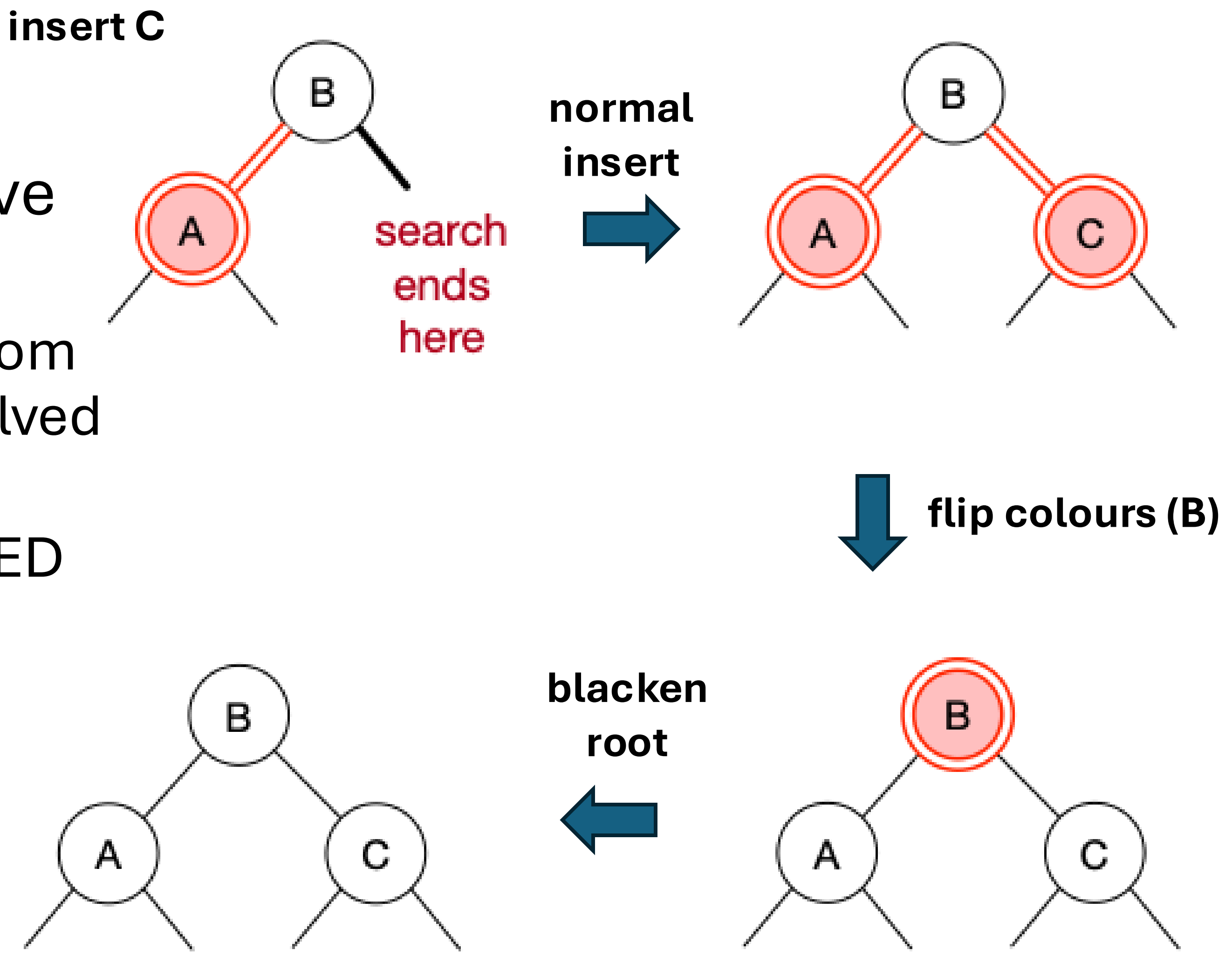
- Let's consider when the tree is a single 3-node
- There are three possibilities in this case:
  - The new key is **larger** than those in the tree
  - The new key is **smaller** than those in the tree
  - The new key is **between** those in the tree
- NOTE: These are the same cases we had when analysing insertion in 2-3 Trees





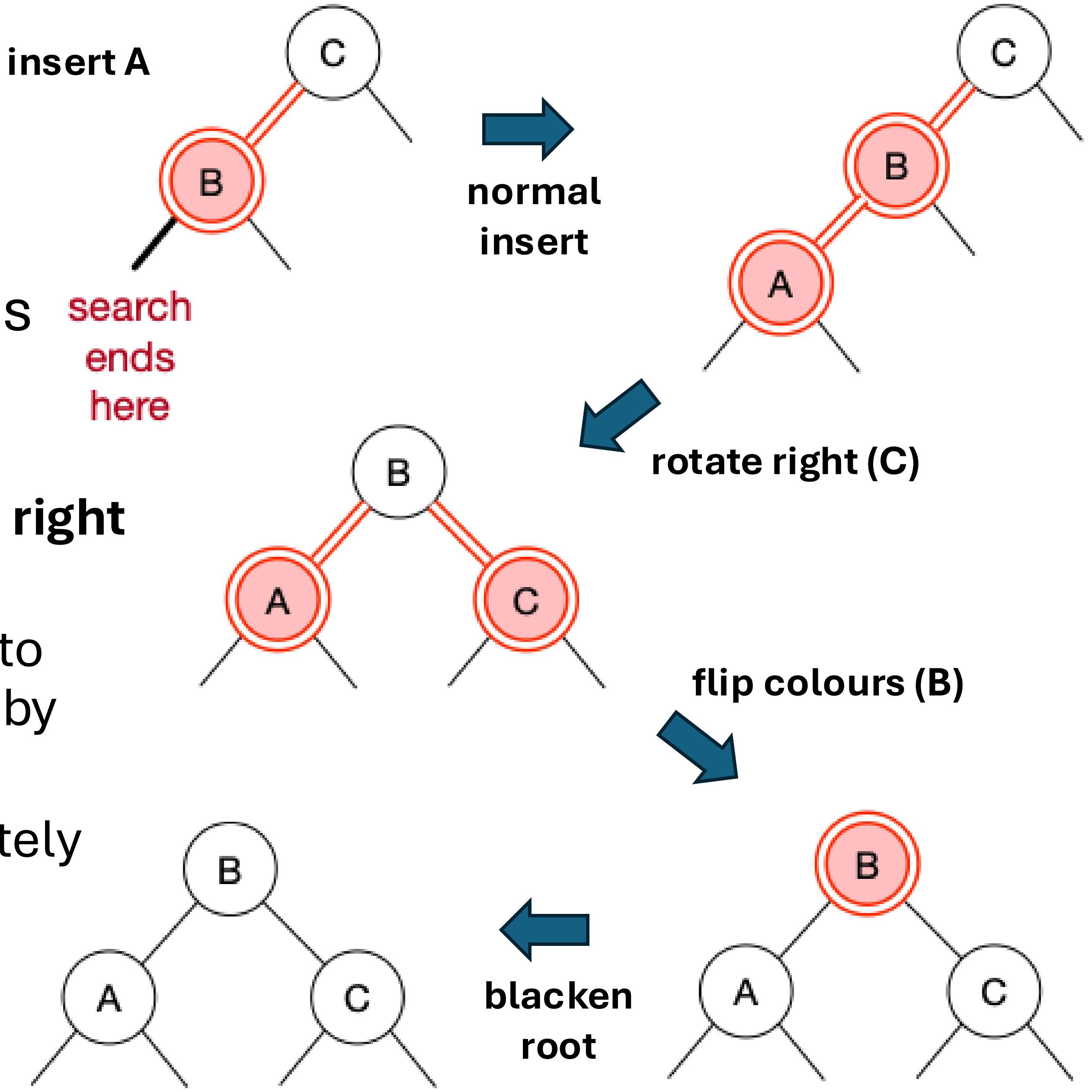
## Red-Black Trees

- When the key is **larger**, we have the easiest case
  - We have to RED links coming from the same parent that can be solved by **flipping the colours**
  - The **root** is temporarily made RED
  - But it's immediately restored restored to **BLACK**



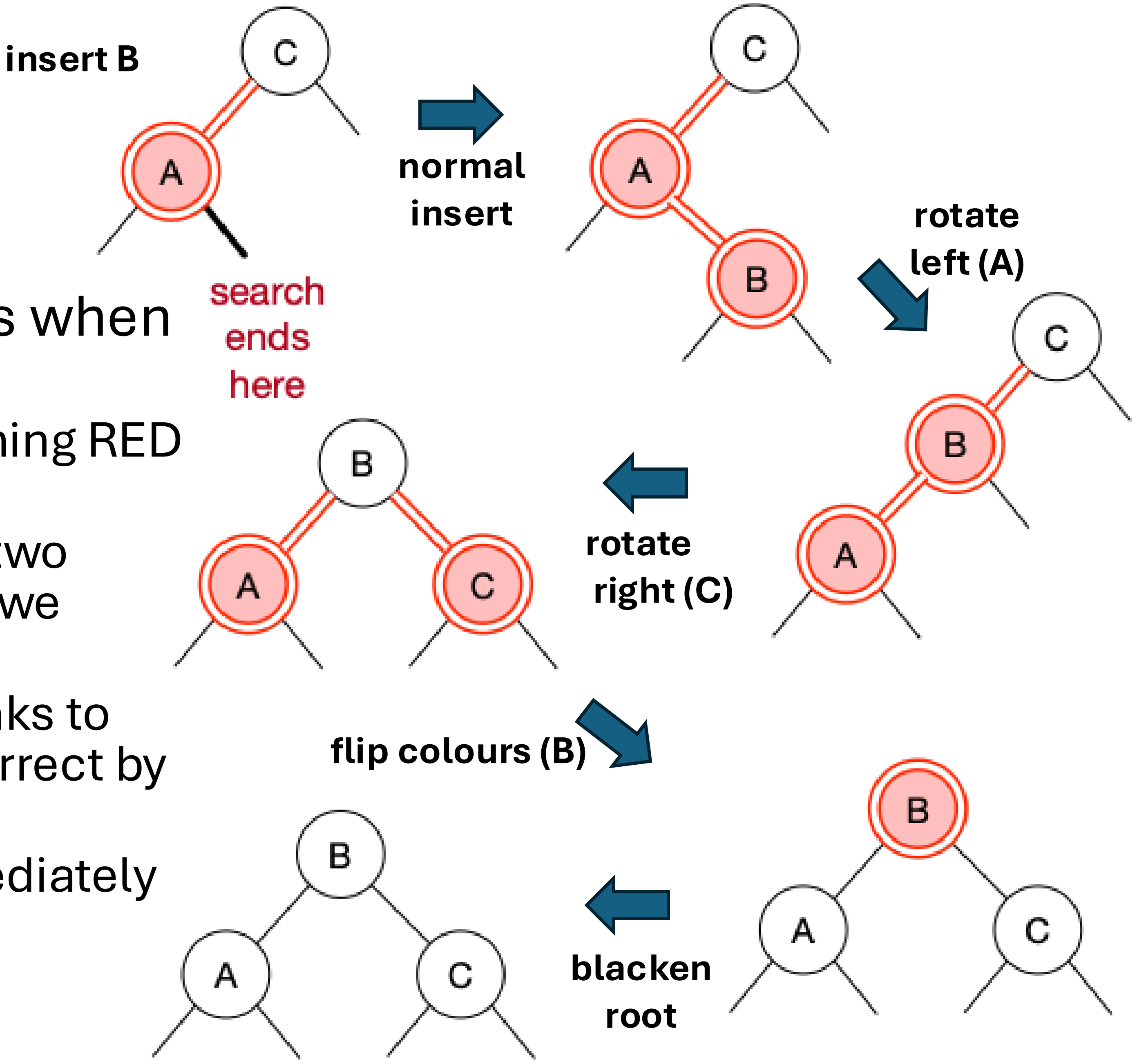
# Red-Black Trees

- The next case we'll consider is when the key is **smaller**
  - First, to avoid a chain of two consecutive RED links we do a **right rotation**
  - But this creates two RED links to the same parent that we solve by **flipping colours**
  - As before, the **root** is immediately changed to **BLACK**



## Red-Black Trees

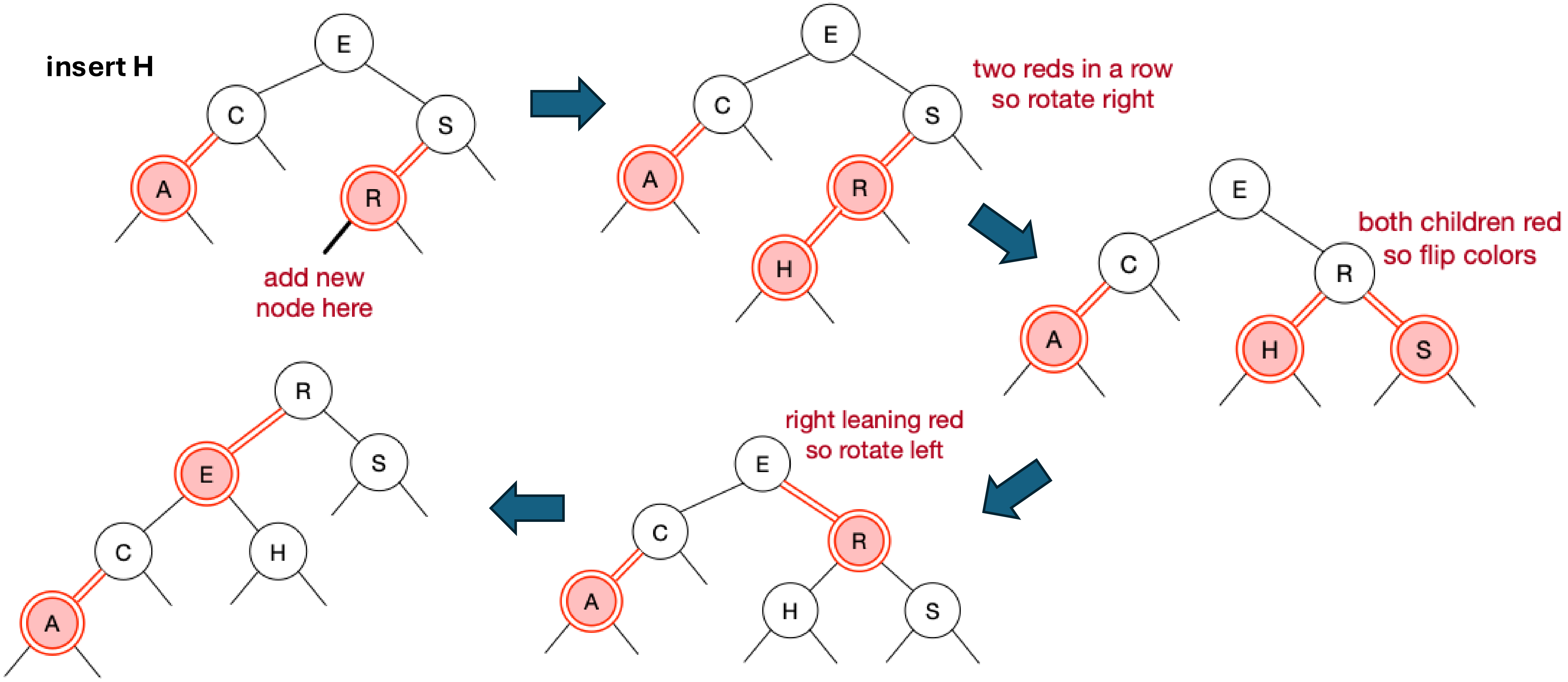
- The last case to consider is when the key is  in **between**
  - First, we solve the right leaning RED link with a **left rotation**
  - This causes two a chain of two consecutive RED links that we solve by a **right rotation**
  - But this creates two RED links to the same parent that we correct by **flipping the colours**
  - As always, the **root** is immediately changed to **BLACK**



## Red-Black Trees

- The only case we must consider is **inserting in a 3-Node** that is at the **bottom** of the tree
- Locally, we'll have the **same three cases**, and we'll proceed as when the 3-Node was the only node in the tree (the root)
  - NOTE: **All the three cases ended by a flipping colours operation**
- So, what we'll be the **difference** if any?
  - That now, as **the node is no longer the root**, it **won't be changed to BLACK**
  - So, **a RED link is propagated up the tree**
- How will it be treated?
  - By the **beauty of recursion**: from the point of view of the parent, it will be as it was inserted as a new node (which are always RED)
  - So, the very structure of the recursion will take care of that !!

