TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

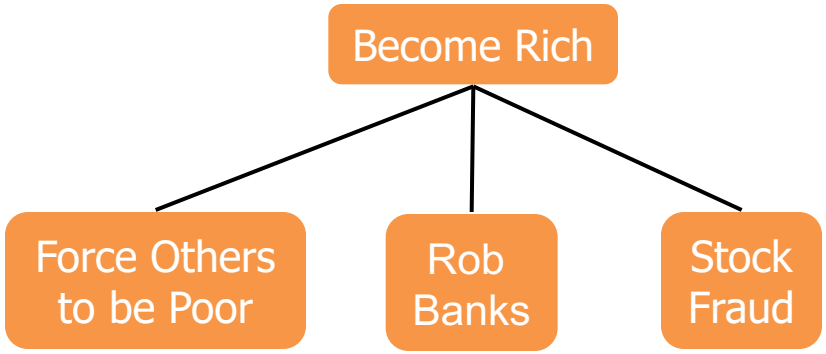
Data structure and Algorithm

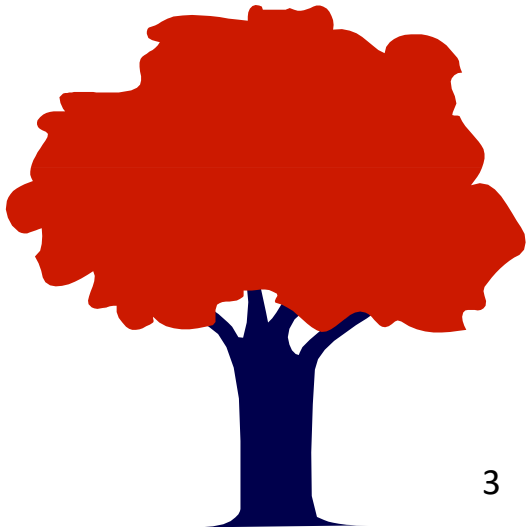
Nguyễn Khánh Phương

Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

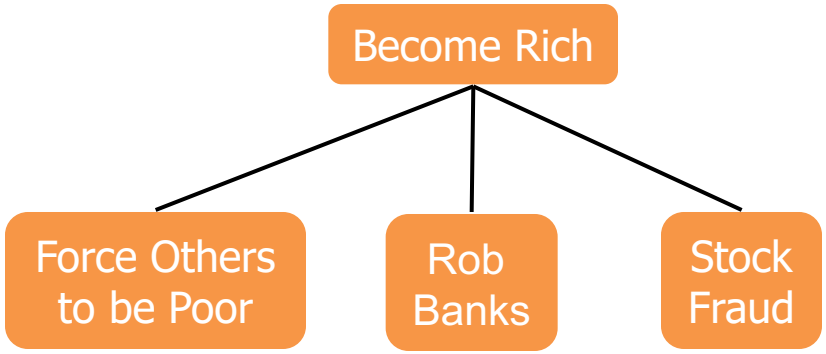
Chapter 3. Tree

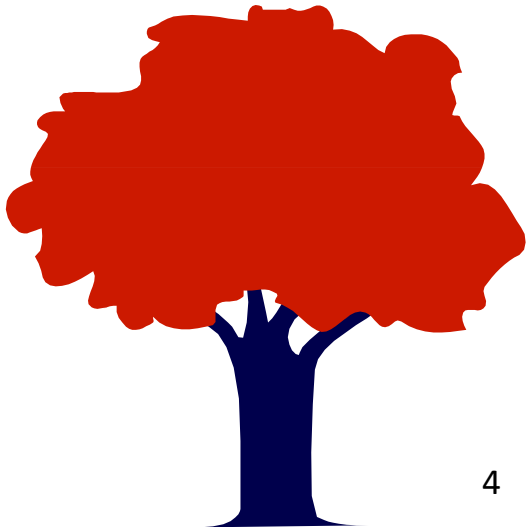
Contents





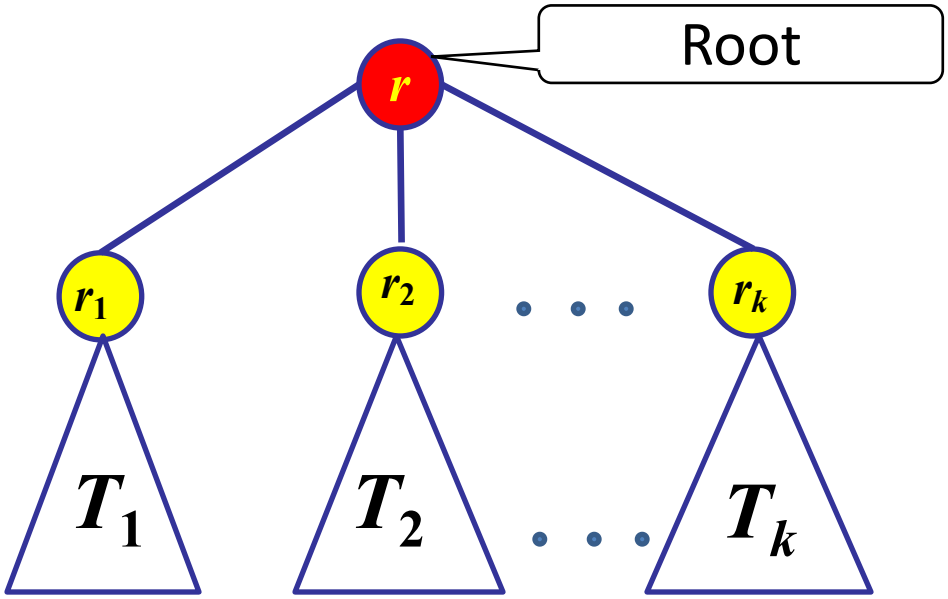
Contents





Tree: definition

- A *tree* is a finite set of one or more nodes such that
  - There is a specially designated node called *root*.
  - The remaining nodes are partitioned into n>=0 disjoint set $T_1,\ldots,T_k$, where each of these sets is a tree. $T_1,\ldots,T_k$ are called the *subtrees* of the root.
- Every node in the tree is the root of some subtree



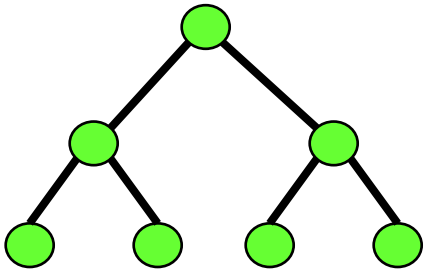
3.1. Definitions

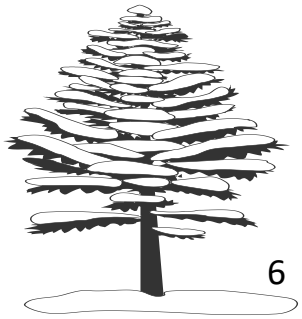
3.1.1. Tree definition

3.1.2. Tree terminology

3.1.3. Ordered tree

3.1.4. Tree ADT

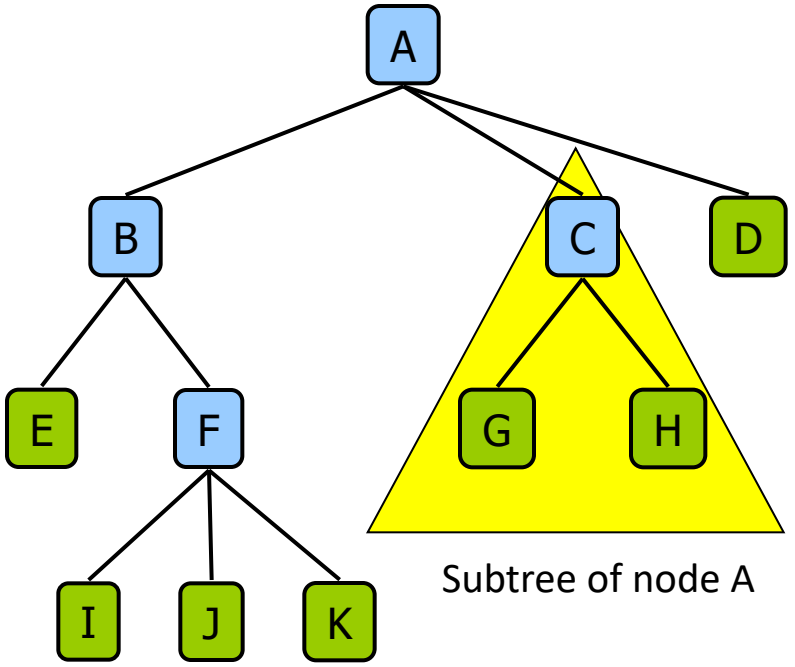




3.1.2. Tree terminology

- **Root**: node without parent (A)
- **Siblings**: nodes share the same parent (B, C, D are siblings; I, J, K are siblings; E and F are siblings; G and H are siblings)
- **Internal node**: node with at least one child (blue nodes: A, B, C, F)
- **External node** (leaf): node without children (green nodes: E, I, J, K, G, H, D)
- **Ancestors** of a node: its' parent, grandparent, grand-grandparent, etc.
- **Descendant** of a node: its' child, grandchild, grand-grandchild, etc.
- **Subtree** of a node: a tree whose root is a child of that node

Example:
Child of B: E, F
Parent of E: B
Ancestor of F: B, A
Descendant of B: E, F, I, J, K

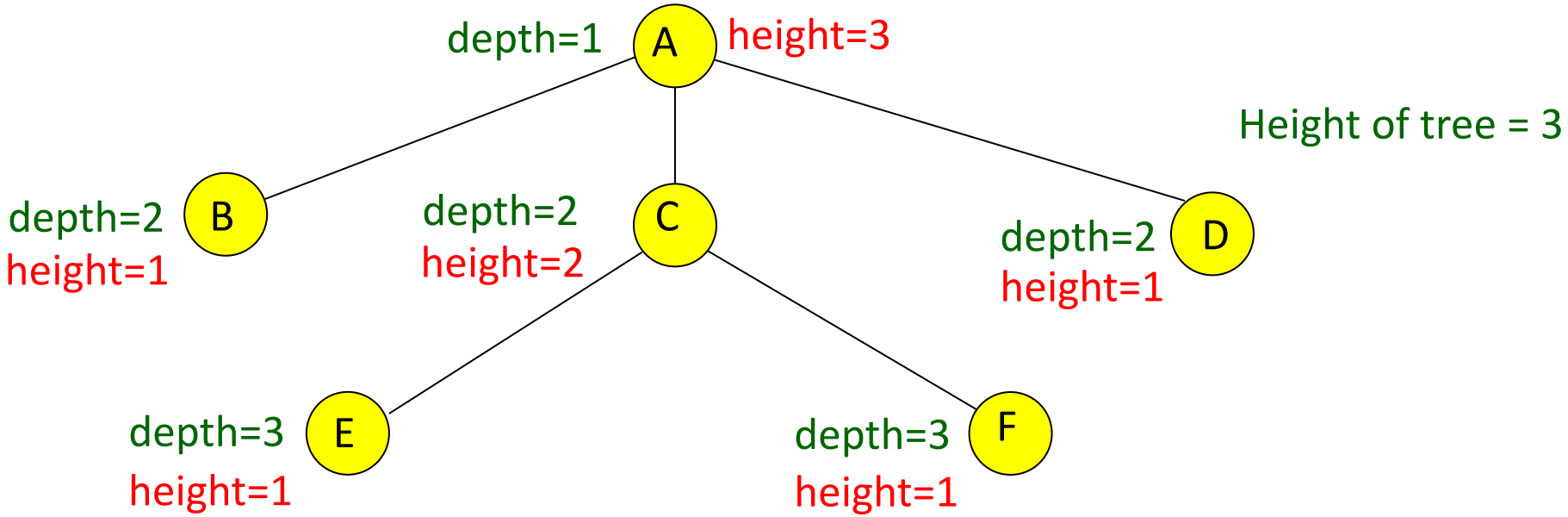


3.1.2. Tree terminology

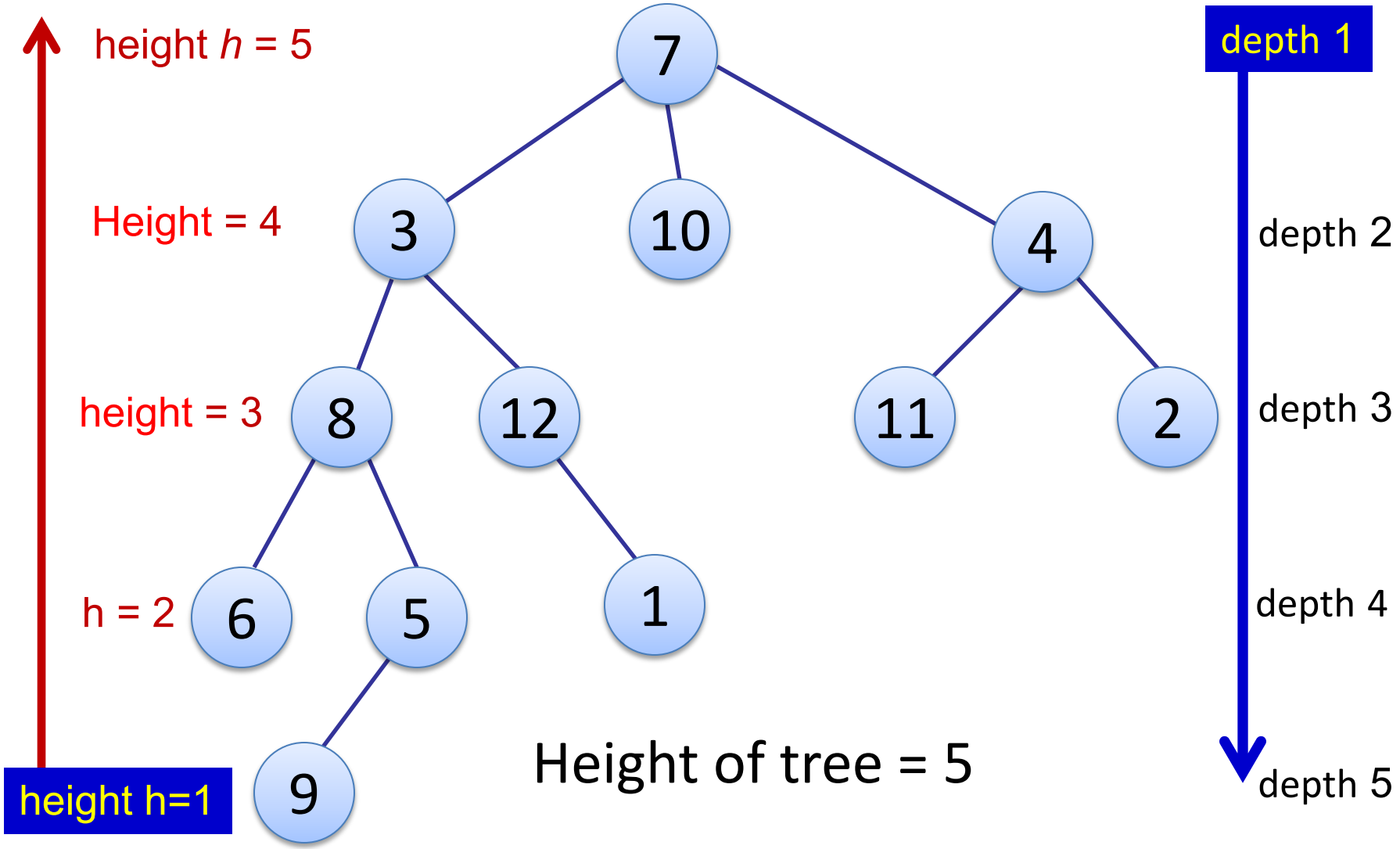
- Path: a sequence of nodes and edges connecting a node with a descendant
- Length of a path = number of edges = number of nodes - 1

