Chapter 5: Trees

1. Introduction
2. Overview
3. Examples of using trees
4. Implementation
5. Binary trees
6. Binary Search Trees

Introduction

- In Array we have:

  + Direct access by index (fast)

  - Insertion and deletion require shifting

- In linear linked lists we have:

  + Insertion and deletion are done only by chain modification.

  - Slow sequential access.

- **Trees represent a compromise between the two:**

  + Relatively quick access to an item.

  + Cost-effective addition and removal.

Introduction

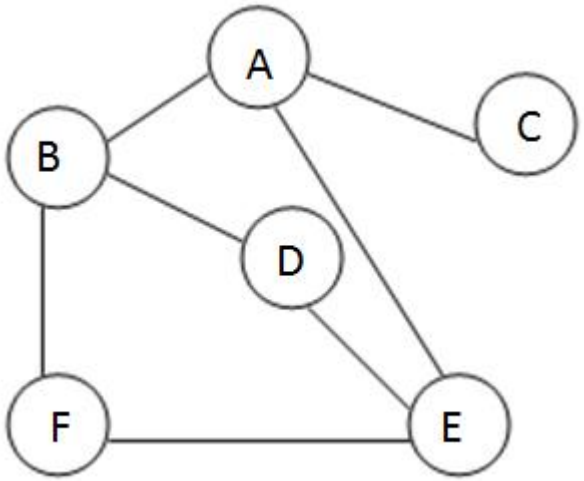
- The tree structure is one of the most important and distinctive structures in computer science, enabling the creation of highly efficient algorithms.
- Several computer treatments have a tree-structured nature, such as:
  - The representation of programs processed by a computer,
  - The representation of a table of contents….
  - Representation of arithmetic expressions

Overview (Definition)

LCraph Concept

A graph G= (S, A) is defined by:

- A set of vertices **S** .
- a set of edges **A** .



Examples of graphs

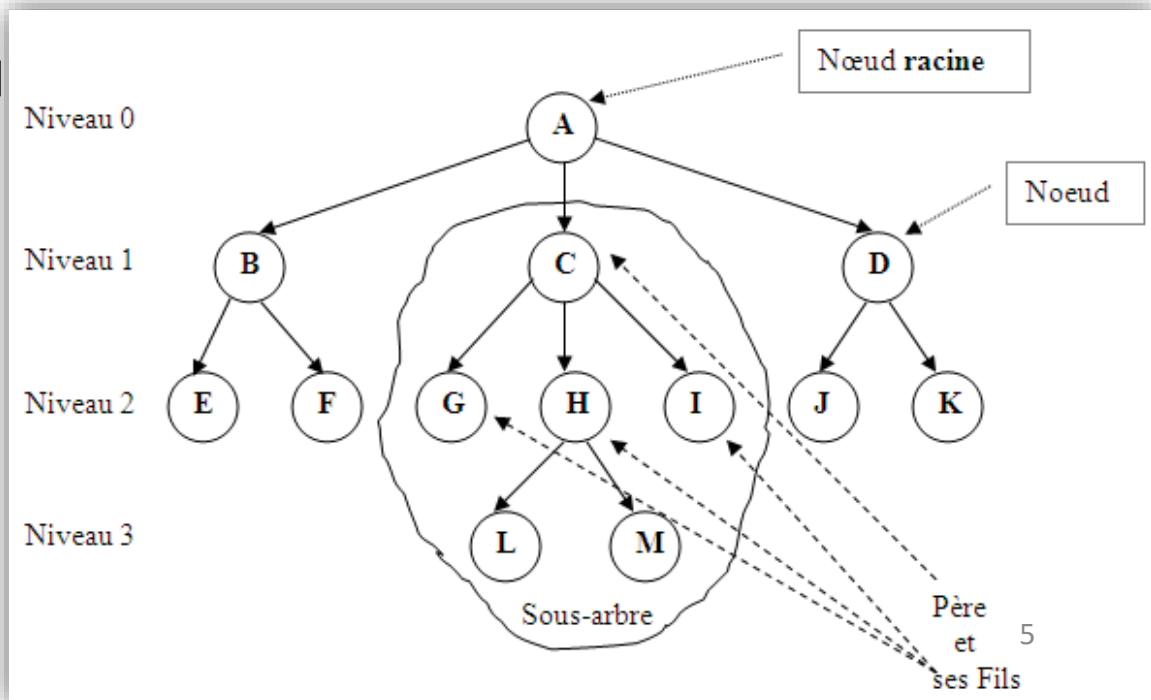
(cities, roads),

(machines, networks),

TREE: A tree is a non-linear structure, it is a **graph** without cycles where each node **has at most one predecessor** .

Overview (General notions)

- The **predecessor**, if it exists, is called **the father** (father of C = A, father of L= H)
- The **successor,** if it exists, is called **a child** (Children of A = { B,C,D }, Sons of H = { L,M })
- The node that has no predecessor is called **root** (A)
- The node that has no successor is called **a Leaf** (E,F,G,L,J,...)
- **Descendants** of C={G,H,I,L,M
- **Ascendants** of L={H,C,A }



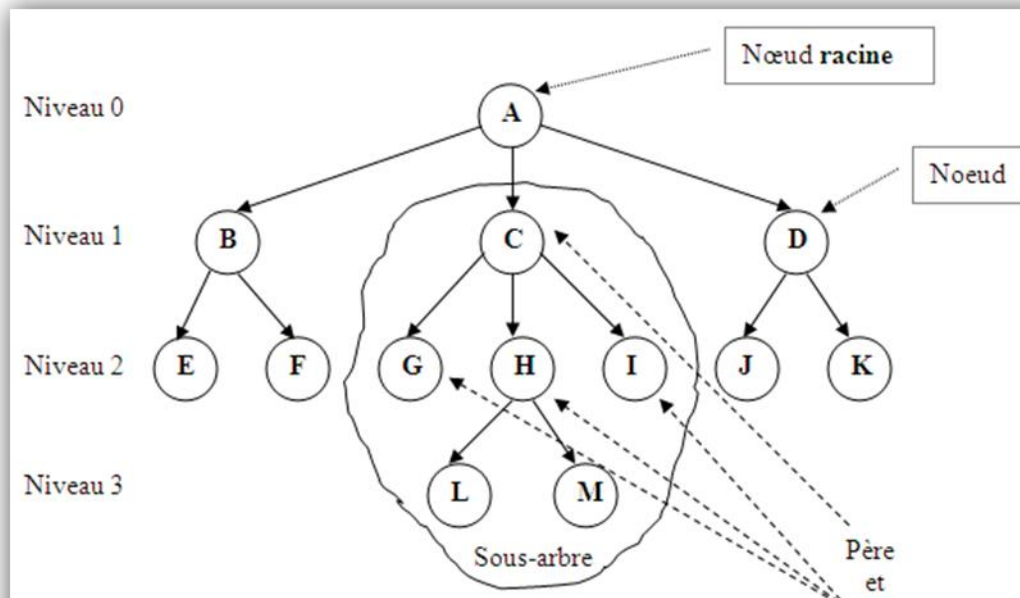
General (Tree measurements)

Size of a tree: is the number of nodes it has.

- Size of this tree = 13
- An empty tree has size 0.

Node Level: The level of a node is the number of nodes between it and the root.

- Root level = 0
- The level of each node is equal to the level of its parent plus 1
- Level of E,F,G,H,I,J,K = 2



General (Tree measurements)

Height (Depth) of a tree: the maximum level in that tree.