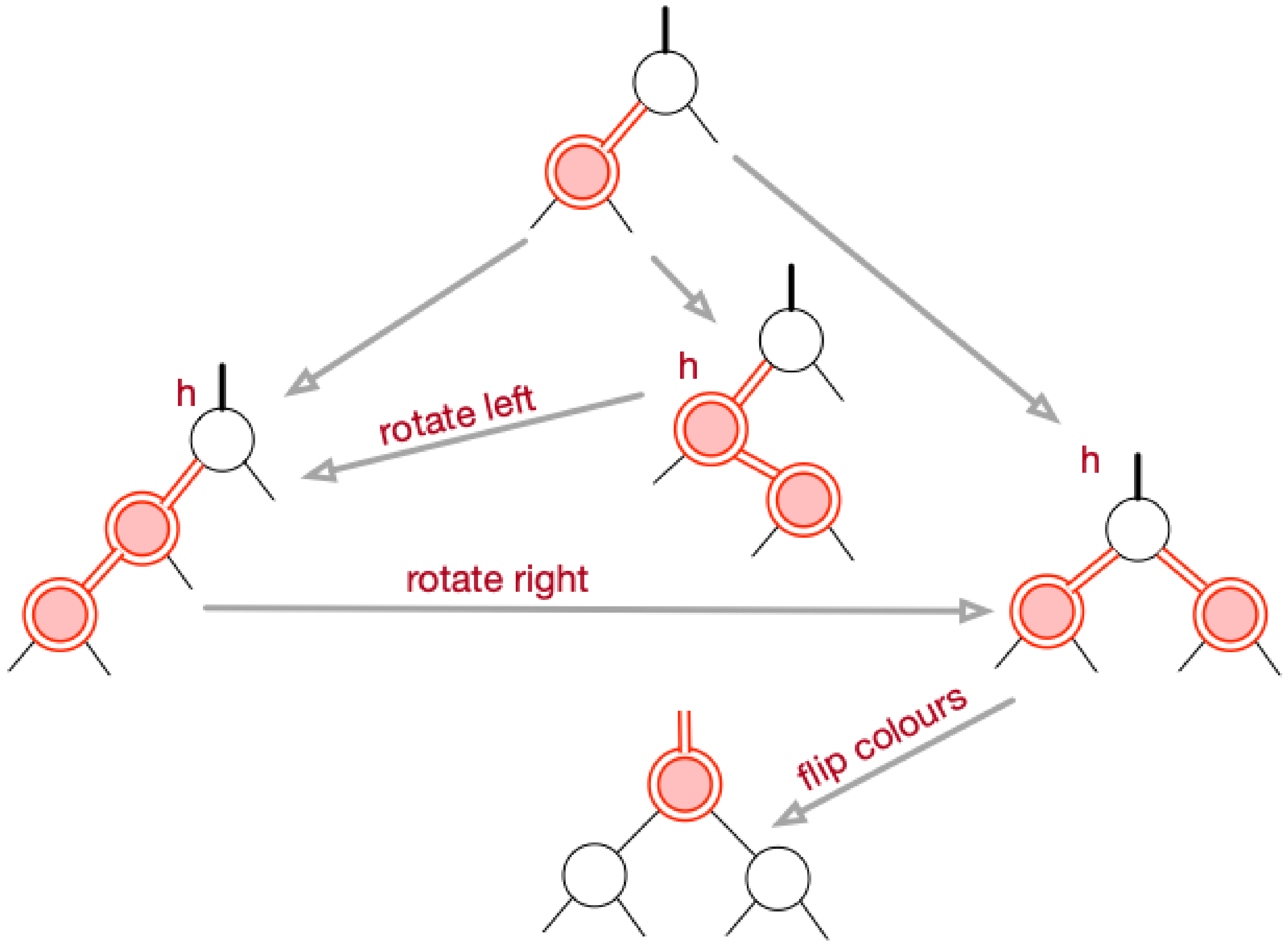
# Red-Black Trees


insert H
E
C
S
A
R
add new node here
two reds in a row so rotate right
H
both children red so flip colors
right leaning red so rotate left

## Red-Black Trees

We can sum-up all these three cases, the three possible insertion points, the transformations and the passing up of the RED link in a single diagram

NOTE: The returns to callers (parent node) are not shown and understanding them is key to mastering the algorithm.



# Red-Back Trees

- After doing the normal top-down insertion in a BST, we arrange bottom-up the invariant violations that we may have created
- And thanks to recursion,
  - when we produce a problem up to the tree ...
  - ... it will be taken care of when the current recursive call is ended ...
  - ... in the call corresponding to its parent

```
private Node put(Node h, Key key, Value val) {
   if (h == null) return new Node(key, val, RED, 1);

   int cmp = key.compareTo(h.key);
   if        (cmp < 0) h.left  = put(h.left,  key, val);
   else if (cmp > 0) h.right = put(h.right, key, val);
   else                      h.val   = val;

   // fix-up any right-leaning links
   if (isRed(h.right) && !isRed(h.left))       h = rotateLeft(h);
   if (isRed(h.left)  &&  isRed(h.left.left)) h = rotateRight(h);
   if (isRed(h.left)  &&  isRed(h.right))     flipColors(h);

   h.size = size(h.left) + size(h.right) + 1;

   return h;
}
```

# B-Trees

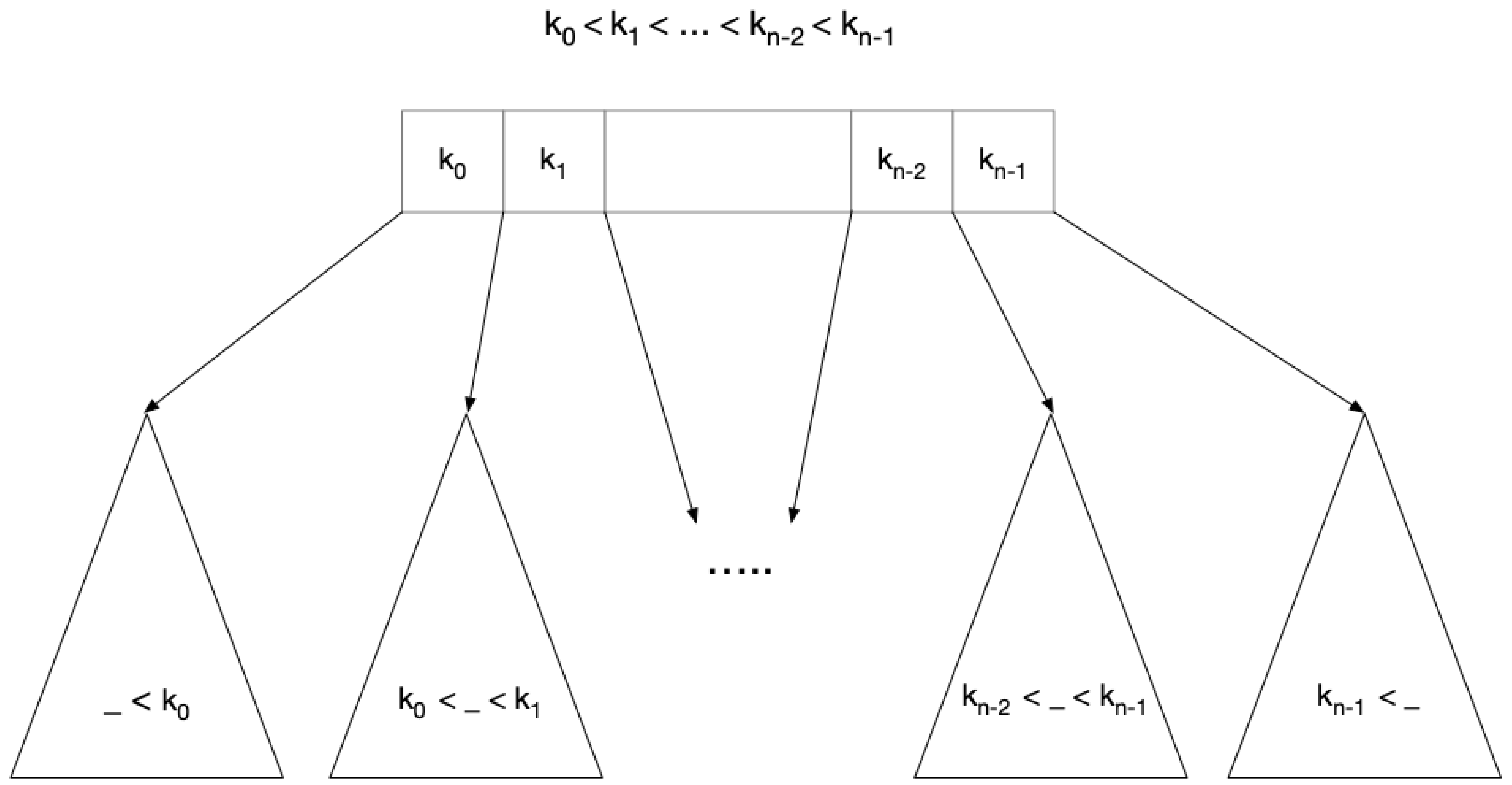
## B-Trees

- B-Trees are balanced search trees designed to work well on **disk drives** or other **direct-access secondary storage devices**
  - They're like red-black trees, but they are better at minimizing the number of data access operations
  - They differ from red-black trees in that nodes can have many children (from a few to hundreds), *i.e.* **the branching factor can be quite large**
- So, B-Trees
  - Generalize binary search trees
  - And their insert and remove operations leave the B-Tree balanced
  - So, the height of the tree is $\mathcal{O}(\log n)$
- As we did in the case of binary search trees, we'll only show the keys in the tree because the value associated to it is simply a payload.

## B-Trees

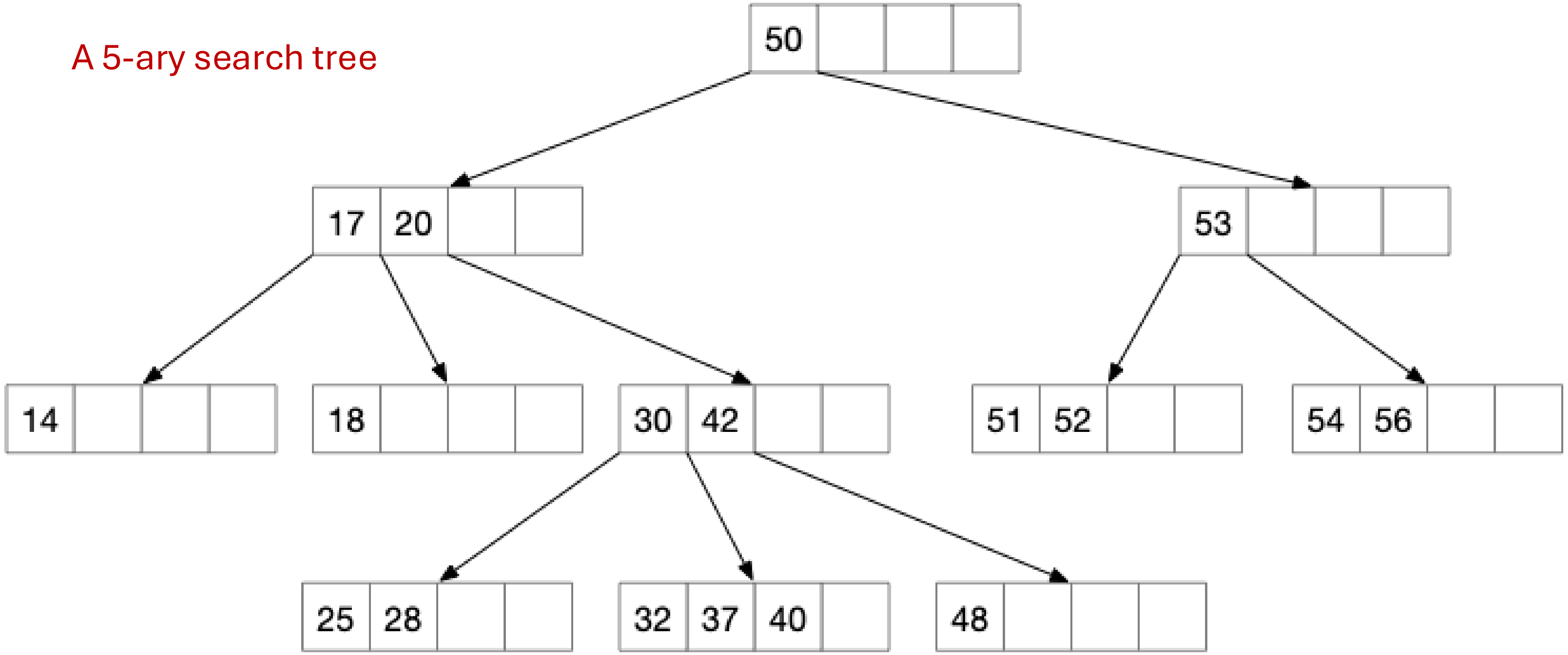
- An **m-ary search tree** generalizes binary search trees by
  - $\boldsymbol{m}$ is the order of the tree, that is, the maximum degree of any node
  - $\mathbf{n}$ is the number of keys associated with any node ($n < m$) and ($n + 1$) is the number of children of this node
    - $\mathbf{k_0, k_1, \cdots, k_{n-1}}$ are the keys associated with a node
  - The keys at each node are sorted increasingly so $\boldsymbol{k_i} < \boldsymbol{k_{i+1}}$
  - The properties of a search tree are respected, so that for a key $\boldsymbol{k_i}$
    - All keys in the **first $\boldsymbol{i}$ subtrees** are **smaller**
    - All keys in the **last $\boldsymbol{n - i}$ subtrees** are **bigger**

