Data Structures
Fall 2018
Trees

Luis de Marcos Ortega

luis.demarcos@uah.es

Contents

- Basic terminology
- The ADT tree
- Implementations of trees
- Binary trees
- Binary search trees
- AVL trees

Bibliography

- Chapter 3 & Chapter 5 of:
  - A.V. AHO., J.E. HOPCROFT., J.D. ULLMAN. 1987. “Data Structures and Algorithms.” Addison-Wesley.

Terminology

- A tree imposes a hierarchical structure on a collection of items
- Familiar examples of trees are genealogies and organization charts
- Examples of uses of trees:
  - analyze electrical circuits
  - represent the structure of mathematical formulas

Terminology

- Trees also arise naturally in many different areas of computer science
- For example:
  - database systems: trees are used to organize information
  - compilers: to represent the syntactic structure of source programs
  - file systems (directories)
  - hierarchy of classes in OOP languages
  - Menus in applications

Terminology

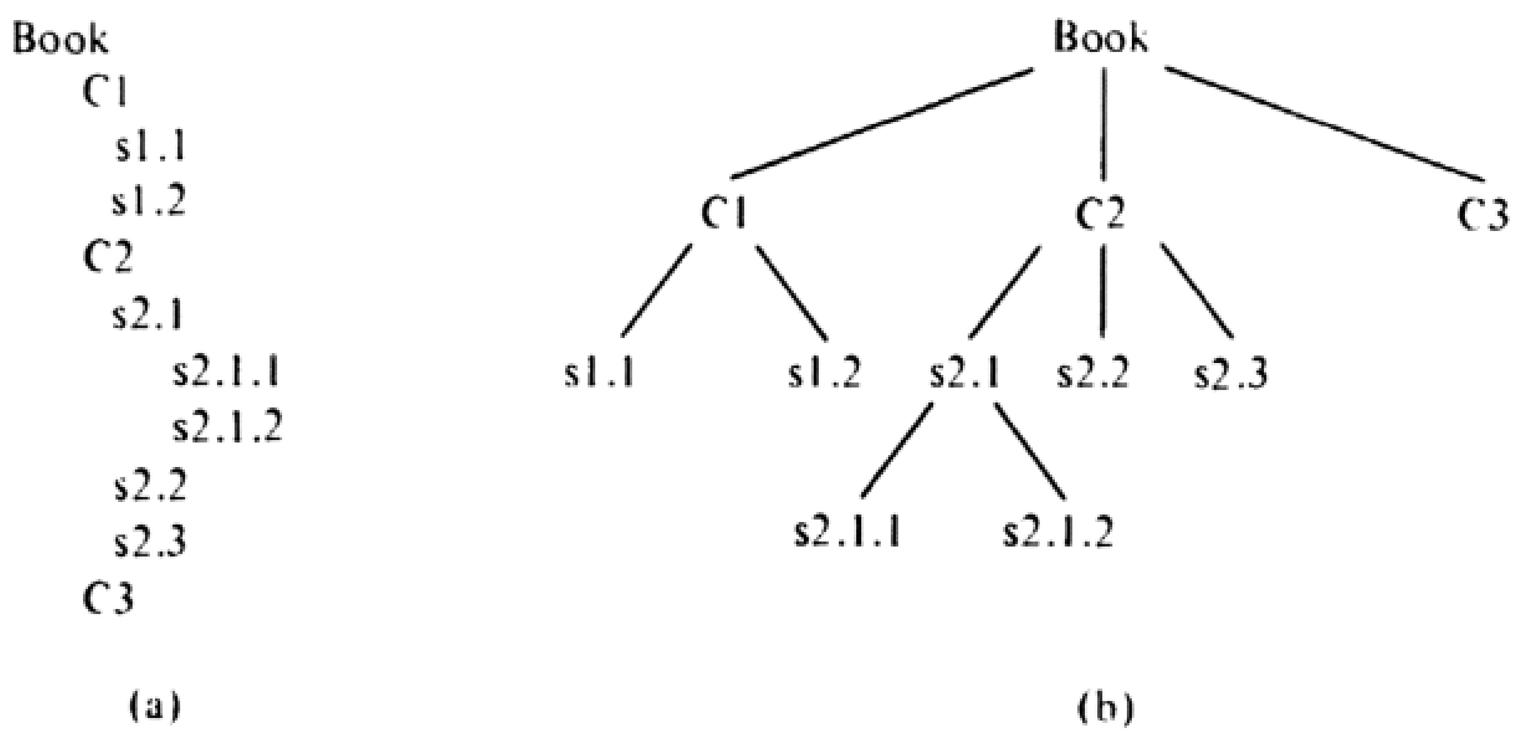
- A tree is a collection of elements called *nodes*, one of which is distinguished as a *root*, along with a relation ("parenthood") that places a hierarchical structure on the nodes.

Terminology – Definition

- Formally, a *tree* can be defined recursively in the following manner:
  1. A single node by itself is a tree. This node is also the root of the tree.
  2. Suppose n is a node and $T_1, T_2, \ldots, T_k$ are trees with roots $n_1, n_2, \ldots, n_k$, respectively. We can construct a new tree by making n be the parent of nodes $n_1, n_2, \ldots, n_k$. In this tree n is the root and $T_1, T_2, \ldots, T_k$ are the *subtrees* of the root. Nodes $n_1, n_2, \ldots, n_k$ are called the *children* of node n.
  - *null tree*, a "tree" with no nodes, which we shall represent by Λ

Terminology

- Example: Table of contents of a book

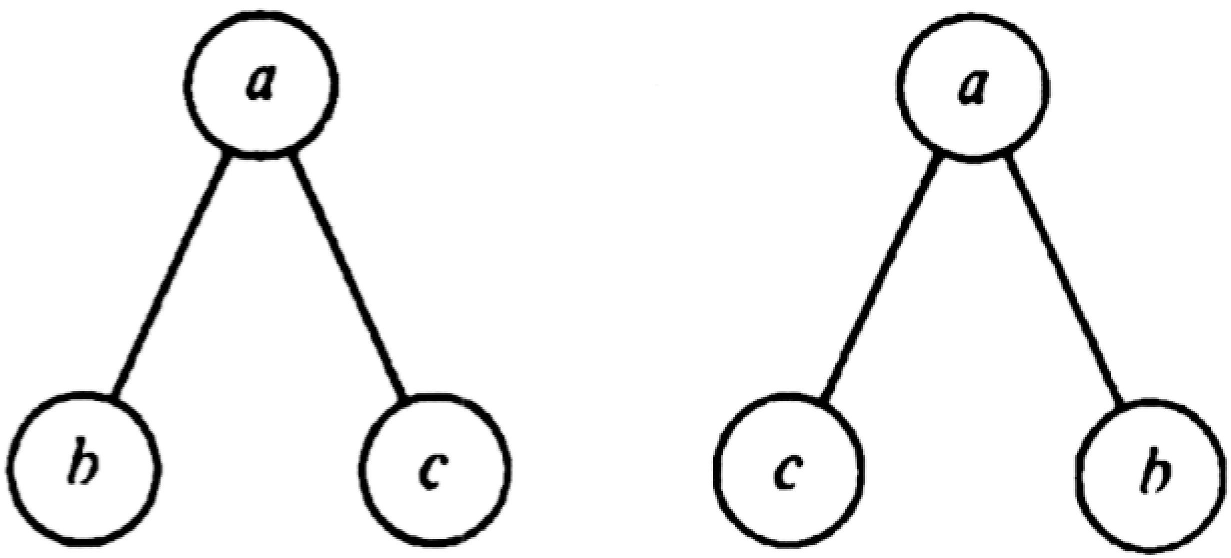


Terminology

- *Parent, child, siblings* (children of the same node)
- *Path* from one node to another
- *Length* of a path
- *Ancestor* of a node
- *Descendant* of a node
- A node with no descendants is called a *leaf*
- *Height* of a node – longest distance to a leaf
  - *Height of a tree* is the height of the root.
- *Depth* of a node – distance to the root

Terminology – Order of nodes

- The children of a node are usually ordered from left-to-right.