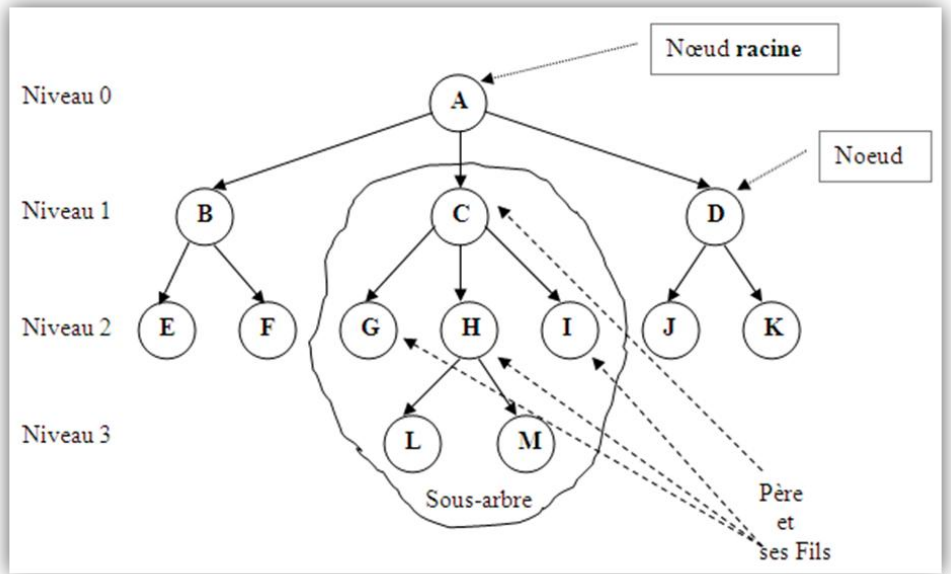
- Height of previous tree = 3

Degree of a node: The degree of a node is equal to the number of its children.

- Degree of (A = 3, B = 2, C = 3, E = 0, H = 2,...)

Degree of a tree: is the maximum degree of its nodes.

- The previous tree degree = 3.



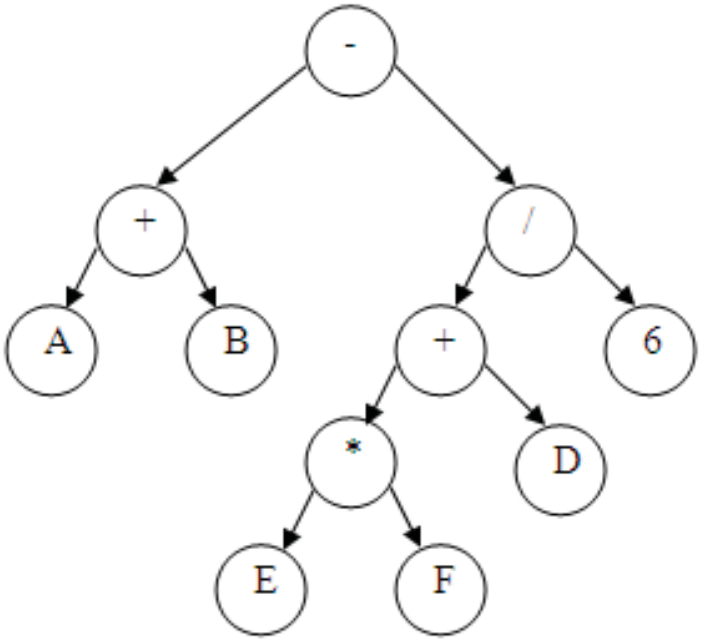
Generalities (Tree typology)

- **N-ary tree :** An n-ary tree of order n is a tree where the maximum degree of a node is equal to n.
- **Binary tree:** It is a tree where the maximum degree of a node is equal to 2.
- **Binary Search Tree:** This is a binary tree where the key of each node is greater than those of its left descendants, and less than those of its right descendants.

Examples of using trees

Representation of arithmetic expressions:

(A+B) - (E*f + d)/6

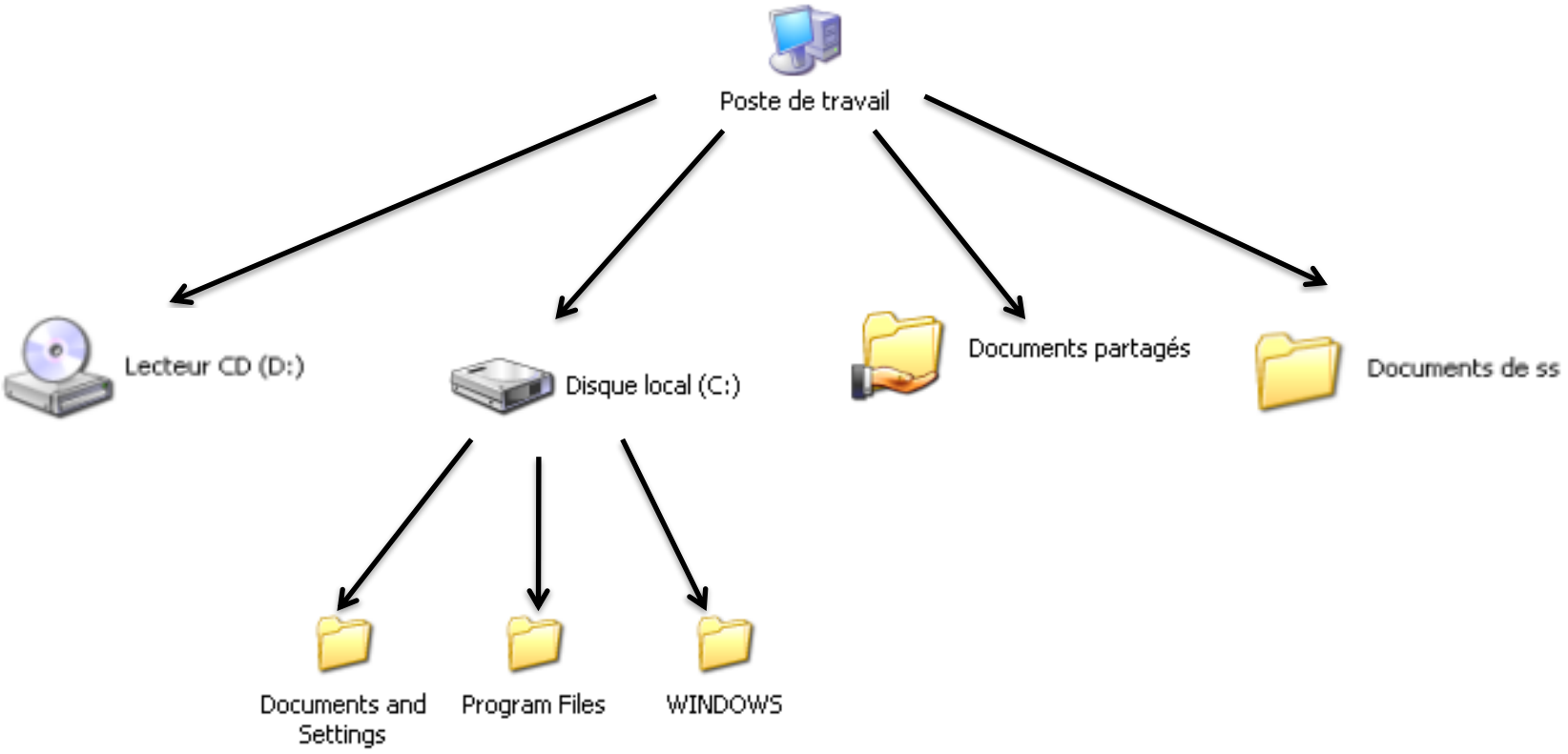


Exercise: give the tree representation of the following expressions:

1. A*B/3-8*C
2. 3*(9-2)+(5-2*8)

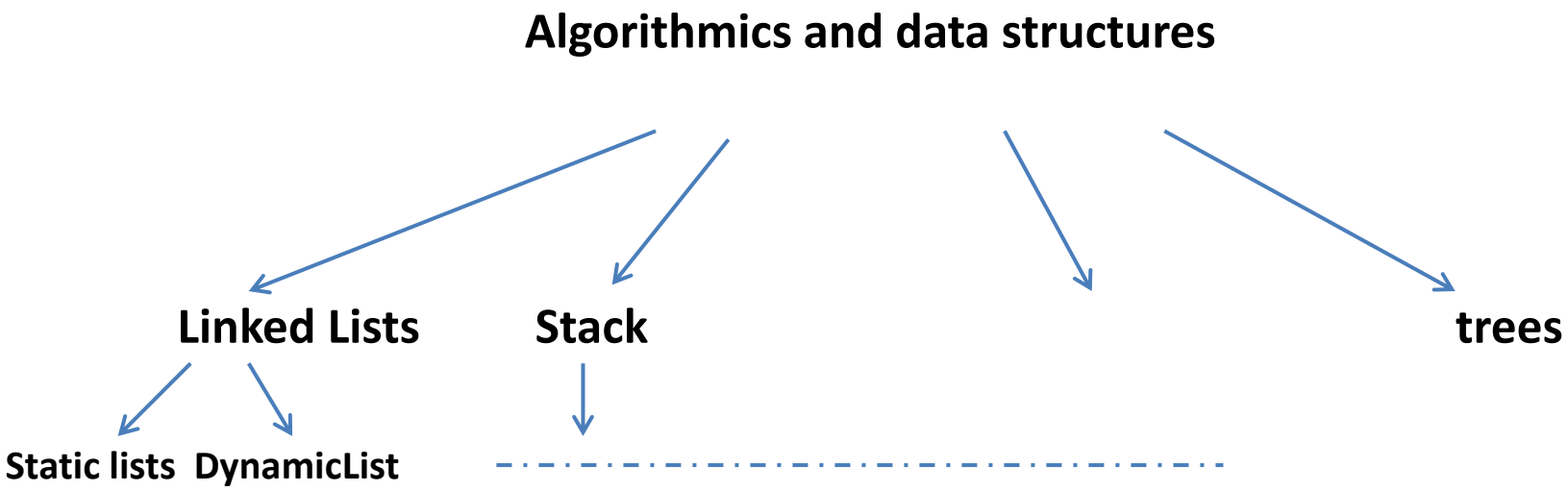
Examples of using trees

File representation in Windows.



Examples of using trees

Representation of tables of contents.

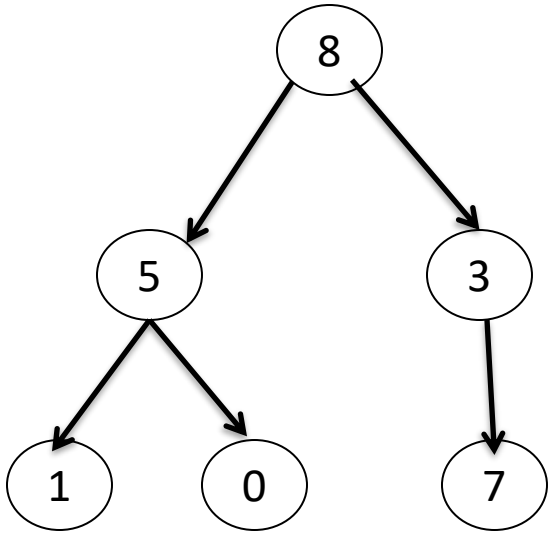


Implementation

Trees can be represented by non-linear arrays or lists:

a) Static representation

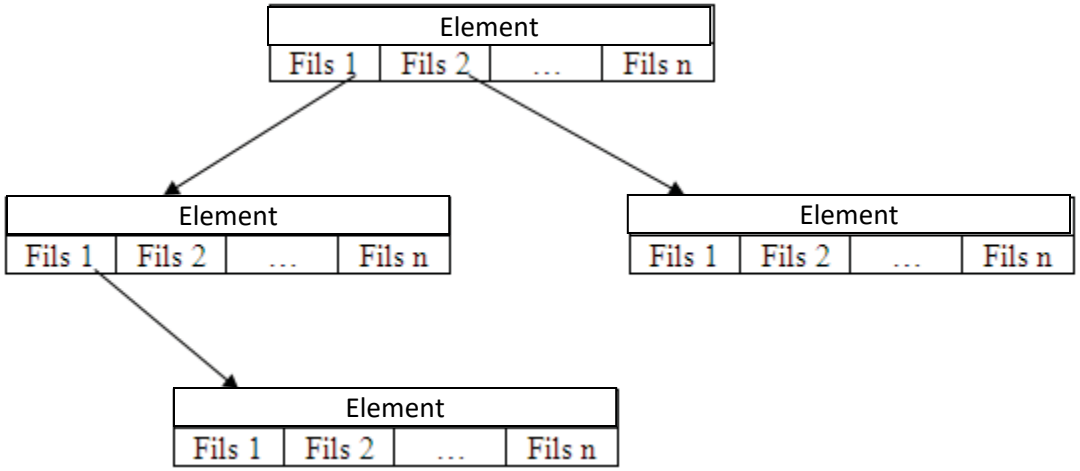
Below is the representation of an integer tree by an array:



	element	Son 1	Son 2
0	8	1	2
1	5	3	4
2	3	5	-1
3	1	-1	-1
4	0	-1	-1
5	7	-1	-1

Implementation

b) Dynamic representation: using a non-linear list where each node contains an element with the list (An array or a list linked) of his sons.



Type definition

Type Structure **Node**

Ele: typeqq ;

Children: Array[NbChildren] of (***Node**);

END ;

Type Tree: * Node;

2. Binary trees

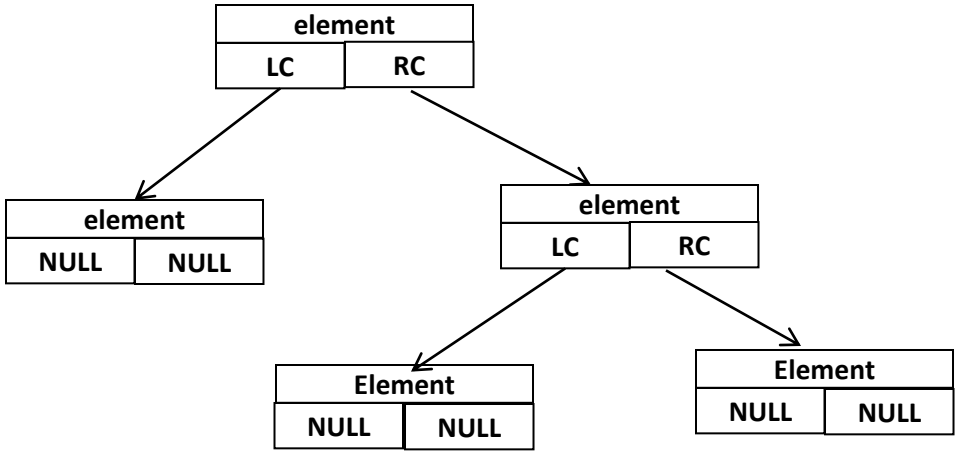
2. 1 definition

Let us consider a set **of nodes** to which "values" are associated (the elements to be stored: integer, real, structure, etc.). A **binary tree** can be defined recursively with the following cases:

- **Empty Tree (NULL)**: A binary tree can be empty, represented by NULL.
- **Single Node**: A binary tree may consist of only a single node (**root)**.
- **Root with Left Subtree**: A binary tree may have a root node with a left binary subtree attached to it. The right child in this case is NULL.
- **Root with Right Subtree**: A binary tree may have a root node with a right binary subtree attached to it. The left child in this case is NULL.
- **Root with Both Subtrees**: A binary tree may have a root node with both a left binary subtree and a right binary subtree attached to it.