## B-Trees



## B-Trees

A 5-ary search tree



## B-Trees

- m-ary search trees have the same problem as simple binary search trees
  - They can be unbalanced
  - So, insert / delete / search operations are not logarithmic
- **B-Trees are balanced m-ary search trees**
- They were developed by Rudolf Bayer & Edward W, McCreight in 1970
  - Organization and Maintenance of Large Ordered Indices, SIGFIDET Workshop 1970: 107-141
- **NOTE**: There are many variations of them, and we'll follow Ribó's presentation

## B-Trees

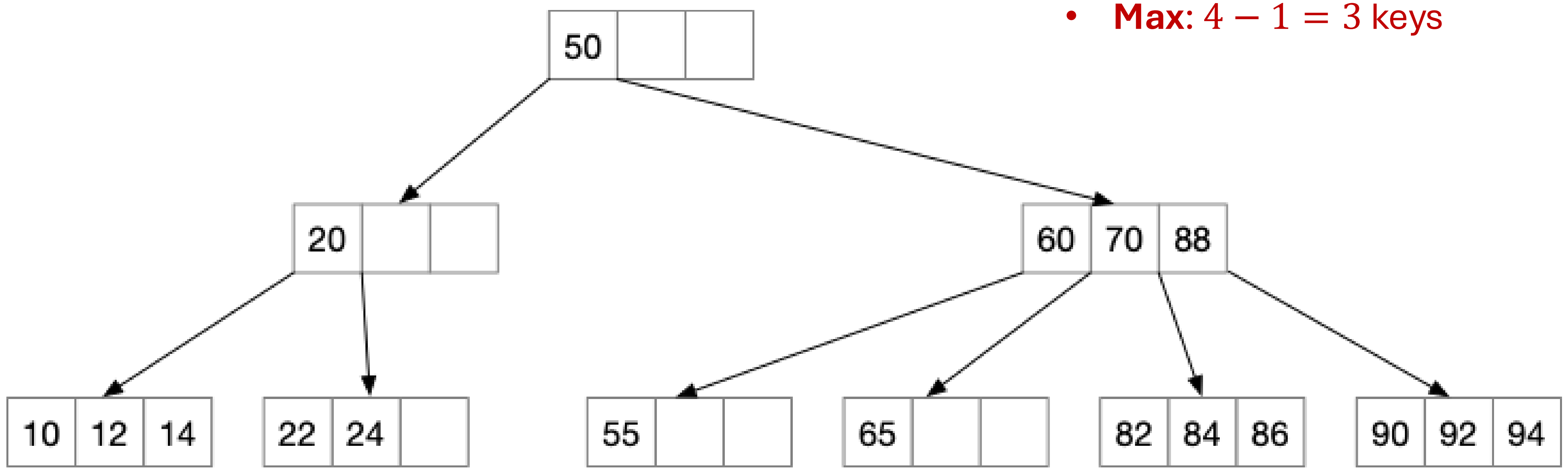
- A **B-Tree** of **order $\boldsymbol{m}$** (with $m \geq 3$) is a **m-ary search tree** such that
  - **The root** must have **at least 2** children (and **1** key)
    - Unless it is a leaf, or the b-tree is empty
  - **All non-root nodes** must have **at least $\boldsymbol{\left\lceil \frac{m}{2} \right\rceil - 1}$ keys**
    - So, if it's an **internal** node, it'll have **at least $\boldsymbol{\left\lceil \frac{m}{2} \right\rceil}$ children**
  - **All nodes** will have **at most $\boldsymbol{m - 1}$ keys**
  - **All leaves** are at the **same level**

- So, the cost of searching for a key in a B-Tree of order $m$ and size $n$ is $O(\log_{\left\lceil \frac{m}{2} \right\rceil} n)$

## B-Trees

B-Tree of order **4**

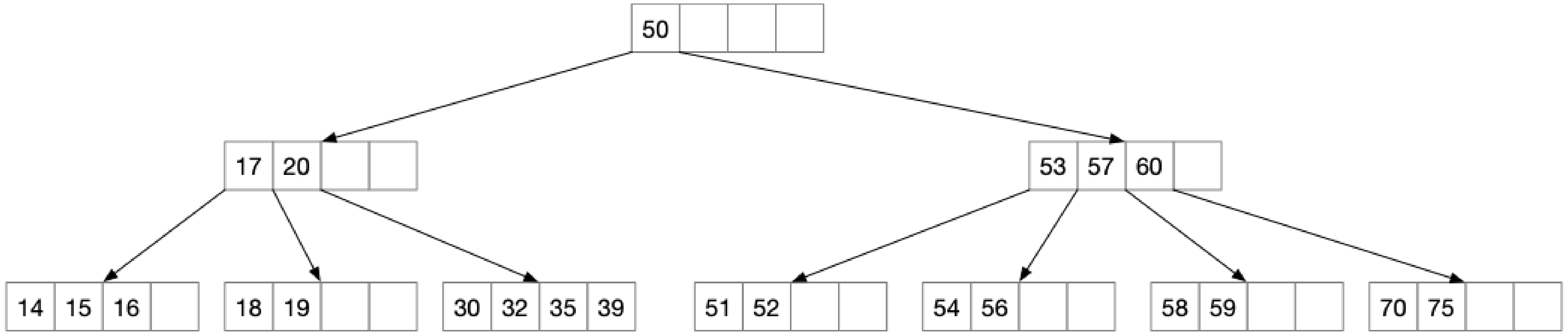
- **Min:** $\left\lceil \frac{4}{2} \right\rceil - 1 = 2 - 1 = 1$ key
- **Max:** $4 - 1 = 3$ keys



# B-Trees

B-Tree of order **5**

- **Min:** $\left\lceil \frac{5}{2} \right\rceil - 1 = 3 - 1 = 2$ keys
- **Max:** $5 - 1 = 4$ keys



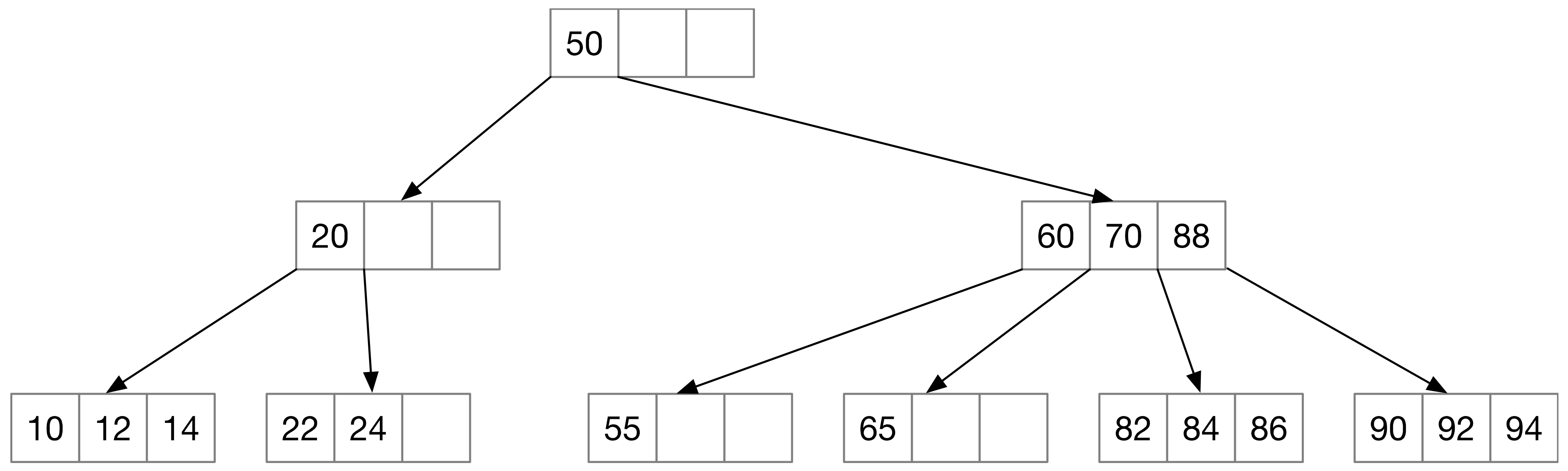
## B-Trees

- **Insertion of the pair: $\boldsymbol{k \rightarrow v}$**
    1. Search the key $k$ in the tree
    2. If the key is found, substitute the current value associated to it by $v$
    3. If not, consider the node (a leaf) $h$ in which to add the pair $(k, v)$
    4. If the node $h$ is not full, we add the pair to it
    5. If it is,
        1. Split the leaf in two, considering also the new key to be added.
           The split consist in taking the median value, leaving the smaller keys in the node and creating a new node with the bigger keys.
        2. The median value (with the new node attached to it as right child) is inserted in the parent node, and the steps 3 & 4 are repeated with the parent as the $h$ node (which now it's not a leaf)
           **NOTE:** When moving up the key, if the key had already a right child, as its right child must be the new node, we must perform a *little arrangement* (a kind of *rotation*)

# B-Trees

Let’s consider the following B-Tree (**order** $\boldsymbol{m = 4}$)

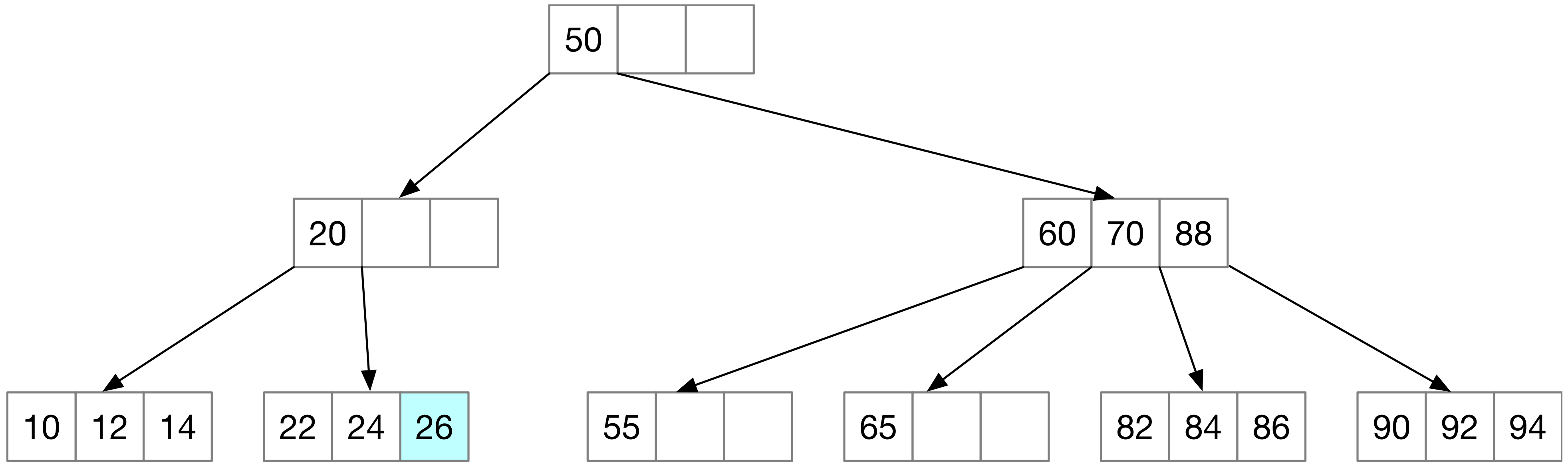
- **Minimum number ok keys** = $\left\lceil \frac{4}{2} \right\rceil - 1 = 2 - 1 = \mathbf{1}$



# B-Trees

Let’s **insert key 26**

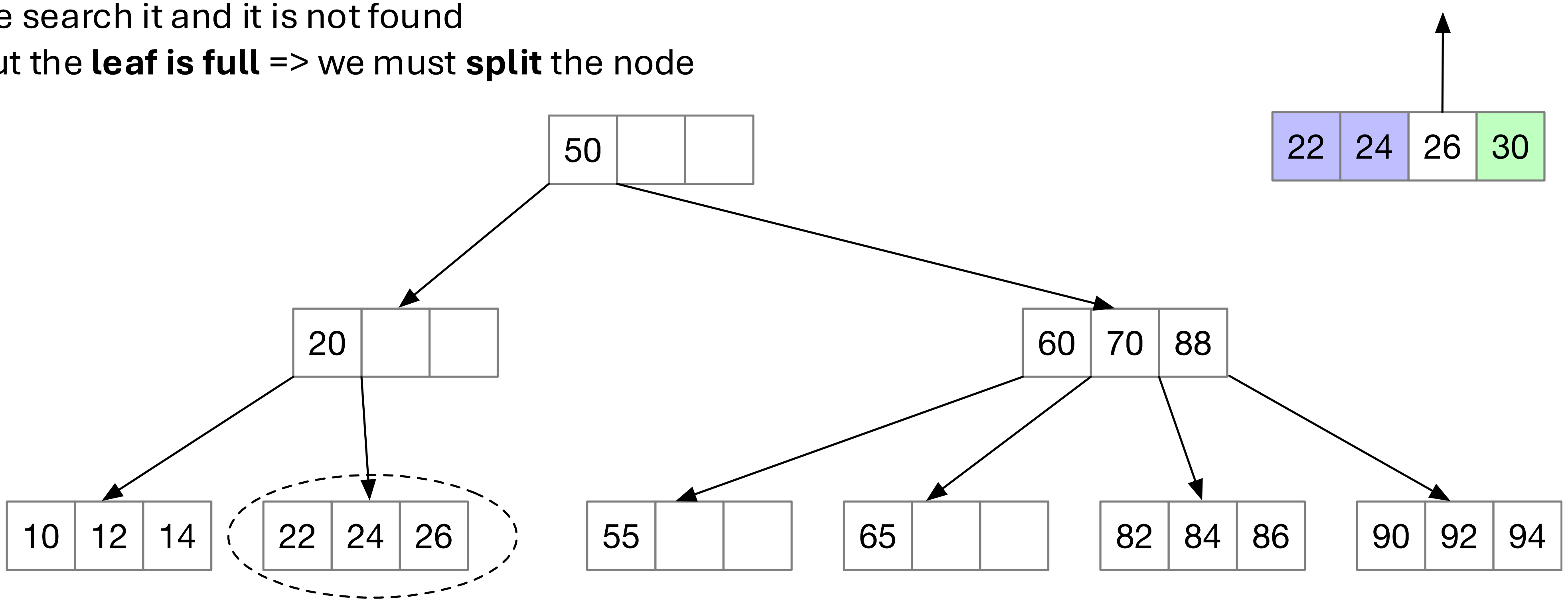
- we search it and it is not found
- **leaf** is **not full**, so we **add 26** to it



