Hierarchical Data Structures

CE-1103 Algorithms and Data Structures

Disclaimer / Descargo de Responsabilidad

Esta presentación corresponde a una guía usada por el profesor durante las clases. La misma ha sido modificada para ser utilizado en el modelo de cursos asistidos por tecnología. No es una versión final, por lo que la misma podría requerir todavía hacer algunos ajustes. Para aspectos de evaluación esta presentación es solo una guía, por lo que el estudiante debe profundizar con el material de lectura asignado y lo discutido en clases para aspectos de evaluación.

This presentation corresponds to a guide material used by the professor during classes. It has been modified to be used in the model of technology-assisted courses. It is not a final version, so it may still require some adjustments. For evaluation aspects, this presentation is only a guide, so the student should delve with the assigned reading material and what has been discussed in class.

What is a tree?

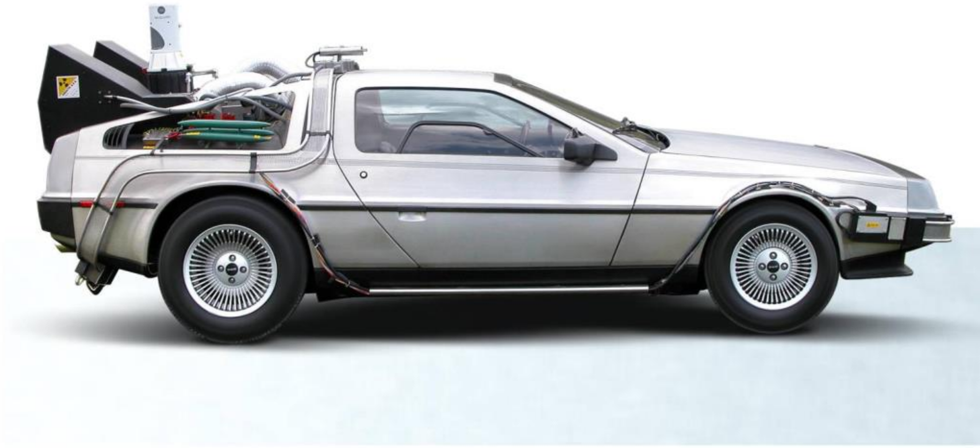
- A tree is one of the fundamental data storage structure used in programming

- It combines the advantages of an ordered array and a linked list
  - Fast searches
  - Fast insertion and deletion

Going back...

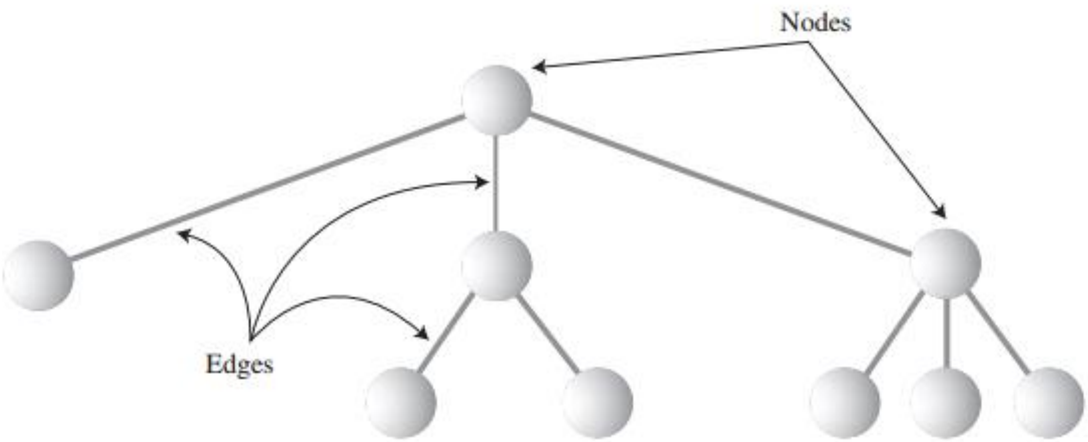
- About arrays
  - Why searching is fast?
  - Why insertion/deleting is expensive and slow?

- About linked-list
  - Why searching is slow?
  - Why insertion/deletion is fast?



What is a tree!?

- A tree consists of **nodes** connected by **edges**. Nodes are represented as circles and the edges as lines connecting the circles



What is a tree!?

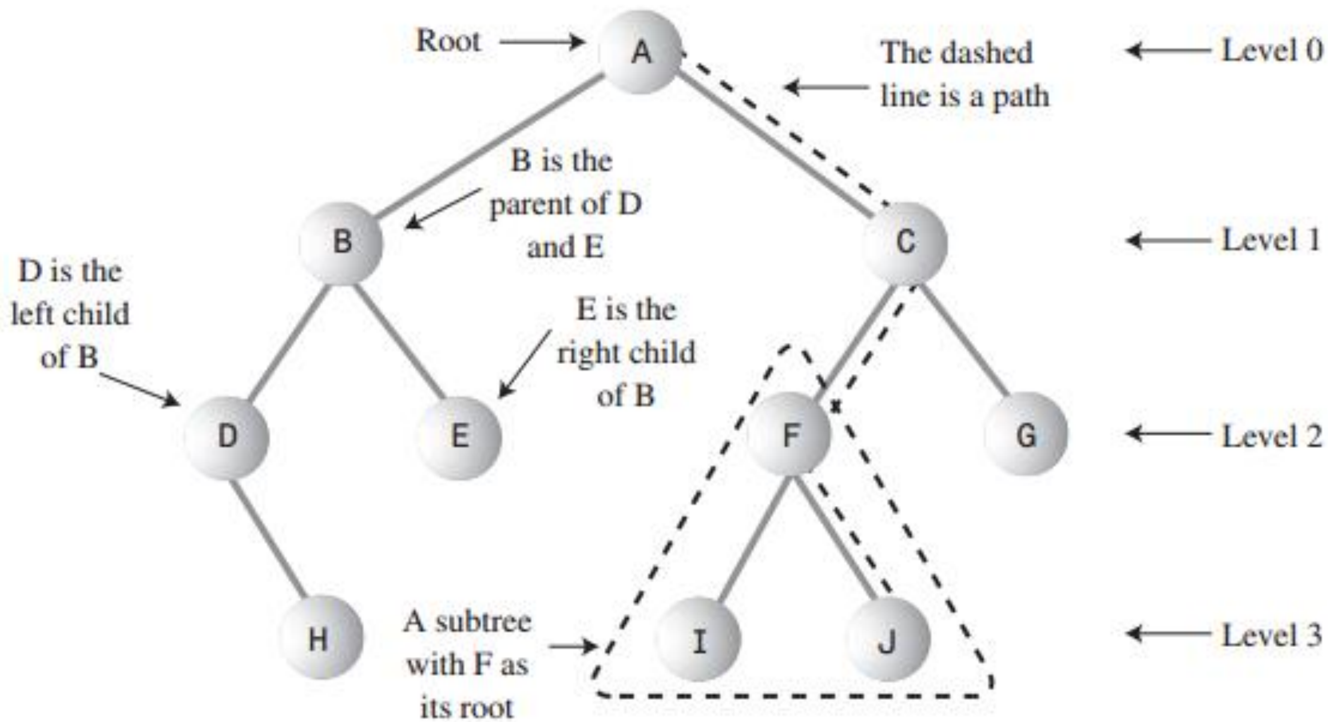
- Just like in linked-lists, nodes are composed of data and references (edges) to other nodes
- Given that trees are mathematical entities, there is many theoretical knowledge about trees
- A tree is specialization of a graph

What is a tree!?

- Trees are small on top and large on the bottom. Like inverted real life trees

- There are many types of trees:
  - Binary trees
  - Heap trees
  - AVL
  - Splay
  - B, B+, B*
  - Expression trees
  - N-ary trees

Tree
Terminology



Trees
Applications

- Implement file systems
- Organizate data that needs to be searched
- In databases, data is stored as B trees or any of its variations

Trees
Applications

- Compression algorithms

- Compilers use trees to represent syntactic expressions

Tree terminology

- "Walking" from node to node results in a sequence of nodes, this is called **path**
- The node at the top of the tree is called the root. **There is only one root**
- Any node (except the root) has exactly one **edge** running upward to another node. The node above is called the **parent** of the node

Tree terminology

- Any node may have one or more lines running downward to other nodes. These nodes below are called **children**
- A node that has no children is called a **leaf**
- Any node can be considered to be the root of a **subtree**. A subtree contains all descendents of a node

Tree terminology

- Nodes with the same parent are called **siblings**
- The **length** of a path is the number of edges on the path. There is a path of length zero from every node to itself
- The **depth** of a node is the length of the unique path from the root to the node.

Tree terminology

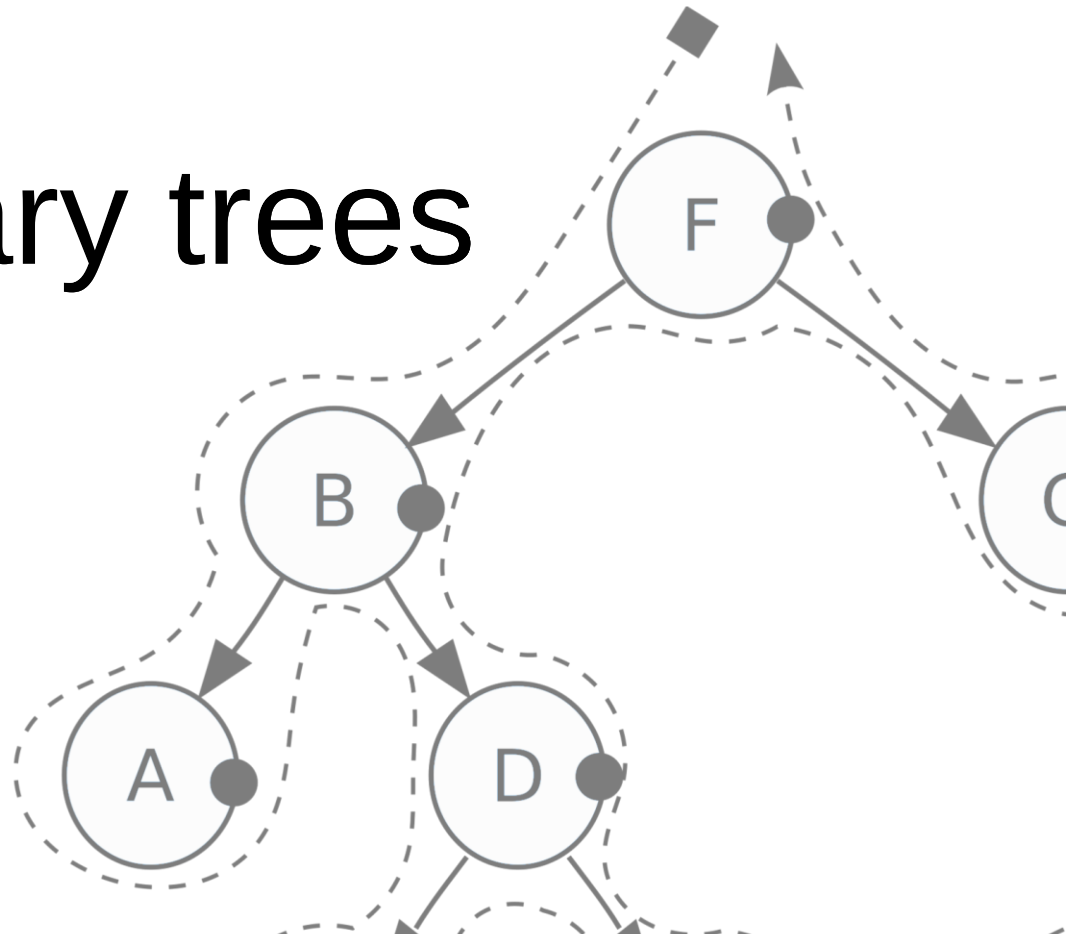
- Root is at a **depth** of zero
- The **height** of a tree is the length of the longest path from the node to a leaf

Tree terminology

- A node is **visited** when program control arrives at the node, usually for the purpose of carrying out some operation on the node. If nothing is done on the node, then is not visiting.

- **Traverse** a tree is visiting all the nodes in some specific order

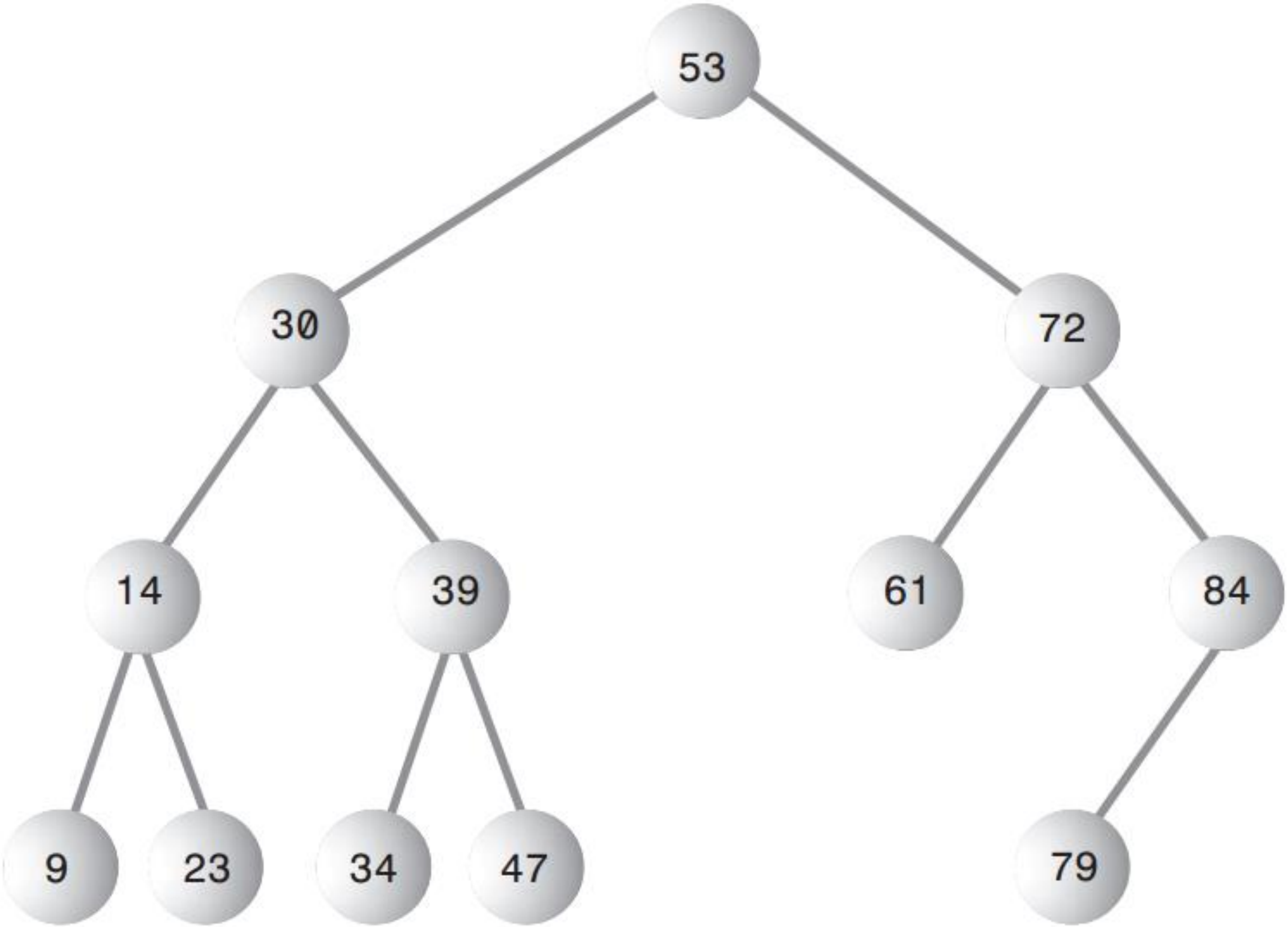
Binary trees



What is a binary tree?

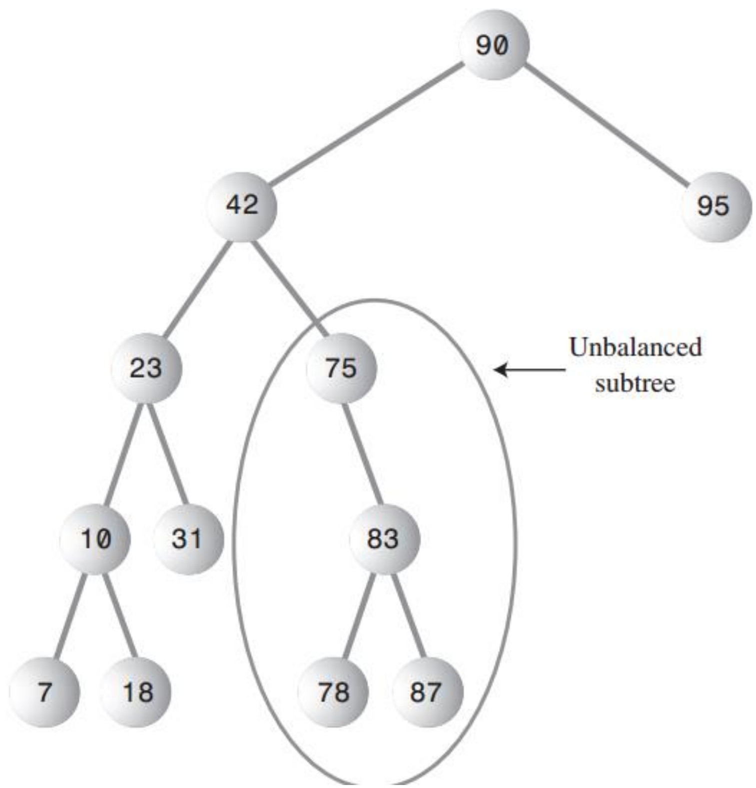
- Is a tree where each node can have **at most** two children
- The two children are called the **left child** and the **right child**
- If the left child node key is smaller than the key of the parent and the right child key is greater, the tree is **called binary search tree (BST)**

What is a binary tree?



What is a binary tree?

- Depending on the insertion order, a tree can become unbalanced.



Binary search trees

Implementation

- Trees can be implemented using arrays or using dynamic memory like a linked-list

- The most common way of implementation is using dynamic memory. We will focus on this approach

Binary search trees

Implementation

```
public class Node<T extends Comparable<T>> {
  T element
  Node<T> left;
  Node<T> right;

  public Node(T element) {
    this(element, null, null);
  }

  public Node(T element, Node<T> left, Node<T> right) {
    this.element = element;
    this.left = left;
    this.right = right;
  }
}
```

Binary search trees
Implementation

```
public class BinaryTree<T extends Comparable<T>> {
  private Node<T> root;

  public BinaryTree() {
    this.root = null;
  }

  public boolean isEmpty() {
    return this.root == null;
  }
}
```

Binary search trees
Contains operation

- Returns true if there is a node in the tree that has item *element* or false if there is no such node

- If the tree is empty it just returns false, otherwise make a recursive call on a subtree, either left or right

Binary search trees

Contains operation

```
public class BinaryTree<T extends Comparable<? super T>> {
  private Node<T> root;

  public BinaryTree() {
    this.root = null;
  }

  public boolean isEmpty() {
    return this.root == null;
  }

  public boolean contains(T element) {
    return this.contains(element, this.root);
  }
}
```

Binary search trees
Contains operation

```
public class BinaryTree<T extends Comparable<? super T>> {
  private boolean contains(T element, Node<T> node) {
    if (node == null) {
      return false;
    } else {
      int compareResult = element.compareTo(node.element);

      if (compareResult < 0)
        return contains(element, node.left);
      else if (compareResult > 0)
        return contains(element, node.right);
      else
        return true;
    }
  }
}
```

Binary search trees
min/max operation

```
public class BinaryTree<T extends Comparable<? super T>> {
  public Node<T> findMin() {
    if (this.isEmpty) {
      return null;
    } else {
      return this.findMin(this.root).element;
    }
  }

  public Node<T> findMax() {
    if (this.isEmpty) {
      return null;
    } else {
      return this.findMax(this.root).element;
    }
  }
}
```

Binary search trees

min/max operation

```
public class BinaryTree<T extends Comparable<? super T>> {
  private Node<T> findMin(Node<T> node) {
    if (node == null)
      return null;
    else if (node.left == null)
      return node;
    else
      return findMin(node.left);
  }
  private Node<T> findMax(Node<T> node) {
    if (node!= null)
      while (node.right != null) {
        node = node.right;
      }
    return node;
  }
}
```

Binary search trees

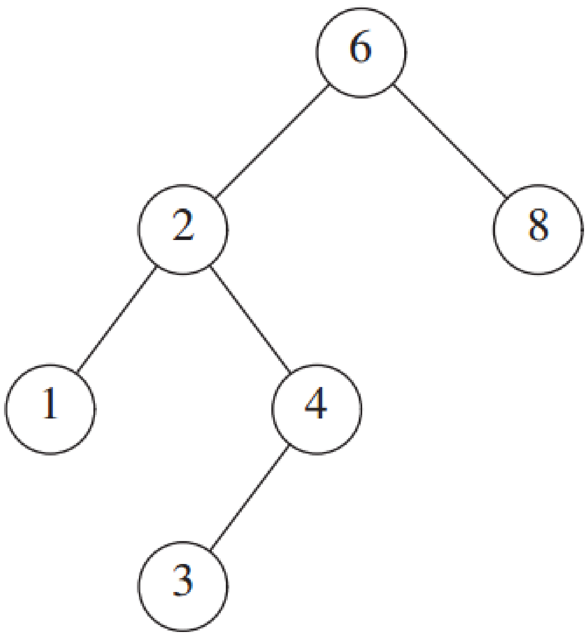
Insert operation

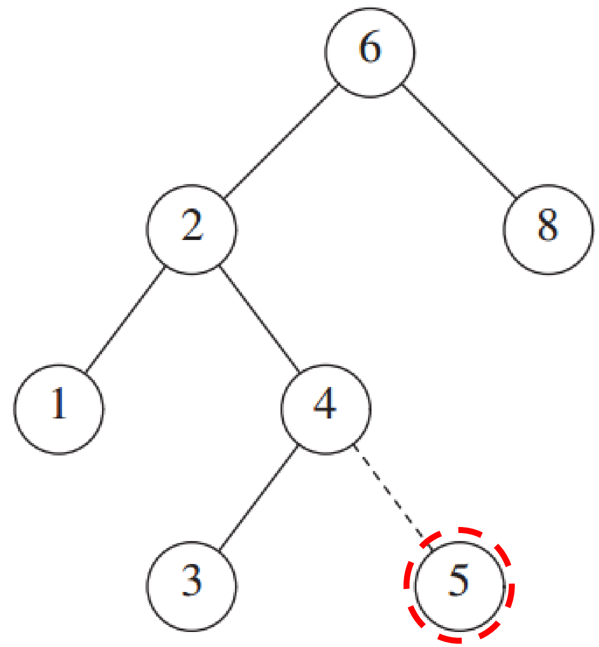
- To insert element in the tree T, proceed down the tree as in the contains operation

- If element is found, do nothing or update something in the node. Otherwise insert element at the last spot on the path traversed

Binary search trees

Insert operation





Binary search trees

Insert operation

```
public class BinaryTree<T extends Comparable<? super T>> {
  public void insert(T element) {
    this.root = this.insert(element, this.root);
  }
  private Node<T> insert(T element, Node<T> current) {
    if (current == null)
      return new Node<T>(element, null, null);

    int compareResult = element.compareTo(node.element);

    if (compareResult < 0)
      current.left = this.insert(element, node.left);
    else if (compareResult > 0)
      current.right = this.insert(element, node.right);

    return current;
  }
}
```

Binary search trees
Delete operation

- Delete is the hardest operation

- Once we have found the node we want to delete, we have to consider several possibilities
  - When node is a leaf
  - When node has one child
  - When node has two child

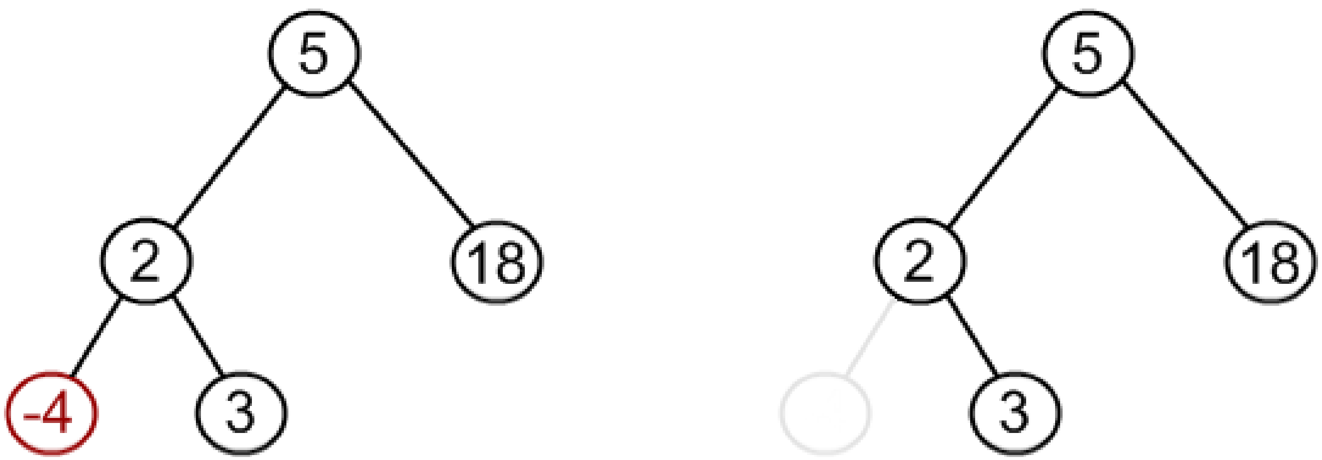
Binary search trees
Delete operation

- When the **node is a leaf**, is as simple as deleting that node.

Binary search trees

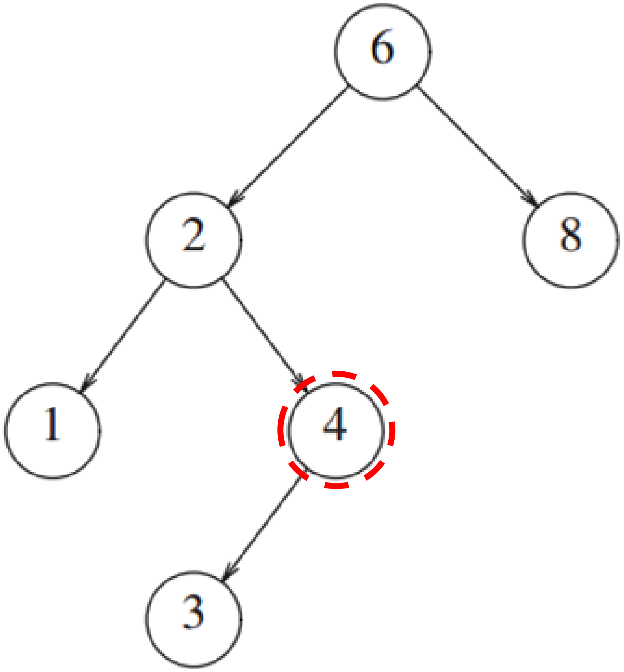
Delete operation

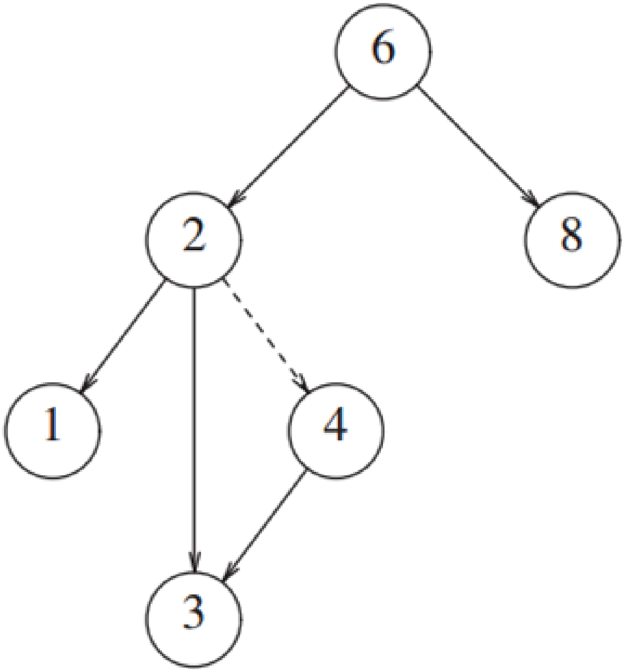
- Remove -4



Binary search trees
Delete operation

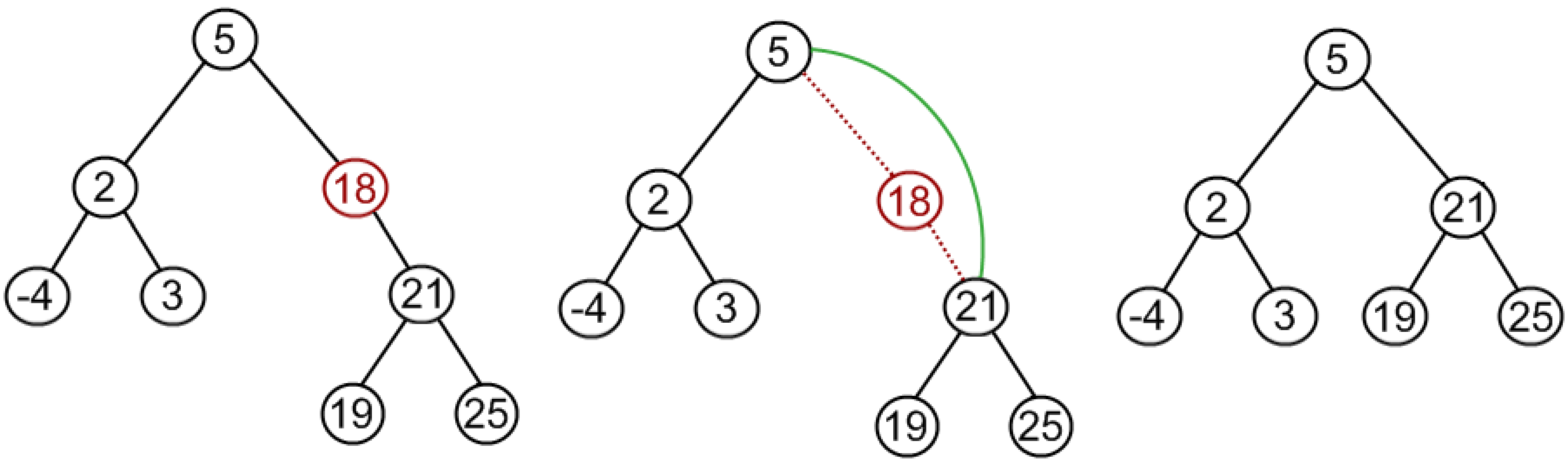
- When the **node has one child,** the parent has to adjust its link to bypass the node





Binary search trees
Delete operation

- Remove 18



Binary search trees

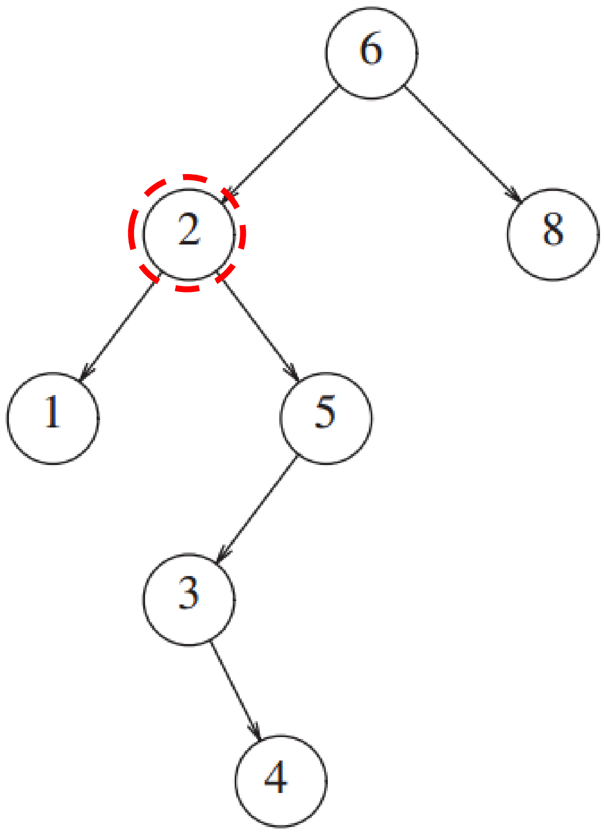
Delete operation

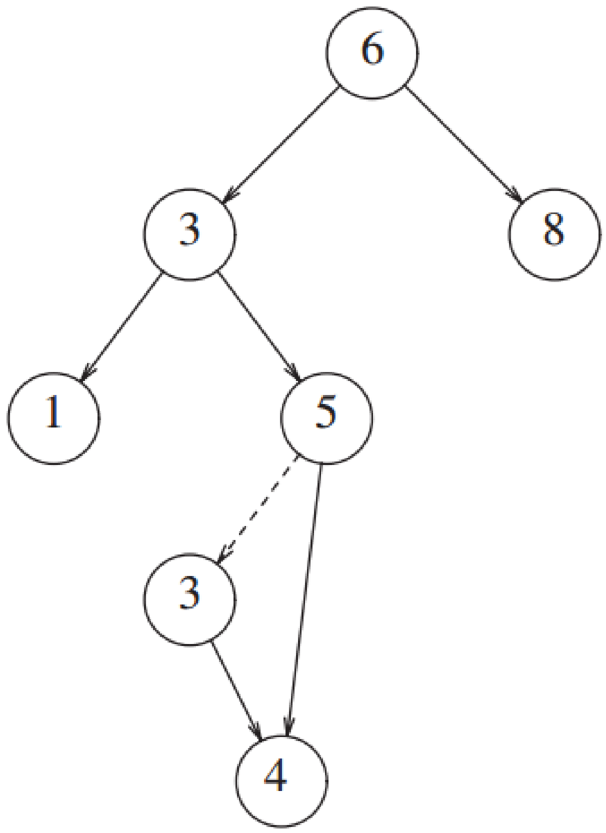
- When the **node has two children,** the strategy is to replace the data of the current node with the smallest data of the right subtree and recursively delete that node (the one that replaced)

- Or replace the data of the current node with the biggest data of the left subtree and recursively delete that node (the one that replaced)

Binary search trees

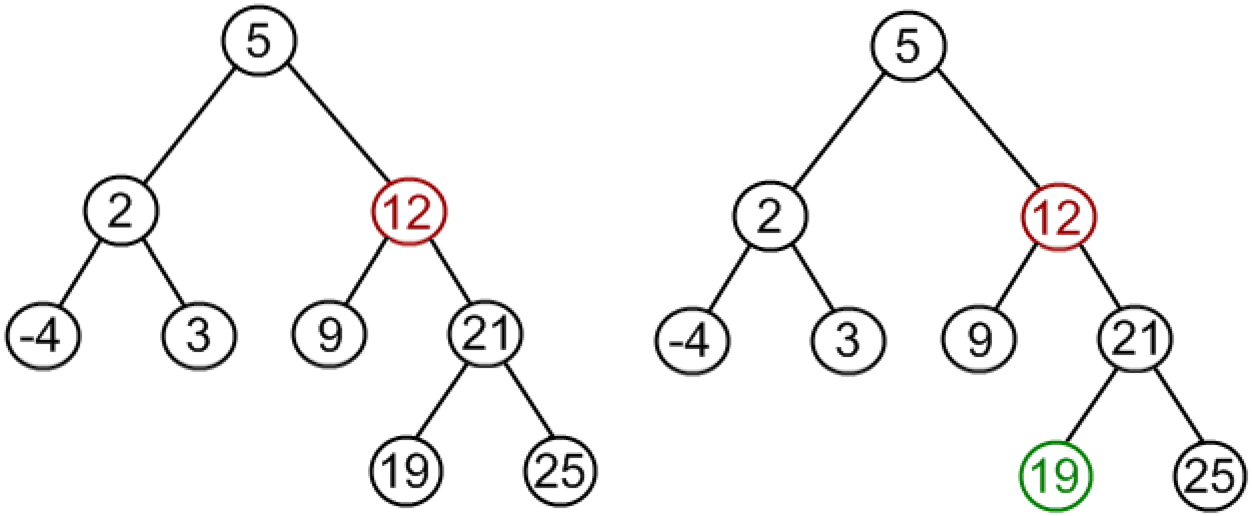
Delete operation





Binary search trees
Delete operation

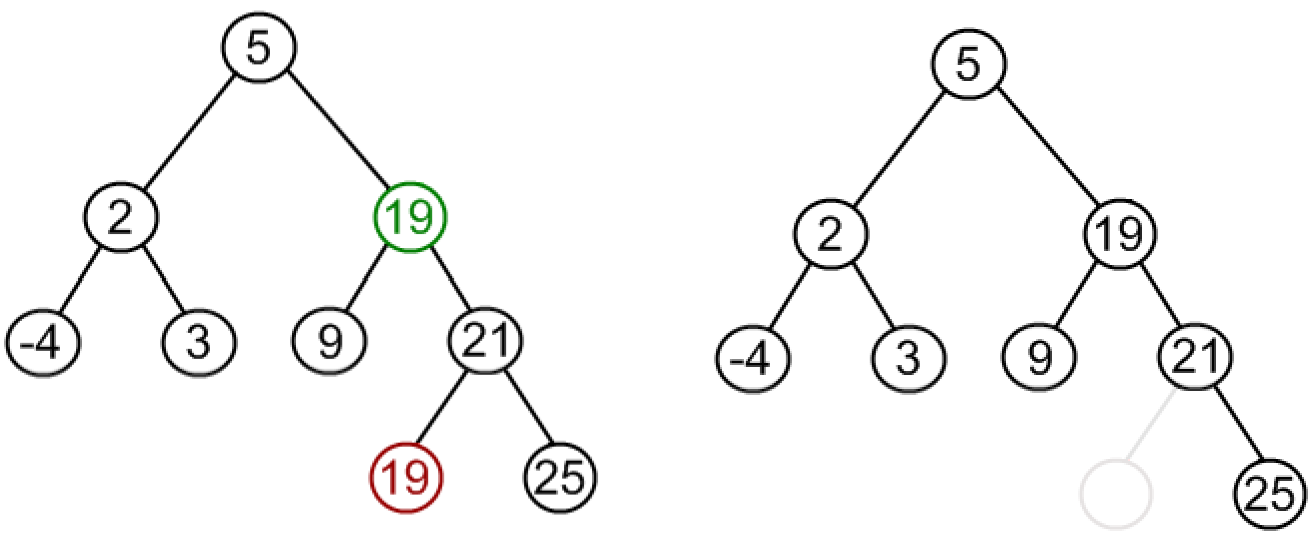
- Remove 12



Binary search trees

Delete operation

- Remove 12



Binary search trees

Delete operation

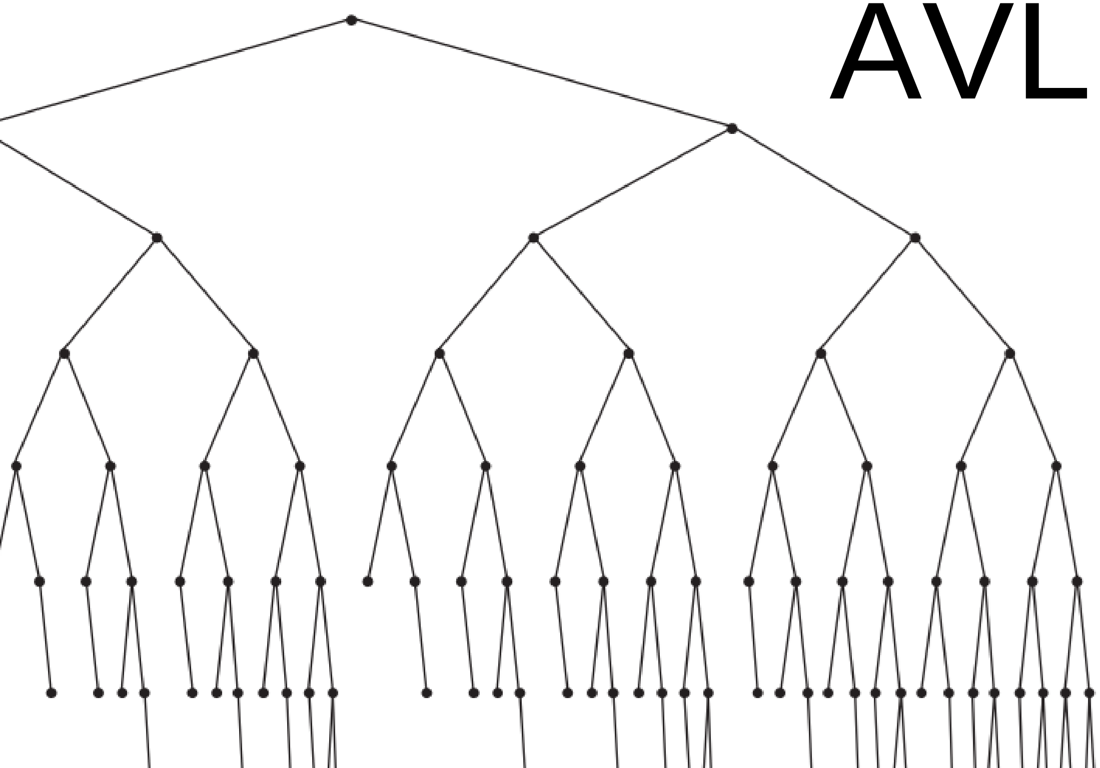
```
public class BinaryTree<T extends Comparable<? super T>> {
  public void remove(T element) {
    this.root = this.remove(element, this.root);
  }
}
```

Binary search trees

Delete operation

```
public class BinaryTree<T extends Comparable<? super T>> {
  public void remove(T element, Node<T> node) {
    if (node == null)
      return node;

    int compareResult = element.compareTo(node.element);

    if (compareResult < 0)
      node.left= remove(element, node.left);
    else if (compareResult > 0)
      node.right = remove(element, node.right);
    else if (node.left != null && node.right != null){
      node.element = findMin(node.right).element;
      node.right = remove(node.element, node.right)
    } else {
       node = node.left != null ? node.left : node.right;
    }
    return node;
  }
}
```

AVL



What is an AVL tree?

