Estruturas de Informação

Trees

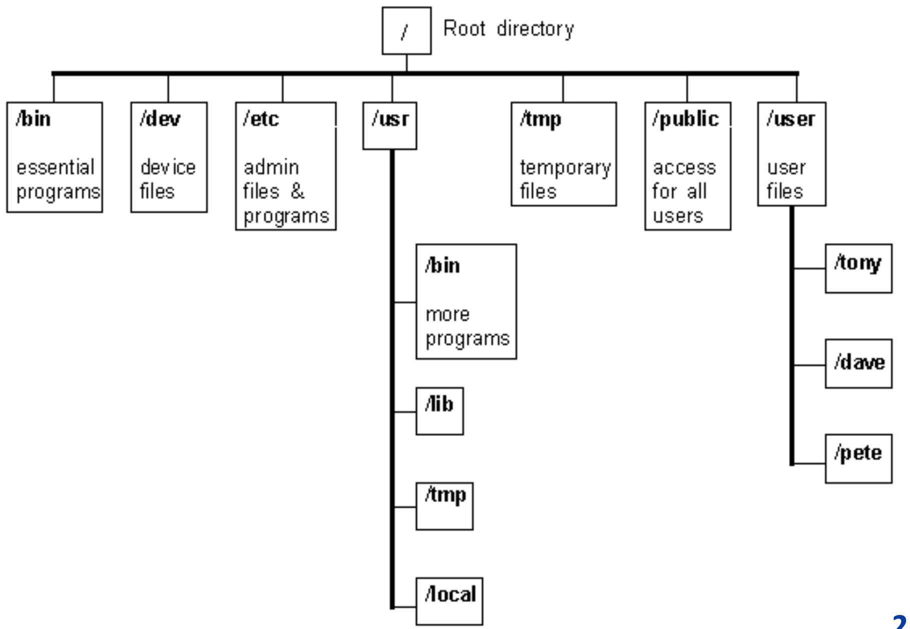
Fátima Rodrigues
mfc@isep.ipp.pt
Departamento de Engenharia Informática (DEI/ISEP)

Trees

- In computer science, a tree is an ADT which stores elements hierarchically
- Tree consists of nodes with a parent-child relation

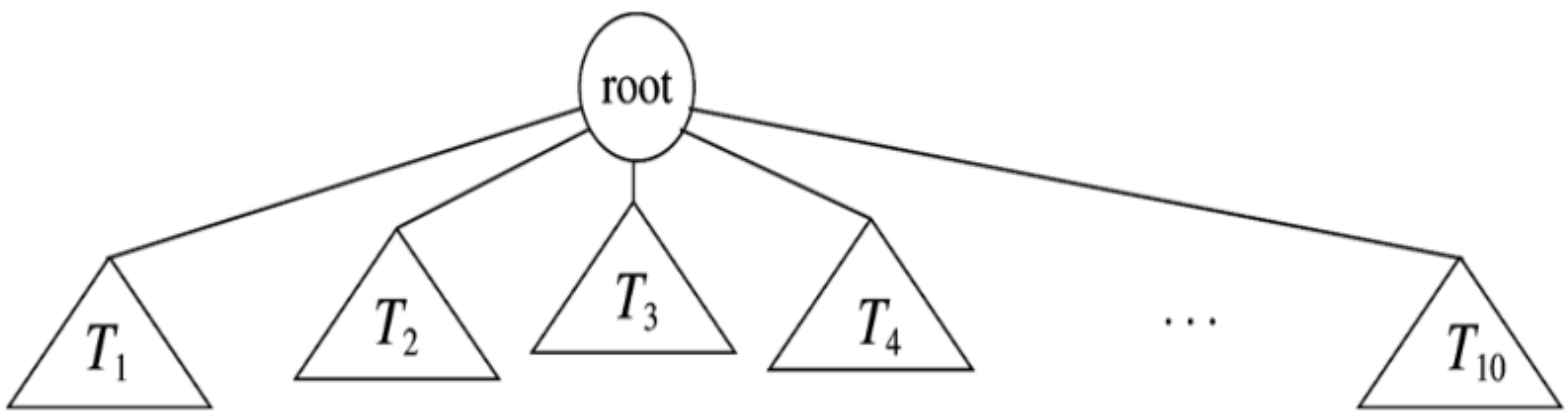
Applications:

- File systems
- Programming environments
- Taxonomies
- Image Representation
- Database Indexes
- ....



Tree – Definition

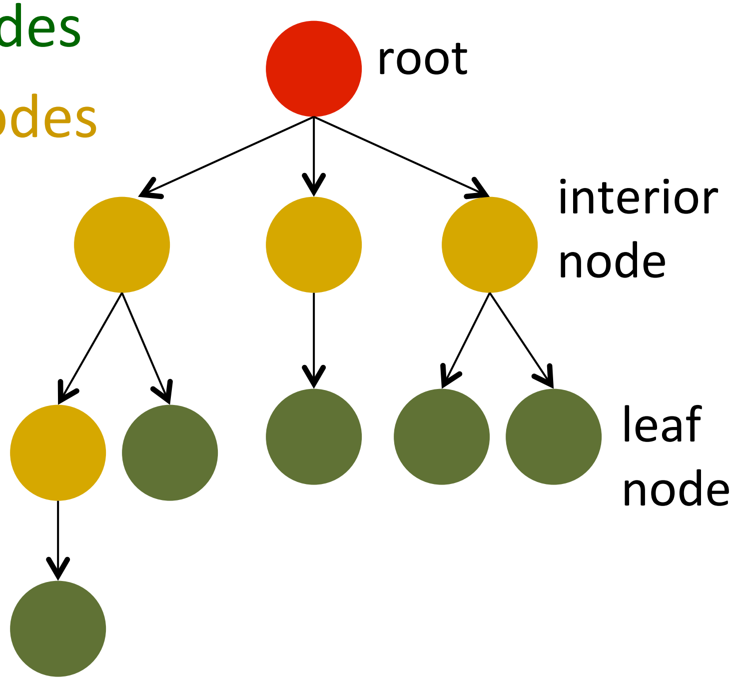
- A tree is a set of nodes that may be empty
- If not empty, then there is a distinguished node r, called root and zero or more non-empty subtrees T_1, T_2, ... T_k, each of whose roots are connected by a directed edge from r
- Every node in a tree is the root of a subtree
- Each node of the tree, different from the root, has a unique parent node



Generic tree

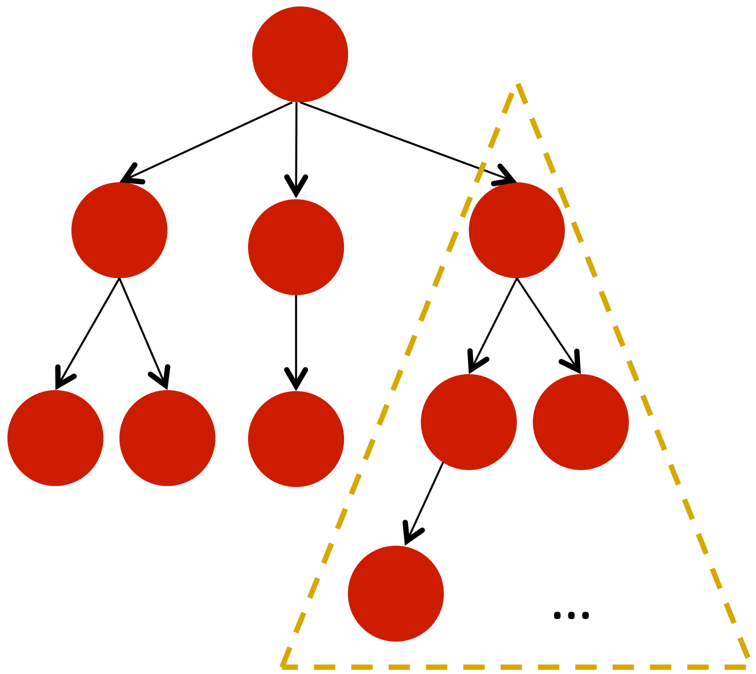
Tree Terminology

- Tree = Set of nodes connected by arcs (or edges)
- Every tree has a single root node – node without parent node
- A parent node points to (one or more) other nodes
- Nodes pointed to are children
- Every node (except the root) has exactly one parent
- Nodes with no children are leaf nodes
- Nodes with children are interior nodes



Tree Terminology

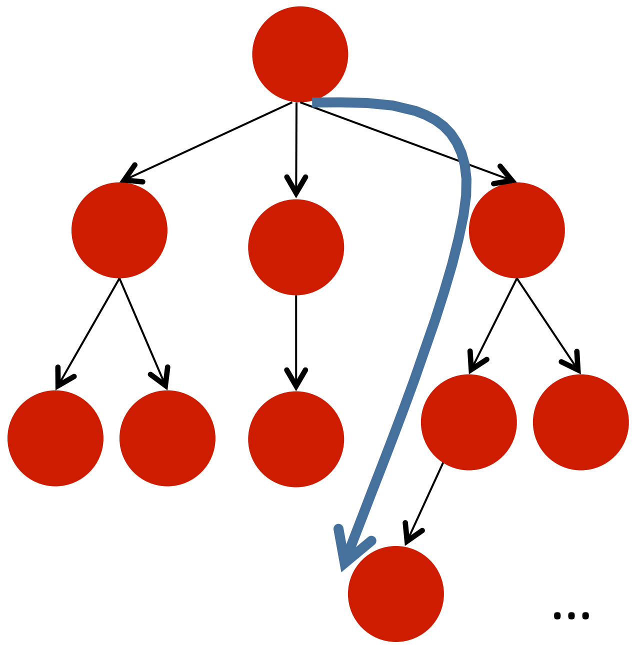
- Any node can be considered the root of a subtree



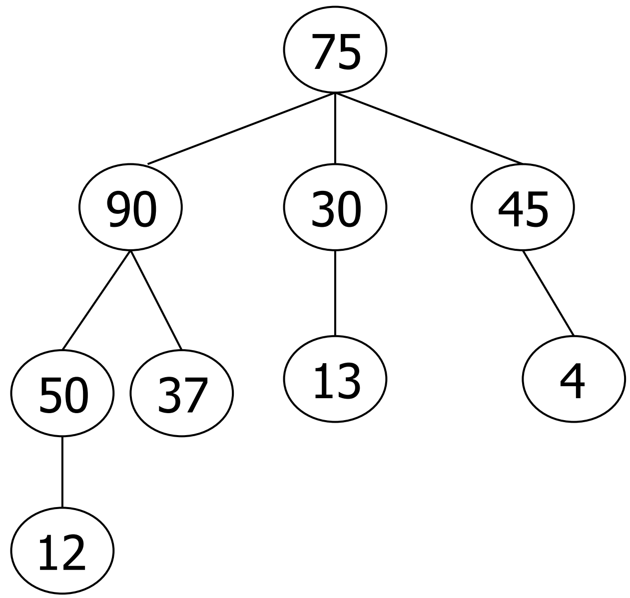
How many subtrees are there?

Tree Terminology

- There is a single, unique path from the root to any node
- A path's length is equal to the number of arcs traversed



Ascendants and descendants of a Node

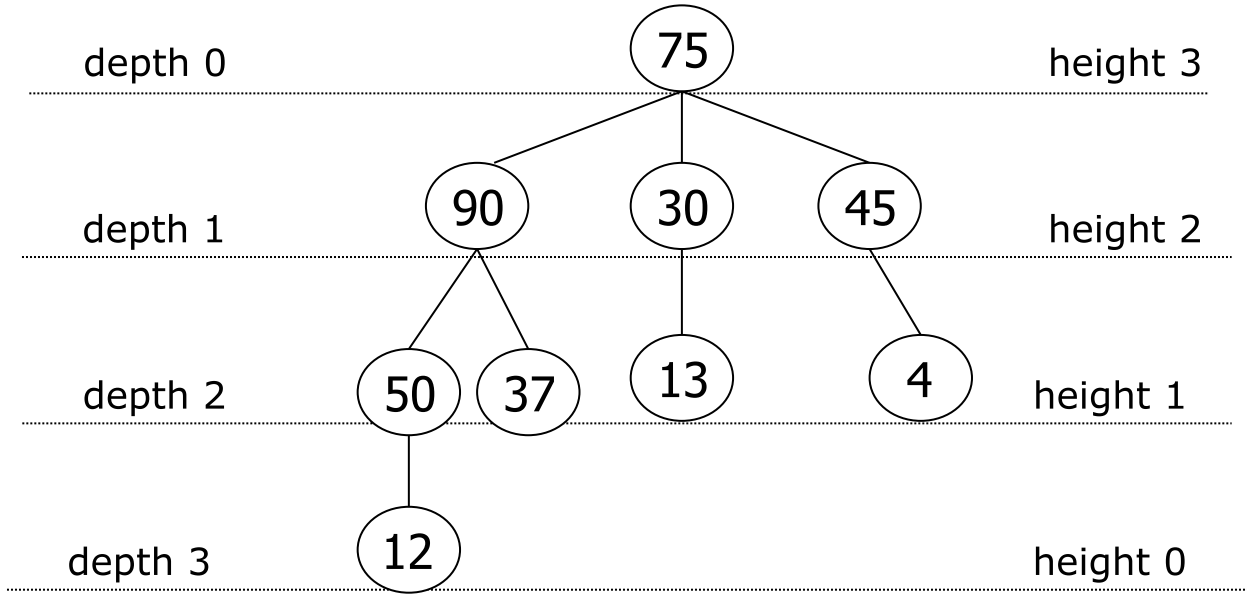


- The ascendants of a node are the nodes that are in the path from the node to the root of the tree. Ascendants of node 12: 50, 90, 75
- The descendants of a node are all the nodes reachable from that node. Descendants of node 90: 50, 37, 12
- All nodes in a tree are descendants of the root