# B-Trees

Let’s **insert key 30**

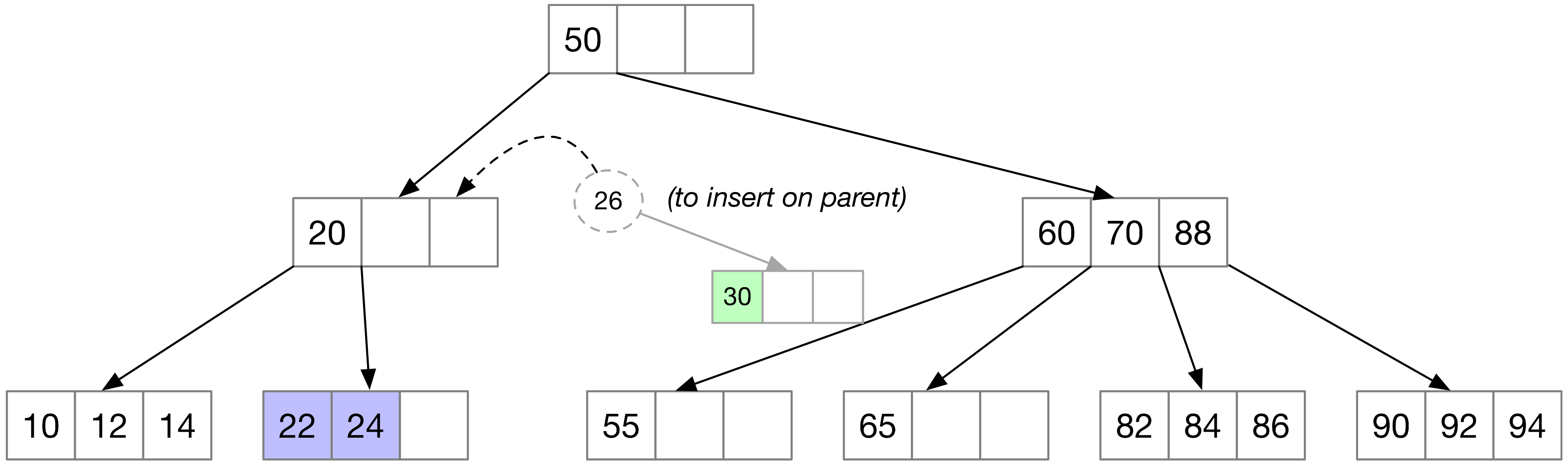
- we search it and it is not found
- but the **leaf is full** => we must **split** the node



# B-Trees

Let’s **insert key 30**

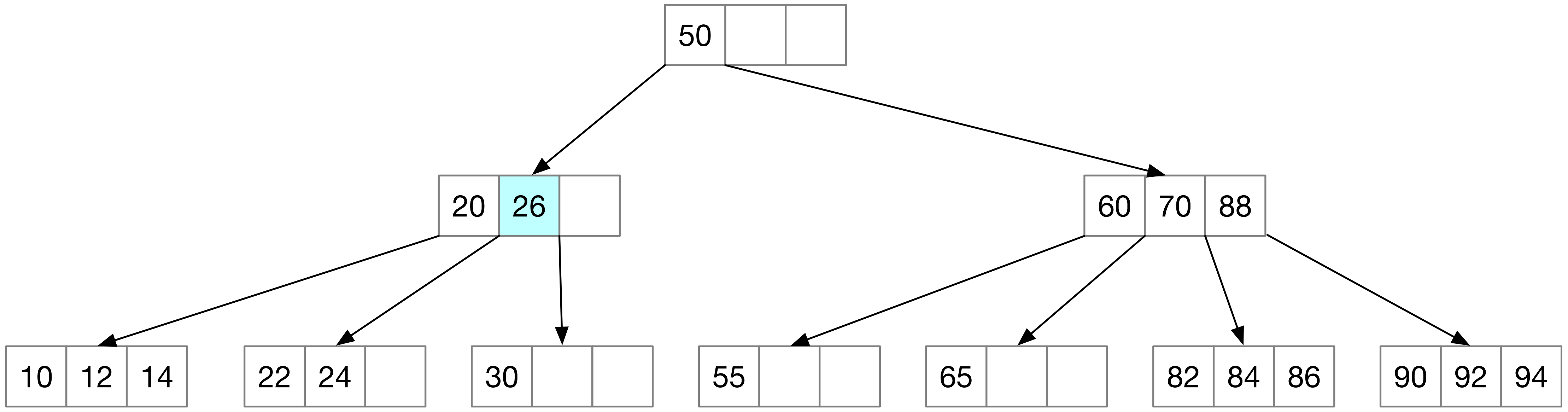
- we search it and it is not found
- but the **leaf is full** => we must **split** the node
- we **insert 26 on parent**
    - as 26 was on a left, it had no right child, so we don’t need any rearrangement



## B-Trees

Let’s **insert key 30**

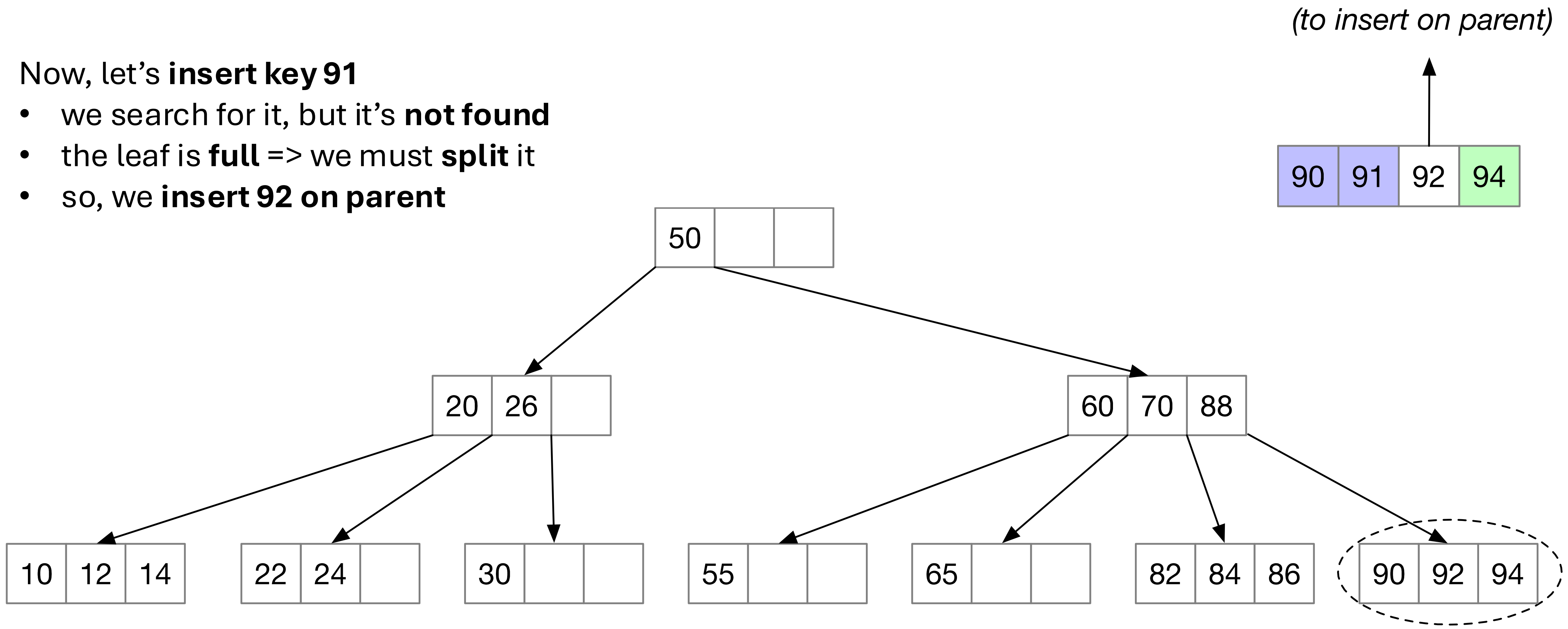
- Let’s **insert key 26** on parent
    - The node is **not full**, so we **add 26** to it



# B-Trees

Now, let’s **insert key 91**

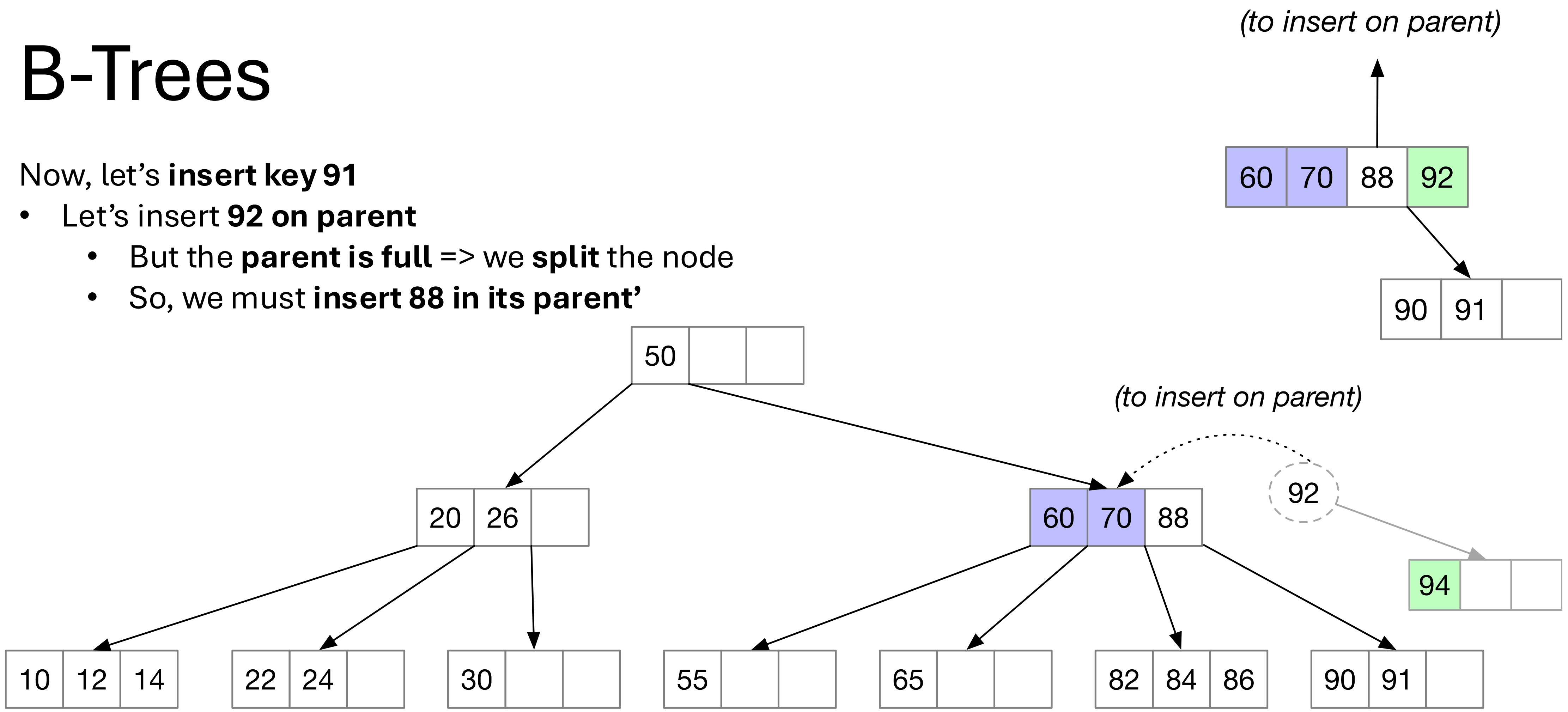
- we search for it, but it’s **not found**
- the leaf is **full** => we must **split** it
- so, we **insert 92 on parent**



# B-Trees

Now, let’s **insert key 91**

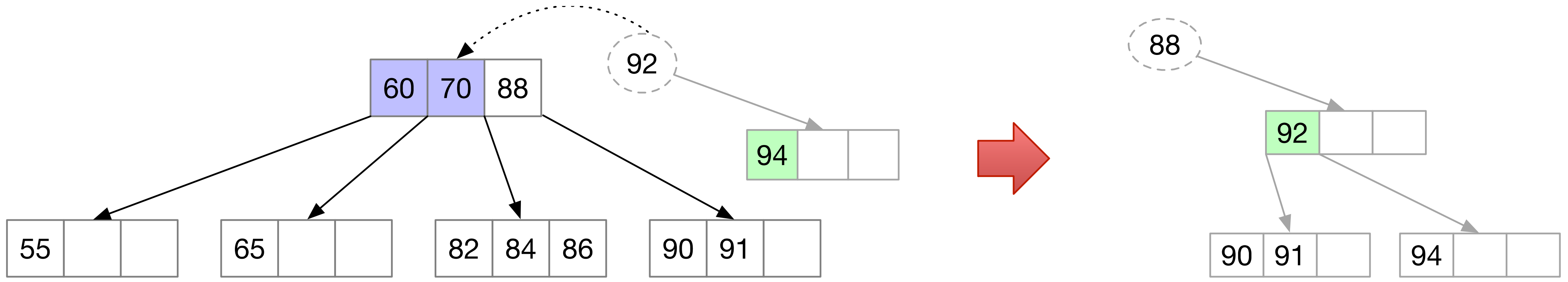
- Let’s insert **92 on parent**
    - But the **parent is full** => we **split** the node
    - So, we must **insert 88 in its parent’**



# B-Trees

Now, let’s **insert key 91**

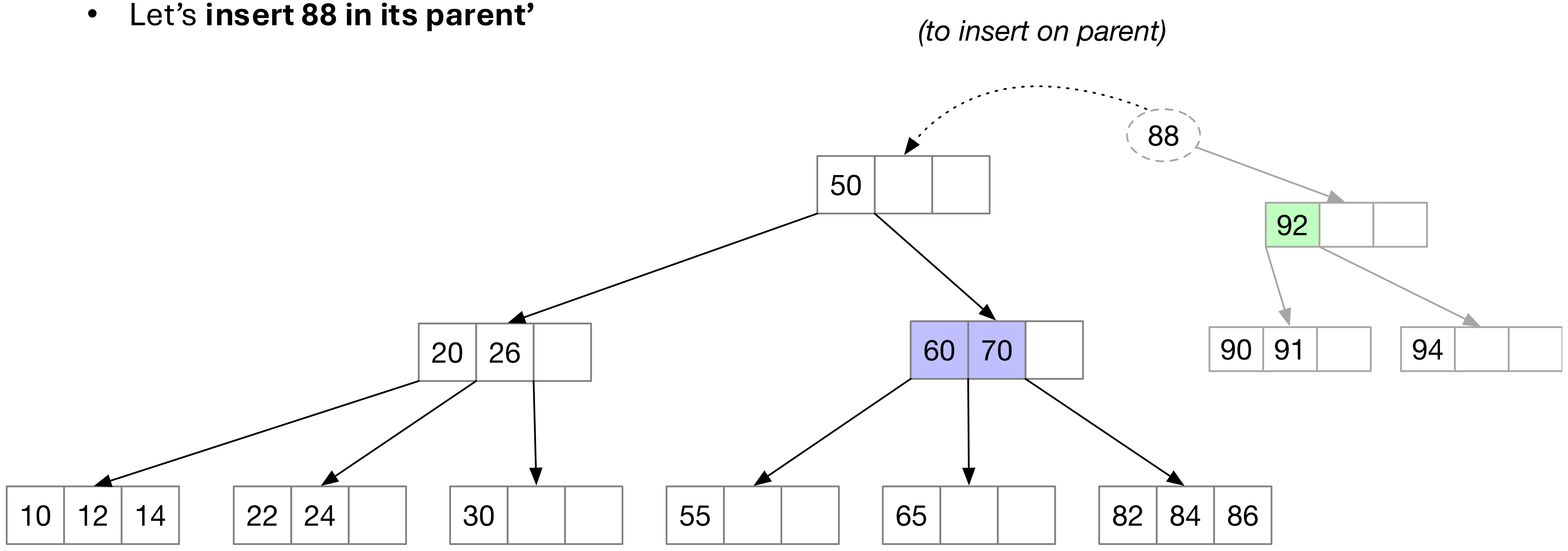
- Let’s insert **92 on parent**
    - But the **parent is full** => we **split** the node
    - So, we must **insert 88 in its parent’**
        - As the right child is the new node, we must do a rearrangement if 88 already has a right node