(e.g.: Length of path (A ➔ C ➔E) = 2; length of path (C➔F) = 1)

- Depth/level of a node N = 1 + length of path from root to N
- Height of node N = 1 + length of longest path from N to a leaf
- Height (depth) of tree = height of root (= maximum depth of any node on the tree)

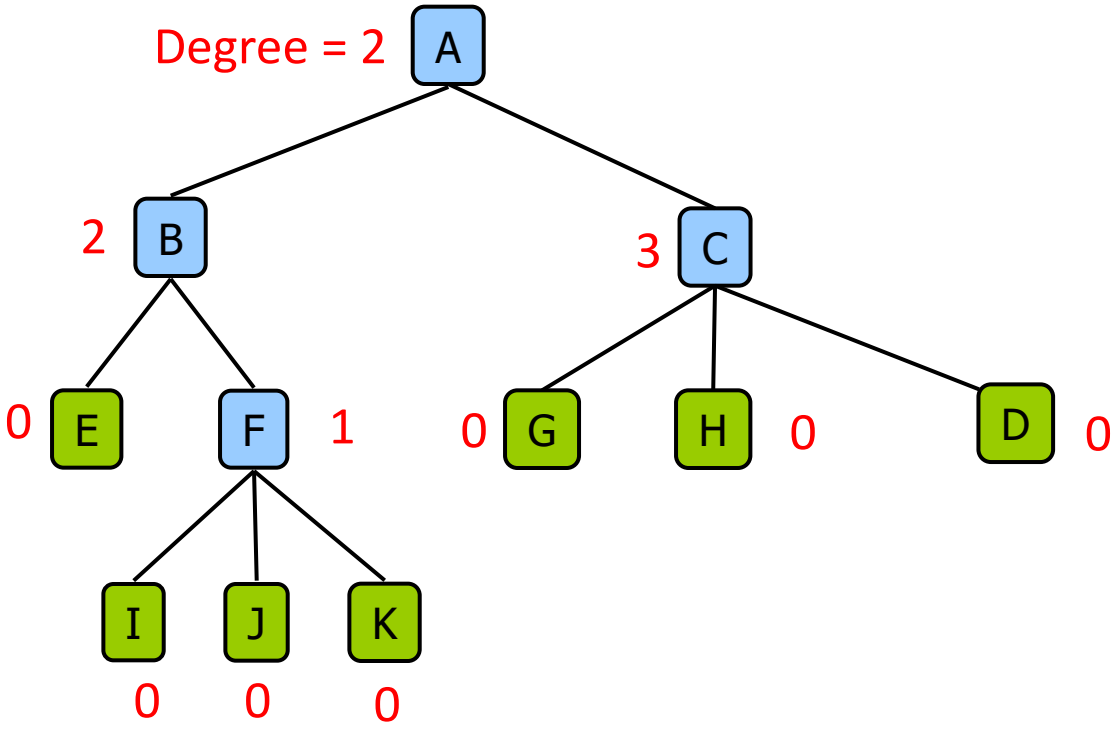


Height and depth/level



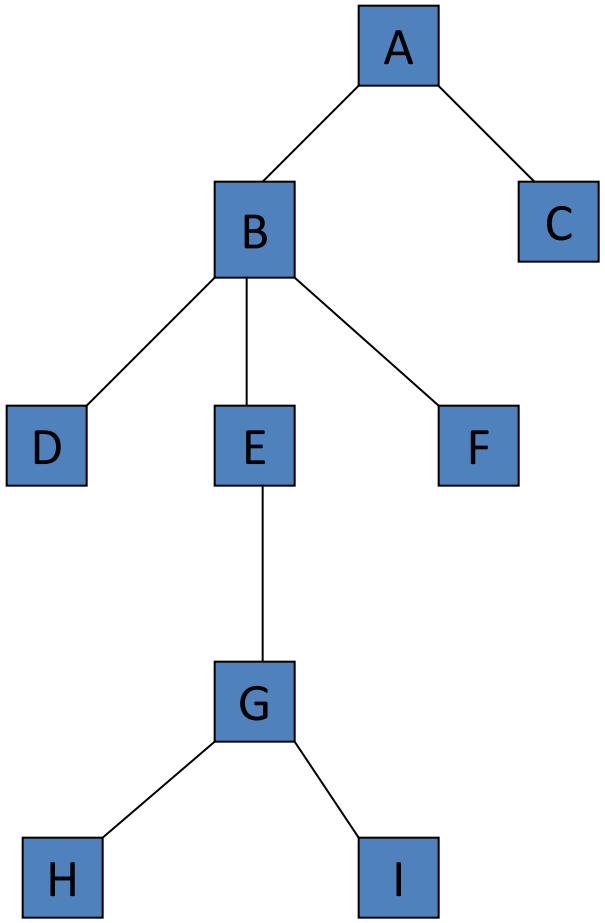
3.1.2. Tree terminology

- **Degree** of a node: the number of its children (the number of its subtree)
- **Degree** of a tree: the maximum degree of any node on the tree



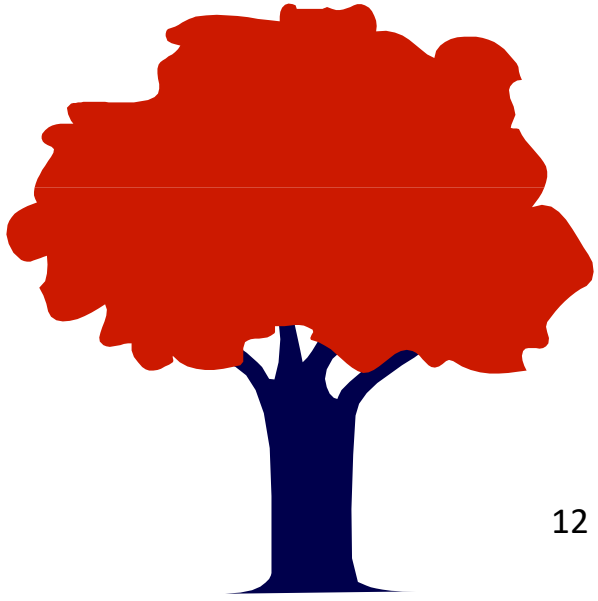
Degree of tree = ?

Example: Tree Properties



Property	Value
1) Number of nodes	
2) Root Node	
3) Leaves	
4) Internal nodes	
5) Ancestors of H	
6) Descendants of B	
7) Siblings of E	
8) Right subtree of A	
9) Left subtree of A	
10) Height of this tree	
11) Degree of this tree	

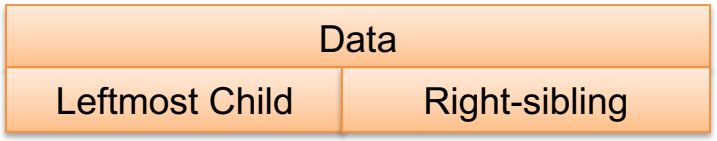
Contents



3.2. Tree representation : Pointers

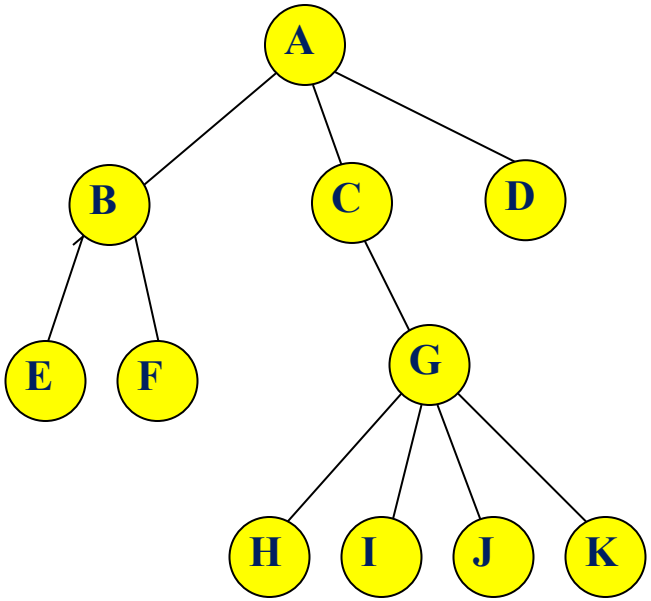
- Each node on the tree could either:
  - Have not any child, or have exactly one leftmost child
  - Have not any right-sibling, or have exactly one right-sibling

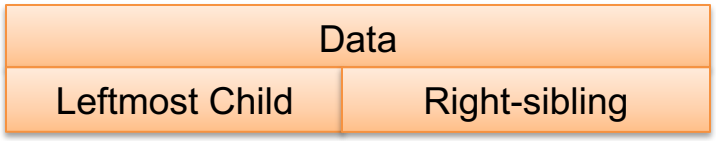
Thus, in order to represent a tree, we could store the information about the leftmost child and right-sibling of each node:



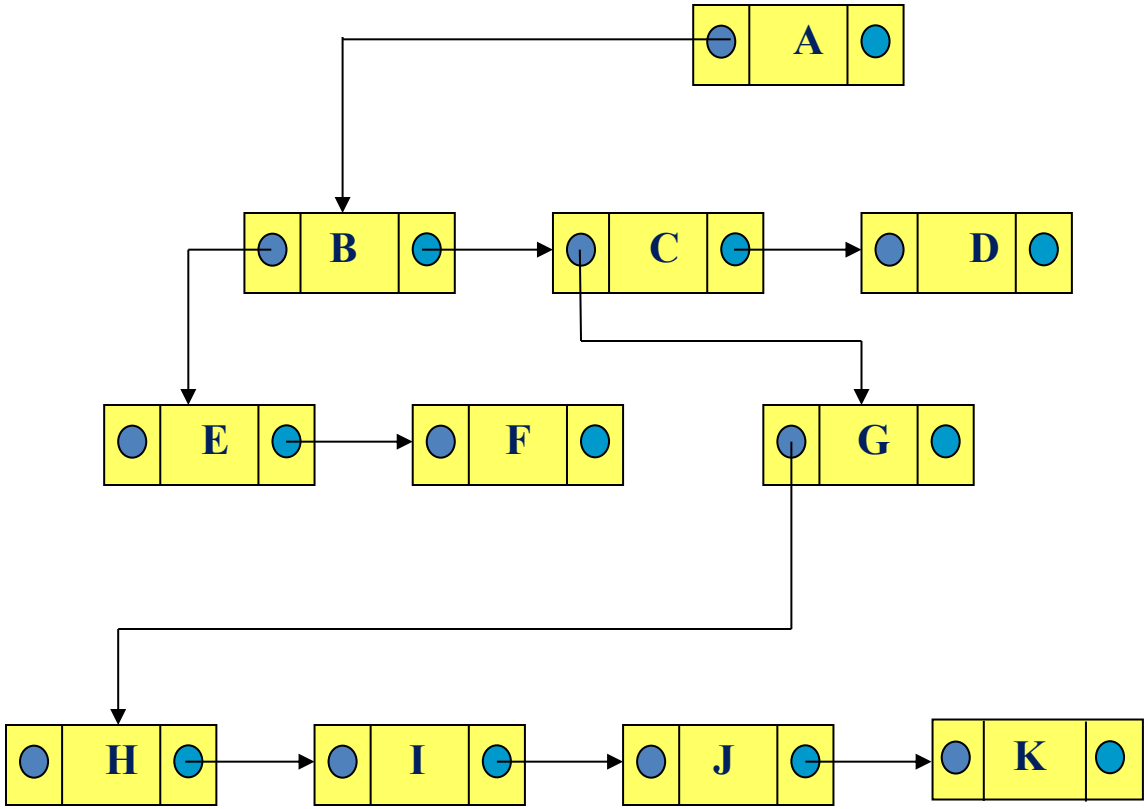

```
typedef struct
{
  int data;  // data of each node
  struct treeNode  * leftmost_child;
  struct treeNode  * right_sibling;
}treeNode;
treeNode * Root;
```