2.2 Implementation

Dynamic representation: using a non-linear list, each node contains an element with the address of its left child (LC) and the address of its right child (RC)



Type definition

Type structure **Node** :

Ele: Element; // type Element means any type

LC: * **Node** ; // pointer to left child

RC: * **Node** ; // pointer to the right child

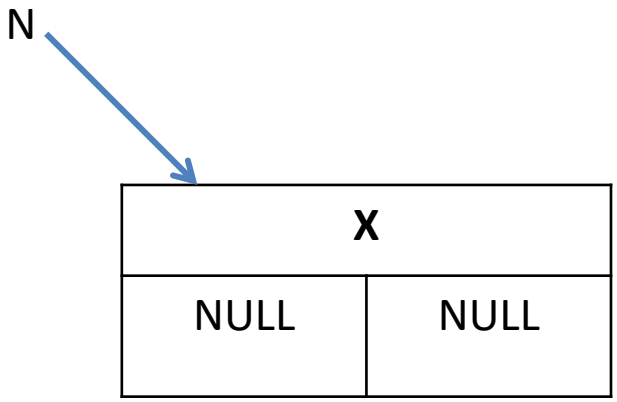
END ;

Type Tree : * Node ;

2.3 Operations

1.Create_node function : creates a node containing the value x and without children (LC=NULL, RC=NULL) and returns a pointer to the created node.

```
Function Create_node ( x: Element ): Tree
    N:Tree ;
Begin
    N ← Allocate ( Node );
    N->element ← x;
    N-> RC ←NULL ;
    N-> LC←NULL ;
    Return (N)
END
```



2.3 Operations

2. The Content function: returns the content (value) of a node.

```
Function Content (a: Tree): Element
Begin
    Returns (a -> element)
END
```

3. Is_empty: Tests whether a tree is empty or not .

```
Function Is_empty(a: Tree): boolean
Begin
    Returns (a=NULL)
END
```

2.3 Operations

4. **The LeftChild(a: Tree) function: returns the left son of a tree (the left subtree).**

```
Function LeftChild (a: Tree): Tree
Begin
    Returns (a- >LC)
END
```

5. **The RightChild function (a: Tree): returns the right son of a tree (the right subtree).**

```
Function RightChild (a: Tree): Tree
Begin
    Returns (a- >RC)
END
```

2.3 Operations

6. **Is_leaf :** Tests whether a node, identified by its address, is a leaf or not.

```
Function Is_leaf (N: Tree): boolean
Begin
    Returns (N!=NULL and LeftChild(N)=NULL and RightChild(N)=NULL)
END
```

7. **The Size function:** returns the number of elements in a tree a.

```
Function Size (a: Tree): boolean
Begin
    if a= NULL then
        returns 0;
 else return 1 + Size( LeftChild(a)) + Size( RightChild(a));
 end if
END
```

2.4 Traversing a tree

Two deferent approaches:

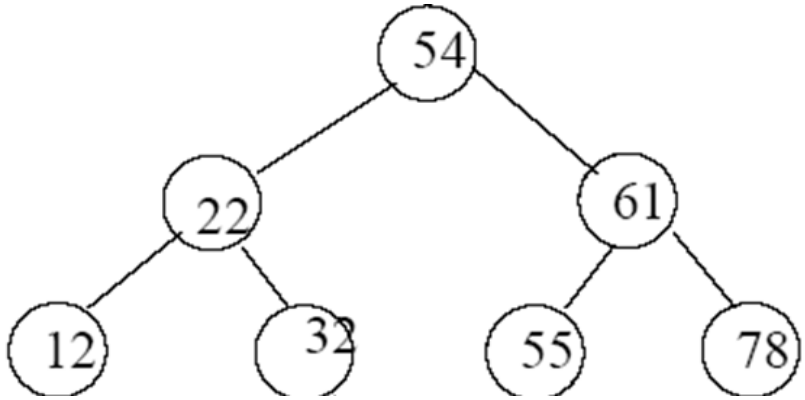
1. **Depth traversal of a binary tree:** includes the following traversal methods:
   - **Preorder Traversal** (Root-Left-Right)
   - **Inorder Traversal** (Left-Root-Right)
   - **Postorder Traversal** (Left-Right-Root)

2. **Breadth-first traversal of a binary tree:** explores all the nodes at each depth level of the binary tree before moving on to the next level

2.4 Path operations

Preorder : The rule of this traversal is very simple: **the root of the tree is processed before its descendants.**

```
Procedure Preorder (a: tree)
Begin
If a != NULL then
      Write(content(a));
      Preorder ( LeftChild(a));
      Preorder ( RightChild(a));
Endif
END
```



Example:

Preorder travarsal of this tree gives: **54 22 12 32 61 55 78**

2.4 Path operations

Inorder : The rule of this traversal is as follows: **the root of the tree is processed between its descendants.**

Procedure **Inorder** (a: tree)

Begin

If! is_empty(a) then

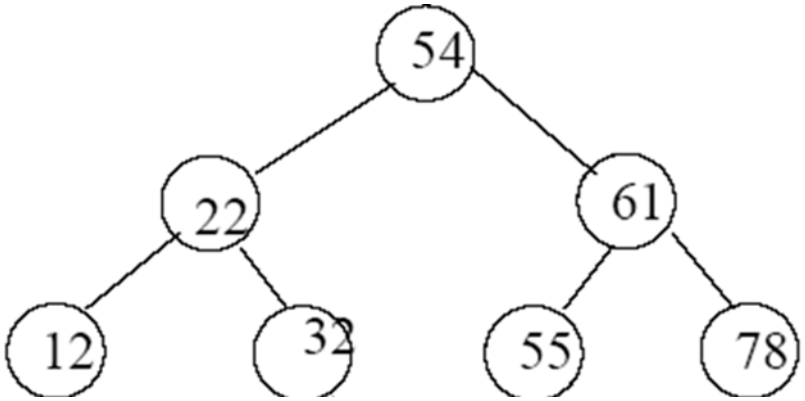
Inorder (LeftChild(a));

Write (content (a));

Inorder (RightChild(a)) ;

Endif

END



Example

Inorder travarsal of the tree A gives: **12 22 32 54 55 61 78**

2.4 Path operations

Postorder : The rule for this traversal is as follows: **the root of the tree is processed after its descendants.**

Procedure **Postorder** (a: tree)

Begin

If! is_empty(a) then

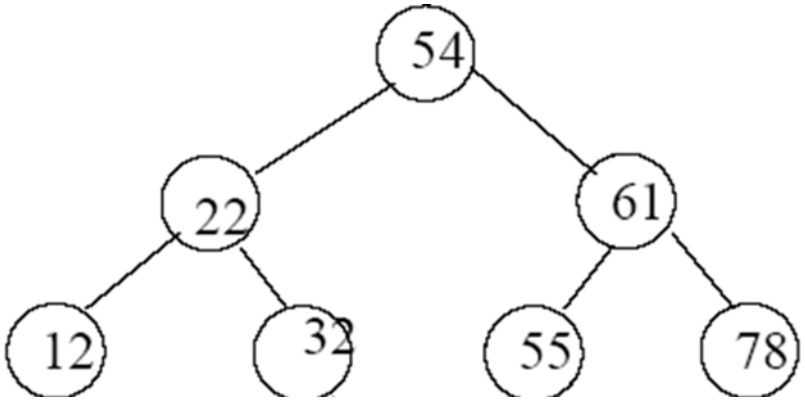
Postorder (LeftChild(a));