  (except for the root)

Height and depth of a tree

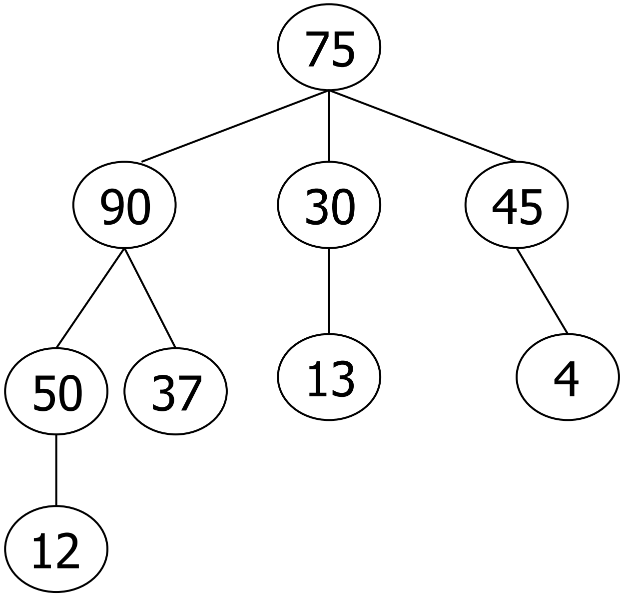
- Height of a node = max. path length from the node to a leaf
  - Height of a leaf node = 0
  - Height of the tree = Height of the root
- Depth of a node = path length from the root to that node
  - Depth of the root = 0



Degree of a tree

- Degree of a tree is the maximum degree of its nodes

- Degree of a node is equal the number of its children's



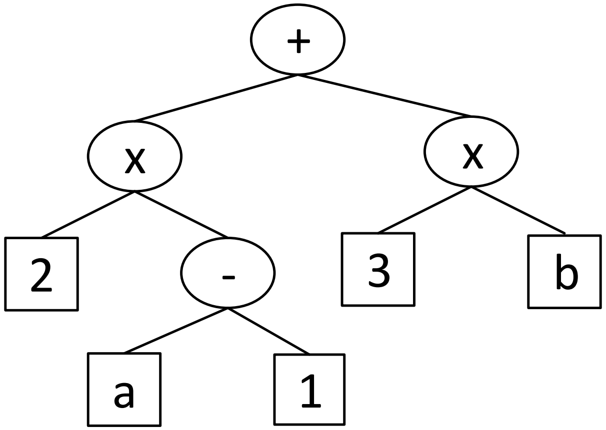
Degree of node 90: 2
Degree of a leaf node: 0
Degree of the tree: 3

Binary Trees

Binary Tree – Definition

- A binary tree is a special case of a K-ary tree whose nodes have exactly two child references
- A binary tree is a rooted tree in which no node can have more than two children AND the children are distinguished as left and right

Applications:
- Arithmetic expressions
- Decision processes
- Searching
- ....



Arithmetic expression: ((2 x (a - 1)) + (3 x b))

Binary Tree – Properties

In a binary tree

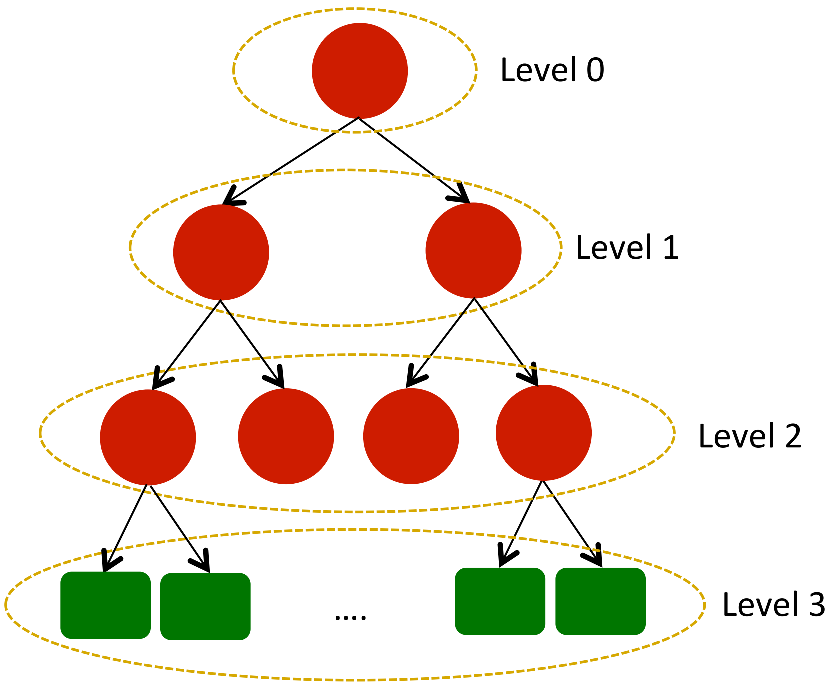
- level 0 has at most 1 = 2^0 node
- level 1 has at most 2 = 2^1 nodes
- level 2 has at most 4 = 2^2 nodes

…

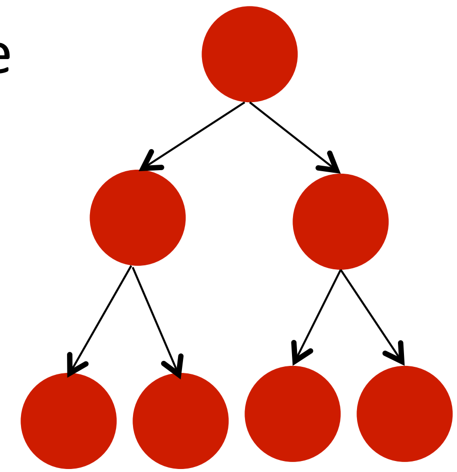
- level d has at most 2^d nodes

A binary tree of height h has:

- minimum: h + 1 nodes
- maximum: $2^{h+1} - 1$ nodes



Binary Tree – Properties



A full binary tree is a binary tree in which every node is a leaf or has exactly two children

- A full binary tree with n internal nodes has n + 1 leaves

A perfect binary tree is a full binary tree in which all leaves have the same depth

- The number of nodes in a perfect binary tree is $2^{h+1} - 1$ nodes, where h is height

$$n = 2^{h+1} - 1$$
$$2^{h+1} = n + 1$$
$$\log_2 (2^{h+1}) = \log_2 (n + 1)$$
$$h = \log_2 (n + 1) - 1$$

Tree Traversals

Depth-First Traversals

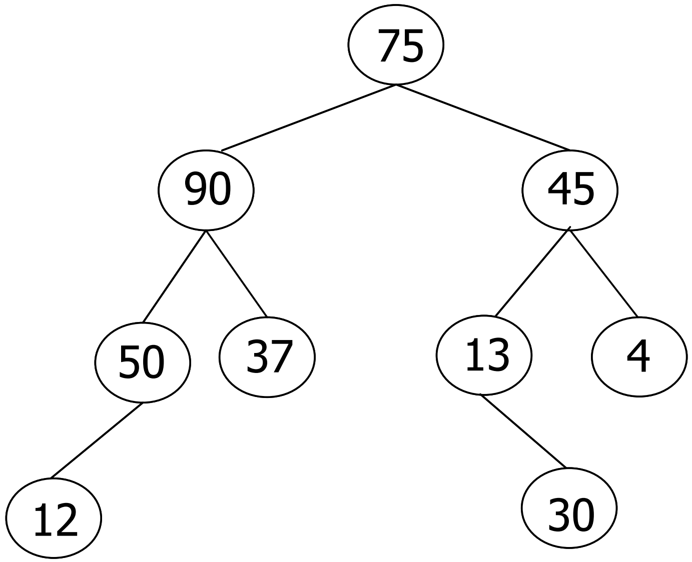
- Pre-order – root, left subtree, right subtree
- In-order – left subtree, root, right subtree
- Pos-order – left subtree, right subtree, root

Breadth-First Traversal

- Level-order – all the positions at depth *d are visited* before the positions at depth $d + 1$

- A breadth-first traversal is a common approach used in software for playing games

Tree Traversals - Exemplification



Depth-First

- Pre-order 75, 90, 50, 12, 37, 45, 13, 30, 4
- In-order 12, 50, 90, 37, 75, 13, 30, 45, 4
- Pos-order 12, 50, 37, 90, 30, 13, 4, 45, 75

Breath-First

- Level-order 75, 90, 45, 50, 37, 13, 4, 12, 30

Pre-Order Traversal

In a preorder traversal the node is visited before both its subtrees, left and right

```
Algorithm void preOrder(Node<E> node){

    if (node == null)
        return;

    visit(node)
    preOrder(node.getLeft())
    preOrder(node.getRight())
}
```

Time Complexity: O(?)

If visit(node) is O(1), then the complexity of preOrder is O(n)

Pre-Order Traversal – iterative algorithm

The iterative algorithm needs an auxiliary stack

```
Algorithm void IterativpreOrder(Node<E> node) {

   r = node
   do {
      while (r != null){
         visit(r)
         stk.push(r)
         r=r.getLeft()
      }
      if (!stk.isEmpty()){
         stk.pop()
         r=r.getRight()
      }
   } while (stk.isEmpty() && r != null)
}
```

In-Order Traversal

In an in-order traversal a node is visited after its left subtree and before its right subtree

```
Algorithm void inOrder(Node<E> node){

  if (node == null)
     return;
  inOrder(node.getLeft())
  visit(node)
  inOrder(node.getRight())
}
```

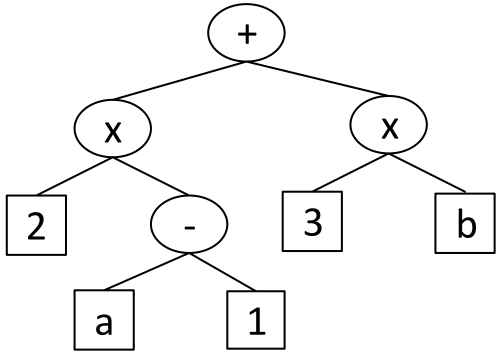
Specialization of In-Order Traversal:

writing of an arithmetic expression

Specialization of In-Order Traversal

Write an Arithmetic Expression

- write operand or operator when visiting node
- write "(" before traversing left subtree
- write ")" after traversing right subtree



((2 x (a - 1))+(3 x b))

```
Algorithm void  writeExpression(Node<String> node, String str){

   if (node.getLeft()){
      str += "("
      writeExpression(node.getLeft(),str)
   }
   str += node.getElement()
   if (node.getRight()){
      writeExpression(node.getRight(),str)
      str += ")"
   }
}
```

Pos-Order Traversal

In a pos-order traversal a node is visited after both its subtrees, left and right

```
Algorithm void posOrder(Node<E> node){

  if (node == null)
     return;

  posOrder(node.getLeft())
  posOrder(node.getRight())
  visit(node)
}
```

Iterative Algorithm: needs two stacks

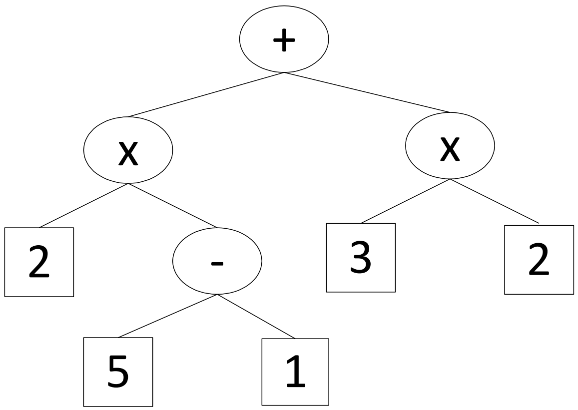
Specialization of a Pos-Order Traversal:

evaluation an arithmetic expression

Specialization of pos-Order Traversal

Evaluate an Arithmetic Expression

- recursive method returning the value of a subtree
- when visiting an internal node, combine the values of the subtrees



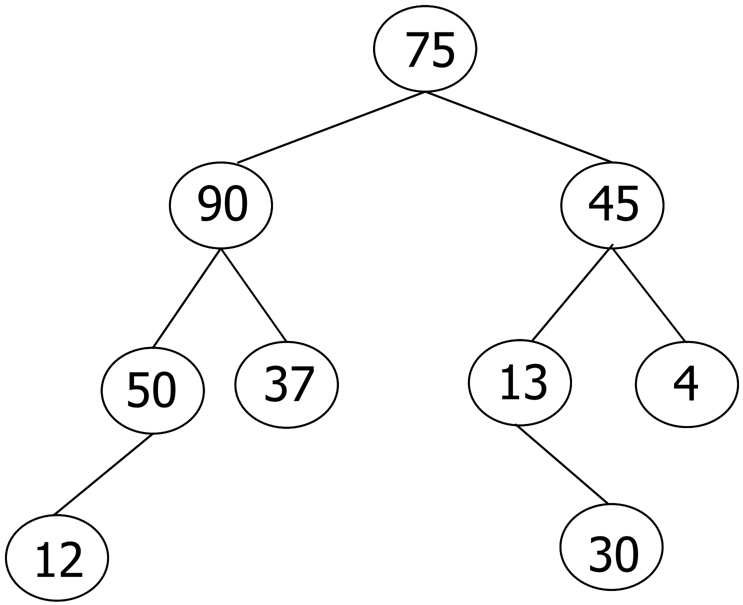

```
Algorithm double evalAritExpr(Node<String> node){

   if (node.getLeft() == null && node.getRight() == null)
      return node.getElement()
   else {
      x ← evalAritExpr(node.getLeft())
      y ← evalAritExpr(node.getRight())
      operator ← node.getElement()
      return makeOperation(x,y,operator);
   }
}
```

Breadth First Traversal

In a breadth first traversal the nodes of the tree are visited level by level

```
Algorithm void breadthfirst (){
   Initialize queue Q to contain root()
   while (Q not empty) {
      p = Q.dequeue()
      visit(p)
      for (each child c in children(p))
         Q.enqueue(c)
   }
}
```

Search an Element



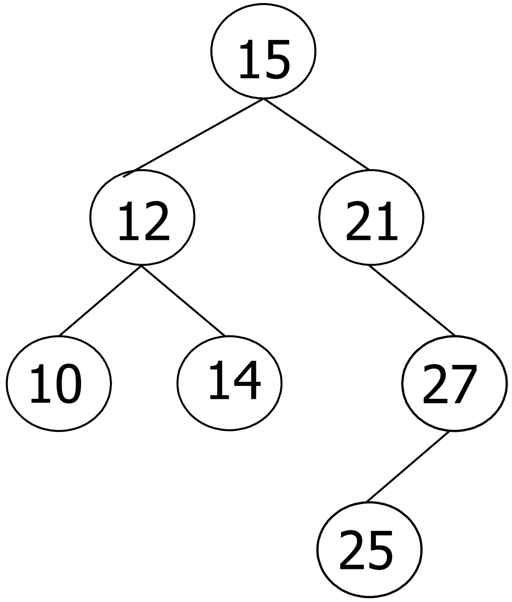
Time Complexity: O(?)