3.2. Tree representation : Leftmost-Child, Right-Sibling representation

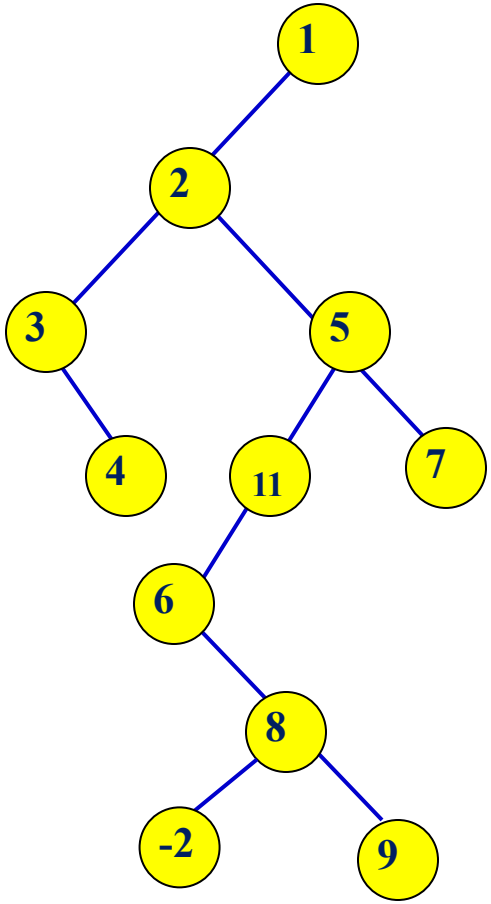


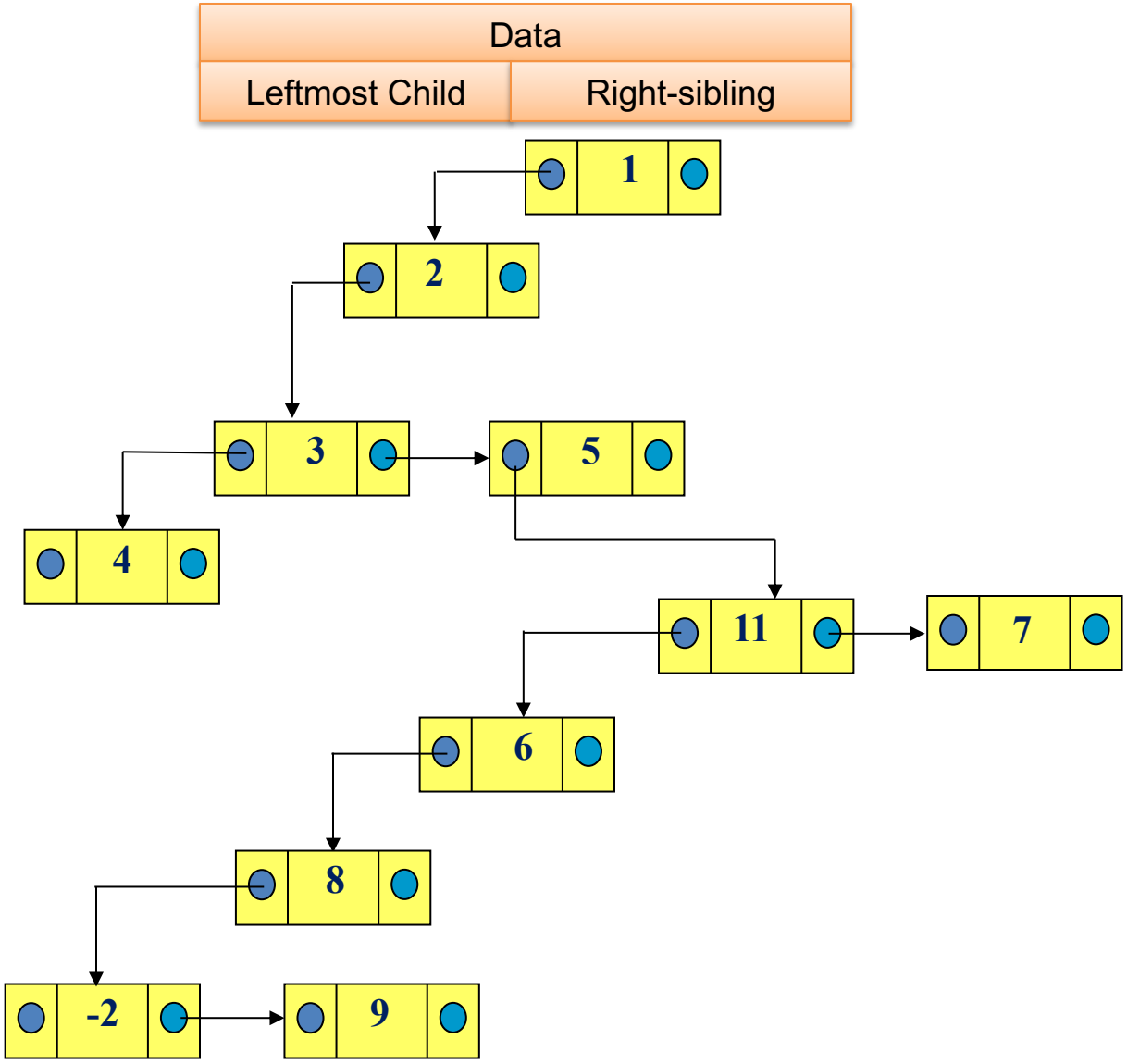



```
typedef struct
{
  char data;  // data of each node
  struct treeNode  * leftmost_child;
  struct treeNode  * right_sibling;
}treeNode;
treeNode * Root;
```



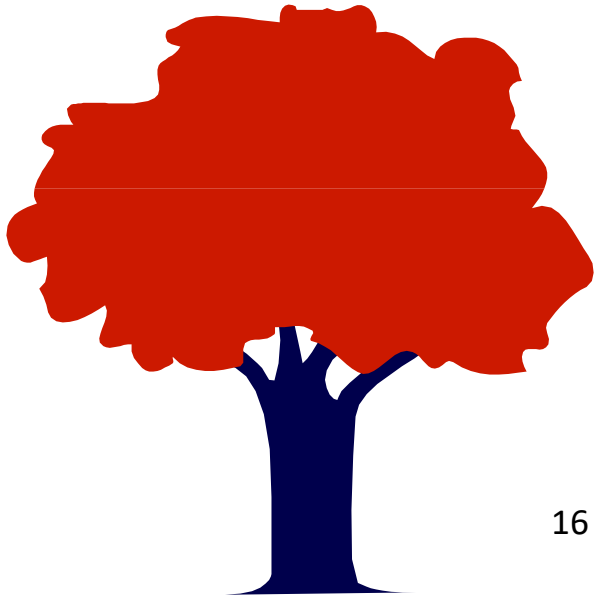
3.2. Tree representation : Leftmost-Child, Right-Sibling representation






```
typedef struct
{
  int data;  // data of each node
  struct treeNode  * leftmost_child;
  struct treeNode  * right_sibling;
}treeNode;
treeNode * Root;
```

Contents



3.3. Tree Traversal

Three main methods:

- Preorder:
  - visit the root
  - traverse in preorder the children (subtrees)

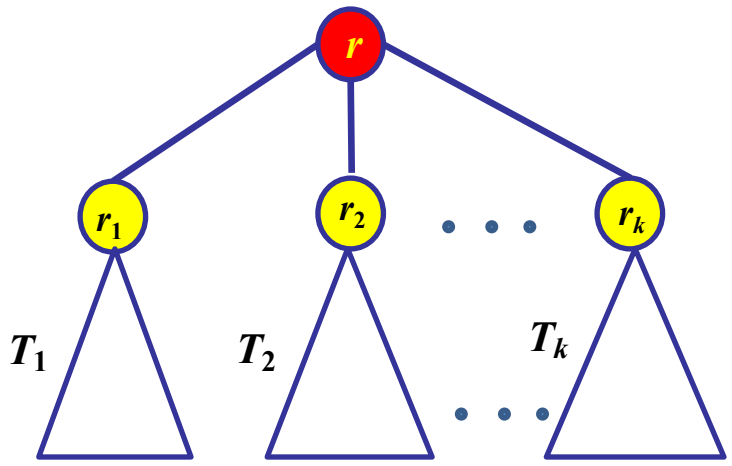
- Postorder
  - traverse in postorder the children (subtrees)
  - visit the root

- Inorder
  - traverse in inorder the left-most children (the leftmost subtree)
  - visit the root
  - traverse in inorder the remaining children that not the leftmost one

Preorder Traversal

- In a preorder traversal, a node is visited before its descendants.

Example: Preorder traversal on tree T:



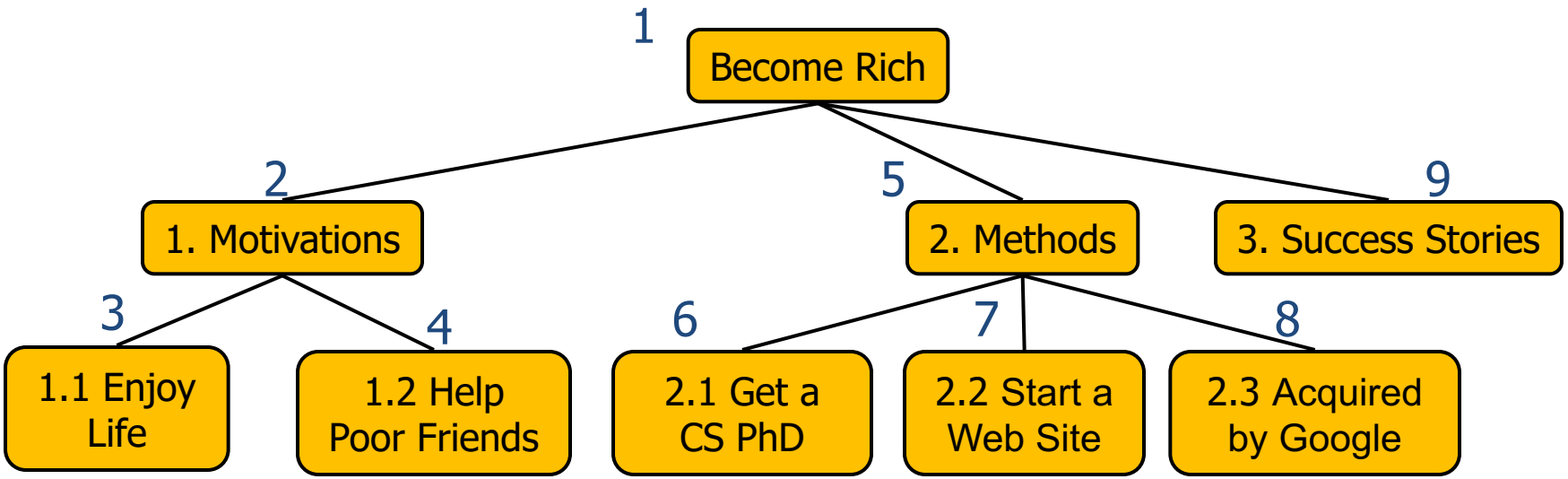

```
procedure preorder(T: ordered rooted tree)
  r := root of T
  visit r                          Step 1
  for each child c of r from left to right
  begin
     T(c) := subtree with c as its root
     preorder(T(c))                Step 2, 3…
  end
```

- Step 1: visit root ***r***,
- Step 2: visit T_1 in preorder,
- Step 3: visit T_2 in preorder
- ……..
- Step k+1: visit T_k in preorder

Preorder Traversal

- Application: print a structured document

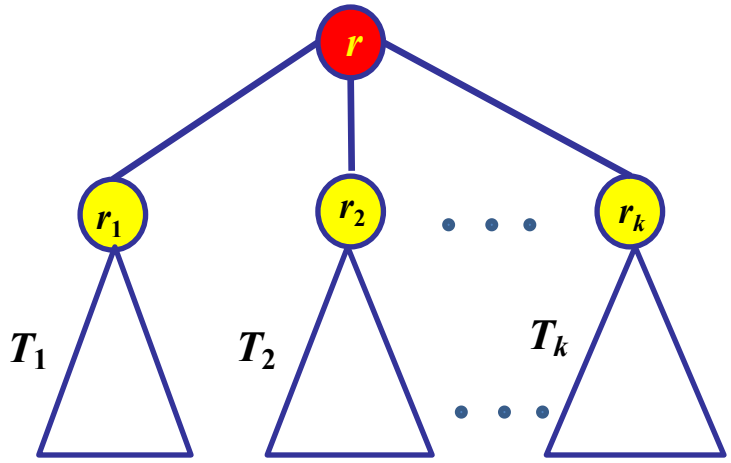
```
procedure preorder(T: ordered rooted tree)
  r := root of T
  visit r
  for each child c of r from left to right
  begin
     T(c) := subtree with c as its root
     preorder(T(c))
  end
```



Postorder Traversal

- In a postorder traversal, a node is visited after its descendants.

Example: Postorder traversal on tree T:



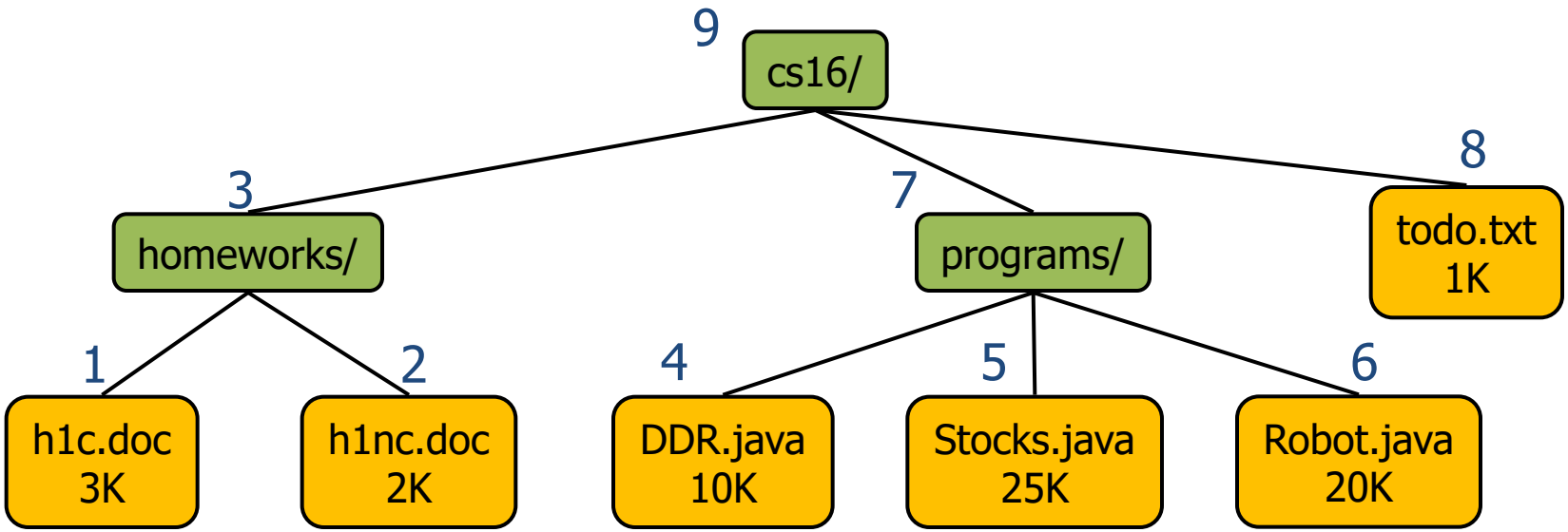

```
procedure postorder(T: ordered rooted tree)
   r := root of T
   for each child c of r from left to right
   begin
       T(c) := subtree with c as its root
       postorder(T(c))
   end
   visit r
```

- Step 1: visit T_1 in postorder,
- Step 2: visit T_2 in postorder,
- ……..
- Step k: visit T_k in postorder,
- Step k+1: visit root r

Postorder Traversal

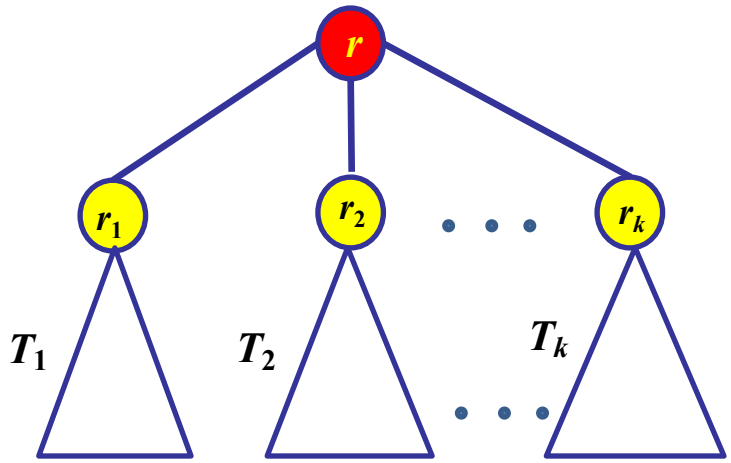
- Application: compute space used by files in a directory and its subdirectories

```
procedure postorder(T: ordered rooted tree)
   r := root of T
   for each child c of r from left to right
   begin
        T(c) := subtree with c as its root
        postorder(T(c))
   end
   visit r
```



Inorder Traversal

- Inorder traversal on tree T:



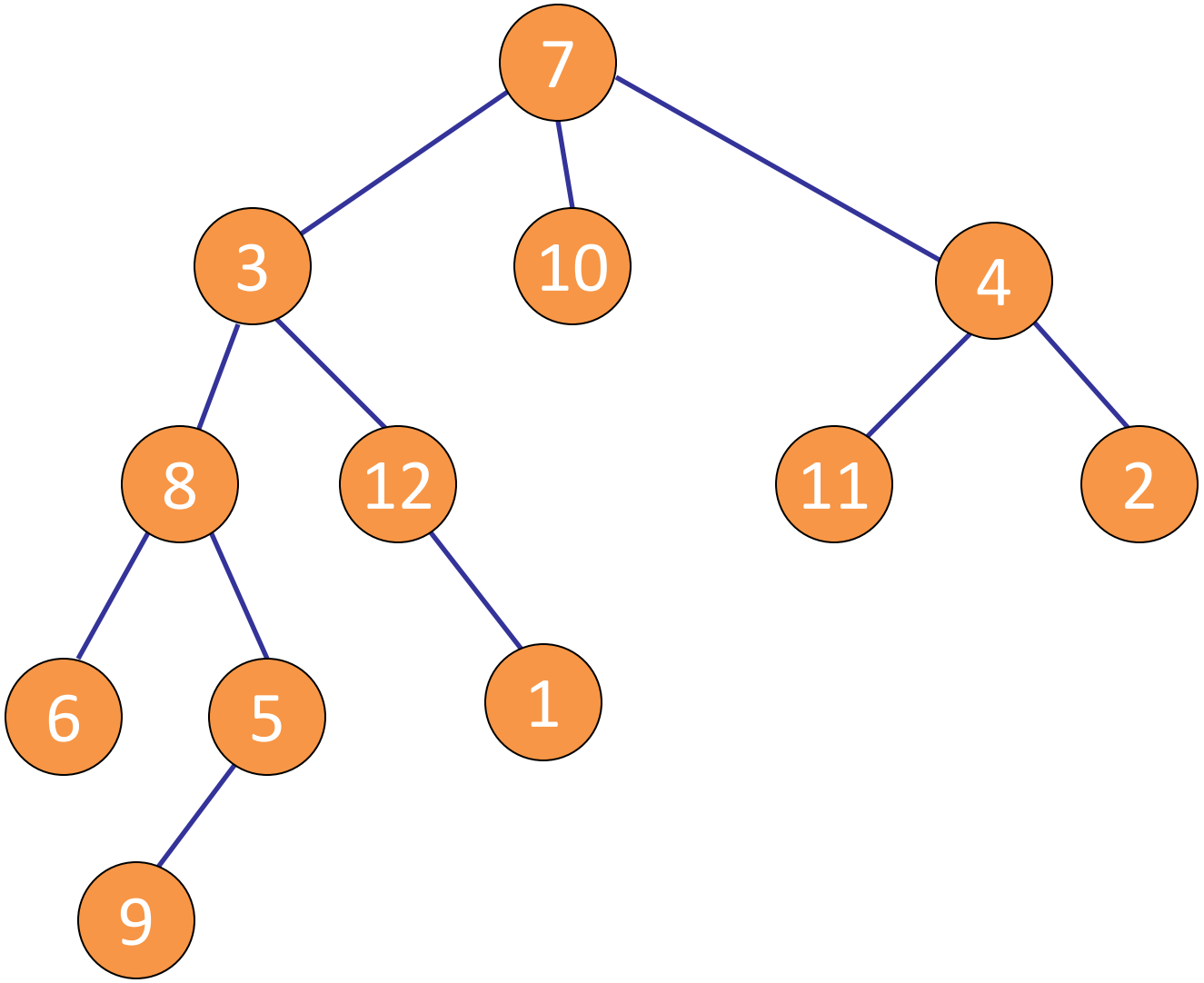
- Step 1: visit T_1 in inorder,
- Step 2: visit root ***r***,
- Step 3: visit T_2 in inorder,
- ……..
- Step k+1: visit T_k in inorder

```
procedure inorder(T: ordered rooted tree)
  r := root of T
  if r is a leaf then visit r
  else
  begin
     l := first child of r from left to right
     T(l) := subtree with l as its root
     inorder(T(l))        Step 1

     visit r              Step 2
     for each child c of r except for l left to right
        T(c) := subtree with c as its root
        inorder(T(c))     Step 3, 4..
  end
```

Example: tree traversal

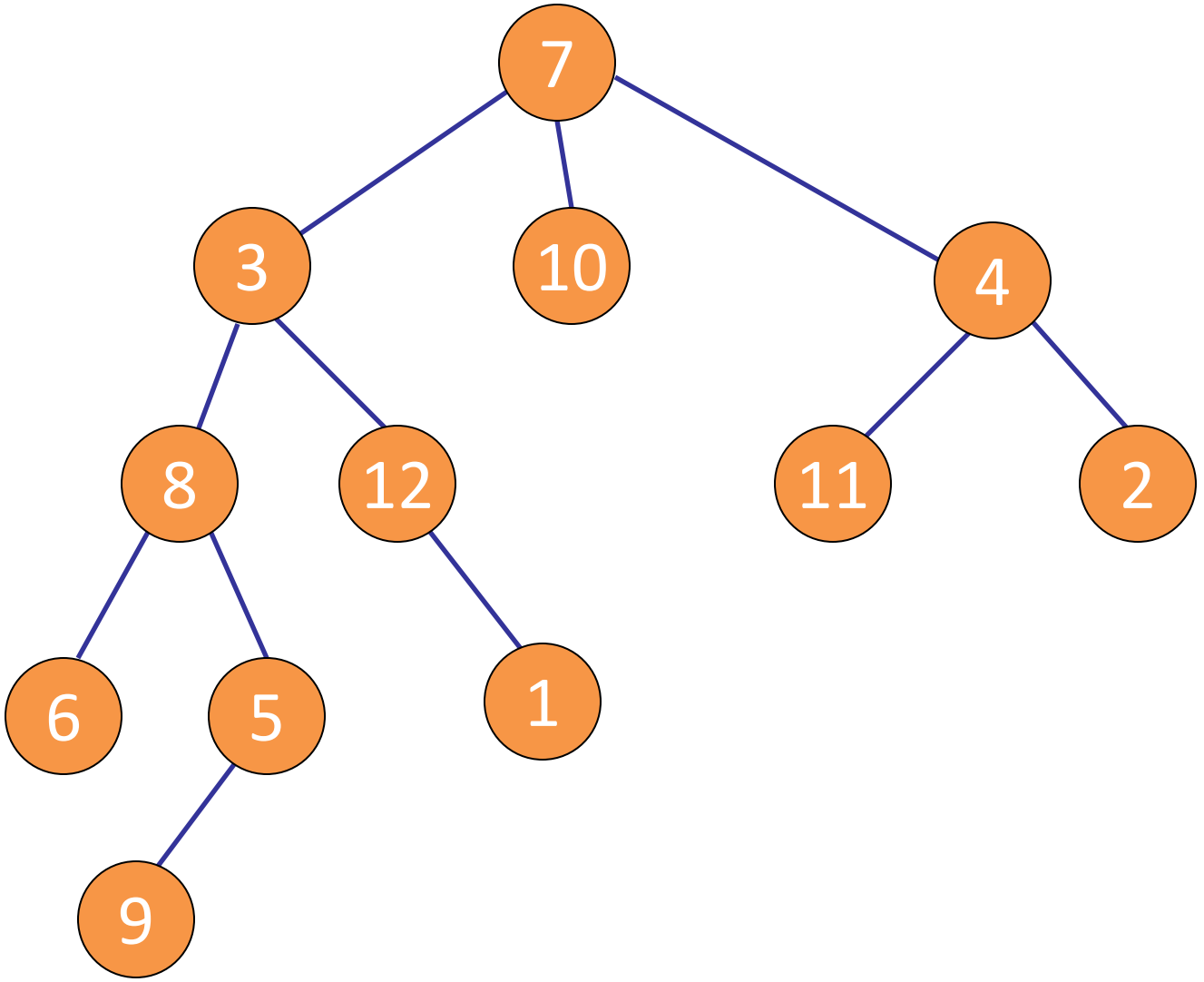
Given the ordered tree T rooted at node 7. Show the order of nodes that they are visited if we traverse the tree T in:

- Preorder
- Inorder
- Postorder



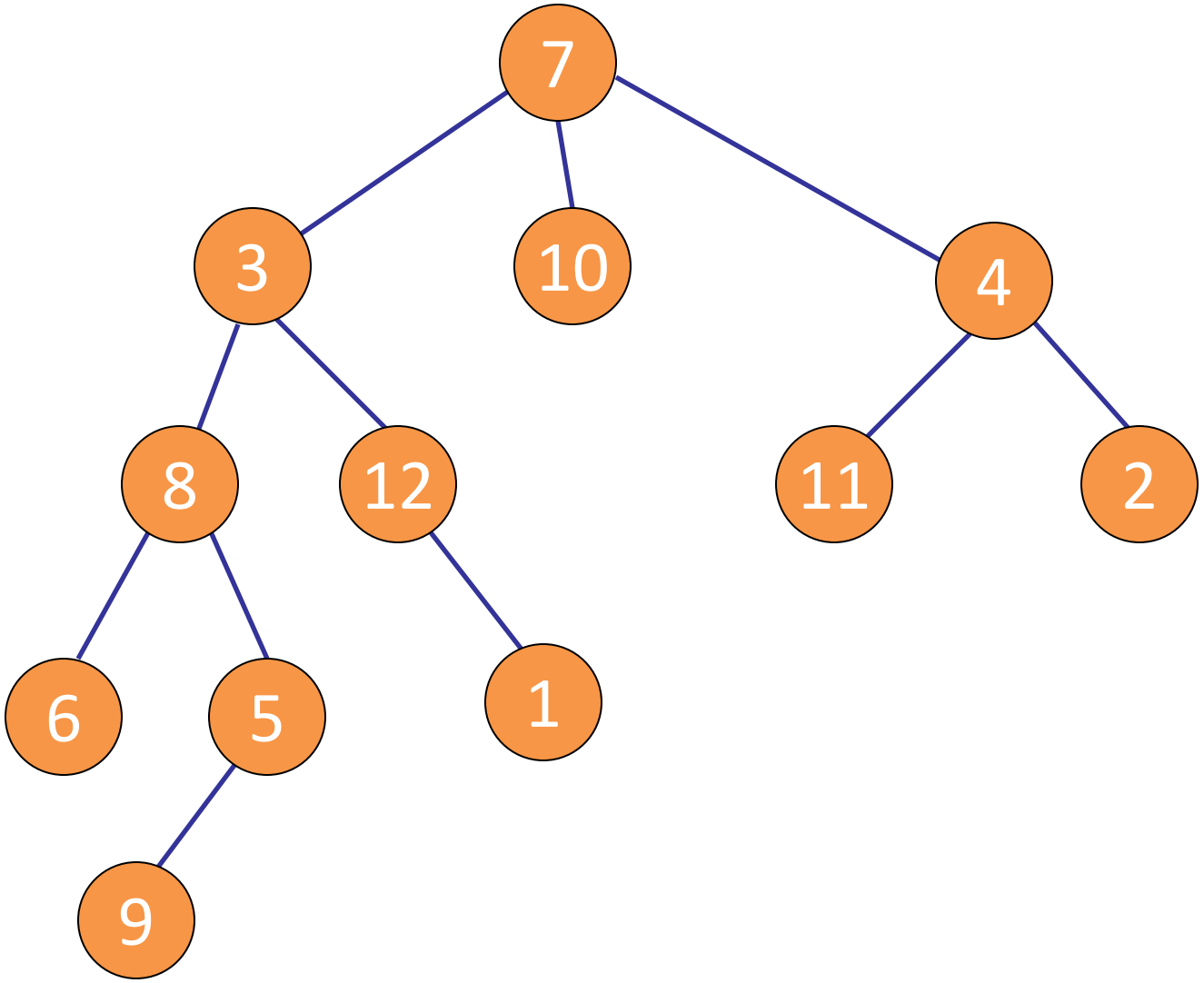
Preorder: Root, left child, right child

- 7,
- 3,
- 8,
- 6,
- 5,
- 9,
- 12,
- 1,
- 10,
- 4,
- 11,
- 2



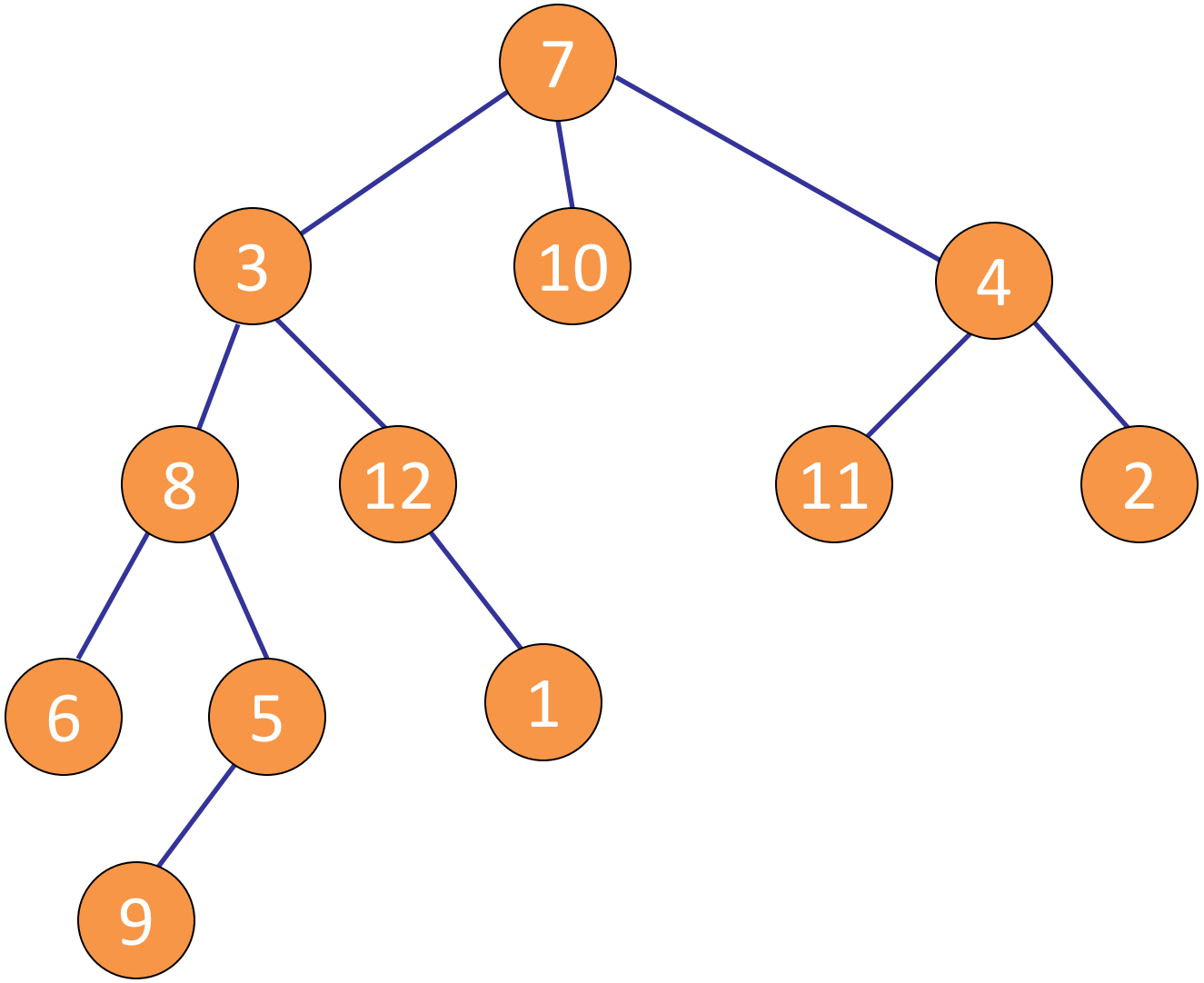
Inorder: Left child, Root, right child

- 6,
- 8,
- 9,
- 5,
- 3,
- 12,
- 1,
- 7,
- 10,
- 11,
- 4,
- 2



Postorder: Left child, right child, Root

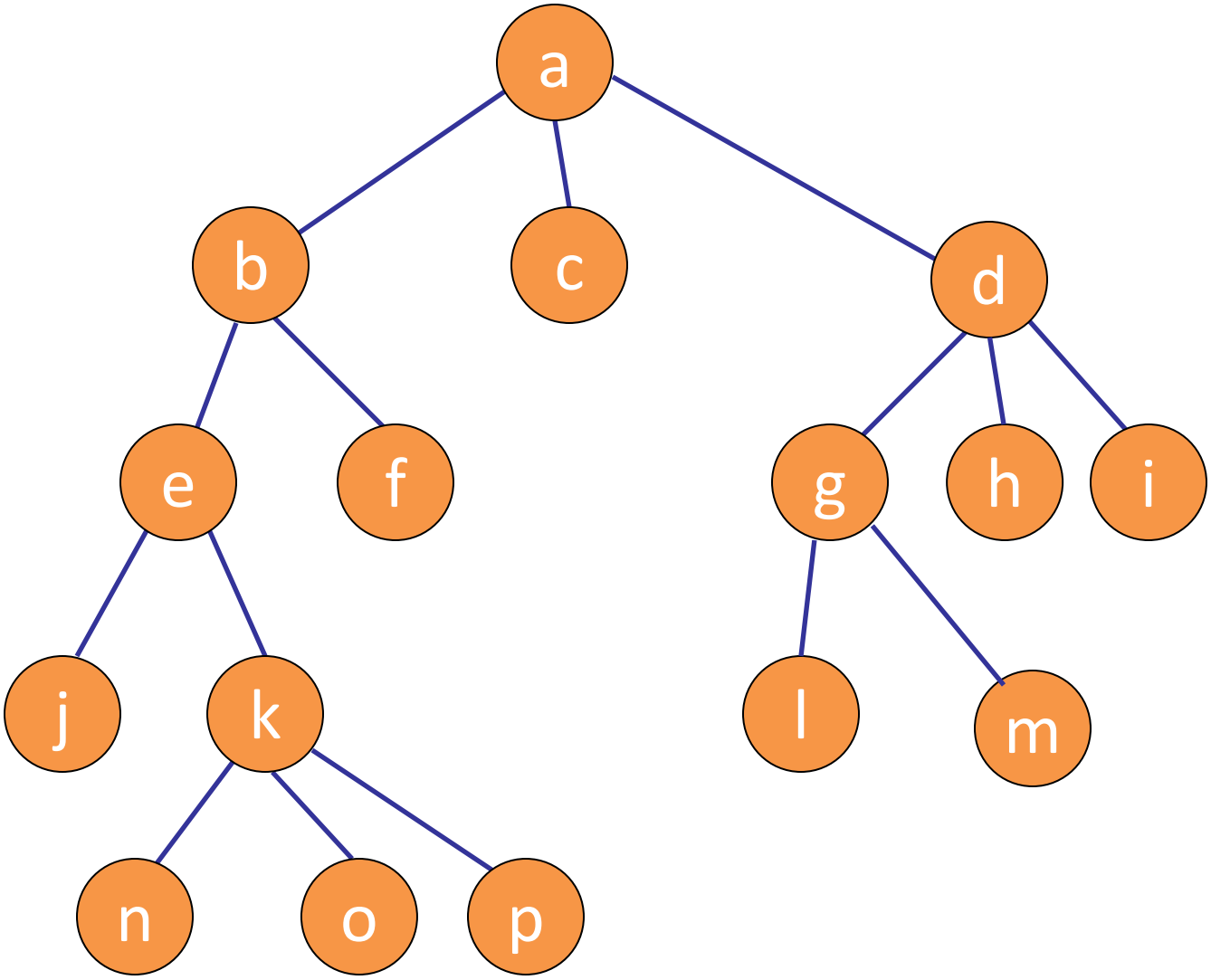
- 6,
- 9 ,
- 5,
- 8,
- 1,
- 12,
- 3,
- 10,
- 11,
- 2,
- 4,
- 7



Example: tree traversal

Given the ordered tree T rooted at node a. Show the order of nodes that they are visited if we traverse the tree T in:

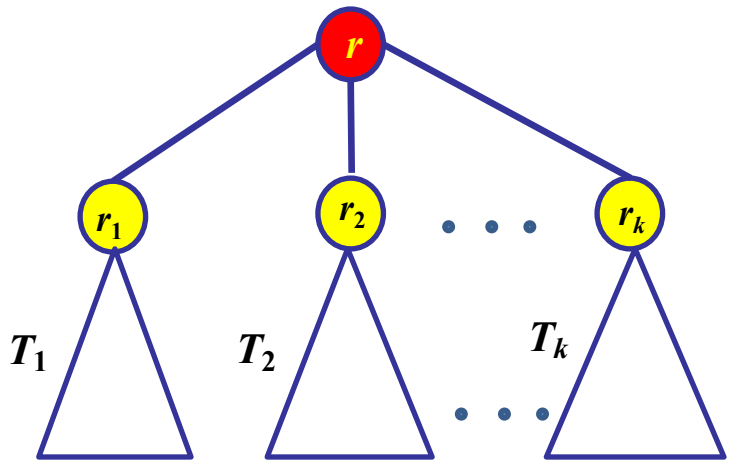
- Preorder
- Inorder
- Postorder



Preorder Traversal

- In a preorder traversal, a node is visited before its descendants.

Example: Preorder traversal on tree T:

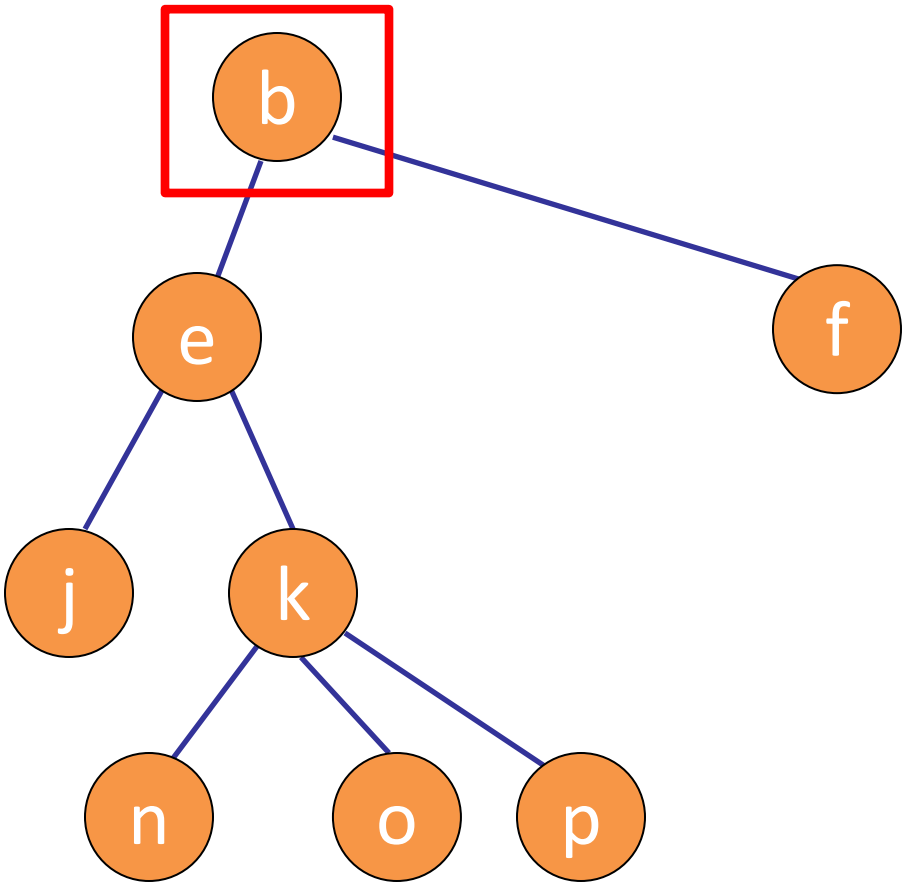



```
procedure preorder(T: ordered rooted tree)
  r := root of T
  visit r                          Step 1
  for each child c of r from left to right
  begin
     T(c) := subtree with c as its root
     preorder(T(c))                Step 2, 3…
  end
```

- Step 1: visit root ***r***,
- Step 2: visit T_1 in preorder,
- Step 3: visit T_2 in preorder
- ……..
- Step k+1: visit T_k in preorder

PreOrder

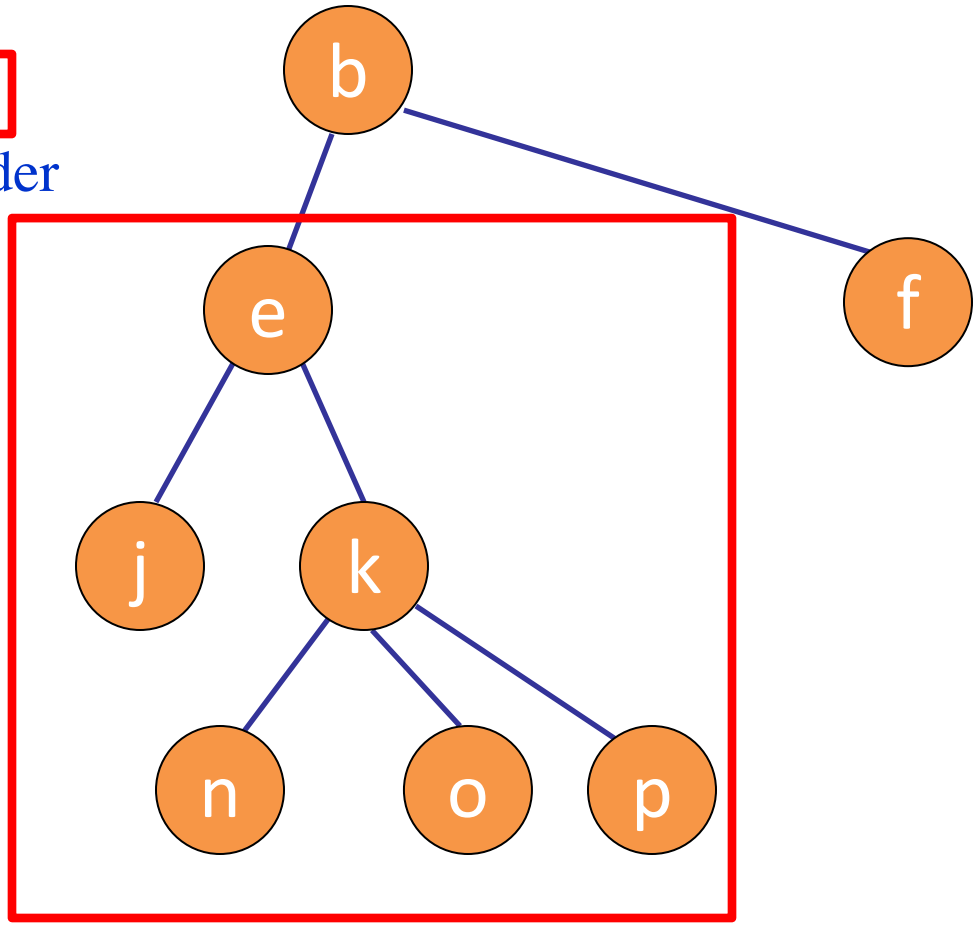
1. Visit root
2. Visit leftmost subtree in preorder
3. Visit remaining subtrees in preorder



b

PreOrder

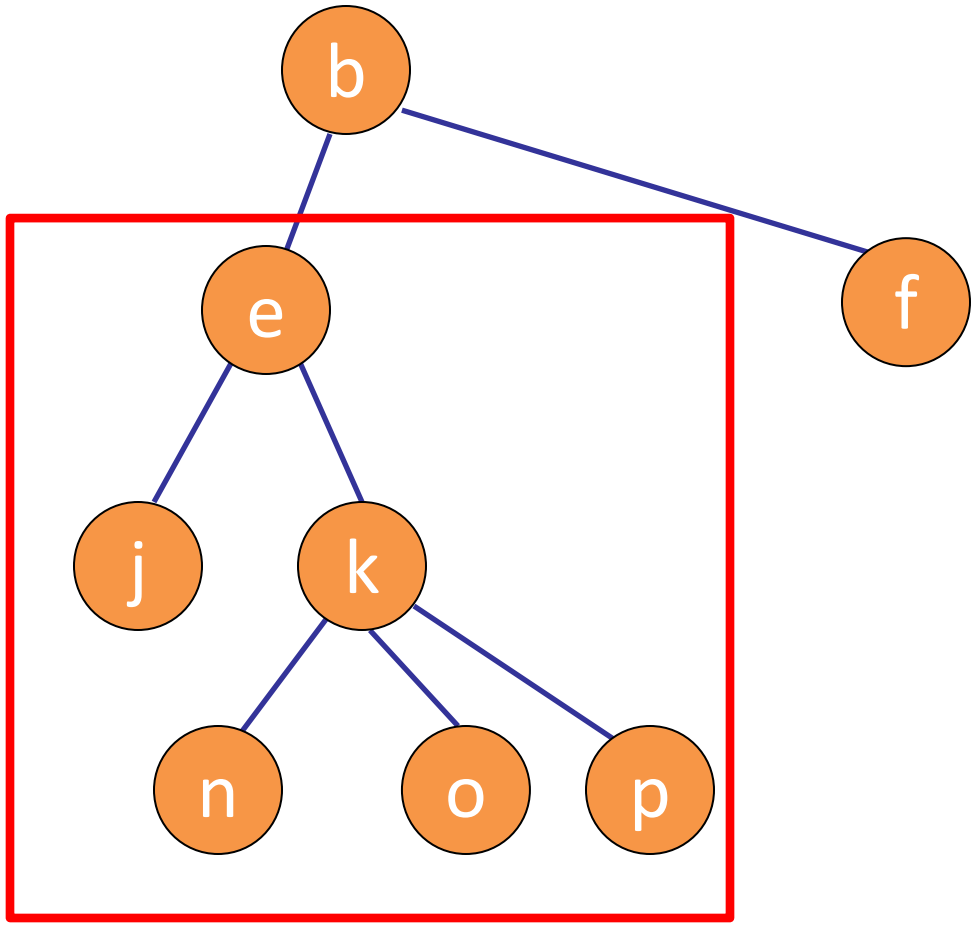
1. Visit root
2. Visit leftmost subtree in preorder
3. Visit remaining subtrees in preorder