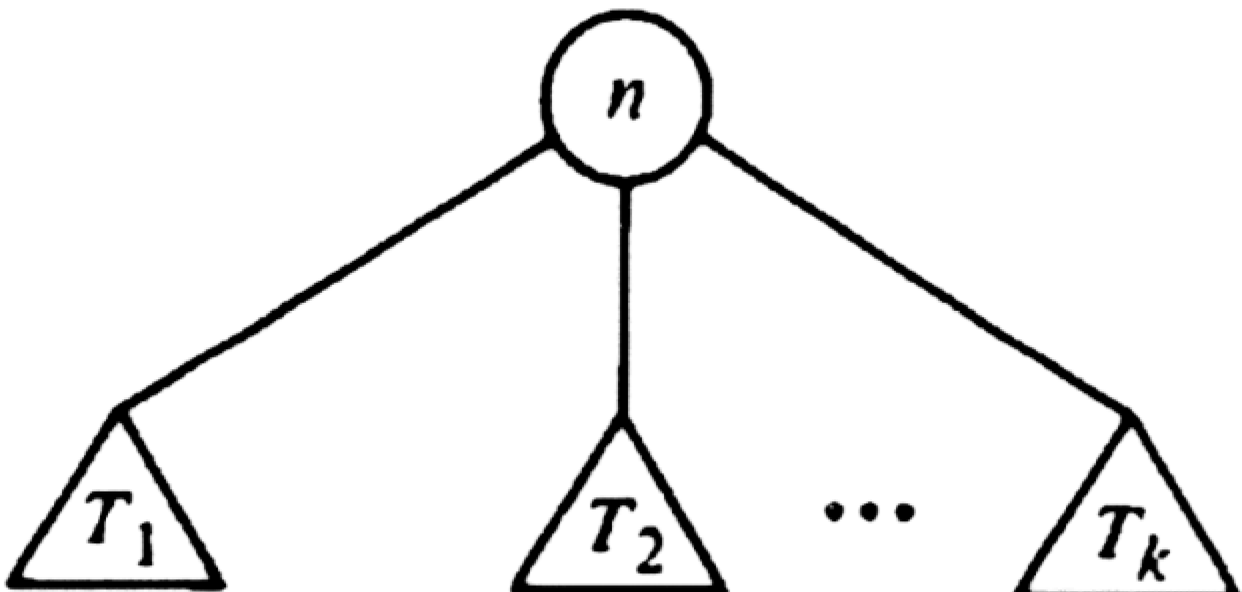


Terminology – Traversing

- There are several useful ways in which we can systematically order (or traverse) all nodes of a tree. The three most important orderings are called <u>preorder, inorder and postorder</u>; these orderings are defined recursively as follows
  - If a tree T is null, then the empty list is the preorder, inorder and postorder listing of T.
  - If T consists a single node, then that node by itself is the preorder, inorder, and postorder listing of T.

Terminology – Traversing

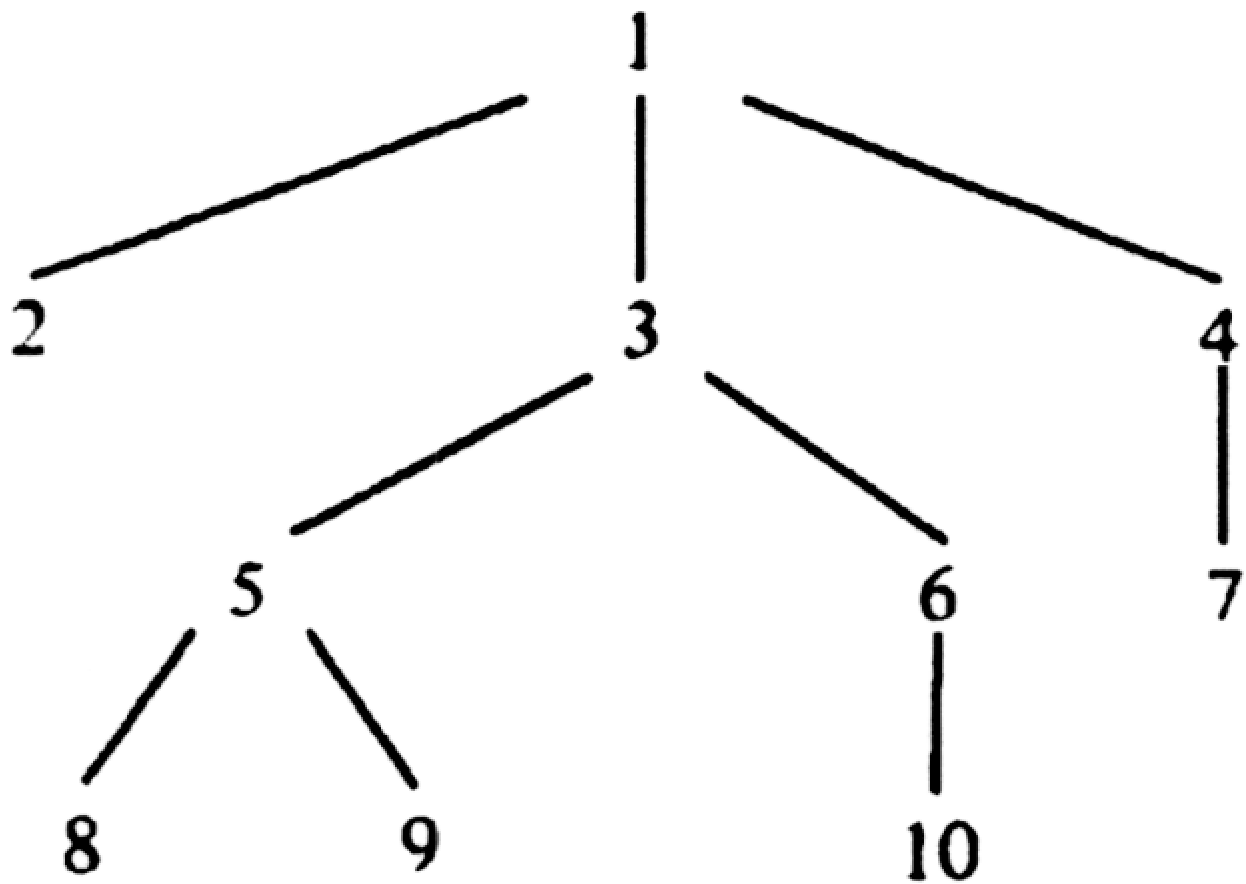
- Otherwise, let T be a tree with root n and subtrees T_1, T_2, . . . , T_k, as suggested in Fig.



Terminology – Traversing

- The *preorder listing* (or *preorder traversal*) of the nodes of T is the root n of T followed by the nodes of T_1 in preorder, then the nodes of T_2 in preorder, and so on, up to the nodes of T_k in preorder.
- The *inorder listing* of the nodes of T is the nodes of T_1 in inorder, followed by node n, followed by the nodes of $T_2, \ldots, T_k$, each group of nodes in inorder.
- The *postorder listing* of the nodes of T is the nodes of T_1 in postorder, then the nodes of T_2 in postorder, and so on, up to T_k, all followed by node n.

Terminology – Traversing



Terminology – Traversing

void tree::PREORDER (*n*: node)

(1) list *n*;

(2) **for** each child *c* of *n*, if any, in order from the left **do** PREORDER(*c*)

endmethod { PREORDER }

- To make it a POSTORDER procedure, we simply reverse the order of steps (1) and (2).