Binary Search Trees

Binary Search Tree (BST)

Is a binary tree where every node value is:

- **Greater than** all its left descendants
- **Less than** to all its right descendants



The elements in the BST must be comparable

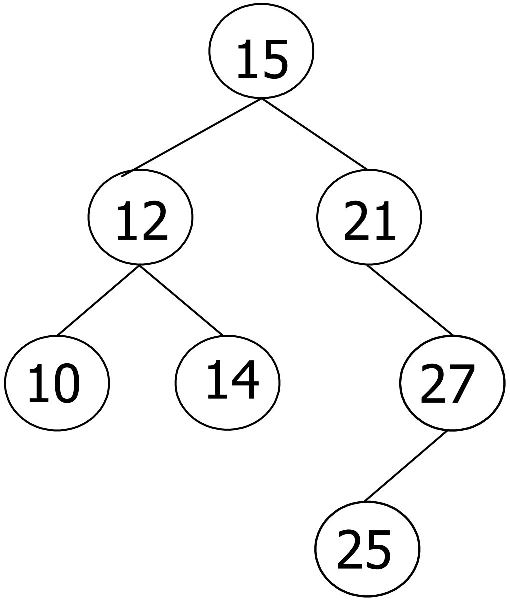
Duplicates are not allowed

Each subtree of a BST is also a BST

Applications:

- Symbol tables in compilers, "assemblers“
- Used in implementing efficient priority-queues (heaps)
- ….

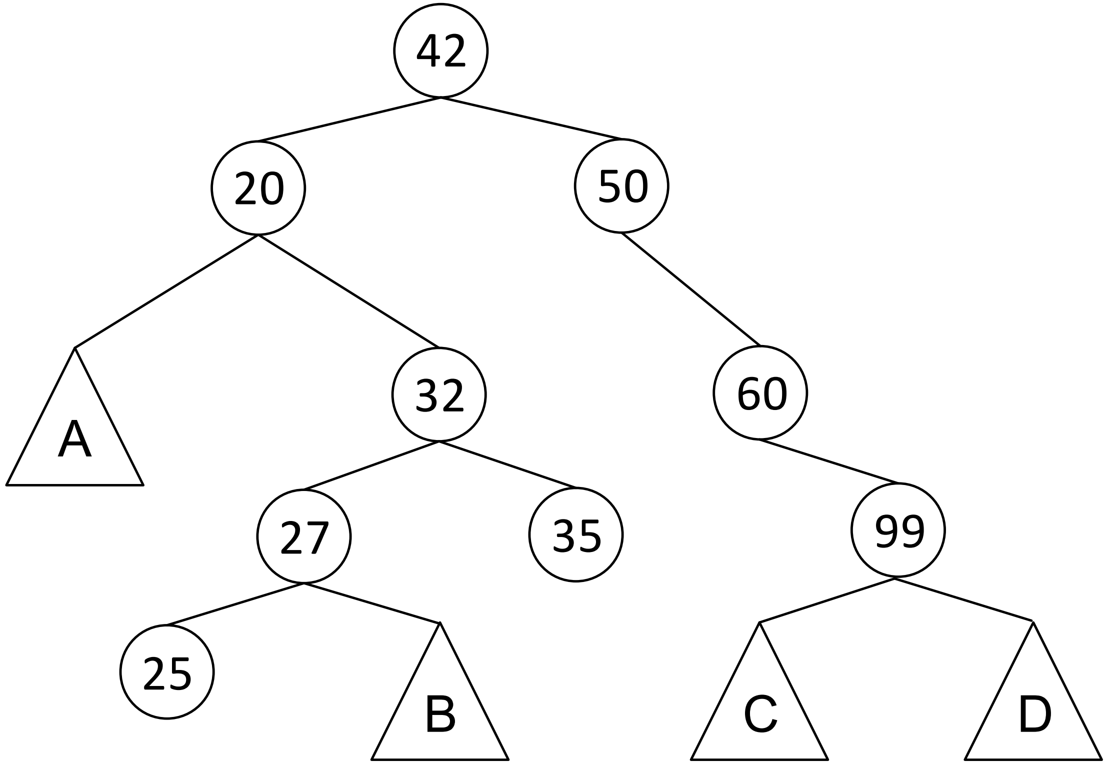
Binary Search Tree



- In-order: 10, 12, 14, 15, 21, 25, 27
- Pre-order: 15, 12, 10, 14, 21, 27, 25
- Pos-order: 10, 14, 12, 25, 27, 21, 15
- Level-order: 15, 12, 21, 10, 14, 27, 25

BST in-order traversal returns elements in sorted order

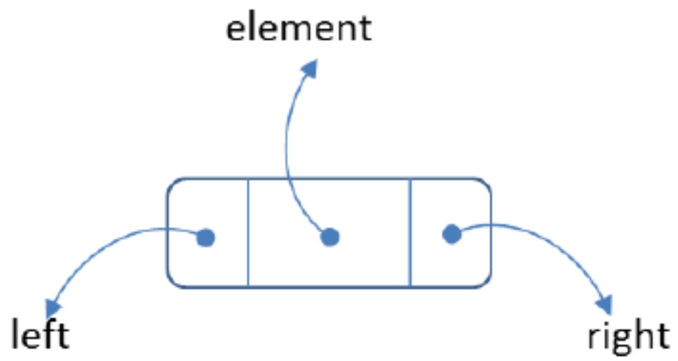
A Binary Search Tree of Integers



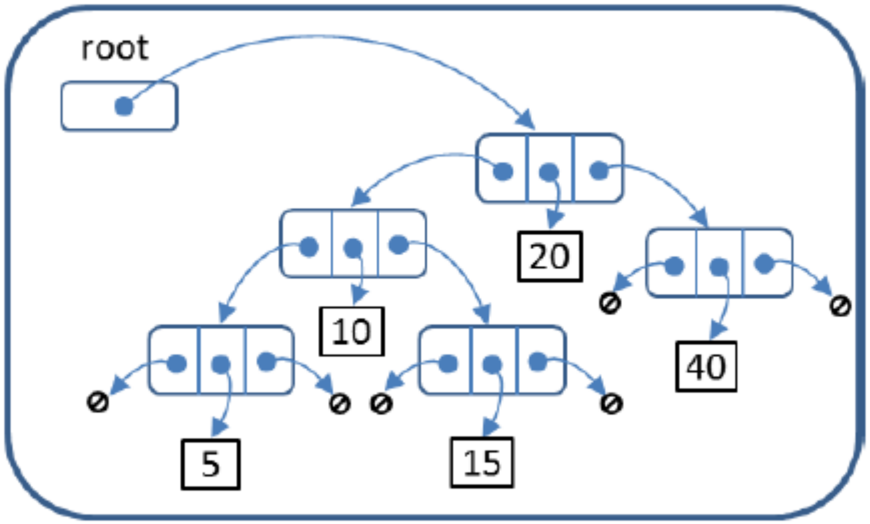
Describe the values which might appear in the subtrees labeled A, B, C, and D

Binary Search Tree ADT

Node

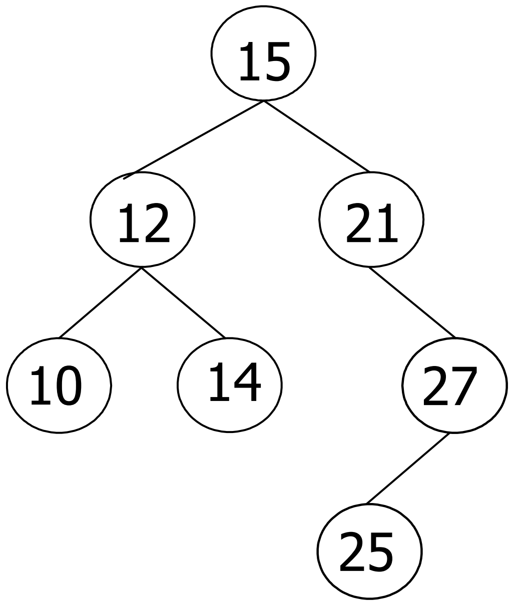


BST




```
public BST()

public boolean isEmpty()

public int size()

public void insert(E element)

public void remove(E element)
```

Search for an Element



- Start at root
- At each node, compare value to node value:
  - Return true if match
  - If value is less than node value, go to left child (and repeat)
  - If value is greater than node value, go to right child (and repeat)
  - If node is null, return false

Time Complexity

The maximum number of comparisons to conclude whether or not the key is in the tree is the maximum height of tree: $h + 1$

If the tree is (more or less) balanced, all the leaf nodes with the same depth, the height of the tree can be relate with the total number of elements n

$$n = 2^{(h+1)} - 1$$
$$2^{(h+1)} = n + 1$$
$$h+1 = \log_2 (n+1)$$
$$h = \log_2 (n+1) - 1$$

For all values of $n \geq 1$, there is a constant C, such that:

$$\log_2 (n+1) - 1 \leq C \times \log_2 n$$

$T(n) = O(\log n)$

Search for an Element

```
Algorithm Node<E> search(Node<E> node, E elem){
   if (node == null)
      return null
   if (node.getElement() == elem)
      return node
   if (node.getElement() > elem)
      return search(node.getLeft(),elem)
   else
       return search(node.getRight(),elem)
}
```

Time Complexity: O(?)

- **Best case**: the element is at the root
- **Average Case**: the tree is balanced
- **Worst Case**: the element doesn't exist, the tree degenerates in a list

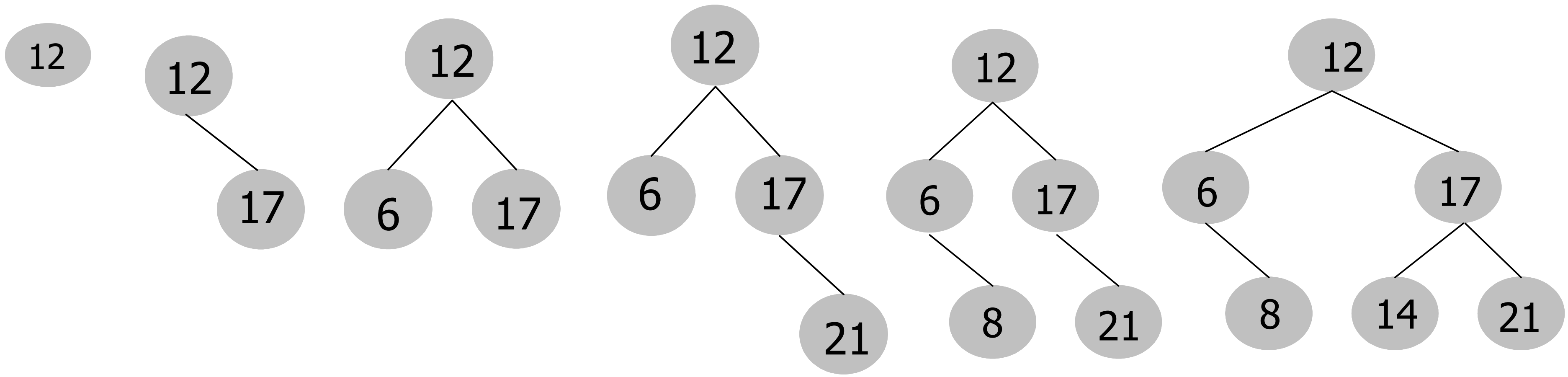
Search for an Element – iterative version

```
Algorithm boolean search(E elem) {
  node = root
  find = false
  while (node != null && !find){
     if (node.getElement() == elem)
        find = true
     if (node.getElement() > elem)
        node = node.getLeft()
     if (node.getElement() < elem)
       node = node.getRight()
  }
  return find
}
```

Insertion

- Start at the root
- successively down the tree from the root choosing the appropriate sub-tree
- Arriving in a leaf, insert in the appropriate side

