BM267 - Introduction to Data Structures

5. Trees

Ankara University

Computer Engineering Department

Bulent Tugrul

Objectives

Learn about the definitions, characteristics and implementation details for:

- General trees
- Rooted trees
- Binary and N-ary trees
- Tree operations, tree traversal algorithms

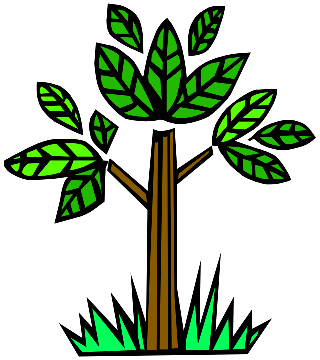
Tree Structures

One of the most frequently used ordering methods of data.

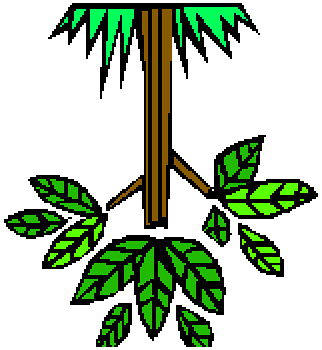
Many logical organizations of everyday data exhibit tree structures

- Promotional tournaments
- Organizational charts
- Hierarchical organization of entities
- Parsing natural and computer languages
- Game trees
- Decision trees
- ...

Trees



A real tree



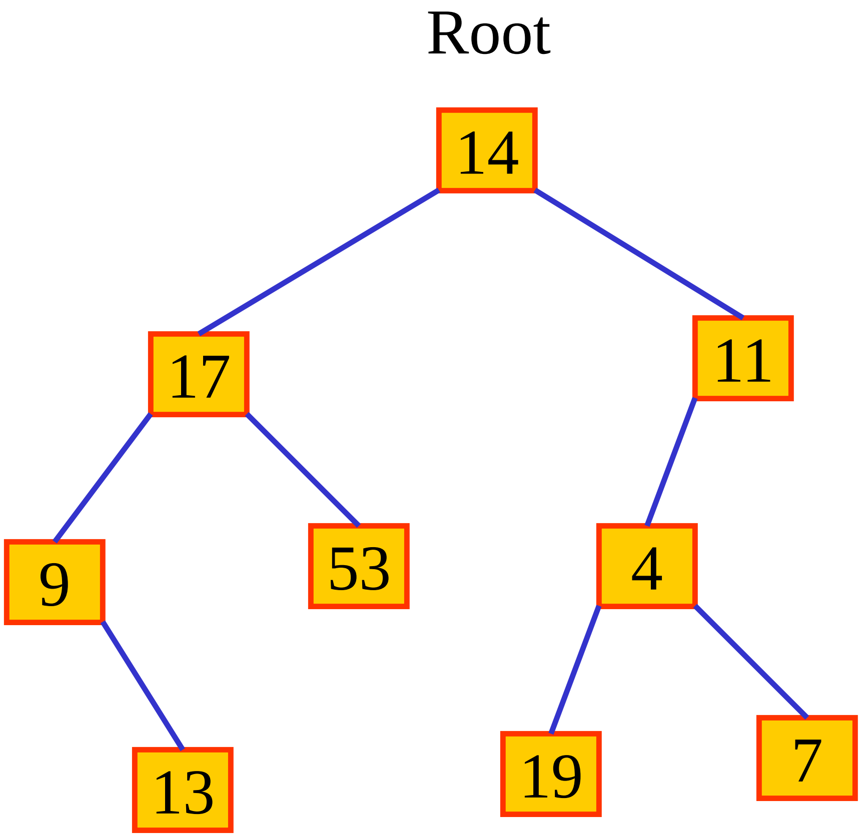
Computer Scientist's tree

Trees

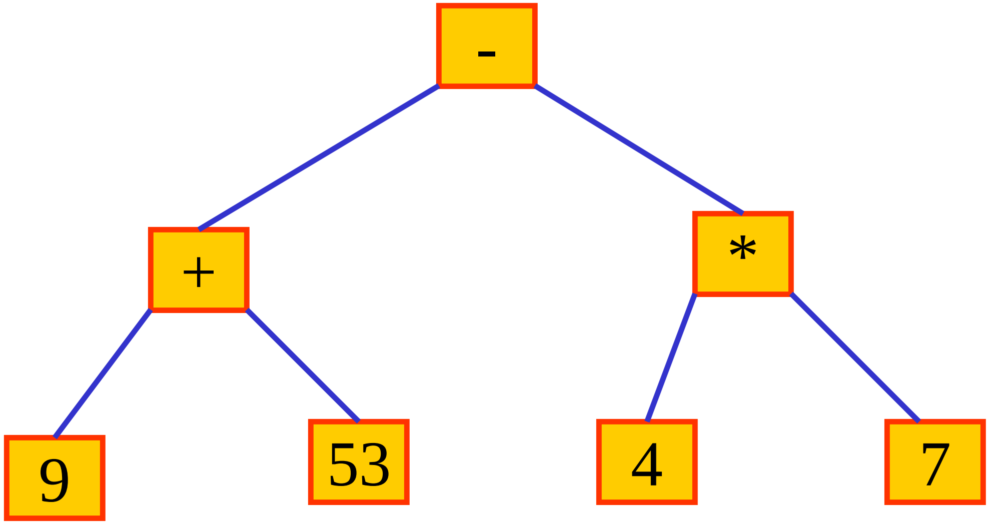
A general **tree** is a nonempty collection of vertices (**nodes**) and connections between nodes (**edges**) that satisfy certain rules. These rules impose a hierarchical structure on the nodes with a parent-child relation.

There is only one connecting path between any two nodes.

Trees



Trees

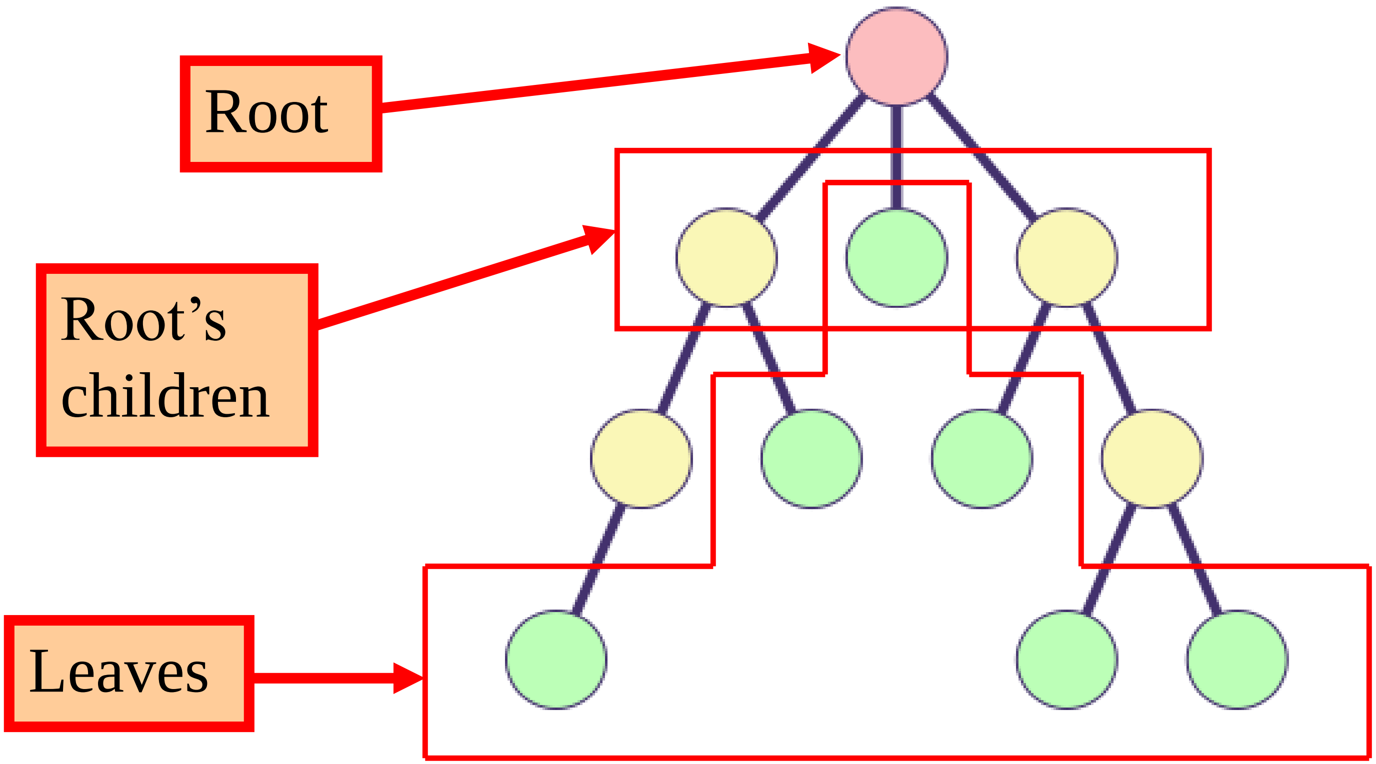


= (9 + 53) – (4 * 7)

Rooted Trees

- There is a unique node called **root node**. Node “14” is the root of the tree.
- The **parent** of a node is the node linked above it. Every non-root node has a unique **parent**. Node 17 is the parent of 9 and 53.
- The nodes whose parent is node n are n’s **children**. The children of Node 17 are 9 and 53.
- Nodes without children are **leaves**. Nodes 13, 53, 19, and 7 are leaves.
- Two nodes are **siblings** if the have the same parent. 9 and 53 are siblings of each other.