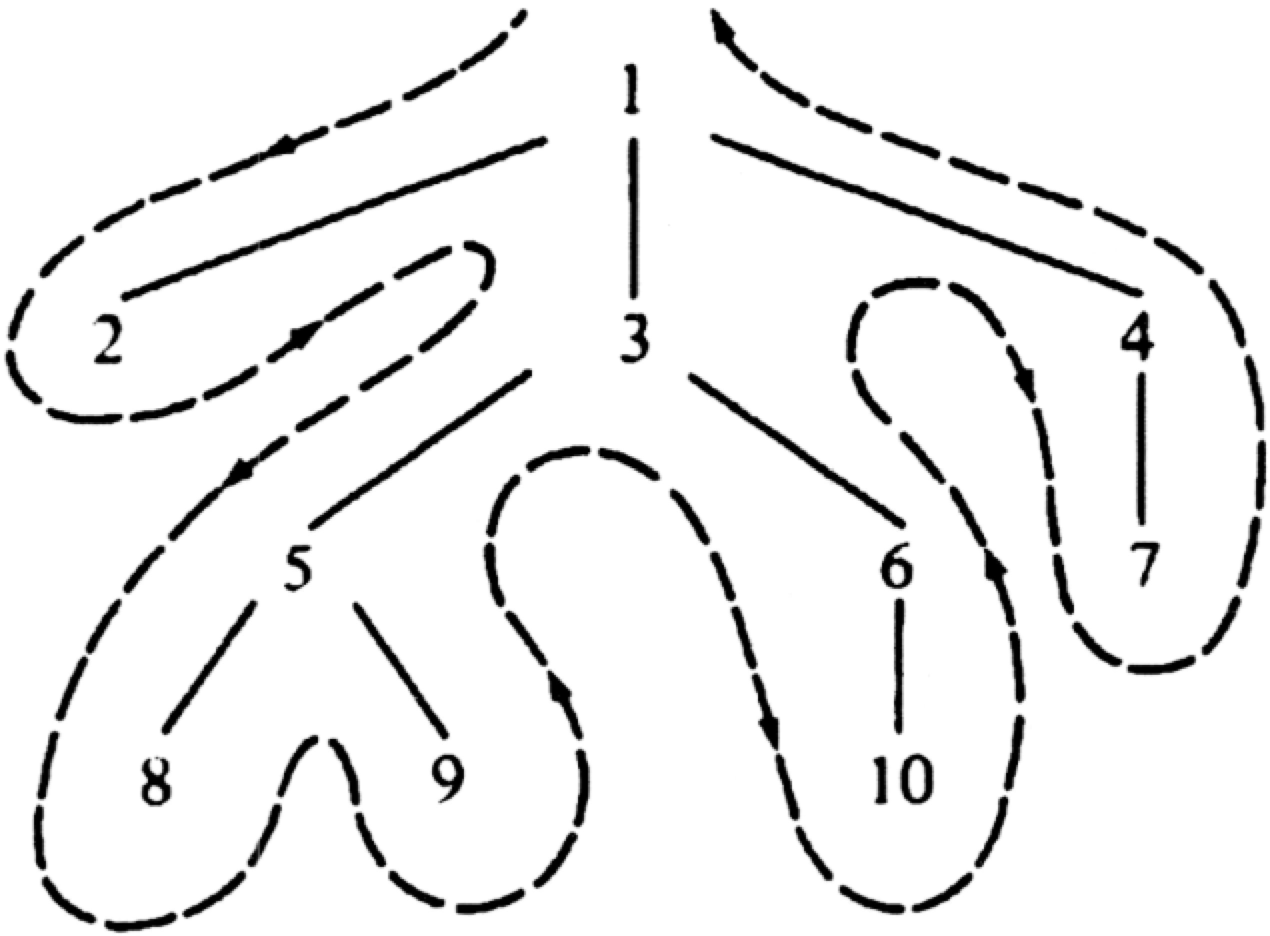
Terminology – Traversing

void tree::INORDER (*n*: node)
  if *n* is a leaf **then**
    list *n*
  else begin
    INORDER(leftmost child of *n*)
    list *n*
    for each child *c* of *n*, except for the leftmost, in order from the left **do**
      INORDER(*c*)
  endelse
endmethod { INORDER }

Terminology – Traversing

- Useful trick: walk around the outside of the tree, starting at the root, moving counterclockwise
- For preorder, we list a node the first time we pass it. For postorder, we list a node the last time we pass it, as we move up to its parent. For inorder, we list a leaf the first time we pass it, but list an interior node the second time we pass it.

Terminology – Traversing

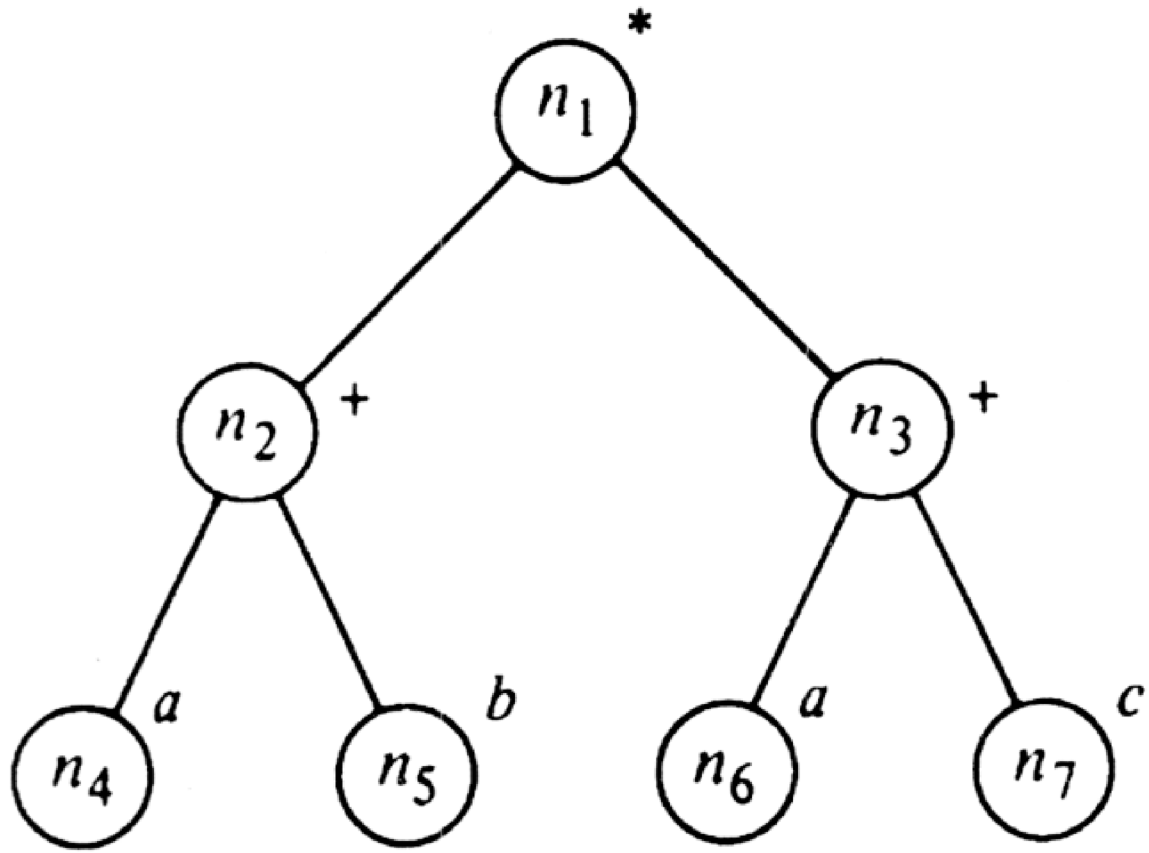


Terminology – Labels

- Often it is useful to associate a *label*, or value, with each node of a tree, in the same spirit with which we associated a value with a list element in the previous chapter. That is, the label of a node is not the name of the node, but a value that is "stored" at the node.

Terminology – Labels

- Example: labeled tree representing the arithmetic expression (*a*+*b*) * (*a*+*c*)



Terminology – Labels

- Example:
  - Preorder: prefix form of an expression

    *+*ab*+*ac*

  - Postorder: *postfix* (or *Polish*) representation of an expression

    ab+*ac*+*

  - Inorder: infix expression itself (with no parentheses)

    a+*b* * *a*+*c*

The ADT tree

```
spec TREE[NODE]
     genres tree, node, label
     operations
          parent: node tree -> node
          leftmost_child: node tree -> node
          right_sibling: node tree -> node
          label: node tree -> label
          create: label tree tree -> tree
          root: tree -> node
          makenull: tree -> tree
endspec
```

The ADT tree

- PARENT(*n*, *T*). This function returns the parent of node *n* in tree *T*. If *n* is the root, which has no parent, Λ is returned.
- LEFTMOST_CHILD(*n*, *T*) returns the leftmost child of node *n* in tree *T*, and it returns Λ if *n* is a leaf, which therefore has no children.
- RIGHT_SIBLING(*n*, *T*) returns the right sibling of node *n* in tree *T*, defined to be that node *m* with the same parent *p* as *n* such that *m* lies immediately to the right of *n* in the ordering of the children of *p*.

The ADT tree

- LABEL(n, T) returns the label of node n in tree T. We do not, however, require labels to be defined for every tree.
- CREATEi(v, T_1, T_2, . . . , Ti) is one of an infinite family of functions, one for each value of i = 0, 1, 2, ... CREATEi makes a new node r with label v and gives it i children, which are the roots of trees T_1, T_2, . . . , Ti, in order from the left. The tree with root r is returned. Note that if i = 0, then r is both a leaf and the root.