b

PreOrder

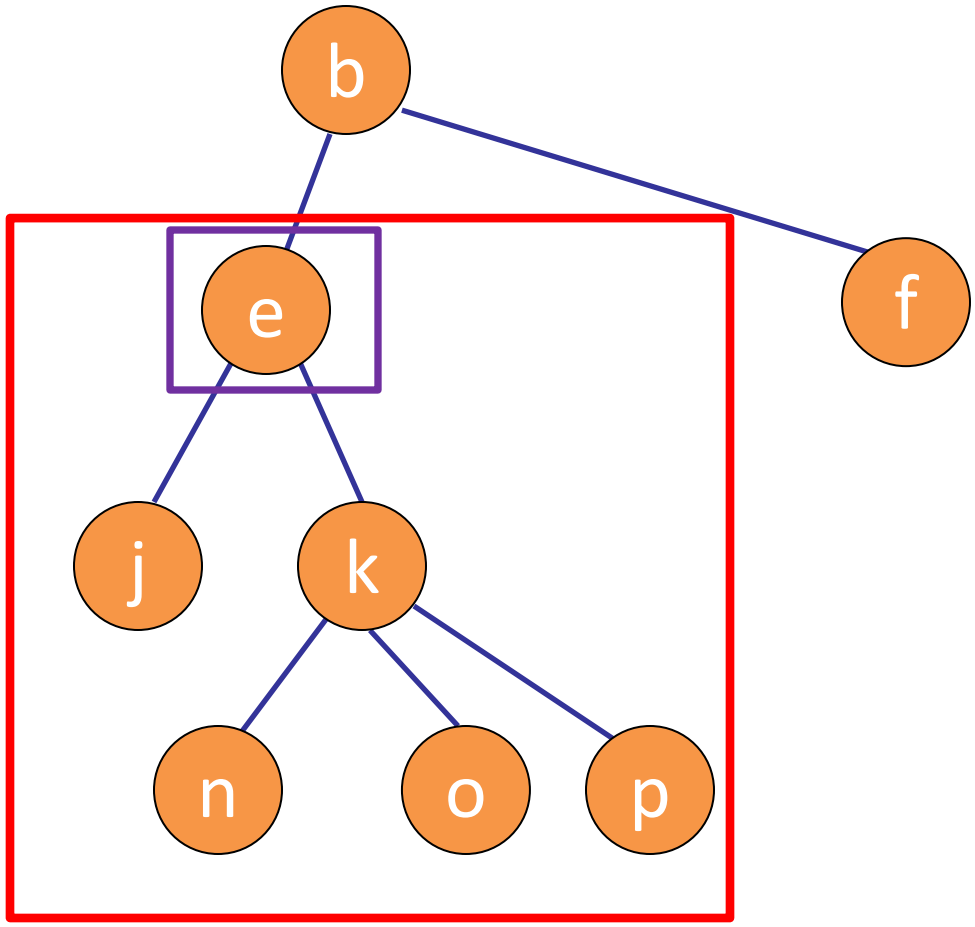
1. Visit root
2. Visit leftmost subtree in preorder
   2.1. Visit root
   2.2. Visit leftmost subtree in preorder
   2.3. Visit remaining subtrees in preorder
3. Visit remaining subtrees in preorder



b

PreOrder

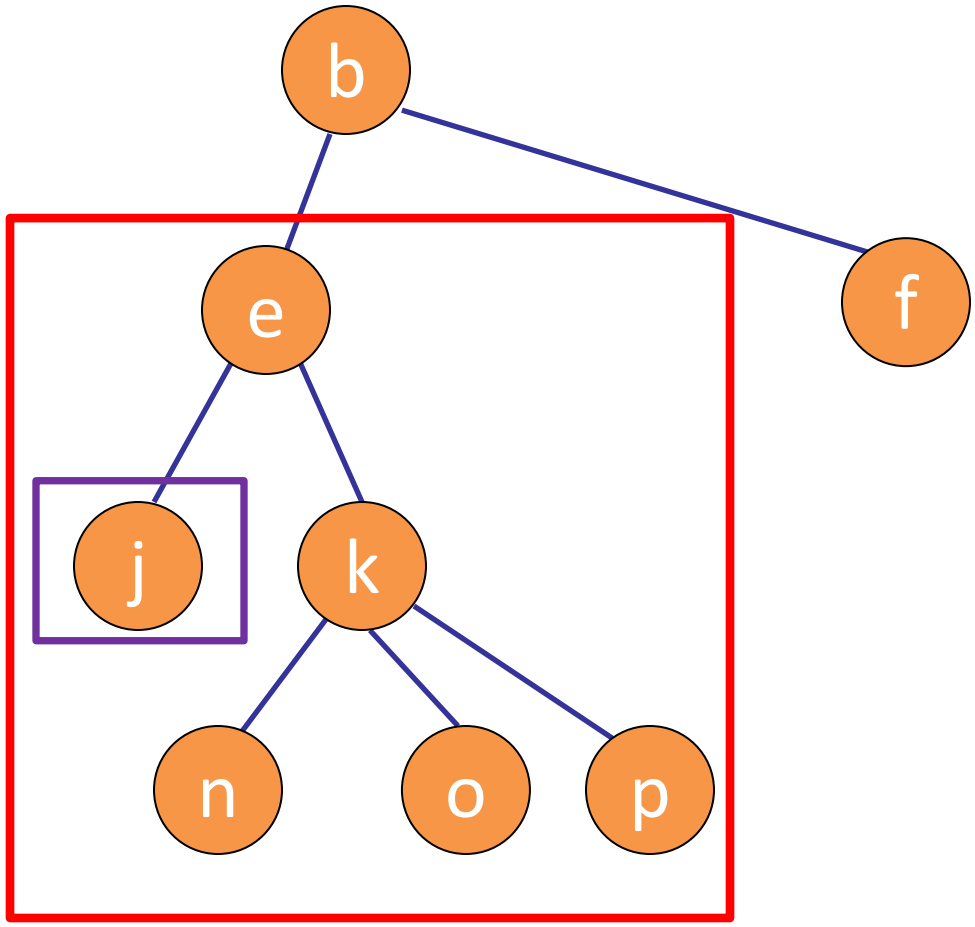
1. Visit root
2. Visit leftmost subtree in preorder
   2.1. Visit root
   2.2. Visit leftmost subtree in preorder
   2.3. Visit remaining subtrees in preorder
3. Visit remaining subtrees in preorder



b e

PreOrder

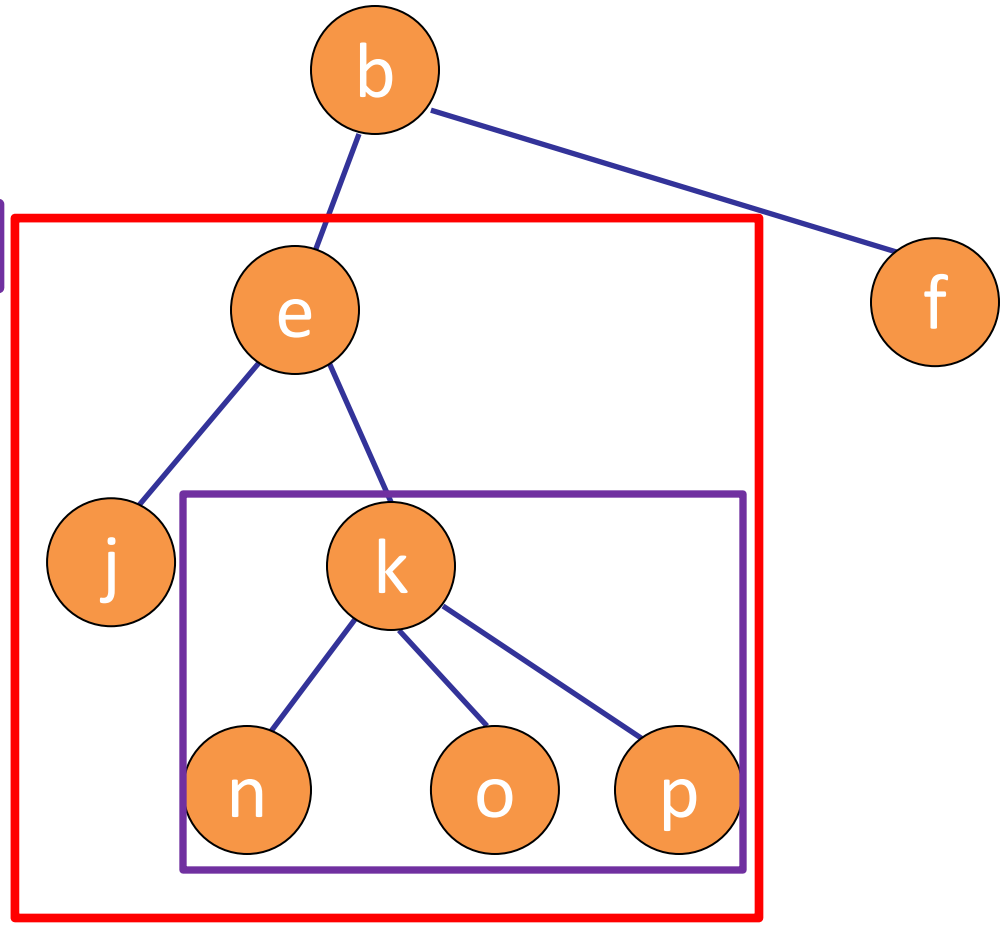
1. Visit root
2. Visit leftmost subtree in preorder
   2.1. Visit root
   2.2. Visit leftmost subtree in preorder
   2.3. Visit remaining subtrees in preorder
3. Visit remaining subtrees in preorder



b e j

PreOrder

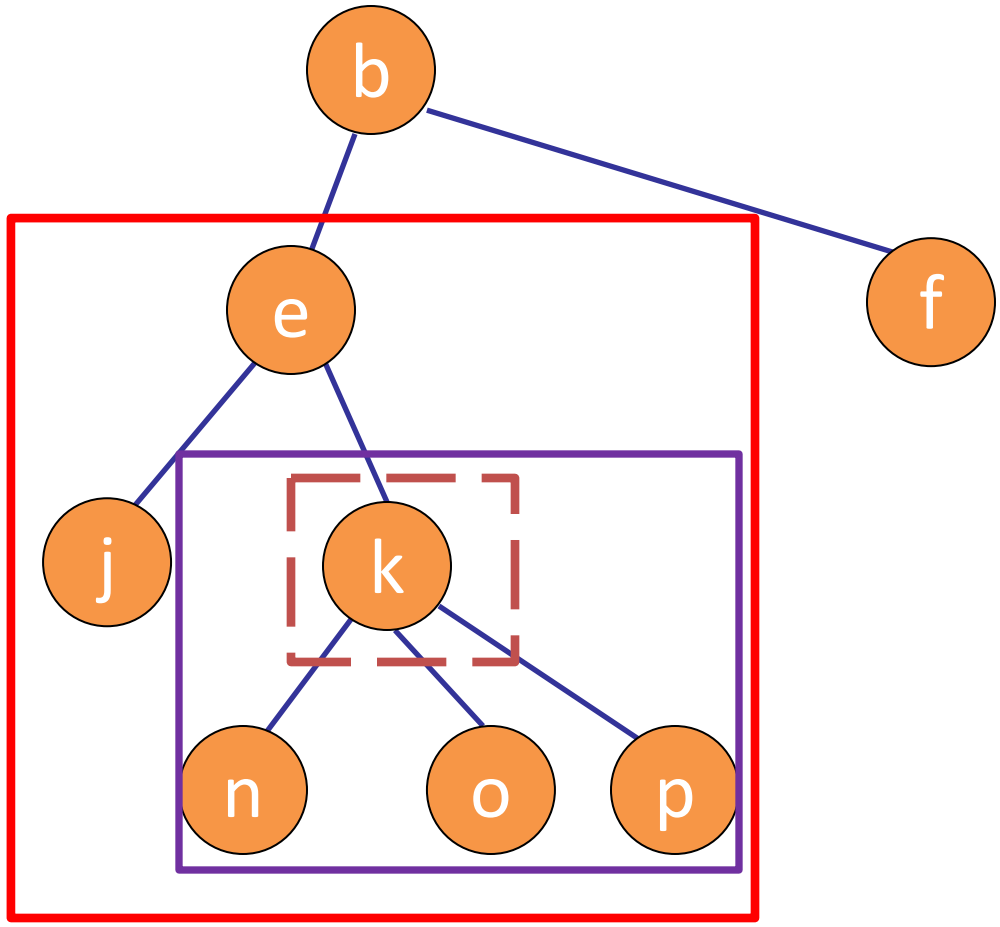
1. Visit root
2. Visit leftmost subtree in preorder
   2.1. Visit root
   2.2. Visit leftmost subtree in preorder
   2.3. Visit remaining subtrees in preorder
3. Visit remaining subtrees in preorder



b e j

PreOrder

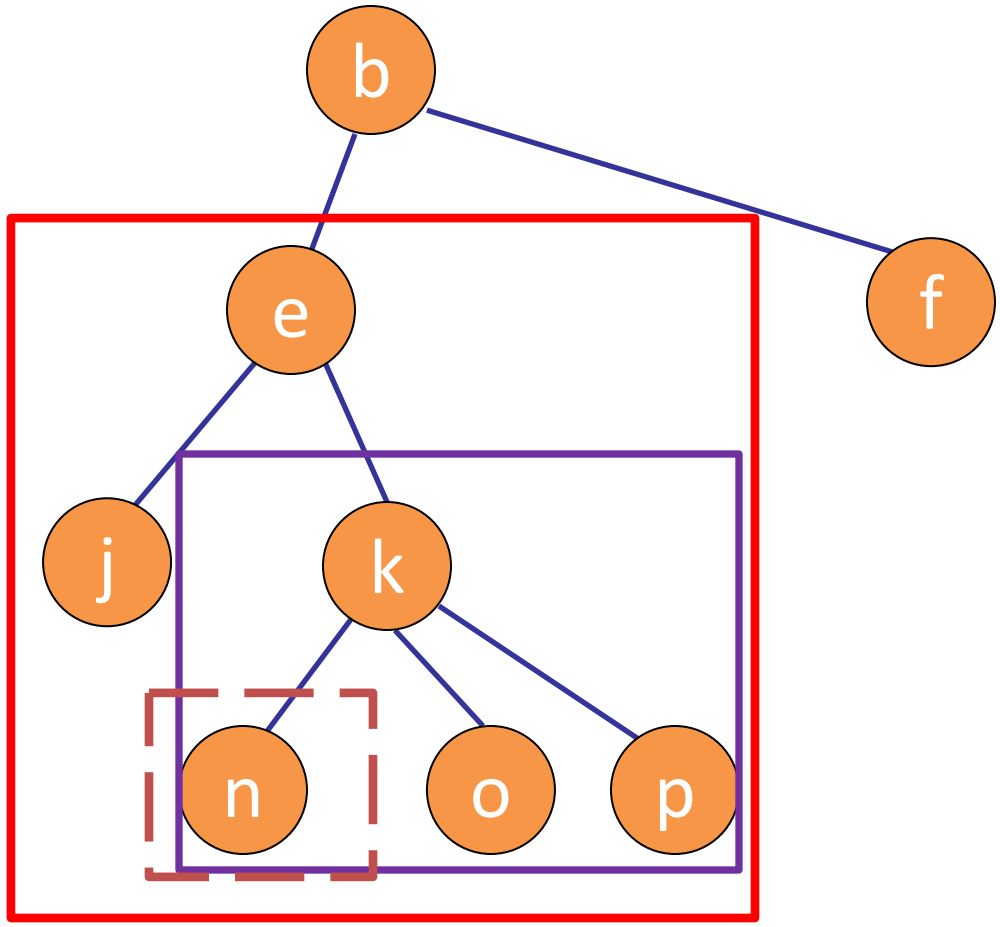
1. Visit root
2. Visit leftmost subtree in preorder
   - 2.1. Visit root
   - 2.2. Visit leftmost subtree in preorder
   - 2.3. Visit remaining subtrees in preorder
     - 2.3.1. Visit root
     - 2.3.2. Visit leftmost subtree in preorder
     - 2.3.3. Visit remaining subtrees in preorder
3. Visit remaining subtrees in preorder