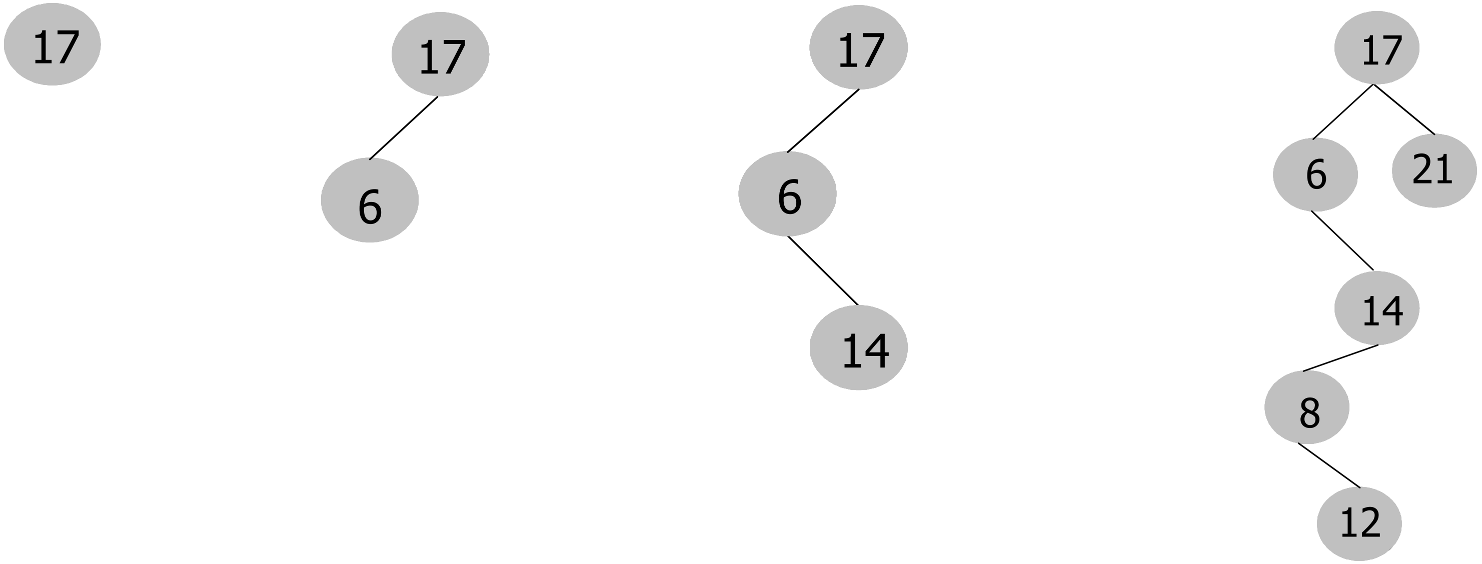
  The shape of the tree depends on the order of elements insertion:
  12, 17, 6, 21, 8, 14



Insertion

The shape of the tree depends on the order of elements insertion:

17, 6, 14, 21, 8, 12



What happens if the elements are inserted into the tree in ascending or descending order?

Insertion

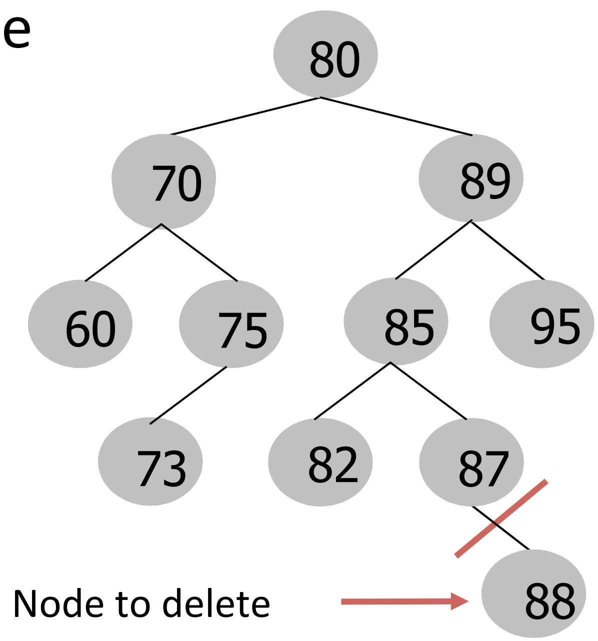
```
Algorithm Node<E> insert(Node<E> node, E elem){

    if (node == null)
       return new Node(element, null, null)

    if (node.getElement() == element)
        node.setElement(element)
    else
        if (node.getElement() > elem)
           node.setLeft(insert(node.getLeft(),elem))
        else
            node.setRight(insert(node.getRight(),elem))
    return node
}
```

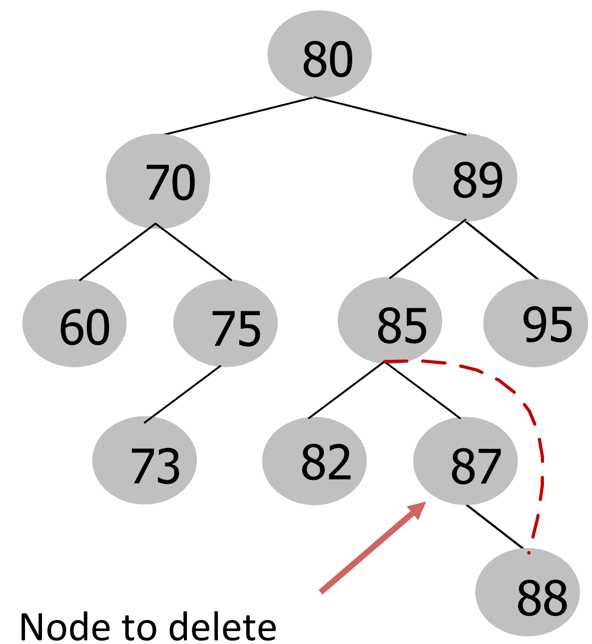
Deletion

When delete a node three cases can happen:

1. the node is a leaf (it hasn't subtrees)
2. the node has only one subtree
3. the node contains two subtrees (left and right)

The first two cases (1 and 2) are solved by adjusting the pointer of the previous node (parent node) that points to the node we want to eliminate





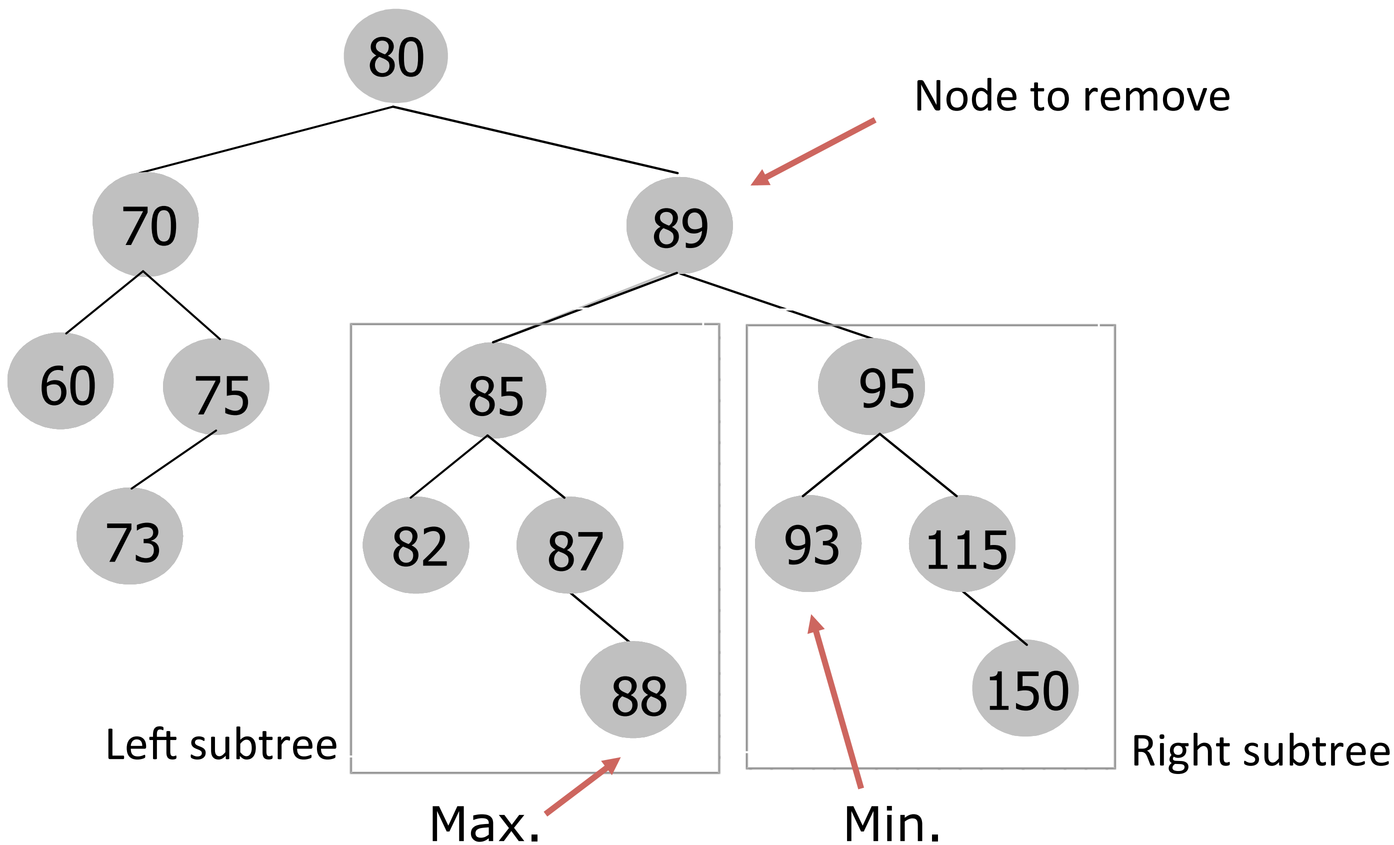
Deletion

Case 3:

- replace the node to eliminate with the greatest node of the left subtree of the node to delete

or

- replace the node to eliminate with the smaller node of the right subtree of the node to eliminate



Deletion

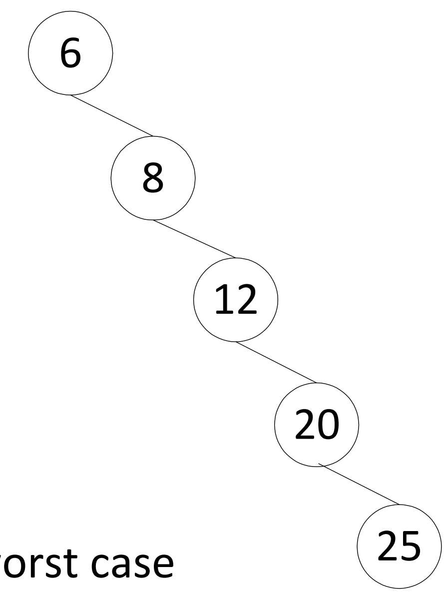
```
Algorithm Node<E> remove(E elem, Node<E> node) {
    if (node == null)
       return null

    if (node.getElement() == elem) {

        if (node.getLeft() == null && node.getRight()== null)
           return null

        if (node.getLeft() == null)
           return node.getRight()

        if (node.getRight() == null)
           return node.getLeft()

        E min = smallestElement(node.getRight())
        node.setElement(min)
        node.setRight(remove(min, node.getRight()))  }
    else if (node.getElement() > elem)
         node.setLeft(remove(elem,node.getLeft()))
    else
         node.setRight(remove(elem,node.getRight()))
    return node }
```

Performance BST methods

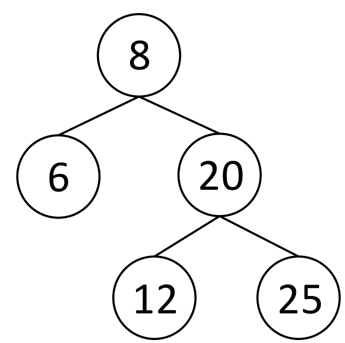
The analysis of search, insert and remove is similar

- In each case, h nodes are visited
- If each node is visited at O(1)
- The methods take O(h) time

The height h is O(n) in the worst case and O(log n) in the best case



worst case



best case

To make sure height h of a tree is always O(log n), the tree must be balanced

Balanced Trees or AVL Trees

Balanced Trees

- In a balanced tree for all of its nodes the height of the left subtree is approximately equal to the height of the right subtree, which guarantees that the height of the tree is always O(logn)

- This is achieved by an extra processing cost on the construction of the tree to maintain it balanced, but this is compensated when the data is often retrieved

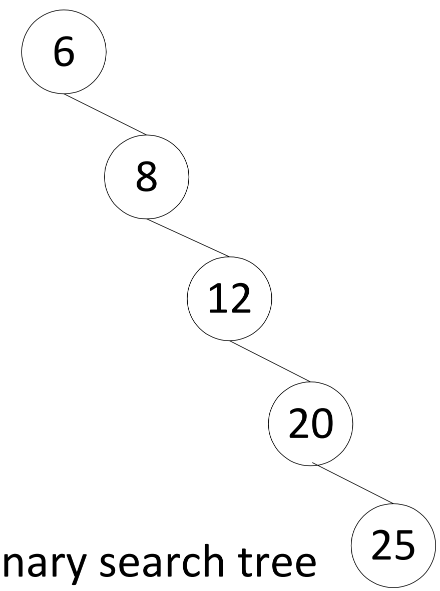
- The idea of maintaining a balanced binary tree dynamically i.e., as nodes are inserted/removed, was proposed in 1962 by two Soviet called Adelson-Velskii and Landis - AVL tree

AVL Tree - Definition

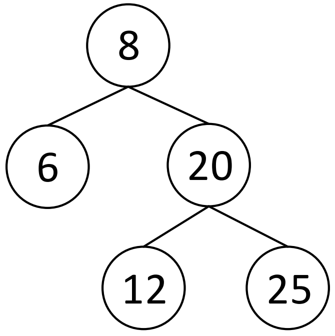
An AVL tree is a binary search tree such that for every internal node the heights of its children trees can differ by at most 1

Thus, each node has a Balance Factor (BF)

BF (node) = height (right subtree) - height (left subtree)