Rooted Trees

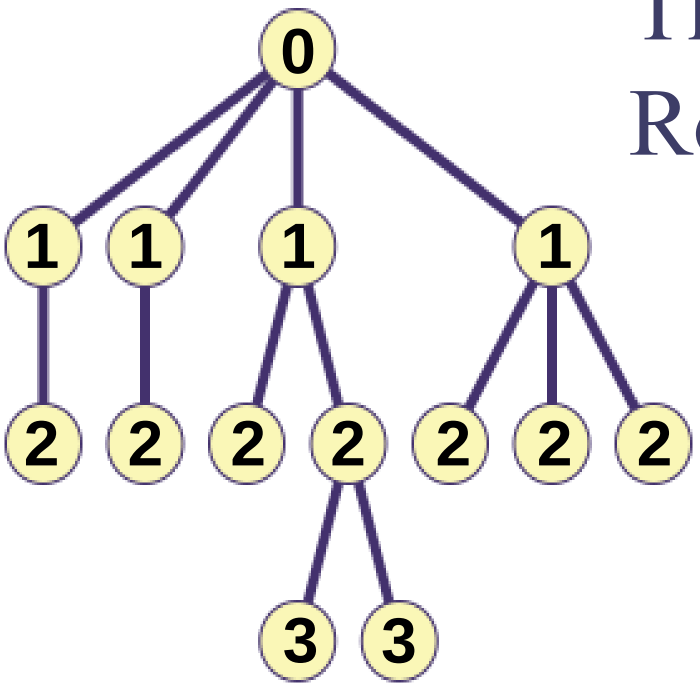


Rooted Trees

- An **empty tree** has no nodes
- The **descendants** of a node are its children and the descendents of its children
- The **ancestors** of a node are its parent (if any) and the ancestors of its parent
- An **ordered tree** is one in which the order of the children is important; an **unordered tree** is one in which the ordering of the children is not important.
- The **branching factor** of a node is the number of children it has.

Rooted Trees

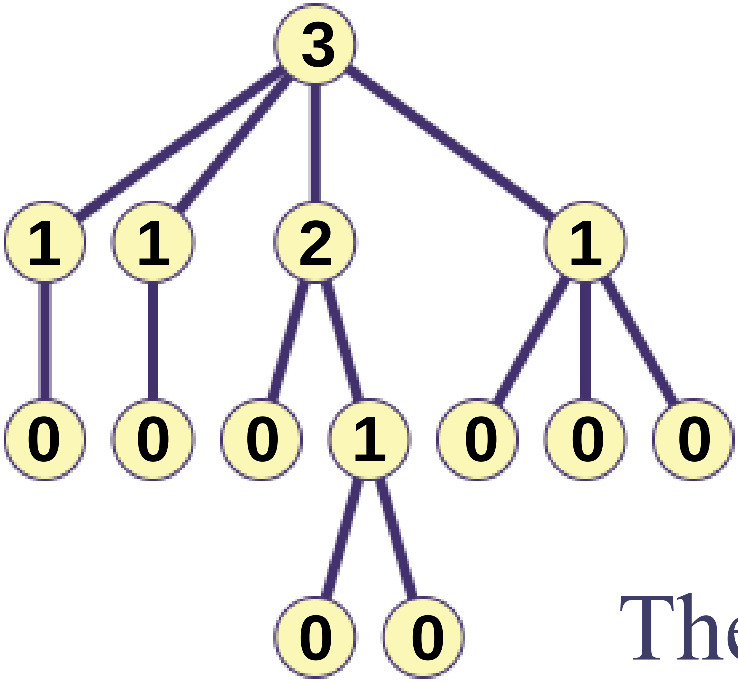
The **depth** or **level** of a node n is the number of edges on a path from the root to n.



The depth of the root is 0.
Root is at level 0.

Rooted Trees

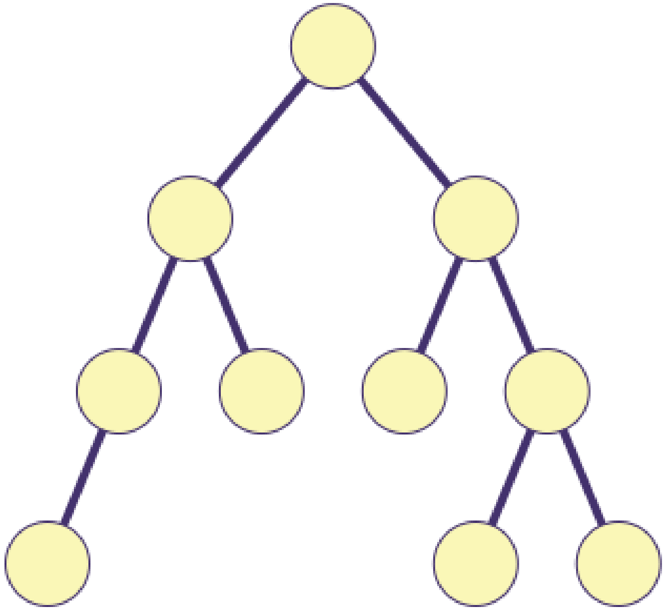
The **height** of a node n is the number of edges on the longest path from n to a descendent leaf.



The height of each leaf is 0.

Binary Trees

A **binary tree** is a special rooted tree in which every node has at most 2 children.



Children are ordered: every child is explicitly designated as left or right child.

Binary Trees

- The **i-th level** of a binary tree contains all nodes at depth **i**.
- The **height** of a binary tree is the height of its root.
- The i-th level of a binary tree contains at most $\mathbf{2^i}$ nodes.
- A binary tree of height **h** contains at most $\mathbf{2^{h+1}-1}$ nodes.
- A binary tree of height h has at most $\mathbf{2^h}$ leaves.