# B-Trees

Now, let’s **insert key 91**

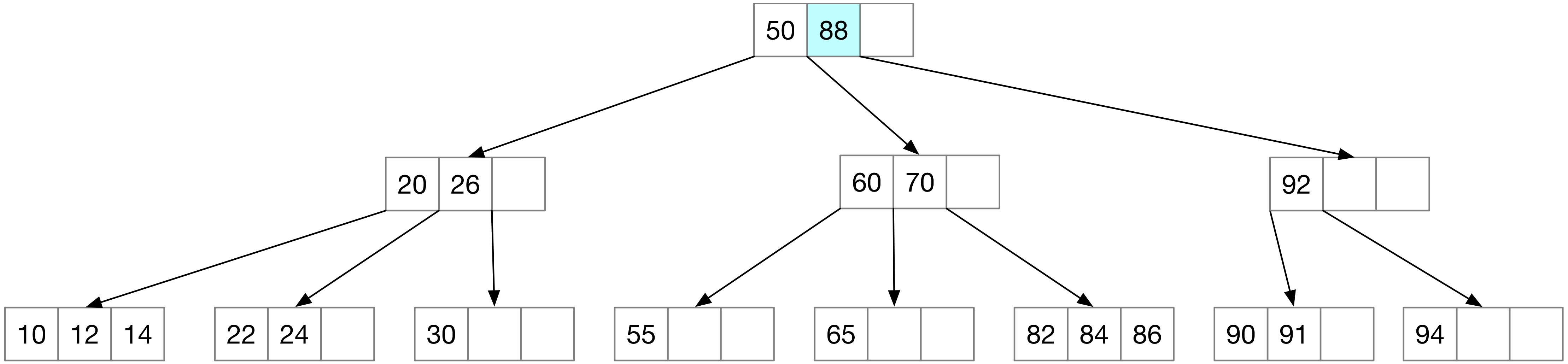
- Let’s insert **92 on parent**
    - Let’s **insert 88 in its parent’**



# B-Trees

Now, let’s **insert key 91**

- Let’s insert **92 on parent**
    - Let’s **insert 88 in its parent’**
        - The node is **not full**, so we **add** to it



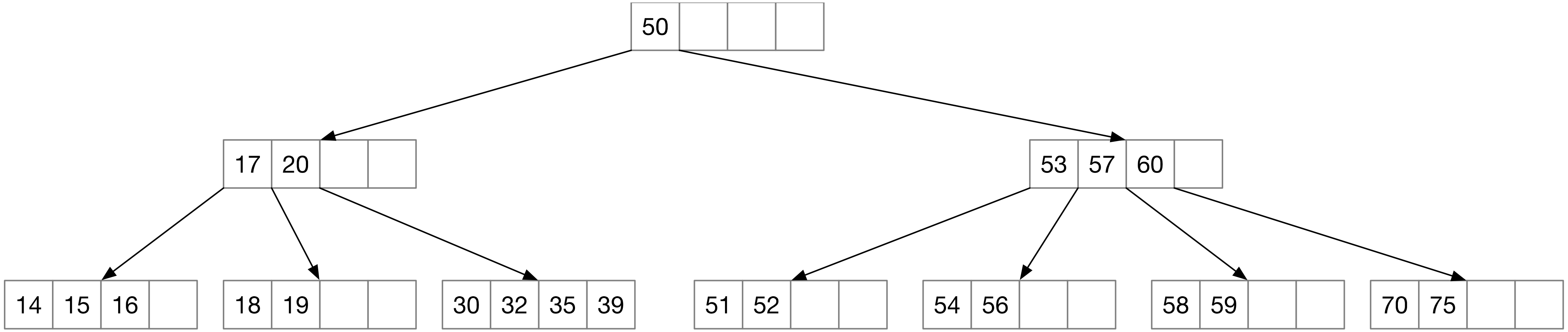
## B-Trees

- **Deletion of the key $\boldsymbol{k}$**
  1. Search the key $k$ in the tree
  2. If the key is found in an **internal** node
     - **i.** **Substitute** it by the pair $(\boldsymbol{k'}, \boldsymbol{v'})$, where $k'$ is the **next of key $\boldsymbol{k}$** in ascending order
     - **ii.** **Delete** (recursively) key $\boldsymbol{k'}$ (from the **leaf**)
  3. If the key is found in a **leaf** $node$, **remove** it from the leaf
     - i. If $node$ has **at least** $\boldsymbol{\left\lceil \frac{m}{2} \right\rceil - 1}$ keys remaining, we're **finished**
     - ii. If not, but there is and **adjacent sibling** with and **excess ok keys** (i.e. more than$\left\lceil \frac{m}{2} \right\rceil - 1$), then **redistribute keys with sibling and parent**
     - iii. If not, **merge** $node$ with an **adjacent sibling**. This will make the parent lose a child and a key. Return to 3.1 using the parent as $node$

# B-Trees

Let’s consider the following B-Tree (**order** $\boldsymbol{m} = \mathbf{5}$)

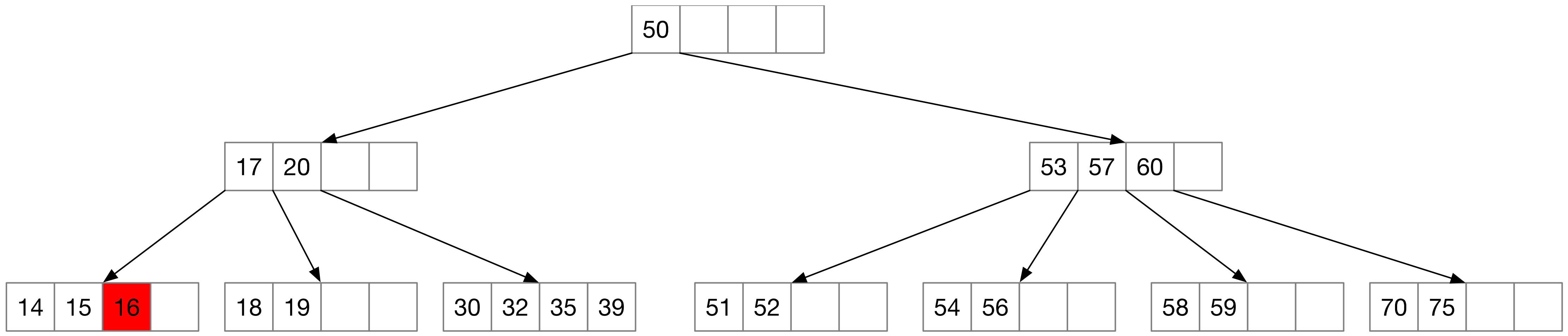
- **Minimum number ok keys** = $\left\lceil \frac{5}{2} \right\rceil - 1 = 3 - 1 = \mathbf{2}$
- node with **3 or 4** keys **-> excess**
- node with **1** key **-> shortage**



# B-Trees

Let's **delete key 16**

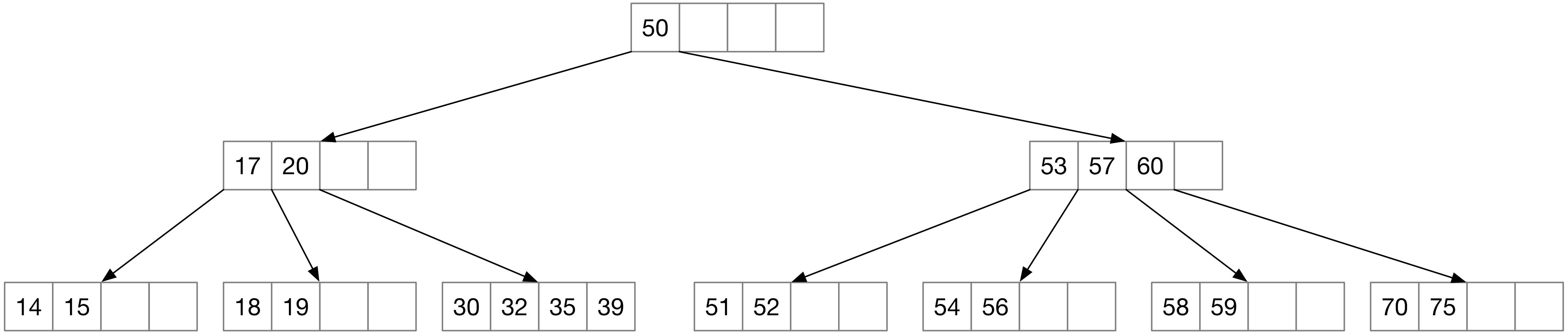
- we find it on a leaf



# B-Trees

Let’s **delete key 16**

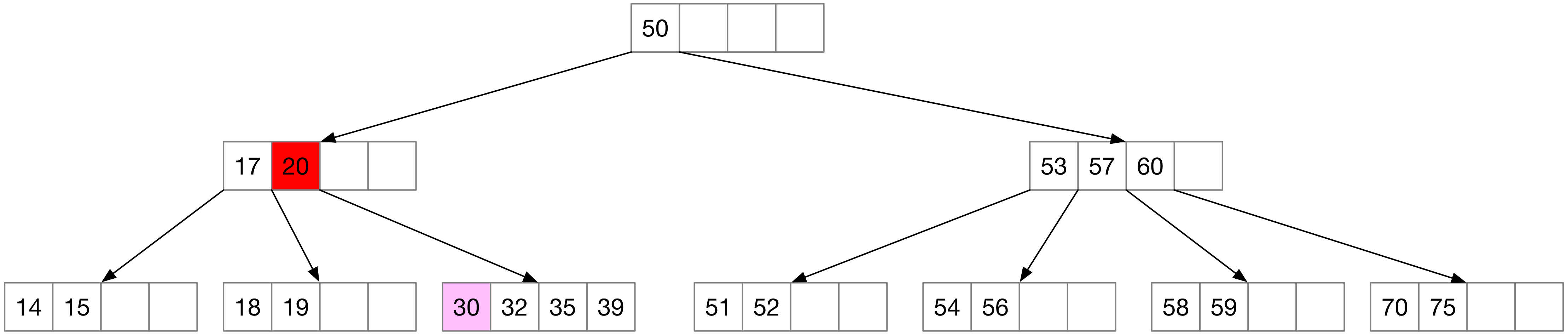
- we find it on a **leaf**
- we **remove** it, but **no shortage**, so we’re **finished**



## B-Trees

Let's **delete key 20**

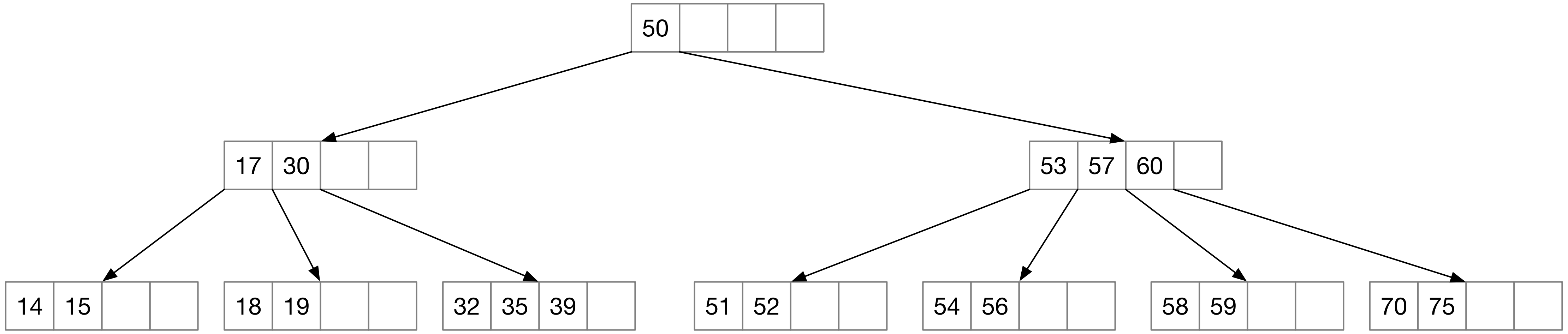
- we find it on an **internal** node
- we **find the next** key, which is 30
    - we **substitute** 20 with 30
    - we **delete** 30 from the **leaf**



# B-Trees

Let’s **delete key 20**

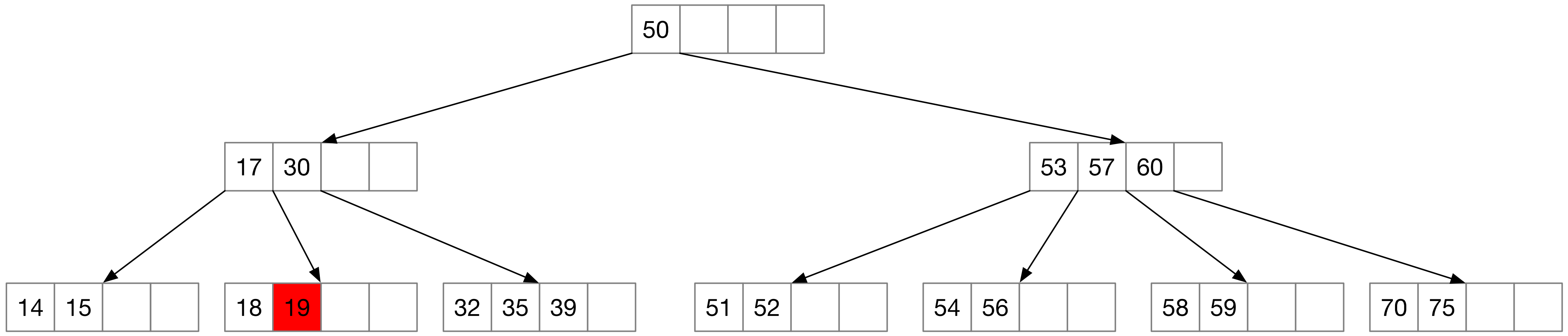
- Let’s **remove 30** from the leaf
    - we have **no shortage**, so we’re **finished**