The ADT tree

void tree::PREORDER (*n*: node)

    {list the labels of the descendants of *n* in preorder}

    var *c*: node

    print(LABEL(*n*, *T*))

    c := LEFTMOST_CHILD(*n*, *T*)

    while *c* <> null **do**

        PREORDER(*c*)

        c := RIGHT_SIBLING(*c*, *T*)

    endwhile

endproc { PREORDER }

We call PREORDER(ROOT(T)) to get the preorder of tree *T*

Implementations of trees

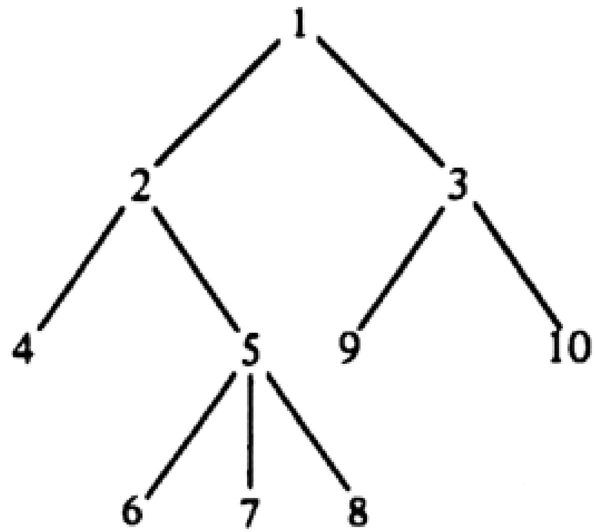
- We are going to present three different implementations:
  - Array representation
  - Representation by list of children
  - Leftmost-child, right-sibling representation

- We are only going to consider the third one for our implementations

Implementation of trees

- Array representation
  - Linear array *A* in which entry *A*[*i*] is a pointer or a cursor to the parent of node *I*
  - *A*[*i*] = 0 if node *i* is the root
  - This representation uses the property of trees that each node has a unique parent
  - It does not facilitate operations that require information of children

Implementation of trees



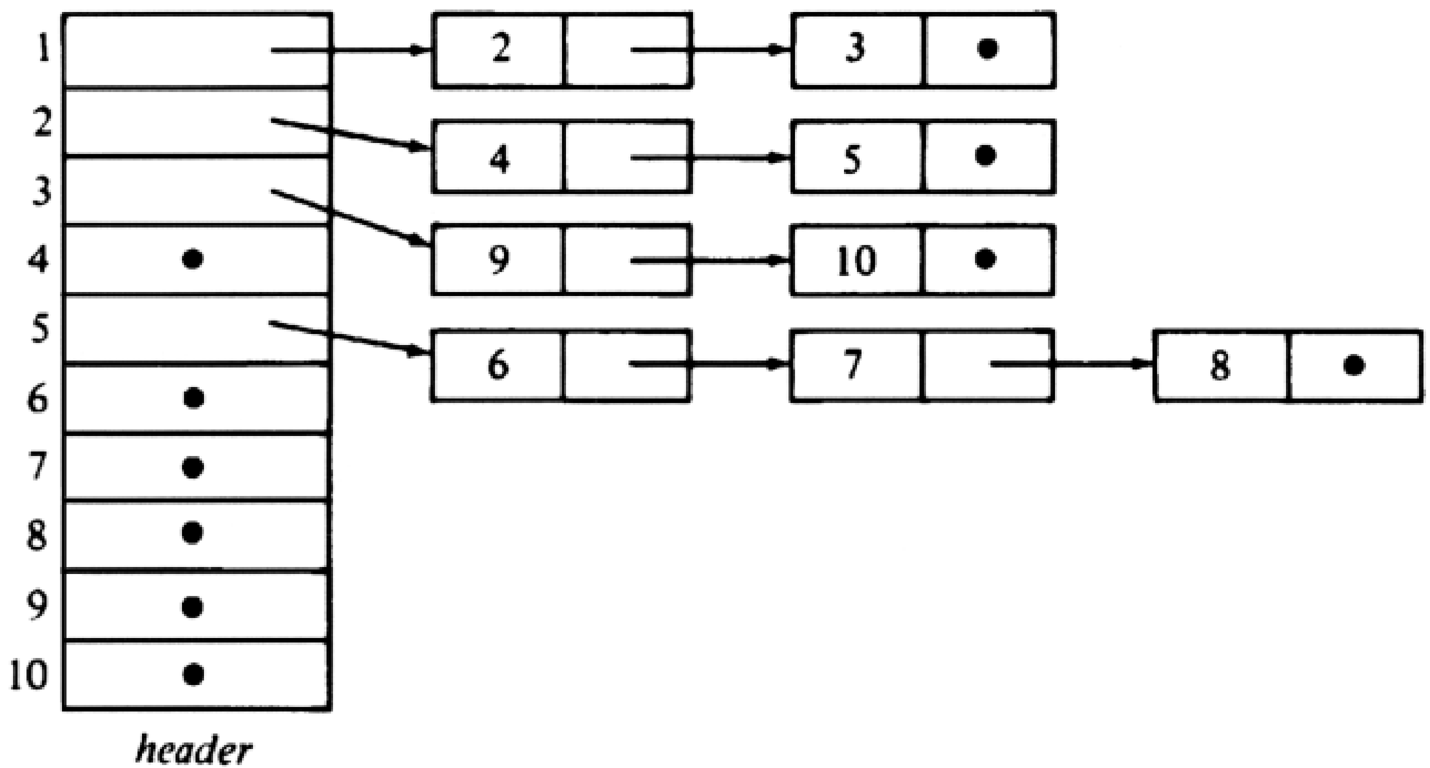
(a) a tree

	1	2	3	4	5	6	7	8	9	10
A	0	1	1	2	2	5	5	5	3	3

(b) parent representation.

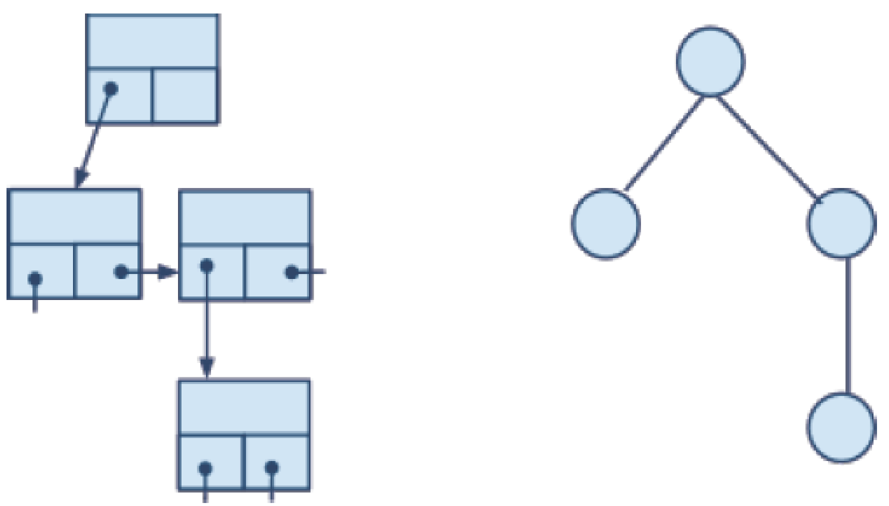
Implementation of trees

- Representation by list of children
  - form for each node a list of its children