- AVL means Adelson-Velskii and Landis

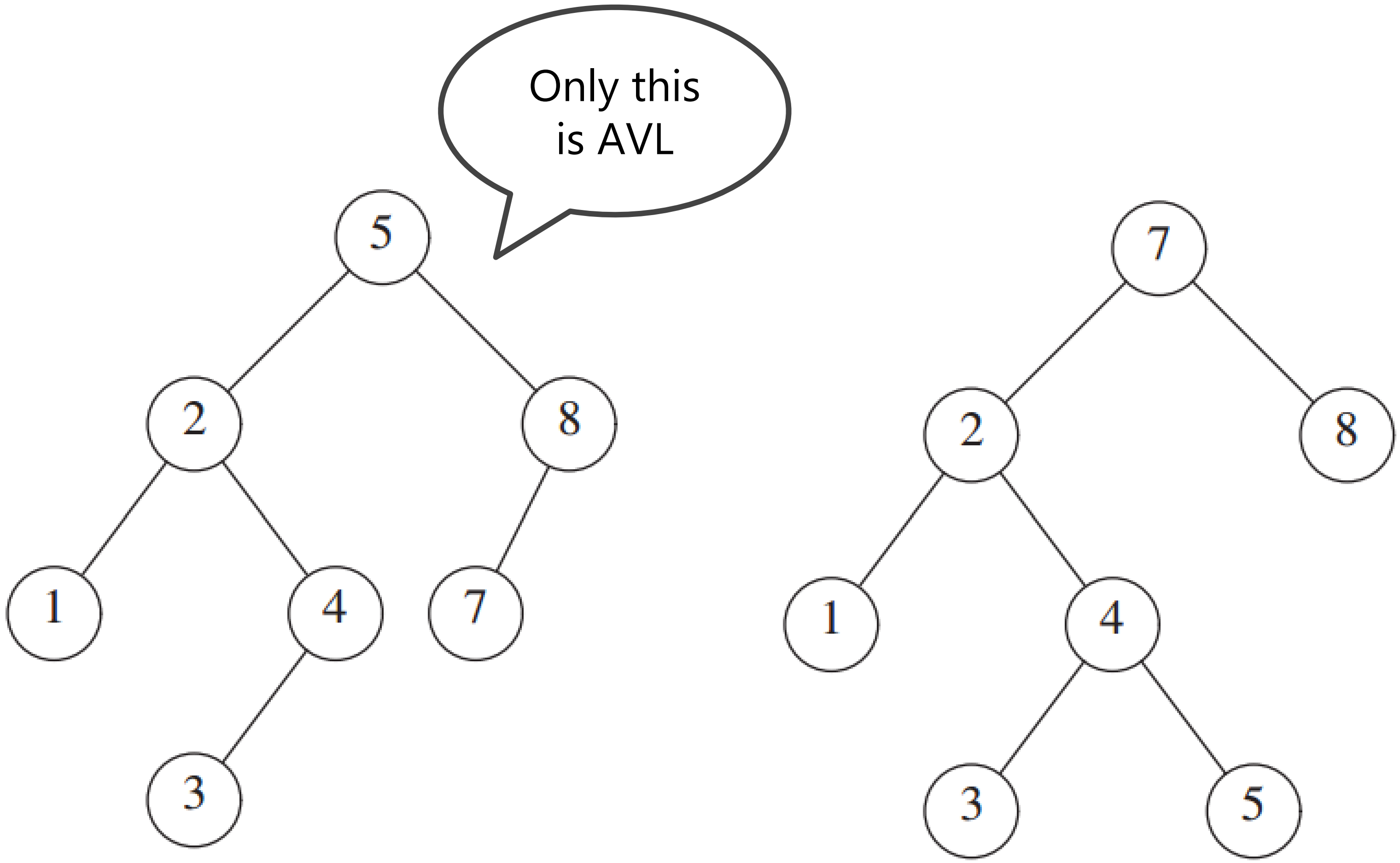
- Is a binary search tree with a balance condition. This condition is easy to maintain ensuring that the depth of the tree is O(logN)
  - Height of the left and right side can only differ by one

What is an AVL tree?

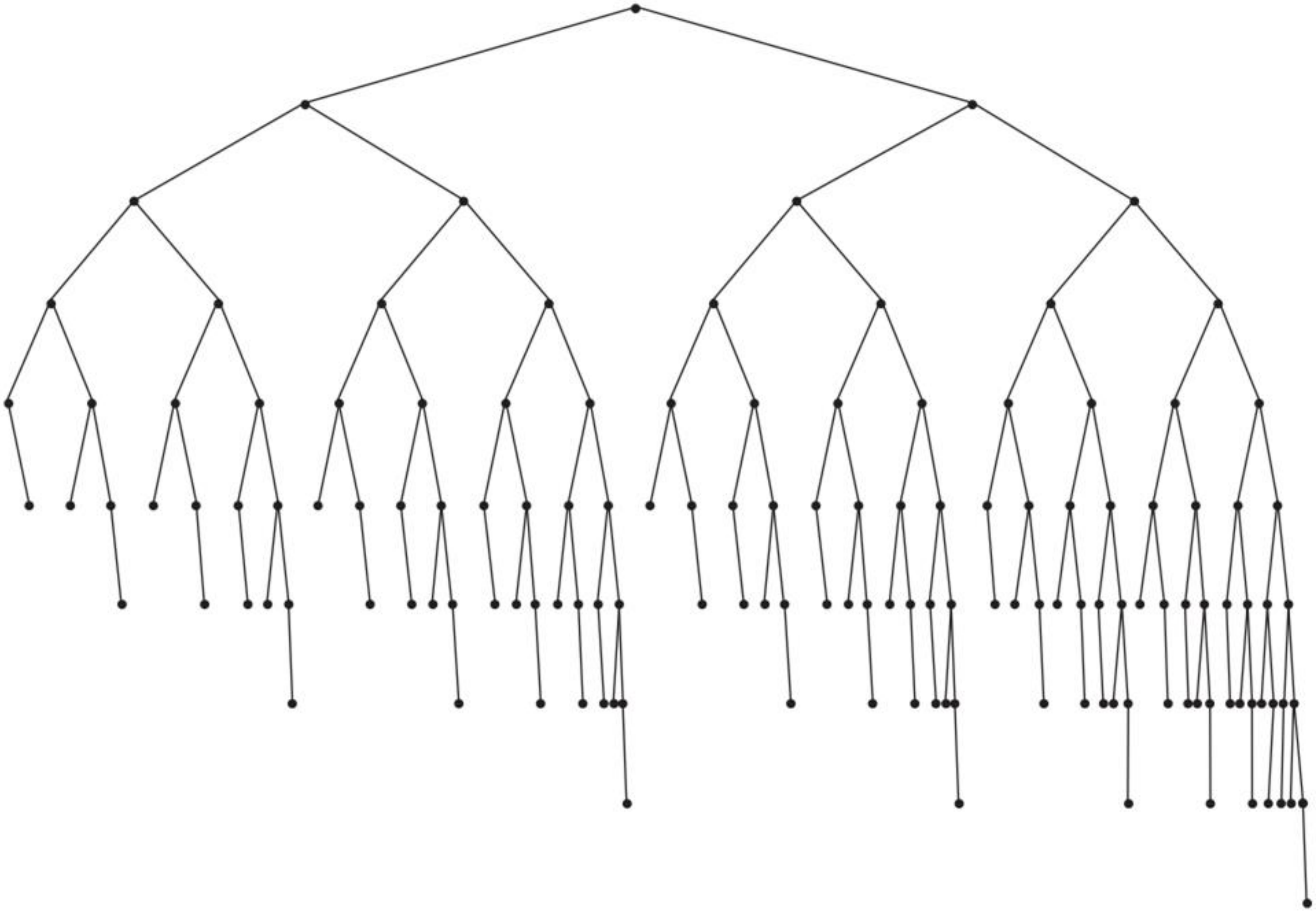
- In the AVL context, the height of a tree is the maximum level of a tree plus one

- A null tree has a height of zero

What is an AVL tree?

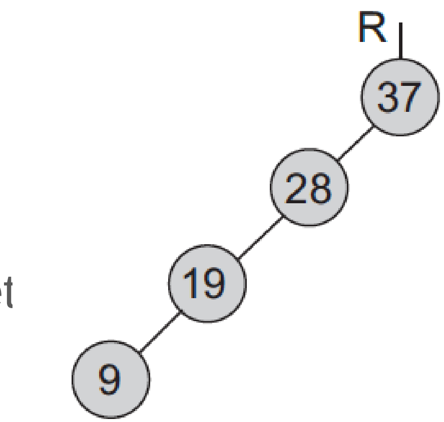


What is an AVL tree?



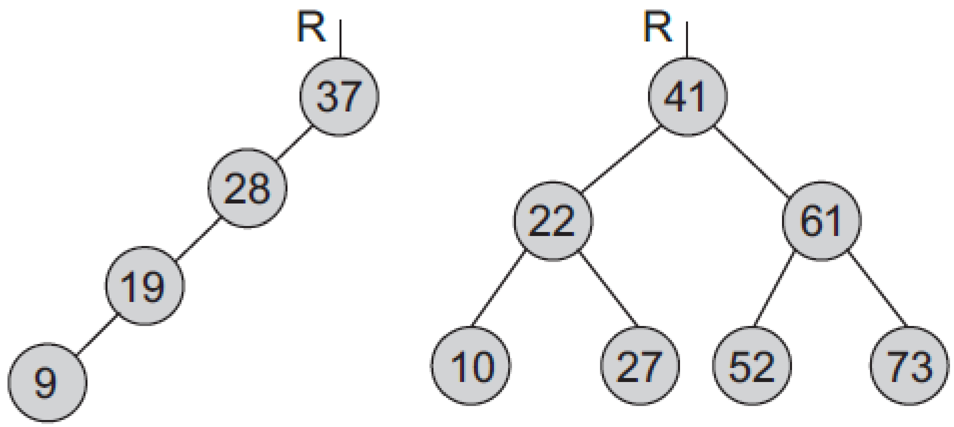
A tree can get unbalanced

- In an specific sequence a binary search tree can become like this:

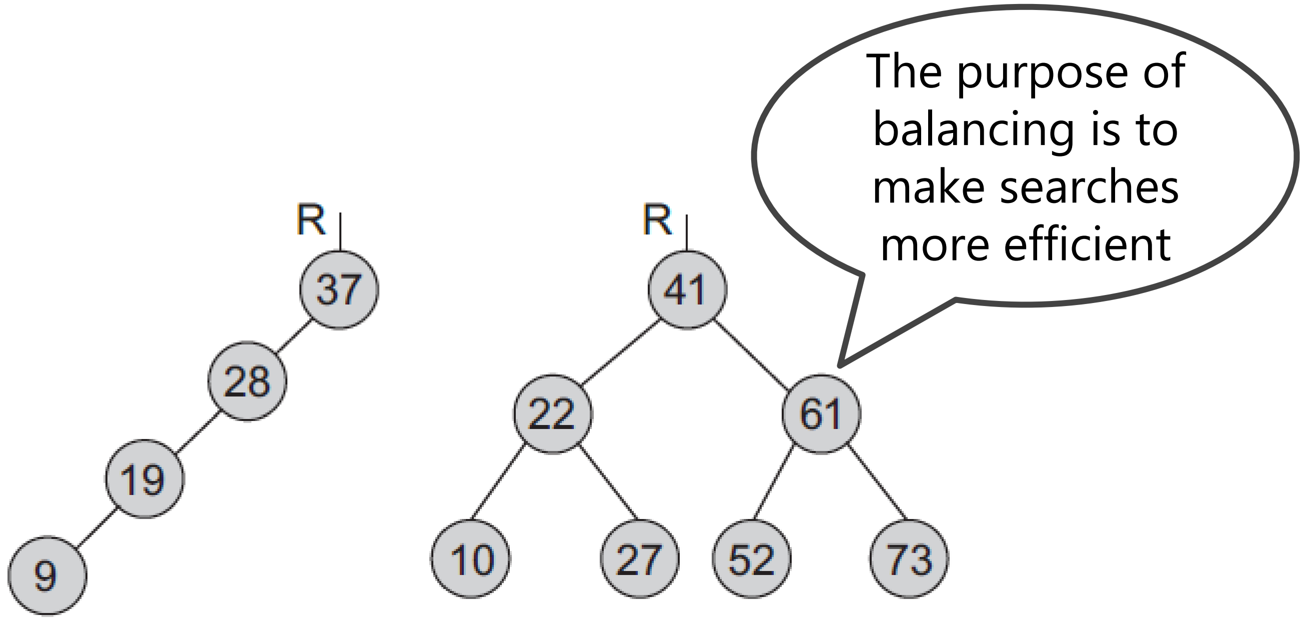


- In this scenario, searches will be complet the purpose of the tree.
- A tree like this is called unbalanced

A tree can get unbalanced



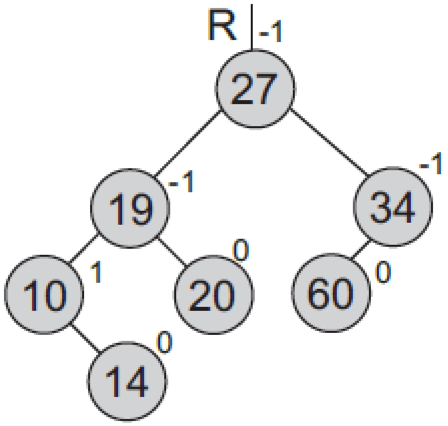
A tree can get unbalanced



AVL tree

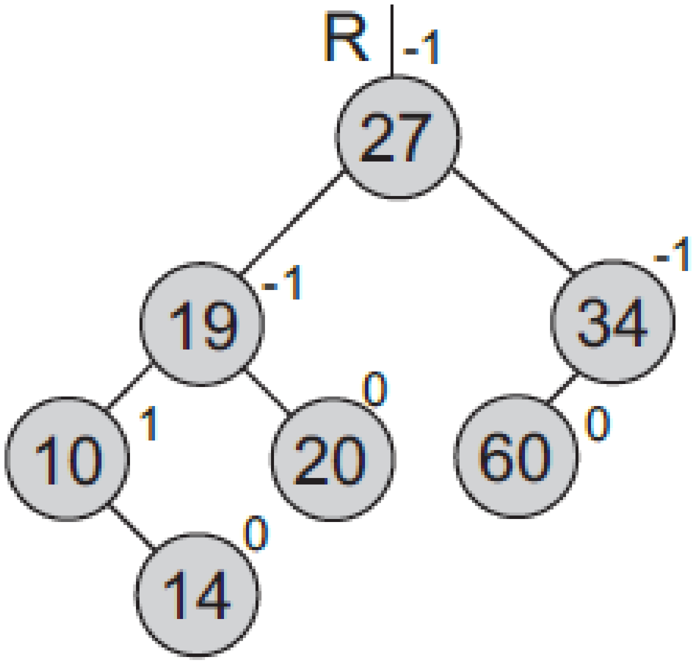
- Each node of an AVL tree has a **balance factor**

- Balance factor is defined as the height of the right subtree minus the height of the left subtree.
    - Balance factor can be 1,0,-1



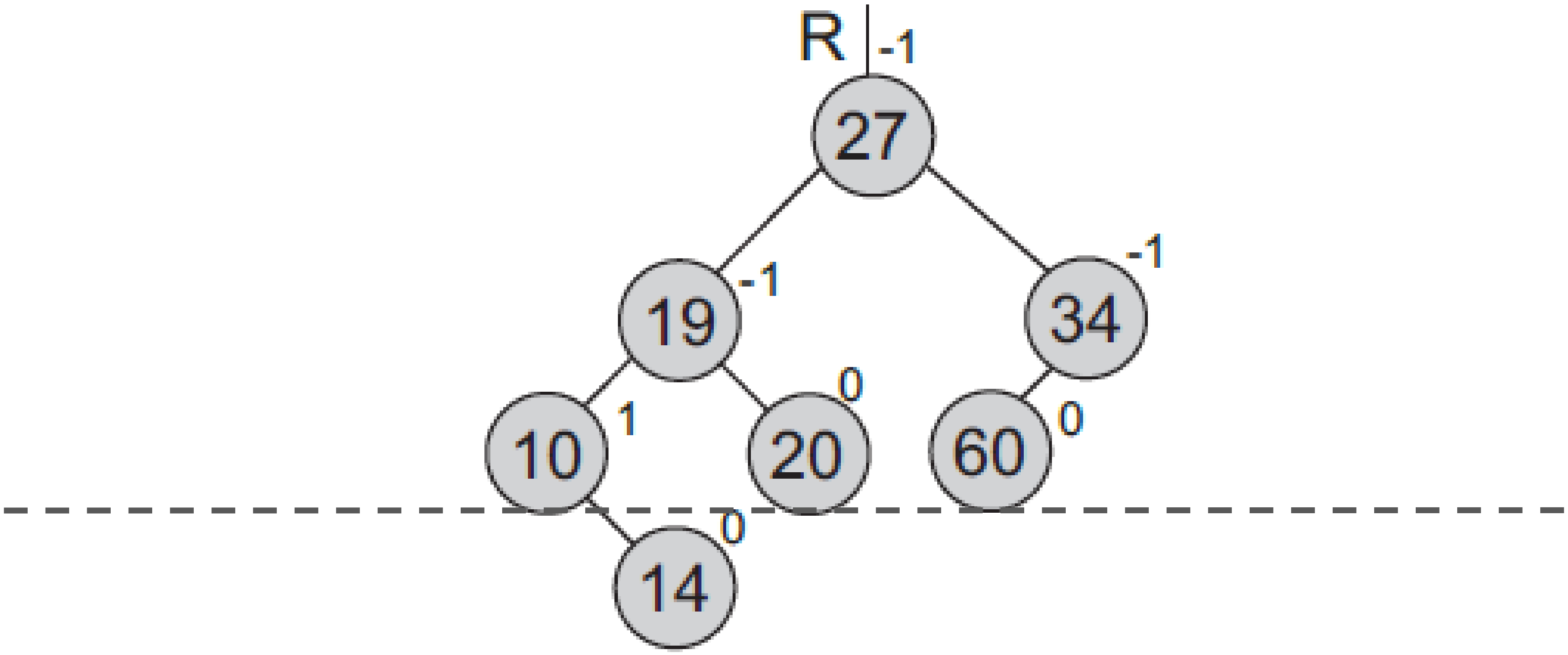
AVL tree

- Let’s calculate balance factor



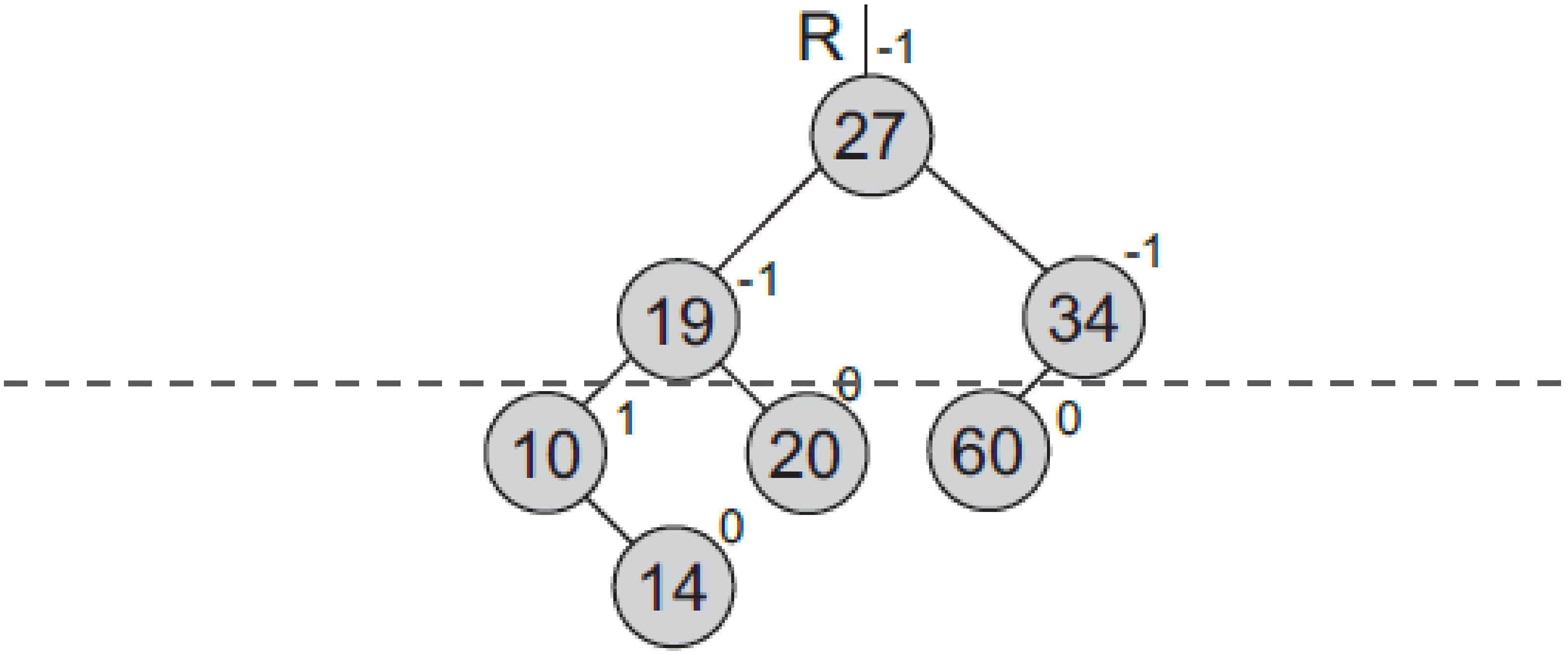
AVL tree

- Let’s calculate balance factor



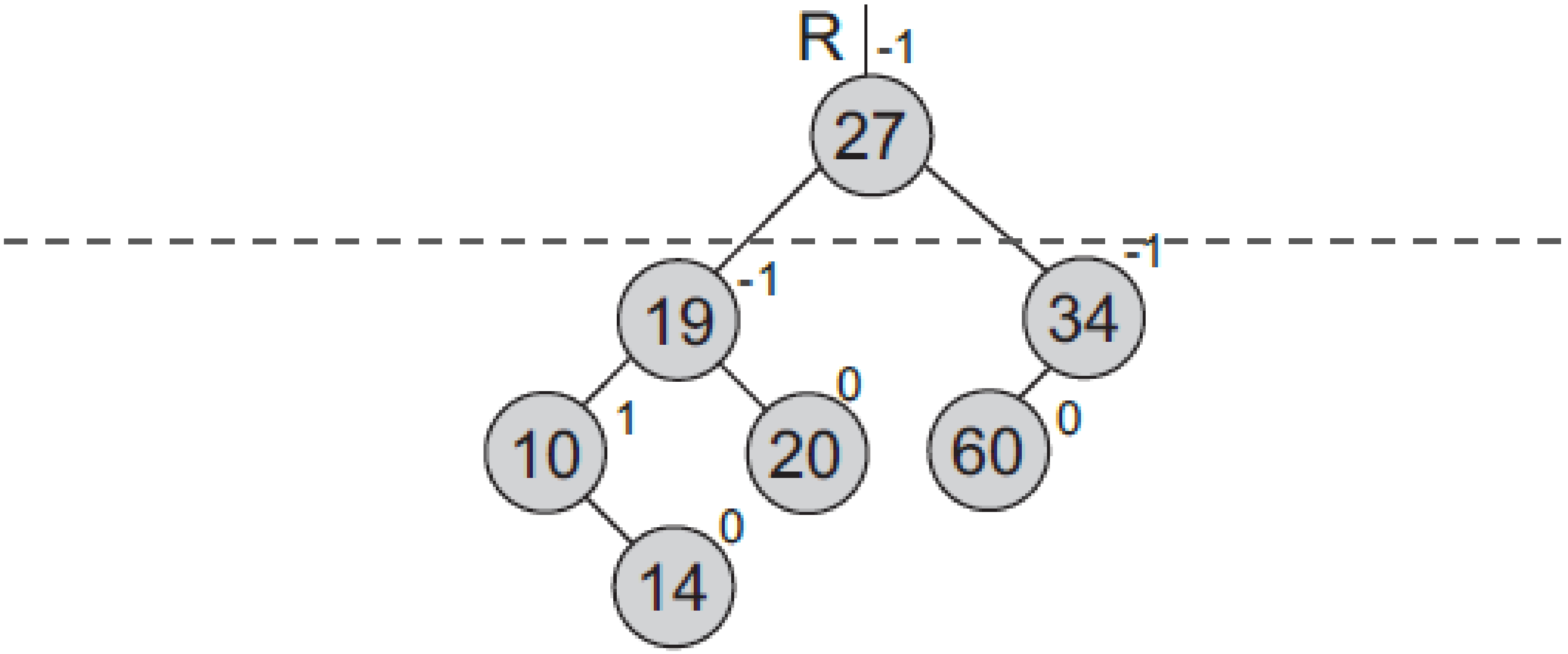
AVL tree

- Let’s calculate balance factor



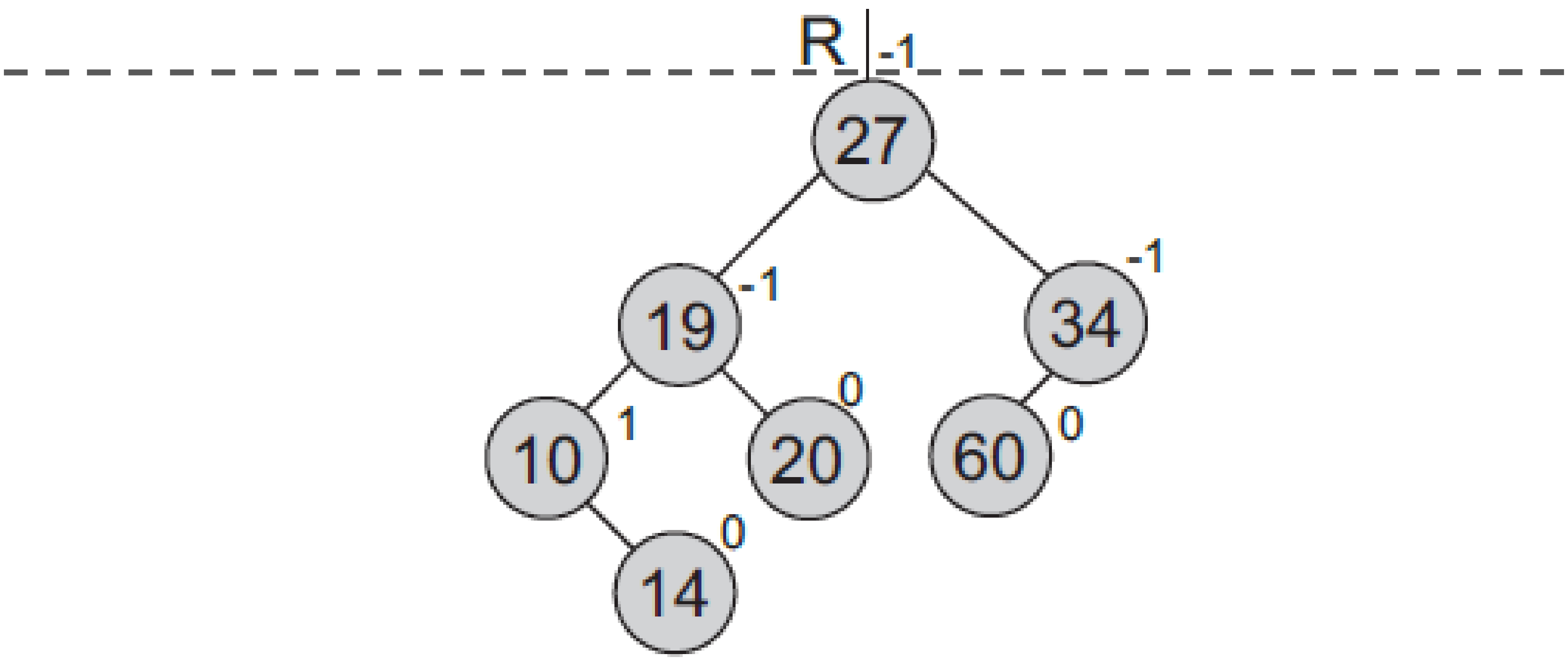
AVL tree

- Let’s calculate balance factor



AVL tree

- Let’s calculate balance factor



AVL Tree
Insert operation

- Inserting a new node can result in a violation of the AVL condition

- If a violation happens, the AVL condition has to be restored using a modification of the tree known as **rotation**

- After the insertion, only nodes that are on the path from the insertion point to the root have their balance altered

AVL Tree
Insert operation

- Follow the same process as the binary search algorithm. New node is added as a leaf with a balance factor of 0

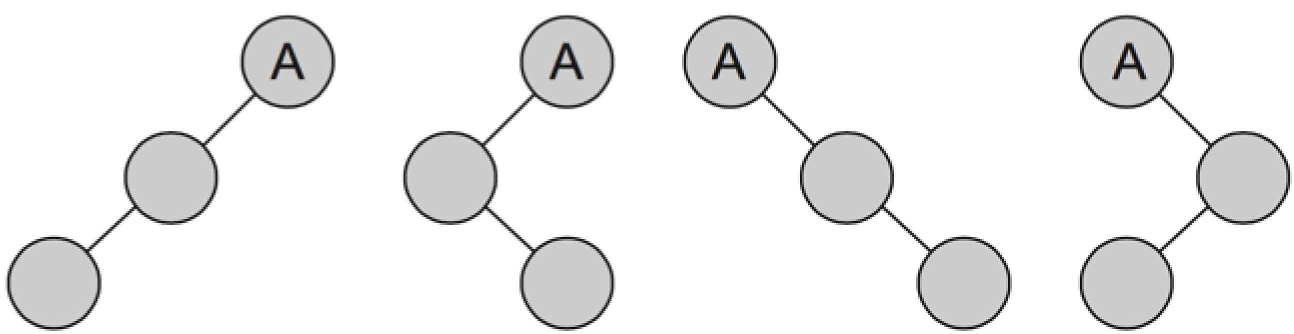
- Backtrack the path to recalculate the balance factor

- Only the nodes in the search path could have change its value

AVL Tree

Insert operation

- Let **α** be the node that needs to be rebalanced, there can be four violation cases:
    a. An insertion in the left subtree of the left child of α
    b. An insertion in the right subtree of the left child of α
    c. An insertion in the left subtree of the right child of α
    d. An insertion in the right subtree of the right child of α

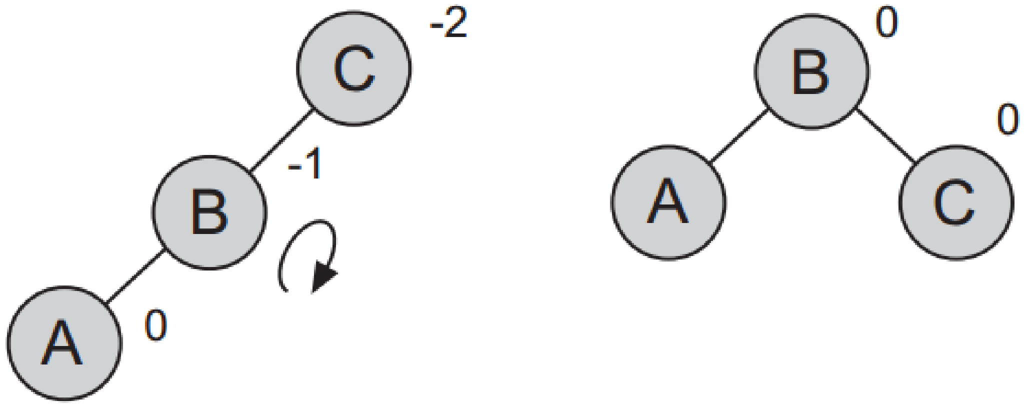


AVL Tree
Insert operation

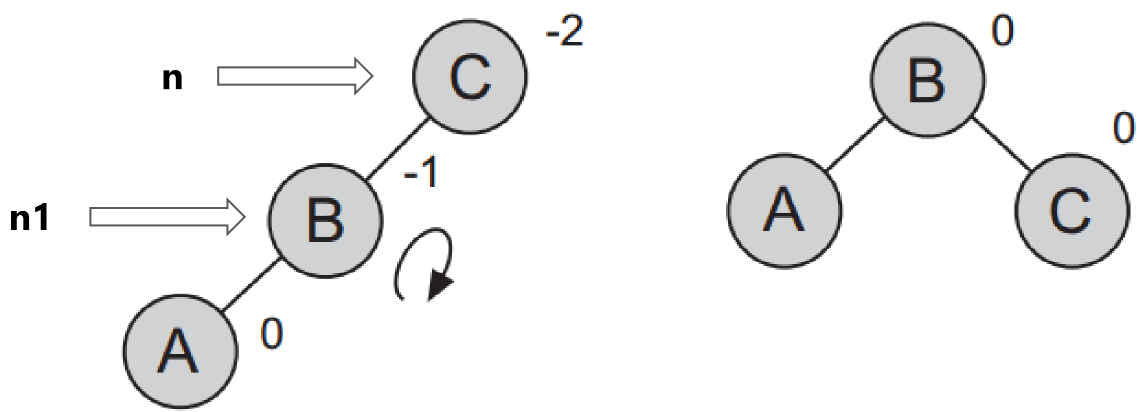
- *Left-left* and *right-right* can be resolved with a single rotation
- *Left-right* and *right-left* can be resolved with a double rotation
- **Backtrack stops** when reaching the root or after performing a rotation. A rotation balances the whole tree

AVL Tree

Single rotation



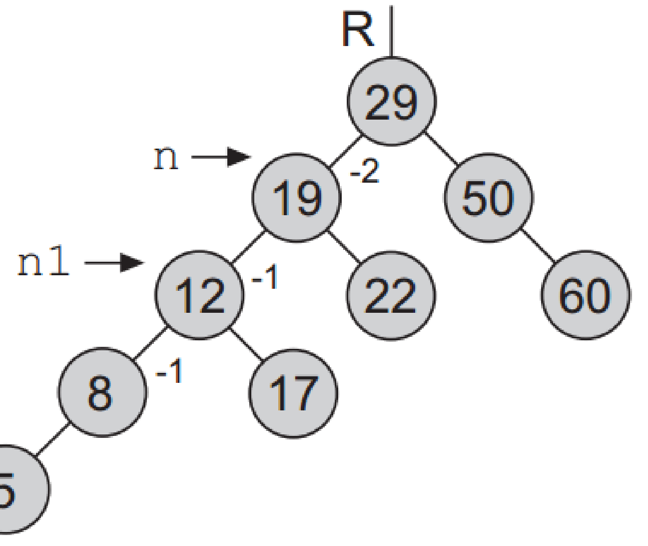
AVL Tree
Single rotation




```
n.left = n1.right;
n1.right = n;
n = n1;
```

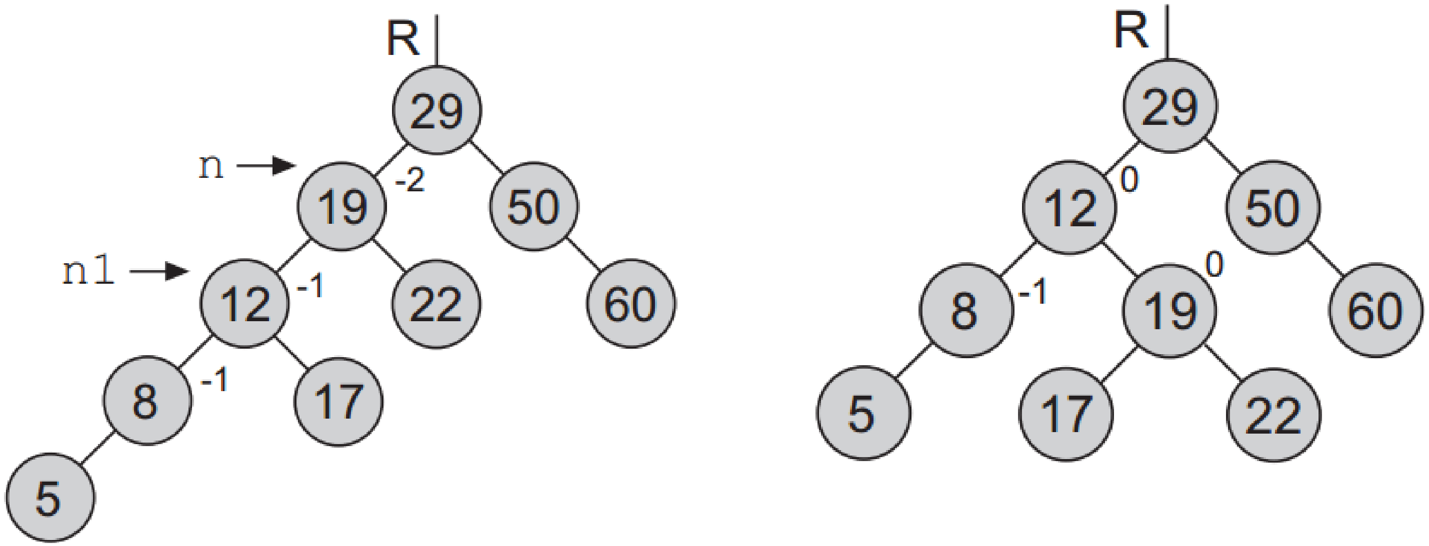
AVL Tree

Single rotation




```
n.left = n1.right;
n1.right = n;
n = n1;
```

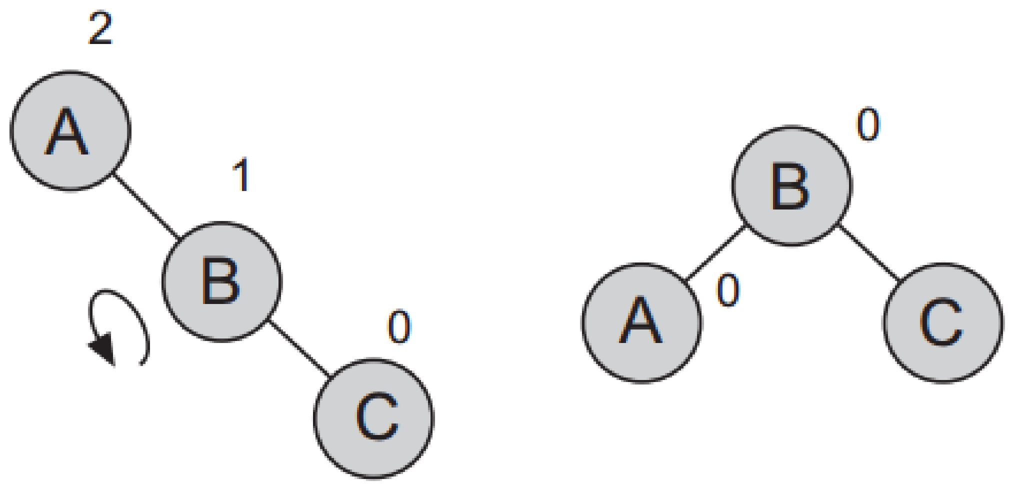
AVL Tree
Single rotation




```
n.left = n1.right;
n1.right = n;
n = n1;
```