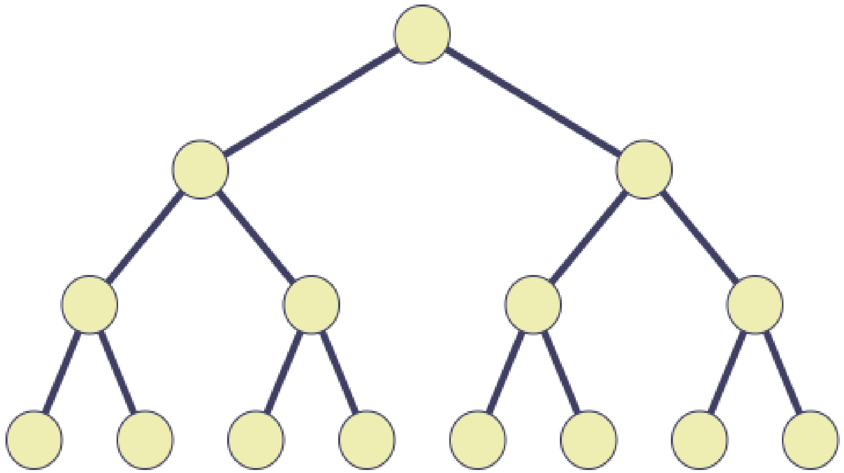
Binary Trees

Level 0

Level 1

Level 2

Level 3



2^0

2^1

2^2

2^3

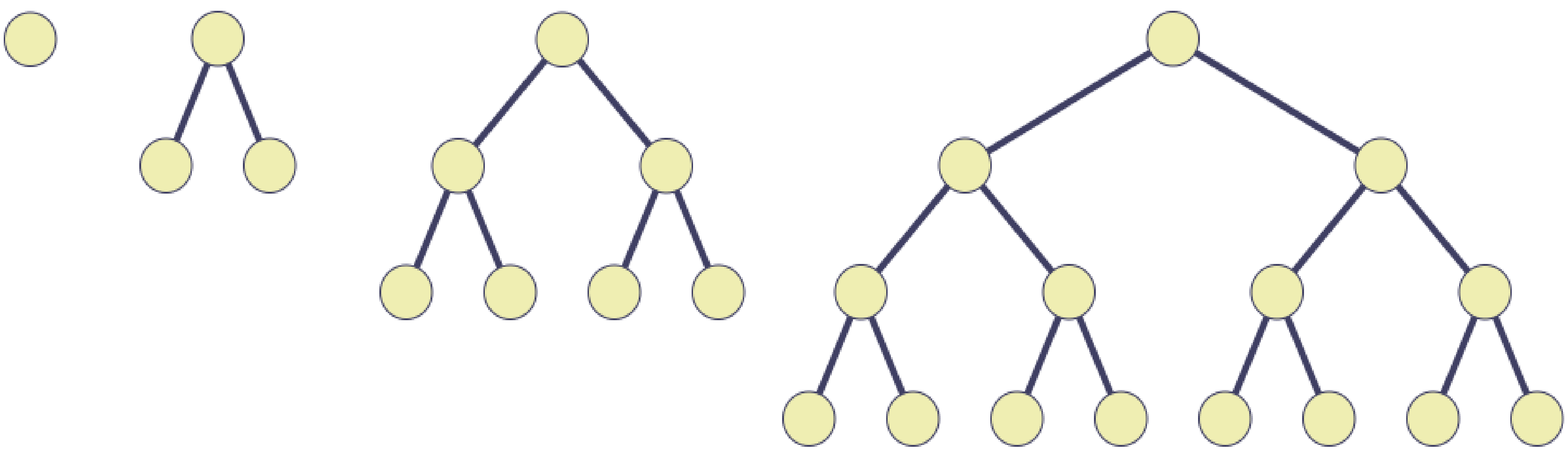
$$\text{Total nodes} = 2^h + 2^{h-1} + \ldots + 2^2 + 2^1 + 2^0$$

$$= \frac{2^{h+1} - 1}{2 - 1}$$

Binary Trees

A binary tree is **complete(perfect)** if:

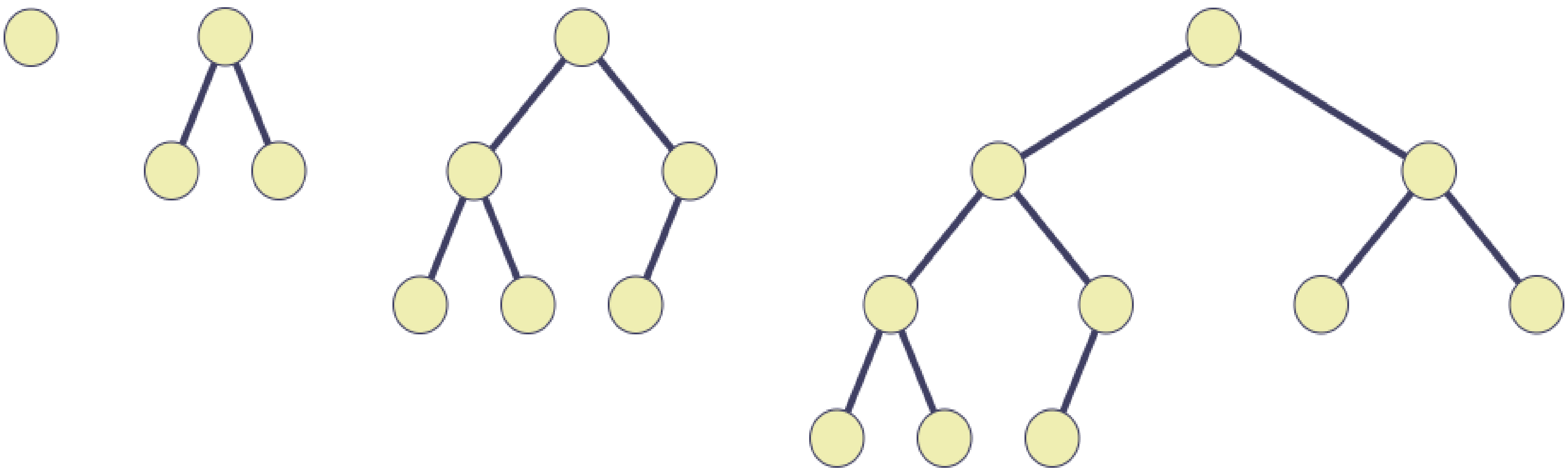
- Every node has either zero or two children. (Every internal node has two children.)
- Every leaf is at the same level.



Binary Trees

A binary tree is **almost complete (perfect)** if

- All levels of the tree are complete, except possibly the last one.
- The nodes on the last level are as far left as possible.



Binary Trees

- A **almost complete** binary tree of height h contains between $\mathbf{2^h}$ and $\mathbf{2^{h+1} - 1}$ nodes.
- A **almost complete** binary tree of size **n** has height h = **floor(log n)**.

$$\mathbf{2^h <= n <= 2^{h+1} - 1}$$

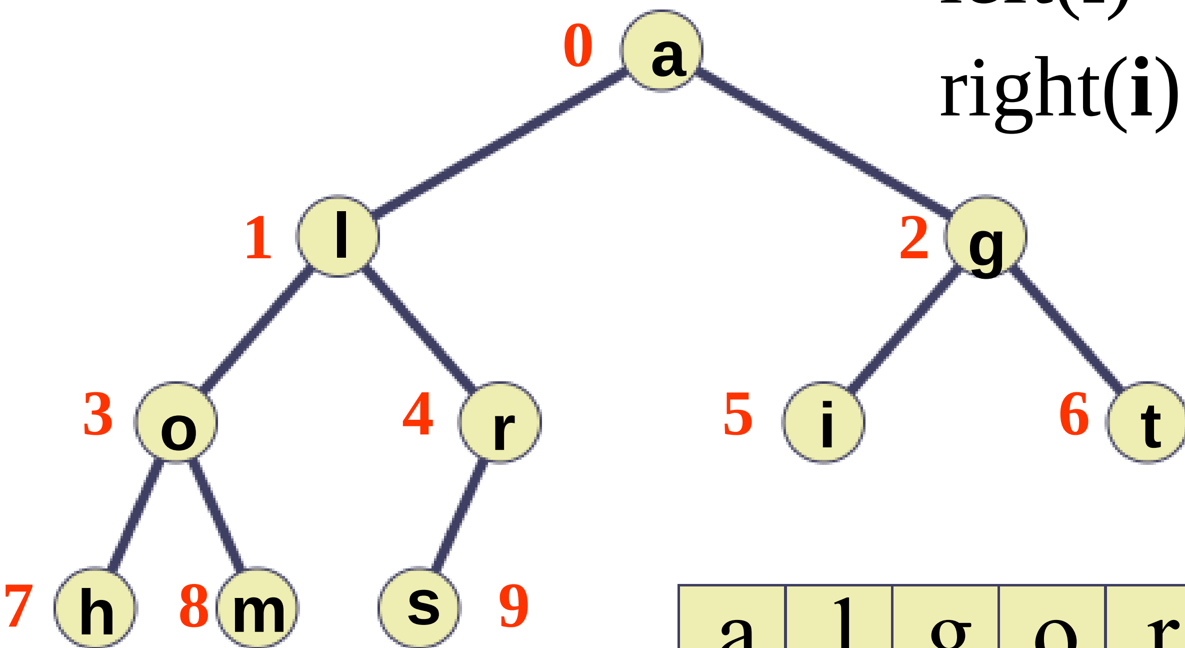
$$h <= \log n < h+1$$

Binary Trees

$\text{parent}(\mathbf{i}) = \mathbf{(i-1) / 2}$ **(integer division)**

$\text{left}(\mathbf{i}) = \mathbf{2i + 1}$

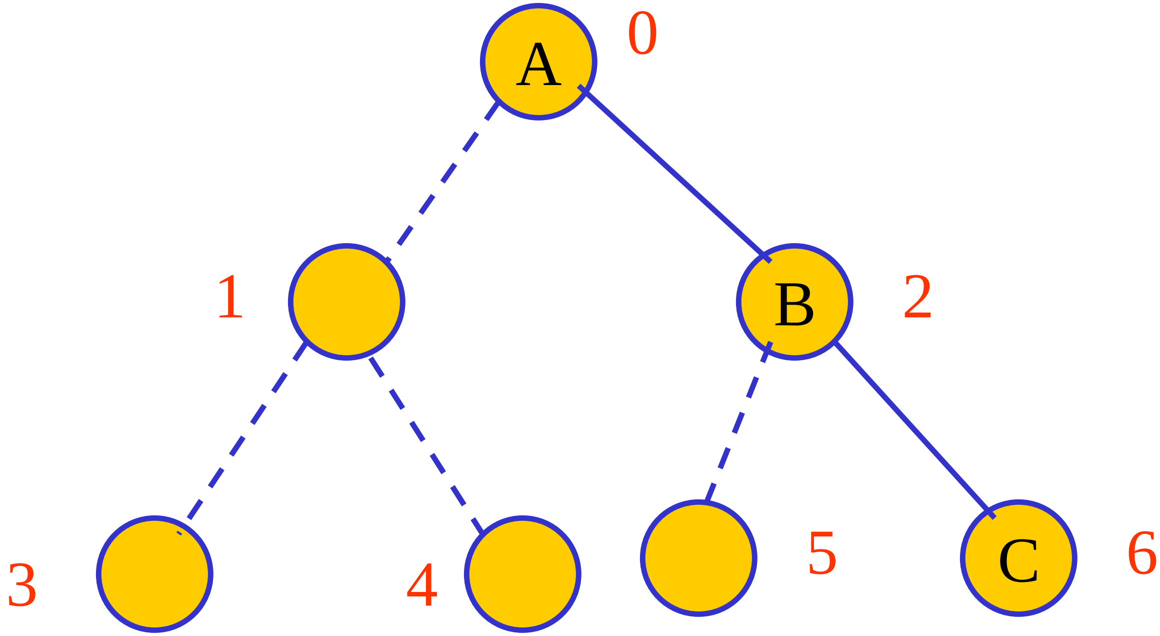
$\text{right}(\mathbf{i}) = \mathbf{2i + 2}$