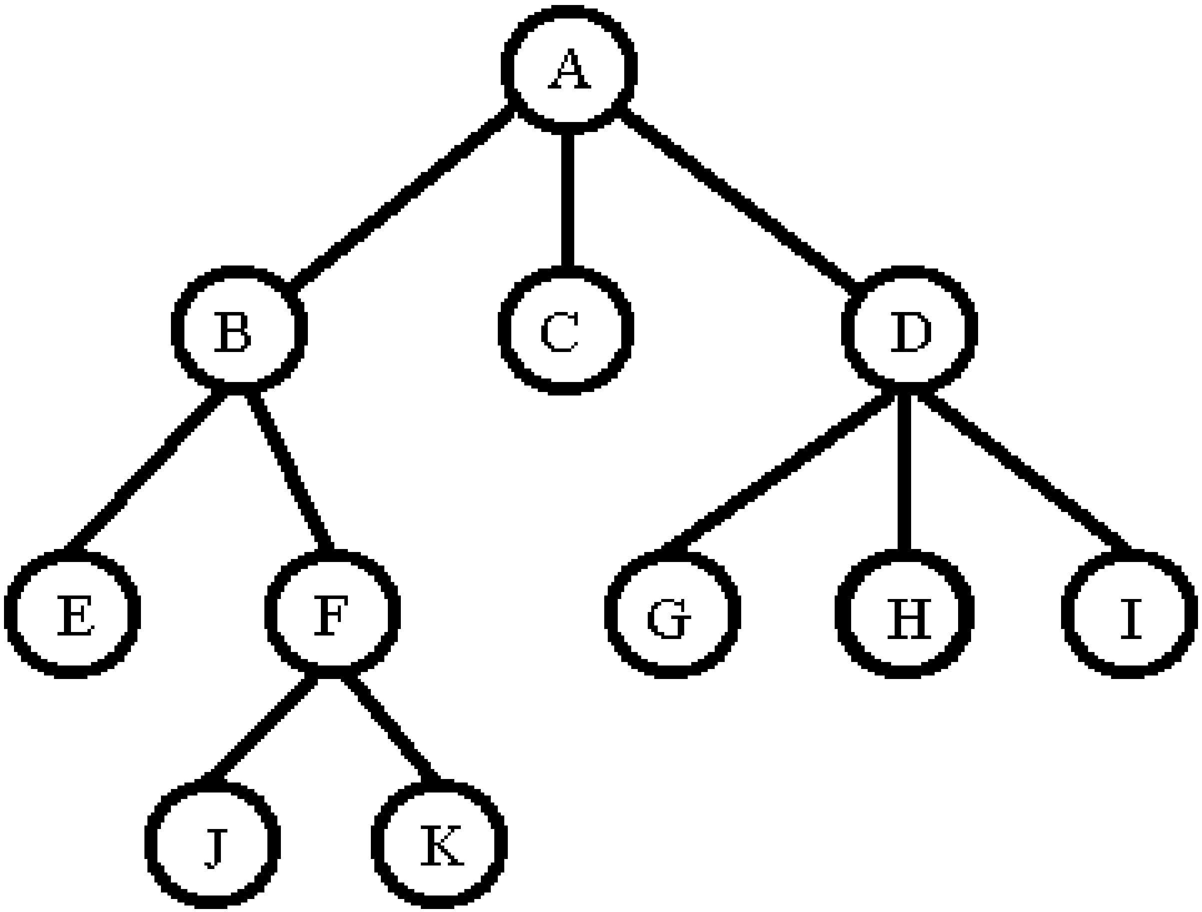


Implementations of trees

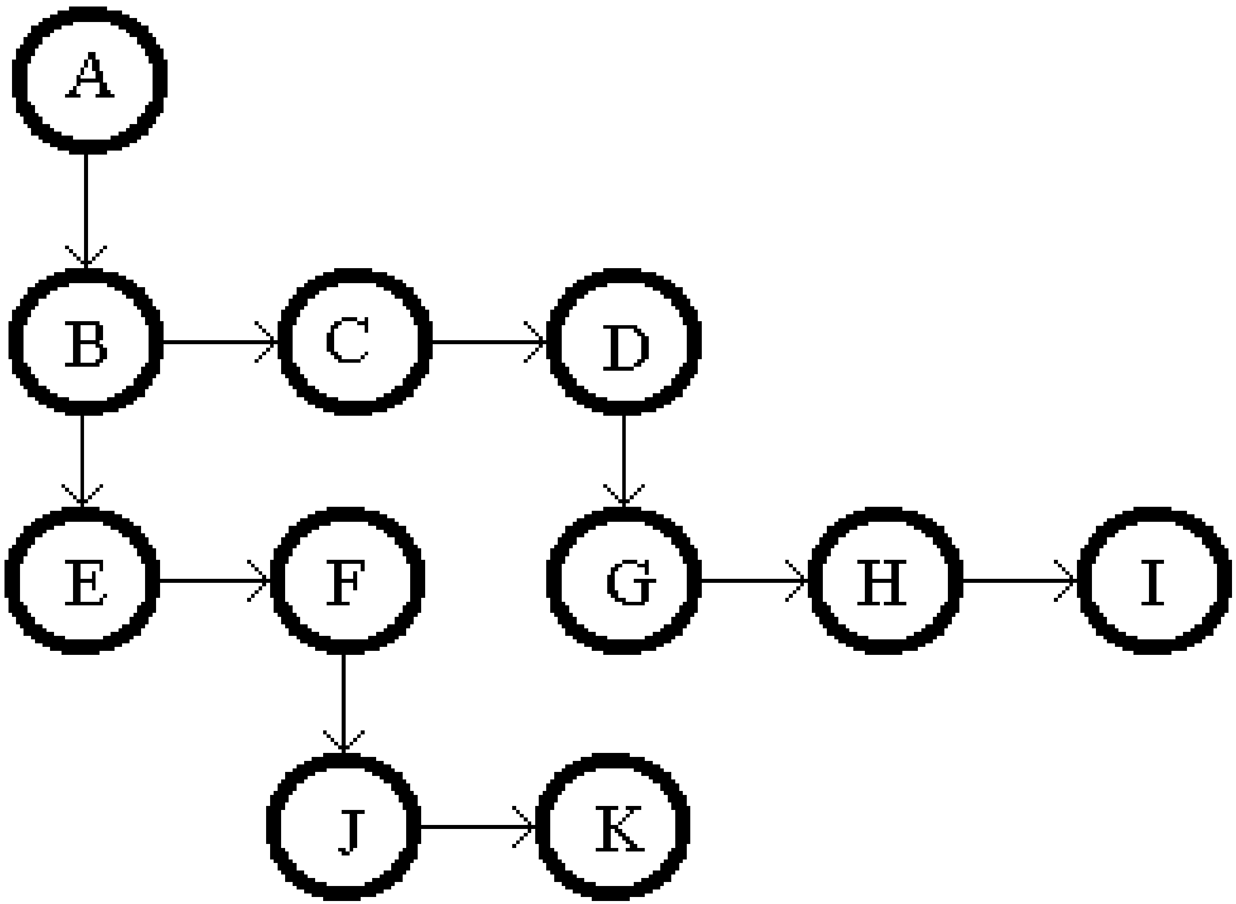
- Leftmost-child, right-sibling representation
  - Each node has reference to its leftmost child and right sibling only.
  - Each leaf has a null for leftmost child and each rightmost child has a null for right sibling reference.



Implementation of trees



Implementation of trees



Implementations of trees

```
node = record
    element: label
    leftmostchild: ^node
    rightsibling: ^node
endrecord

tree: ^node {or a full class}
label: elementtype
```

Implementations of trees

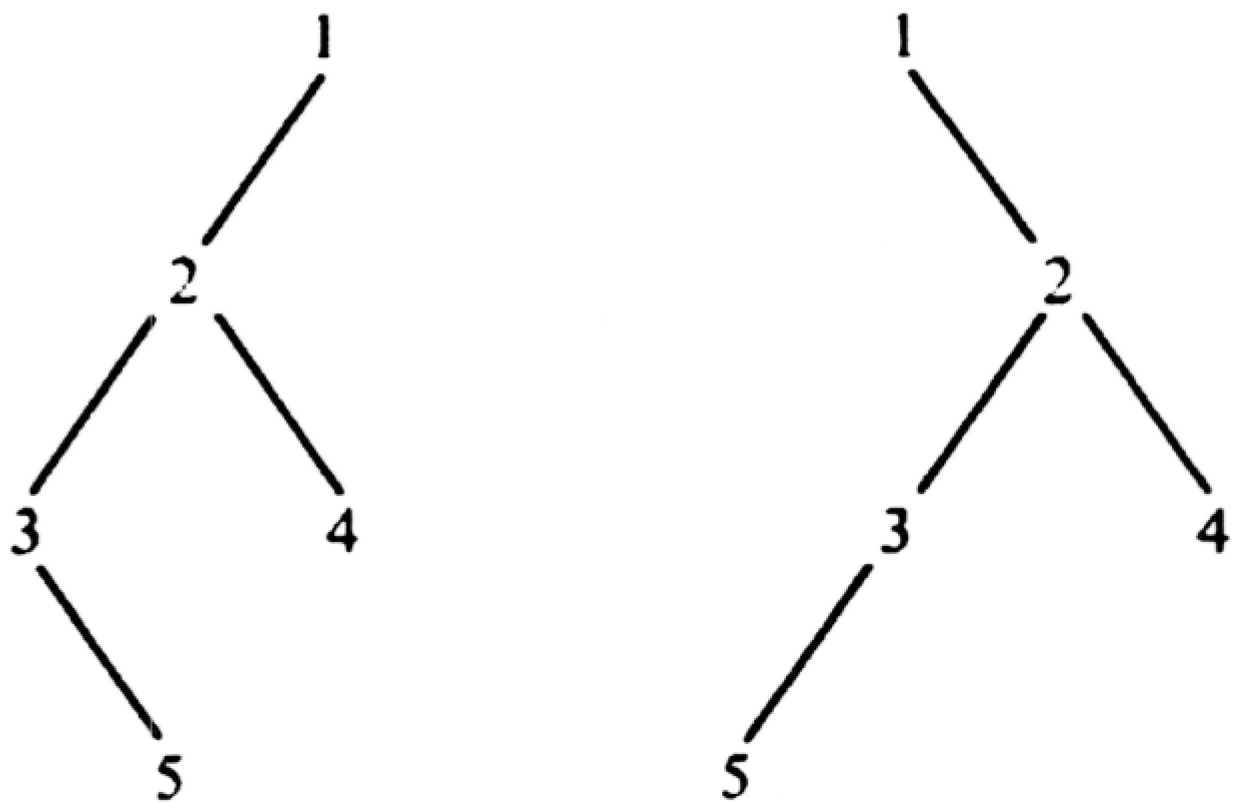
- Running time of operations
  - parent – O(n)
  - leftmost_child – O(1)
  - right_sibling – O(1)
  - label – O(1)
  - create – O(1)
  - makenull – O(1)
    - O(n) to dipose every element – traverse the tree (postorder)
  - root – O(1)

Binary trees

- *binary tree*, which is either an empty tree, or a tree in which every node has either no children, a *left child*, a *right child*, or both a left and a right child.
- The fact that each child in a binary tree is designated as a left child or as a right child makes a binary tree different from the ordered, oriented tree (alse called “ordinary” tree or “general” tree)