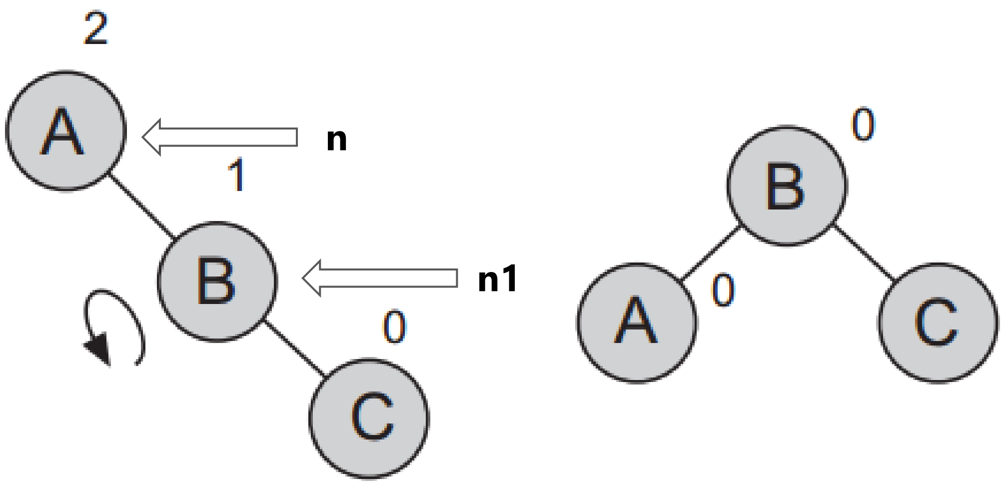
AVL Tree

Single rotation



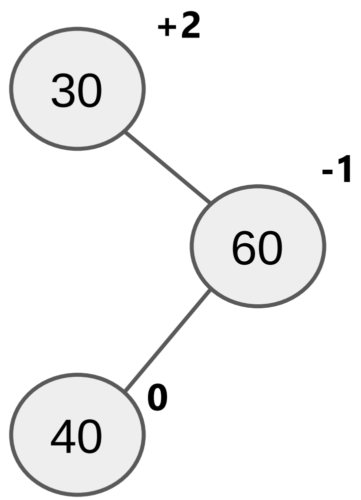
AVL Tree

Single rotation



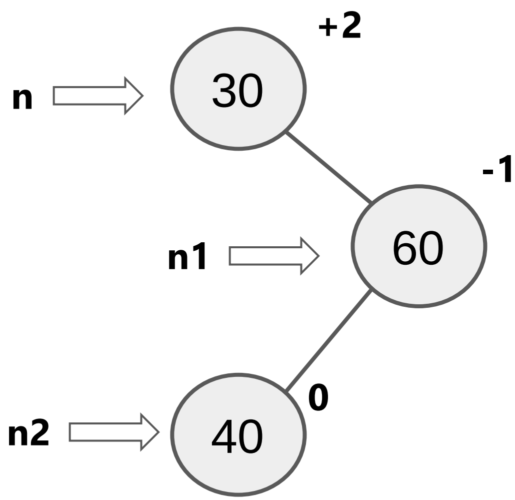

```
n.right = n1.left;
n1.left = n;
n = n1;
```

AVL Tree
Double rotation

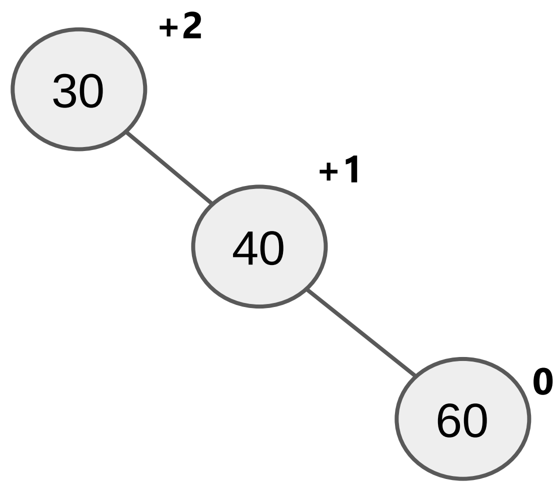


AVL Tree

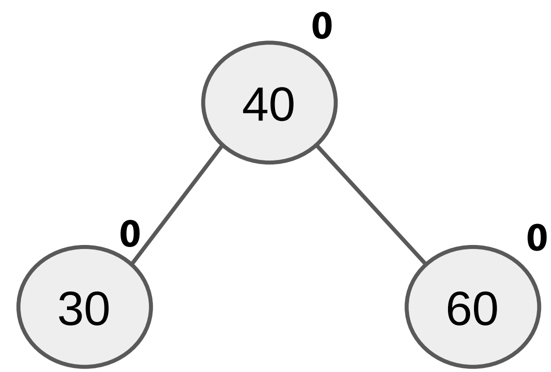
Double rotation



Single rotation (LL) performed on right subtree



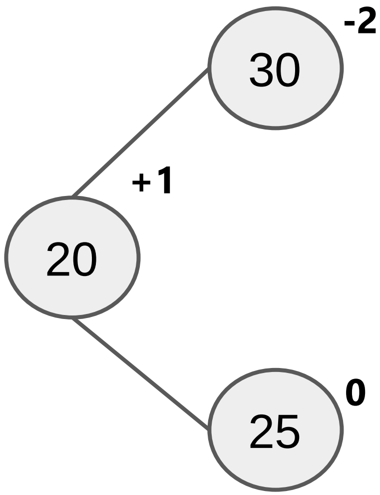
Single rotation (DD) performed on “root”



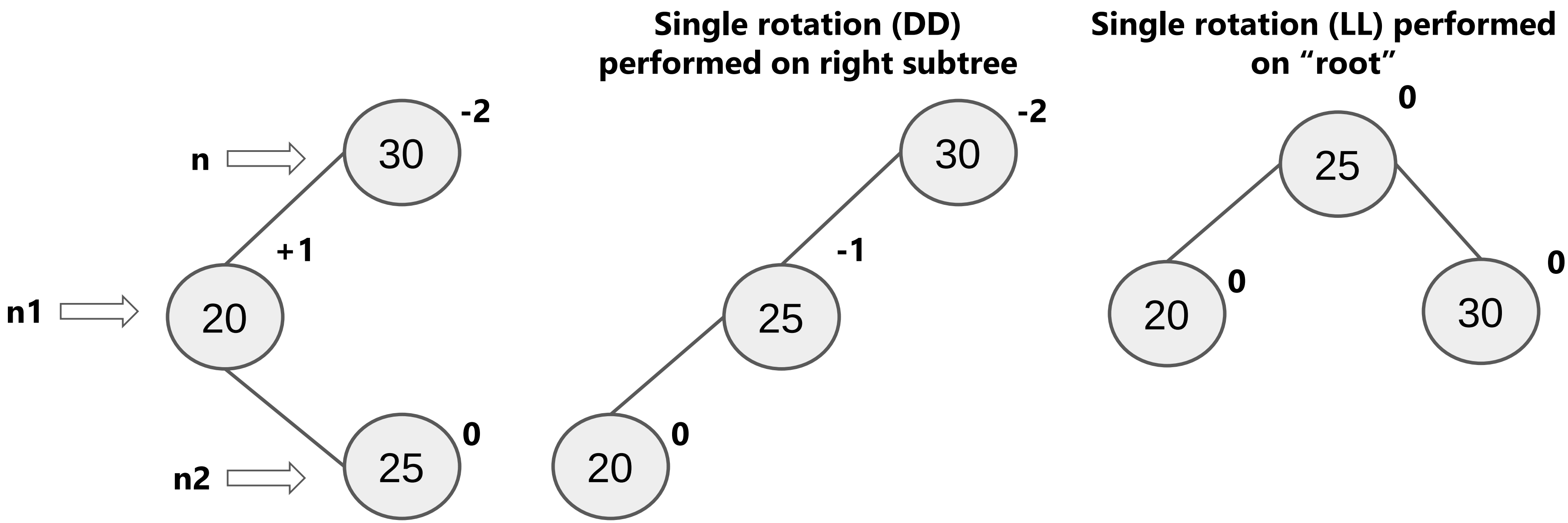

```
n1.left = n2.right;
n2.right = n1;
n.right = n2.left;
n2.left = n;
n = n2;
```

AVL Tree

Double rotation



AVL Tree
Double rotation




```
n1.right = n2.left;
n2.left= n1;
n.left = n2.right;
n2.right = n;
n = n2;
```

AVL Tree
Implementation

```
public class AvlNode {
  int element;
  AvlNode left;
  AvlNode right;
  int height;

  public AvlNode(int element) {
    this(element, null, null);
  }

  public AvlNode(int element, AvlNode left, AvlNode right) {
    this.element = element;
    this.left = left;
    this.right = right;
    this.height = 0;
  }
}
```

AVL Tree
Implementation

```
public class AvlTree {
  private static final ALLOWED_IMBALANCE = 1;
  private int height(AvlNode t) {
    return t == null ? -1 : t.height;
  }
  private AvlNode insert(int x, AvlNode t) {
    if (t == null) {
      return new AvlNode(x);
    }

    if (x < t.element) {
      t.left = insert(x, t.left);
    } else if (x > t.element) {
      t.right = insert(x, t.right);
    }
    return balance(t);
  }
}
```

AVL Tree
Implementation

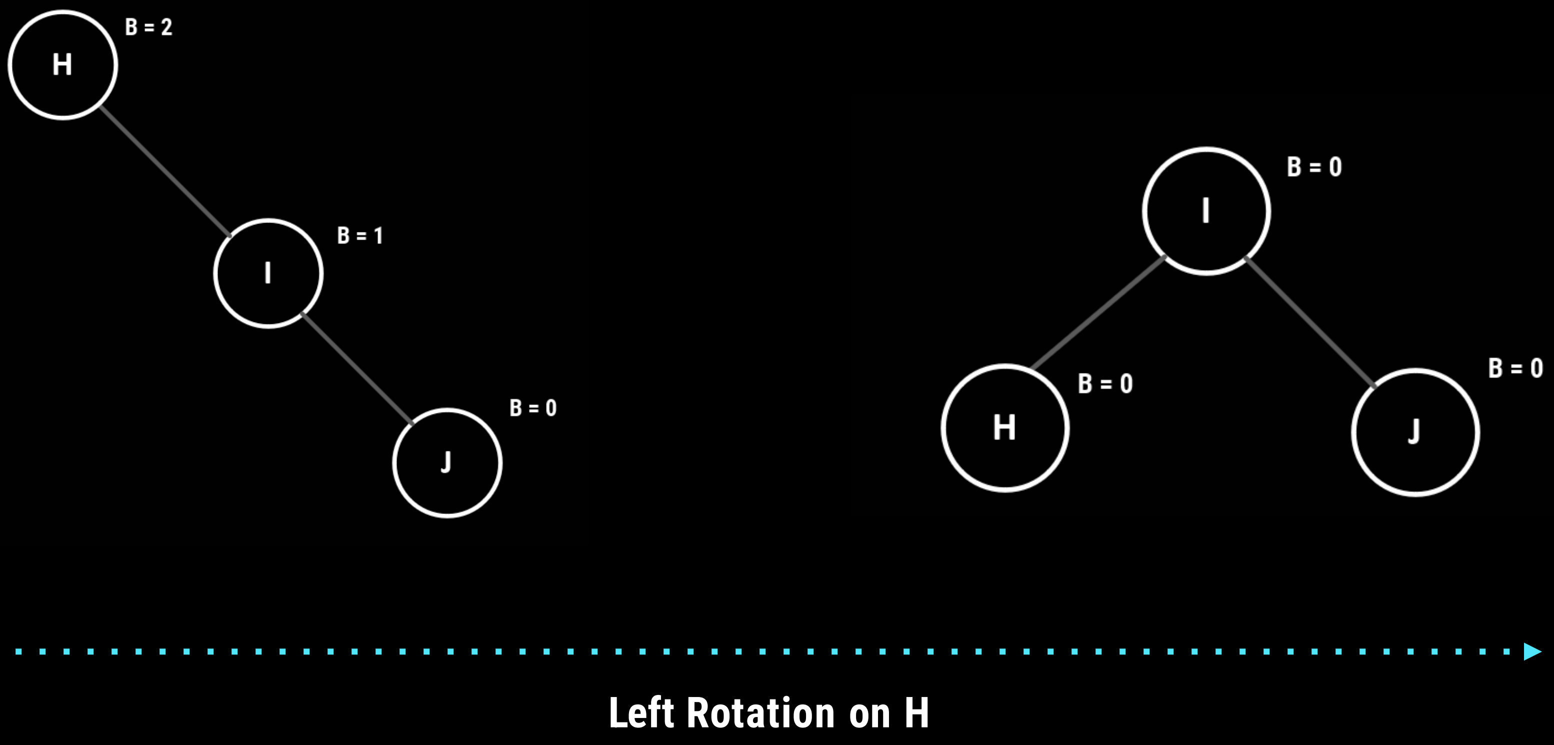
```
private AvlNode balance(AvlNode t) {
  if (t == null)
    return t;

  if (height(t.left) - height(t.right) > ALLOWED_IMBALANCE) {
    if (height(t.left.left) >= height(t.left.right))
      t = rotateWithLeftChild(t);
    else
      t = doubleWithLeftChild(t);
  } else {
    if (height(t.right) - height(t.left) > ALLOWED_IMBALANCE){
      if (height(t.right.right) >= height(t.right.left))
        t = rotateWithRightChild(t);
      else
        t = doubleWithRightChild(t);
    }
  }
  t.height = Math.max(height(t.left), height(t.right)) + 1;
  return t;
}
```

AVL Tree
Implementation

```
private AvlNode rotateWithLeftChild(AvlNode k2) {
  AvlNode k1 = k2.left;
  k2.left = k1.right;
  k1.right = k2;
  k2.height = Math.max(height(k2.left), height(k2.right)) + 1;
  k1.height = Math.max(height(k1.left), k2.height) + 1;
  return k1;
}
private AvlNode doubleWithLeftChild(AvlNode k3) {
  k3.left = rotateWithRightChild(k3.left);
  return rotateWithLeftChild(k3);
}
```

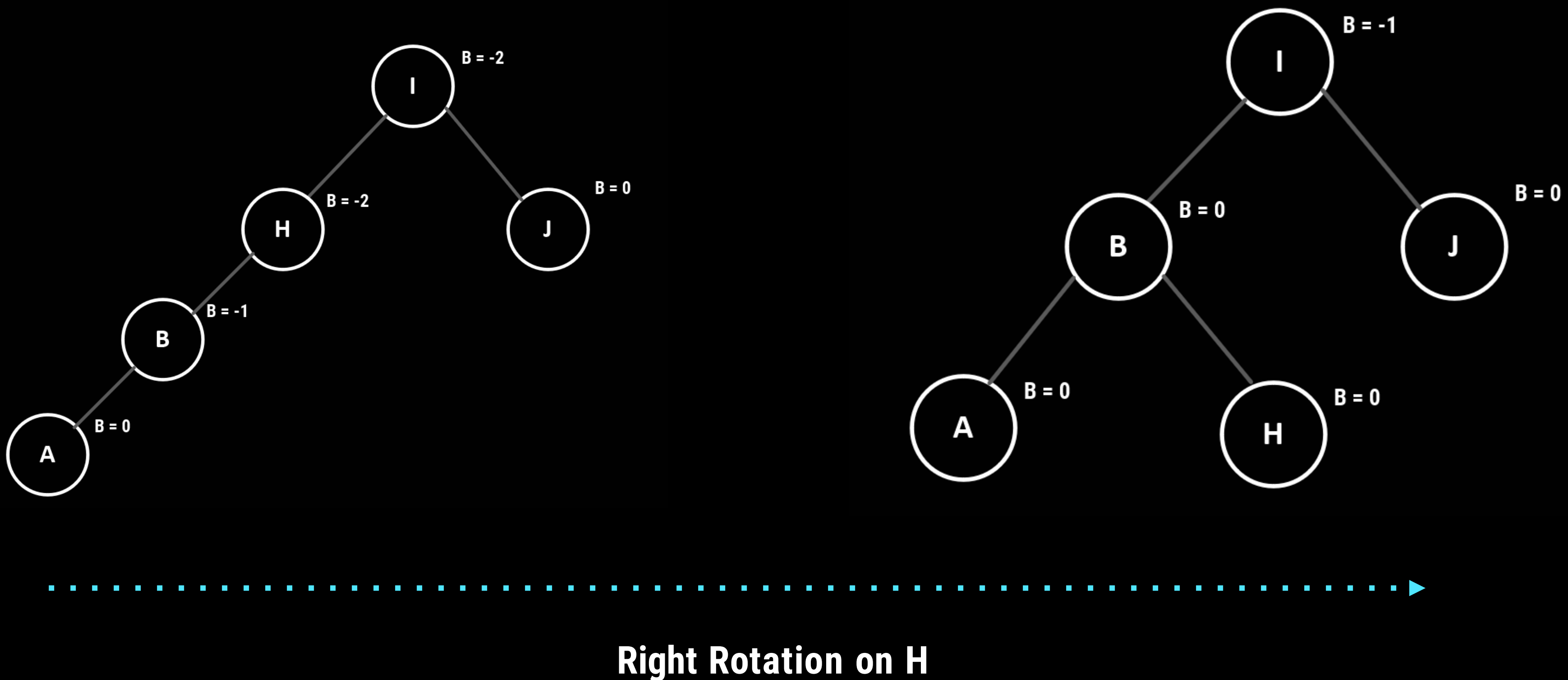
ANOTHER AVL INSERT EXAMPLE

Insert the values: H, I, J, B, A, E



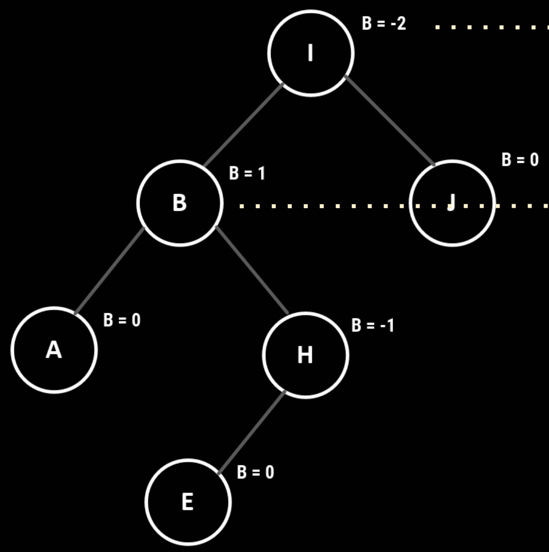
ANOTHER AVL INSERT EXAMPLE

Insert the values: B, A, E



ANOTHER AVL INSERT EXAMPLE

Insert the values: E



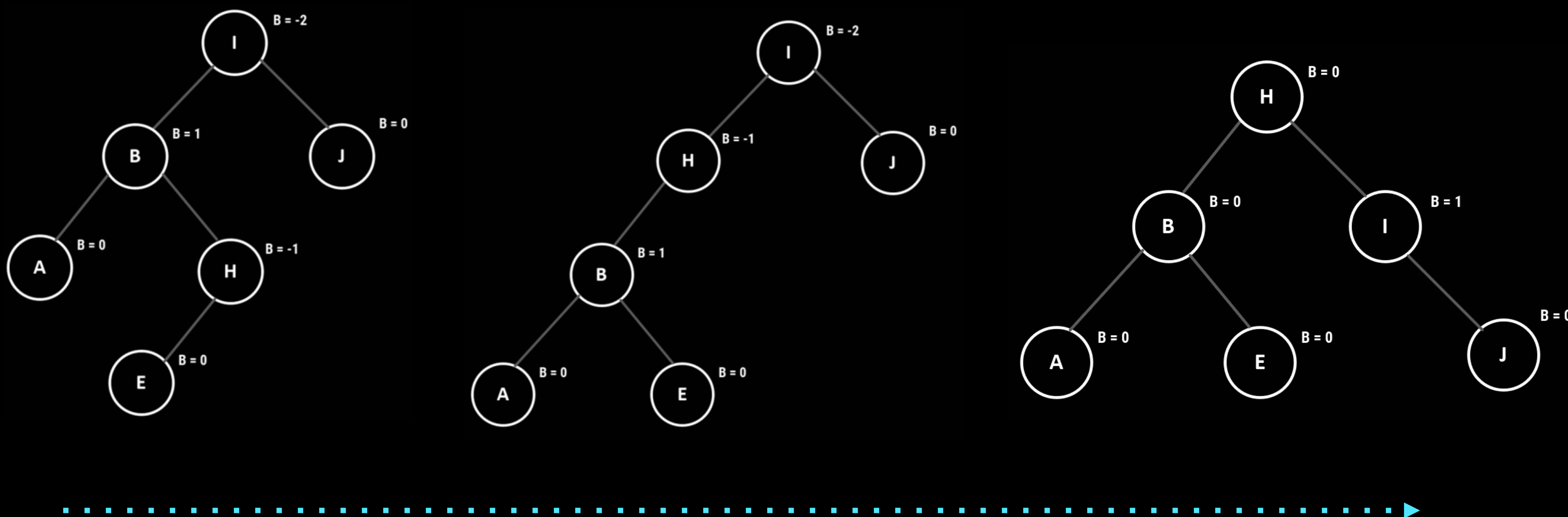
-2 indicates left imbalance

1 indicates right imbalance

Left Right case

ANOTHER AVL INSERT EXAMPLE

Insert the values: E



Left Rotation on B **Right Rotation on I**

Splay tree

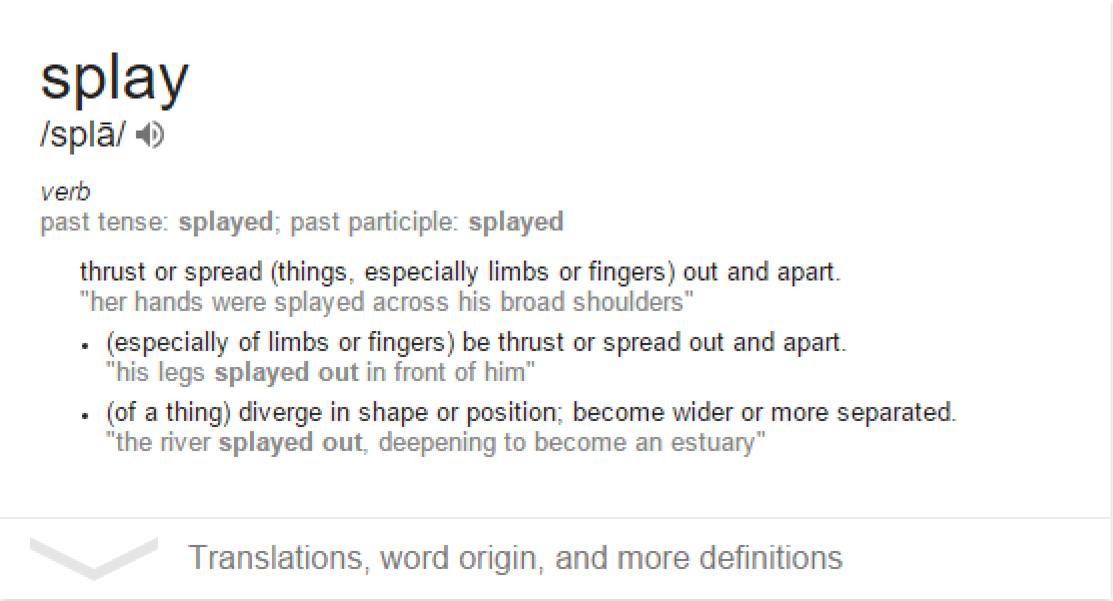
What is a splay tree?

- Is an efficient implementation of a binary search tree that takes advantage of **locality**
  - If a node is accessed, it may probably be accessed frequently

- Is an special tree that focus on **reducing the access time**

What is a splay tree?

- Is an efficient implementa
  locality
  - If a node is a
    accessed fre



- Is an special tree that focus on **reducing the access time**

What is a splay tree?

- Whenever a node is looked up in the tree, the splay tree reorganizes to move that element to the root of the tree

- Elements that are used frequently will likely be near to the top of the tree

- There is no height or balance data maintenance

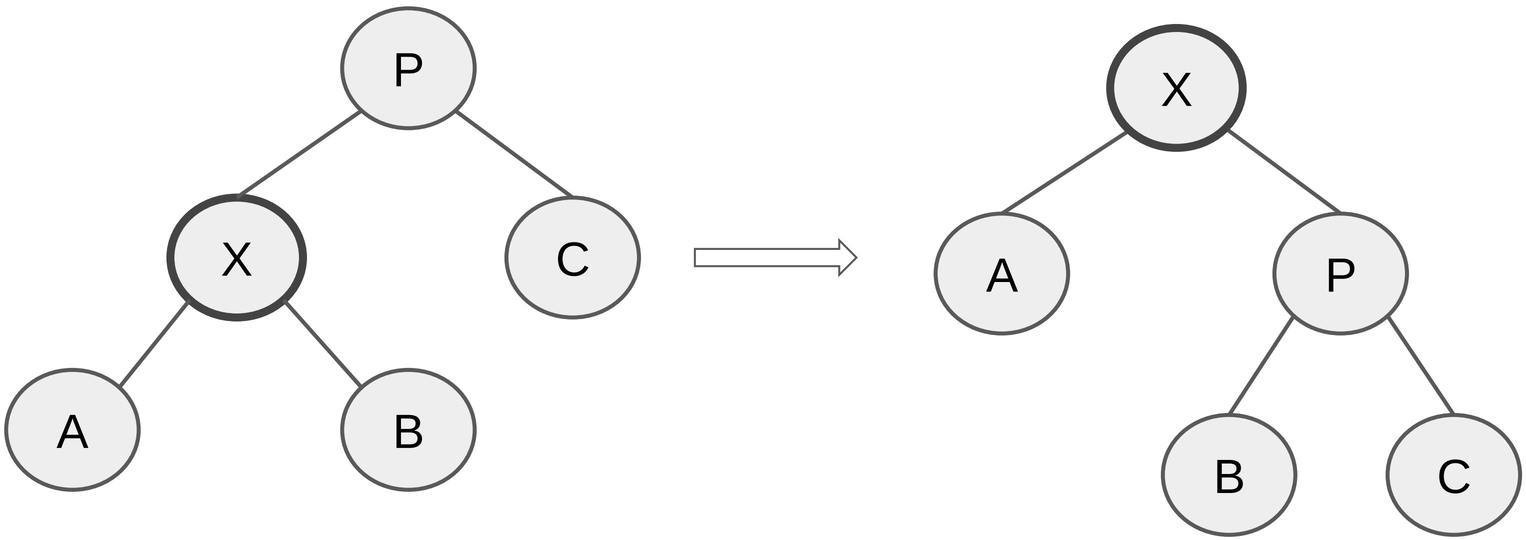
What is a splay tree?

- There are three types of rotations:
  - Zig or Simple rotation
  - Zig-Zag
  - Zig-Zig

Simple rotation (Zig)

- Let X be a non root node on the access path at which we are rotating
- If the parent of X is the root of the tree, we merely rotate X and the root
- Is the same as simple rotation in the AVL tree

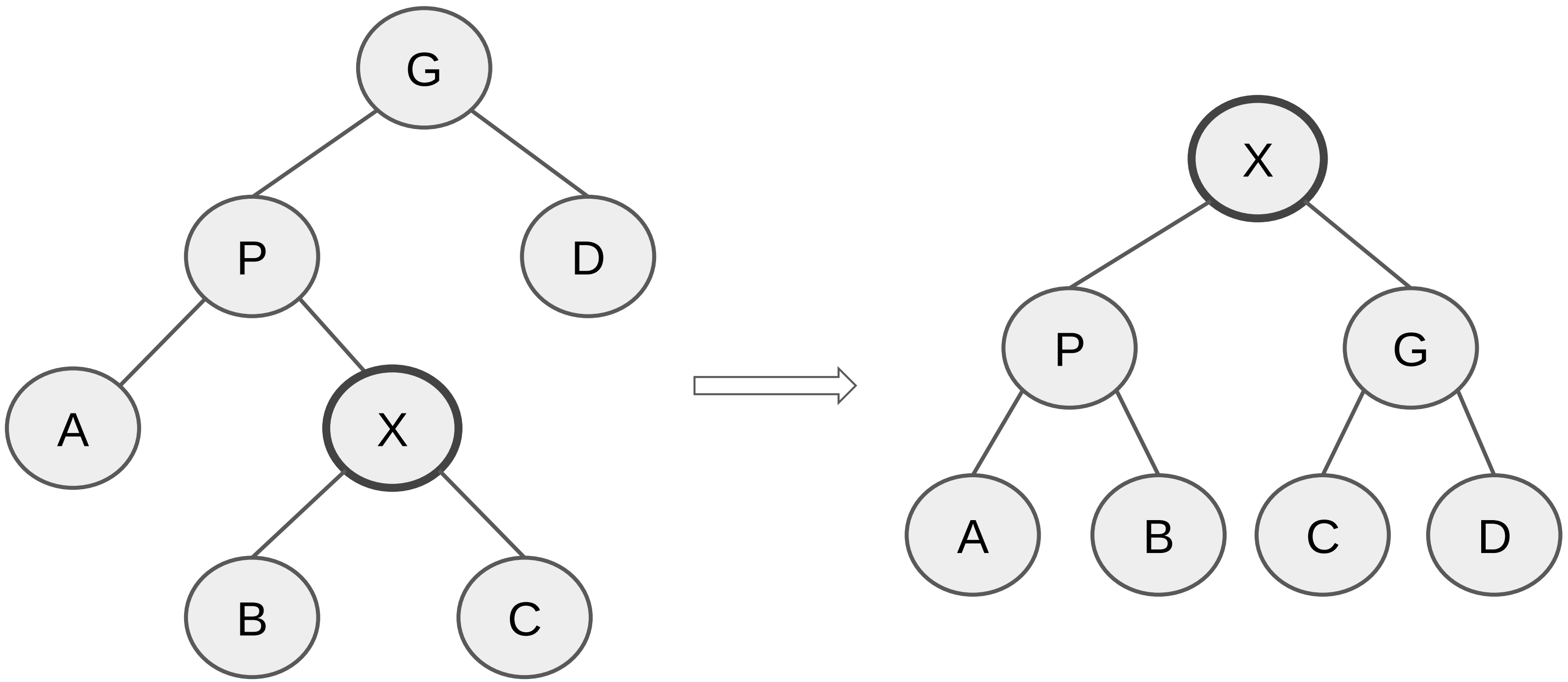
Simple rotation (Zig)



Zig-Zag

- This case involves X, a parent P and a grand-parent G.
- X is a right child and P is a left child
- Is the same as a double rotation of AVL

Zig-Zag

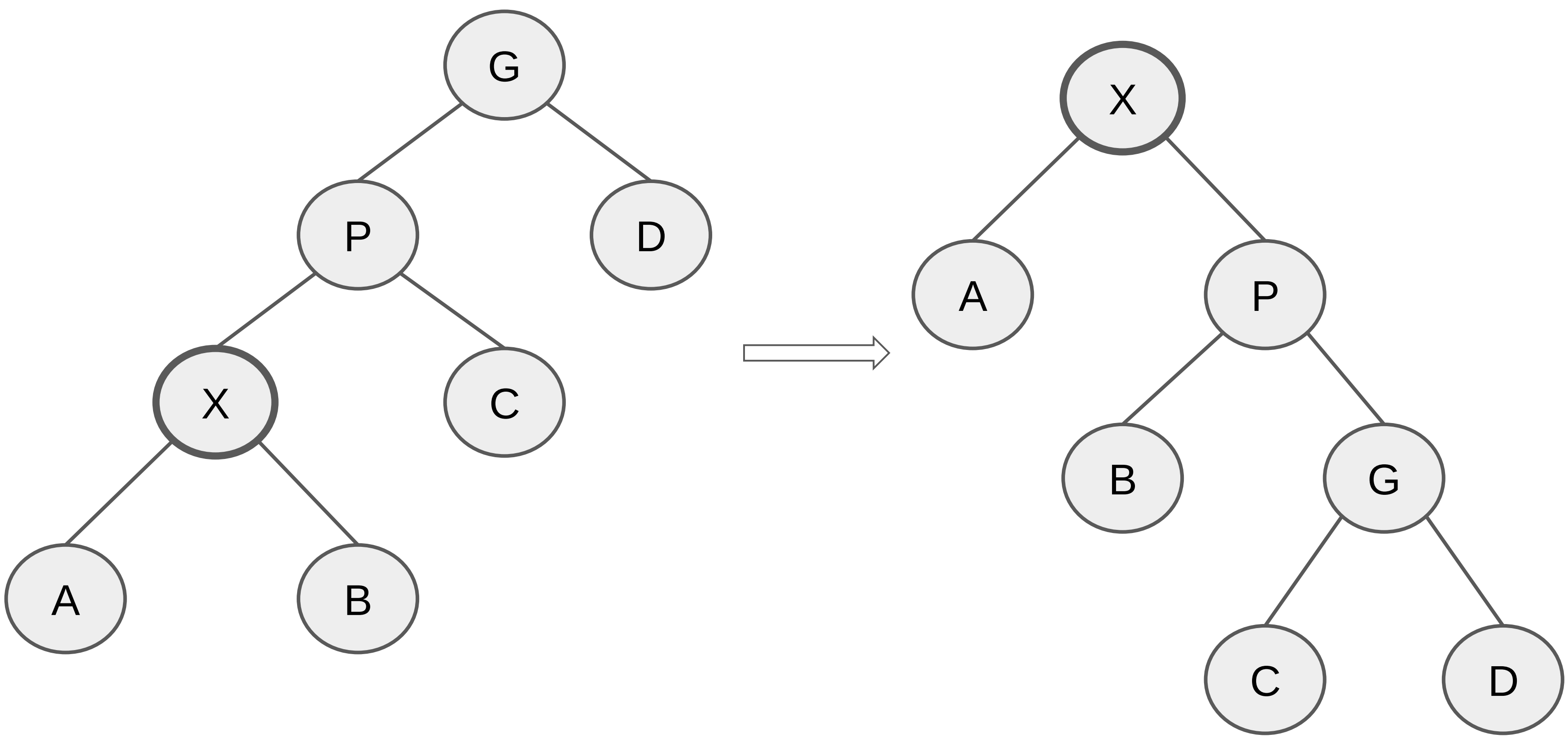


Zig-Zig

- A new kind of rotation

- The parent node P and X are either both left children or both right children

Zig-Zig



Splay tree operations

- **Insertion**: when an item is inserted, a splay is performed
  - Newly inserted item becomes the root of the tree

- **Find**: the last accessed during the search is splayed
  - If the search is successful, the node found is splayed
  - If the search is unsuccessful, the last node compared is splayed

Splay tree operations

- **FindMin and FindMax**: perform a splay after the success

- **DeleteMin**
  - Perform a FindMin. This brings the minimum to the root and there will be no left child
  - Use the right child as the new root

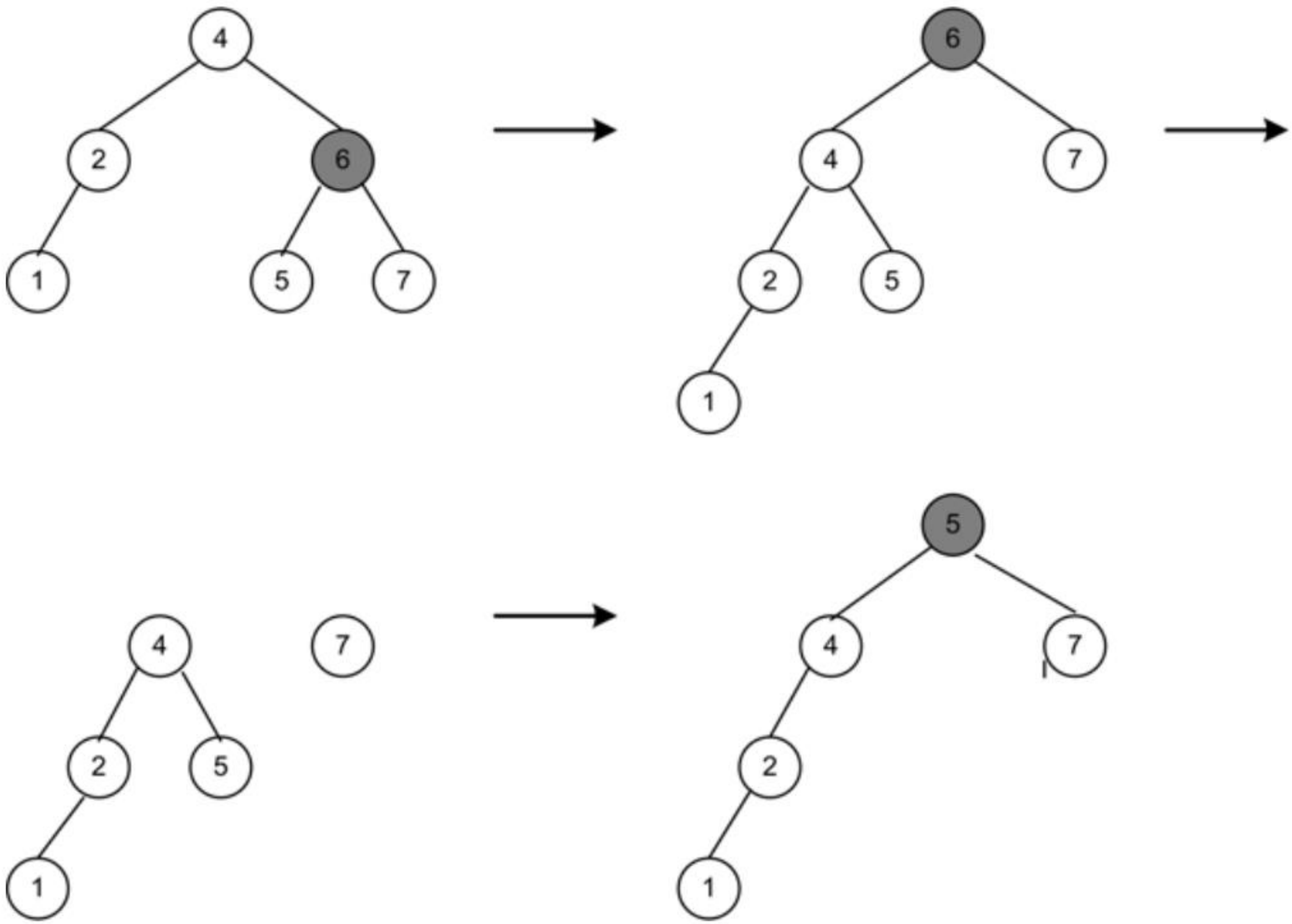
Splay tree operations

- **DeleteMax**
  - Perform a FindMax
  - Set the root to the post-splay left child

Splay tree operations

- **Remove**:
  - Access the node to be deleted, bringing it to the root
  - Delete the root leaving two subtress left (L) and right (R)
  - Find the largest in L using DeleteMax operation, thus the root of L will have no right child
  - Make R the right child of L's root

Splay tree operations

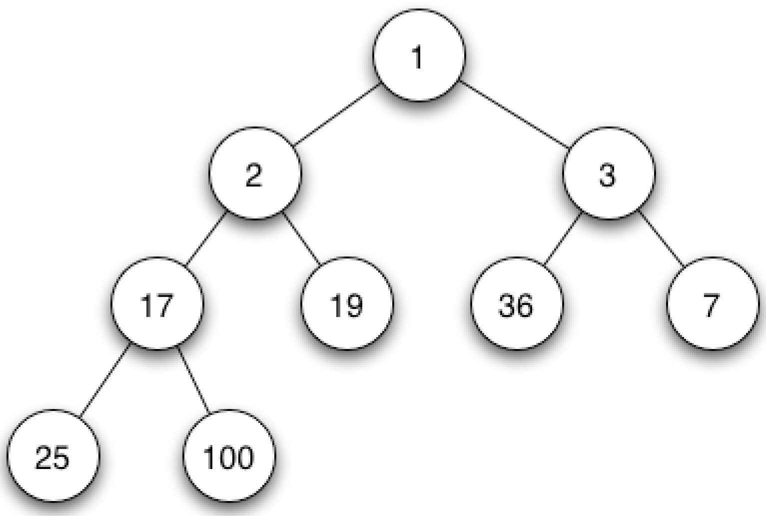


Heap

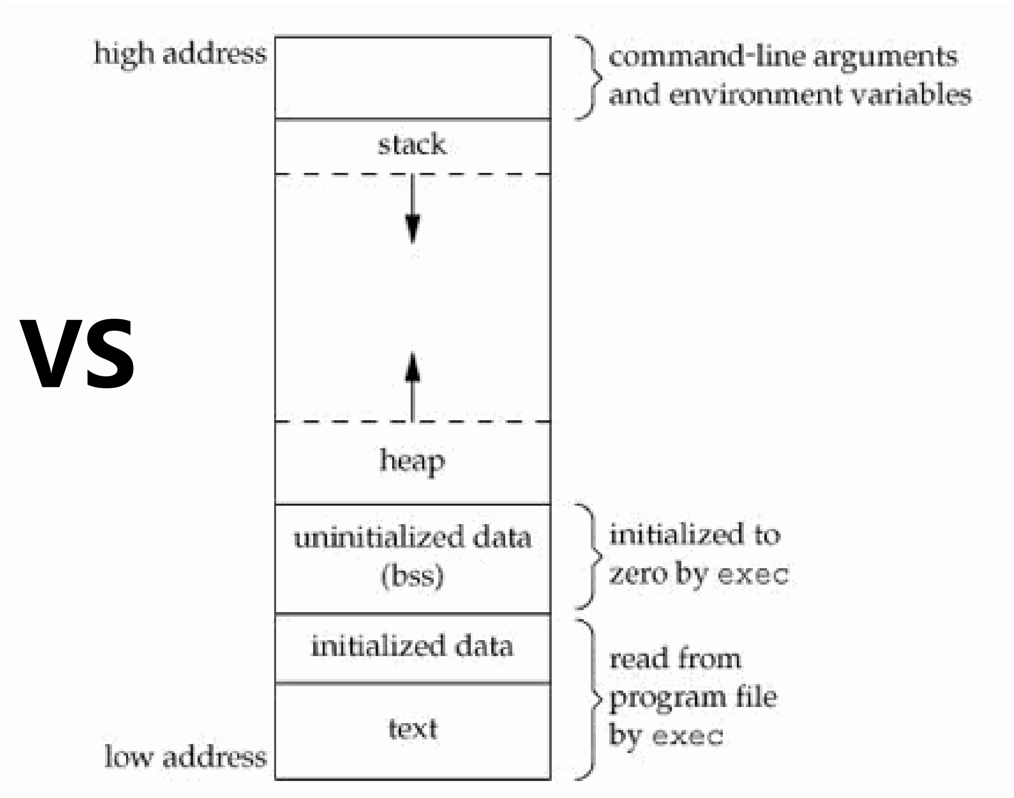
What is a heap?