Postorder (RightChild(a));

write (content(a));

Endif

END



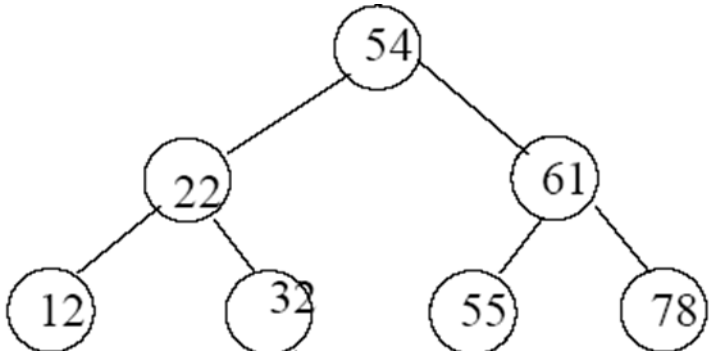
Example

Postorder travarsal of tree A gives: **12 32 22 55 78 61 54**

2.4 Path operations

2. Breadth-first traversal

- All nodes at depth **i** must have been visited before the first node at depth **i + 1** is visited.
- Such traversal requires that one remembers all the branches that remain to be visited.
- To do this, we use a queue.



Example:

The Breadth-first traversal

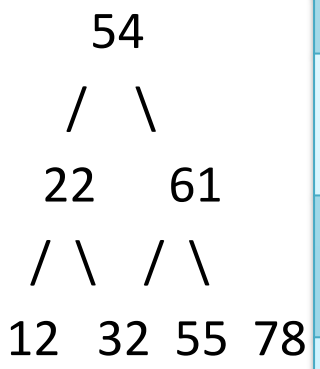
of tree A gives: **54 22 61 12 32 55 78**

2.4 Path operations

Breadth-first traversal procedure

```
Procedure Breadth-first traversal (a: Tree)
      F: queue
Begin
       Initialise(F);
       If ! is_empty(a) then
          Enqueue (F, a)
          While ( ! Is_empty (F) ) do
             write(content(head(F)) );
             if ! Is_empty (LeftChild ( head (F)) then
                      Enqueue (F, LeftChild ( head (F)))
             Endif
             if ! Is_empty ( RightChild( head (F)) then
                      Enqueue(F, threadD ( head (F )) )
             Endif
             Dequeue (F);
          End while
       Endif
END
```

Breadth-first traversal procedure example



State	operation	queue
Initial		Queue F = [ ]
Step 1 : Visit root	Enqueue(54) ;	Queue = [54]
Step 2: 1st iteration	Visit 54 ; Insert children of 54 ; Dequeue 54	Queue = [22, 61]
Step 3: 2nd iteration	Visit 22 ; Insert children of 22 ; Dequeue 22	Queue = [61, 12, 32]
Step 4: 3rd iteration	Visit 61 ; Insert children of 61; Dequeue 61	Queue = [12, 32, 55, 78]
Step 5: 4th iteration	Visit 12 ; Children: none	Queue = [32, 55, 78]
Step 6: 5th iteration	Visit 32 ; Children: none	Queue = [55, 78]
Step 7: 6th iteration	Visit 55 ; Children: none	Queue = [78]
Step 8: 7th iteration	Visit 78; Children: none	Queue = [ ]

Loop ends → Tree traversal complete. Result: 54 22 61 12 32 55 78

2.5 Searching for an element

1. **Function Belongs(a , x) :** tests whether x exists in the tree a or not.

```
Function belong (a: tree, x: typeq ): boolean // Version1
Begin
    If is_empty (a)) then
        return false;
    else
        If( content(a) = x) then
            return true;
        else
            If belongs (LeftChild(a, x)) then
                return true;
            else
                return ( belongs (RightChild(a), x));
            Endif
        Endif
    Endif
END
```

2.5 Searching for an element

Other Solution :

```
Function belong (a: tree, x: typeq ): boolean // Version2
Begin
      If is_empty (a)) then
            return false;
      else
        If( content(a) = x) then
            return true;
        else
          Return ( (belongs (LeftChild(a, x)) ) or ( belongs
      (RightChild(a), x))))

        Endif
      Endif
END
```

2.5 Searching for an element

Version 1	Version 2
• Pedagogical • Easier to explain to beginners • Shows decisions clearly	• More elegant • Mathematically clear • Single return expression
- Uses **explicit IF/ELSE**: • Try left • If found → STOP • Else try right	- evaluates both subtrees but **uses short-circuit OR**: • If left subtree returns **true** → right subtree is never called • If left subtree returns **false** → then evaluate right subtree

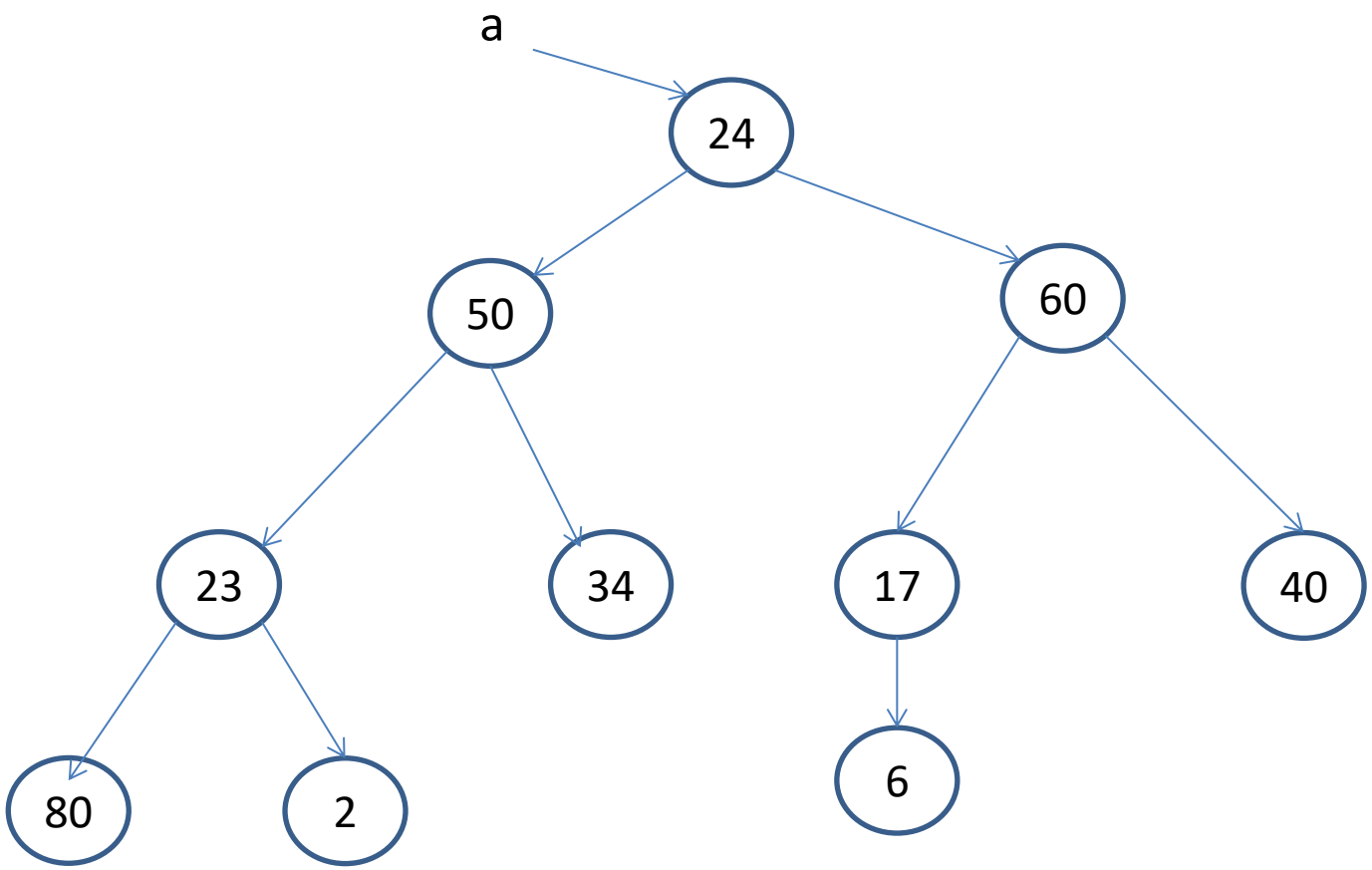
Behavior of the two versions is identical .