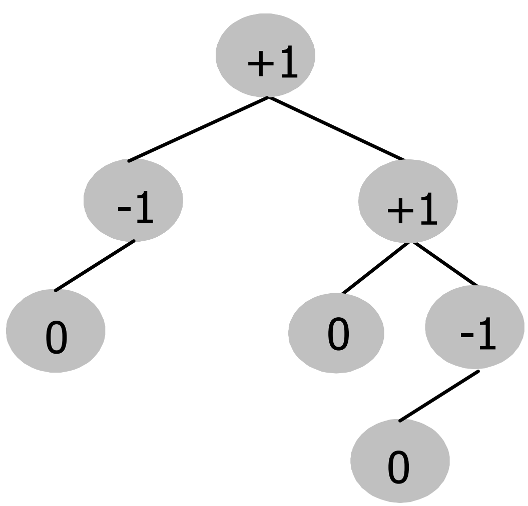
Binary search tree



AVL tree

Balance Factor (BF)

- Negative balance factor of a node means that the height of its left subtree is larger (in at least one node) than the height of its right subtree, left node heavy
- Positive balance factor of a node means that the height of its right subtree is larger (in at least one node) than the height of its left subtree, right node heavy
- Null balance factor of a node means that the height of the left subtree is equal to the height of the right subtree - node balanced

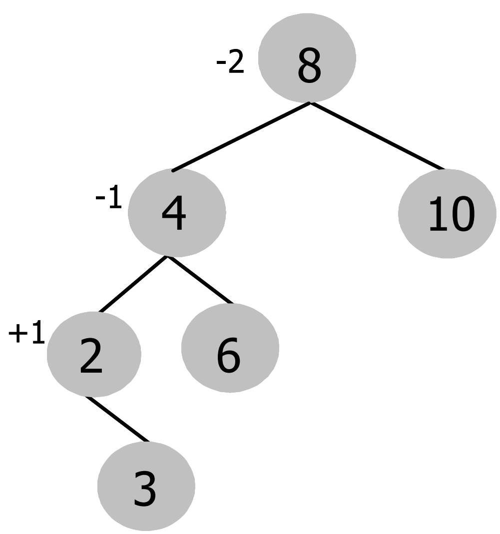


Balancing the Tree

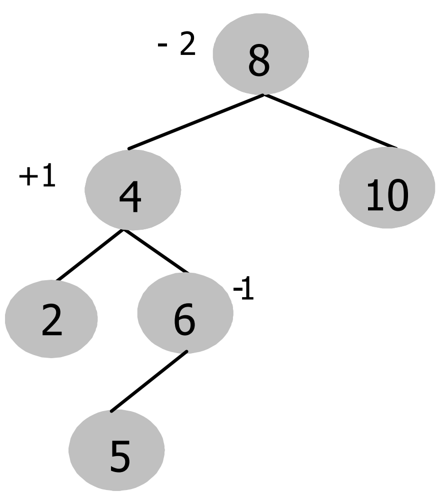
It is necessary whenever the insertion/removal of a node violates the tree balancing property: nodes in the tree with BF $\notin$ [-1,..,1]

The balancing of the tree is achieved with two kind of rotations:

- Simple - when the unbalanced node presents the same BF signal as its child's root node of unbalanced subtree
- Double - when the unbalanced node presents a BF signal contrary to its child's root node unbalanced subtree



Simple rotation

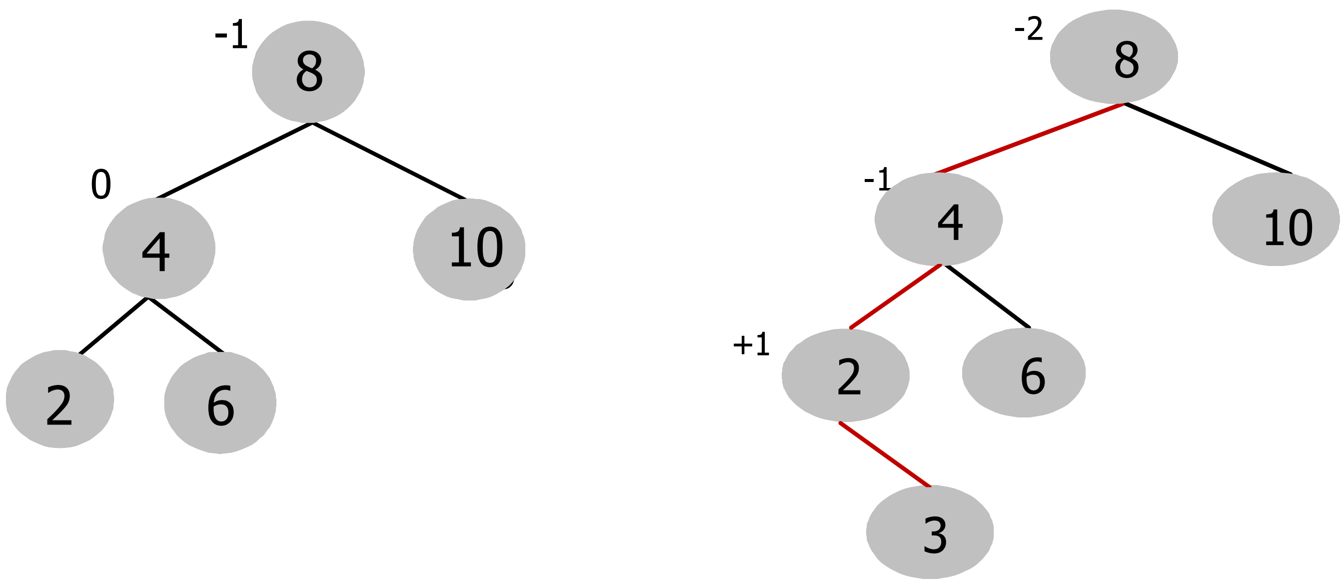


Double rotation

Balancing the Tree

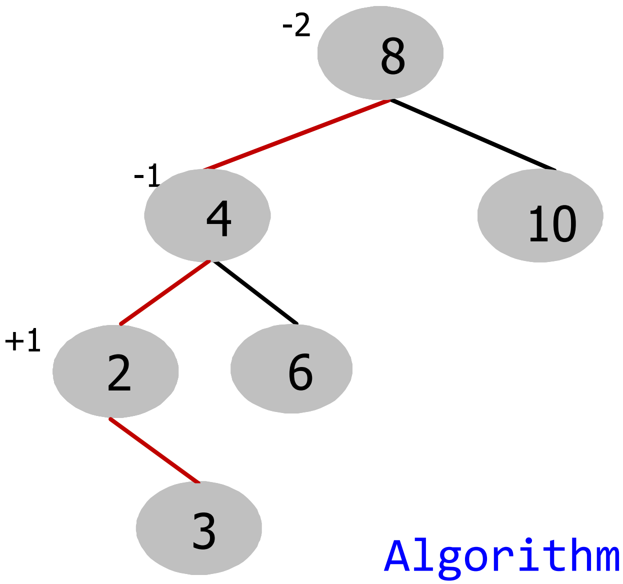
There is a very important property in binary search trees:

- after insertion/removal of a node it is warranted that only the nodes that are on the path between the root and the element inserted/ removed can become imbalanced
- So, the balancing operations are only necessary on the nodes that are on that path



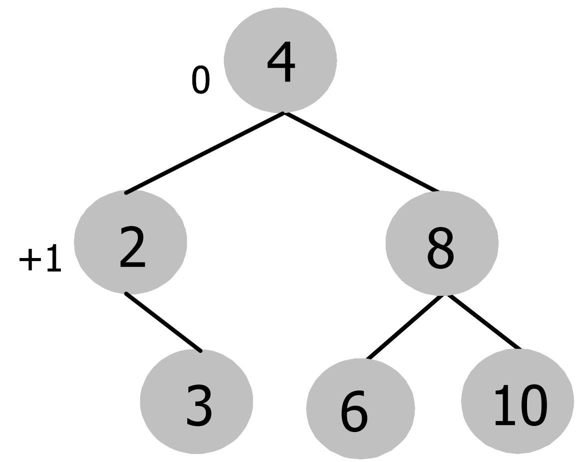
Simple right rotation

When the rotation is simple, it occurs always in the opposite direction of the tree imbalance



Simple right rotation

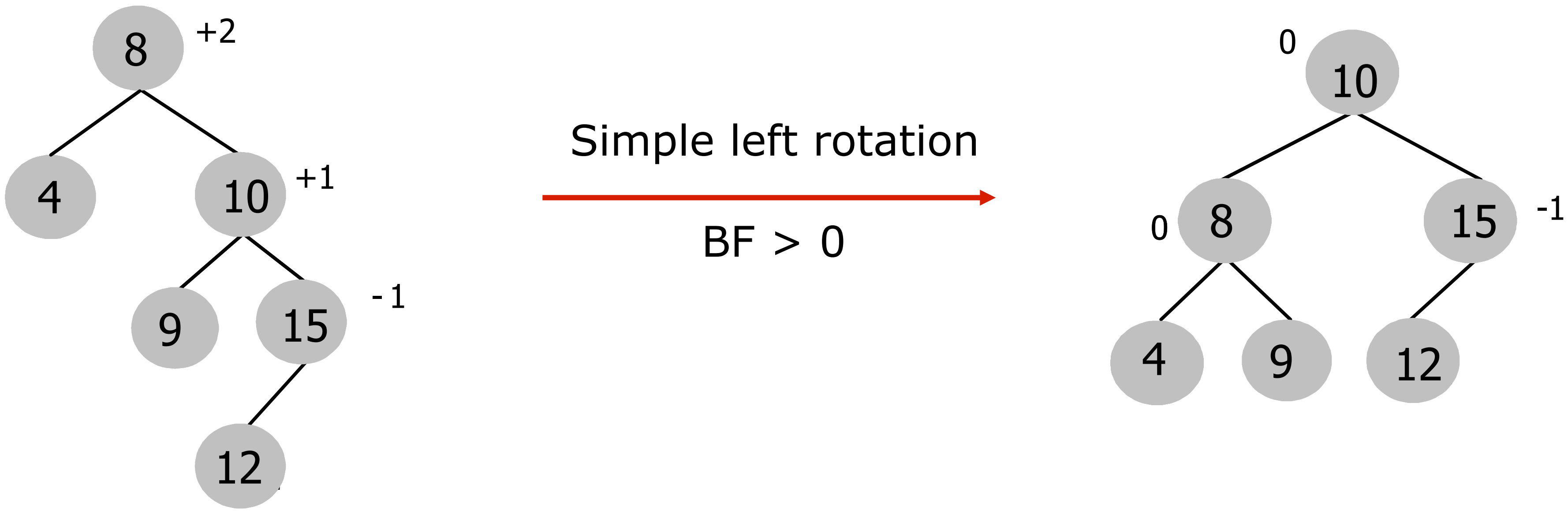
BF< 0




```
Algorithm Node<E> rightRotation (Node<E> node){
        Node<E>  leftson = node.getLeft()
        node.setLeft(leftson.getRight())
        leftson.setRight(node);
        node = leftson
        return node
}
```

Complexity(?)

Simple left rotation

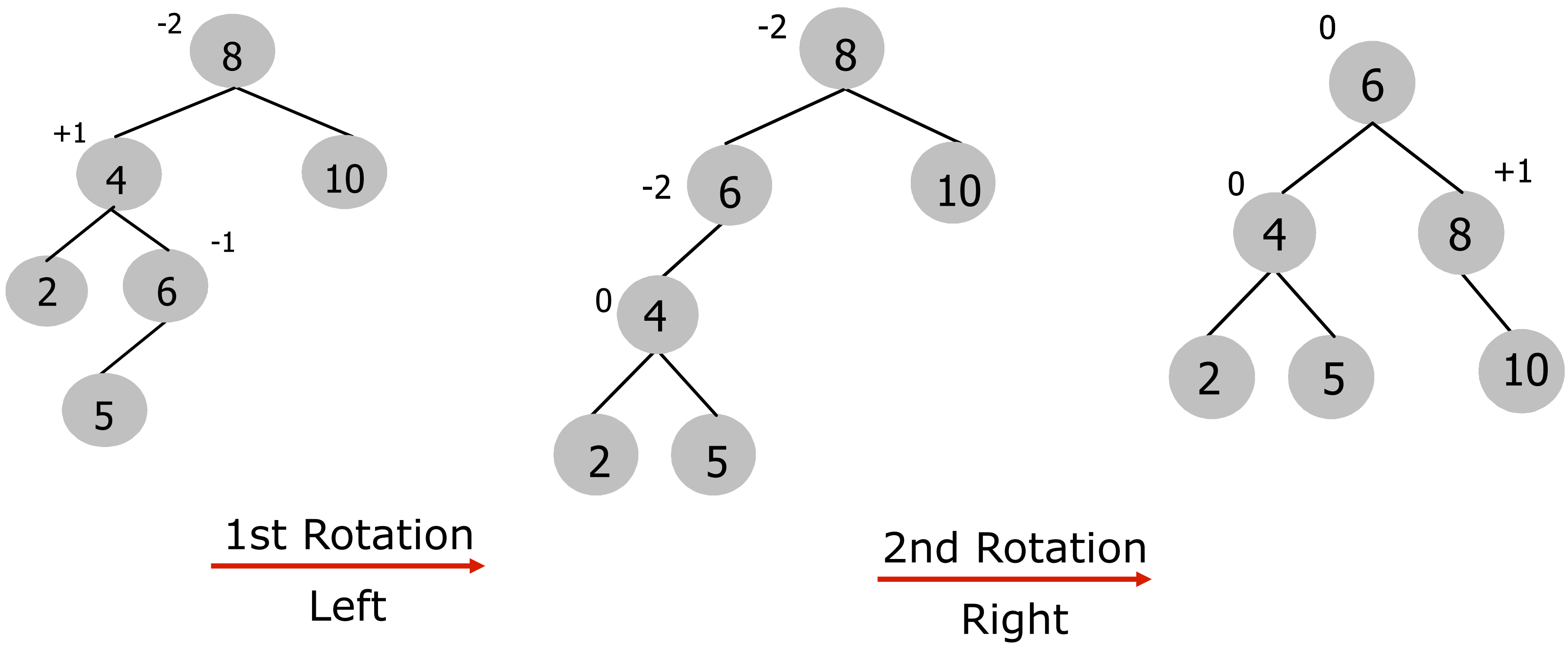



```
Algorithm Node<E> leftRotation (Node<E> node){
        Node<E>  rightson = node.getRight()
        node.setRight(rightson.getLeft())
        rightson.setLeft(node)
        node = rightson
        return node
}
```