- In a previous topic, we learned about **priority queues**
- Traditionally a priority queue can be implemented with arrays or linked list
- There an special kind of tree that can be used to implement a priority queue: heap

What is a heap?



VS



What is a heap?

- Why to choose a heap instead of an array or linked list?
    - Fast insertion O(logN)

- A heap is a binary tree with the following characteristics
    - It is complete
    - Implemented as an array
    - Satisfies the heap condition

What is a heap?

- Why to choose a heap instead of an array or linked list?
  - Fast insertion O(logN)

- A heap is a binary tree with the following charac
  - It is complete
  - Implemented as an array
  - Satisfies the heap condition

It is completely filled in. Reading from left to right across each row, although the last row does not need to be full

What is a heap?

- Why to choose a heap instead of an array or linked list?
    - Fast insertion O(logN)

- A heap is a binary tree with the following characteristics
    - It is complete
    - Implemented as an array
    - Satisfies the heap condition

Stored as an array rather than using references as in a linked list

What is a heap?

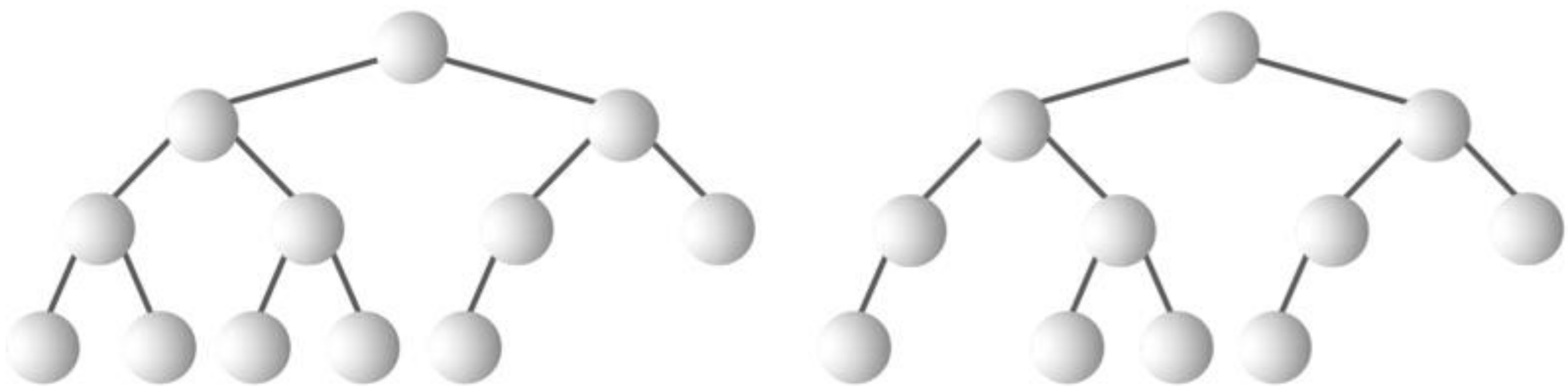
- Why to choose a heap instead of an array or linked list?
  - Fast insertion O(logN)

- A heap is a binary tree with the following characteristics
  - It is complete
  - Implemented as an array
  - Satisfies the heap condition

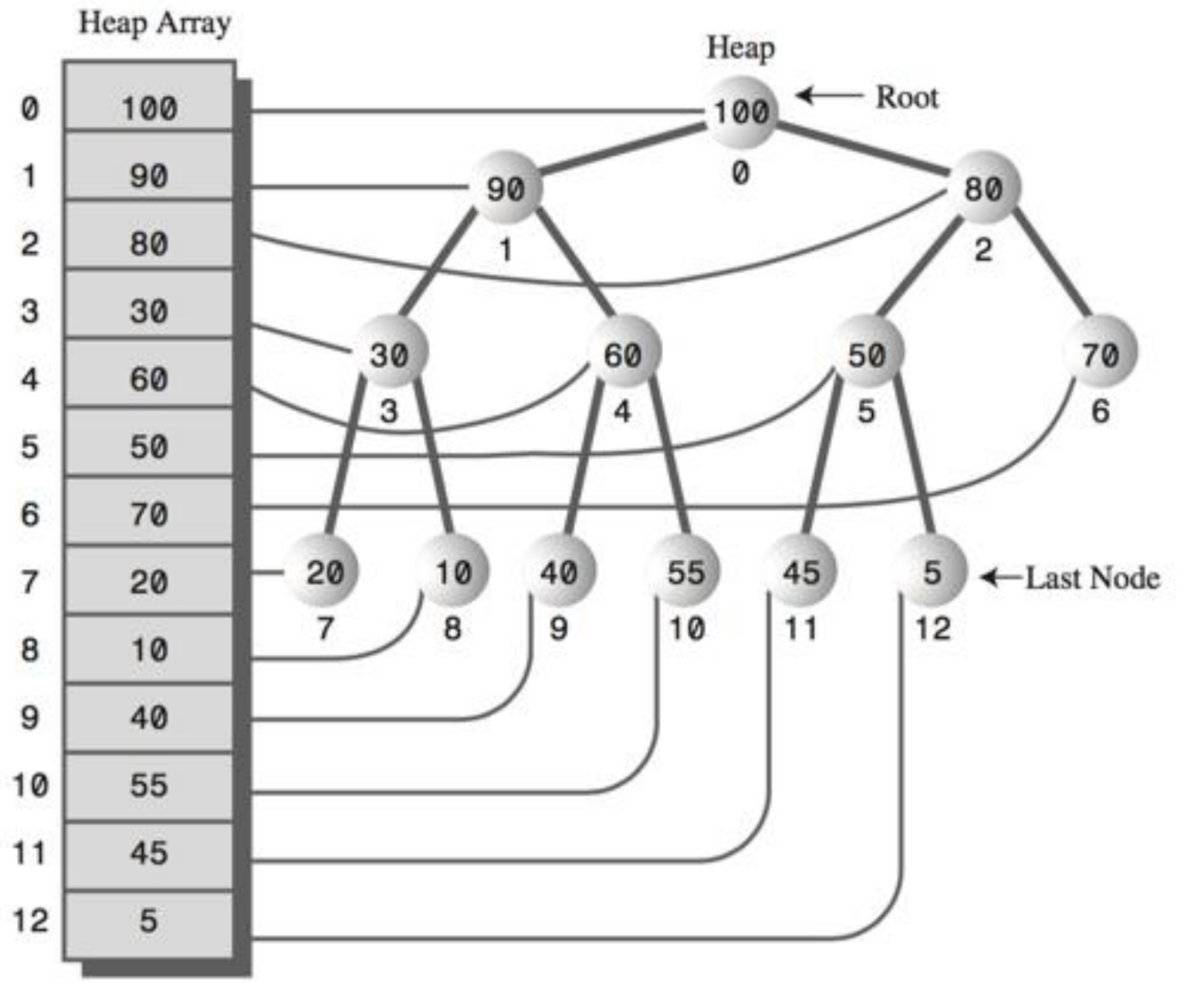
Each node in a heap satisfies the **heap condition**, which means that every node's key is larger than the key of its children

Completeness

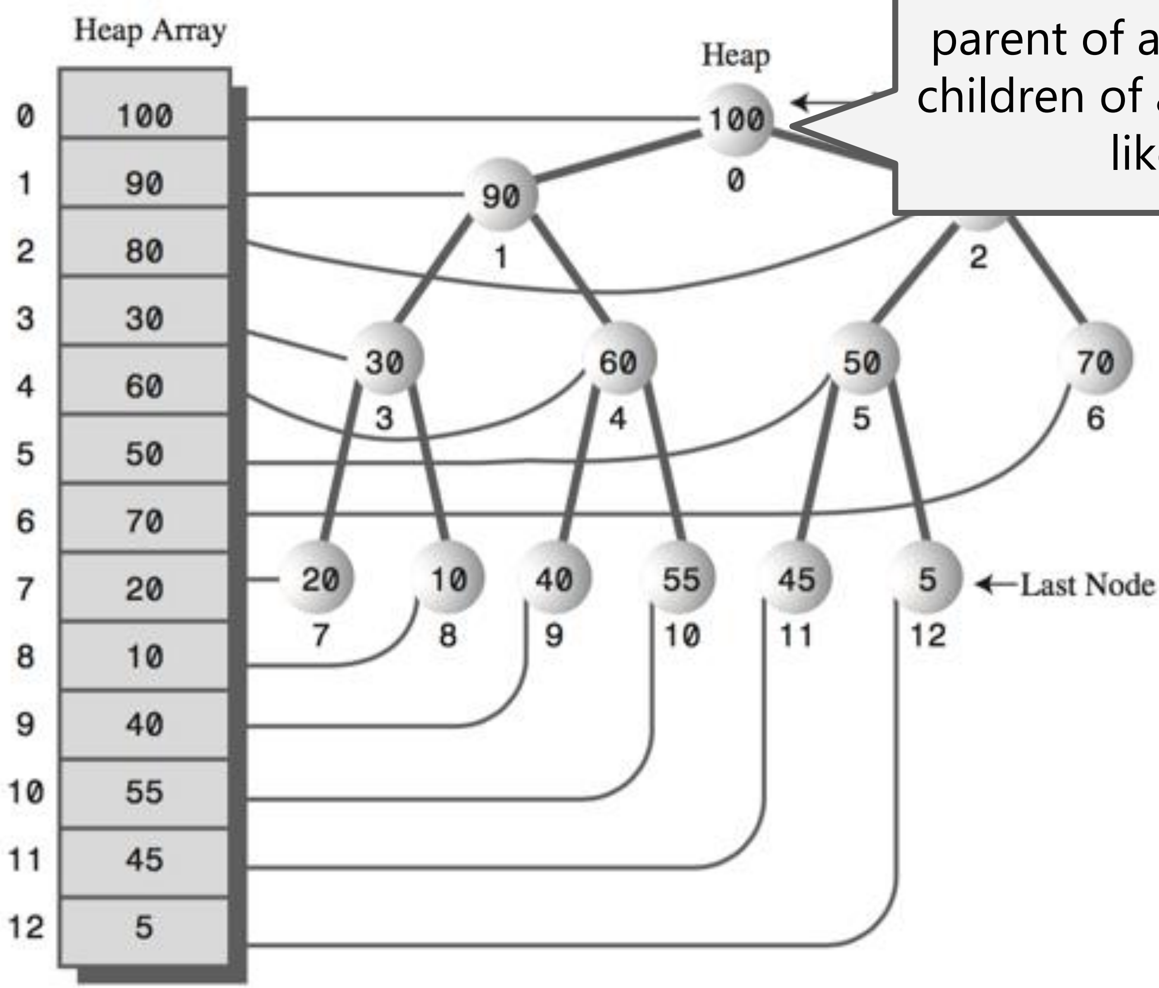
Complete **VS** Incomplete



Implementation

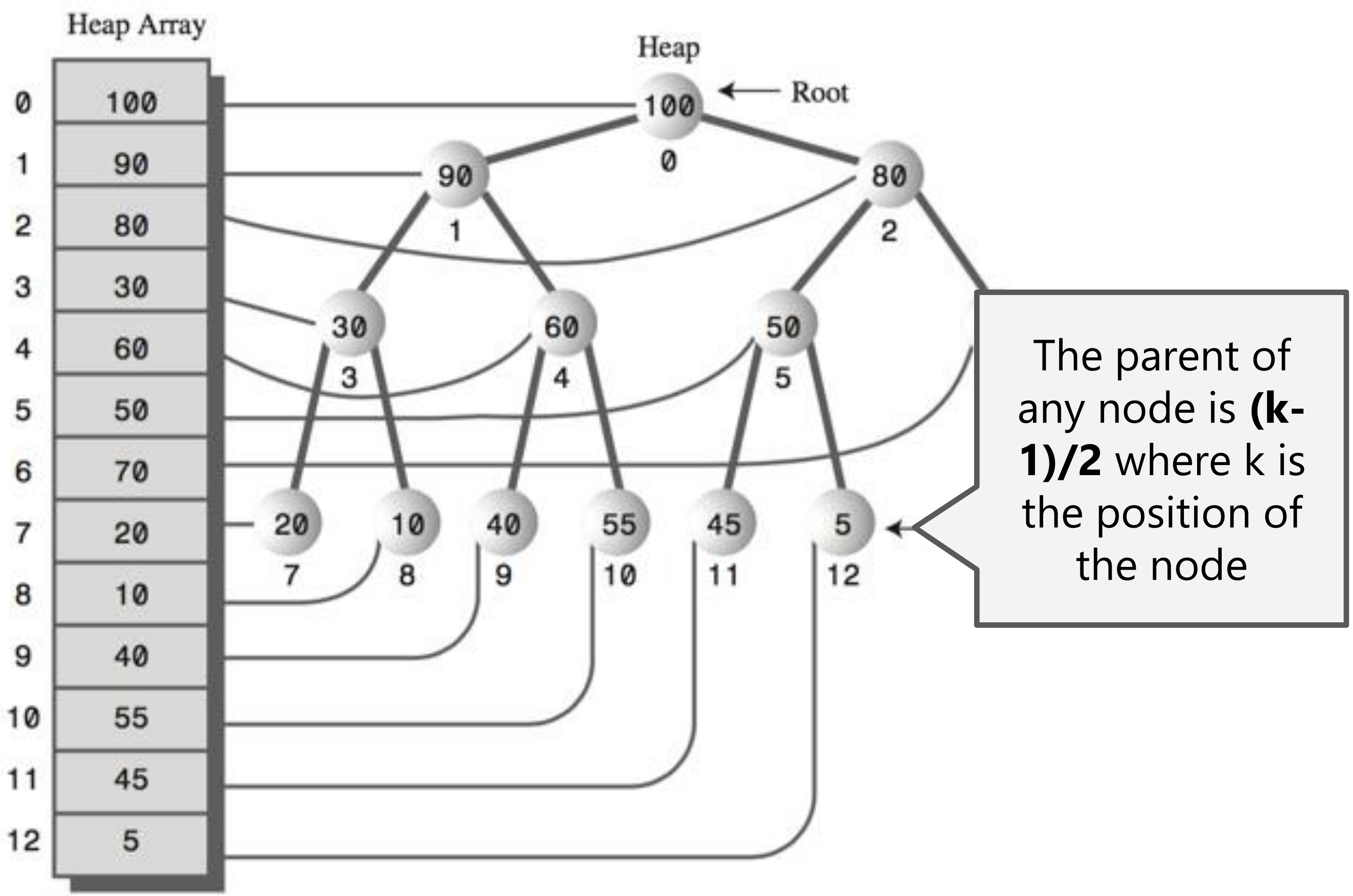


Implementation

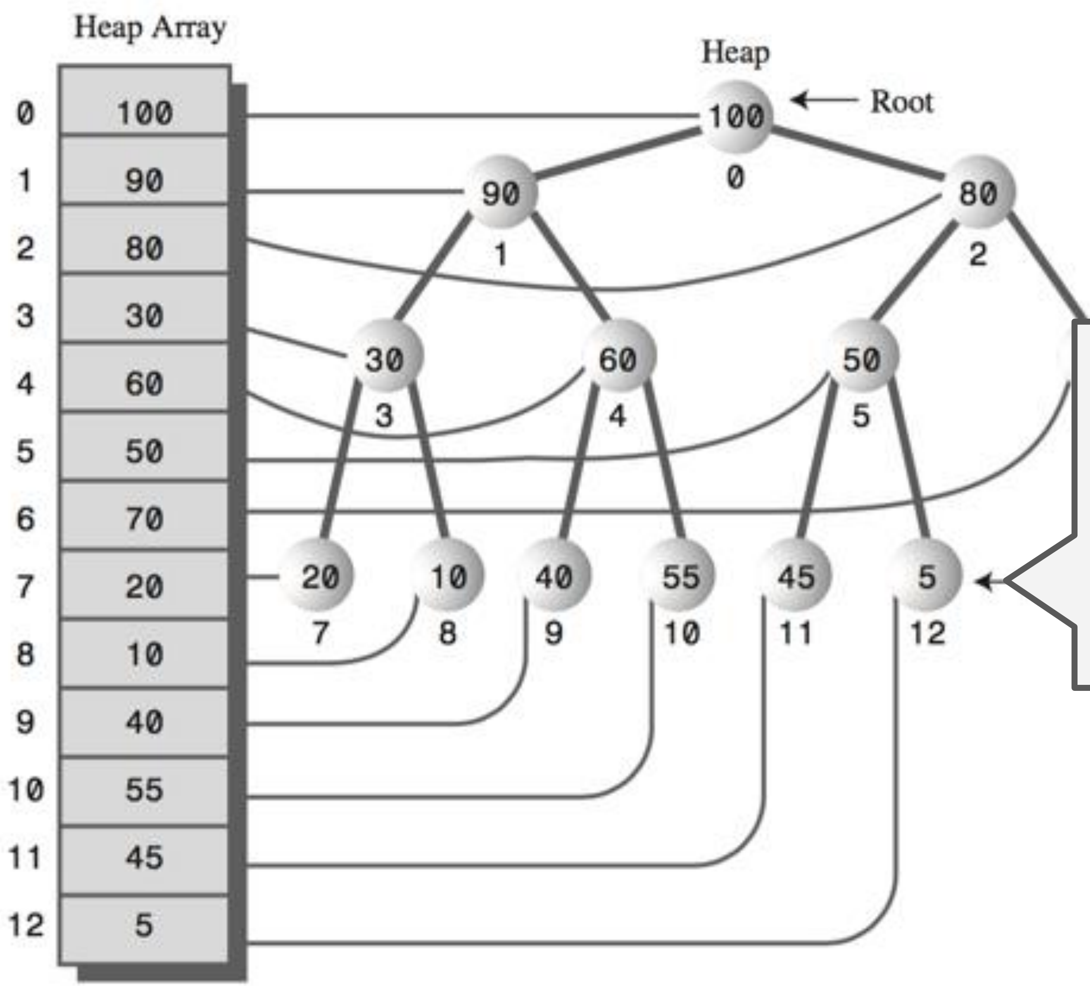


How do you find the parent of any node or the children of a node in a tree like this?

Implementation

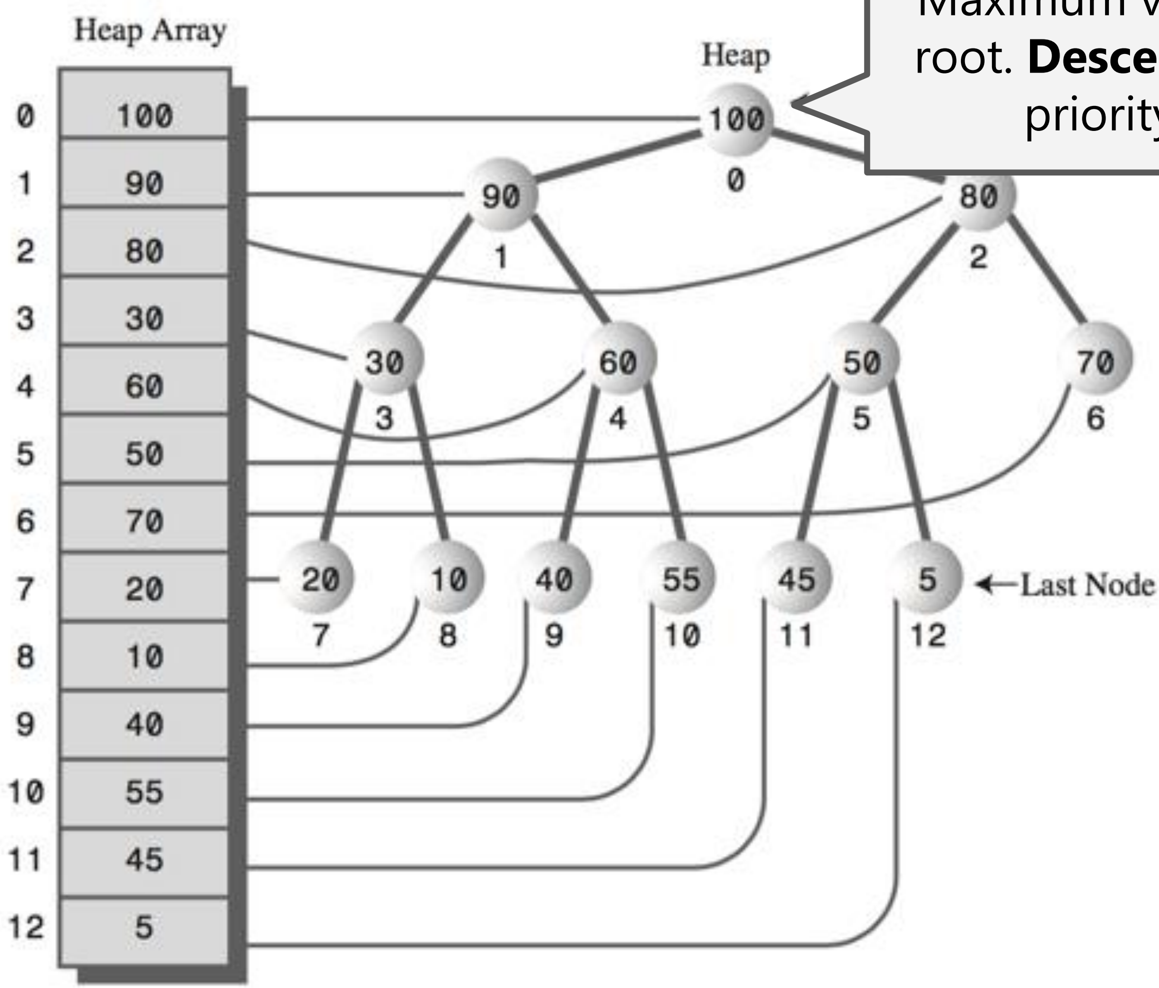


Implementation



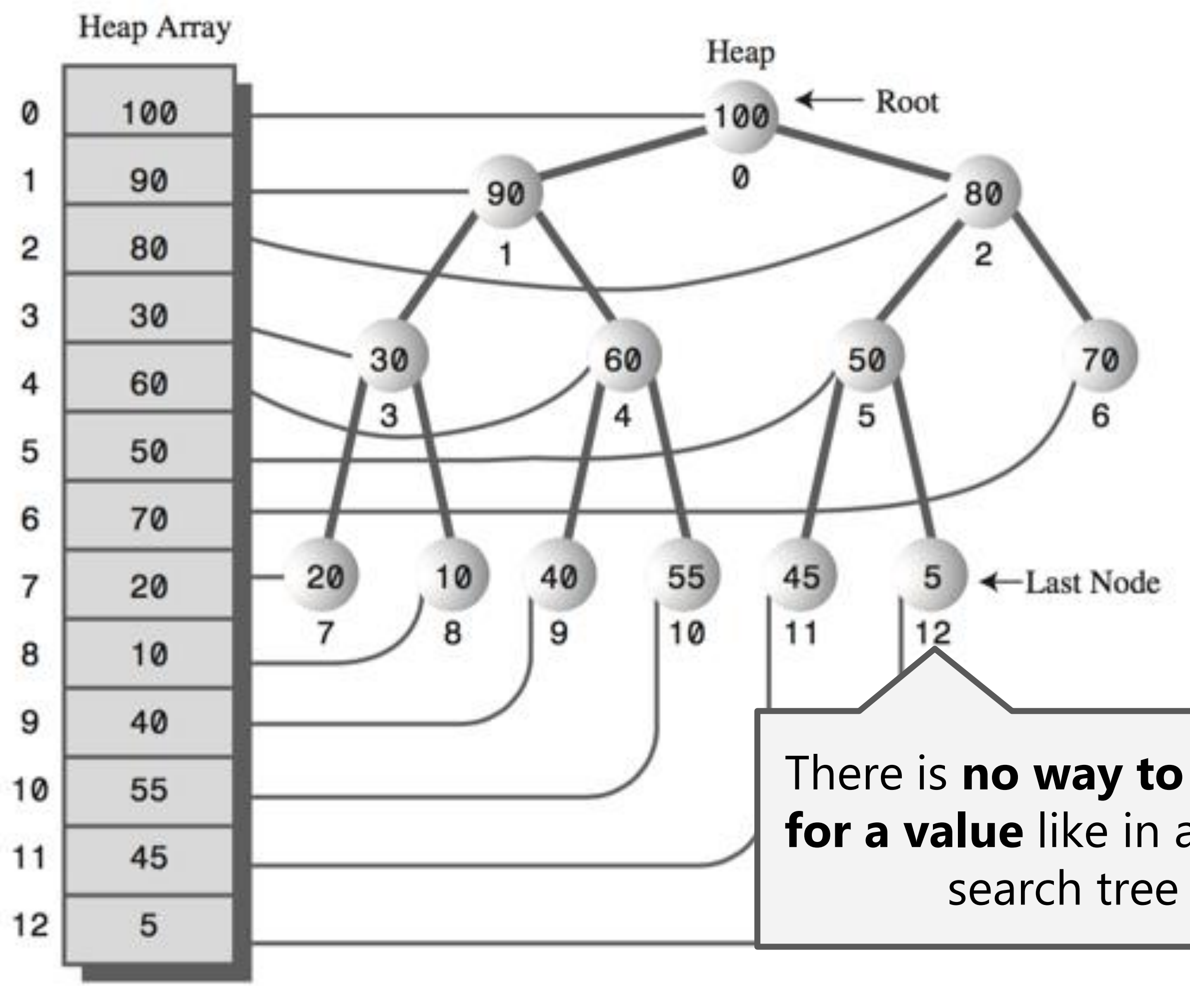
Left child of a node is **(k*2)+1** and right child is **(k*2)+2**

Implementation



Maximum value is in the root. **Descending-order** priority queue

Implementation



There is **no way to search for a value** like in a binary search tree

Removal operation

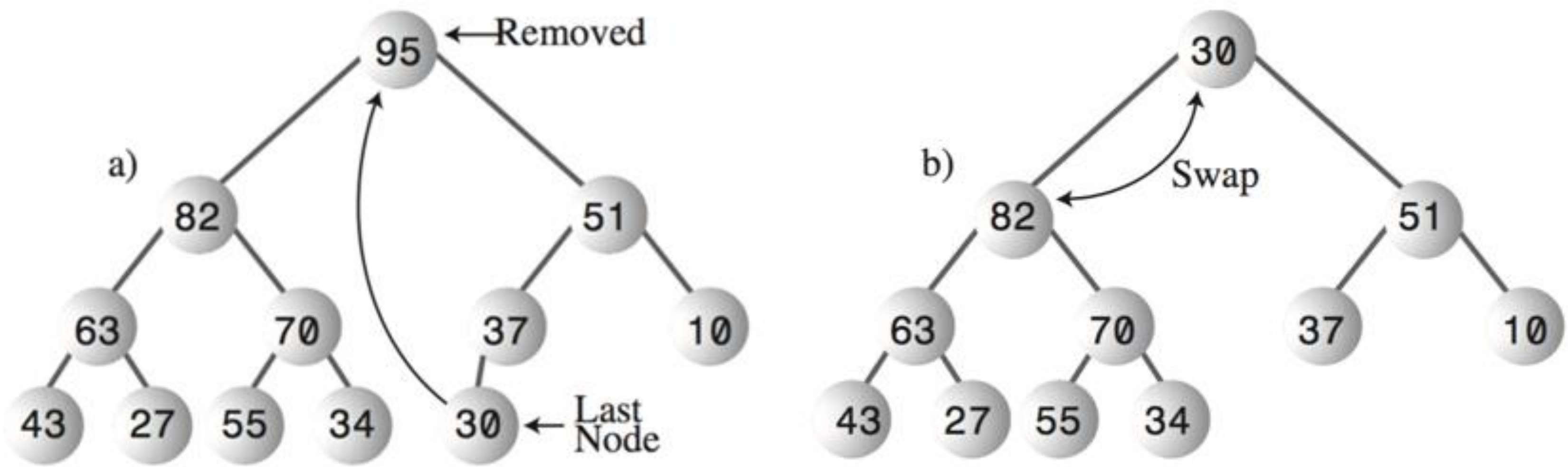
- Means removing the node with the maximum key.

- Maximum key is always the root, so is easy to find

- Removing the root leaves the tree incomplete
  - How to fix it?

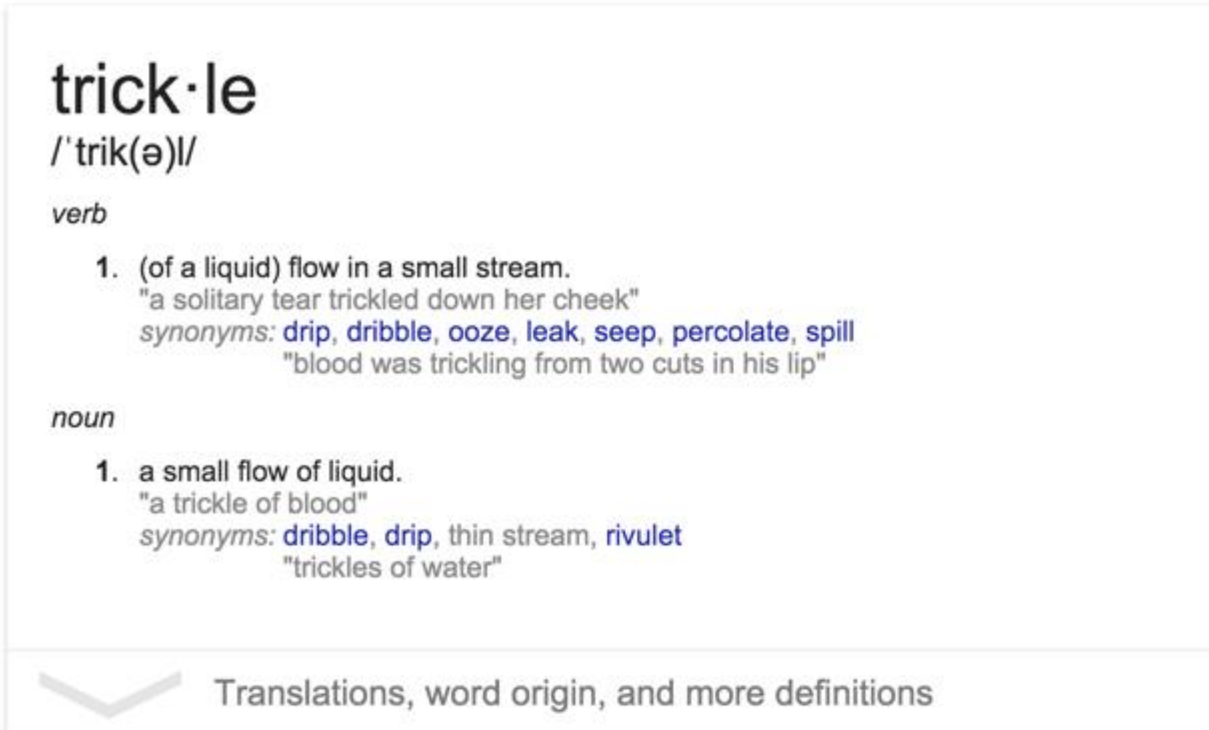
Removal operation

- Removal in summary:
  - Remove the root
  - Move the last node to the root
  - Trickle the last node down until it is below a larger node and above a smaller one



Removal operation

- Removal in sumr
  - Remove the roo
  - Move the last nc
  - Trickle the last r


trick·le

/ˈtrik(ə)l/

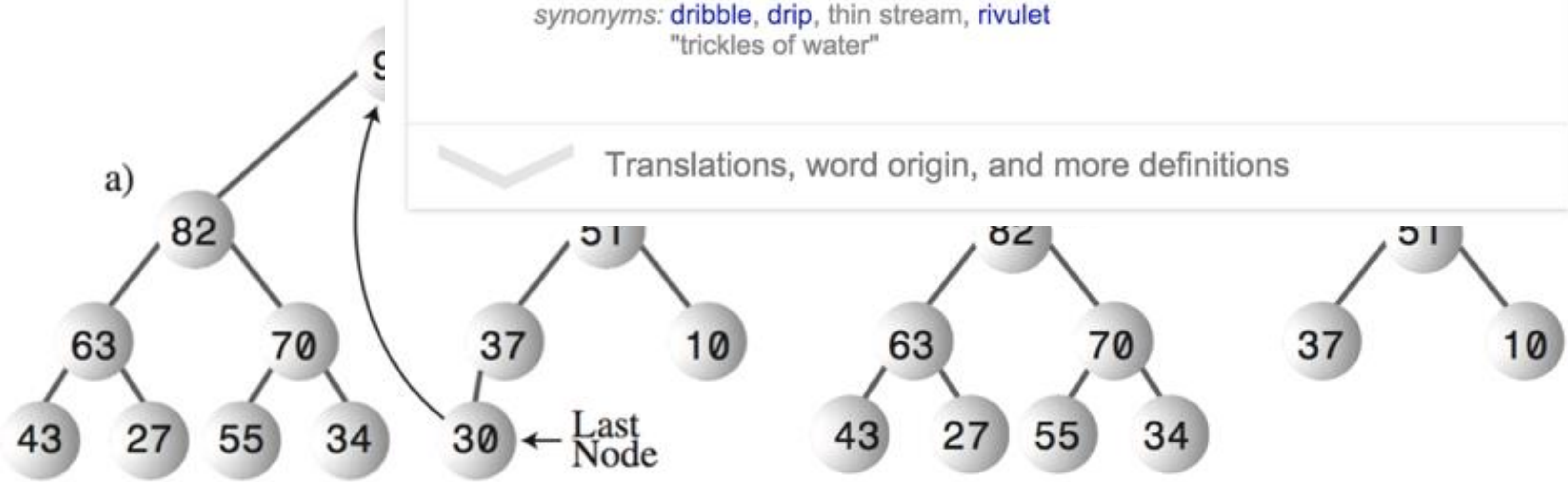
verb

1. (of a liquid) flow in a small stream.
   "a solitary tear trickled down her cheek"
   synonyms: drip, dribble, ooze, leak, seep, percolate, spill
   "blood was trickling from two cuts in his lip"

noun

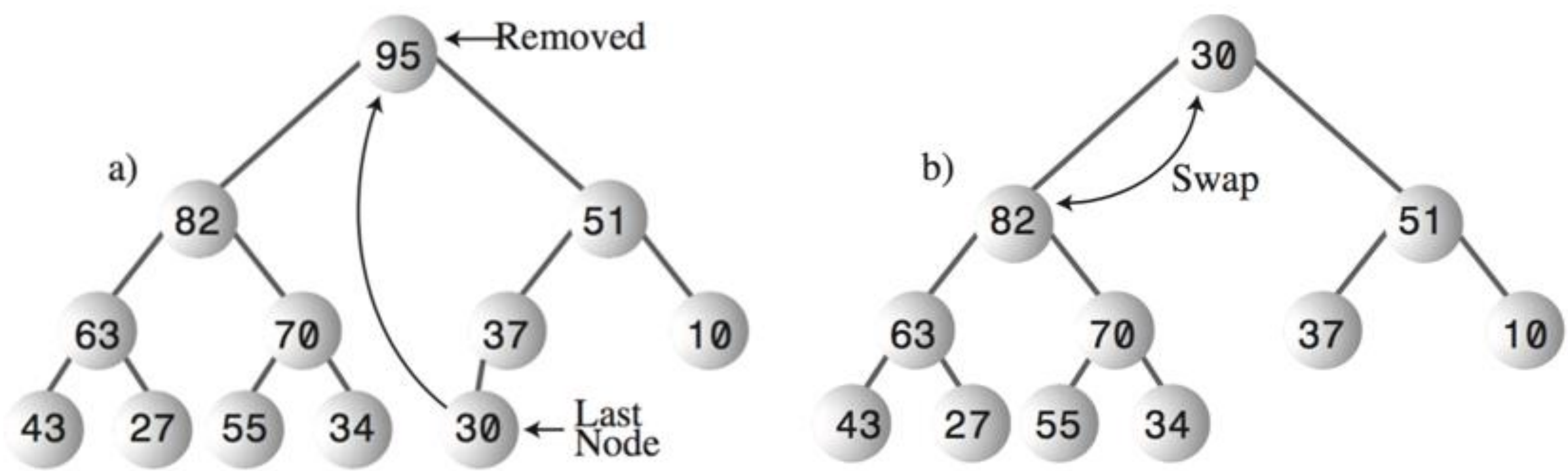
1. a small flow of liquid.
   "a trickle of blood"
   synonyms: dribble, drip, thin stream, rivulet
   "trickles of water"

Translations, word origin, and more definitions



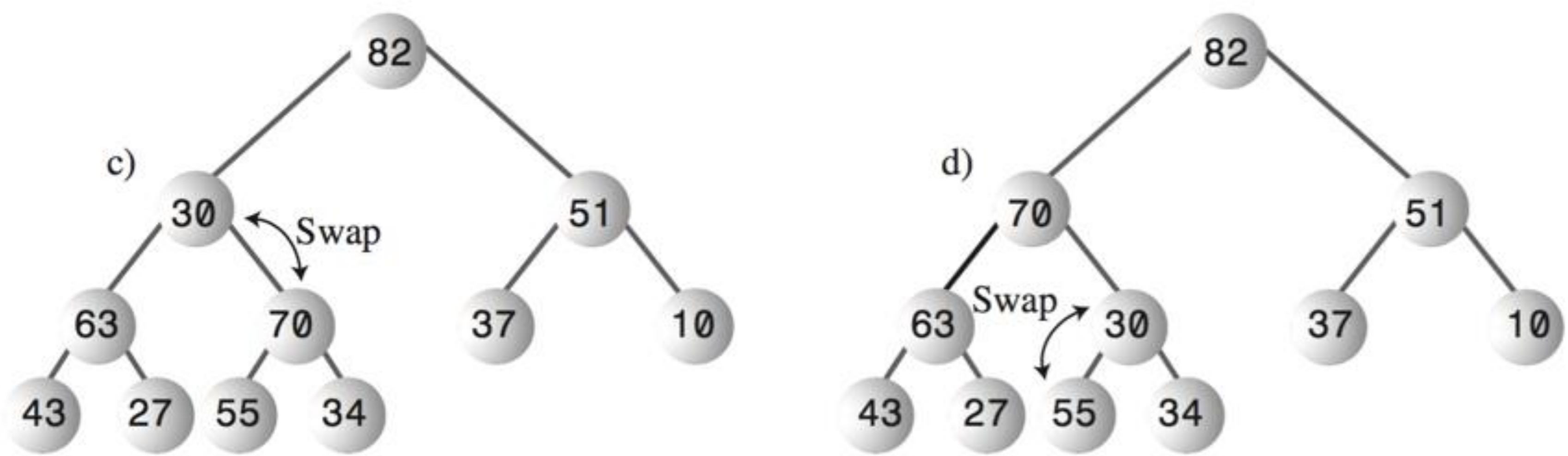
Removal operation

- Removal in summary:
  - Remove the root
  - Move the last node to the root
  - Trickle the last node down until it is below a larger node and above a smaller one



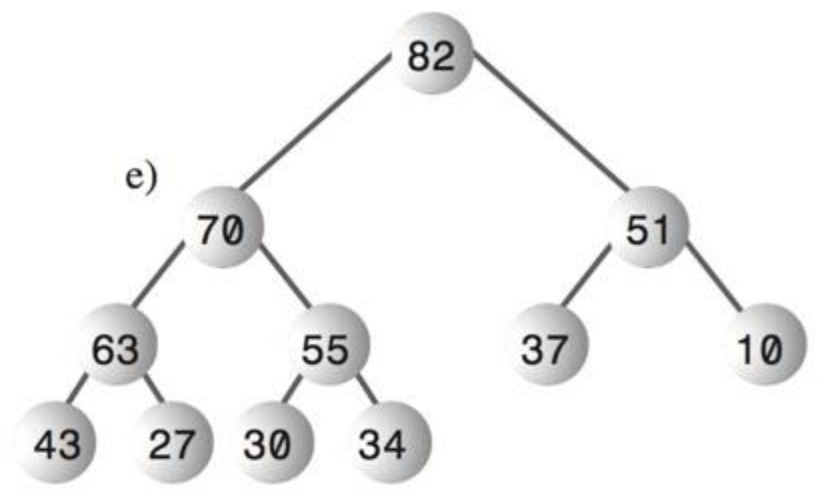
Removal operation

- Removal in summary:
  - Remove the root
  - Move the last node to the root
  - Trickle the last node down until it is below a larger node and above a smaller one



Removal operation

- Removal in summary:
  - Remove the root
  - Move the last node to the root
  - Trickle the last node down until it is below a larger node and above a smaller one