2.5 Searching for an element

2.The search function: returns the address of the node containing x if x exists in the tree, otherwise it returns NULL.

- ✓The function performs a **Depth-First Search (DFS)** ;
- ✓It checks the current node starting with the **root**
- ✓Then recursively explores the **left subtree**
- ✓ If not found, it recursively explores the **right subtree**

- ➢Because the tree is not ordered,we **must search all branches** until a match is found (or the tree is exhausted).

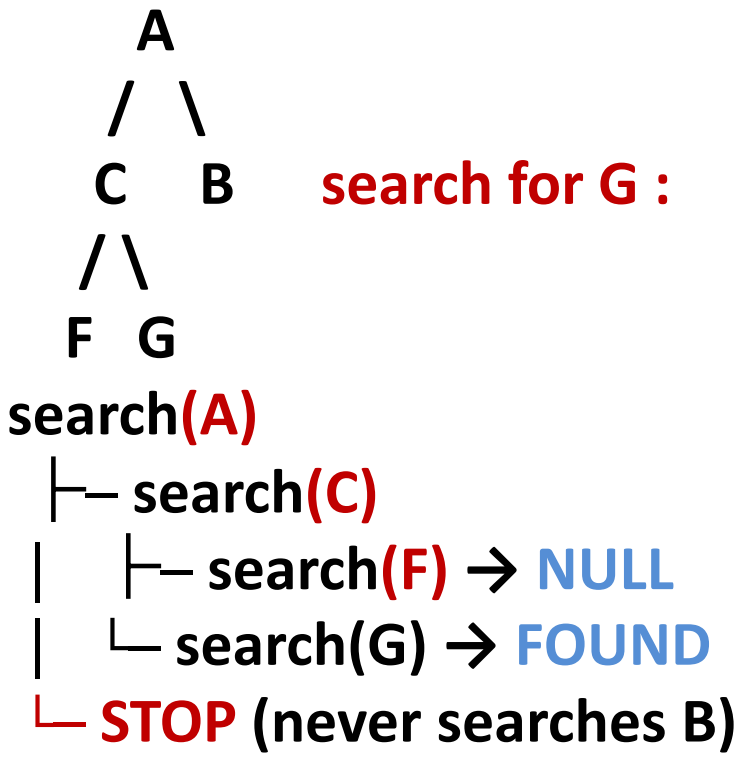


Because the binary tree has no ordering, we must visit the whole left subtree before the whole right subtree

2.5 Searching for an element

```
Function search(a:tree,x: typeq ):tree
    P: Tree;
Begin
    If is_empty(a) then
        return Null ;
    end if
    If( content(a) = x) then
        return  a ;
    endif
    P← search (LeftChild(a), x);
    If  P <>  NULL then
       return P
    Endif
    return ( search (RightChild(a),x));

END
```



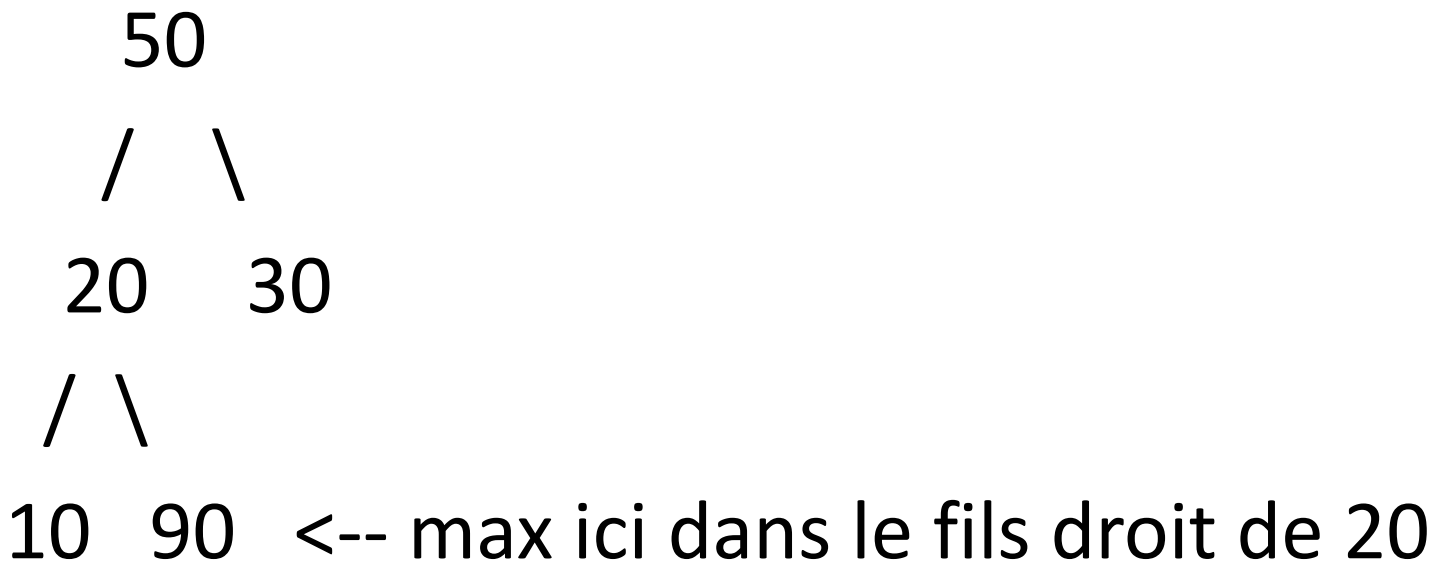
//Searching left and **immediately returning if found**
Avoiding unnecessary search on right subtree

2.6 the maximum of a binary tree

```
Function Max(a : Tree) : typeq
      If is_empty(a) then       // Base case: empty tree has no value
      return -∞      // or the minimum possible value
     EndIf
    m ← content(a)       // Start with the root value
   leftMax ← Max(LeftChild(a))     // Compute the max in the left subtree
   If leftMax > m then
     m ← leftMax
  EndIf
rightMax ← Max(RightChild(a))
  If rightMax > m then       // Compute the max in the right subtree
     m ← rightMax
  EndIf
  return m
End
```

2.6 the maximum of a binary tree

Example :



rightMax = 90

90 > m ? 90 > 20 → OUI

→ m = 90

CALL Max(90) → m = 90

LeftChild(90) = NULL → Max(NULL) = -∞

RightChild(90) = NULL → Max(NULL) = -∞

RETURN 90

(back to Max(20))

leftMax = 10 if 10 > 20 ? NON

→ m = 20

CALL Max(10) → m = 10

LeftChild(10) = NULL → Max(NULL) = -∞

RightChild(10) = NULL → Max(NULL) = -∞

RETURN 10

(back to Max(20))