Binary trees

- Two different binary trees



The ADT binary tree

```
spec BINARY_TREE[NODE]
     genres b_tree, node, label
     operations
          parent: node b_tree -> node
          left_child: node b_tree -> node
          right_child: node b_tree -> node
          label: node b_tree -> label
          create: b_tree b_tree -> tree
          root: b_tree -> node
          makenull: b_tree -> b_tree
endspec
```

The ADT binary tree

```
node = record
    element: label
    leftchild: ^node
    rightchild: ^node
    parent: ^node {optional}
endrecord

b_tree: ^node {or a class}
label: elementtype
```

Binary trees

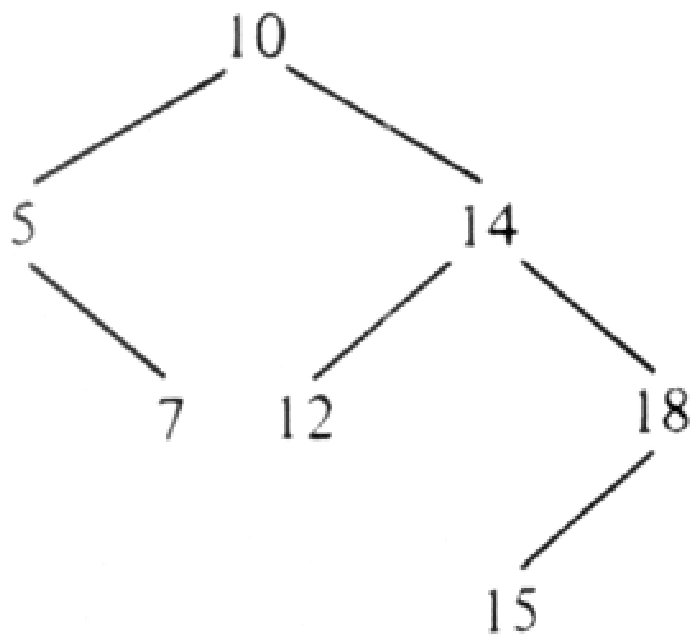
- Running time of operations
  - parent – O(1)
  - left_child – O(1)
  - right_child – O(1)
  - label – O(1)
  - create – O(1)
  - makenull – O(1)
    - O(n) to dipose every element – traverse the tree (postorder)
  - root – O(1)

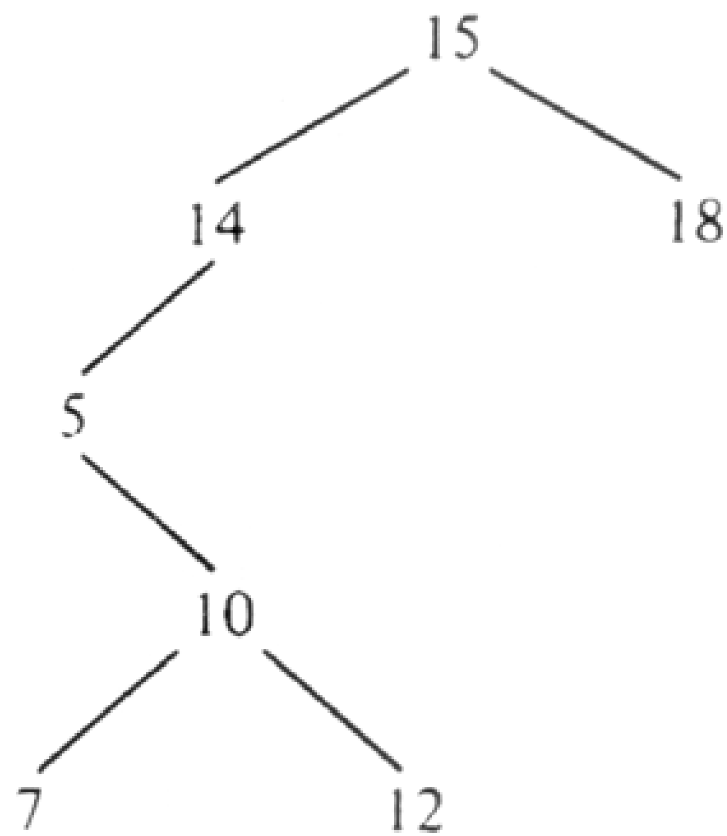
Binary search trees

- A *binary search tree* (BST) is a binary tree in which:
  (1) the nodes are labeled with elements of a set.
  (2) all elements stored in the left subtree of any node *x* are all less than the element stored at *x*, and all elements stored in the right subtree of *x* are greater than the element stored at *x*.
- This condition, called the *binary search tree property*, holds for every node of a binary search tree, including the root.
- BSTs are also called an ordered or sorted binary tree.

Binary search trees

- two binary search trees representing the same set of integers





Binary search trees

- Interesting property: if we list the nodes of a binary search tree in inorder, then the elements stored at those nodes are listed in sorted order.
- Operations on a binary search tree require comparisons between nodes. These comparisons are made with calls to a comparator, which is a subroutine that computes the total order (linear order) on any two values. This comparator can be explicitly or implicitly defined, depending on the language in which the BST is implemented.

The ADT BST

```
spec BINARY_SEARCH_TREE[NODE]
    genres bst, node, label
    operations
        search: label BST -> boolean
        insert: label BST -> BST
        delete: label BST -> BST
endspec
```

The ADT BST

```
node = record
    element: label
    leftchild: ^node
    rightchild: ^node
endrecord

bst: ^node {or a class}
label: elementtype
```

Binary search trees

- **SEARCH** (a.k.a member). Examine the root node:
  - If the tree is null, the value we are searching for does not exist in the tree.
  - If the value equals the root, the search is successful.
  - If the value is less than the root, search the left subtree.
  - Similarly, if it is greater than the root, search the right subtree.

  This process is repeated until the value is found or the indicated subtree is null.

Binary search trees

- **INSERT(x, T)**
  - Test whether T = null, that is, whether the BST is empty. If so, we create a new node to hold x and make A point to it.
  - If the BST is not empty, we search for x more or less as SEARCH does, but when we find a null pointer during our search, we replace it by a pointer to a new node holding x. Then x will be in the right place, namely, the place where the function SEARCH will find it.

Binary search trees

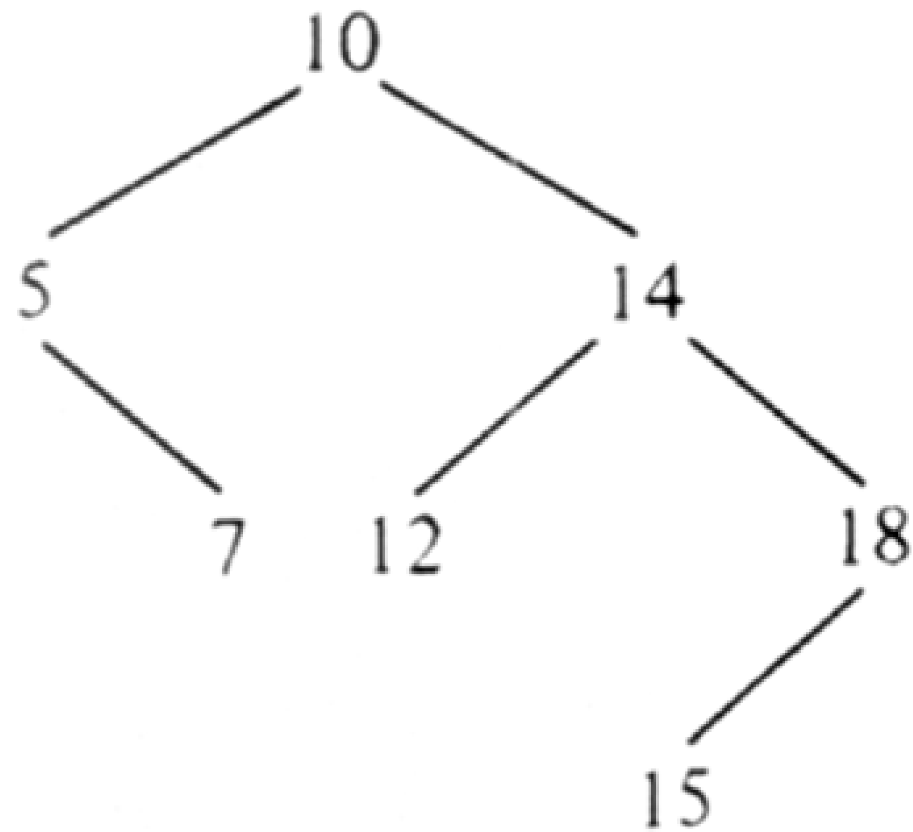
- **DELETE(x,A):**
  - Locate the element *x* to be deleted in the tree.
  - **If *x* is at a leaf**, we can delete that leaf and be done.
    - If it is an interior node, deleting it would disconnect the tree.
  - **If *x* has only one child**, we can replace *x* by that child, and we shall be left with the appropriate BST.
  - **If *x* has two children**, then we must find the lowest-valued element among the descendants of the right child and replace the node to be deleted with it
    - The highest-valued descendant among the descendants on the left would also do as well

Binary search trees

- To write DELETE, it is useful to have a function DELETEMIN(A) that removes the smallest element from a nonempty tree and returns the value of the element removed.