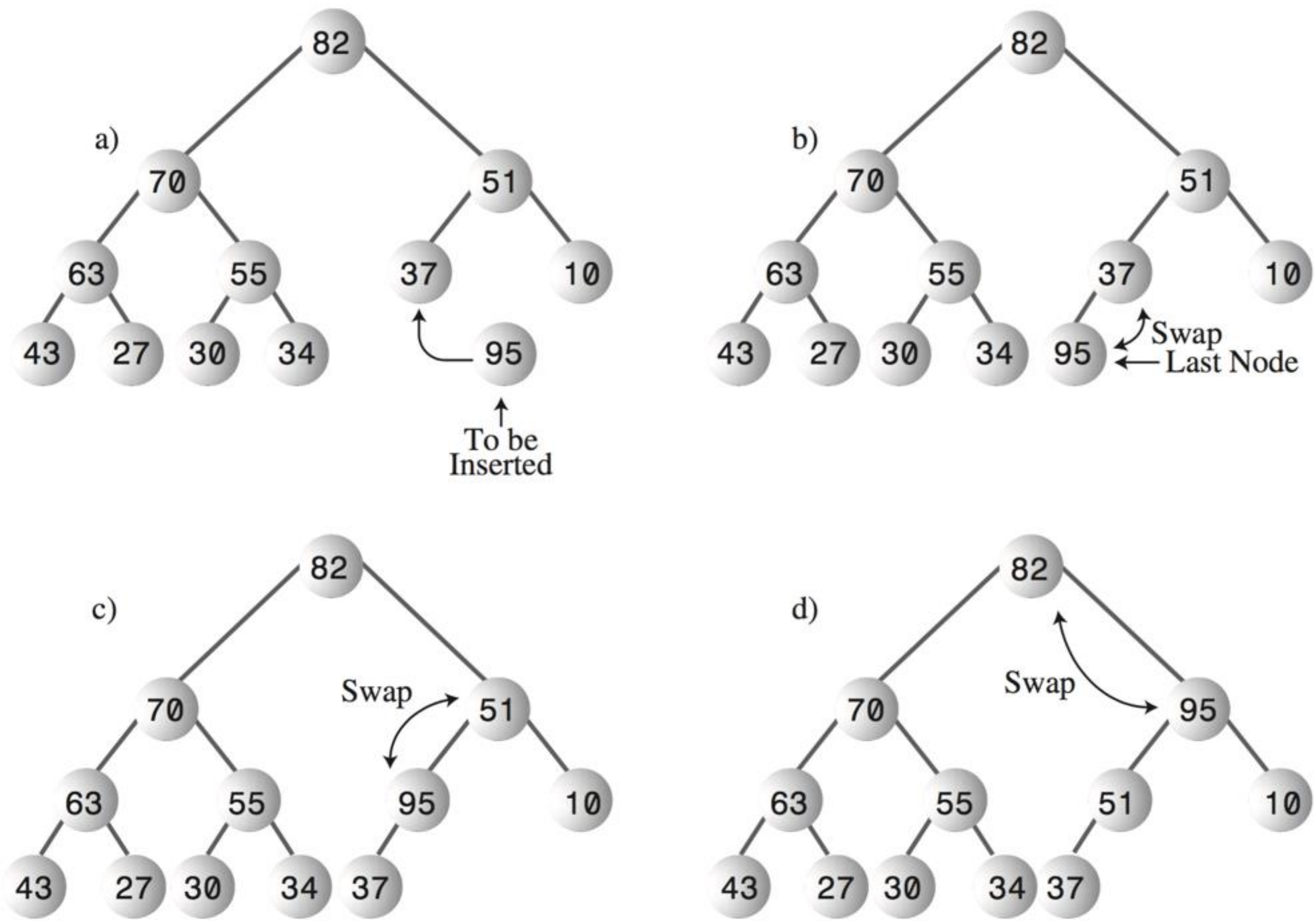
Insertion operation

- Uses **trickle** up instead of trickle down

- The new node is inserted in the first open position at the end of the array

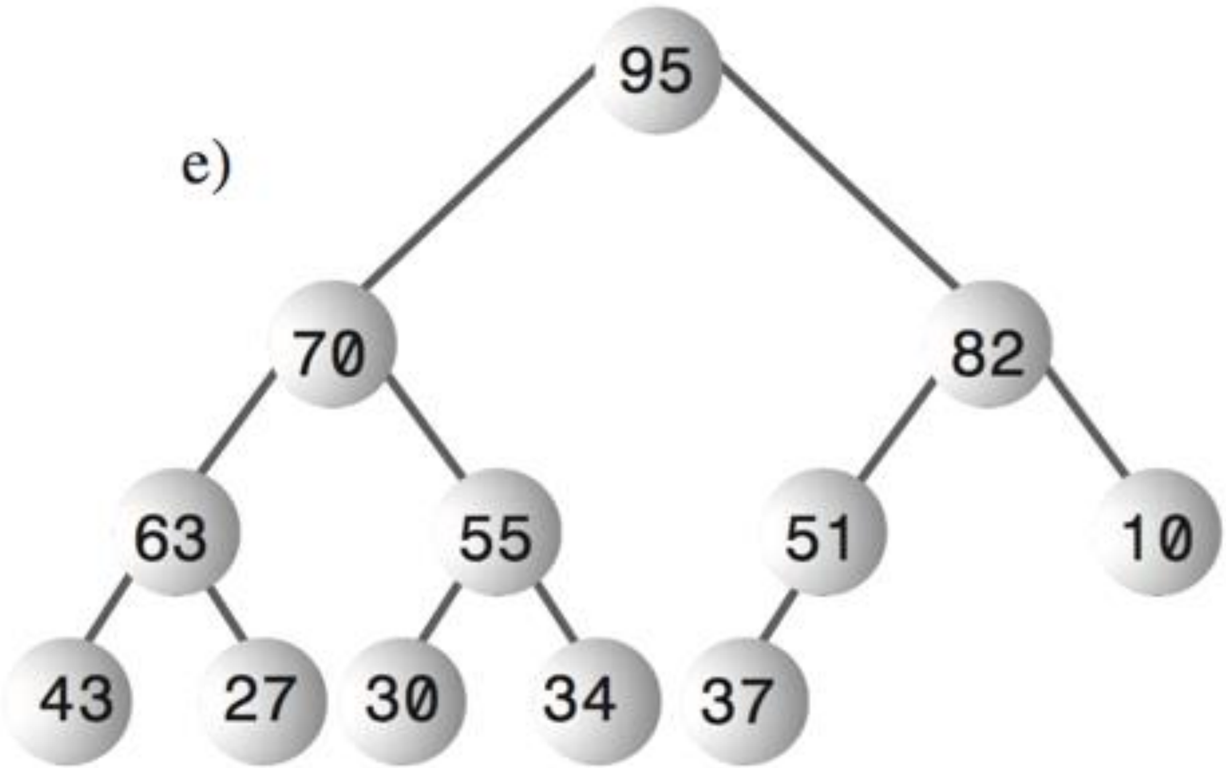
- The problem is that this is likely to violate the heap condition
    - Trickle up until it is below a node with a larger key and above a node with a smaller key

Insertion operation

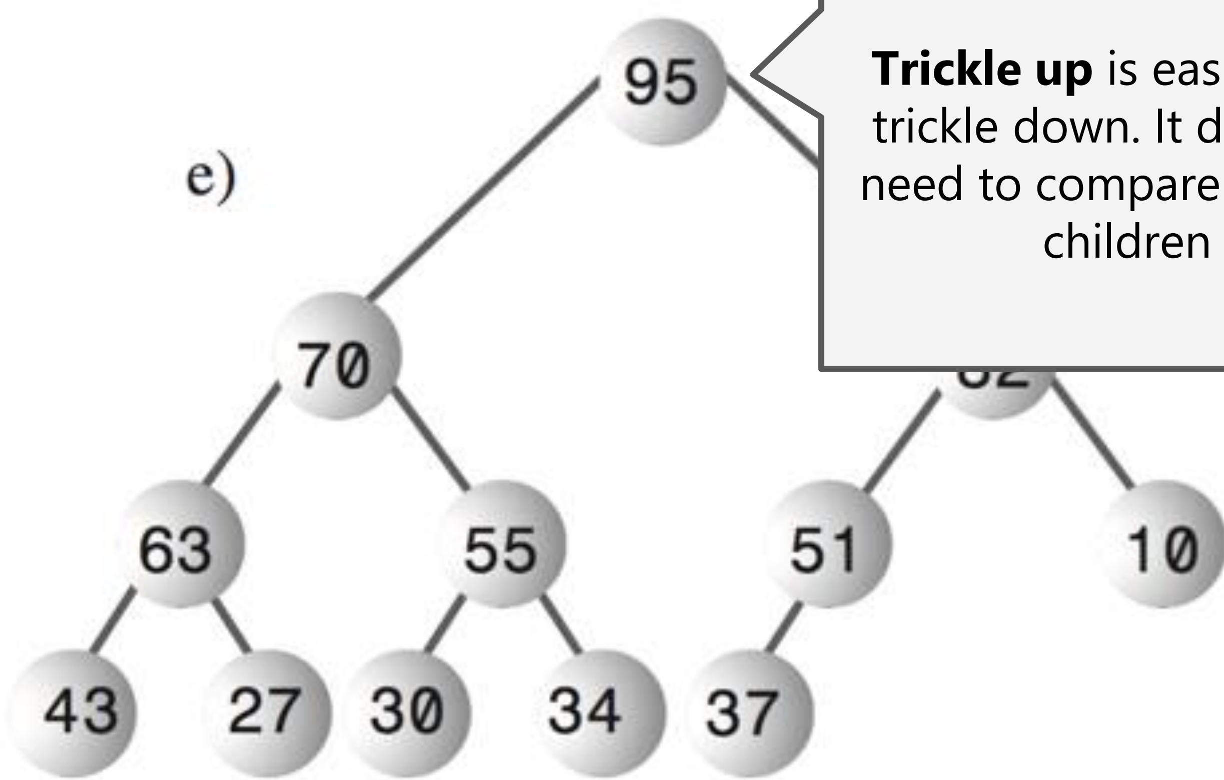


Insertion operation

e)

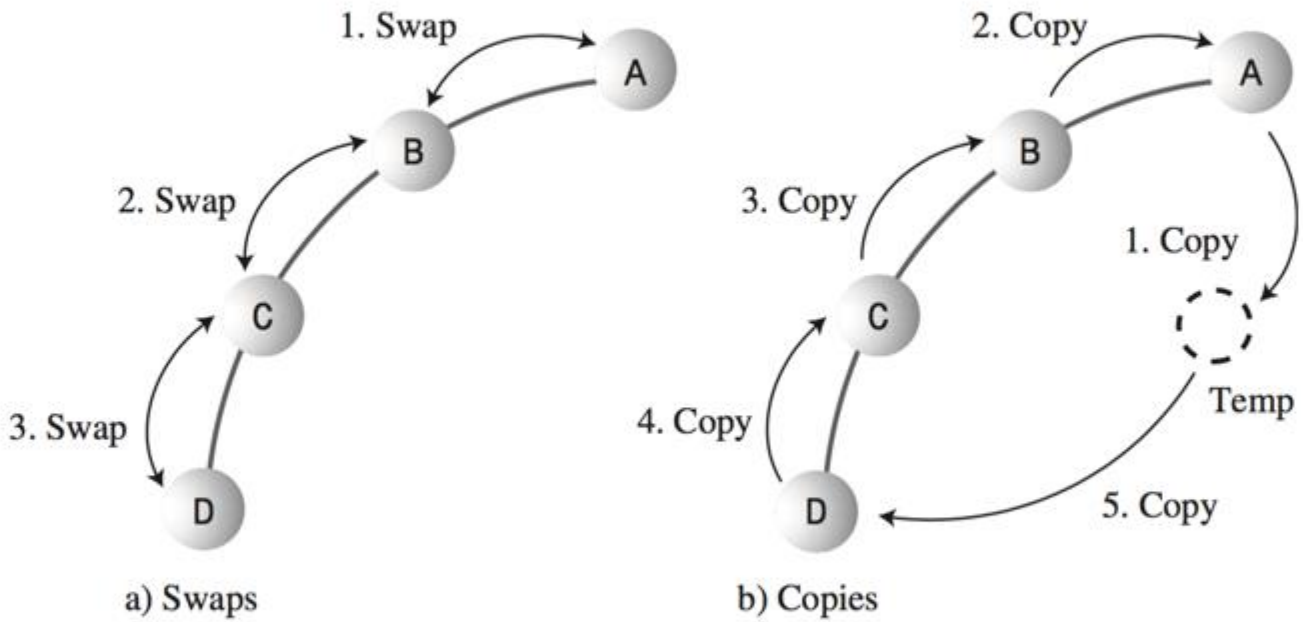


Insertion operation



Trickle up is easier than trickle down. It does not need to compare the two children

Swapping vs copying



Code implementation

```
public boolean insert(int key) {
  if (currentSize == maxSize) {
    return false;
  } else {
    Node newNode = new Node(key);
    heapArray[currentSize] = newNode;
    trickleUp(currentSize++);
    return;
  }
}
```

Code implementation

```
public void trickleUp(int index) {
  int parent = (index-1)/2;
  Node bottom = heapArray[index];

  while (index > 0 &&
         heapArray[parent].getKey() < bottom.getKey()) {
    heapArray[index] = heapArray[parent];
    index = parent;
    parent = (parent - 1) /2
  }
  heapArray[index] = bottom;
}
```

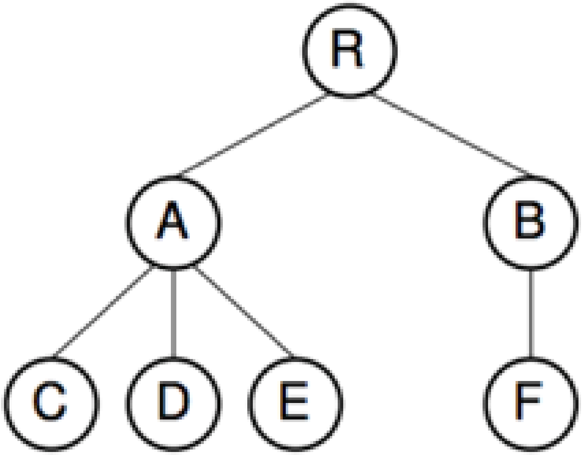
N-ary Trees

What is an n-ary tree?

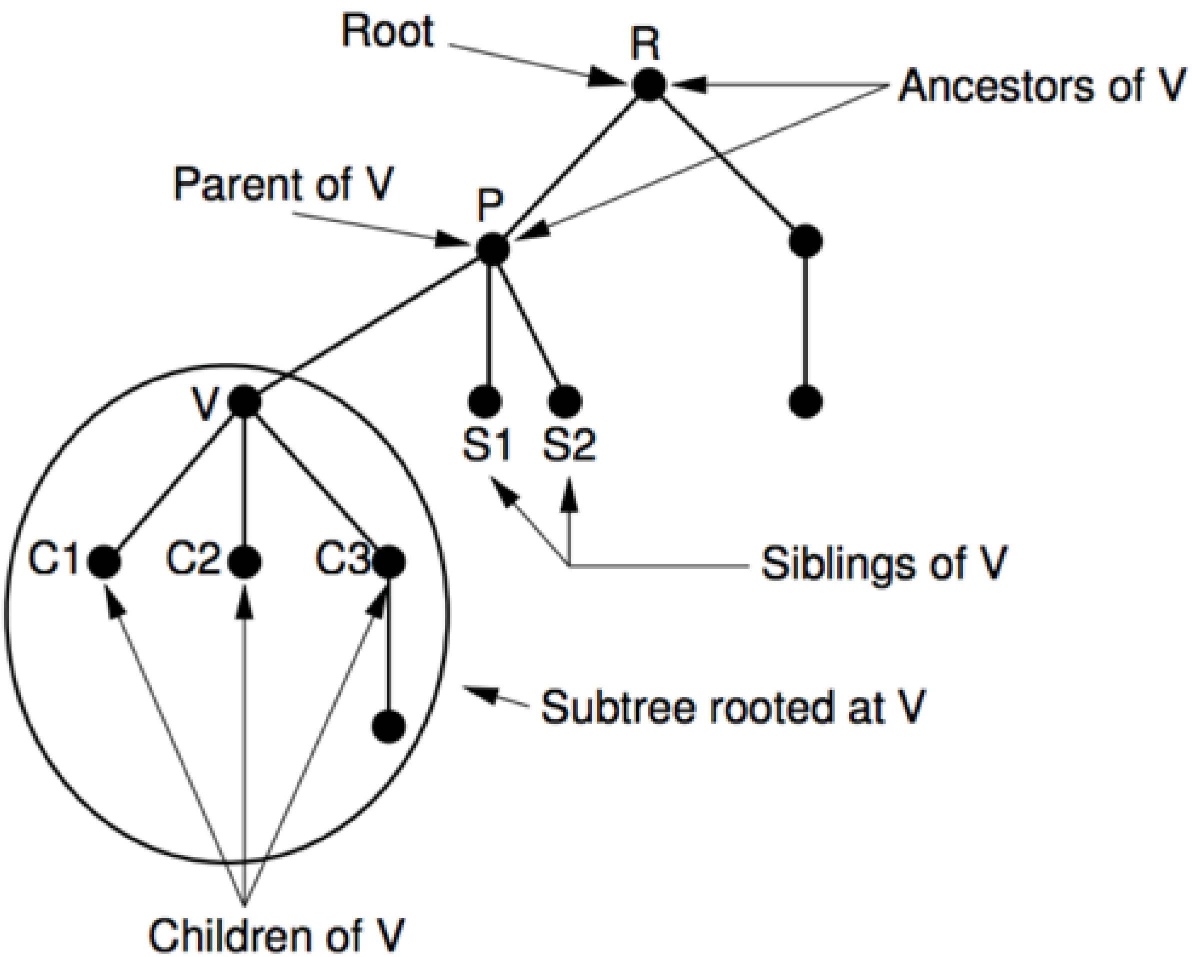
- As the name implies, in an n-ary tree, each node can have **an arbitrary number of children**

- Are definitely harder to implement than binary trees

- Also called non-binary trees, general trees or k-ary trees



What is an n-ary tree?



ADT of n-ary trees

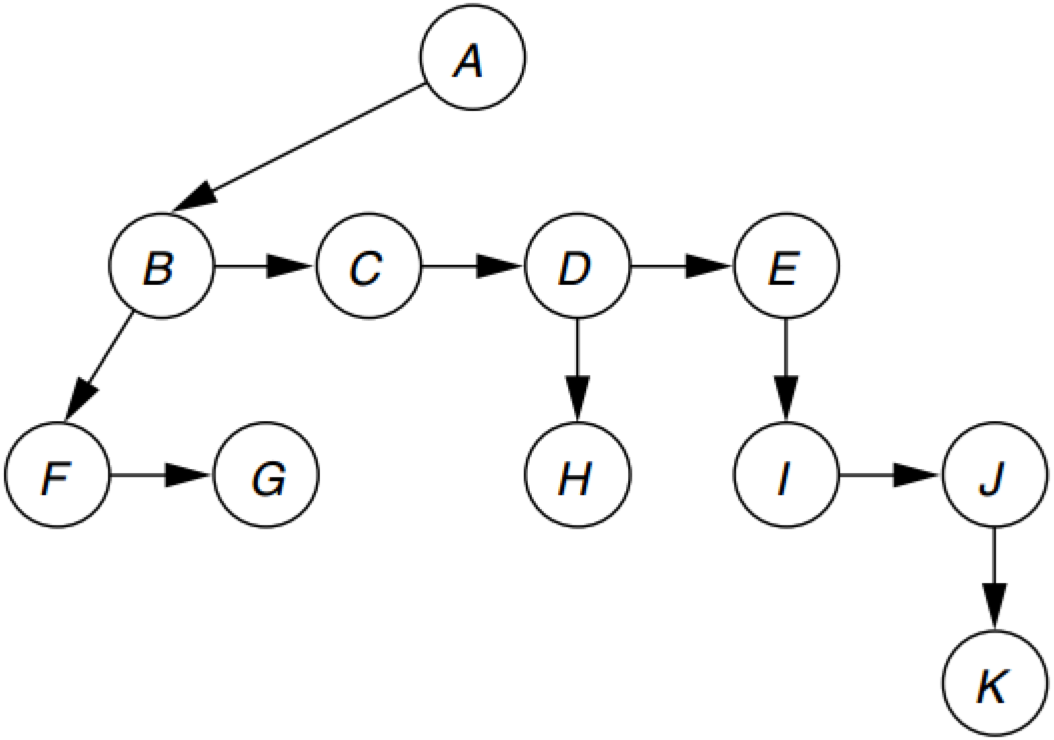
```
class GTNode<E> {
  E getValue();
  boolean isLeaf();
  GTNode getParent();
  GTNode leftMostChild();
  GTNode rightSibling();
  void setValue(E);
  void insertFirst(GTNode);
  void insertNext(GTNode);
  void removeFirst();
  void removeNext();
}
```

N-ary trees

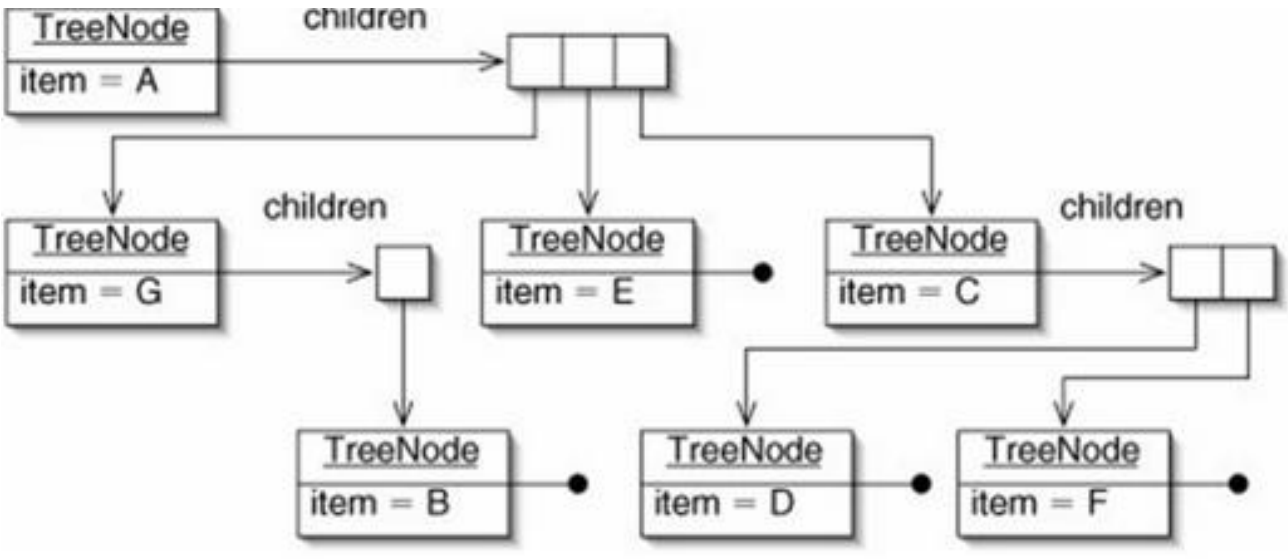
- How to access the children of a tree?
  - Not easy because there is no way to know in advance how many are.

- Provide access to the left most child and from that node, provide access to the next sibling to the right
  - Traverse the children as a list of elements
  - This method is called *First child/next sibling method*

N-ary trees



N-ary trees


TreeNode
item = A
children
TreeNode
item = G
children
TreeNode
item = E
TreeNode
item = C
children
TreeNode
item = B
TreeNode
item = D
TreeNode
item = F

B-trees

Introduction

- The trees we have seen so far assume that the entire data structures is in the main memory of the machine

- When the data grows, it has to be kept in the disk (at least part of it)

Introduction

- Now we are dealing with the latency associated with the disk

- The data structures must be optimized to reduce the disk operations or make them as fast as possible

- The most popular structure for disk bound searching is the B-Tree

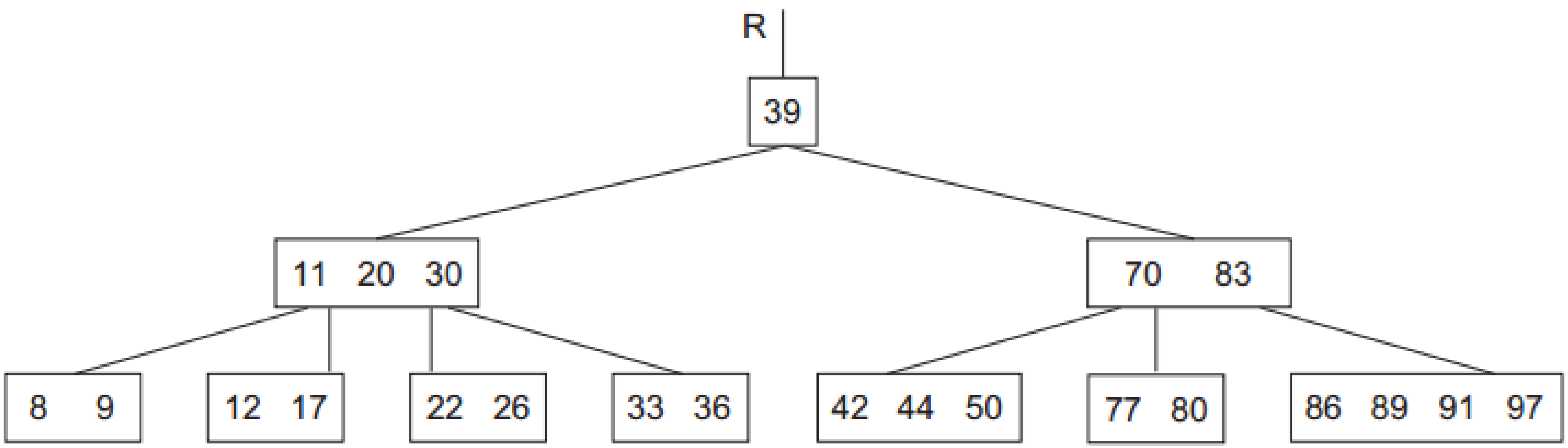
What is a B-Tree?

- A B-tree of order *m*, is a *m*-ary tree with the following properties:
  - All the leaves are at the same level
  - All internal pages, except the root, have a maximum of *m* branches (non empty) and a minimum of *m*/2 branches
  - Key count in each internal page is one minus the branch count. Keys divide the branches in a similar way as a search tree
  - Root has a maximum of m branches but it can have less than 2

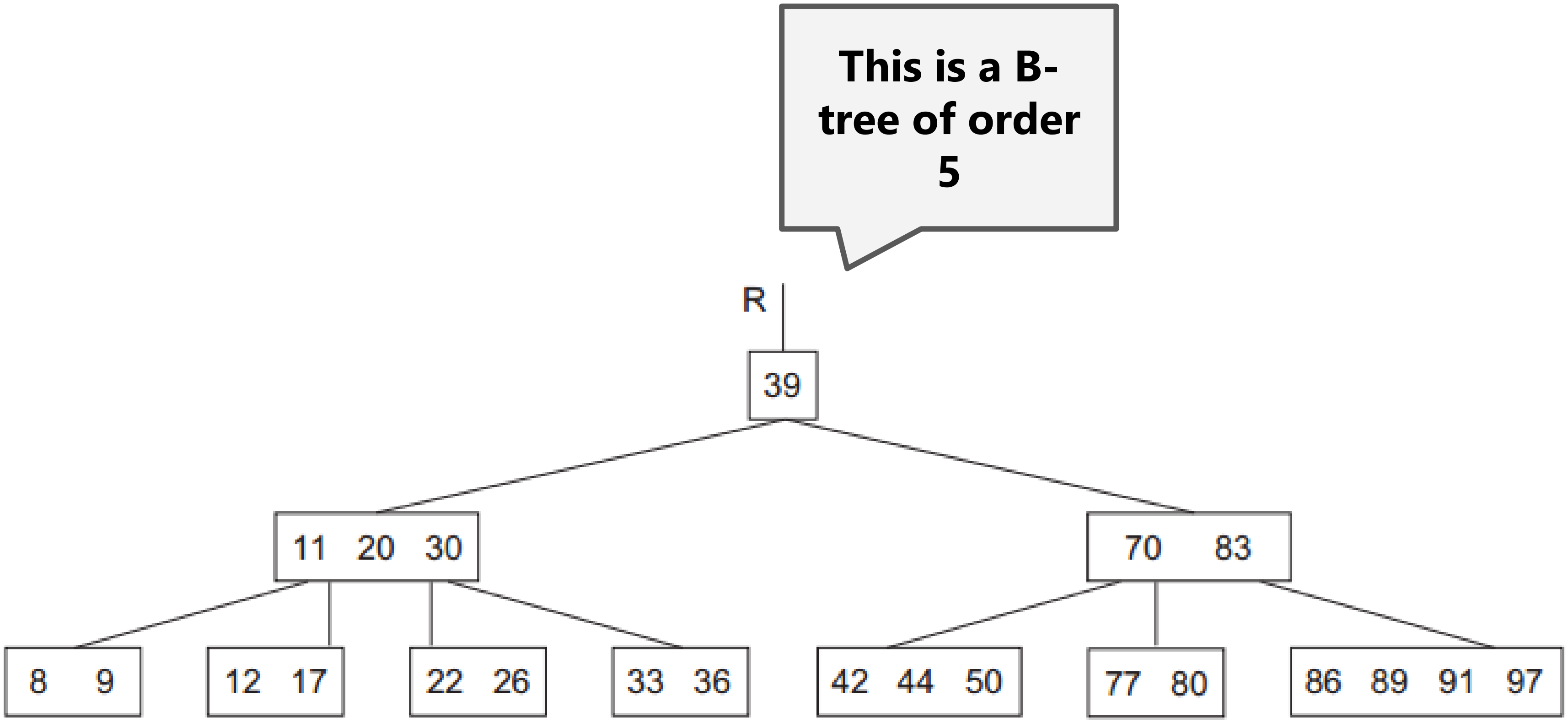
What is a B-Tree?

- A B-tree is always perfectly balanced
- Nodes are usually called pages
- The purpose of a B-Tree is to reduce tree depth, reducing the disk access

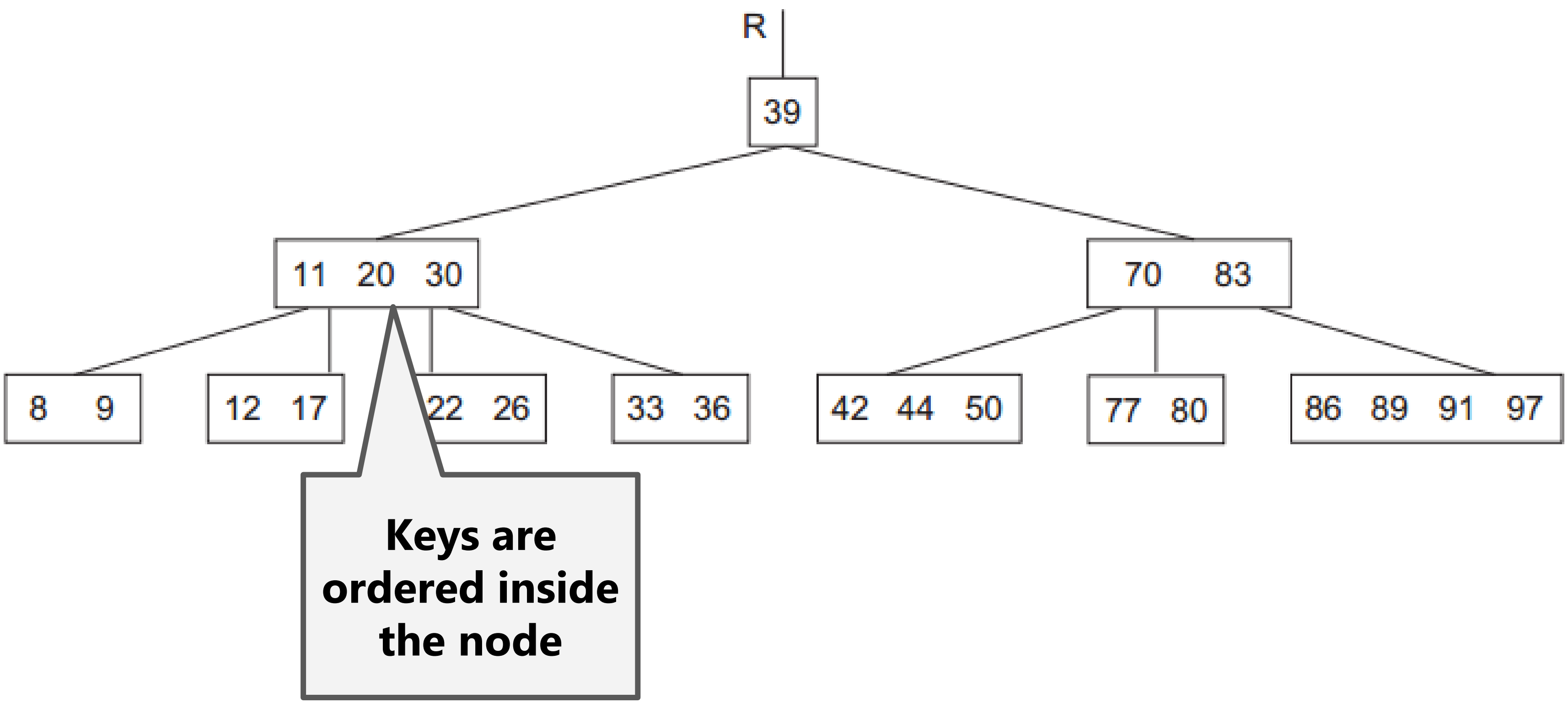
What is a B-Tree?



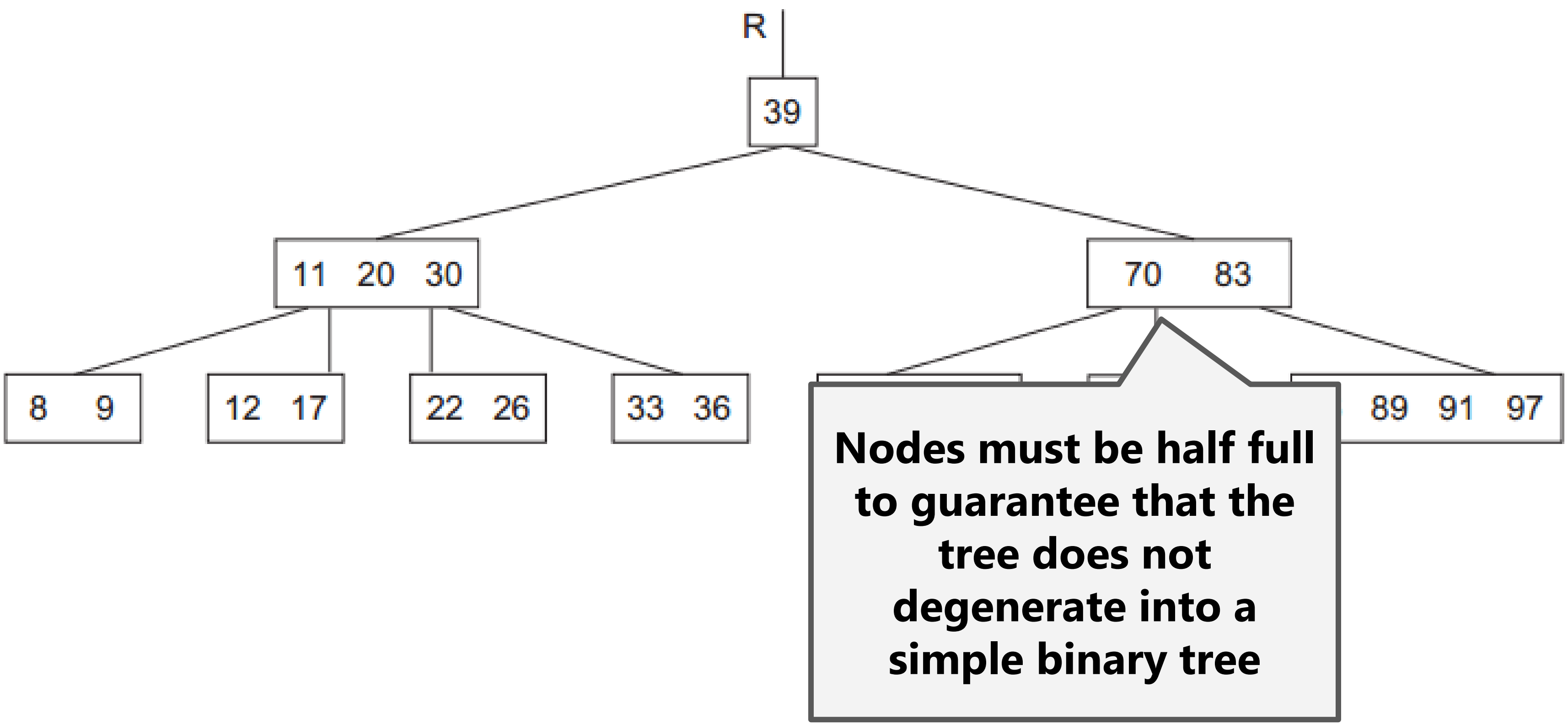
What is a B-Tree?



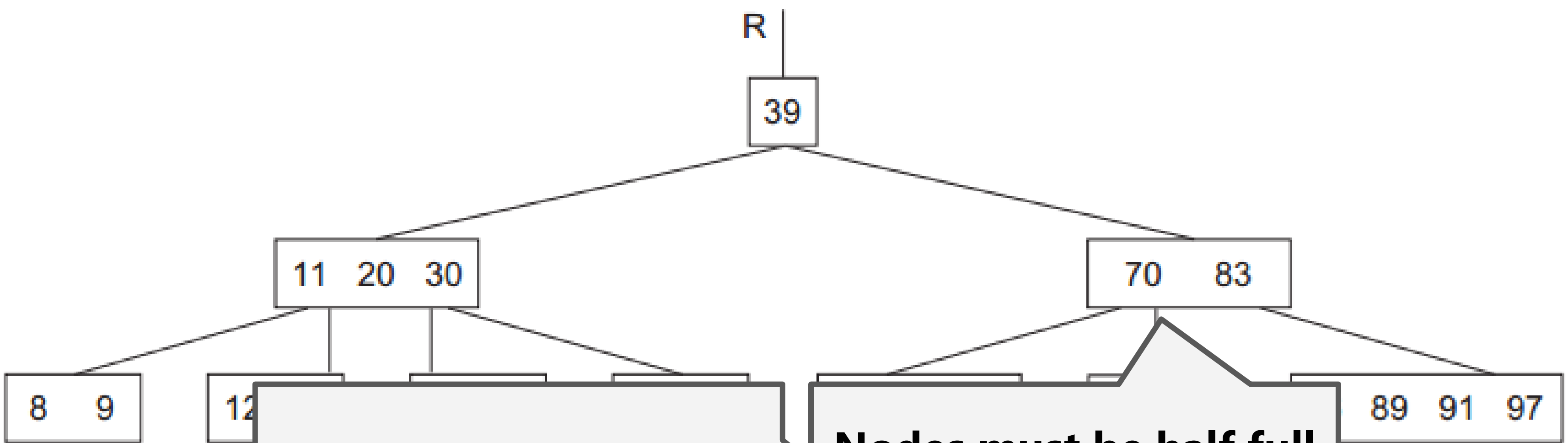
What is a B-Tree?



What is a B-Tree?



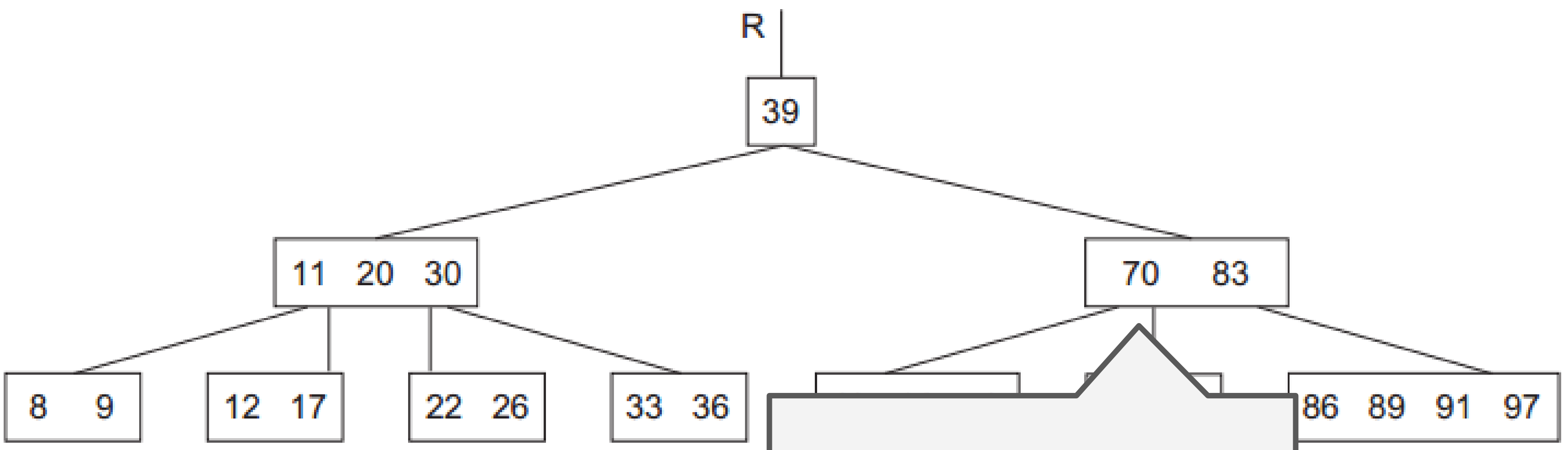
What is a B-Tree?