a	l	g	o	r	i	t	h	m	s
0	1	2	3	4	5	6	7	8	9

Binary Trees

We can also represent incomplete binary trees in an array

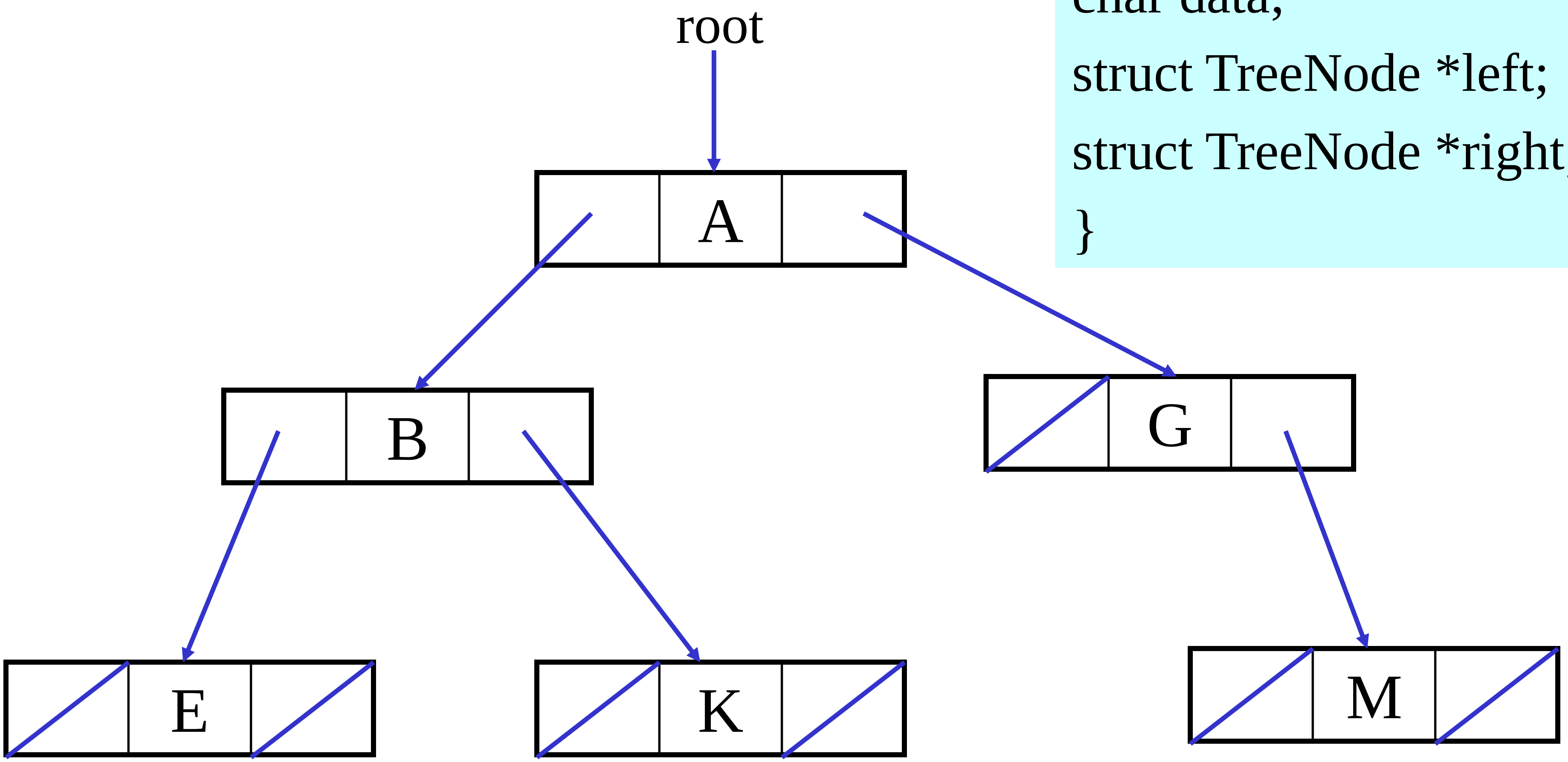


A		B				C
0	1	2	3	4	5	6

Binary Trees

```
struct TreeNode{
char data;
struct TreeNode *left;
struct TreeNode *right;
}
```

Linked representations of binary trees.



Common Binary Tree Operations

- Determine its height
- Determine the number of elements in it
- Display the binary tree on the screen.

Returns the height of the tree.

```
int height(link h)
{     int u, v;
      if (h == NULL)
           return -1;
      u = height(h->l);
      v = height(h->r);
      if (u > v) return u+1;
      else return v+1;   }
```

Binary Trees

Returns the number of elements in the tree.

```
int count(link h){
     if (h == NULL)
          return 0;
     return count(h->l) + count(h->r) + 1;
}
```

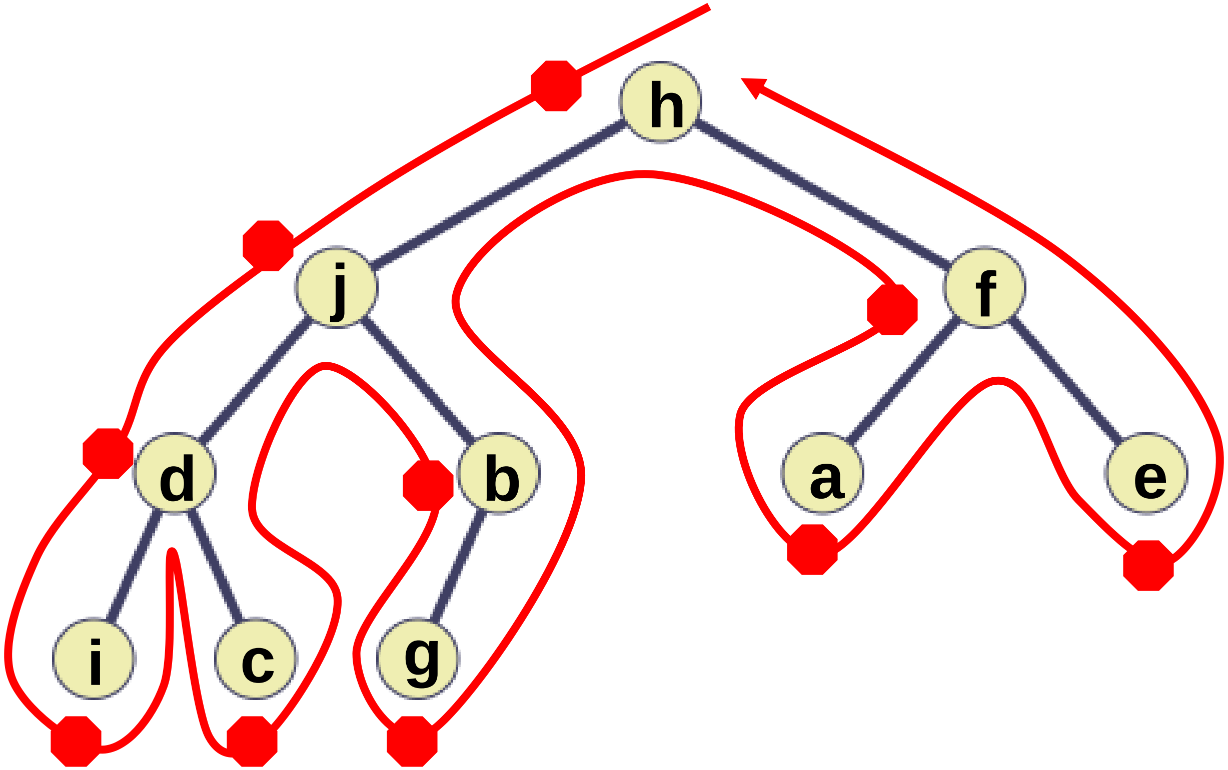
Tree Traversals

- To traverse (or walk) the binary tree is to visit each node in the binary tree exactly once
- Tree traversals are naturally recursive.
- Since a binary tree has two dimensions, there are two possible ways to traverse the binary tree
  - Depth-first - visit nodes on the same path first (start from top, go as far down as possible)
  - Breadth-first - visit nodes at the same level first (start from left, go as far right as possible)