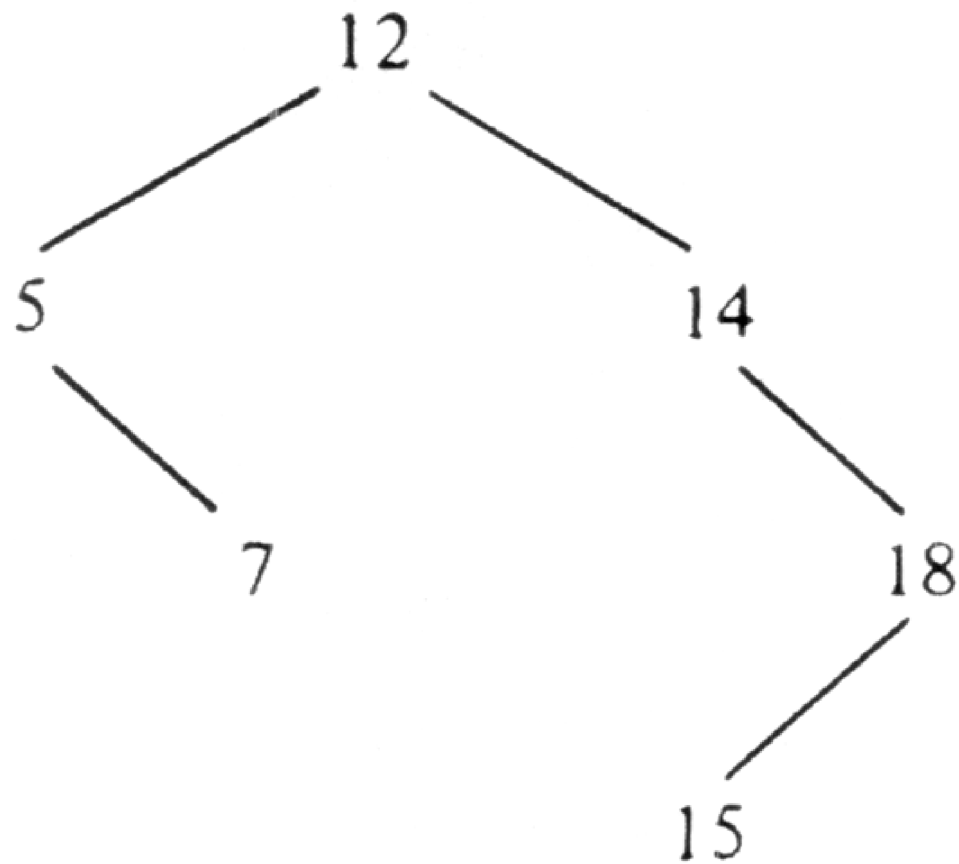
Example

- Delete 10 from the following BST



Example

- the lowest-valued element among the descendants of the right child is 12



Binary search trees

- Running times of BST's operations.

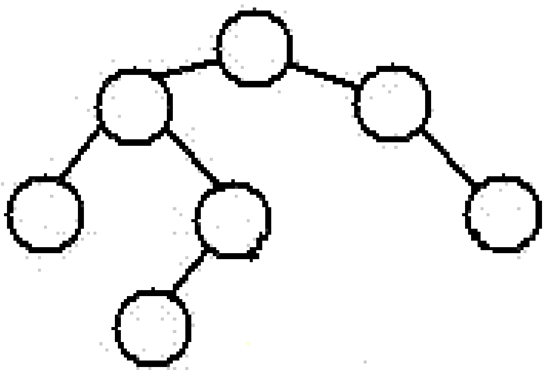
Operation	Average	Worst case
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

- Time analysis of BSTs
  – in (AHO, HOPCROFT & ULLMAN, 1987) "Data Structures and Algorithms." Chapter 5. Section 5.2.

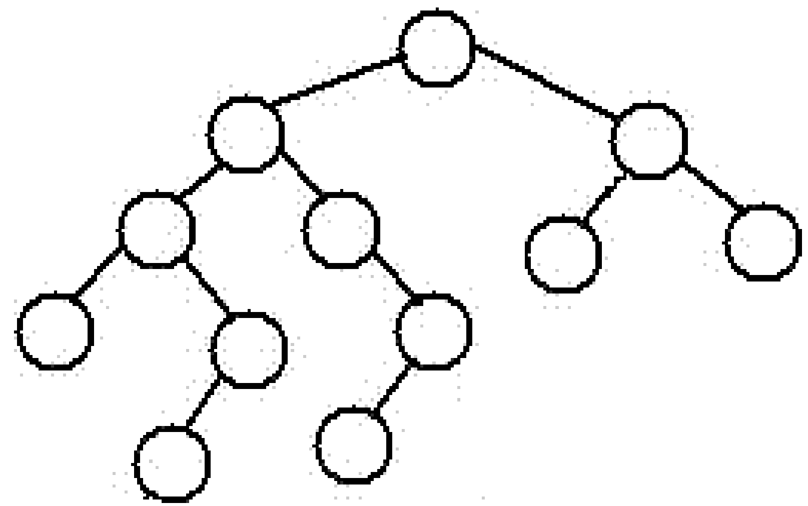
Balanced trees

- A (height) **balanced tree** is a tree where no leaf is much farther away from the root than any other leaf.
- Definition: An empty tree is balanced.
  A non-empty binary tree T is balanced if:
  1. Left subtree of T is balanced
  2. Right subtree of T is balanced
  3. The difference between heights of left subtree and right subtree is not more than 1.

Balanced trees



A height-balanced Tree



Not a height-balanced tree

Balanced trees

- **Complete tree** – A tree in which every level, except possibly the deepest, is entirely filled. At depth n, the height of the tree, all nodes are as far left as possible.

- A complete tree is balanced, but a balanced tree is not necessarily complete.

Balanced trees

- On a BST, some sequences of insertions and deletions can produce binary search trees whose average depth is proportional (or close) to *n*. This suggests that we might try to rearrange the tree after each insertion and deletion so that it is always balanced or complete; then the time for SEARCH and similar operations would always be O(log n).

Balanced trees

- Balanced implementations of trees:
  - AVL trees
  - 2-3 trees
  - red-black trees
  - B trees (B+ trees)
  - T-trees

AVL trees

- An **AVL tree** is a self-balancing binary search tree
  - Named after its two inventors, G.M. Adelson-Velskii and E.M. Landis (1962)
- The **balance factor** of a node is the height of its left subtree minus the height of its right subtree (sometimes opposite)
- A node with balance factor 1, 0, or −1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree. The balance factor is usually stored directly at each node.

AVL trees

- Same ADT as a BST
  - Same ops: Search, Insert, Delete
- The data structure needs to incorporate an integer on every node to store the balance factor.
  - Balance factor = height leftchild – height rightchild
- **SEARCH** is performed exactly as in an unbalanced binary search tree.
  - the tree's structure is not modified by lookups

AVL trees

- **INSERT** – After inserting a node, it is necessary to check each of the node's ancestors for consistency with the rules of AVL. For each node checked, if the balance factor remains −1, 0, or +1 then no rotations are necessary. However, if the balance factor becomes ±2 then the subtree rooted at this node is unbalanced.