Half full means: *at least M/2 to M children*

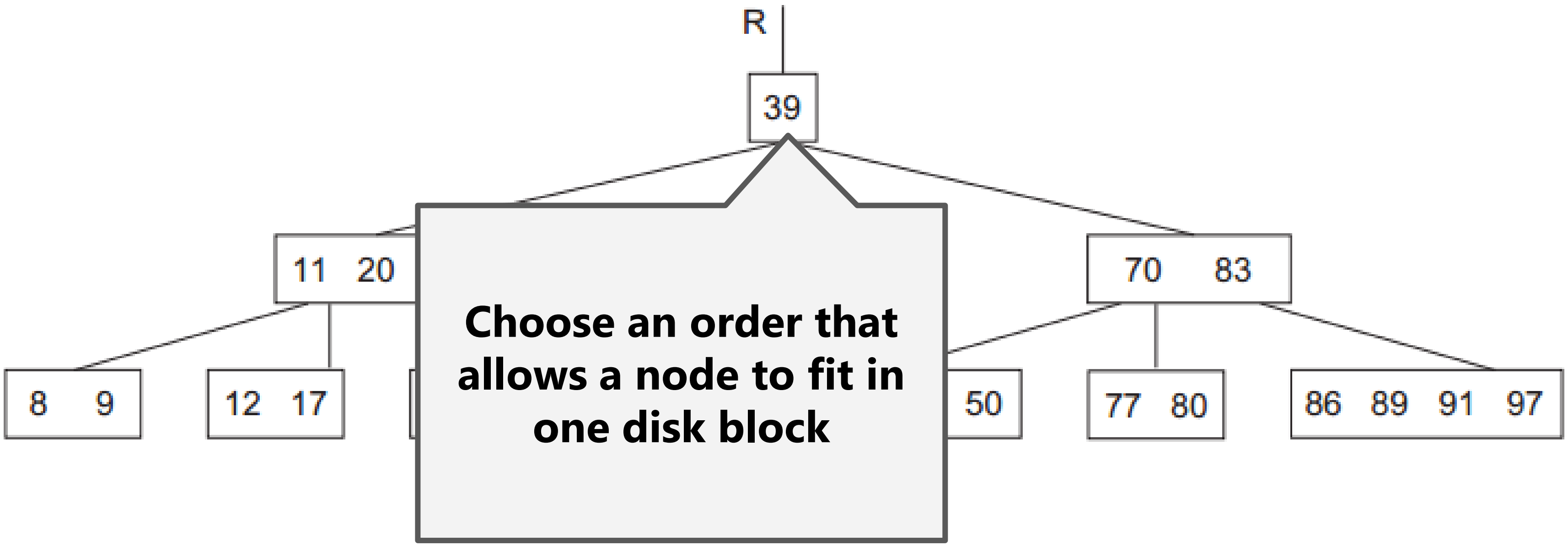
Nodes must be half full to guarantee that the tree does not degenerate into a simple binary tree

What is a B-Tree?

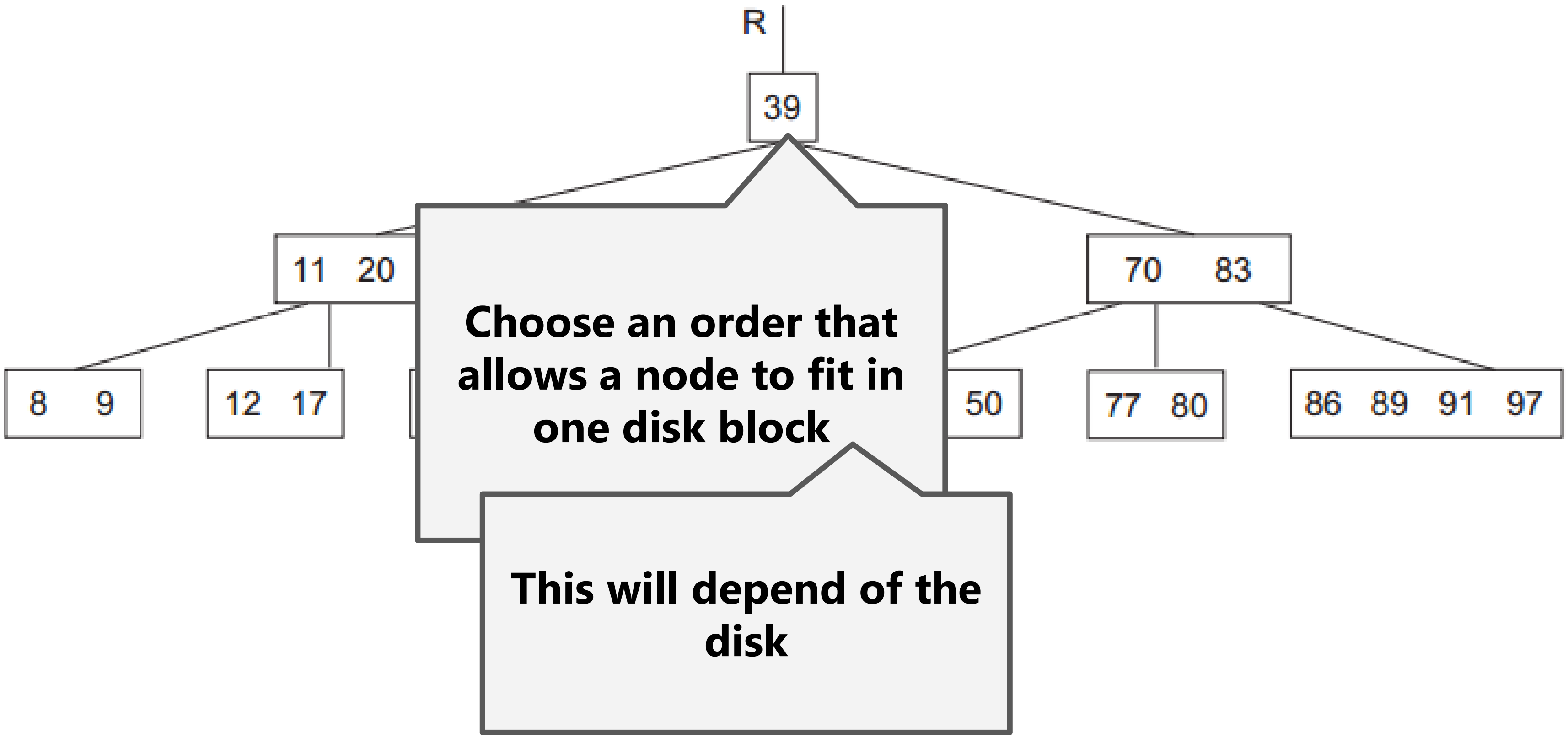


How to choose the order of a B-Tree?

What is a B-Tree?



What is a B-Tree?



B-Trees

How to represent a page?

```
class Page<E> {
  List<E> keys;
  List<Page<E>> branches;
  int count;
  int m;
}
```

B-Trees
Insert operation

- Keep in mind that B-Trees grow upwards, from the root.

- In summary, the insert involves:
    - Search the key to insert in the tree. There will be a search path determined by the keys in the pages
    - If the key is not in the in tree, search path will end in a leave.
    - If the leaf is not full, the insertion is possible
    - If the leaf is full, split starts to happen.

B-Trees
Insert operation

- Splitting a node involves:
    - Separate the node in halves in the same level.
    - Mid key goes upward in the search path repeating the insertion process in the in the precedent node.
    - The ascension of the mid key can propagate up to the root

B-Trees

Insert operation

- Let’s insert the following keys to a B-Tree:

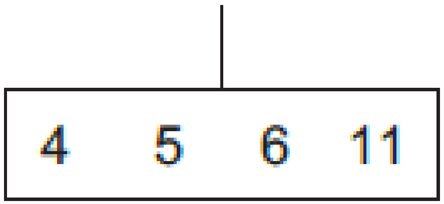
  6 11 5 4 8 9 12 21 14 10 19 28 3 17 32 15 16 26 27

B-Trees

Insert operation

- Let's insert the following keys to a B-Tree:

  6 11 5 4 8 9 12 21 14 10 19 28 3 17 32 15 16 26 27

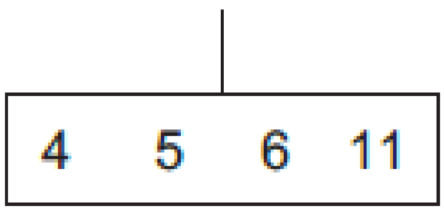


B-Trees

Insert operation

- Let’s insert the following keys to a B-Tree:

  6 11 5 4 8 9 12 21 14 10 19 28 3 17 32 15 16 26 27

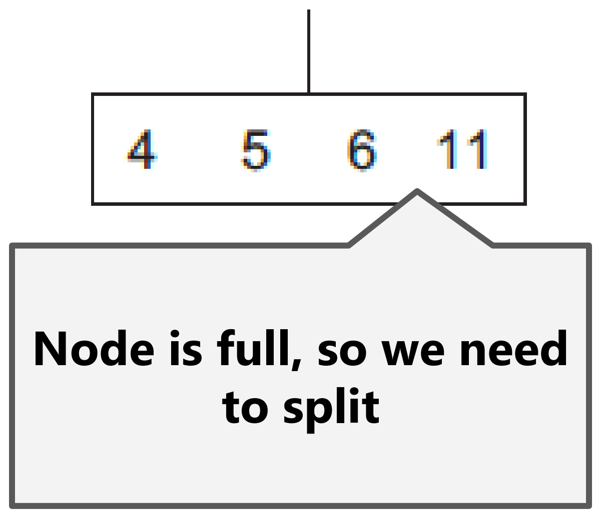


B-Trees

Insert operation

- Let’s insert the following keys to a B-Tree:

6 11 5 4 8 9 12 21 14 10 19 28 3 17 32 15 16 26 27

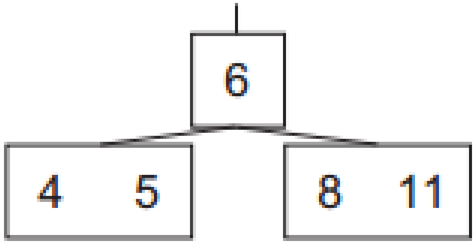


B-Trees

Insert operation

- Let's insert the following keys to a B-Tree:

  6 11 5 4 8 9 12 21 14 10 19 28 3 17 32 15 16 26 27

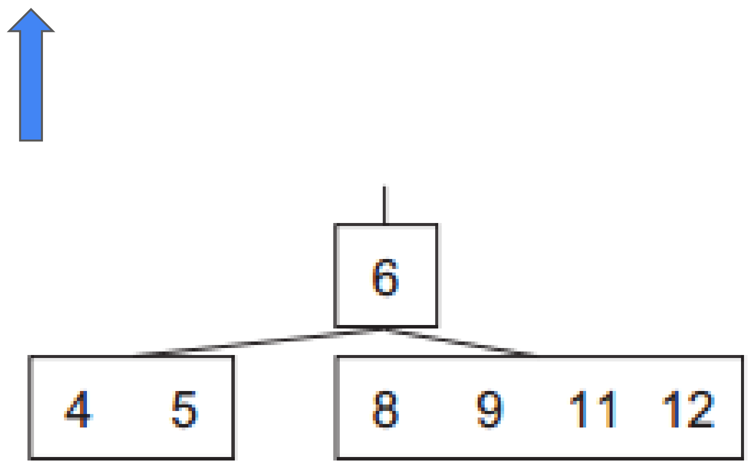


B-Trees

Insert operation

- Let's insert the following keys to a B-Tree:

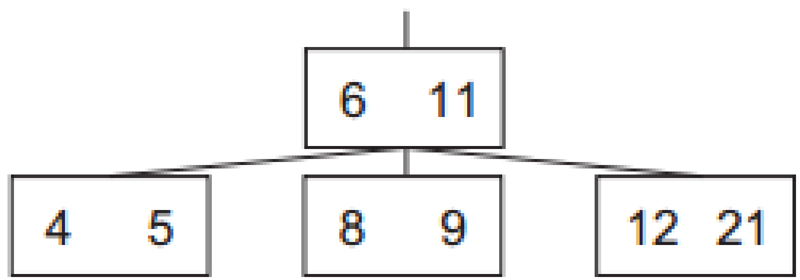
6 11 5 4 8 9 12 21 14 10 19 28 3 17 32 15 16 26 27



B-Trees
Insert operation

- Let’s insert the following keys to a B-Tree:

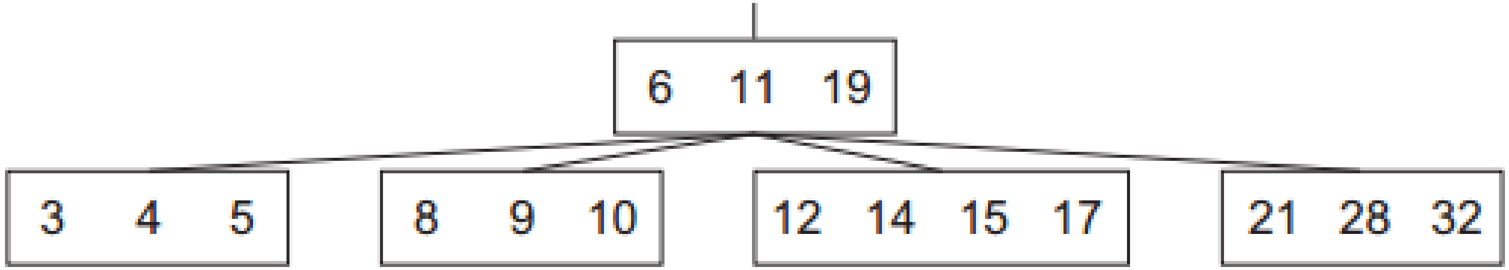
  6 11 5 4 8 9 12 21 14 10 19 28 3 17 32 15 16 26 27



B-Trees
Insert operation

- Let’s insert the following keys to a B-Tree:

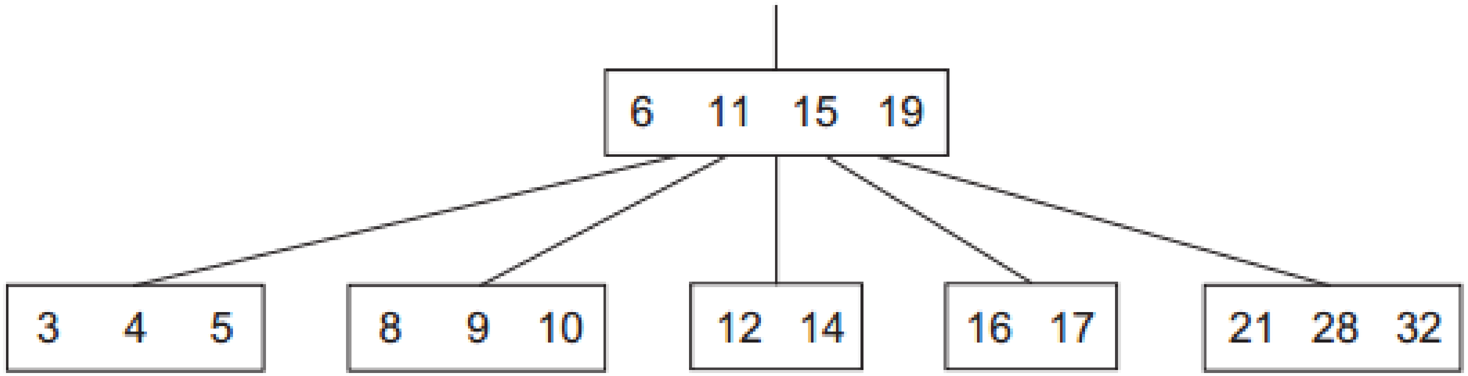
  6 11 5 4 8 9 12 21 14 10 19 28 3 17 32 15 16 26 27



B-Trees
Insert operation

- Let's insert the following keys to a B-Tree:

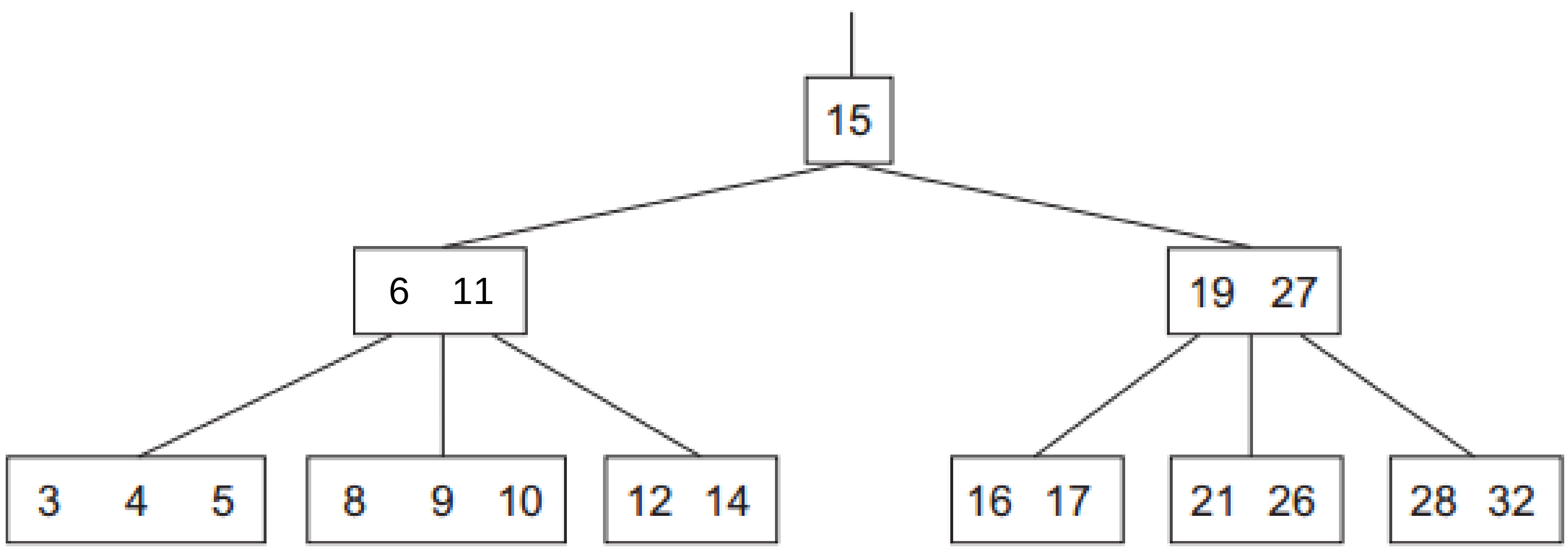
6 11 5 4 8 9 12 21 14 10 19 28 3 17 32 15 16 26 27



B-Trees
Insert operation

- Let’s insert the following keys to a B-Tree:

  6 11 5 4 8 9 12 21 14 10 19 28 3 17 32 15 16 26 27



B-Trees
Delete operation

- Deletion is analogous to insertion but a little more complicated

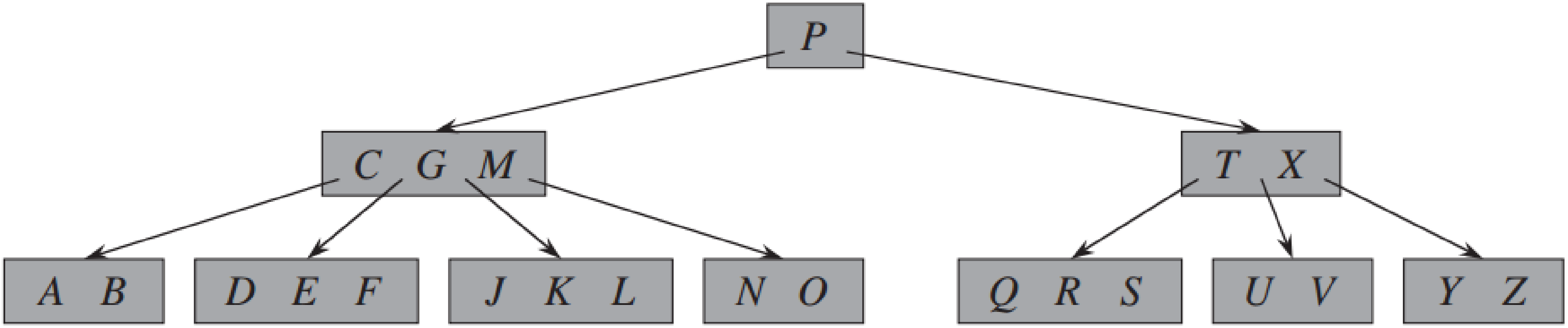
- Must ensure that a node don’t get too small during deletion (only root is allowed to have less than *m* / 2 keys

- There are 3 cases for deleting a key

B-Trees

Delete operation: case #1

- Key *k* is in a node *x* which is a leaf and it have more than (*m* / 2) -1 keys
  - Let’s delete *F*

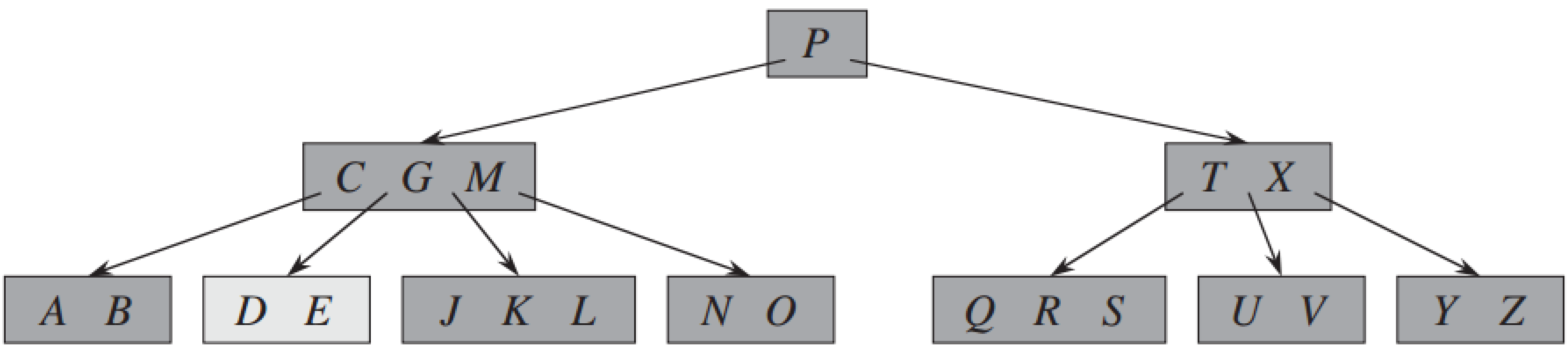


***m* = 6**

B-Trees

Delete operation: case #1

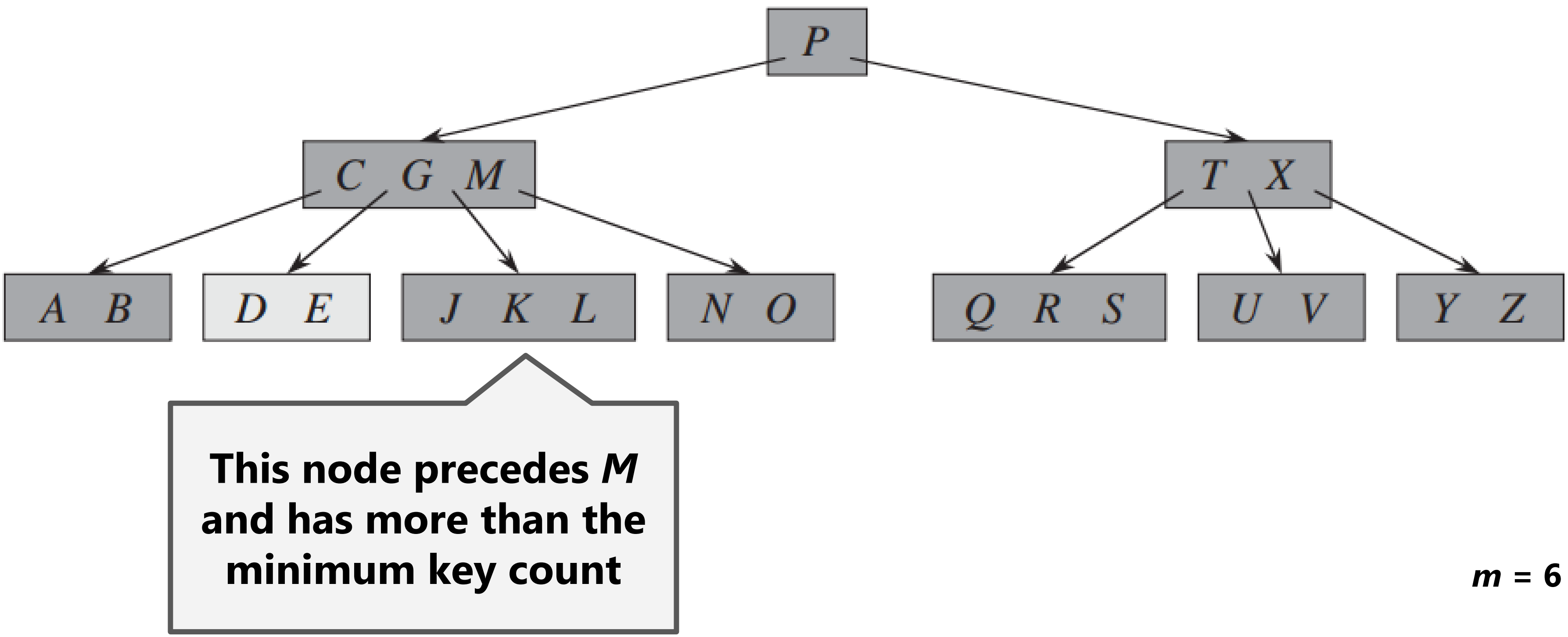
- Key *k* is in a node *x* which is a leaf and it have more than ROUND(*m* / 2) -1 keys
  - Let’s delete *F*



***m* = 6**

B-Trees
Delete operation: case #2a

- Key *k* is in a node *x* which is not a leaf
  - Let's delete *M*

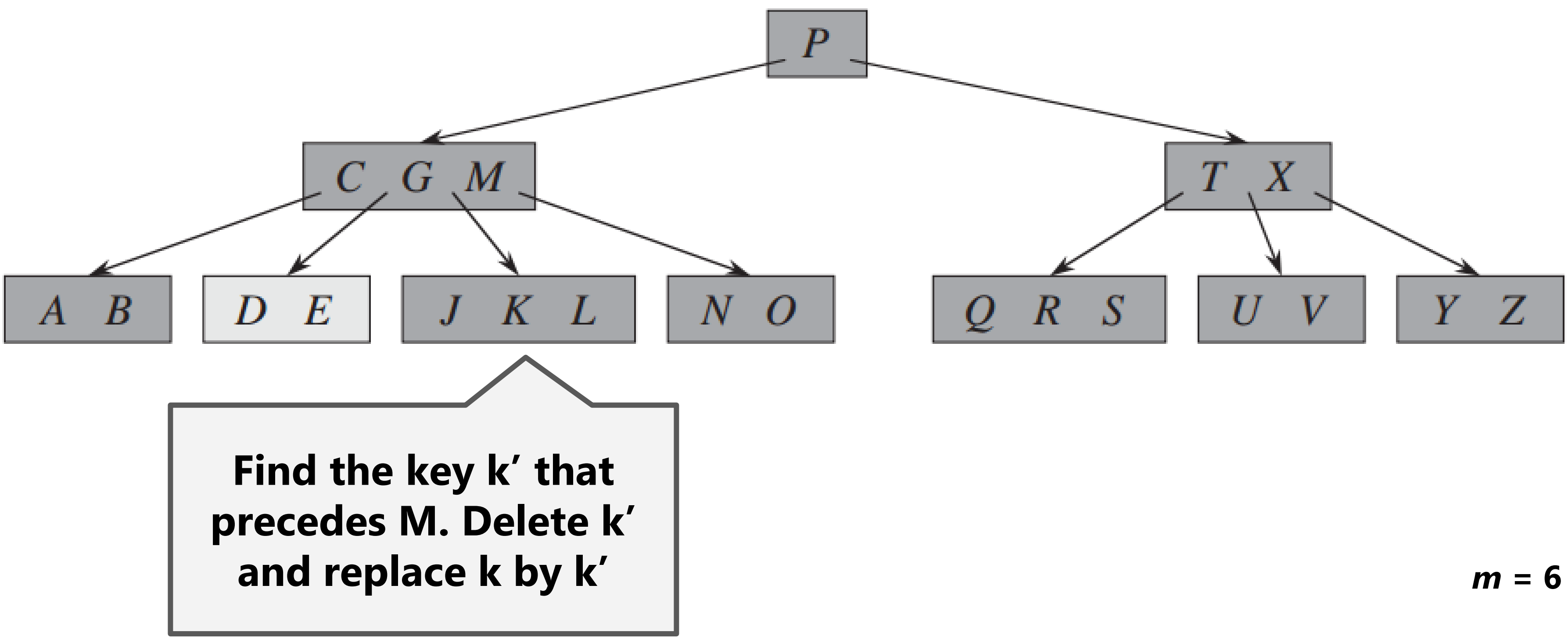


***m* = 6**

B-Trees

Delete operation: case #2a

- Key *k* is in a node *x* which is not a leaf
  - Let's delete *M*

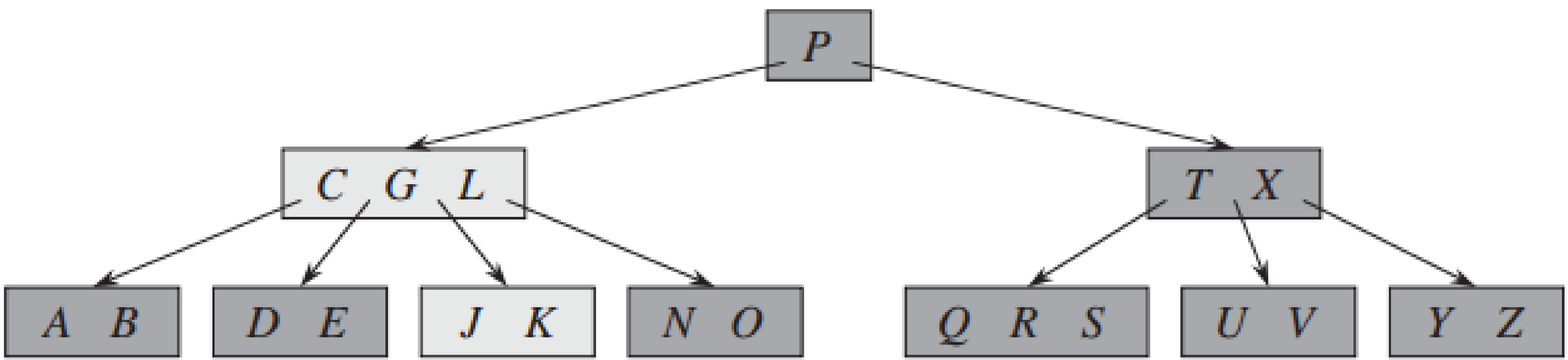


***m* = 6**

B-Trees

Delete operation: case #2a

- Key *k* is in a node *x* which is not a leaf
  - Let's delete *M*



***m* = 6**

B-Trees

Delete operation: case #2b

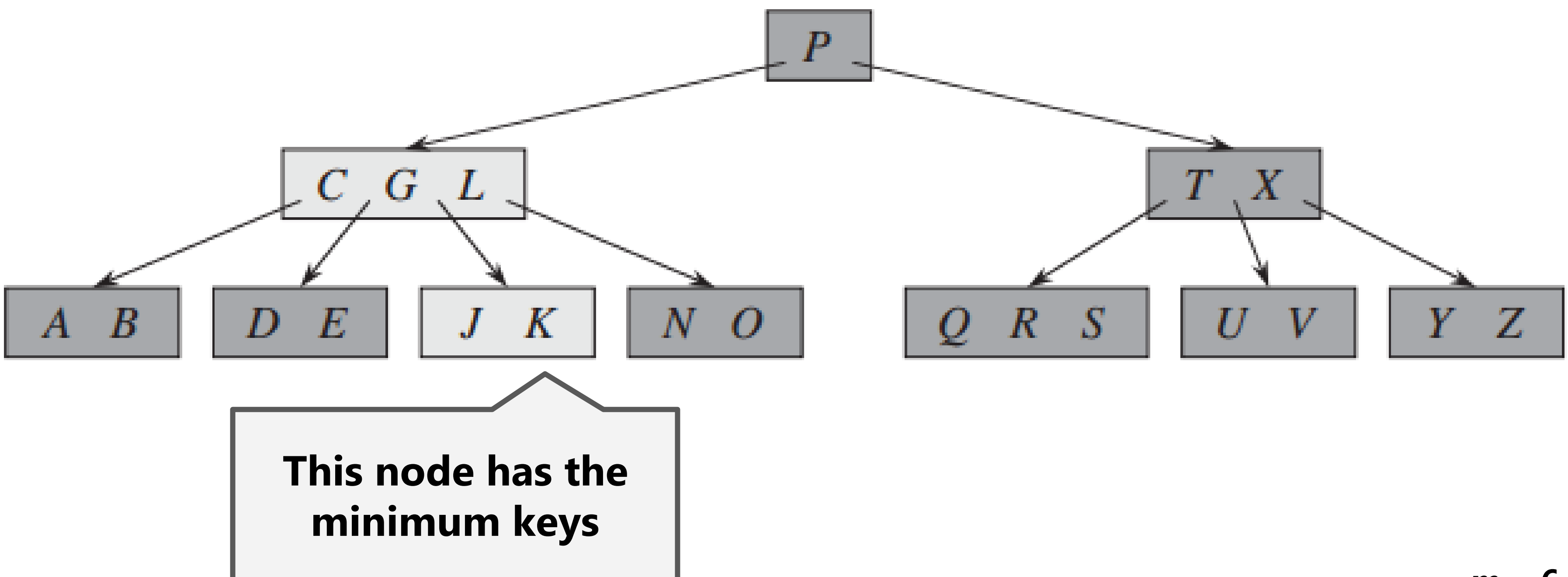
- Is the mirror case of 2a, but looking for the successor of k in the successor node.

$m = 6$

B-Trees

Delete operation: case #2c

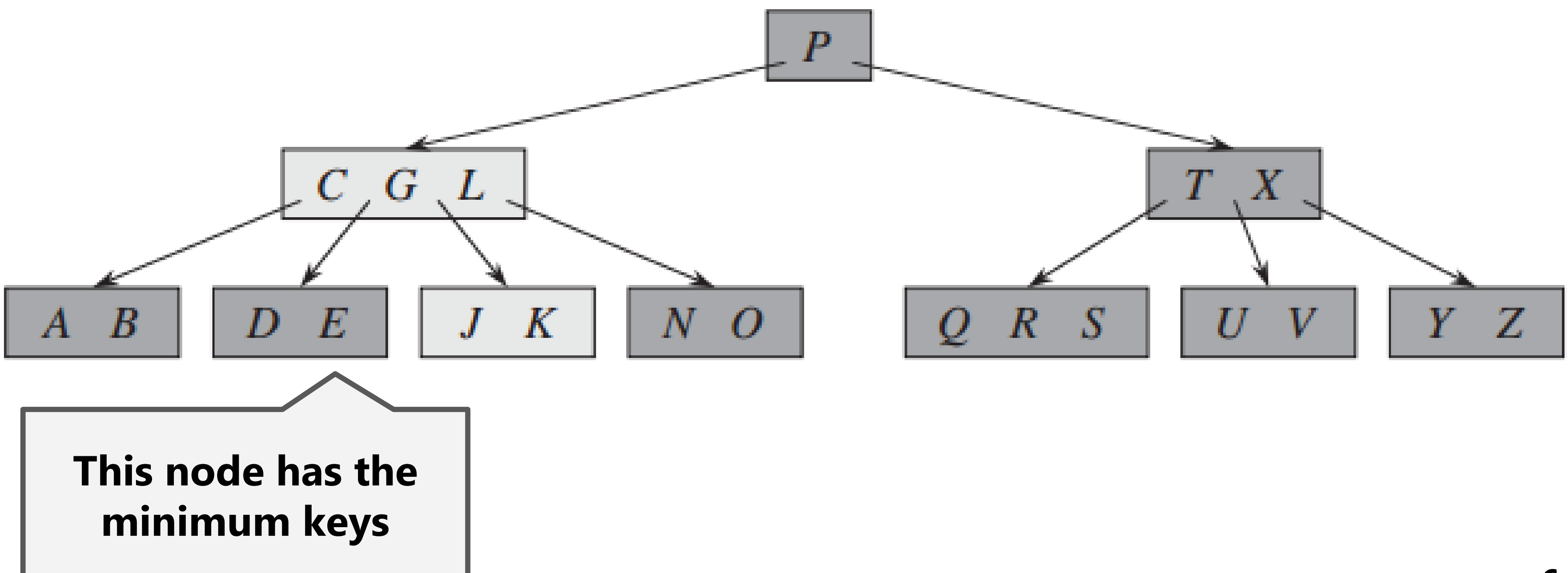
- Key *k* is in a node *x* which is not a leaf
  - Let's delete *G*



***m* = 6**

B-Trees
Delete operation: case #2c

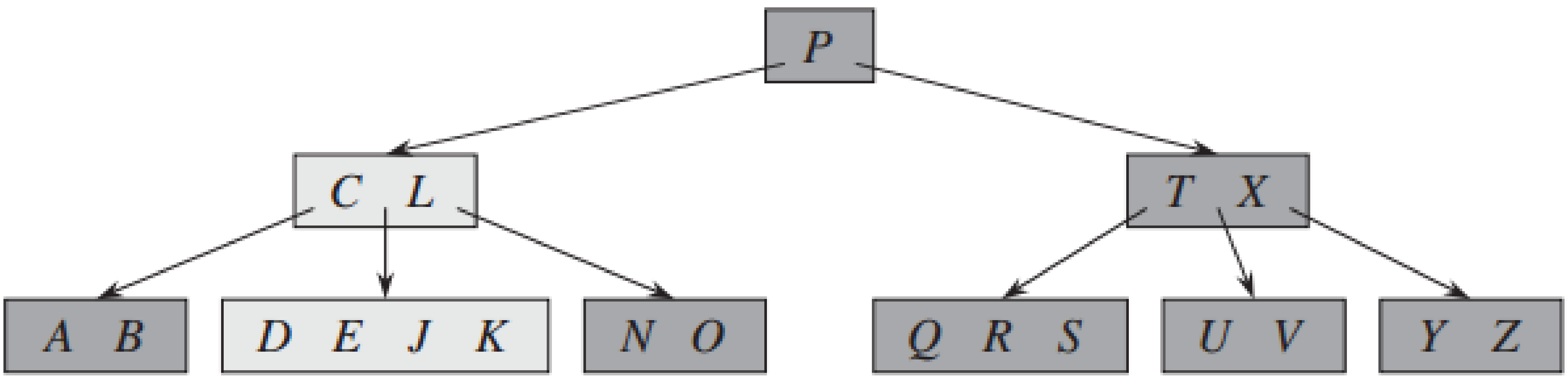
- Key *k* is in a node *x* which is not a leaf
  - Let's delete *G*