Depth-first Traversals (binary trees)

- Since a binary tree has three “parts,” there are three possible ways to traverse the binary tree (from left to right) :
  - Pre-order: the node is visited first, then the children (left to right)
  - In-order: the left child is visited, then the node, then the right child
  - Post-order: the node is visited after the children

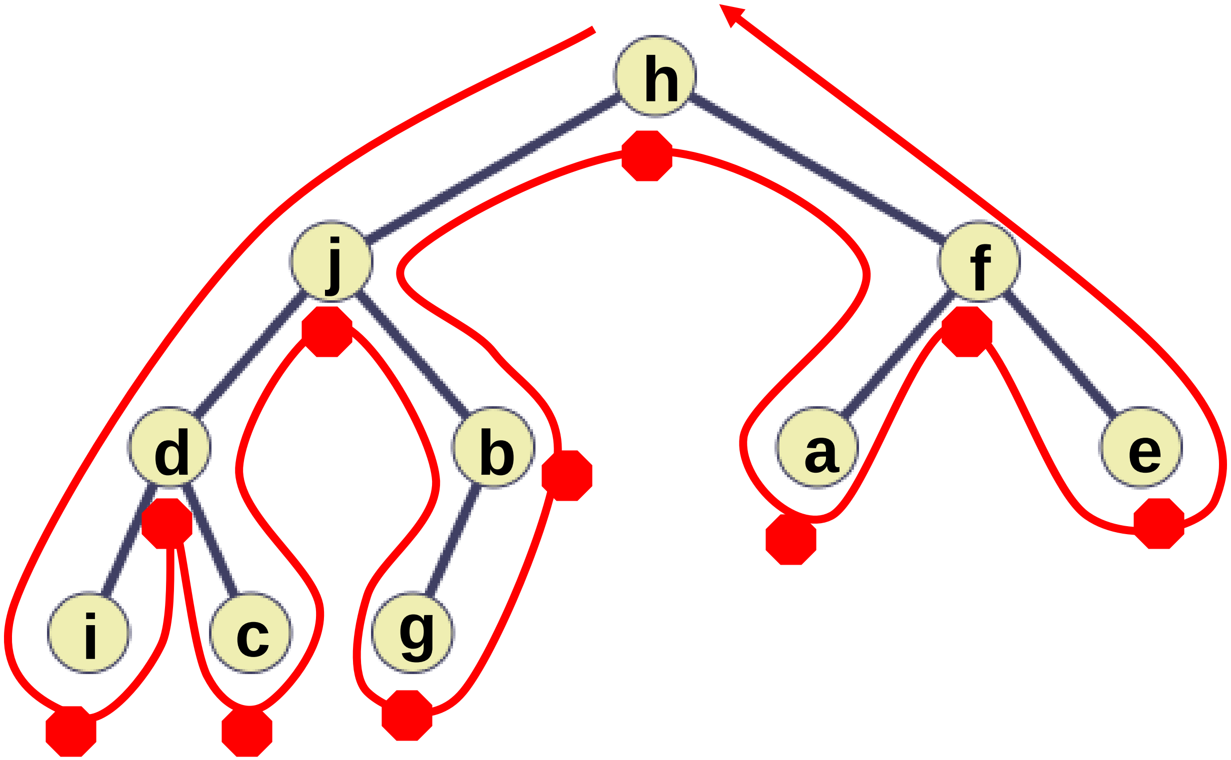
Pre-order Traversal



h j d i c b g f a e

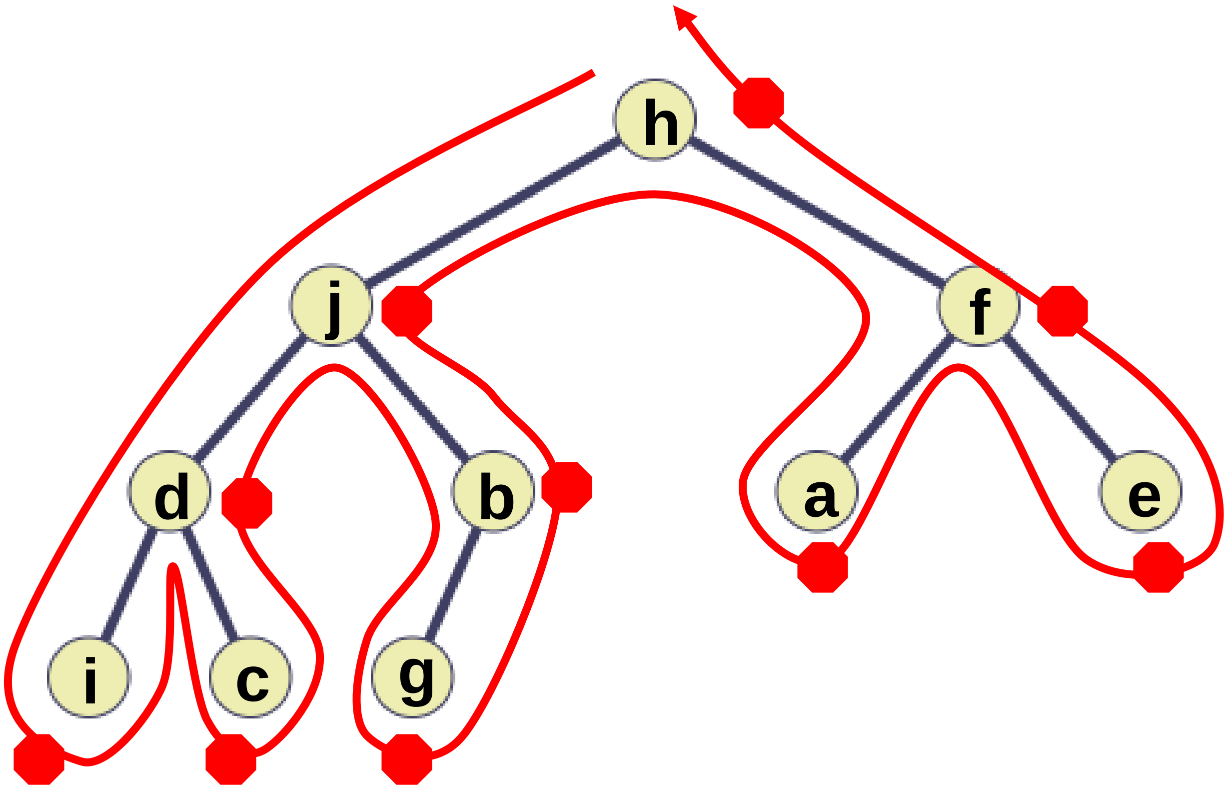
 : Node is visited here

In-order Traversal



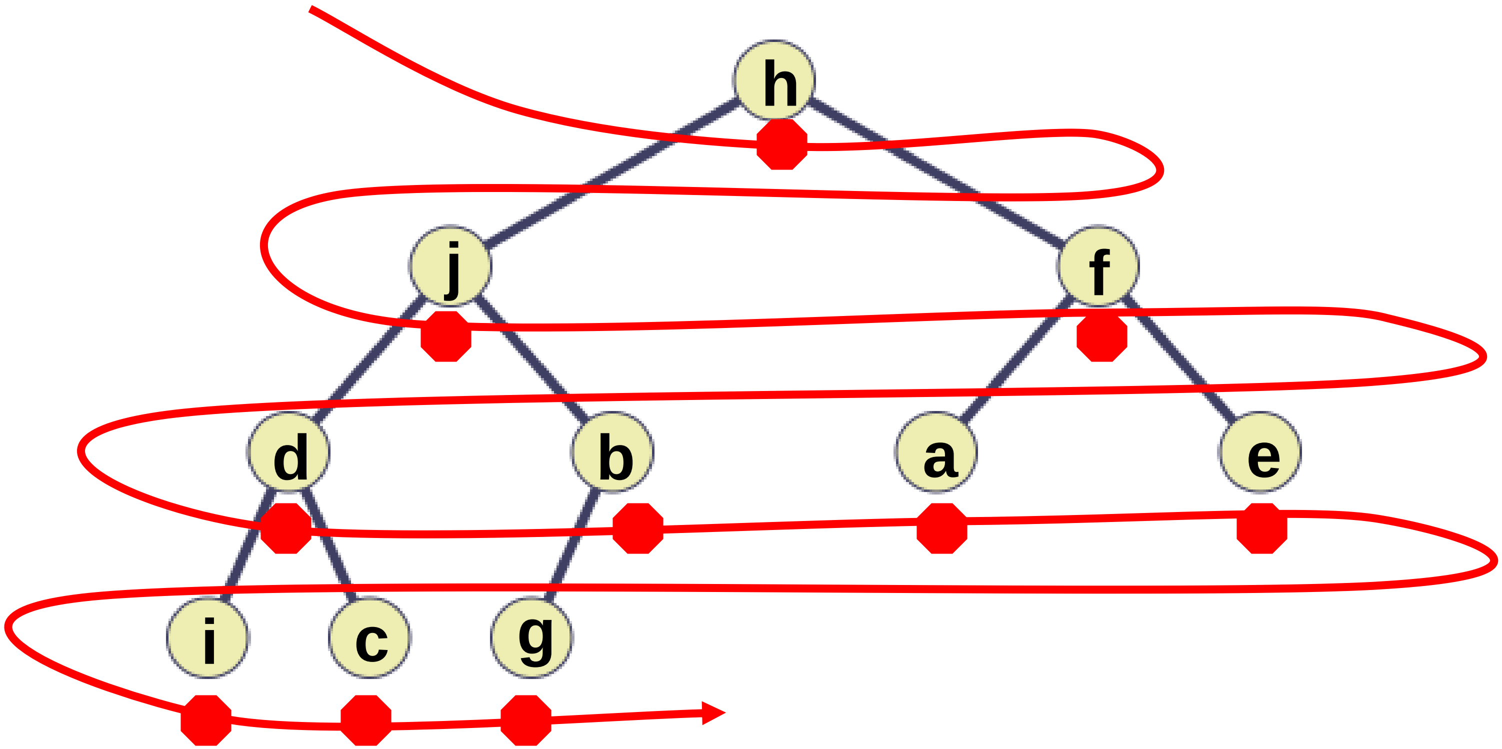
i d c j g b h a f e

Post-order Traversal



i c d g b j a e f h

Breadth-first Traversal



h j f d b a e i c g

Tree Traversal - Preorder

Prints the nodes' data in Preorder

```
void traverse( LINK h )
{
  if (h)
  {
     printf("%d", h->data);   //(prints the node)
     traverse(h->left);
     traverse(h->right);
  }
}
```

Tree Traversal - Inorder

Prints the nodes' data in Inorder

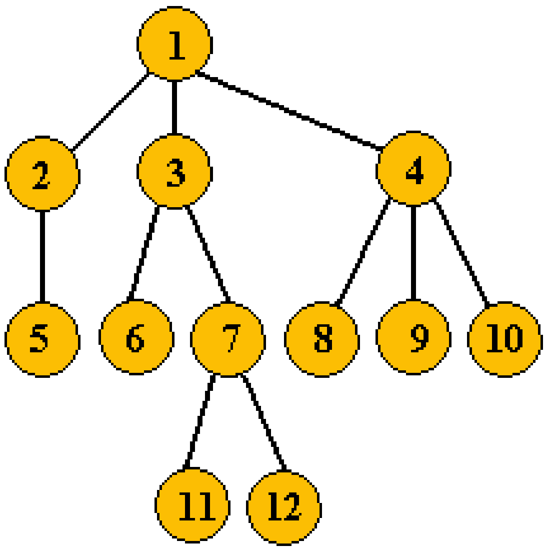
```
void traverse( LINK h )
{
  if (h)
  {
     traverse(h->left);
     printf("%d", h->data);   //(prints the node)
     traverse(h->right);
  }
}
```

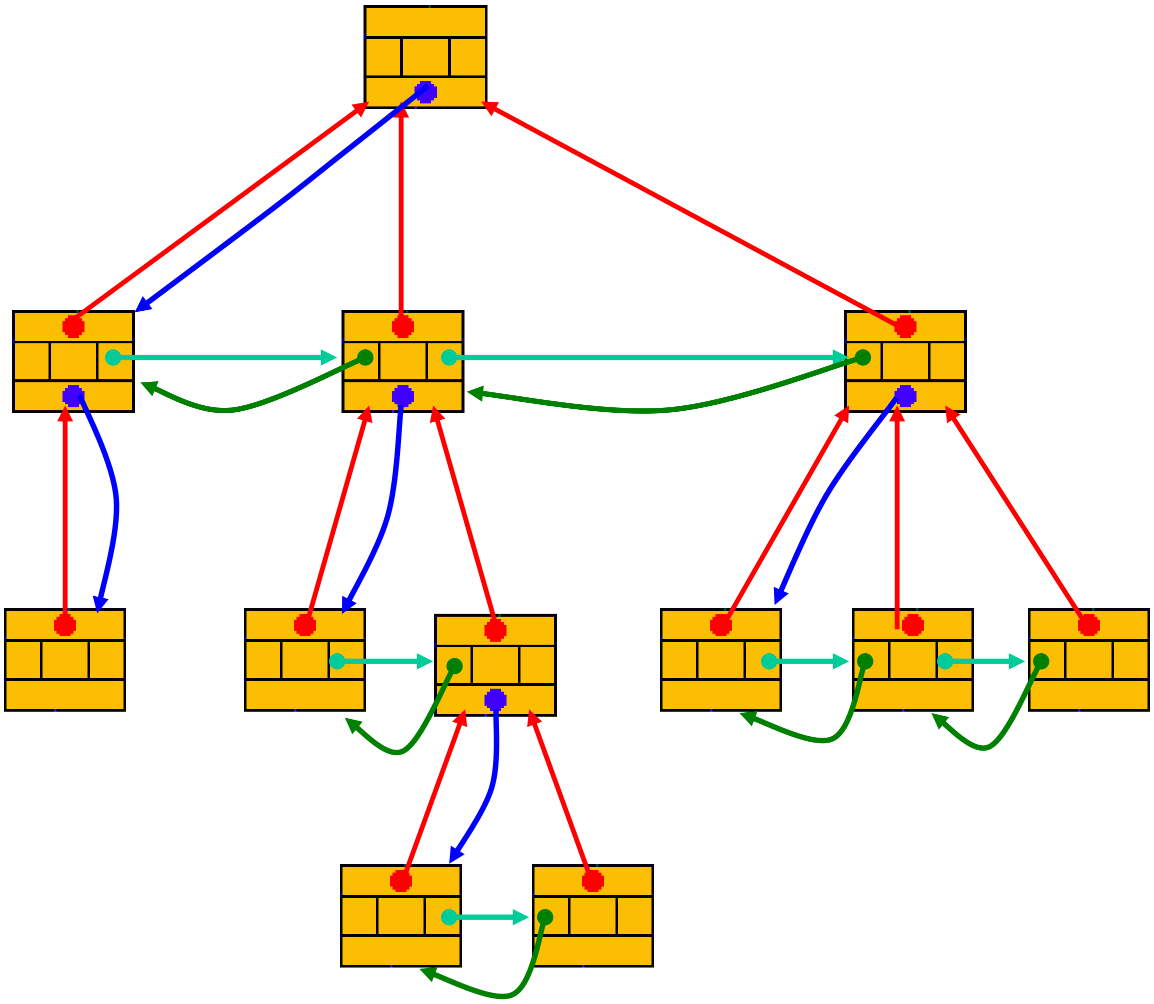
Tree Traversal - Postorder

Prints the nodes' data in Postorder

```
void traverse( LINK h )
{
  if (h)
  {
     traverse(h->left);
     traverse(h->right);
     printf(“%d”, h->data);   //(prints the node)
  }
}
```

Implementing (general) rooted trees





Example: Variable-length codes

ASCII uses 8-bits for coding letters (fixed-length code).

To minimize the space requirements, we can use an alternate coding scheme (variable-length code):

- Let the **most frequently** used letters be represented with **shorter bit** sequences (depends on the language being coded).
- Let the least frequently used letters be represented with longer bit sequences.

Example: Variable-length codes

Requirements: For each possible coded sequence, the sequence must be

- **uniquely decodeable**.
- **instantaneously decodeable** (without the need for further computations or table look-ups).

This philosophy had been employed in **Morse** code.

Also known as **Huffman coding**.

Example: Variable-length codes

Let our alphabet consist of 5 symbols, A, B, C, D, E.

Symbol	Freq.(%)
A	40
B	25
C	15
D	15
E	5

Consider the code for

ABCDE.

Example: Variable-length codes

Assume the following codes were chosen:

Symbol	Code
A	1
B	00
C	01
D	11
E	011

Consider the coding for **ABCDE**.