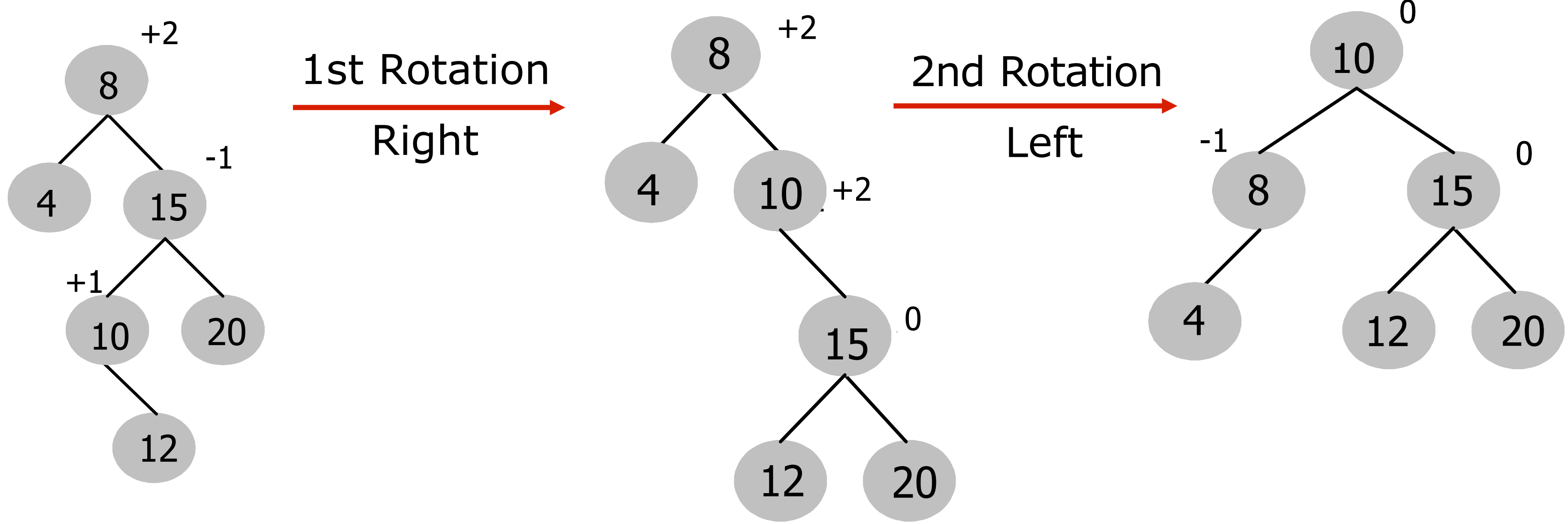
Complexity(?)

Double rotation (Left-Right)

- This case requires rotating the tree child in direction of the imbalance side
- The second rotation always occurs in the opposite direction of the first rotation



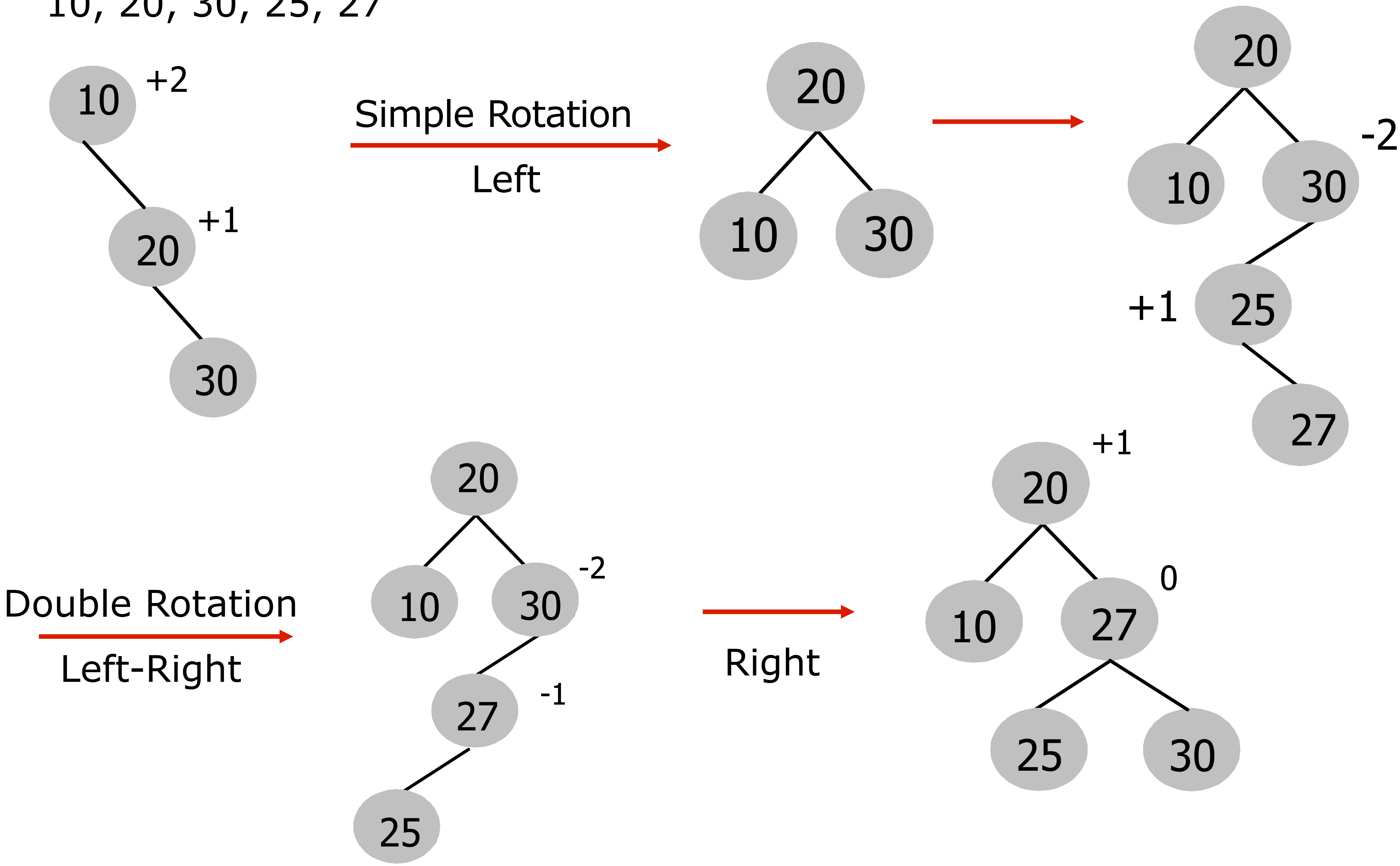
Double rotation (Right-Left)




```
Algorithm Node<E> twoRotations (Node<E> node) {
   if (balanceFactor(node) < 0)  {
      node.setLeft(leftRotation(node.getLeft()));
      node = rightRotation(node) }
   else {
      node.setRight(rightRotation(node.getRight()))
      node = leftRotation(node)  }
   return node
}
```

Insertion in an AVL tree

10, 20, 30, 25, 27

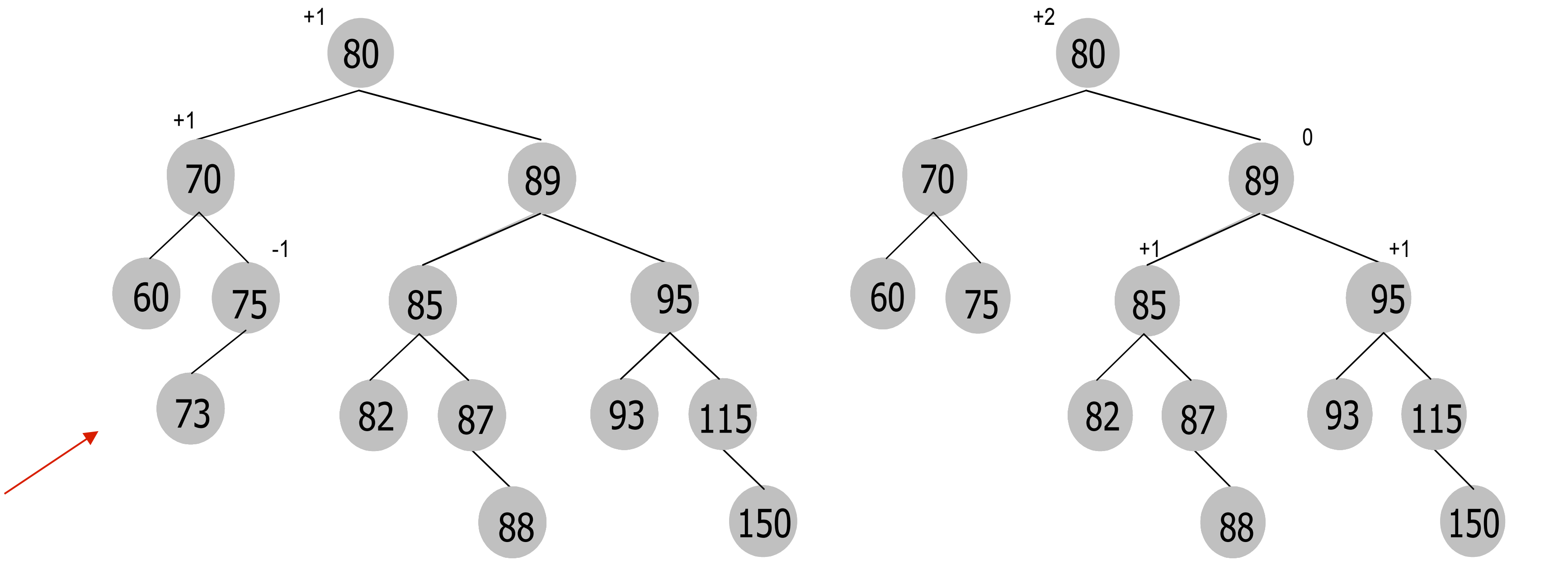


Insertion in an AVL tree

```
Algorithm Node<E> insert(Node<E> node, E elem) {
 if (node == null)
   return new Node(element, null, null)

 if (node.getElement() == element)
    node.setElement(element)
 else
    if (node.getElement() > elem) {
       node.setLeft(insert(node.getLeft(),elem))
       node = balanceNode(node)
    }
    else {
       node.setRight(insert(node.getRight(),elem))
       node = balanceNode(node)
    }
 return node
}
```

Deletion in an AVL tree

After removal of a node, the balance factor of each node on the path between the removed node and the root are recalculated and the necessary rotations are made



Deletion

```
Algorithm Node<E> remove (E elem, Node<E> node) {
    if (node == null)
       return null

    if (node.getElement() == elem) {
       if (node.getLeft() == null && node.getRight()== null)
          return null
       if (node.getLeft() == null)
          return node.getRight()
       if (node.getRight() == null)
          return node.getLeft()
       E smallElem = smallestElement(node.getRight())
       node.setElement(smallElem)
       node.setRight(remove(smallElem, node.getRight()))
       node = balanceNode(node)
    }
    else if (node.getElement() > elem) {
         node.setLeft(remove(elem,node.getLeft()))
         node = balanceNode(node)  }
    else  {
         node.setRight(remove(elem,node.getRight()))
         node = balanceNode(node)  }
    return node
}
```

AVL tree - Sorting

An AVL tree can be used to sort a collection of values:

1. Insert data into the AVL tree: O(??)
2. Copy data from AVL tree into the collection using the ?? traversal: O(??)

Execution time: O(nlog n)

- Matches that of quicksort in benchmarks
- Unlike quicksort, AVL trees don't have problems if data is already sorted or almost sorted (which degrades quicksort to $O(n^2)$)
- However, requires extra storage to maintain both the original data buffer and the tree structure

Binary search tree vs. AVL search tree

- On average 50% of insertions and deletions require rotations
- These lead to a loss of efficiency in the insertion and removal algorithms

Thus, the use of AVL or BST depends on the application:

- applications where the search is the dominant operation should use AVL trees, because they guarantee time complexity O(logn)
- applications where insertions or deletions are the most frequent operations should used binary search tree

B-Trees
Multiway search trees

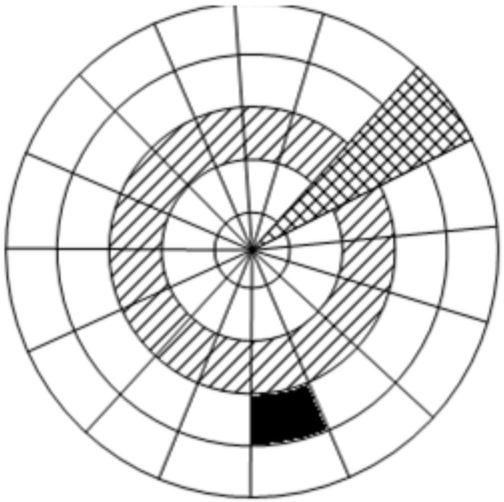
Motivation

- B-tree is a multiway search tree used when data must be stored on the disk, i.e. too large to fit in the memory

- When data is too large to fit in main memory, the number of disk accesses will dominate the running time

- A disk access is MUCH slower than a memory access (mechanical limitations)
  - 25 MIPS machine: 1 disk access cost as much as 250,000 instructions

Disk block



A disk is organized in the following subdivisions:

- tracks
- sectors
- Blocks

Because accessing disk is so slow, data on a disk is stored in blocks of size B

- typical block size is between [1024 … 8192] bytes
- a whole block is read per one disk access
- The basic I/O operation transfers the contents of one disk block to/from main memory

Disc Access time

Constant time memory access is only valid for RAM memories where access speed is limited only by electrical factors

Discs access speed is limited by mechanical factors. The access time to a specified disk location is greater than the time it takes to read the bytes stored in that disc location

- first it is necessary to position the head
  - it is essentially a mechanical and time-consuming operation
- from there the bytes are read in a rapid succession
  - using the disc rotation movement

Binary tree in disk ?!