b e j k

PreOrder

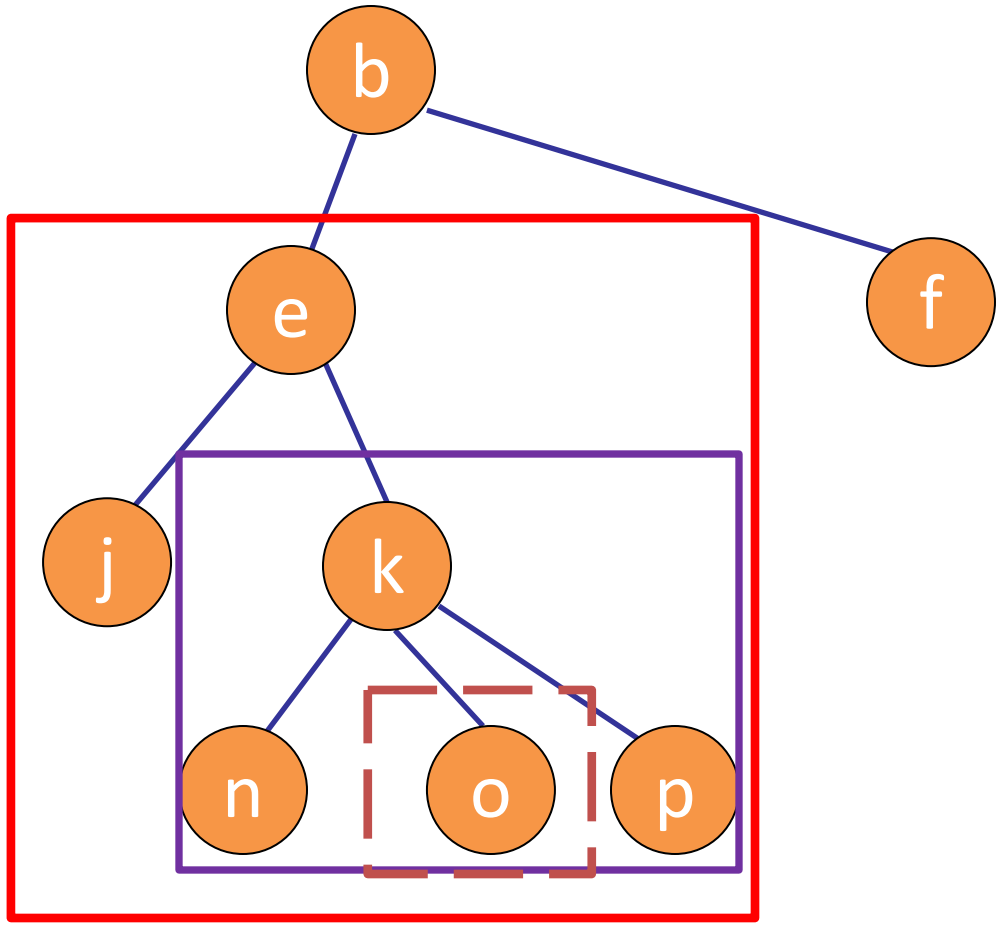
1. Visit root
2. Visit leftmost subtree in preorder
   - 2.1. Visit root
   - 2.2. Visit leftmost subtree in preorder
   - 2.3. Visit remaining subtrees in preorder
     - 2.3.1. Visit root
     - 2.3.2. Visit leftmost subtree in preorder
     - 2.3.3. Visit remaining subtrees in preorder
3. Visit remaining subtrees in preorder



b e j k n

PreOrder

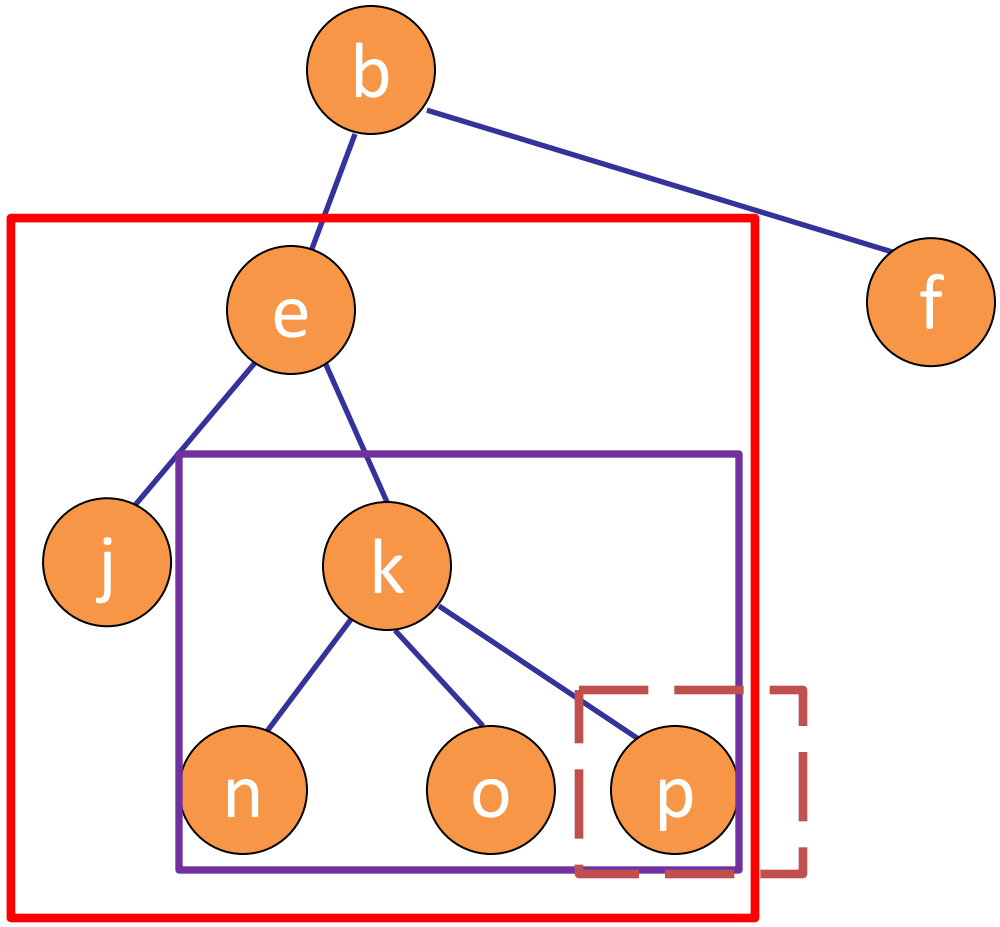
1. Visit root
2. Visit leftmost subtree in preorder
   - 2.1. Visit root
   - 2.2. Visit leftmost subtree in preorder
   - 2.3. Visit remaining subtrees in preorder
     - 2.3.1. Visit root
     - 2.3.2. Visit leftmost subtree in preorder
     - 2.3.3. Visit remaining subtrees in preorder
3. Visit remaining subtrees in preorder



b e j k n o

PreOrder

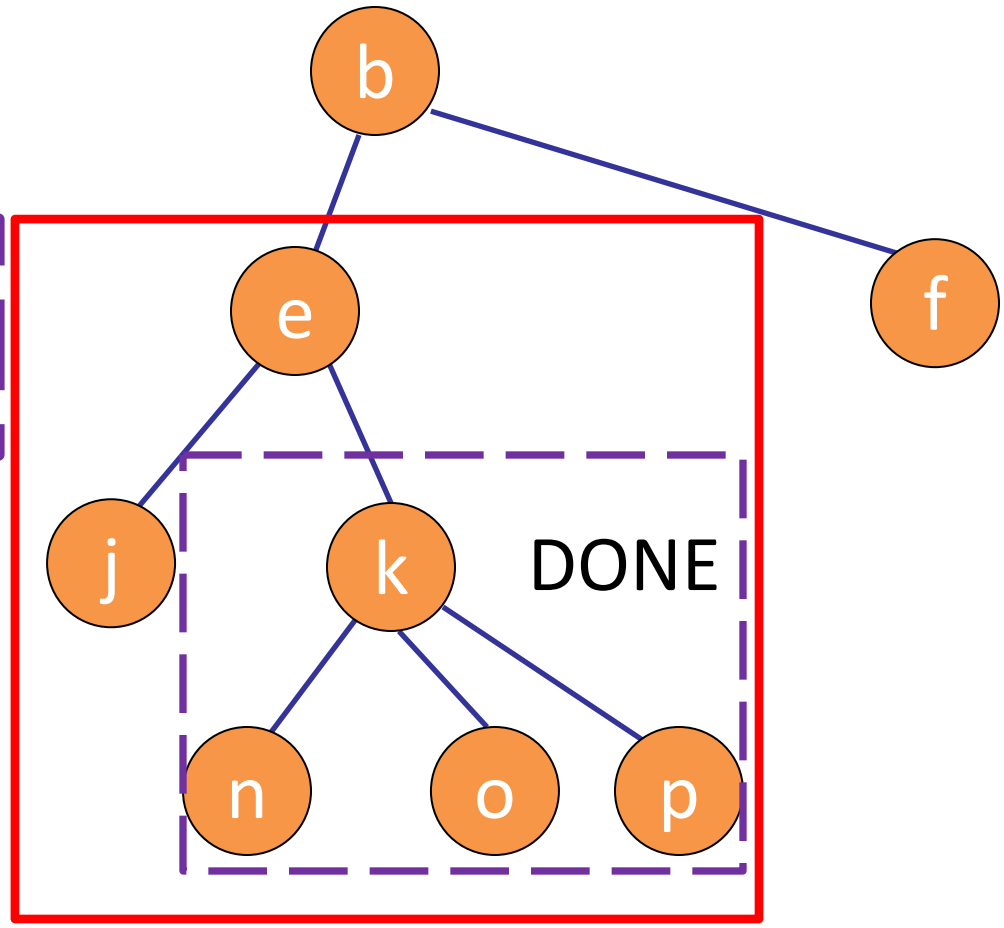
1. Visit root
2. Visit leftmost subtree in preorder
   - 2.1. Visit root
   - 2.2. Visit leftmost subtree in preorder
   - 2.3. Visit remaining subtrees in preorder
     - 2.3.1. Visit root
     - 2.3.2. Visit leftmost subtree in preorder
     - 2.3.3. Visit remaining subtrees in preorder
3. Visit remaining subtrees in preorder



b e j k n o p

PreOrder

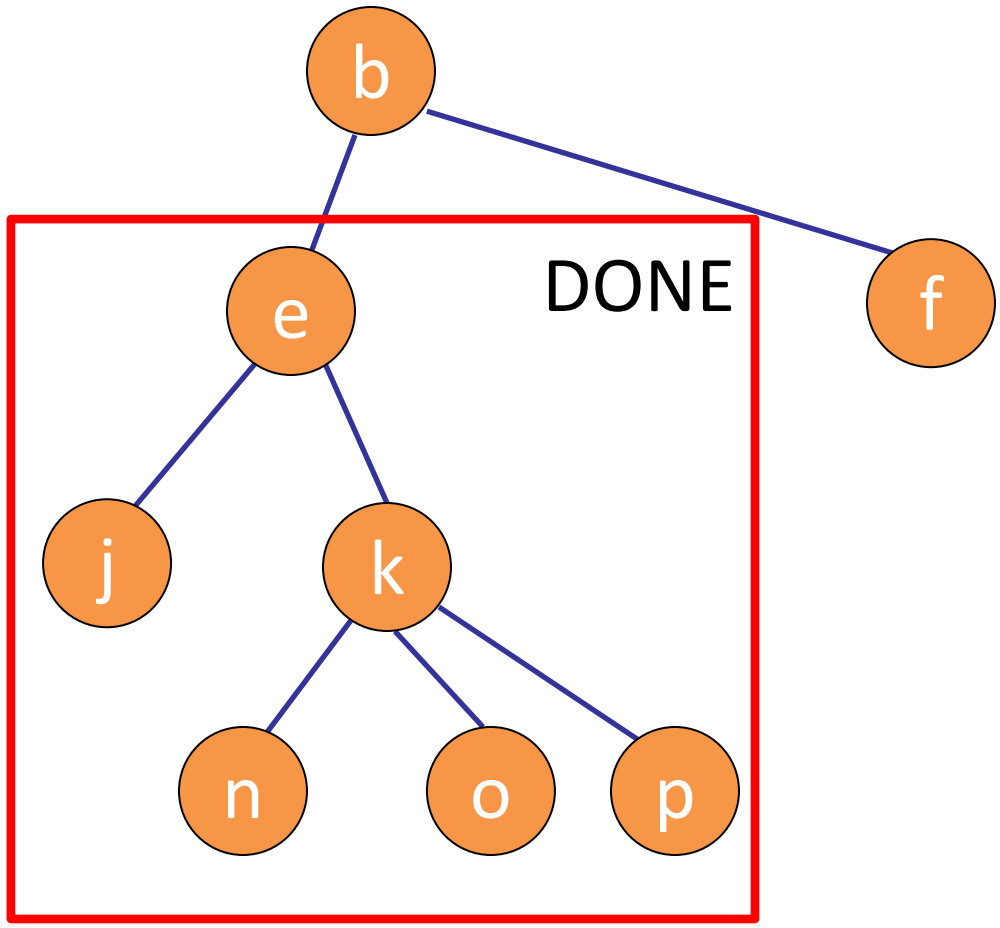
1. Visit root
2. Visit leftmost subtree in preorder
   - 2.1. Visit root
   - 2.2. Visit leftmost subtree in preorder
   - 2.3. Visit remaining subtrees in preorder
     - 2.3.1. Visit root
     - 2.3.2. Visit leftmost subtree in preorder
     - 2.3.3. Visit remaining subtrees in preorder
3. Visit remainingsubtrees in preorder



b e j k n o p

PreOrder

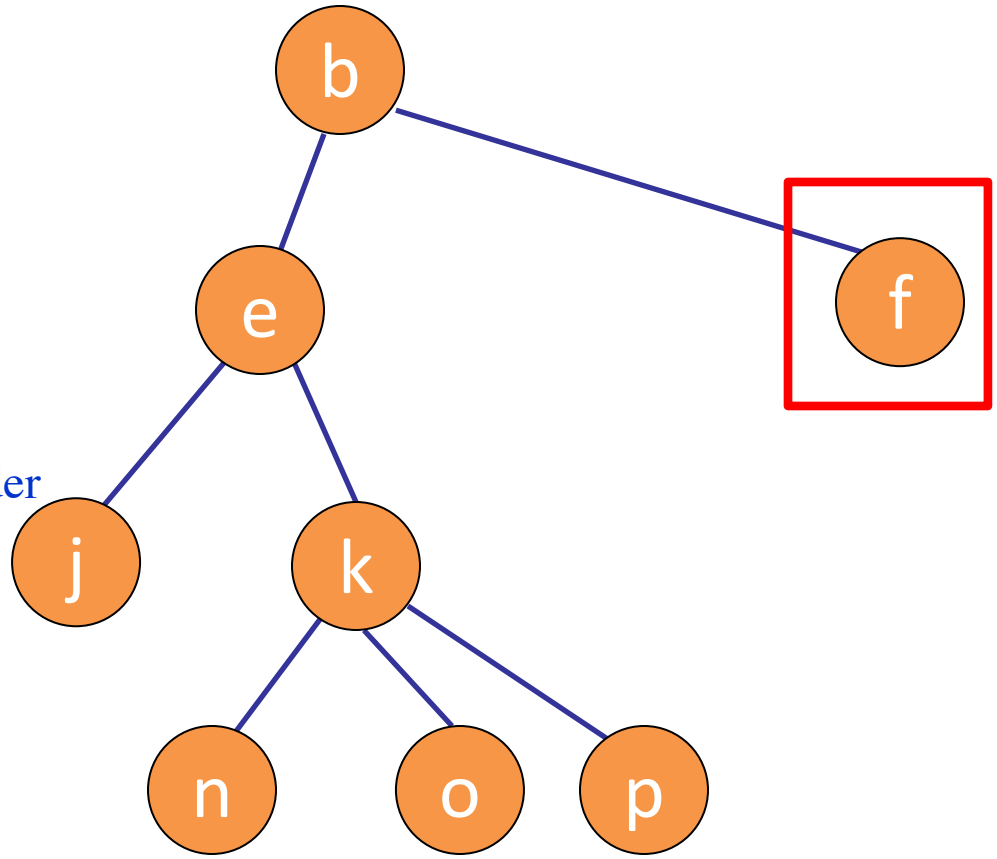
1. Visit root
2. Visit leftmost subtree in preorder
   - 2.1. Visit root
   - 2.2. Visit leftmost subtree in preorder
   - 2.3. Visit remaining subtrees in preorder
     - 2.3.1. Visit root
     - 2.3.2. Visit leftmost subtree in preorder
     - 2.3.3. Visit remaining subtrees in preorder
3. Visit remaining subtrees in preorder