***m* = 6**

B-Trees
Delete operation: case #2c

- Key *k* is in a node *x* which is not a leaf
  - Let’s delete *G*



***m* = 6**

B-Trees

Delete operation: case #3

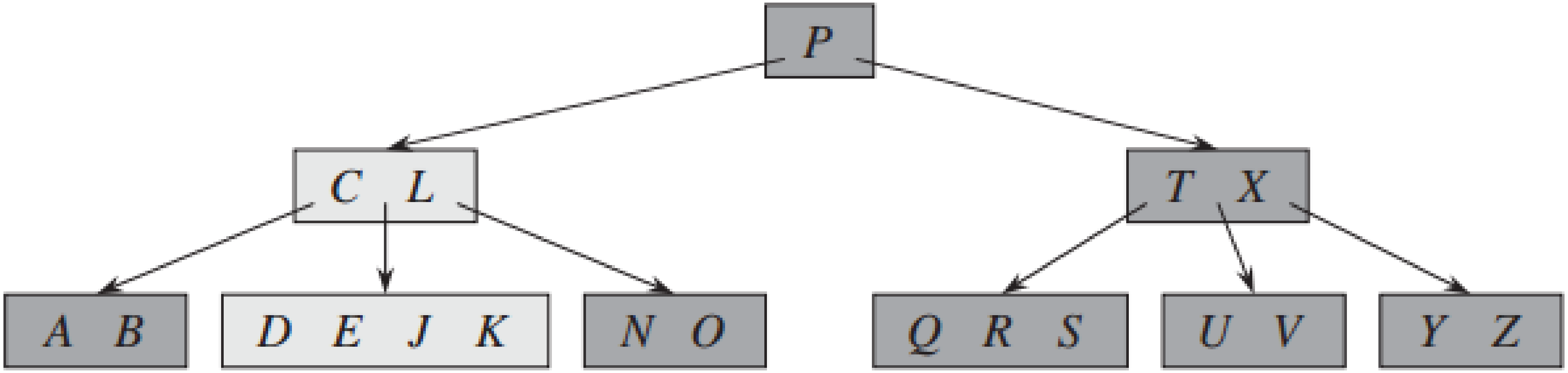
- Key *k* is in a node *x* which is a leaf but its root has the minimum keys ((m/2) - 1) then we have two cases:

m = 6

B-Trees

Delete operation: case #3a

- Delete *D*

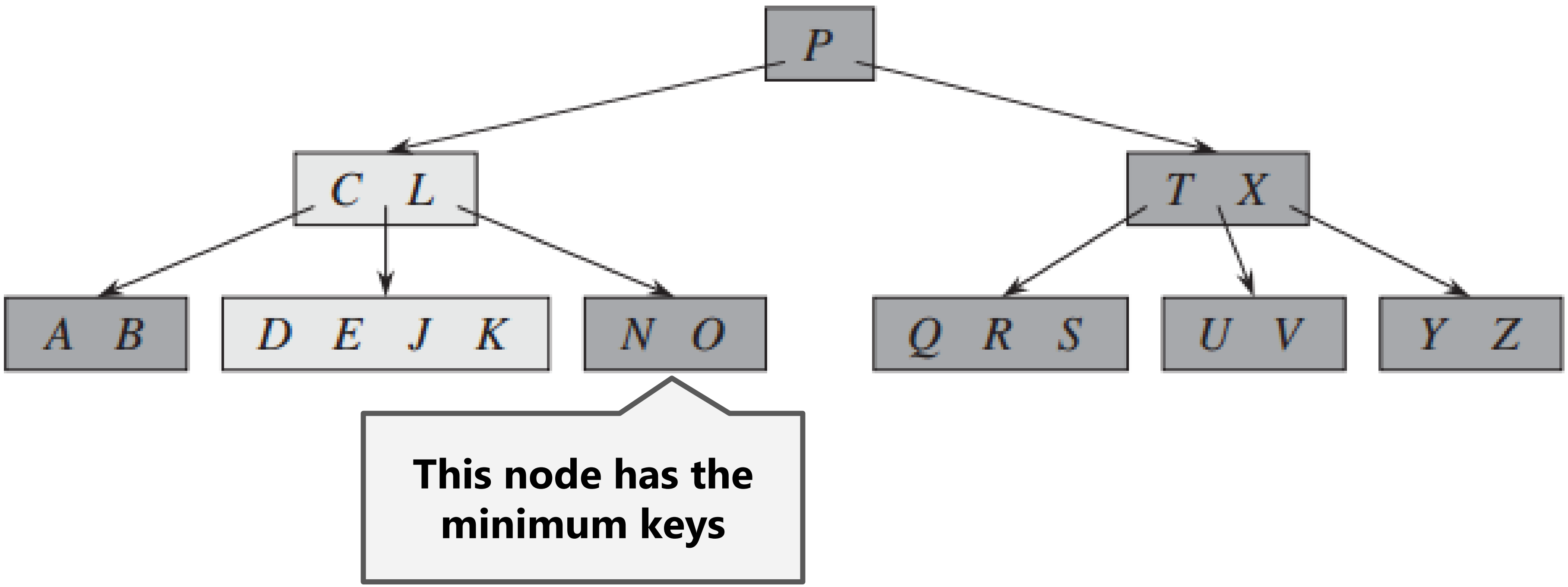


m = 6

B-Trees

Delete operation: case #3a

- Delete *D*

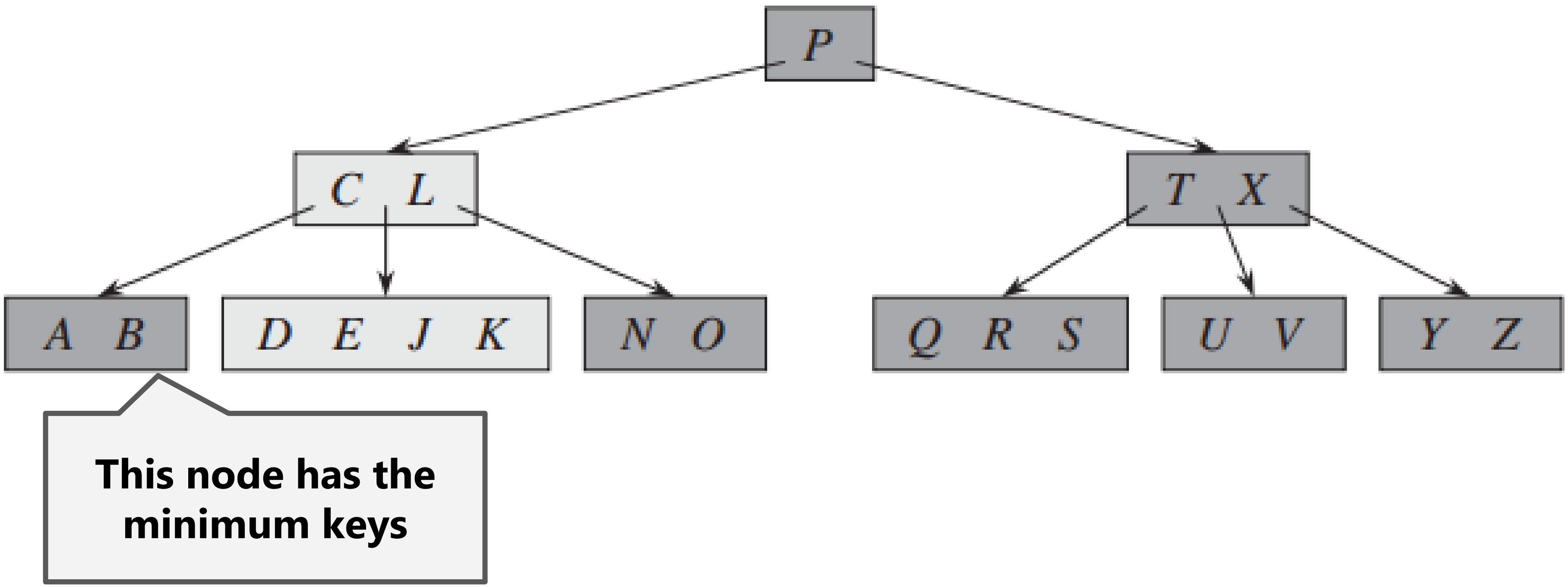


***m* = 6**

B-Trees

Delete operation: case #3a

- Delete *D*

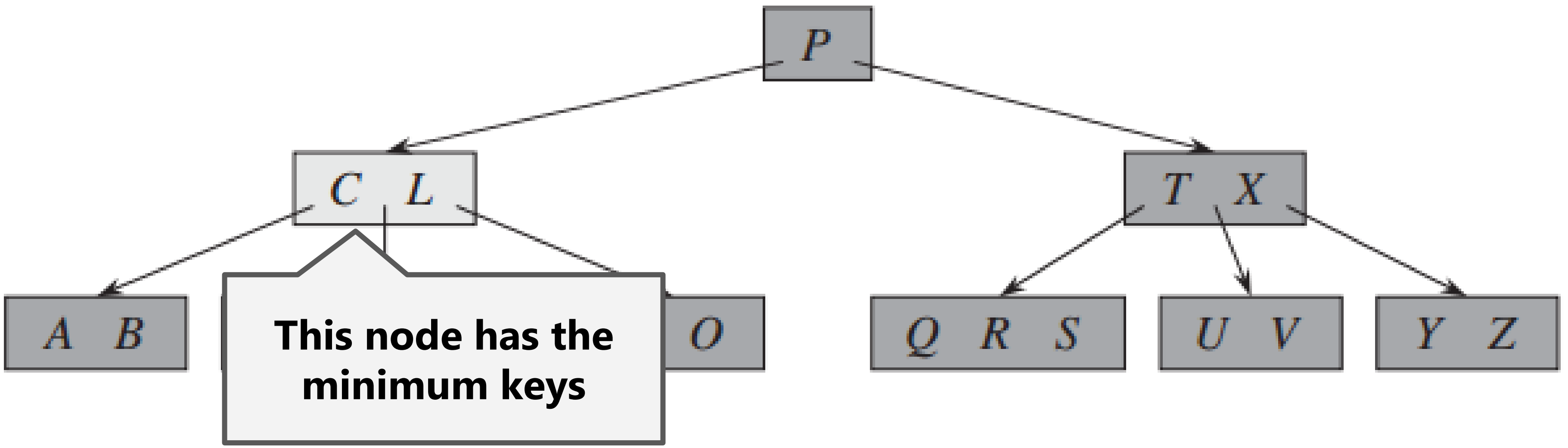


m = 6

B-Trees

Delete operation: case #3a

- Delete *D*

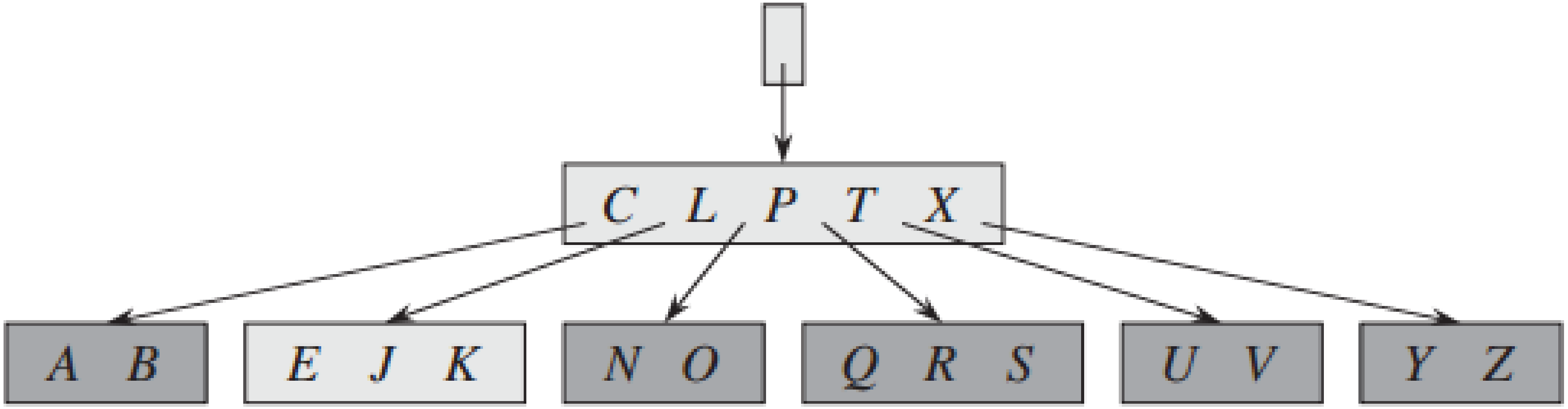


m = 6

B-Trees

Delete operation: case #3a

- Delete *D*

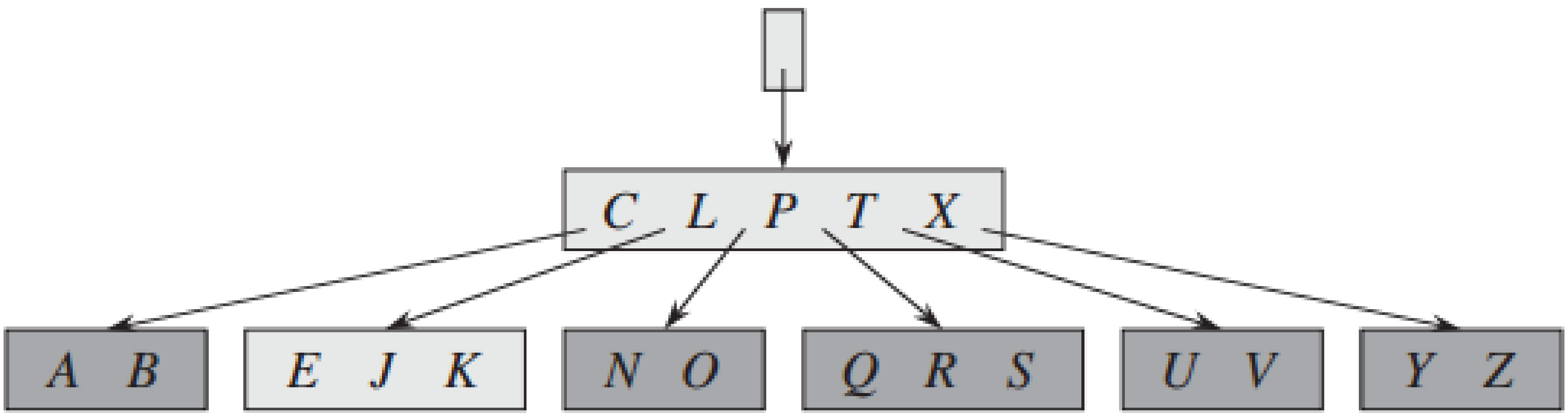


***m* = 6**

B-Trees

Delete operation: case #3b

- Delete *B*

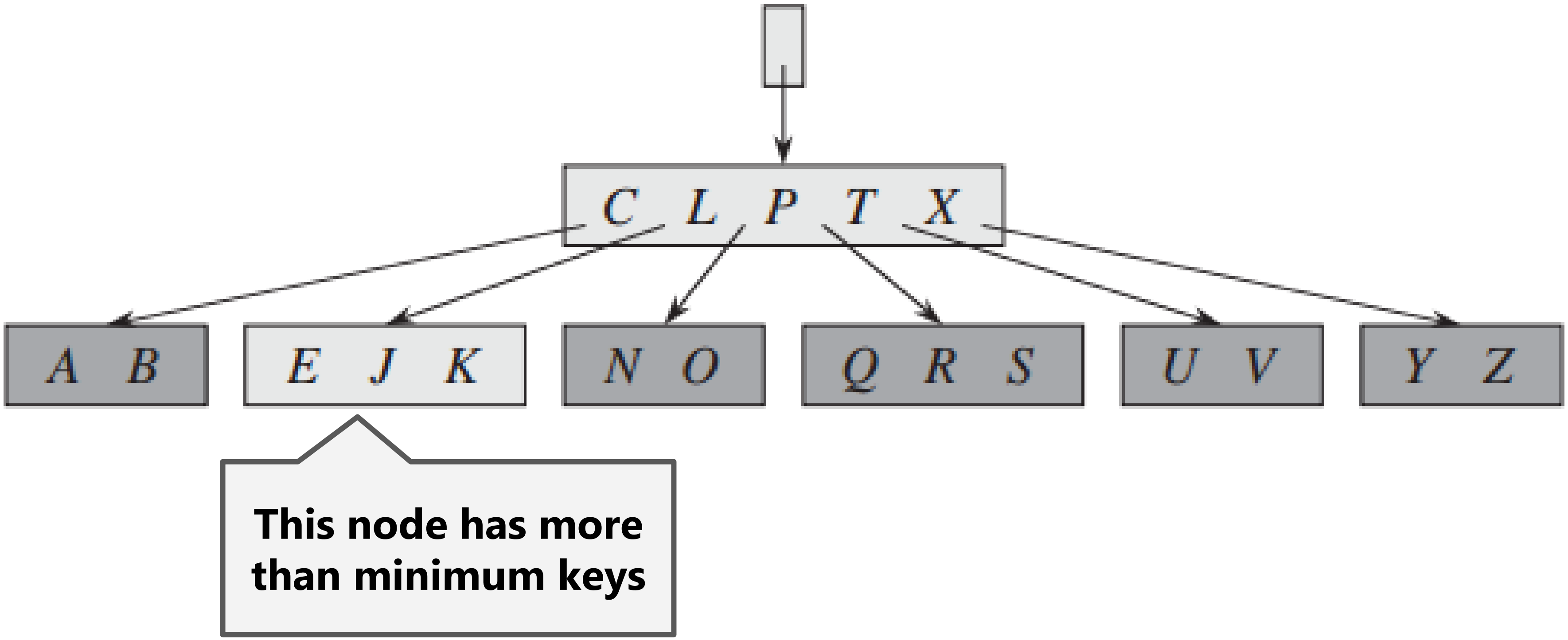


***m* = 6**

B-Trees

Delete operation: case #3b

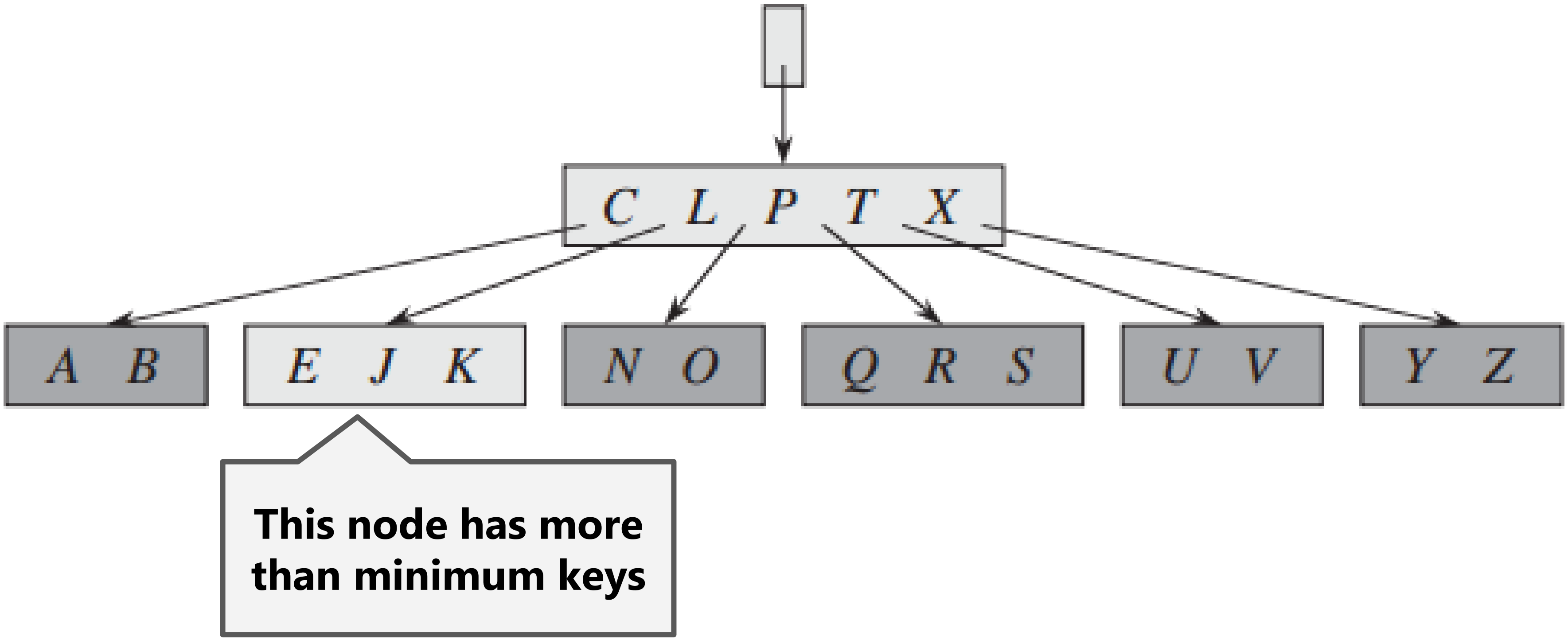
- Delete *B*



***m* = 6**

B-Trees
Delete operation: case #3b

- Delete *B*

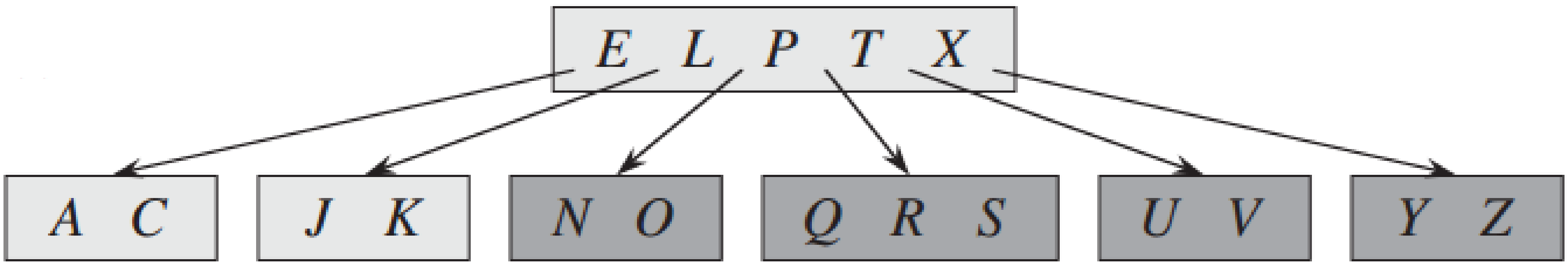


***m* = 6**

B-Trees

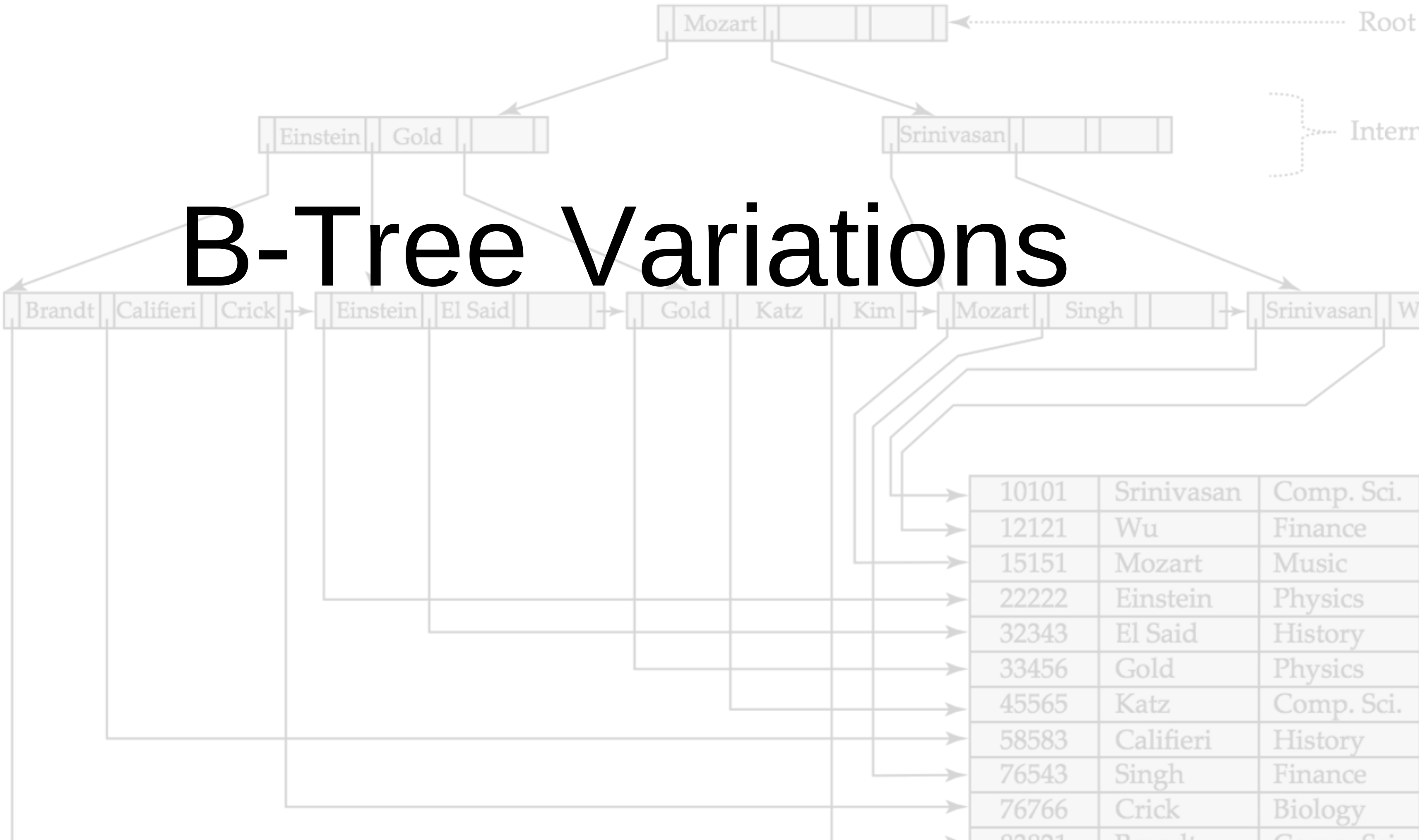
Delete operation: case #3b

- Delete *B*



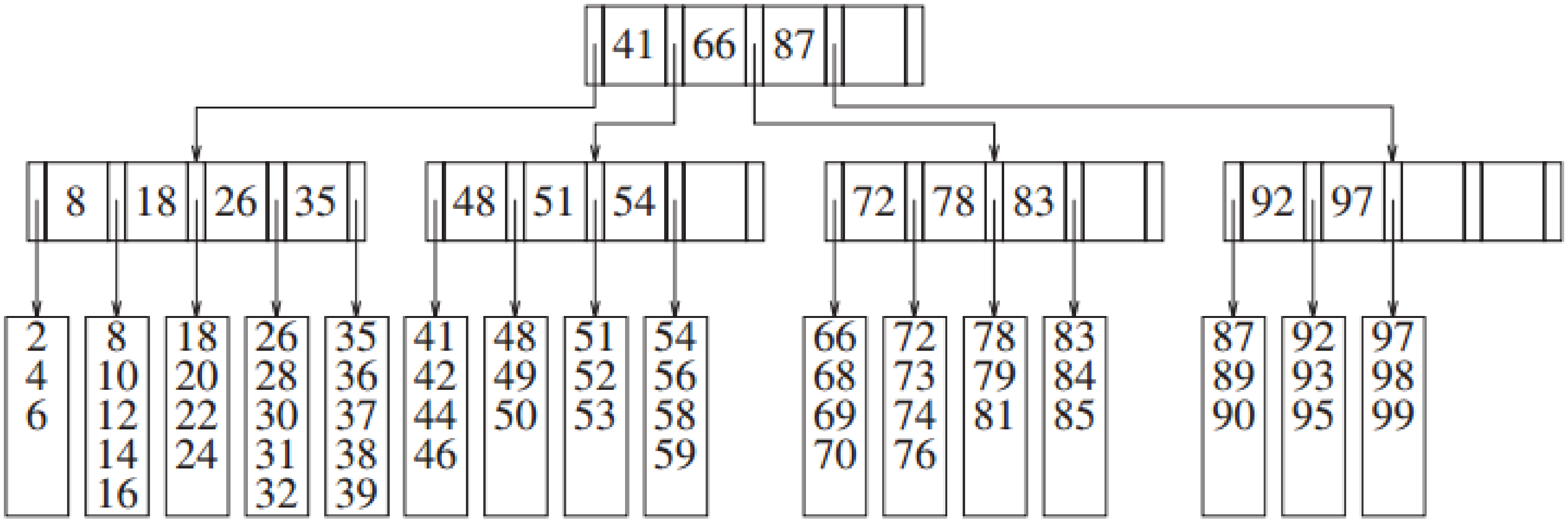
***m* = 6**

B-Tree Variations

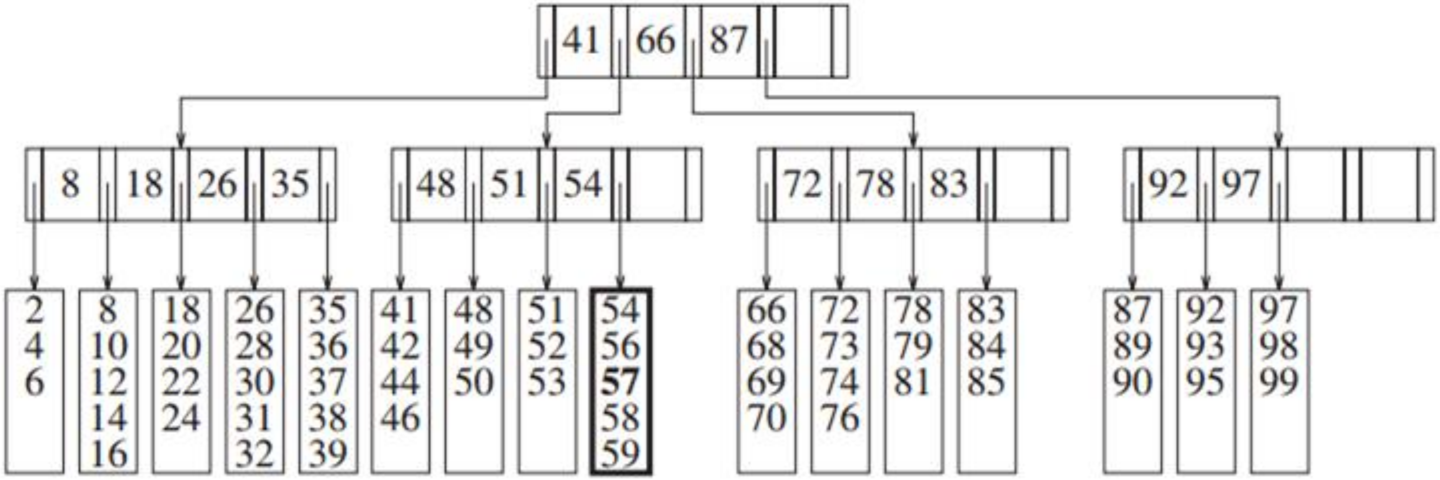


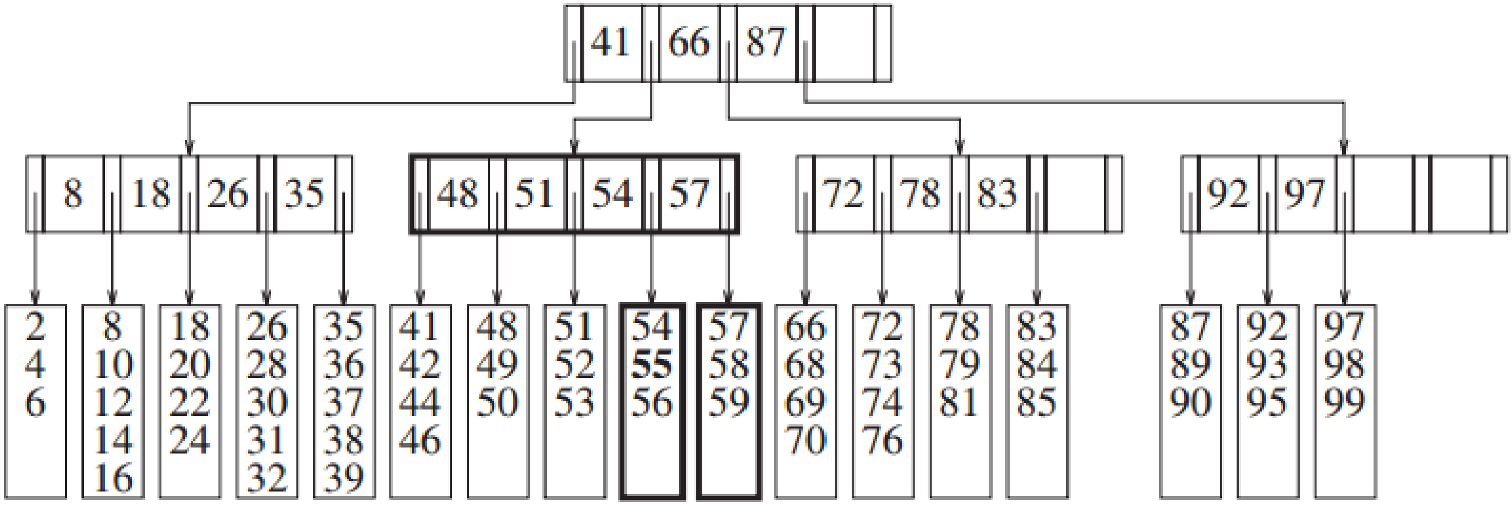
What is B+ Tree?

- Is a variation of B-Tree
  - The data items are stored only at the leaves
  - Internal nodes keep keys of the data at the leaves
  - Usually leaves have pointers to right sibling

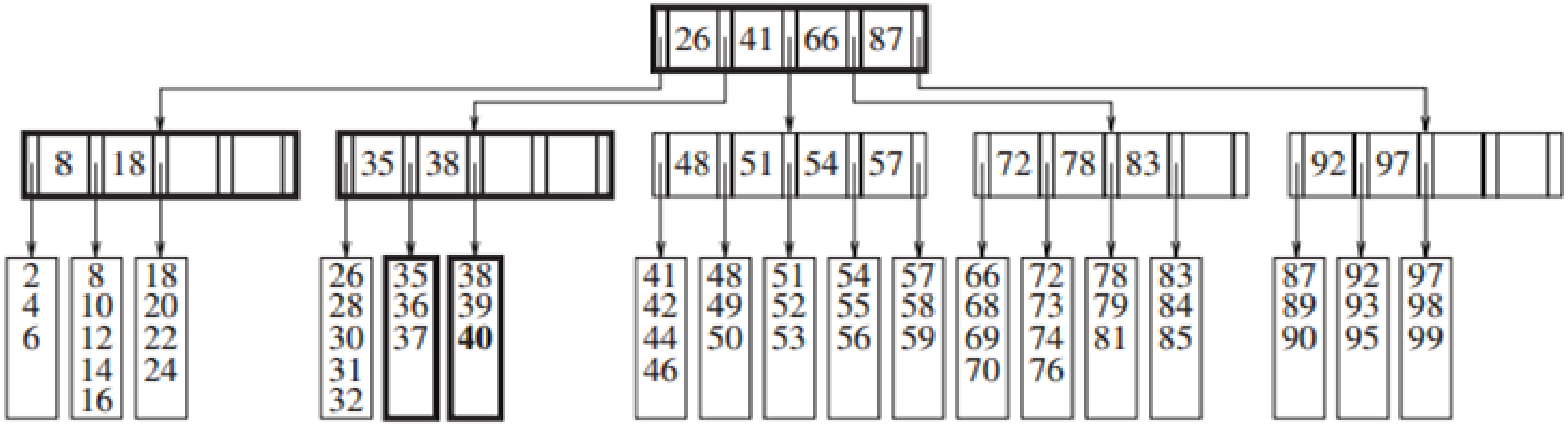


What is B+ Tree?





What is B+ Tree?



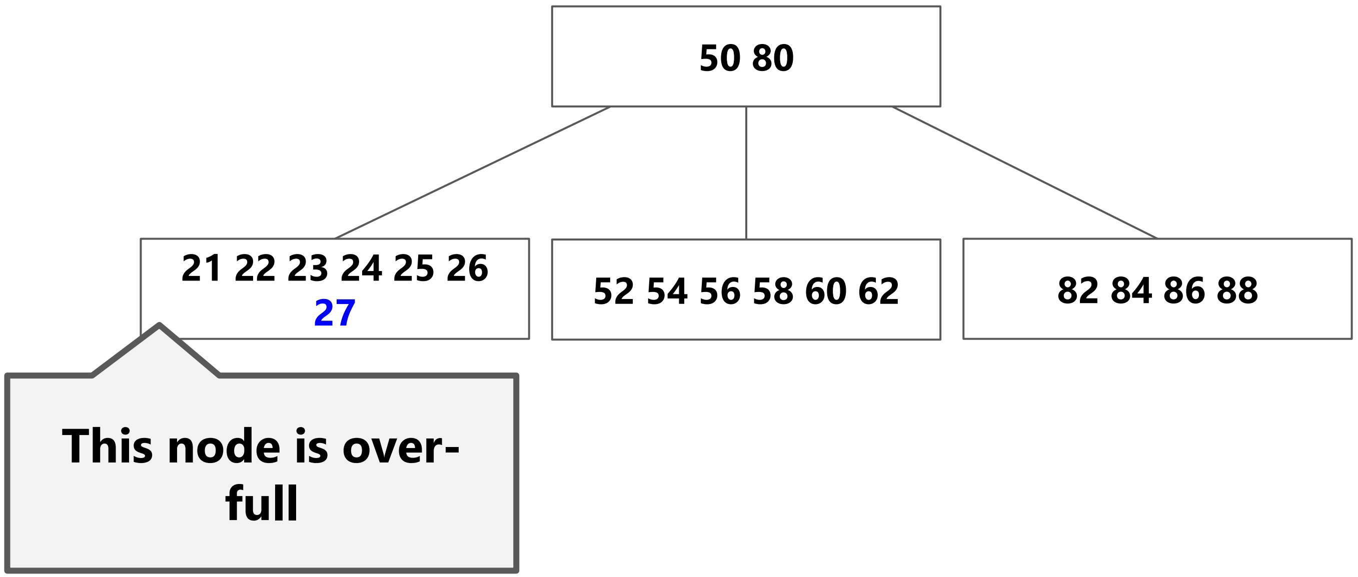
What is B* Tree?

- Is a variation of B-Tree
    - Every node except the root has at most *m* children
    - Every node, except for the root and the leaves, has at least (2*m* - 1) / 3 children
    - The root has at least 2 and most 2[(2m - 2)/3] +1 children
    - All leaves are on the same level
    - A nonleaf node with *k* children contains *k*-1 keys

- Uses space more efficiently, but the insertion process is slower

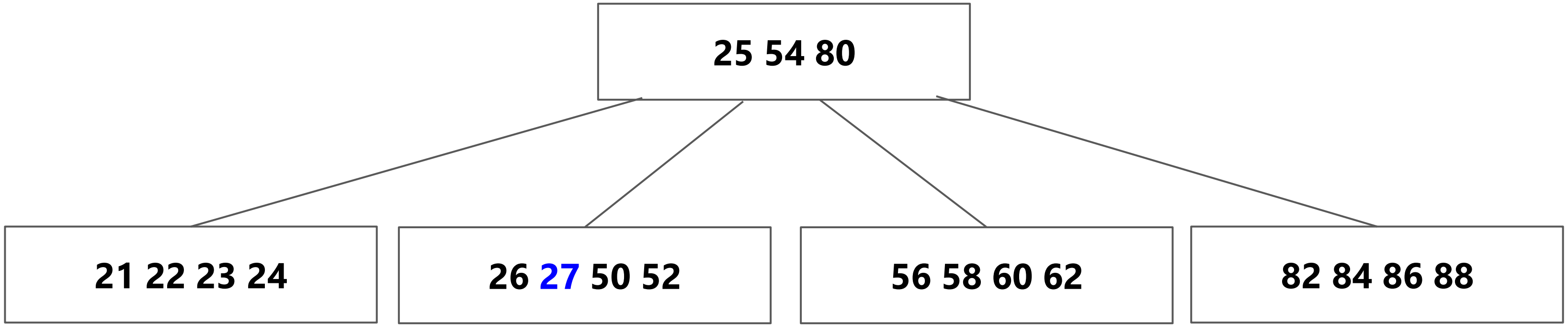
What is B* Tree?

- So how does insertion works?
    - Resist the temptation to split nodes so often. Do a local rotation instead

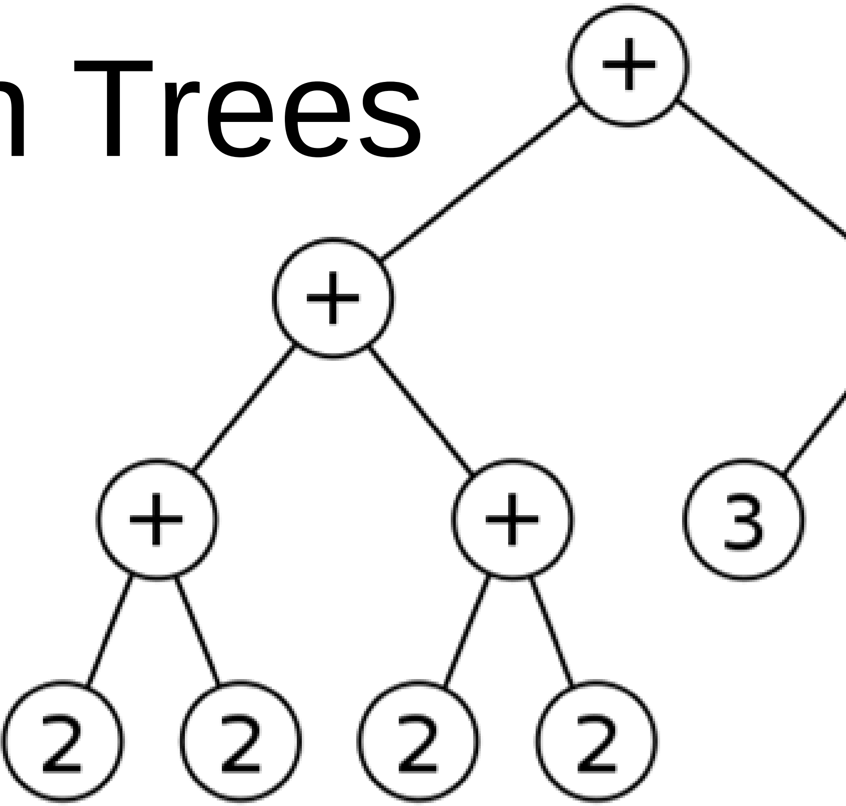


What is B* Tree?

- So how does insertion works?
  - Resist the temptation to split nodes so often. Do a local rotation instead



Expression Trees



Expression Trees

- An important application of the binary trees are expression trees.