CALL Max(20) → m = 20

CALL Max(50) → m = 50

Return 90

rightMax = 30

30 > 90 ? NON → m = 90

leftMax = 90

90 > 50 ? OUI → m = 90

CALL Max(30) → m = 30

LeftChild(30) = NULL → Max(NULL) = -∞

RightChild(30) = NULL → Max(NULL) = -∞

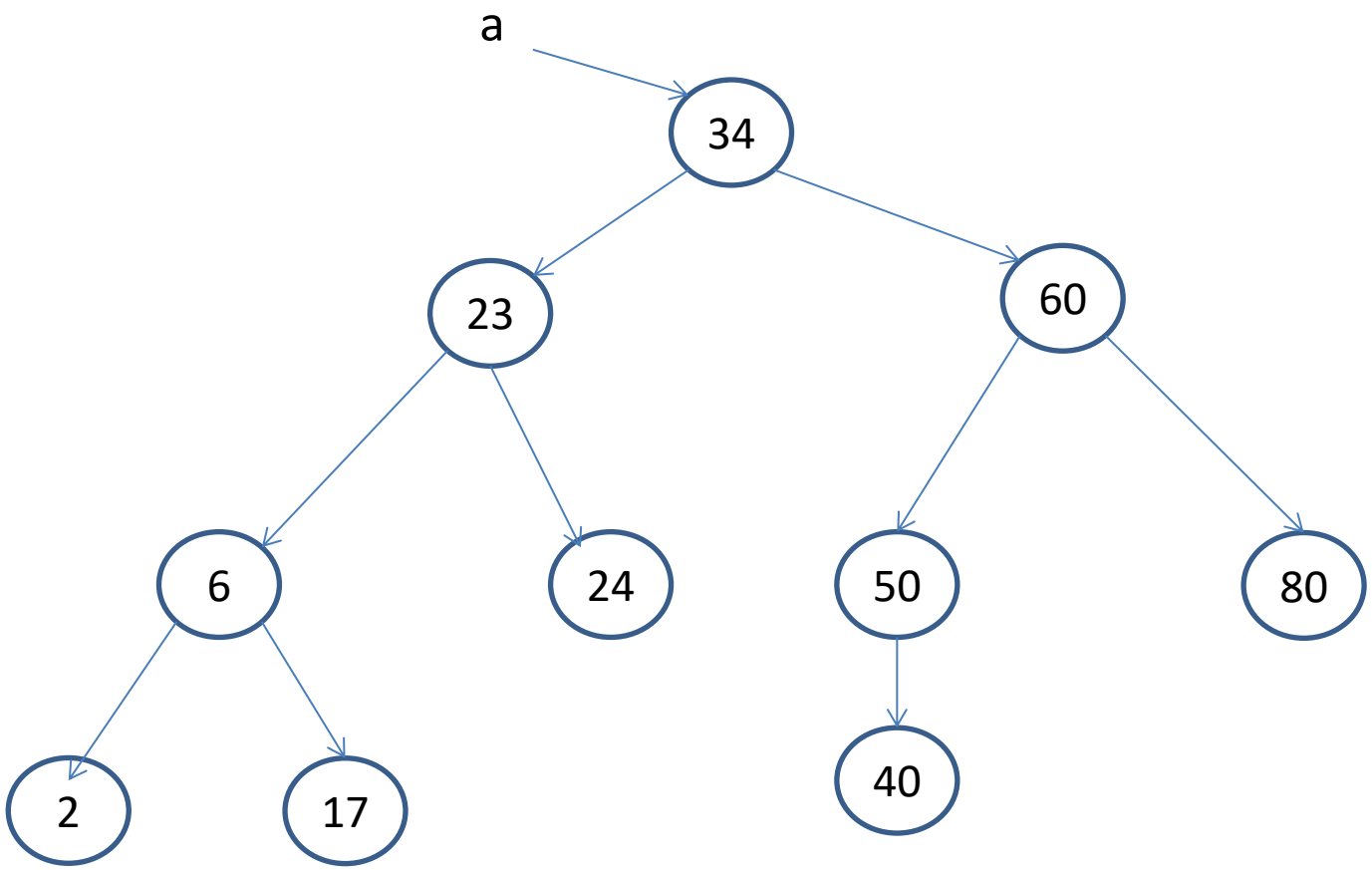
RETURN 30

(back to Max(50))

RETURN 90

(back to Max(50))

3. Binary Search Trees (BST)

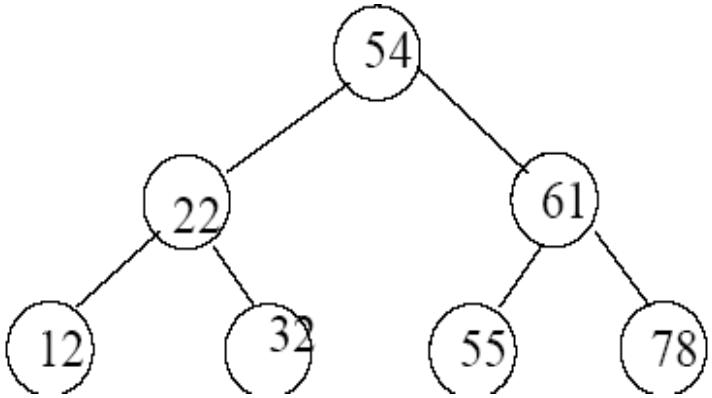

a
34
23
60
6
24
50
80
2
17
40

3.1 definition

A binary search tree is a binary tree that satisfies the following property:

let x and y be two nodes of the tree ,

- *if y is a node in the left subtree of x, then key(y) < key(x),*
- *if y is a node in the right subtree of x, then key(y) > key(x).*



3.2 Operations

1.Belong (x, a): This function tests whether x exists in the tree a or not.

```
Function Belong (a: Tree, X: Type): boolean
Begin
    If is_empty(a) then
        return false;
    else
        If( content(a) =  x) then
                return true;
        else
                If ( x < content(a)) then
                        return ( Belong (x, LeftChild(a)) ;
                else
                        return ( Belong(x, RightChild(a));
                Endif
        Endif
Endif
END
```

3.2 Operations

2. Insert (a:Tree, x:Element) using the prefixed traversal it will be very easy to insert new nodes.

```
Function insert (a: tree, x: type) : tree
Begin
  if is_empty (a)) then
            a ← create_node(x);
  end if
  If ( x < content(a)) then
            a -> LC ← insert (LeftChild(a) , x);
  end if
If ( x > content(a)) then
            a -> RC ← insert (RightChild(a), x );
Endif

return a ;
END
```

3.2 Operations

3.Deletion: Deleting a key in a tree is a more complex operation. Several cases are to be considered depending on the number of children of node x:

- If the element to be deleted does not exist, nothing is done.
- If the element to be deleted has no left child, it is replaced by its right child.
- If the element to be deleted has no right child, it is replaced by its left child.
- If the element to be deleted has two children, it is replaced by the largest (respectively smallest) element of its left (respectively right) subtree .

Algorithm deletion : using one function

- When deleting a node:

🌱 **Case 1 — It's a leaf** → Remove it.

🌿 **Case 2 — It has one child** → Replace it by its child.

🌳 **Case 3 — It has two children**

- → Replace its value by the **smallest value in its right subtree**
  → Delete that smallest node from the right subtree.