To store a binary tree in a disk:

- each node contains only one information unit
- access to each node requires one access to the disc

Example: a binary tree with 1000 elements, $\log_2 1000 \approx 10 \rightarrow 10$ levels

1 search needs 10 disk accesses → 0.1 sec.

100 searches → 100 × 0.1 sec. = **10 sec.**

It is impractical to manipulate binary trees in disk as is done in memory

- It is necessary to devise a multiway search tree that minimize file accesses (by exploiting disk block read)

Extended node – M-ary node

As disk accesses are proportional to the height of the tree, the solution is to use a tree with high branching factor so that the height of the tree is smaller, thus requiring a smaller number of disc accesses

Extended nodes:

- access to each node allows access a great amount of information
- the tree has few levels
- requires a smaller number of disk accesses during search and change operations

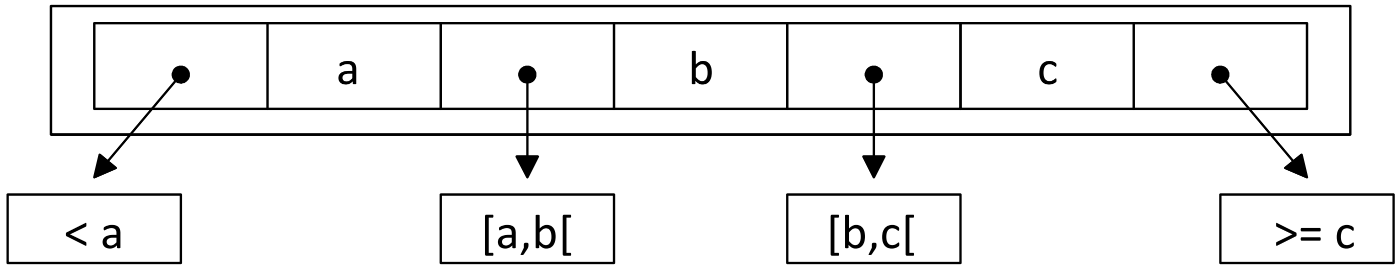
N-ary node

A N-ary node is a node of N order - the order of a node defines the maximum number of its descendants:

- a binary node contains one element and two direct descendants
- an N-ary node contains N-1 elements and N direct descendants

Data organization in a N-ary node

- the elements within the node are ordered
- the reference between two nodes A and B points to the sub-tree whose elements are between a and b



The node as data block

- A B-tree must have a high branching factor so that the height of the tree is smaller, thus requiring smaller number of disc accesses
- How to optimize the order of nodes?
- Should take into account the size of disk block

Example

size of block:	8192 bytes
size of elements	256 bytes
references	4 bytes

$4\,N + 256\,(N-1) \leq 8192 \quad \Rightarrow \quad N \leq 32{,}5$ $\rightarrow$ nodes of order 32

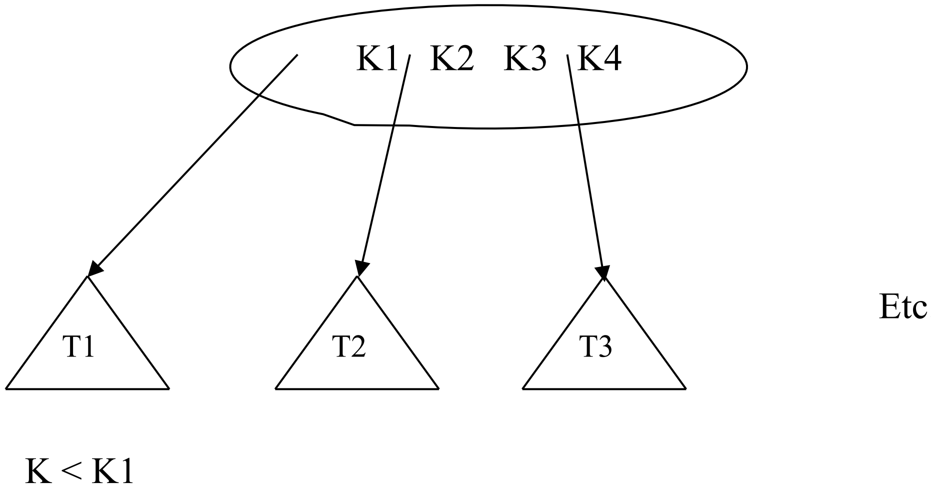
- Minimum limit:

  to avoid blocks with insufficient use

B-Tree Definition

A B-tree of order **M** is a multi-way search tree such that:

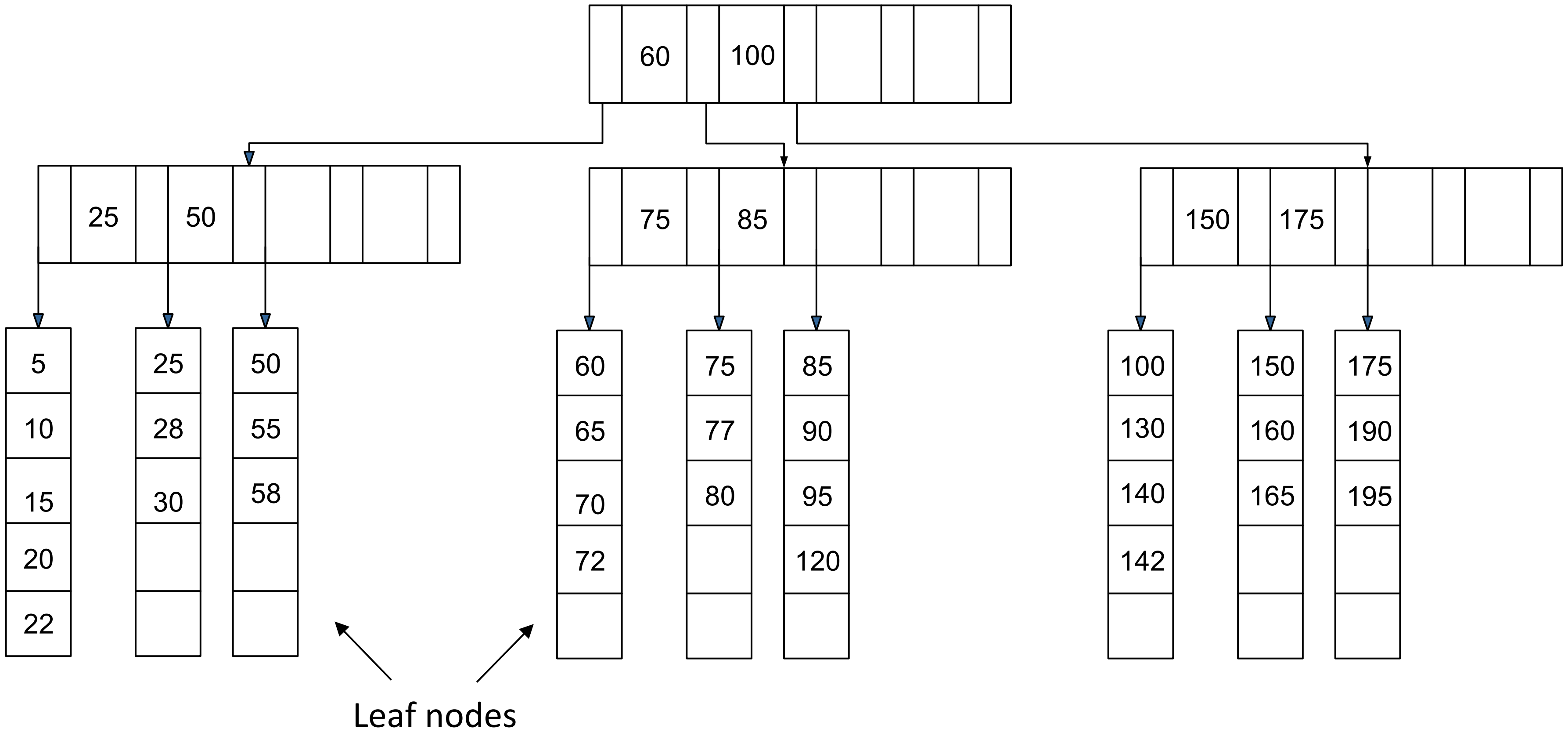
1. The root has between 2 and *M* children
2. other internal nodes have between $\lceil M/2 \rceil$ and M children
3. internal nodes contain only search keys (no data)
4. All leaves are at the same level – tree with perfect balance



Result

- tree is **O(log$_M$ N)** deep
- all operations run in **O(log$_M$ N)** time

B-Tree of order 5



- Minimum occupation ratio of a node: 50%
- The internal nodes have between 3-5 children

B-Tree Dynamics

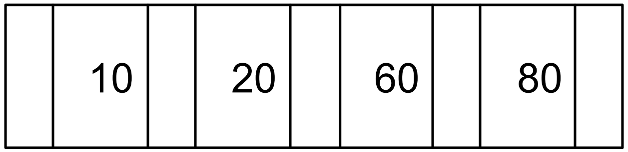
- When a node is full it has to be broken in two nodes
  - each half is a new node filled 50%
- When a node has less than 50% keys it must receive elements from its neighboring nodes or may be fused to another node
  - It must be ensure that any node is filled below 50%

As each node, except the root, has between M/2 and M elements:

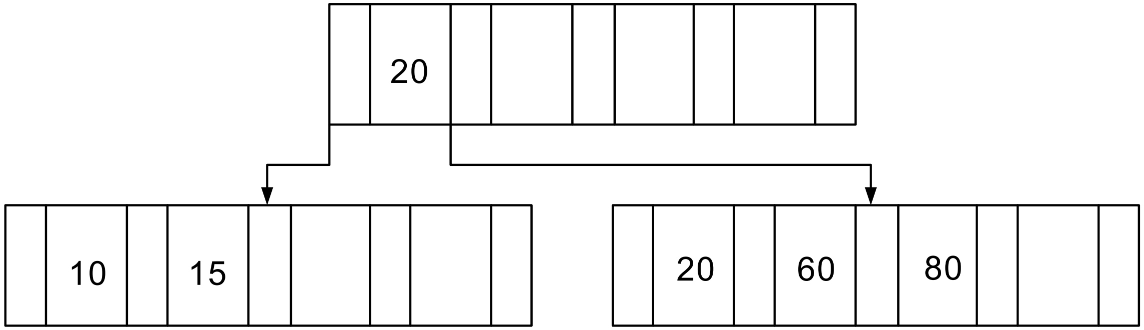
- most of the insertions will only filling incomplete nodes
- just a few will find nodes filled and force to create new nodes
- most of the removals will just empty nodes a bit more
- only a few operations will find nodes at the lower end of occupation and force the distribution of elements or fusion of nodes