# B-Trees

Let’s **delete key 19**

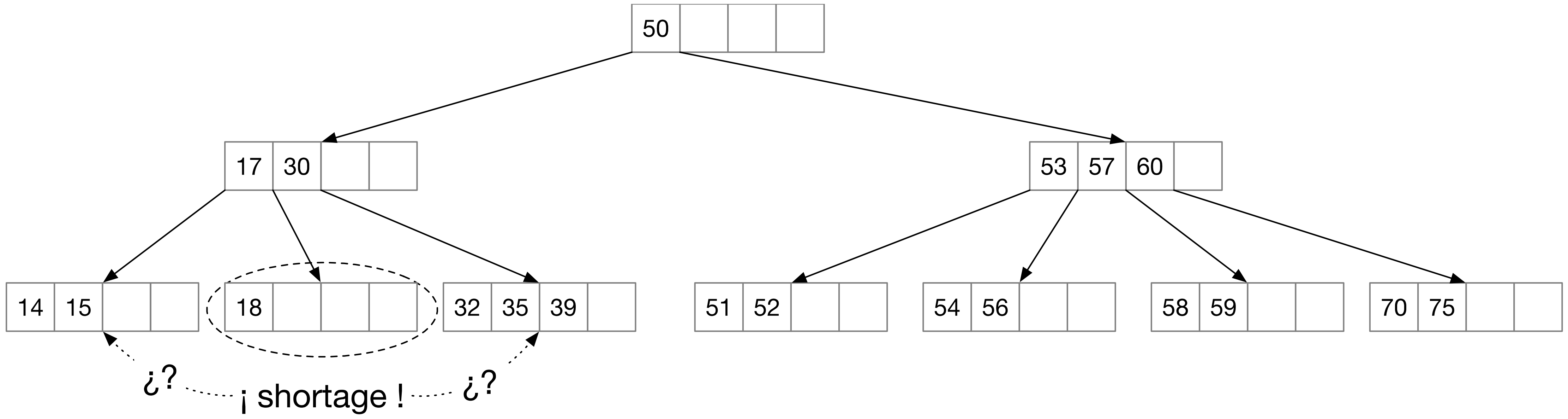
- we find it on a **leaf**, so we **remove** it



# B-Trees

Let’s **delete key 19**

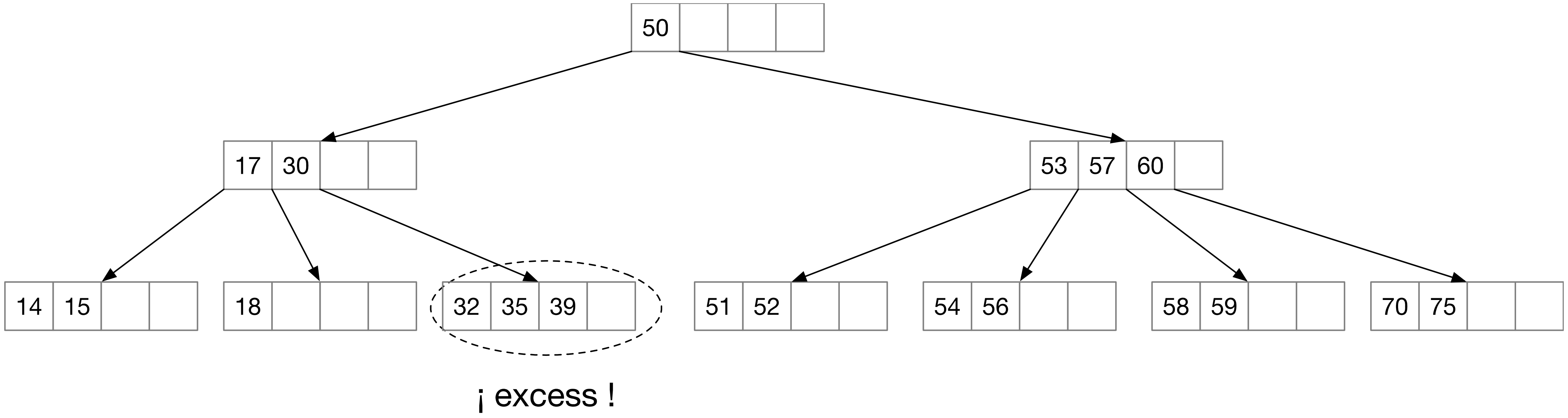
- we find it on a **leaf**, so we **remove** it
- but we incur in **shortage**
    - is there **any adjacent sibling with excess?**



# B-Trees

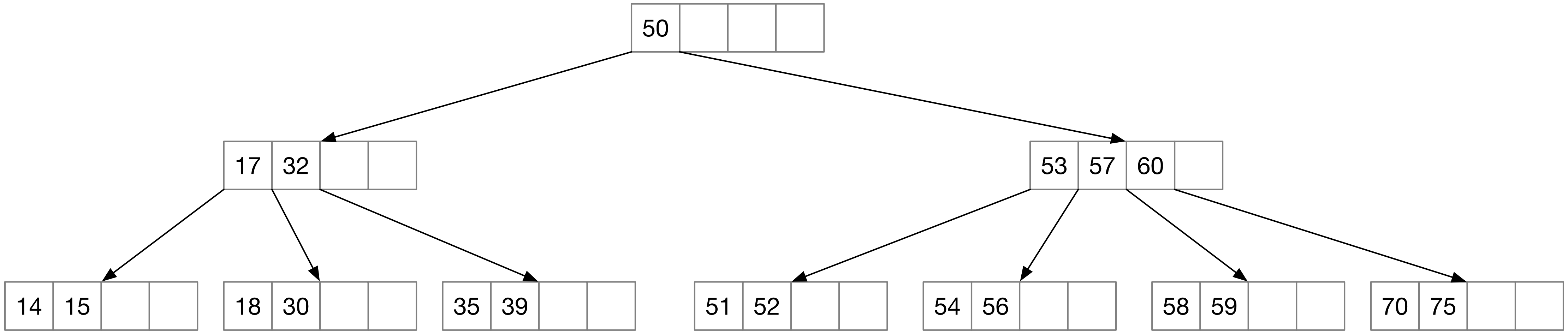
Let’s **delete key 19**

- we find it on a **leaf**, so we remove it
- but we incur in **shortage**
    - is there **any adjacent sibling with excess?**
    - **yes**



# B-Trees

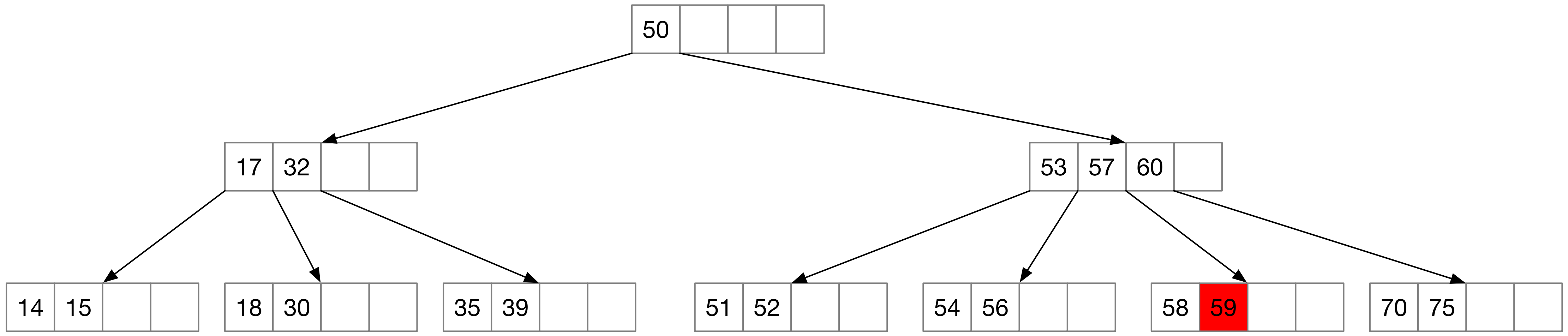
Let’s **delete key 19**
- we find it on a **leaf**, so we remove it
- but we incur in **shortage**
    - is there **any adjacent sibling with excess?**
    - **yes** => we **redistribute** the keys



# B-Trees

Let’s **delete key 59**

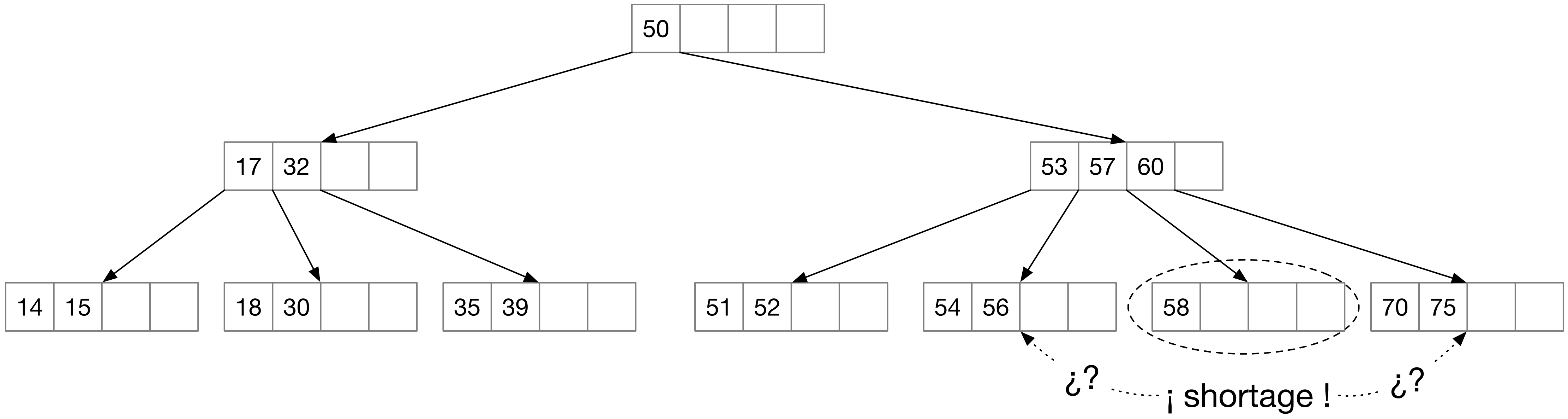
- we search for it, and we find it in a **leaf**



# B-Trees

Let's **delete key 59**

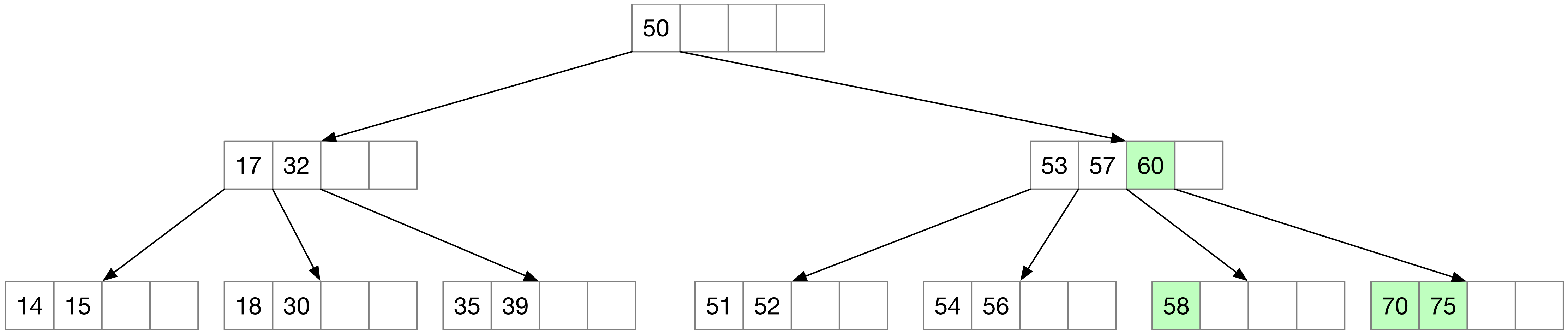
- we search for it, and we find it in a **leaf**
- we **remove** it but we incur in **shortage**
- is there **any adjacent sibling with excess?**
- **no**



# B-Trees

Let’s **delete key 59**

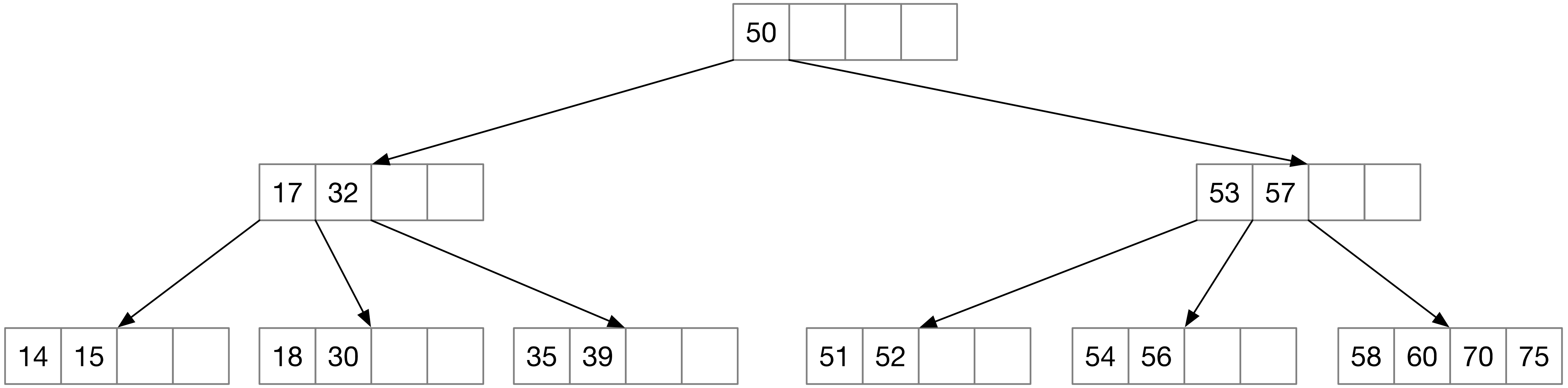
- we search for it, and we find it in a **leaf**
- we **remove** it but we incur in **shortage**
- is there **any adjacent sibling with excess?**
- **no** => we must do a **merge**