- An expression is a sequence of tokens (lexical components that follow a set of specific rules)

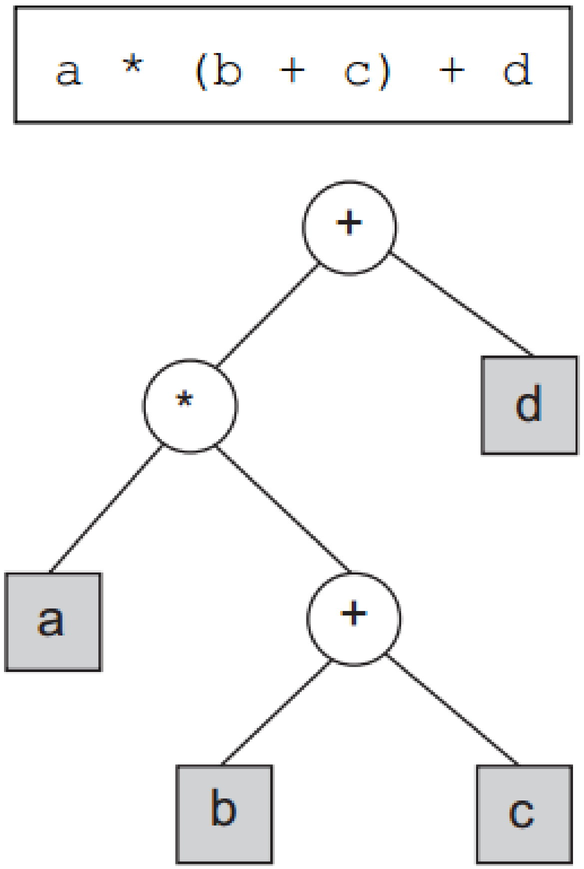
- Think of a token as an operator or keyword in a programming language

Expression Trees

- An expression tree is a binary tree with the following properties:
  - Each leaf is an operand
  - The root and internal nodes are operators
  - Every subtree is a subexpression which its root is an operator

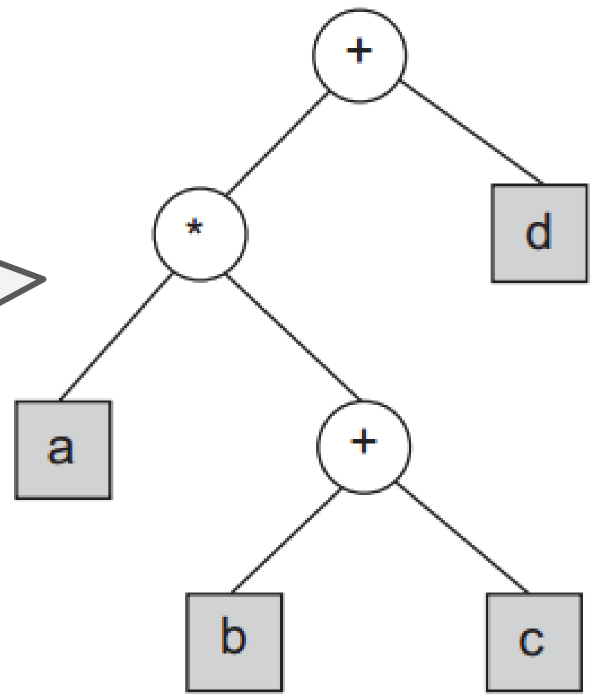
- Mostly used in compilers to represent the programming language expressions in memory

Expression Trees



Expression Trees

```
a * (b + c) + d
```



Where are the parenthesis?

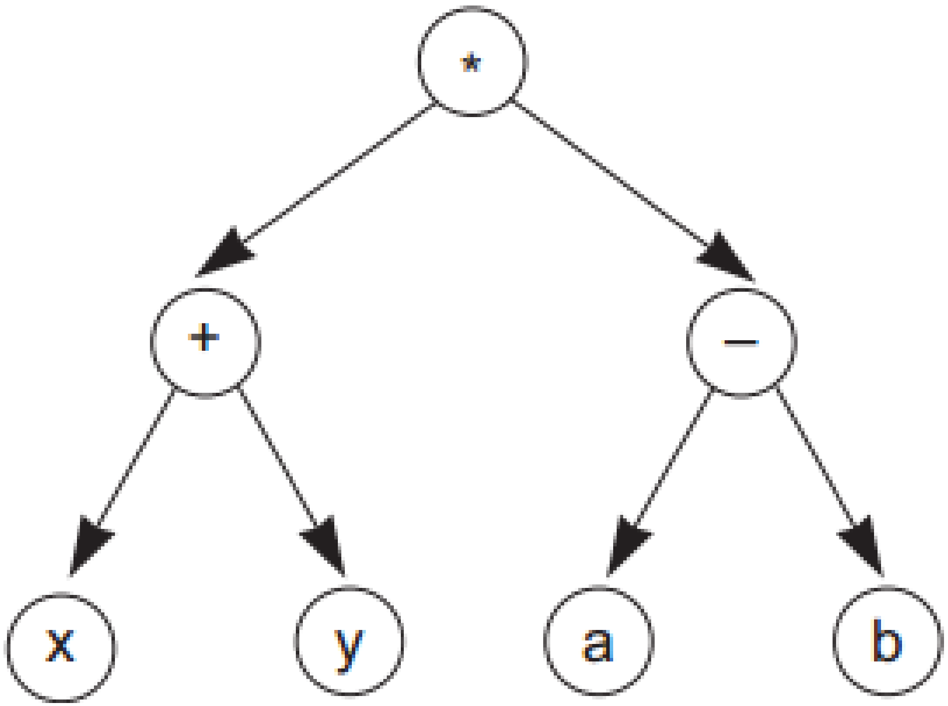
How to read an expression tree?

- Basically all you have to do is traverse the tree in one specific way:
  - Infix
  - Postfix
  - Prefix

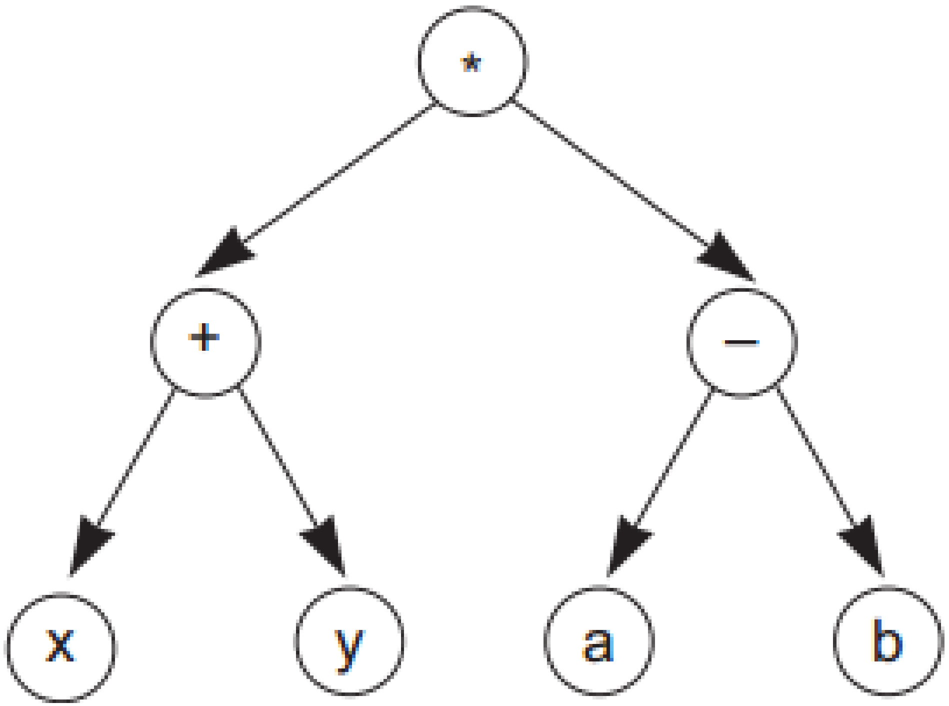
How to read an expression tree?

```
Algorithm infix (tree)
/*Print the infix expression for an expression tree.
 Pre : tree is a pointer to an expression tree
 Post: the infix expression has been printed*/
 if (tree not empty)
    if (tree token is operator)
       print (open parenthesis)
    end if
    infix (tree left subtree)
    print (tree token)
    infix (tree right subtree)
    if (tree token is operator)
       print (close parenthesis)
    end if
 end if
end infix
```

Find the expression!

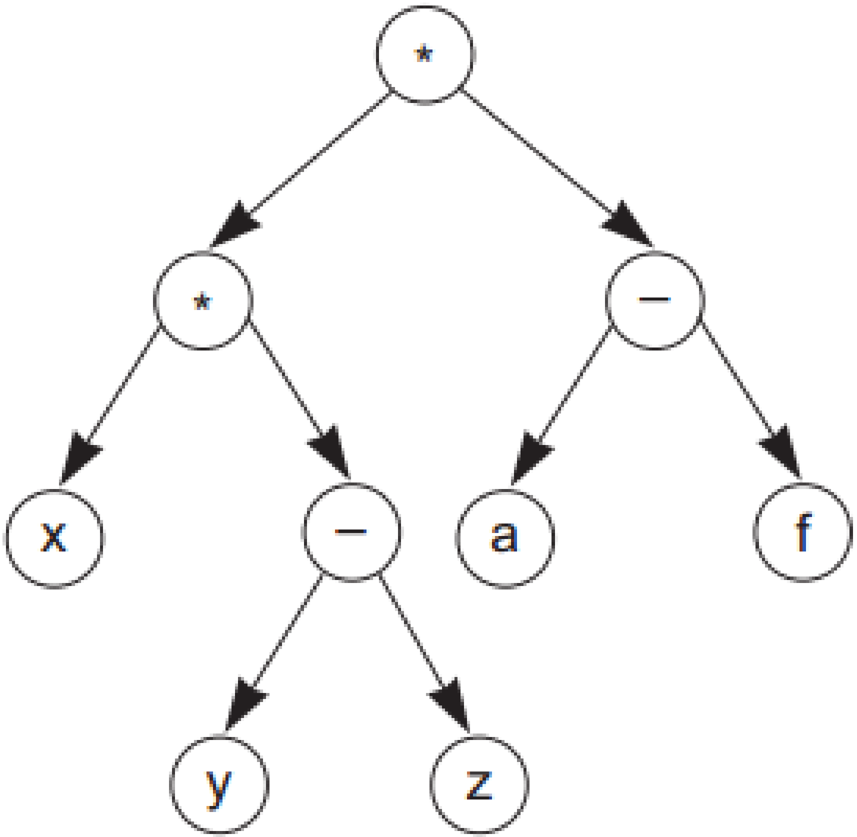


Find the expression!

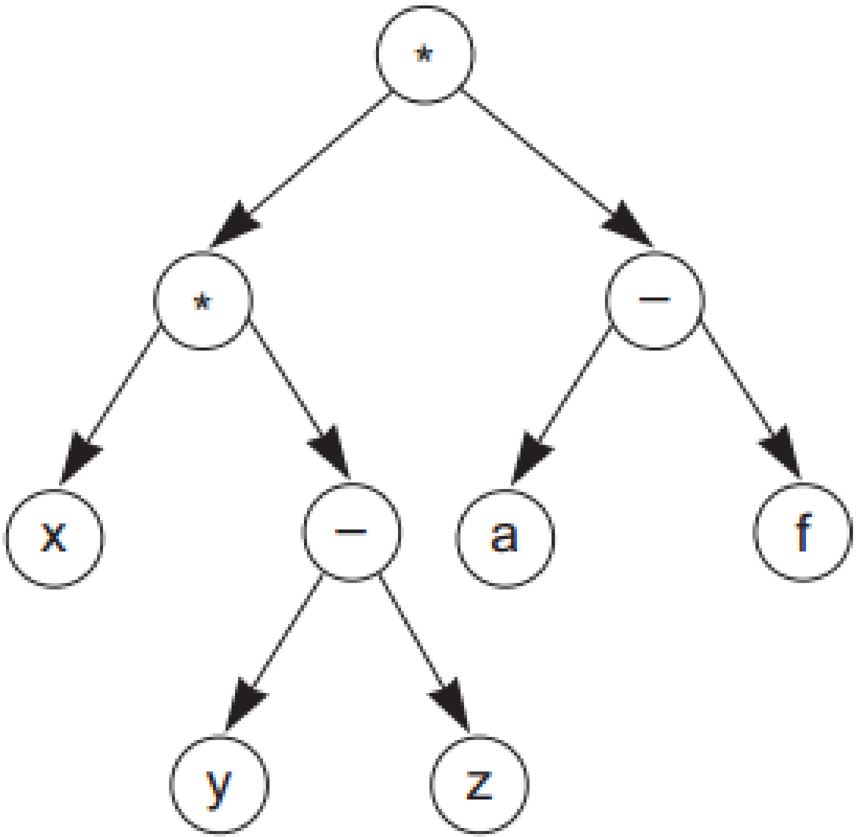


(x + y) * (a - b)

Find the expression!

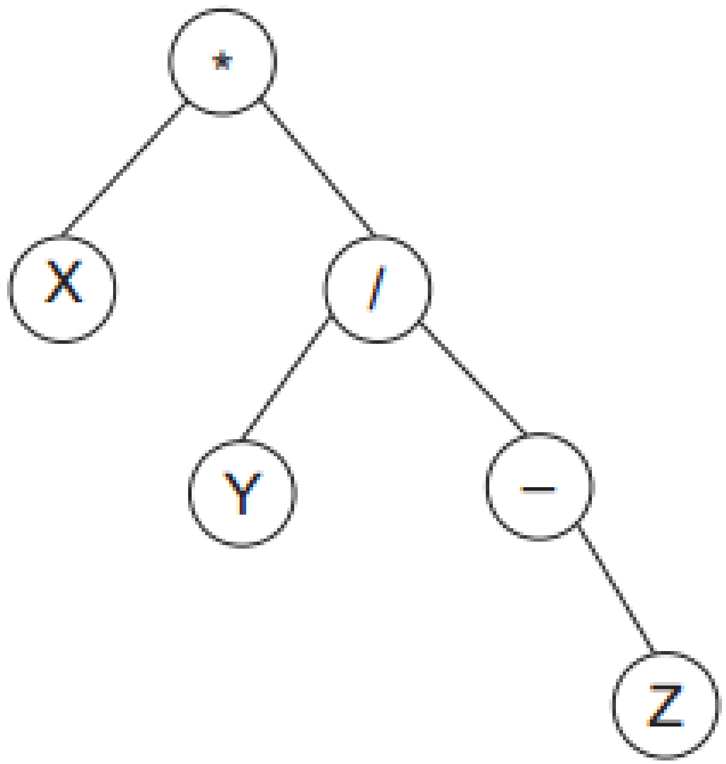


Find the expression!

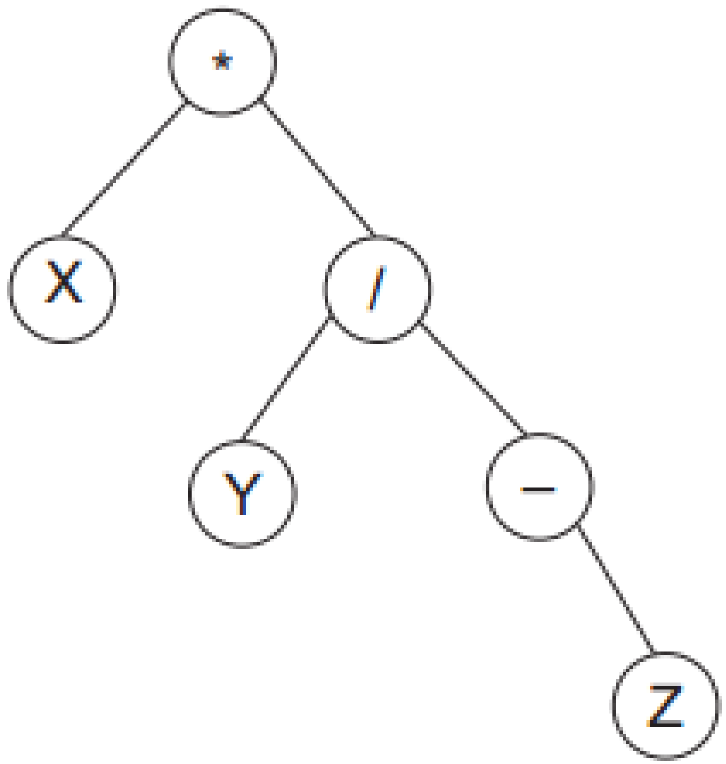


(x * (y - z)) * (a - f)

Find the expression!

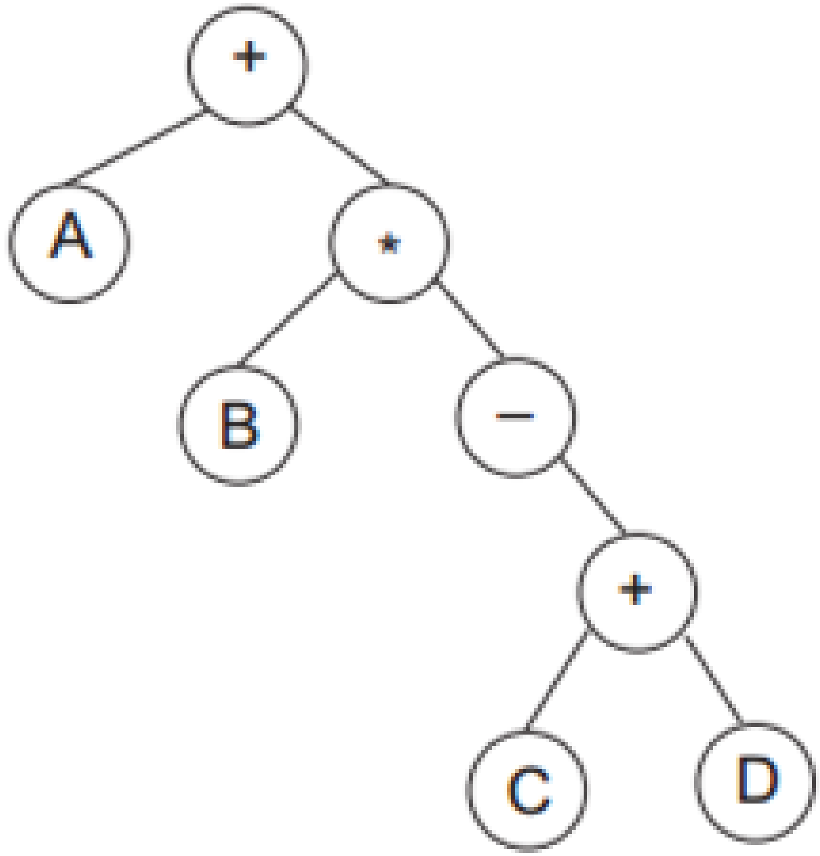


Find the expression!

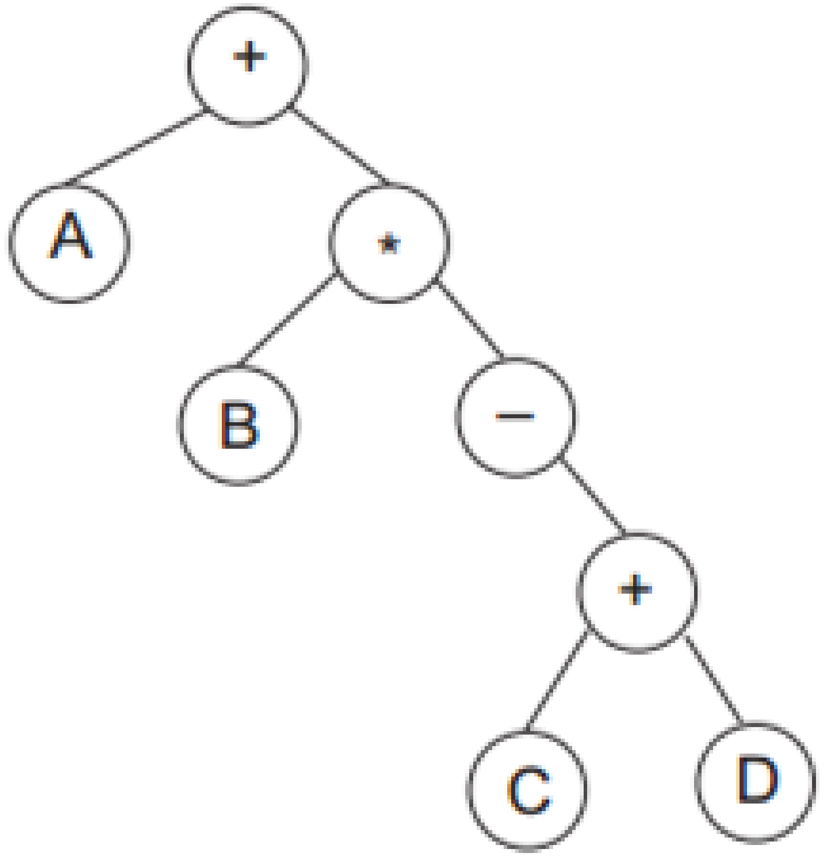


(x * (y / -Z))

Find the expression!

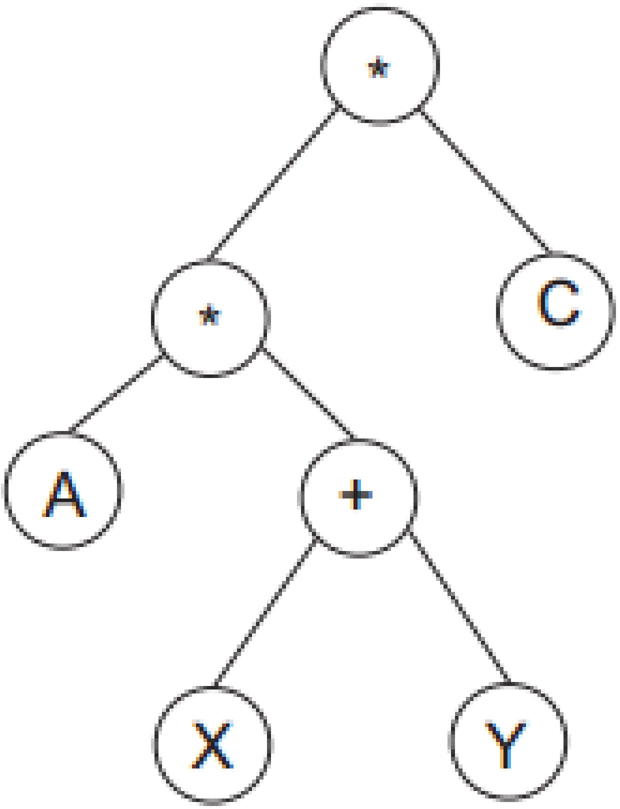


Find the expression!

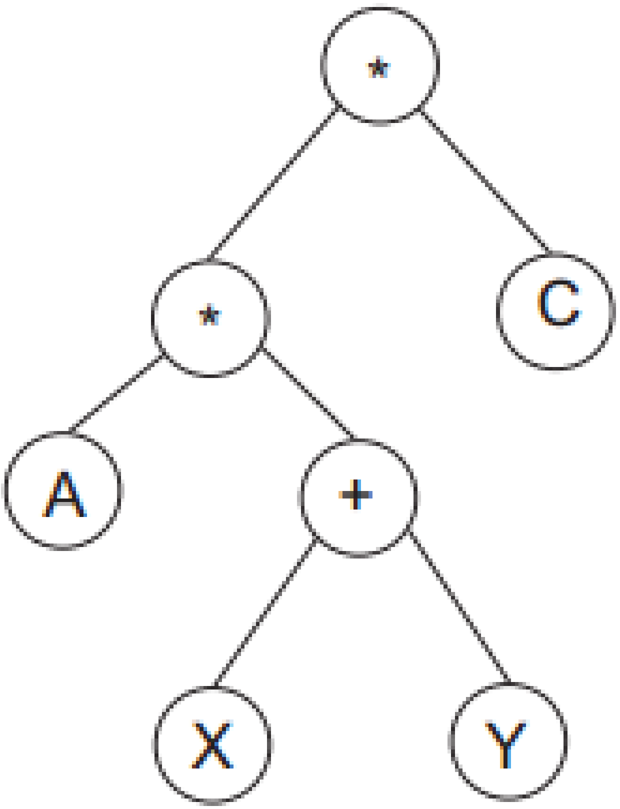


(A + (B * - (C + D)))

Find the expression!



Find the expression!



((A * (X + Y)) * C)

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

Stack

(

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a

Stack

(

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a

Stack

(

Is there another operator at the top?

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a

Stack

(+

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a

Stack

(+ (

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a b

Stack

(+ (

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a b

Stack

(+ (*

Is there another operator at the top?

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a b c

Stack

(+ (*

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a b c

Stack

(+ (*

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a b c *

Stack

(+ (

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a b c *

Stack

(+

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a b c * +

Stack

(

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a b c * +

Stack

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a b c * +

Stack

+

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a b c * +

Stack

+ (((

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a b c * + d

Stack

+ (((

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a b c * + d

Stack

+ (((*

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a b c * + d e

Stack

+ (((*

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a b c * + d e *

Stack

+ ((

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a b c * + d e *

Stack

+ ((+

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a b c * + d e * f

Stack

+ ((+

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a b c * + d e * f +

Stack

+ (

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a b c * + d e * f +

Stack

+ (*

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a b c * + d e * f + g

Stack

+ (*

How to create a tree from expression

(a + (b * c)) + (((d * e) + f) * g)

Output Queue

a b c * + d e * f + g * +

Stack

How to create a tree from expression

Output Queue

a b c * + d e * f + g * +

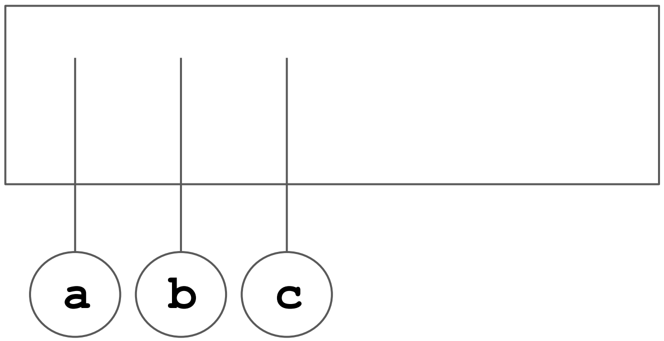
Stack

How to create a tree from expression

Output Queue

a b c * + d e * f + g * +

Stack

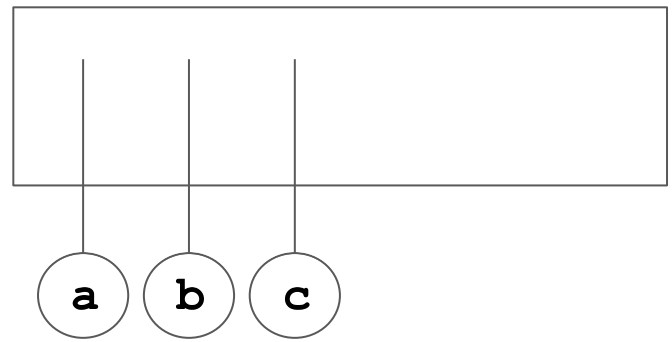


How to create a tree from expression

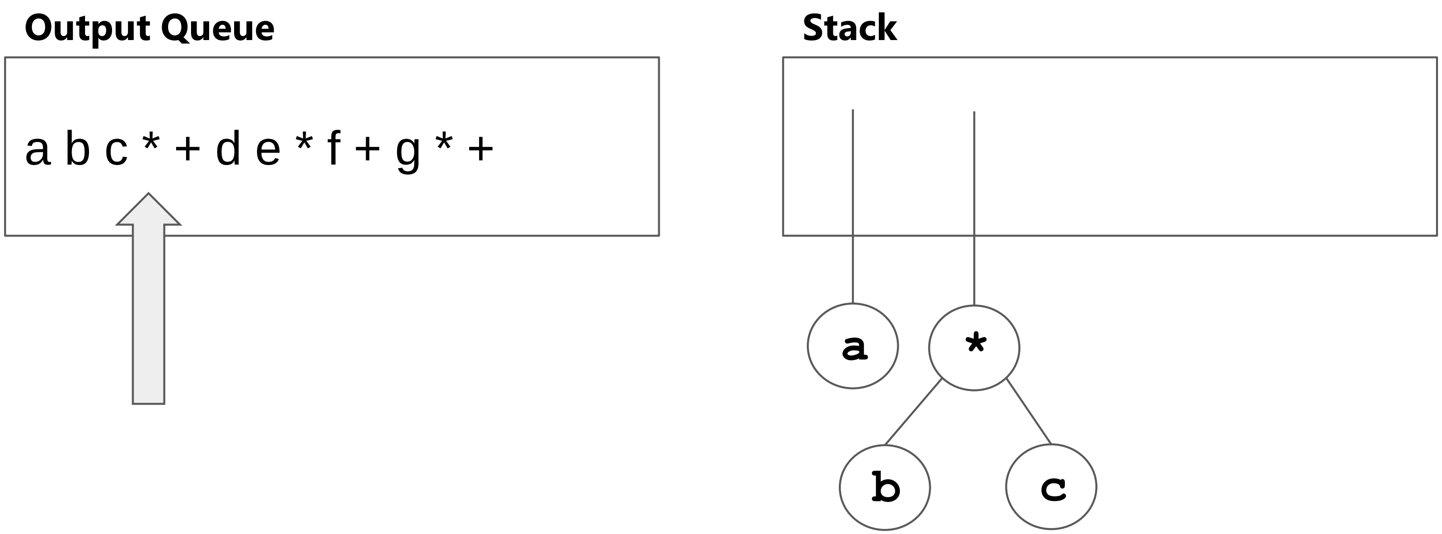
Output Queue

a b c * + d e * f + g * +

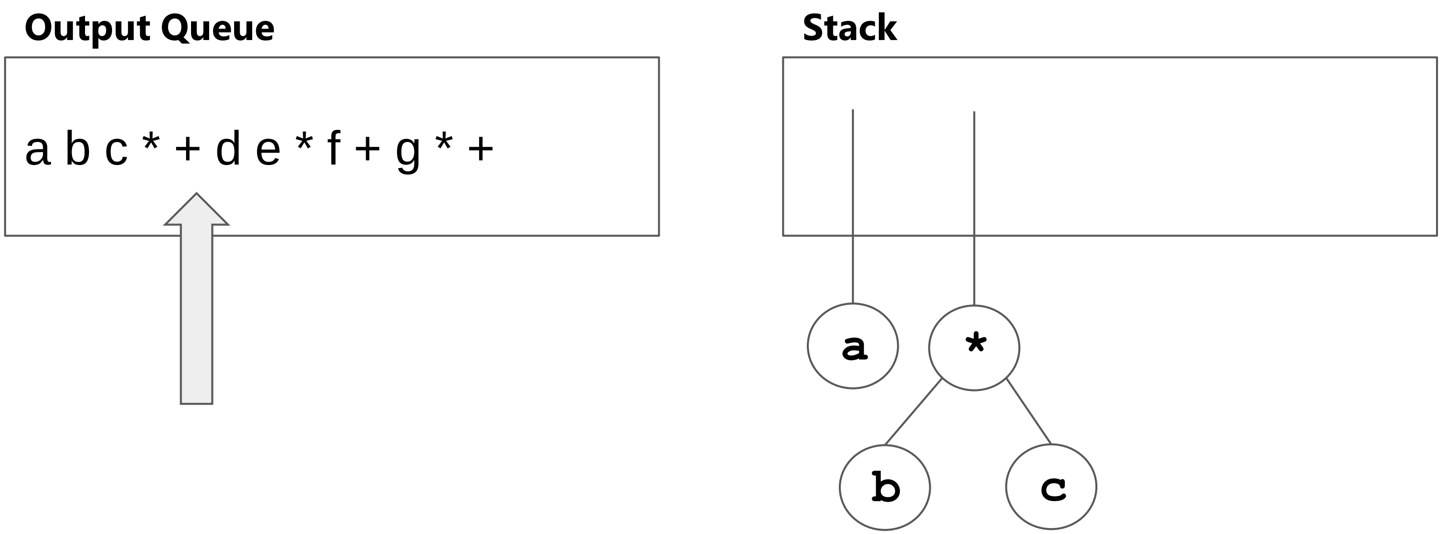
Stack



How to create a tree from expression



How to create a tree from expression

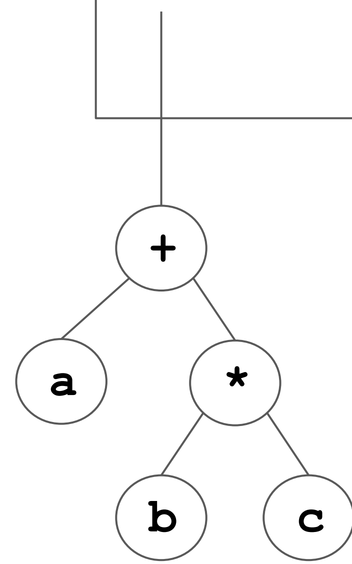


How to create a tree from expression

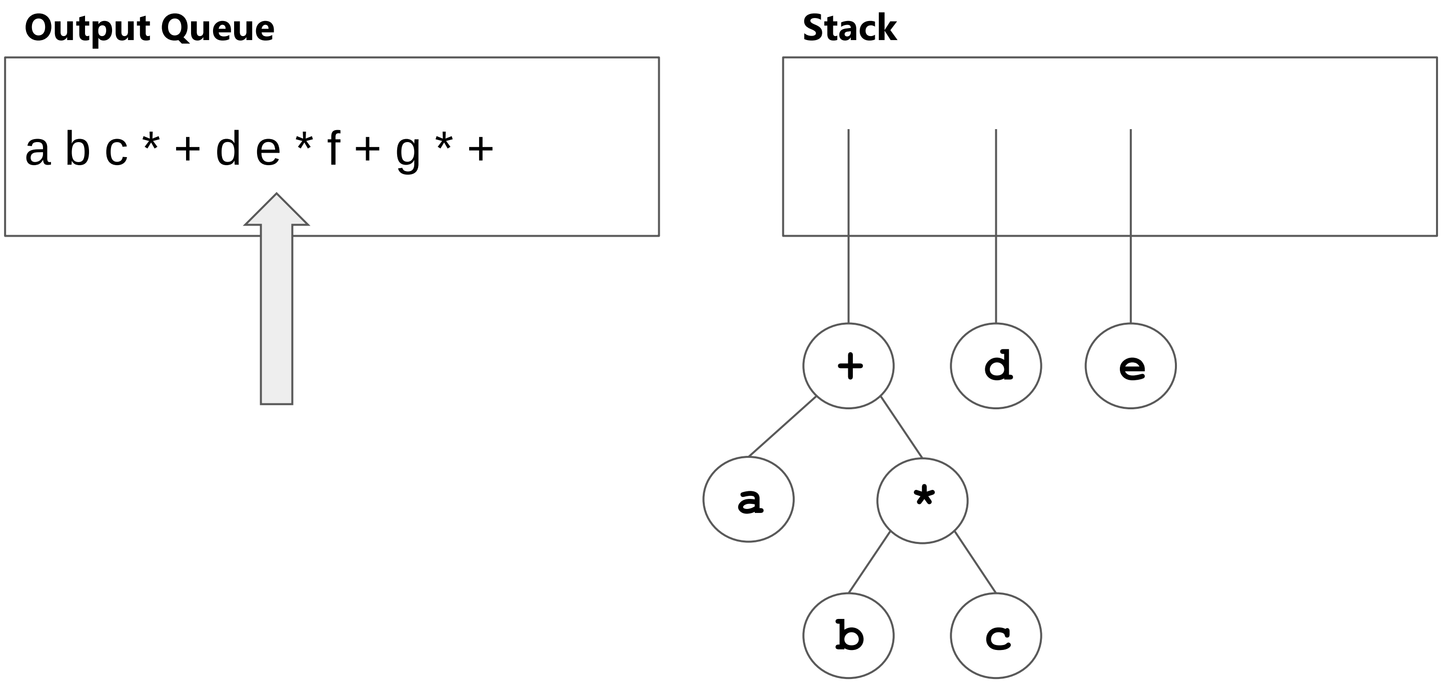
Output Queue

a b c * + d e * f + g * +

Stack



How to create a tree from expression

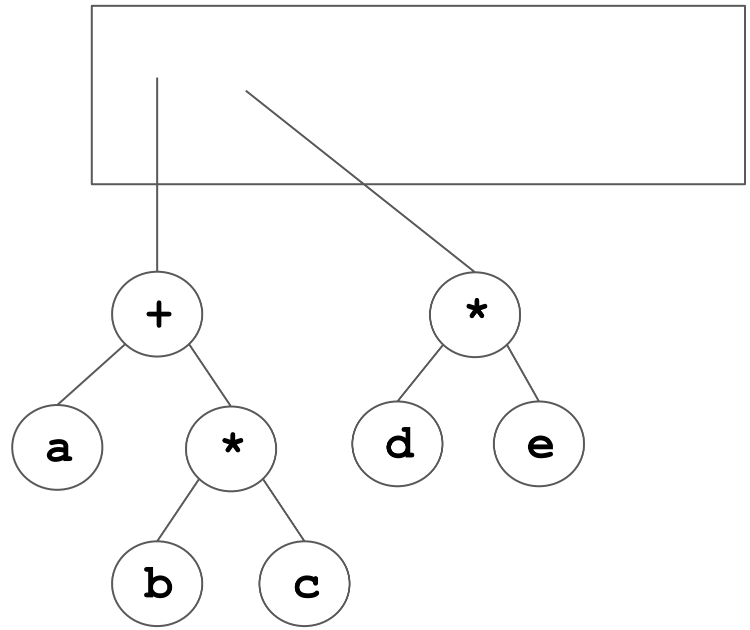


How to create a tree from expression

Output Queue

a b c * + d e * f + g * +

Stack

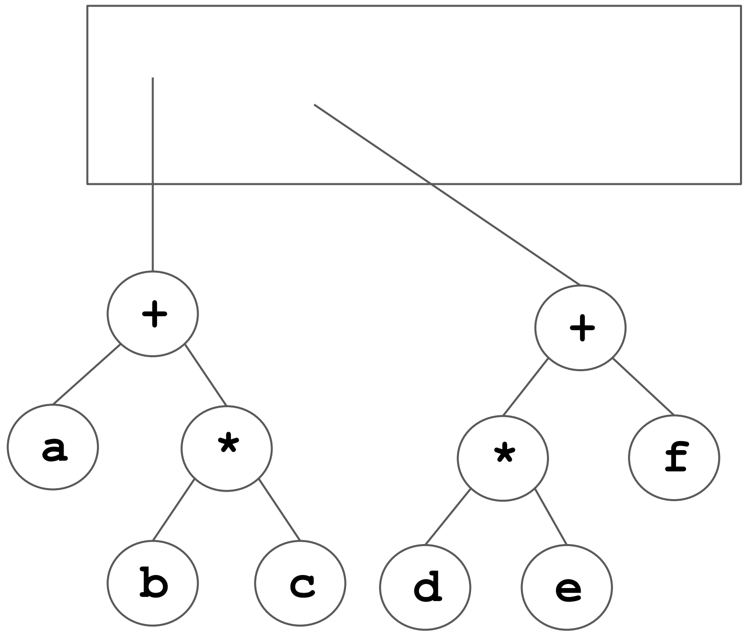


How to create a tree from expression

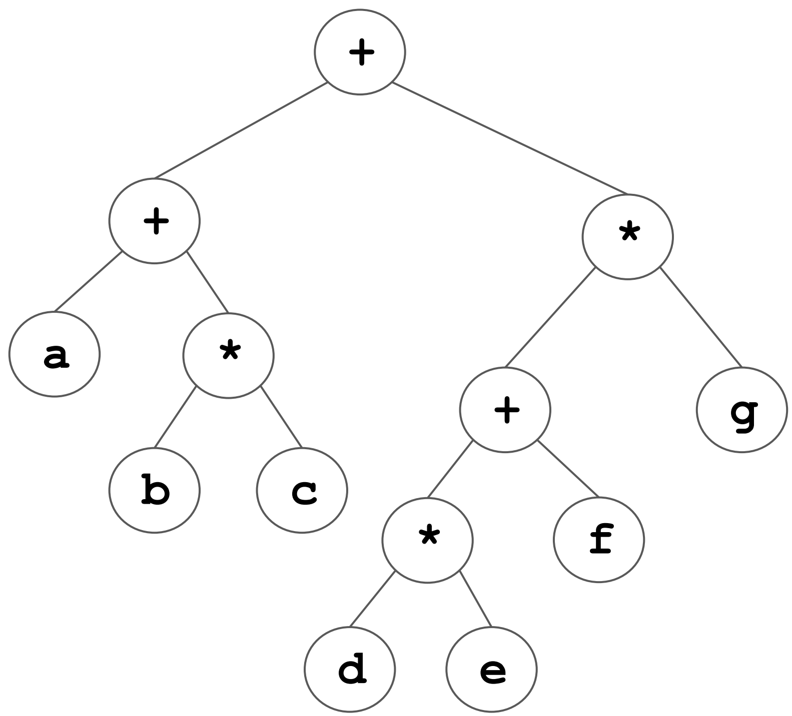
Output Queue

a b c * + d e * f + g * +

Stack



How to create a tree from expression



Hierarchical Data Structures

CE-1103 Algorithms and Data Structures