Algorithm deletion : using one function

```
Function delete(a, x) : Tree
Begin
   If a = Null then
      return Null

   If x < a→Ele then
      a→LC ← delete(a→LC, x)
      return a

   If x > a→Ele then
      a→RC ← delete(a→RC, x)
      return a
   //x = a→Ele (node to delete found)

   // Case 1 node to delete  is a Leaf
   If a→LC = Null AND a→RC = Null then
      Free(a)
      return Null
   endif
   // Case 2 node to delete has One child
    If a→LC = Null then
          Temp ← a→RC
          Free(a)
          return Temp
     If a→RC = Null then
          Temp ← a→LC
          Free(a)
          return Temp
   // Case 3 node to delete has  Two children
   // Find inorder successor = min in right sub-tree
        Succ ← a→RC
        While Succ→LC ≠ Null do
              Succ ← Succ→LC
        EndWhile
   a→Ele ← Succ→Ele      # copy value
   a→RC ← delete(a→RC, Succ→Ele)
   return a
End
```

3.2 Deletion (Solution with 3 functions) :

The difficult case:

deleting a node with two children, requires:

- **Finding the node to delete** → delete(a, x)
- **Deleting the root of its subtree** → deleteRoot(a)
- **Finding the max in the left subtree** → Max(a->LC)
- **Replacing the value**
- **Deleting the original node containing the max**

3.2 Operations

3. Deletion: Three functions cooperate for deleting an element.

A) delete : searches for the node carrying the key to delete.

```
Function delete (a: tree, x: type) : tree
Begin
    If ! is_empty (a)) then
        if ( content(a) = x ) then
                a ← deleteRoot (a);
        else
                If ( x < content(a) ) then
                        a -> LC ← delete (x, leftchild(a));
                else
                        a -> RC ← delete (x, rightchild(a));
                Endif
        Endif
    Endif
        return a;
END
```

3.2 Operations

- the second, **deleteRoot** , performs the deletion according to the cases listed above.

```
Function deleteRoot (a:tree)
Begin
    If( LeftChild(a) =  NULL )
        return RightChild(a) ;
    else
        If (RightChild(a) =  NULL)
                return LeftChild(a) ;
        else
                a-> Ele ← max ( LeftChild(a));
                a ->LC ← delete (LeftChild(a), max (LeftChild (a)),);
                return a ;
        Endif
     Endif
END
```

3.2 Operations

The third, **Max** , returns the largest element in its left subtree.

```
Function Max (a: Tree) : type
Begin
    If (a->RC )= NULL then
        return Content (a) ;
    else
        return max ( RightChild(a)) ;
    Endif
END
```

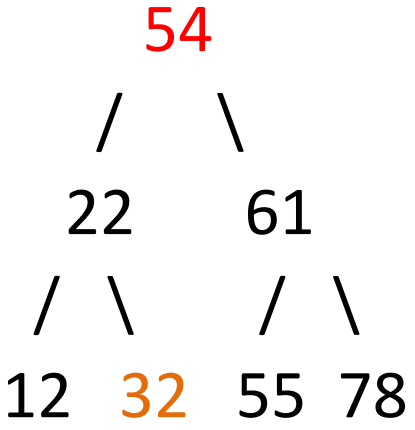
Example: **deleting a node with two children**

deleteRoot(54):

- a->Ele = Max(LeftChild(a)) // Max(22) returns 32
- //a→Ele ← 32

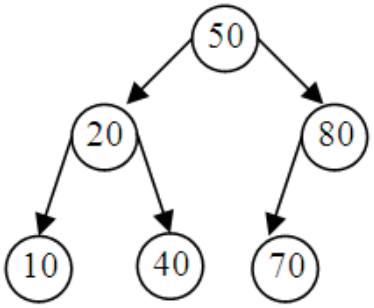
//Replace value of node 54 with 32

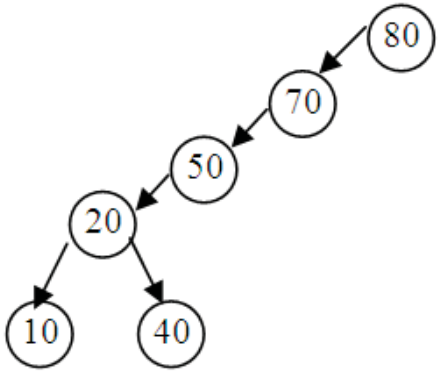
- a->LC = delete(LeftChild(a), Max(LeftChild(a)))
- delete(22, 32)
- deleteRoot must delete the ORIGINAL node containing 32:
- deleteRoot(32) returns NULL
- delete(32, 32) returns NULL
- delete(22, 32) sets 22→RC = NULL and returns 22
- deleteRoot(54) sets new left subtree and returns updated node (now 32)
- delete(54, 54) returns new root 32



3.3 Balancing

Consider the following two BSTs:





- These two BSTs **contain the same elements**, but are organized differently. The depth of the first is less than that of the second.
- If we are looking for element 10 we will have to go through 3 elements (50, 20, 10) in the first tree, on the other hand in the second, we will have to go through 5 elements (80, 70, 50, 20,10).
- The first tree is said to be more balanced.

3.3 Balancing

- A BST is said to be balanced if for any node in the tree the difference between the height of the left subtree and the right subtree is at most equal to 1.
- It is always advisable to work on a balanced tree to ensure the fastest possible search .
- The balancing operation can be done:
  - Every time a new node is inserted,
  - Whenever the imbalance reaches a certain threshold to avoid the cost of the balancing operation which requires a reorganization of the tree.

BST balancing Steps

1. **Linearization (Inorder Traversal)**

   Goal: Convert the tree structure into a sorted array/list.

2. **Reconstruction (Divide and Conquer)**

   Goal: Build a new, perfectly balanced BST from the sorted list.

BST balancing Steps : Linearization

Linearization (Inorder Traversal)

- **Goal:** Convert the tree structure into a sorted array/list.
- **Method:** Perform an **Inorder Traversal** (Left \Root\Right).
- **Result:** A list of all nodes, ordered by their values (ascending).
- **Time Complexity:** O(N).

BST balancing Steps : Reconstruction

2. Reconstruction (Divide and Conquer)

Goal: Build a new, perfectly balanced BST from the sorted list.

Principle: The root of any subtree must be the **median** (middle element) of the list segment it represents.

Method:

1. Select the middle element of the list as the new **Root**.
2. Recursively build the **Left Subtree** from the elements before the median.
3. Recursively build the **Right Subtree** from the elements after the median.

Time Complexity: O(N).

Space Complexity: O(N) (to store the node list).

Balancing an Unbalanced Binary Search Tree (BST)

Step	Function	Action
I	inorder_traversal(root)	Generates sorted_list of all nodes.
II	build_balanced_bst(list, start, end)	Recursively constructs the tree using the median as the root.
Return	new_root	The root of the perfectly balanced tree.

Procedure linearization

Procedure linearization (a: Tree, node_array: Array_of_Trees)

```
Begin
If ! is_empty(a) then
        Inorder_Traversal ( LeftChild(a), node_array);
        // Appends the node address (N) to the end of the array (T)
         Add_to_Array (node_array, a);

        Inorder_Traversal ( RightChild(a), node_array) ;
 Endif
END
```

Function Reconstruction

```
Function Array_to_Balanced_BST (node_array: Array_of_Trees, start: Integer, end: Integer) :
Tree
   mid: Integer;     new_root: Tree;
Begin
   // Base case : empty tree
   If start > end then
      return NULL;
   Endif
    mid ← (start + end) / 2;
      // the middle node becomes the root
   new_root ← node_array[mid];
   // Build recursivly the left sub-tree
   new_root -> LC ← Array_to_Balanced_BST(node_array, start, mid - 1);
   // Build recursivly the right sub-tree
   new_root -> RC ← Array_to_Balanced_BST(node_array, mid +1, end);

   return new_root;
END
```

Principal Balancing Function

```
Function Balance_BST (root: Tree) : Tree
   node_array: Array_of_Trees;

Begin
   Initialise(node_array);
   // get the sorted tree
   linearization (root, node_array);
   // reconstruction
   size_of_array ← Length(node_array);
   return Array_to_Balanced_BST(node_array, 1, size_of_array );
END
```