## B-Trees

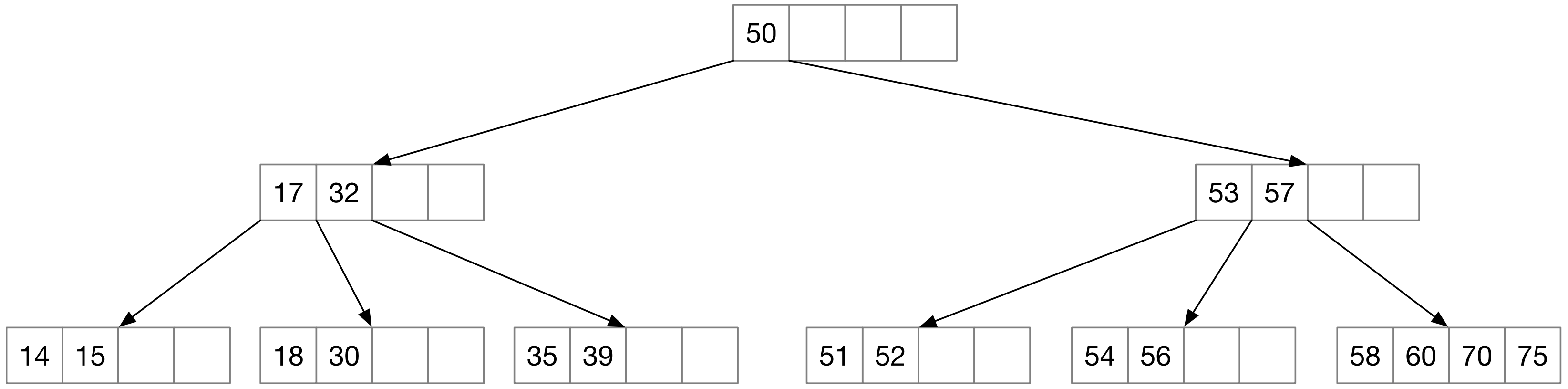
Let’s **delete key 59**

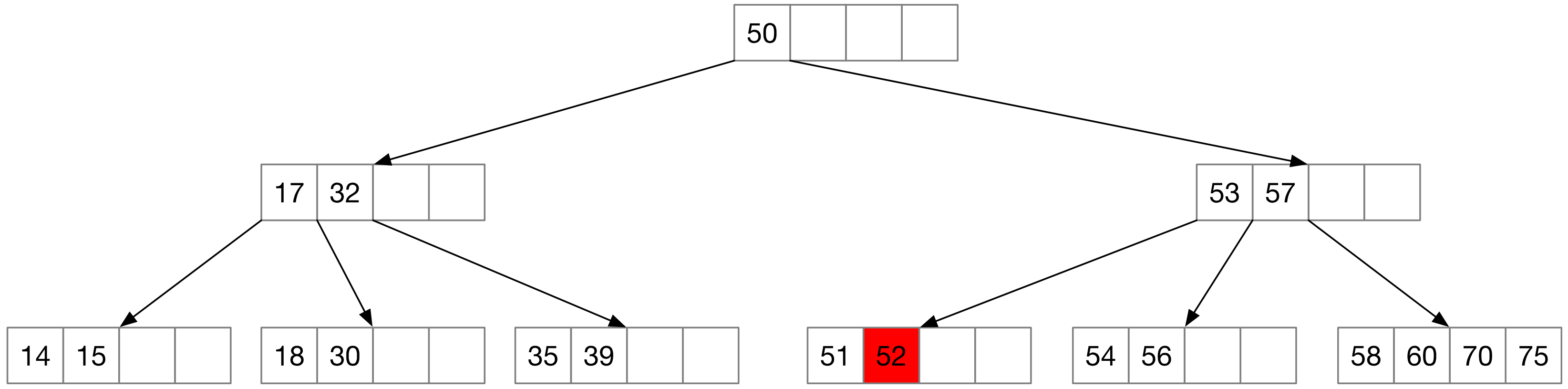
- we search for it, and we find it in a **leaf**
- we **remove** it but we incur in **shortage**
- is there **any adjacent sibling with excess?**
- **no** => we must do a **merge**
- the **parent loses a key** but has **no shortage**, we’re **finished**



# B-Trees

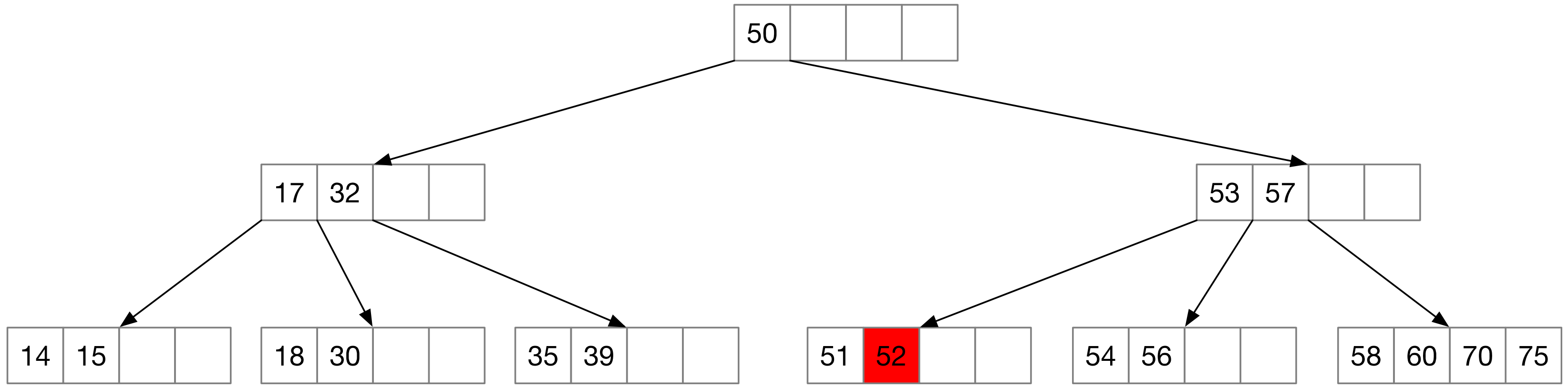
Let’s **delete key 52** (now without subtitles)

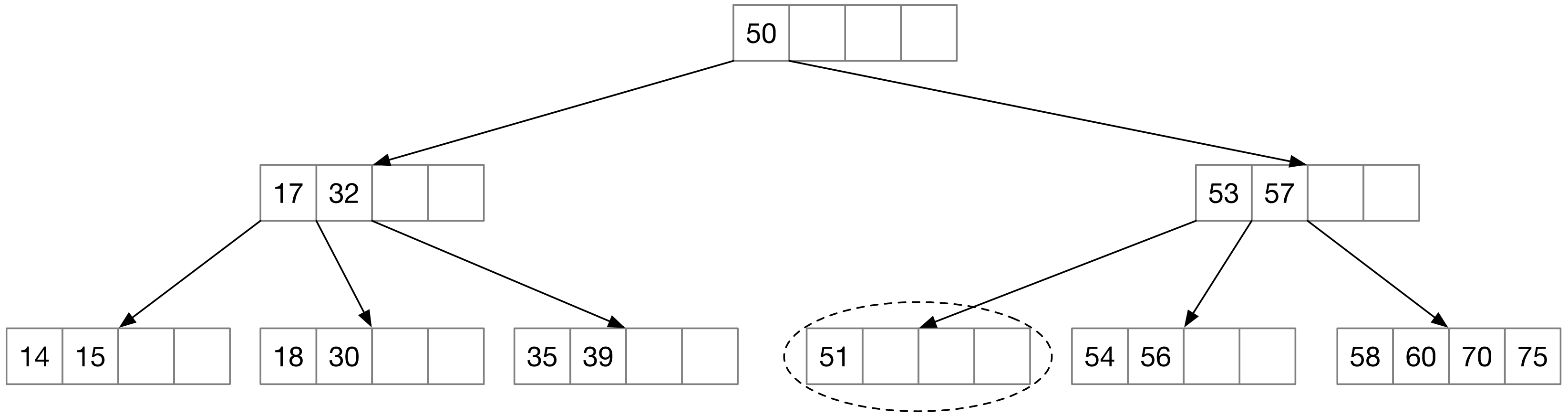




## B-Trees

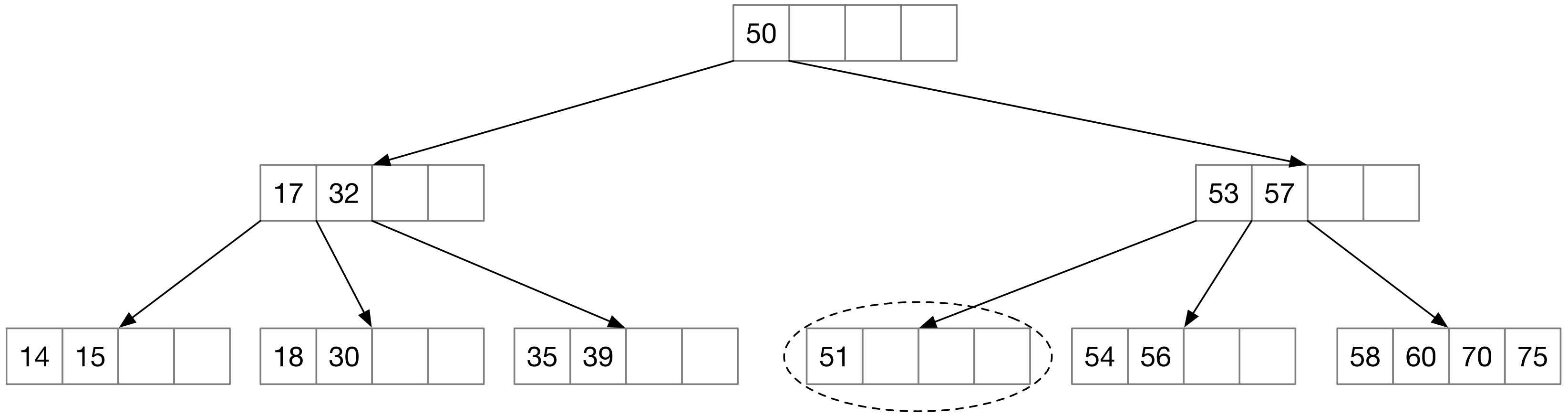
Let’s **delete key 52** (now without subtitles)

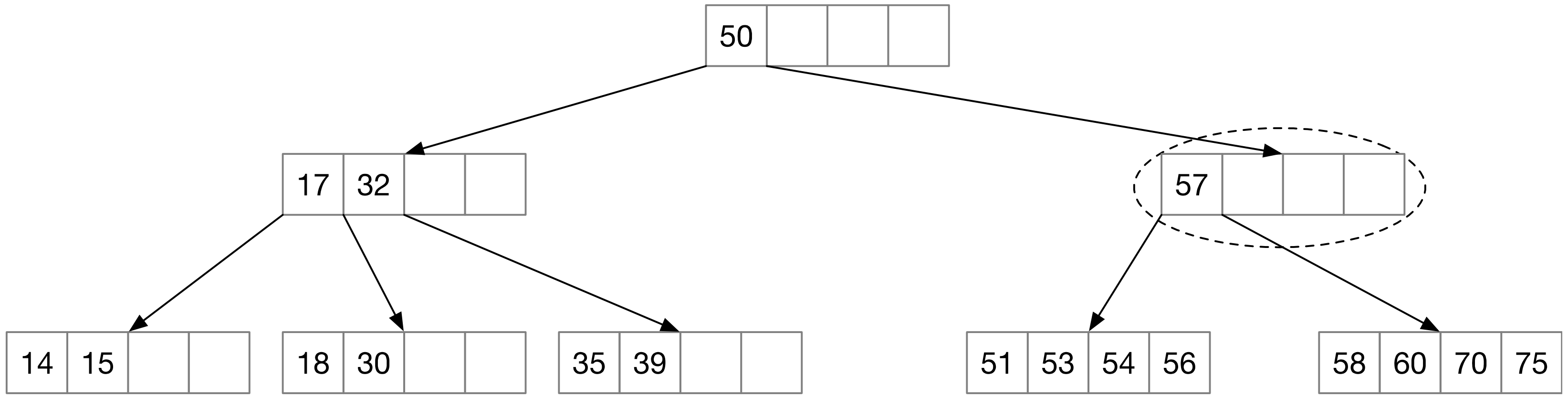




# B-Trees

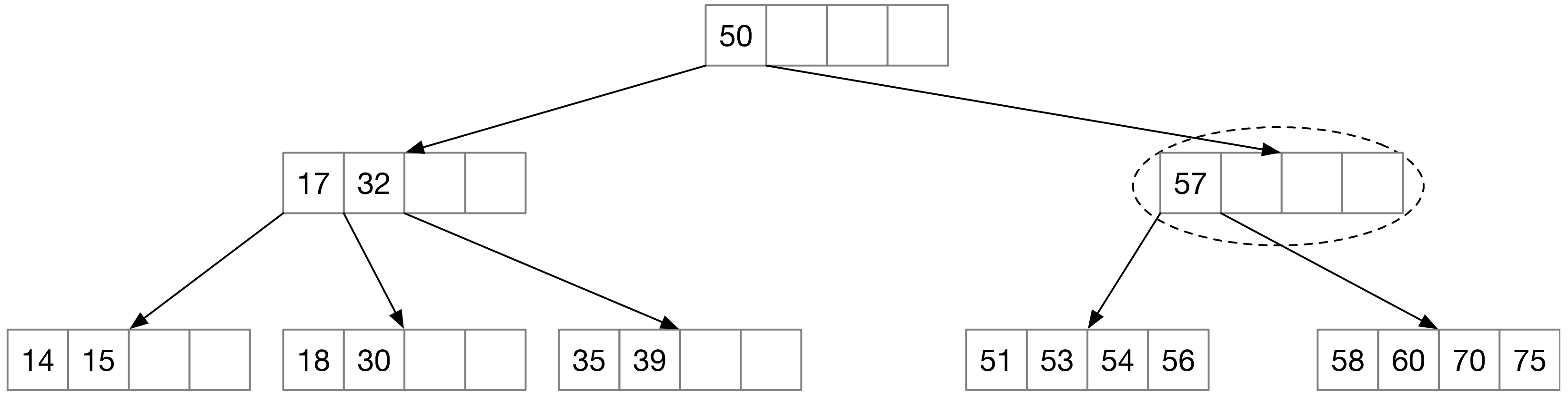
Let’s **delete key 52** (now without subtitles)

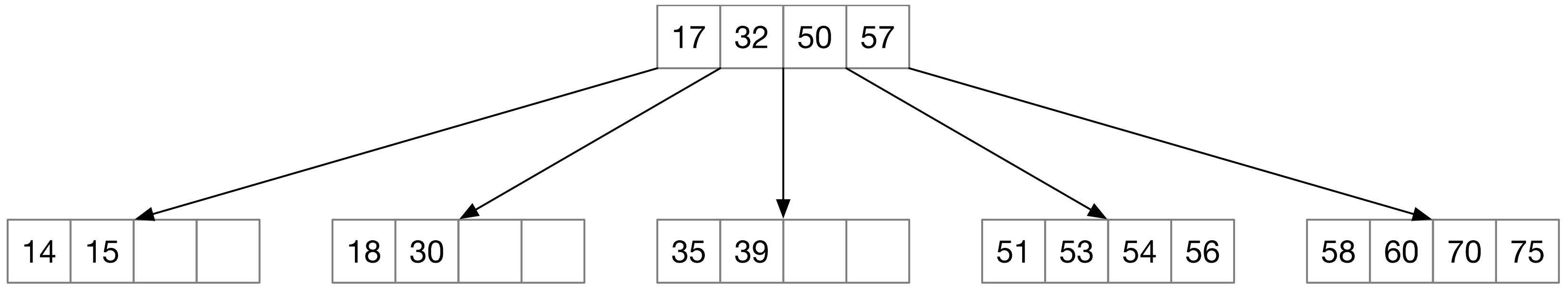




# B-Trees

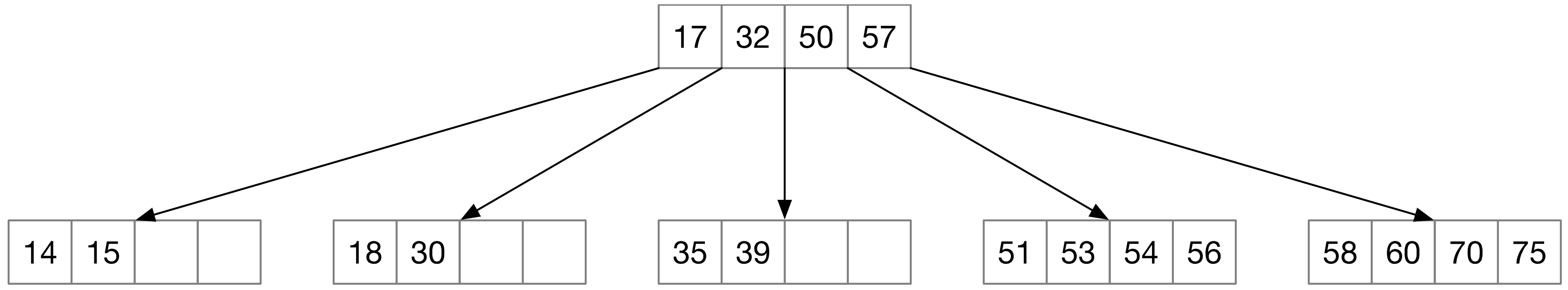
Let’s **delete key 52** (now without subtitles)