B-Tree Insertion

Insert 60, 20, 80, 10 in a B-tree of order 5

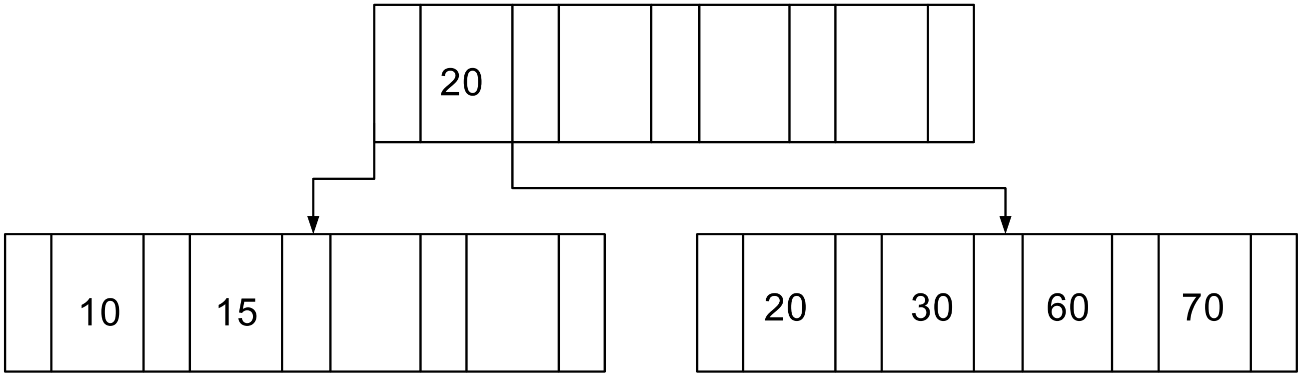


Insert 15

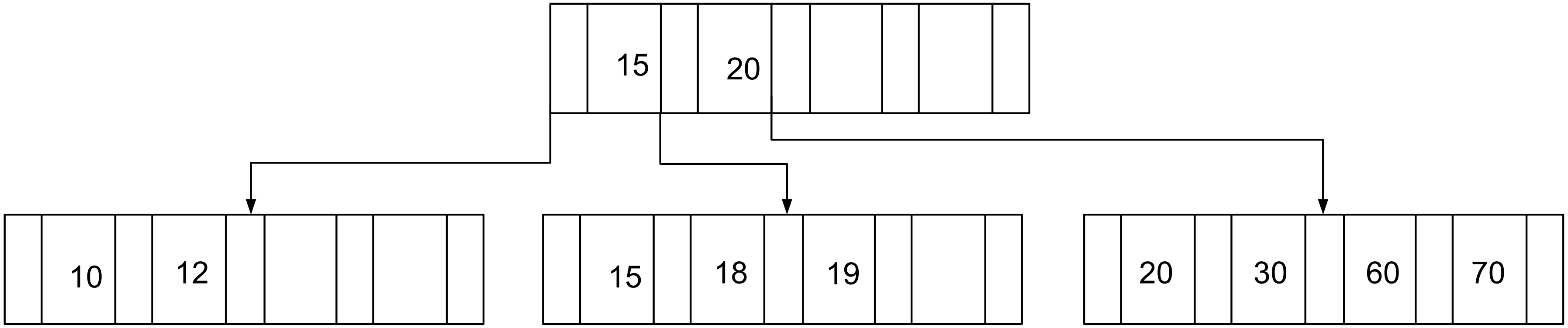


Insert 30

B-Tree Insertion

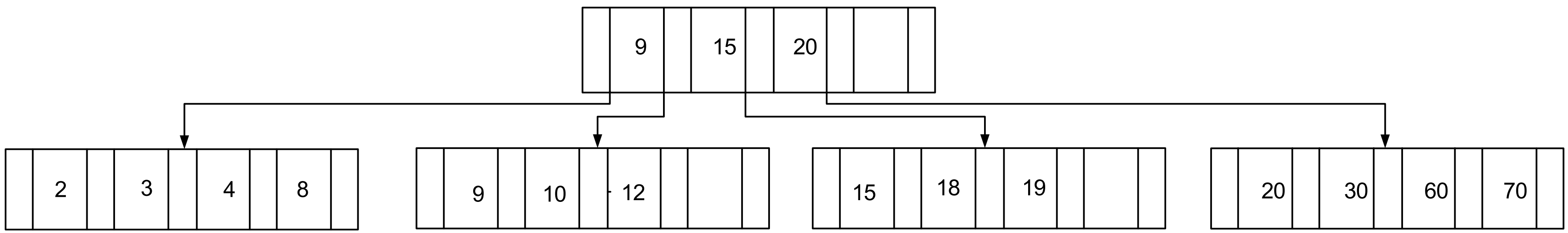


Insert 12, 18, 19

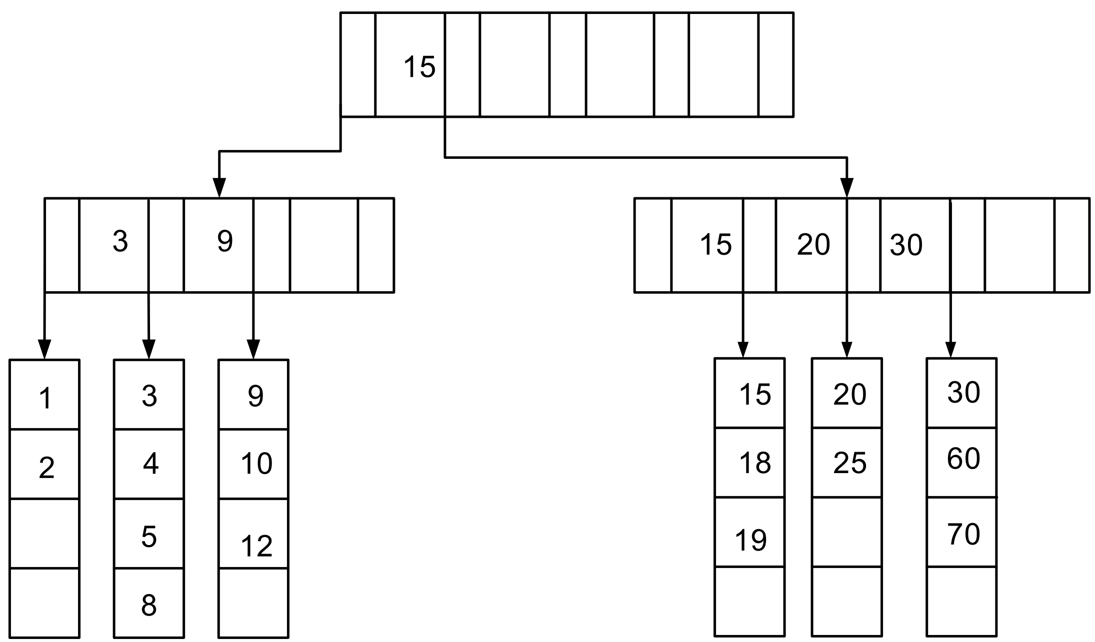


Insert 4,8,9,2,3

B-Tree Insertion



Insert 1,5,25

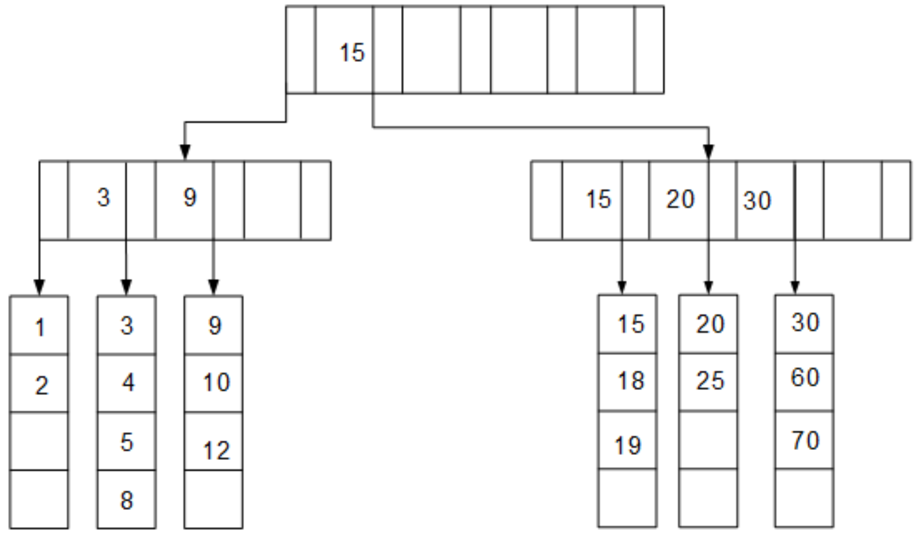


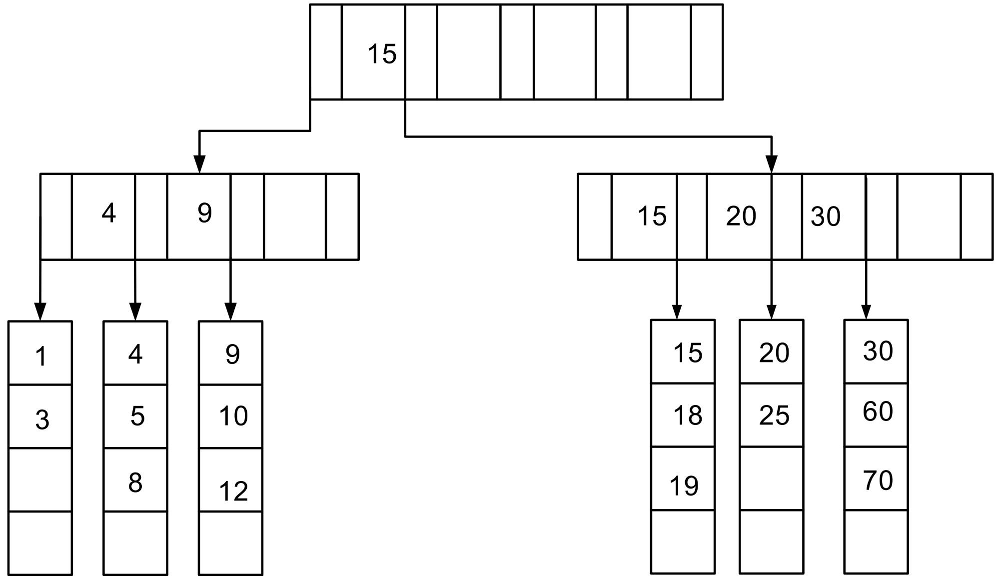
B-Tree Insertion

Leaf Node full	**Internal Node full**	
No	No	Put key in the respective leaf node (ordered position)
Yes	No	1. Split leaf node 2. Left leaf contains keys < middle-key 3. Right leaf contains key >= middle-key 4. Put the middle-key into the parent node (ordered position)
Yes	Yes	1. Split leaf node 2. Left leaf contains keys < middle-key 3. Right leaf contains key >= middle-key 4. Put the middle-key into the parent node (ordered position) 1. Split parent node 2. Left leaf contains keys < middle-key 3. Right leaf contains key >= middle-key 4. Put the middle-key into the parent node (ordered position) If node higher level is full continue to split

B-Tree Deletion

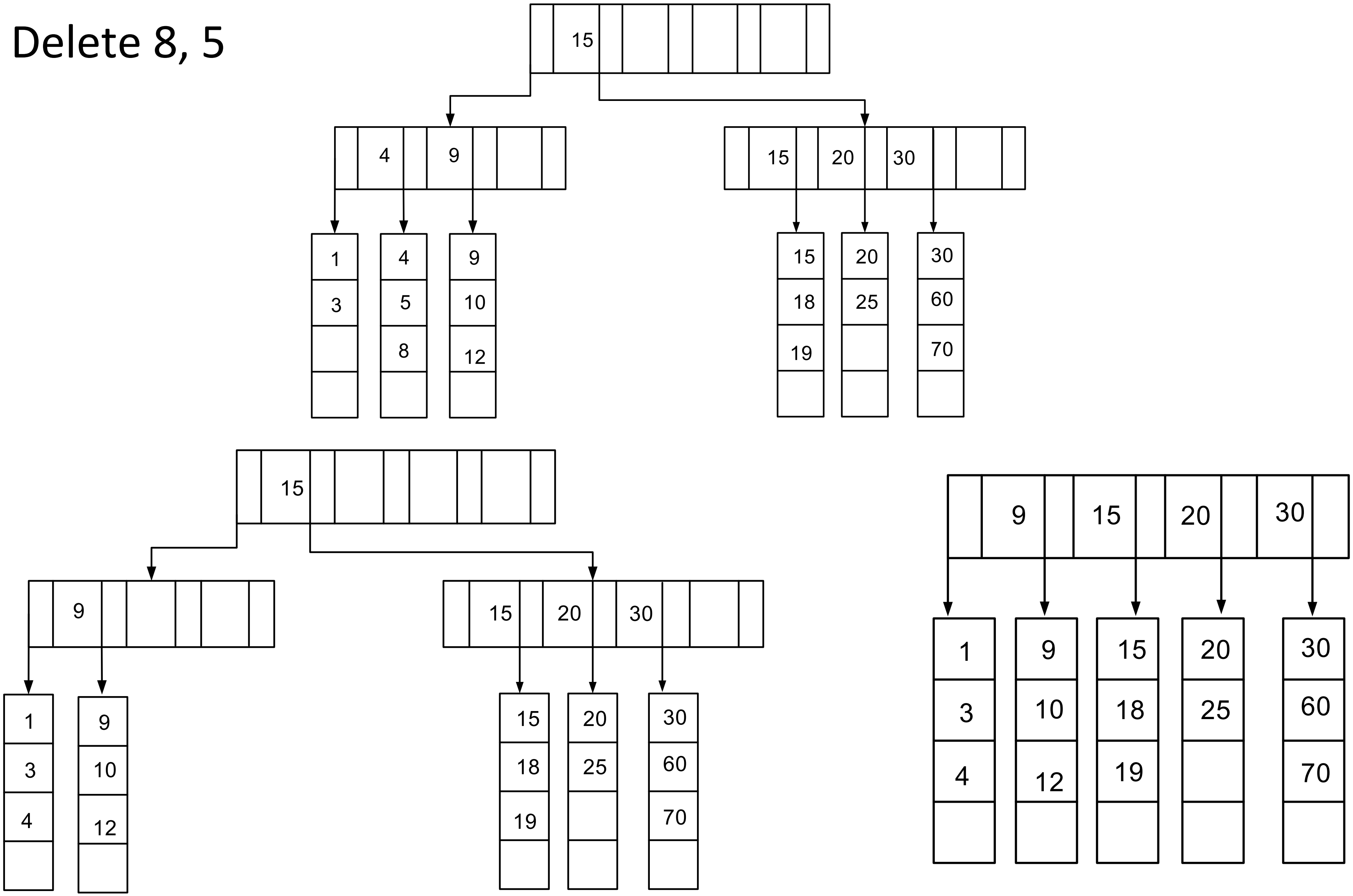
Delete 2





B-Tree Deletion

Delete 8, 5



B-Tree Deletion

Leaf Node 50% filled	Internal Node 50% filled	
No	No	Remove key from leaf node Sort node if necessary
Yes	No	Combine leaf node with his neighbor (left or right) If not possible, merge with a neighbor node Switch node parent to reflect the change
Yes	Yes	Combine leaf node with his neighbor (left or right) If not possible, merge with a neighbor node Combine parent node with its next node Continue combining parent nodes until reach a node with occupation >= 50%

Applications

Auxiliary Databases indexes

Goal: external research, minimize disk accesses

- The nodes have references to disk blocks
- The order of the tree depends on the ratio of the number of records that can fit in a single block (basic unit of disk access)
- Each node must have the size of a disk block (or multiple) so that, accesses to several nodes for a search operation or updating is done only in a single disk transfer
- If $M = 128$, then a B-Tree of height 4 will store at least 30,000,000 items