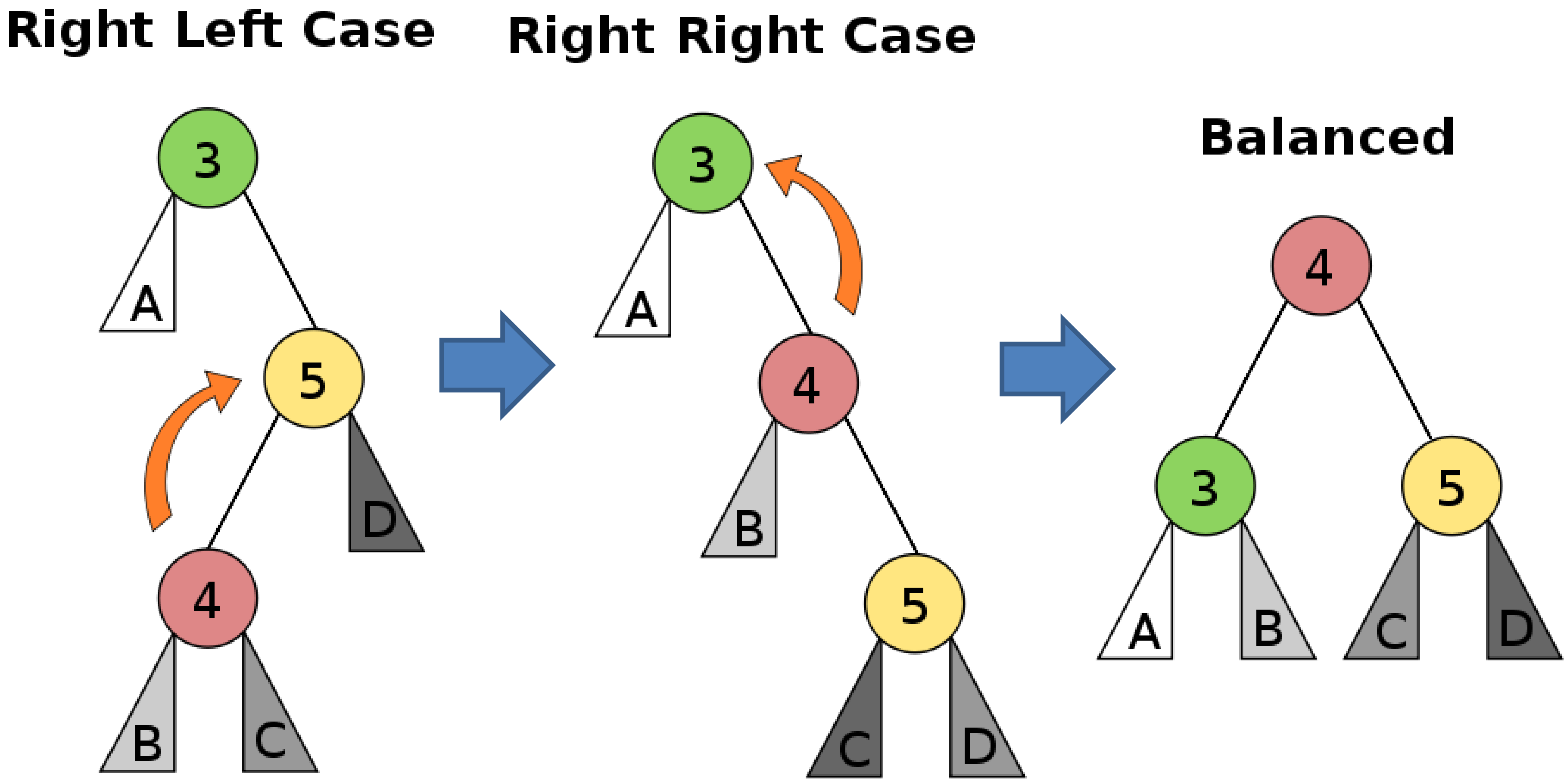
AVL trees

- **INSERT** - four cases which need to be considered
  - Right-Right case
  - Right-Left case
  - Left-Left case
  - Left-Right case
- Balance factors determine which case we are dealing with.

AVL trees

- **Right-Right case** and **Right-Left case**:
  - If the balance factor of a node (P) is -2 then the right subtree outweights the left subtree of the given node, and the balance factor of the right child (R) must be checked. The left rotation with P as the root is necessary.
  - If the balance factor of R is -1 or 0, a **single left rotation** (with P as the root) is needed (**Right-Right case**).
  - If the balance factor of R is +1, two different rotations are needed. The first rotation is a **right rotation** with R as the root. The second is a **left rotation** with P as the root (**Right-Left case**).

AVL trees

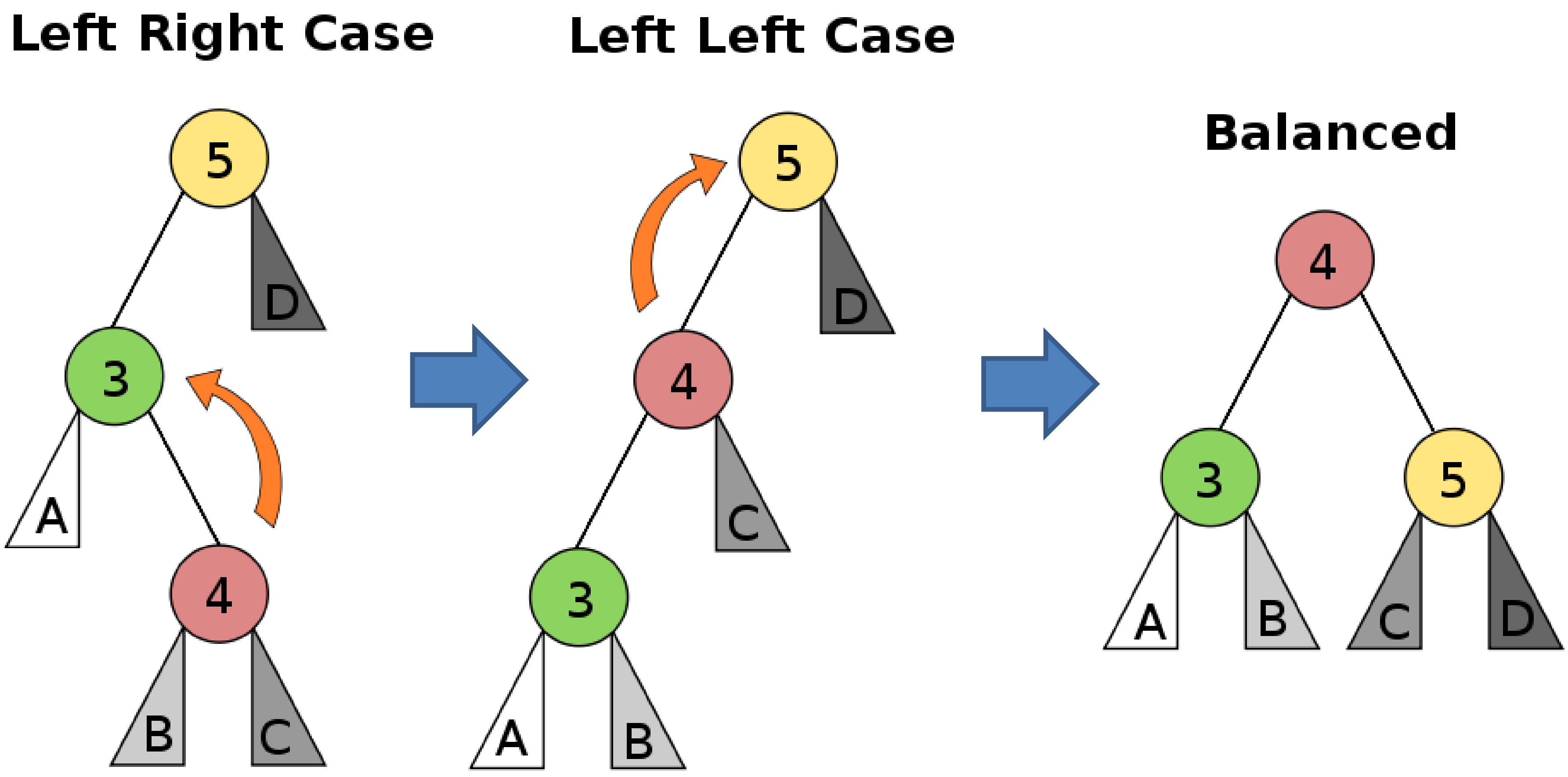


Right Left Case
Right Right Case
Balanced
3
A
5
D
4
B
C
3
A
4
B
5
C
D
4
3
5
A
B
C
D

AVL trees

- **Left-Left case** and **Left-Right case**:
  - If the balance factor of a node (P) is +2, then the left subtree outweighs the right subtree of the given node, and the balance factor of the left child (L) must be checked. The right rotation with P as the root is necessary.
  - If the balance factor of L is +1 or 0, a **single right rotation** (with P as the root) is needed (**Left-Left case**).
  - If the balance factor of L is -1, two different rotations are needed. The first rotation is a **left rotation** with L as the root. The second is a **right rotation** with P as the root (**Left-Right case**).

AVL trees



Left Right Case
Left Left Case
Balanced
5
D
3
A
4
B
C
5
D
4
C
3
A
B
4
3
5
A
B
C
D

AVL trees

- **DELETE**
  - If the node is a leaf or has only one child, remove it.
  - Otherwise, replace it with either the largest in its left subtree (inorder predecessor) or the smallest in its right subtree (inorder successor), and remove that node. (Same as on a BST)
- The node that was found as a replacement has at most one subtree. After deletion, retrace the path back up the tree (parent of the replacement) to the root, adjusting the balance factors as needed. Rebalance as in insertion if necessary.

AVL trees

- Running times of operations on an AVL tree

Operation	Average	Worst case
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Trees

Luis de Marcos Ortega

luis.demarcos@uah.es