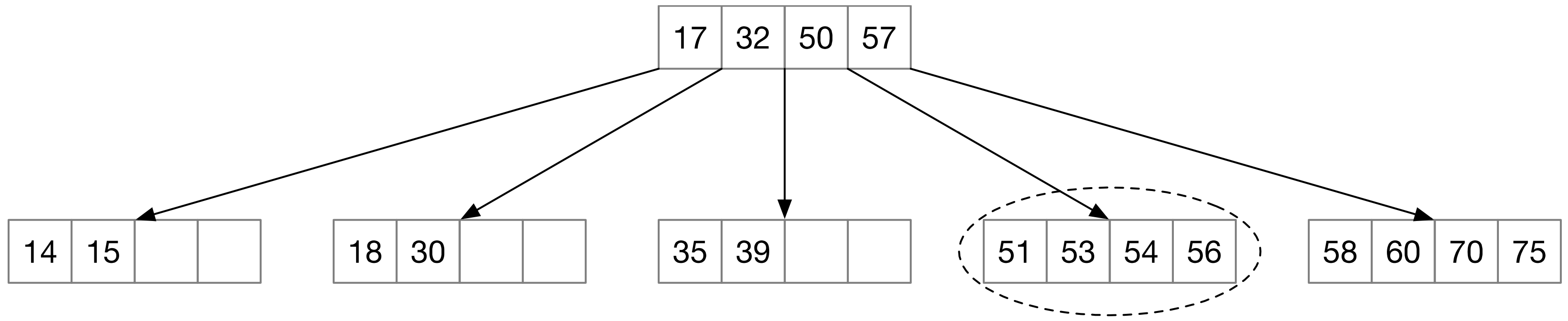




# B-Trees

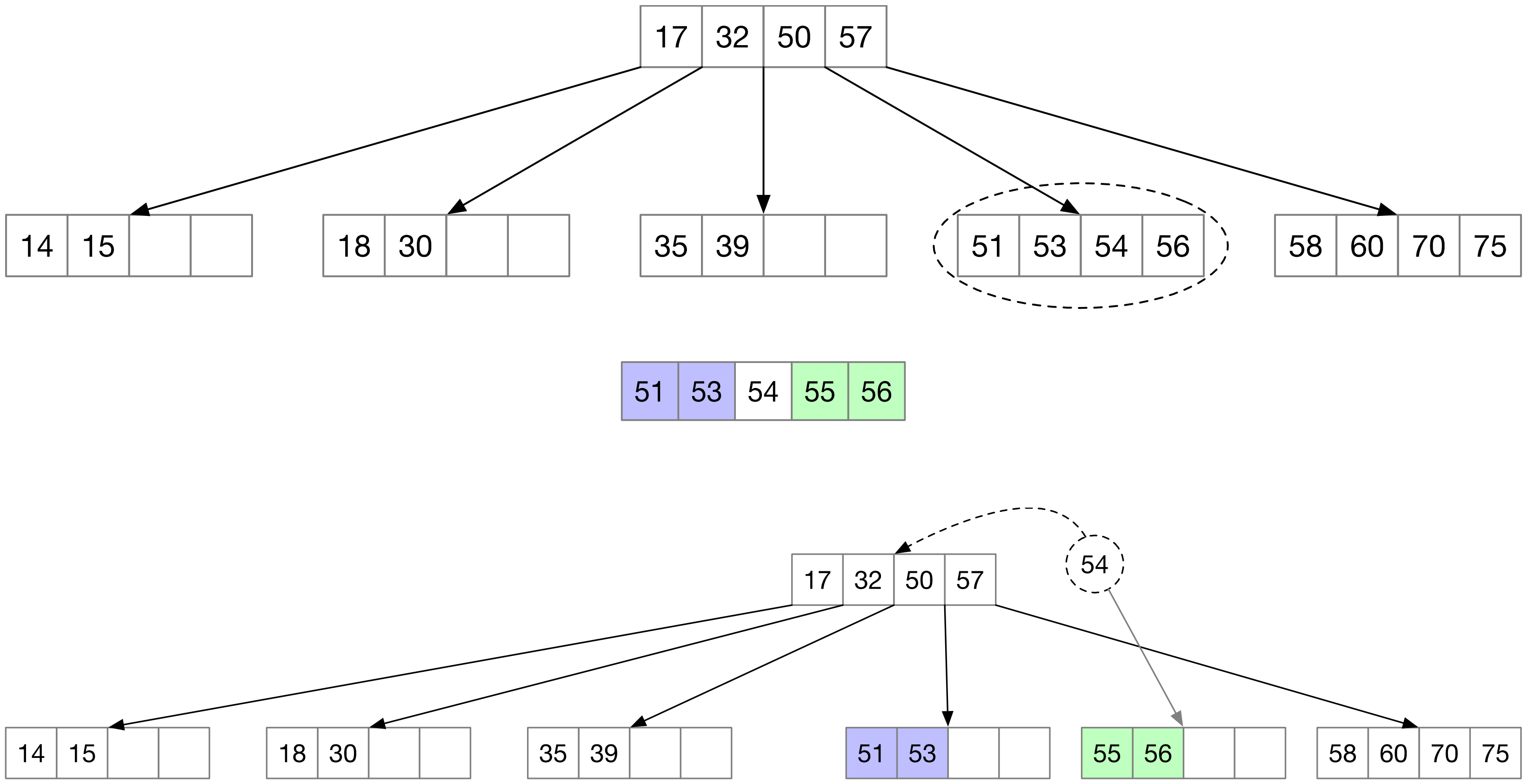
Let’s **insert key 55** (now without subtitles)





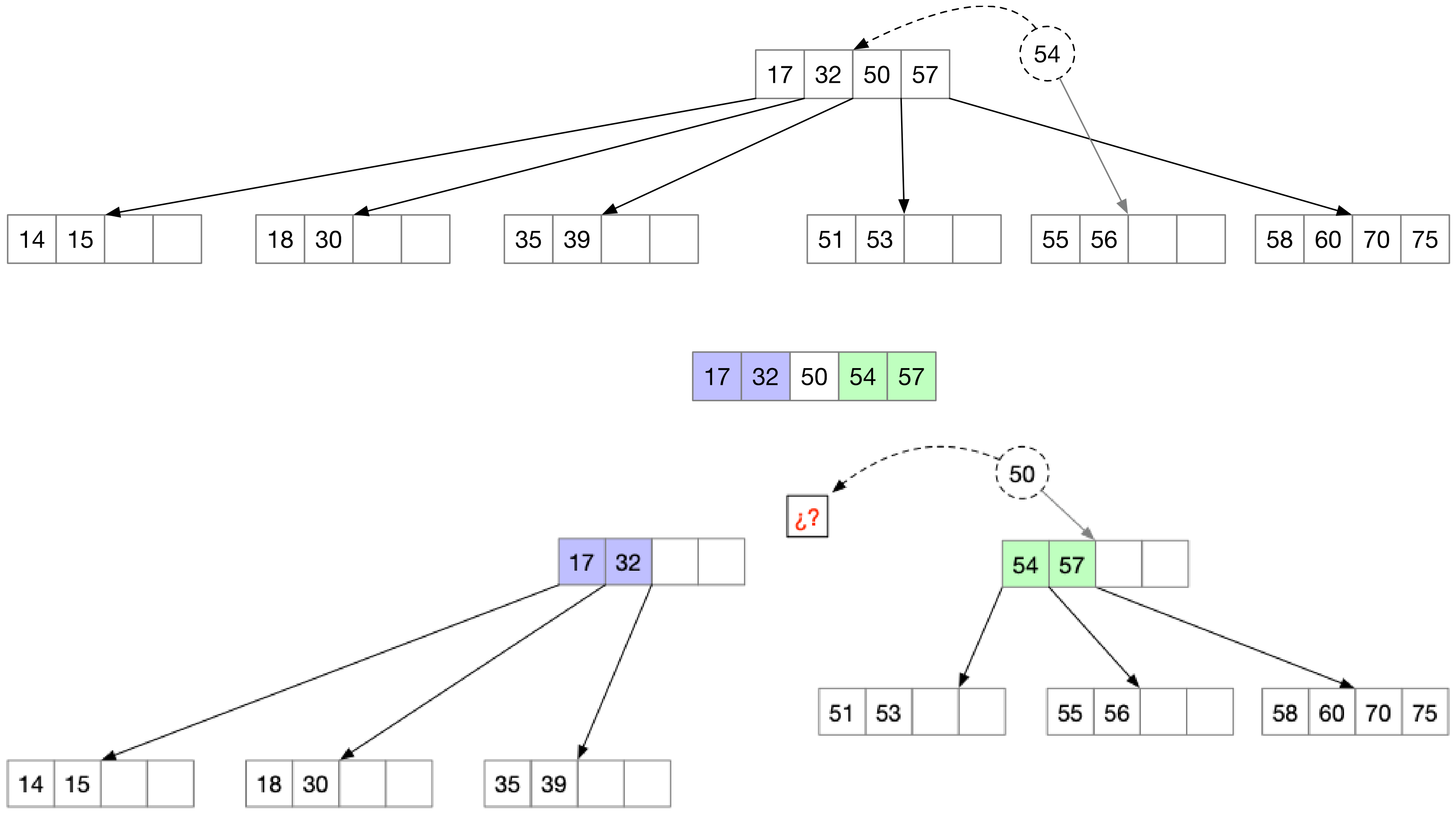
# B-Trees

Let’s **insert key 55** (now without subtitles)



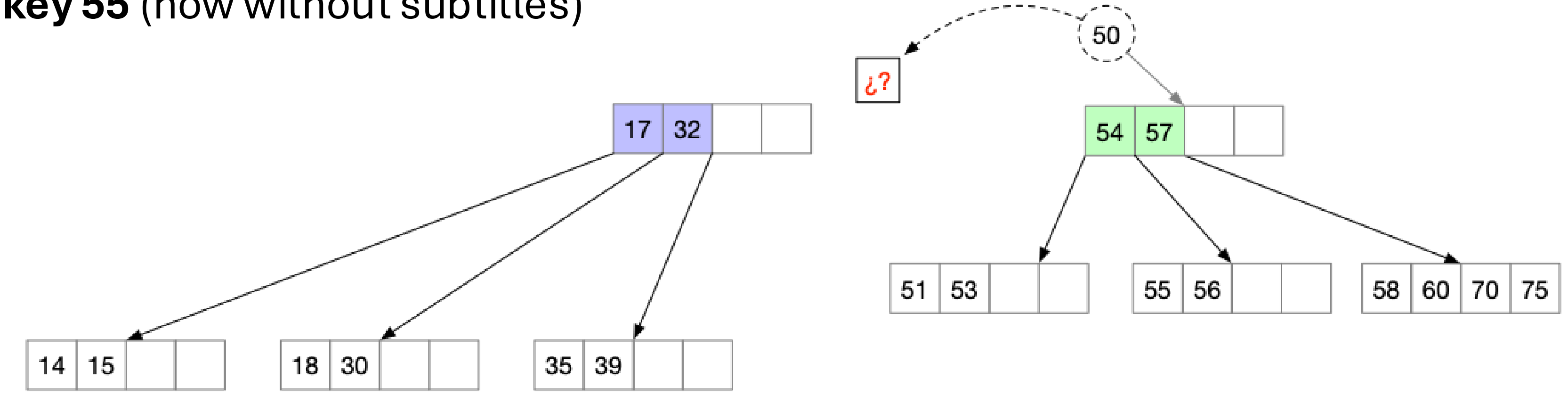
# B-Trees

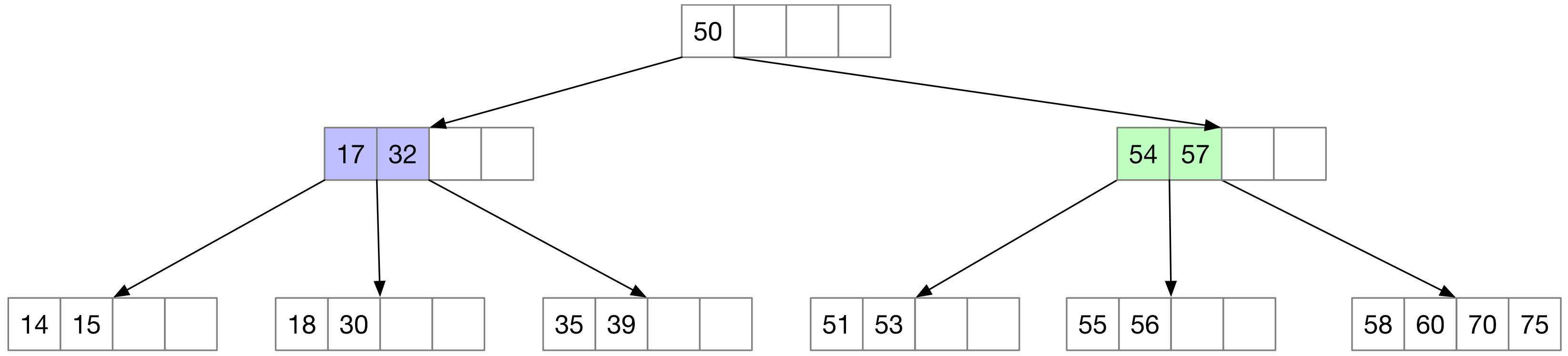
Let's **insert key 55** (now without subtitles)



# B-Trees

Let’s **insert key 55** (now without subtitles)


50
¿?
17 32
54 57
14 15
18 30
35 39
51 53
55 56
58 60 70 75


50
17 32
54 57
14 15
18 30
35 39
51 53
55 56
58 60 70 75