The code will be: **1000111011**

Can you decode it?

1 . 00 . 01 . 11 ? 011

Is **011** = **011** (E) or **01.1** (CA) ?

This code is **not** uniquely decodeable.

Example: Variable-length codes

Assume the following codes were chosen:

Symbol	Code
A	0
B	01
C	011
D	0111
E	111

Consider the coding for **ABCDE**.

The code will be: **0010110111111**

Can you decode it?

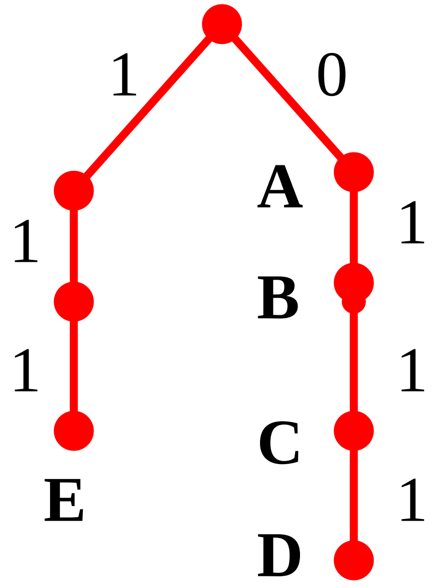
0.01 ? 0110111111
Is **011** = **01** (B) **. 1** or **011** (C) ?

This code is **not** instantaneously decodeable. You have check the next digit.

Example: Variable-length codes

Draw the code tree. Start from the root and follow the edges until a code word is found.

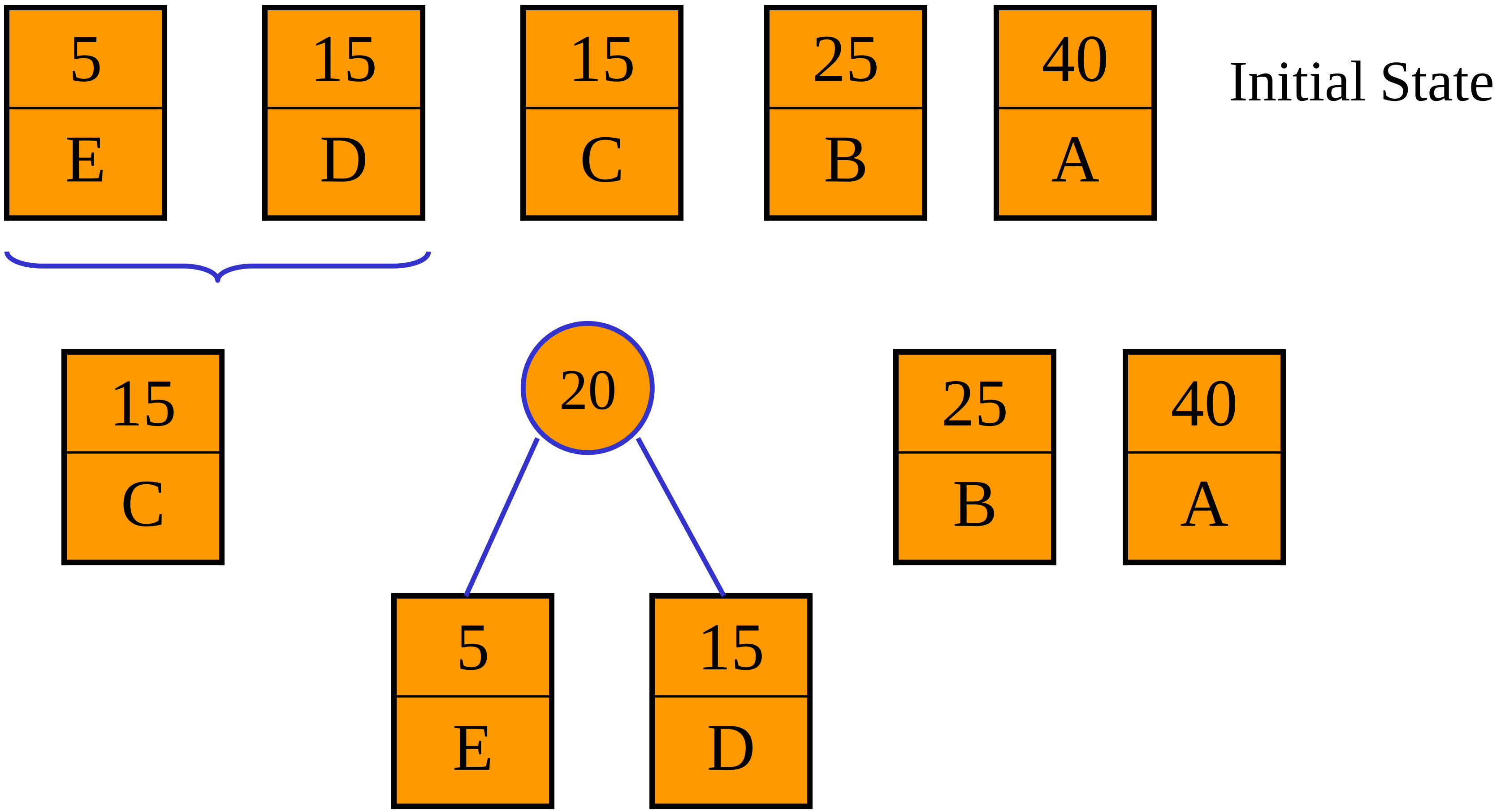
Repeat until decoding is completed.



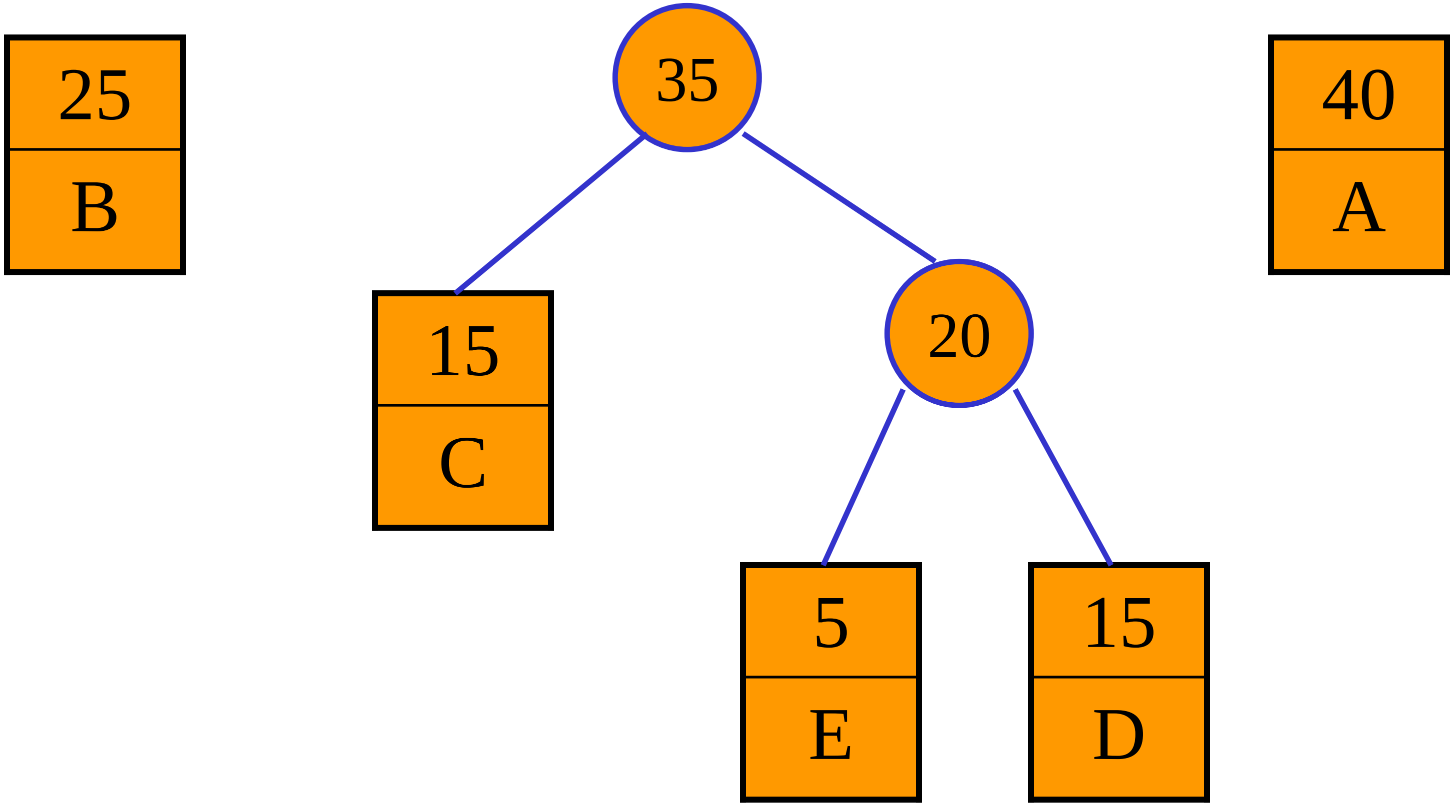
This code is **not** instantaneously decodeable. You have check the next digit (compare the next digit with the next edge on the tree).

However, it is uniquely decodeable.

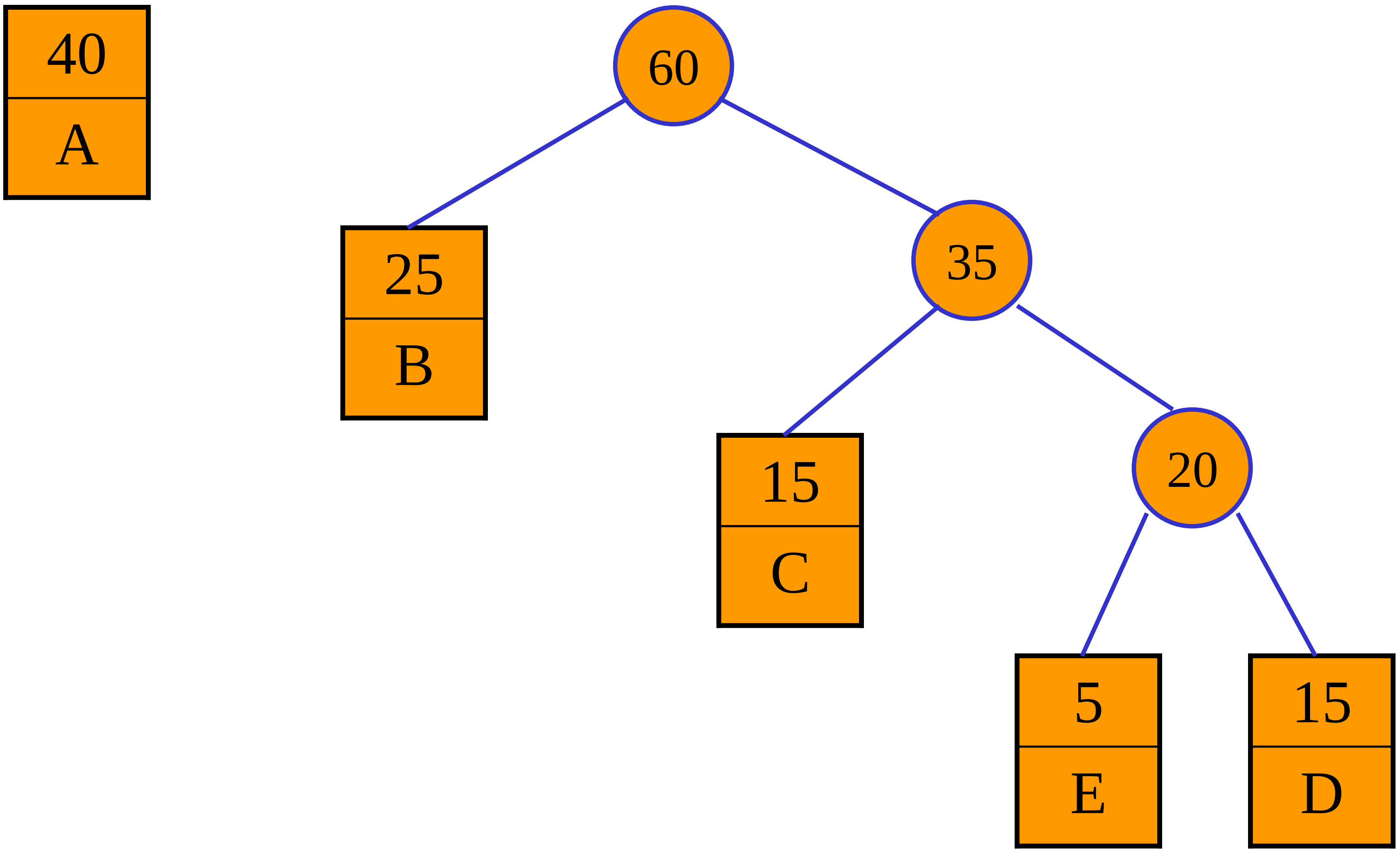
Huffman Coding



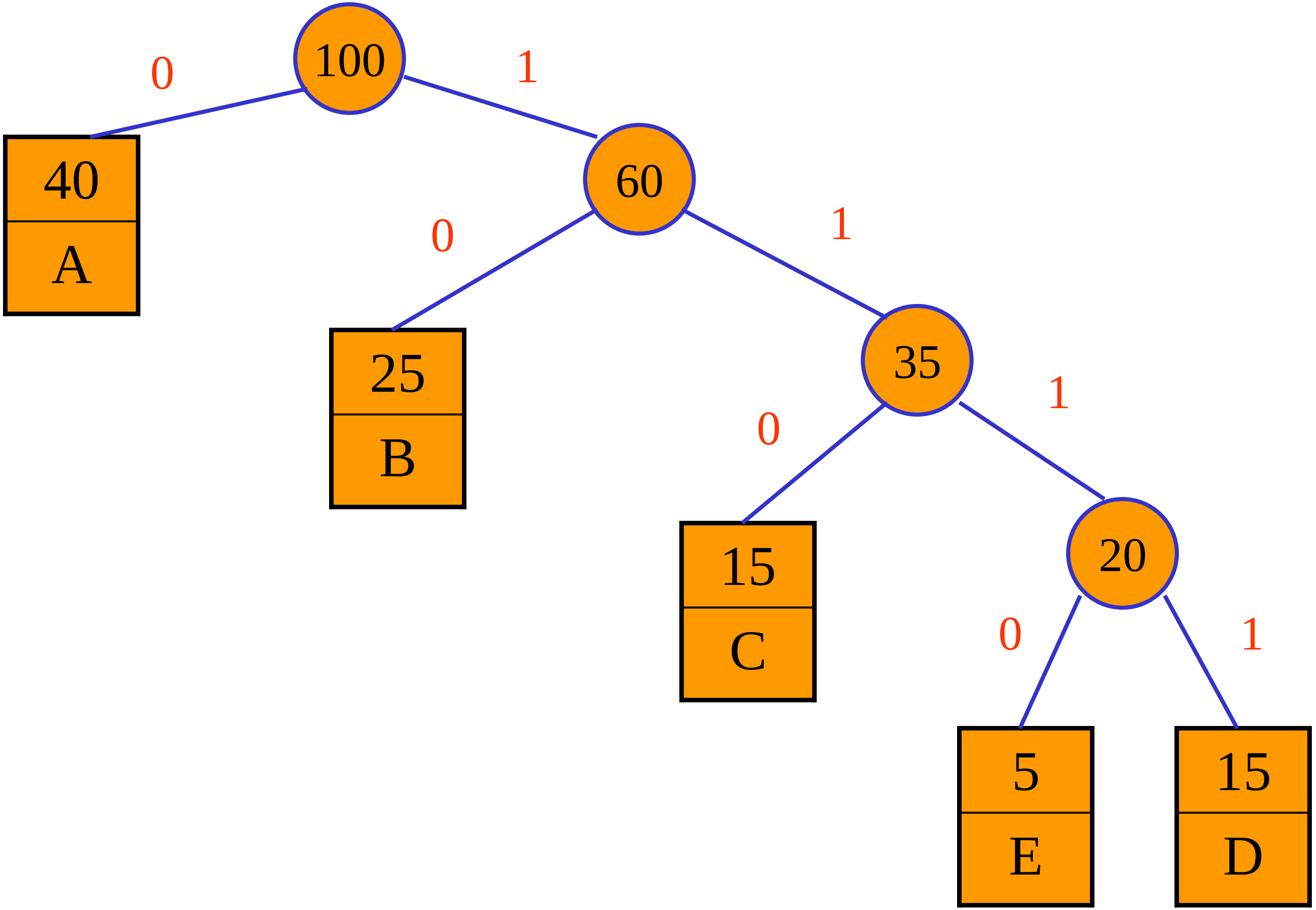
Huffman Coding



Huffman Coding



Huffman Coding



Huffman Coding

Symbol	Code
A	0
B	10
C	110
D	1111
E	1110

Consider the coding for **ABCDE**.

The code will be: **01011011111110**

Can you decode it?

0 . 10 . 110. 1111 . 1110

Analysis: With the given frequencies, the expected number of bits per character is: = 1X0.40 + 2X0.25+ 3X0.15 + 4X0.15 + 4 X 0.05

= 2.25