b e j k n o p

PreOrder

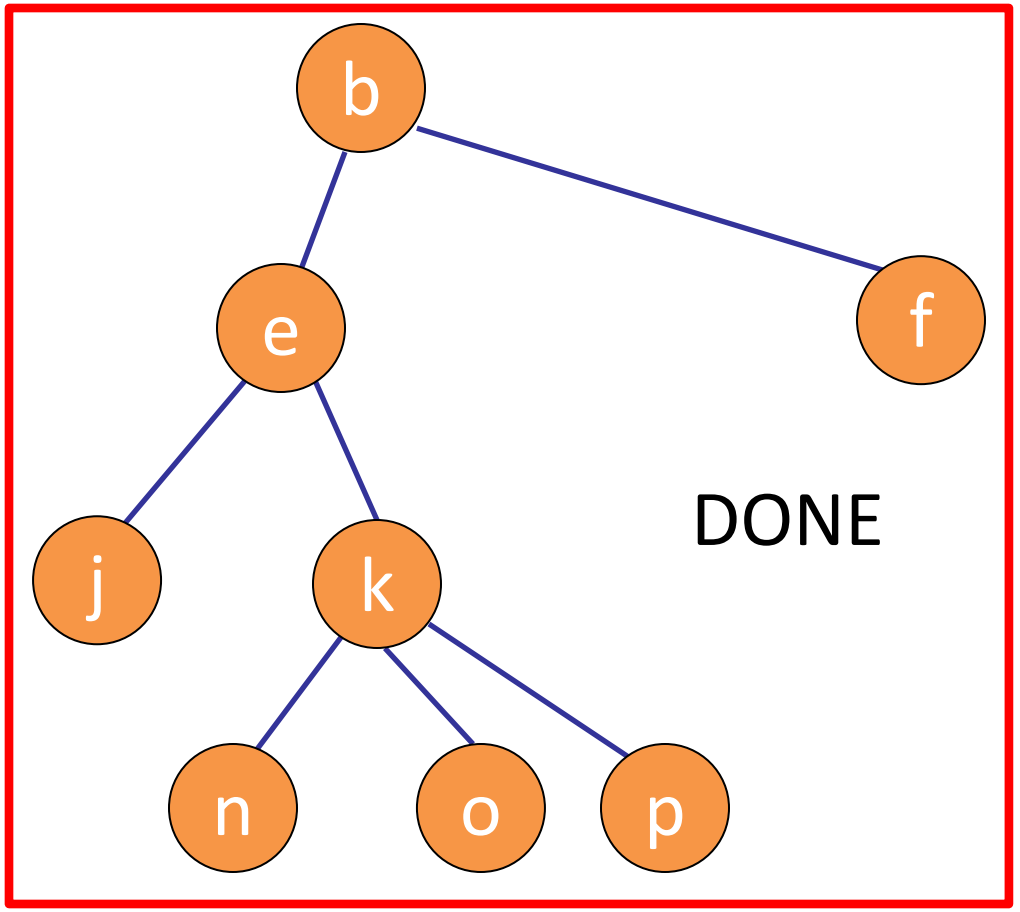
1. Visit root
2. Visit leftmost subtree in preorder
   - 2.1. Visit root
   - 2.2. Visit leftmost subtree in preorder
   - 2.3. Visit remaining subtrees in preorder
     - 2.3.1. Visit root
     - 2.3.2. Visit leftmost subtree in preorder
     - 2.3.3. Visit remaining subtrees in preorder
3. Visit remaining subtrees in preorder



b e j k n o p f

PreOrder

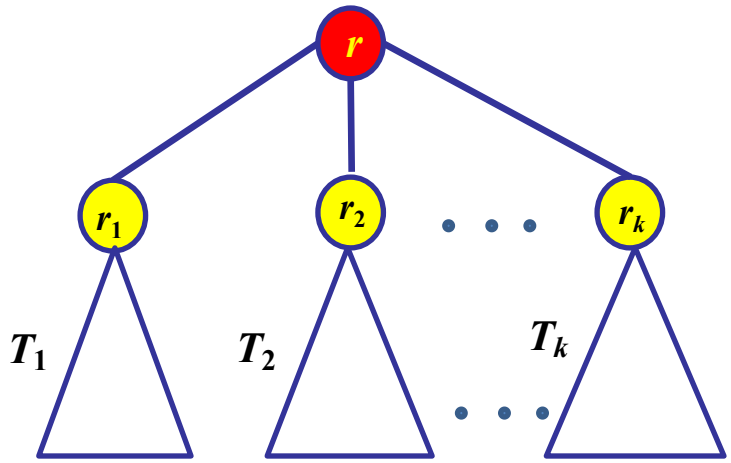
1. Visit root
2. Visit leftmost subtree in preorder
   - 2.1. Visit root
   - 2.2. Visit leftmost subtree in preorder
   - 2.3. Visit remaining subtrees in preorder
     - 2.3.1. Visit root
     - 2.3.2. Visit leftmost subtree in preorder
     - 2.3.3. Visit remaining subtrees in preorder
3. Visit remaining subtrees in preorder



b e j k n o p f

Inorder Traversal

- Inorder traversal on tree T:

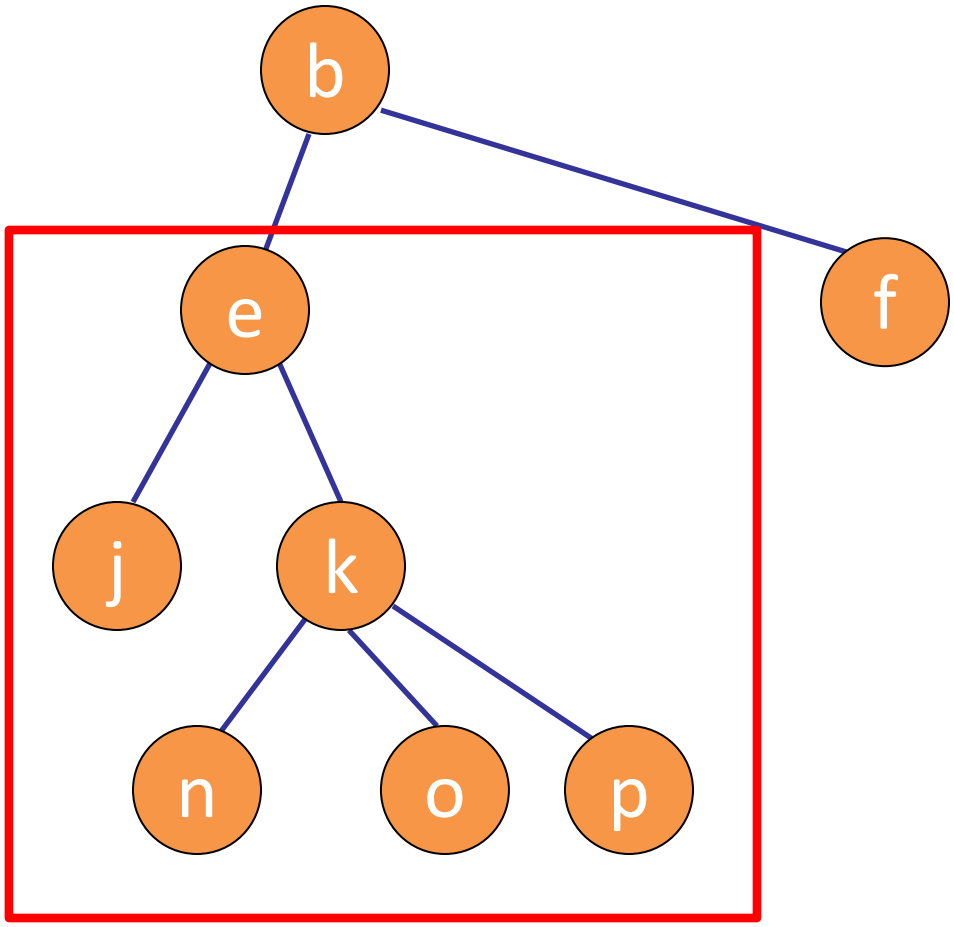


- Step 1: visit T_1 in inorder,
- Step 2: visit root ***r***,
- Step 3: visit T_2 in inorder,
- ……..
- Step k+1: visit T_k in inorder

```
procedure inorder(T: ordered rooted tree)
  r := root of T
  if r is a leaf then visit r
  else
  begin
     l := first child of r from left to right
     T(l) := subtree with l as its root
     inorder(T(l))        Step 1

     visit r              Step 2
     for each child c of r except for l left to right
        T(c) := subtree with c as its root
        inorder(T(c))     Step 3, 4..
  end
```

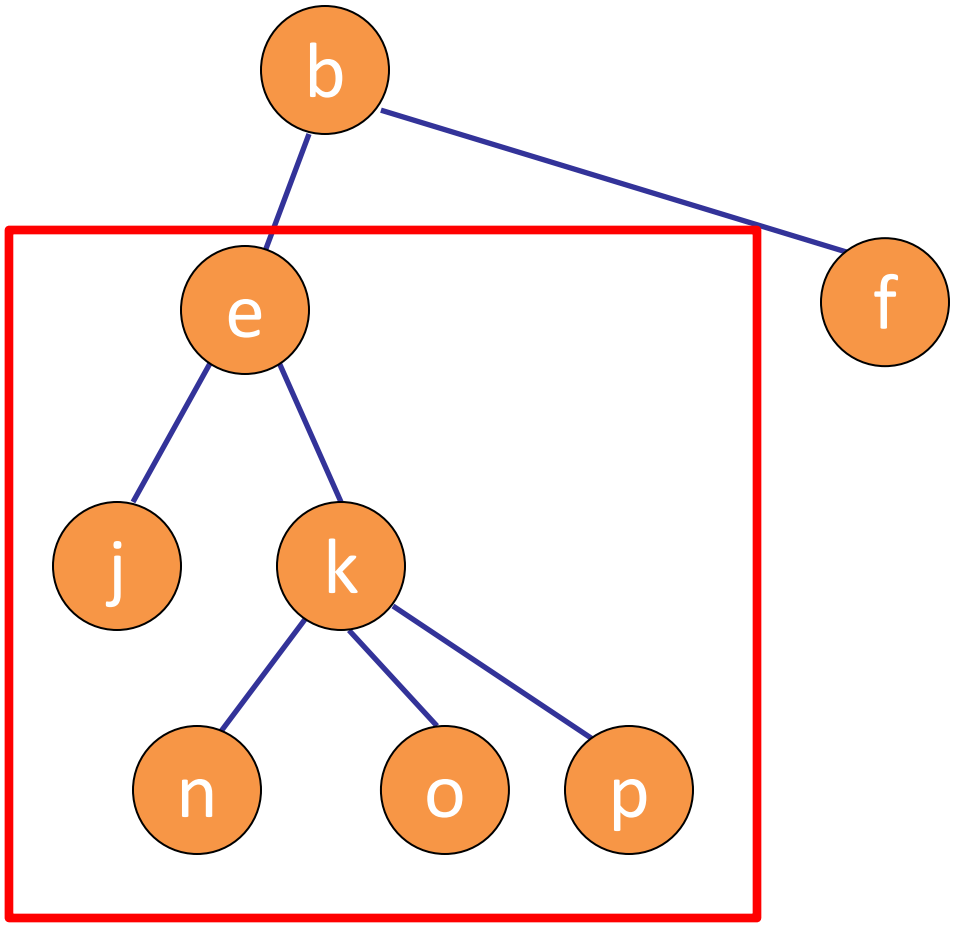
InOrder

1. Visit leftmost subtree in Inorder
2. Visit root
3. Visit remaining subtrees in Inorder



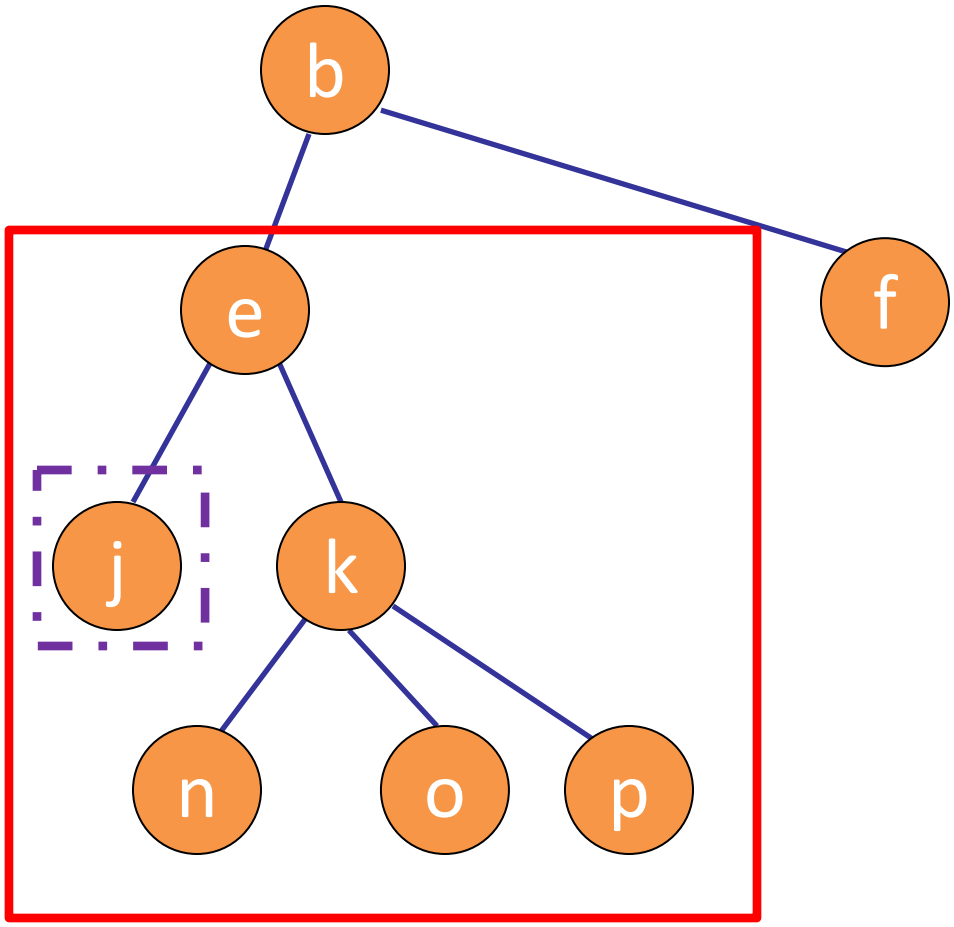
InOrder

1. Visit leftmost subtree in Inorder
   1.1. Visit leftmost subtree in Inorder
   1.2. Visit root
   1.3. Visit remaining subtrees in Inorder
2. Visit root
3. Visit remaining subtrees in Inorder



InOrder

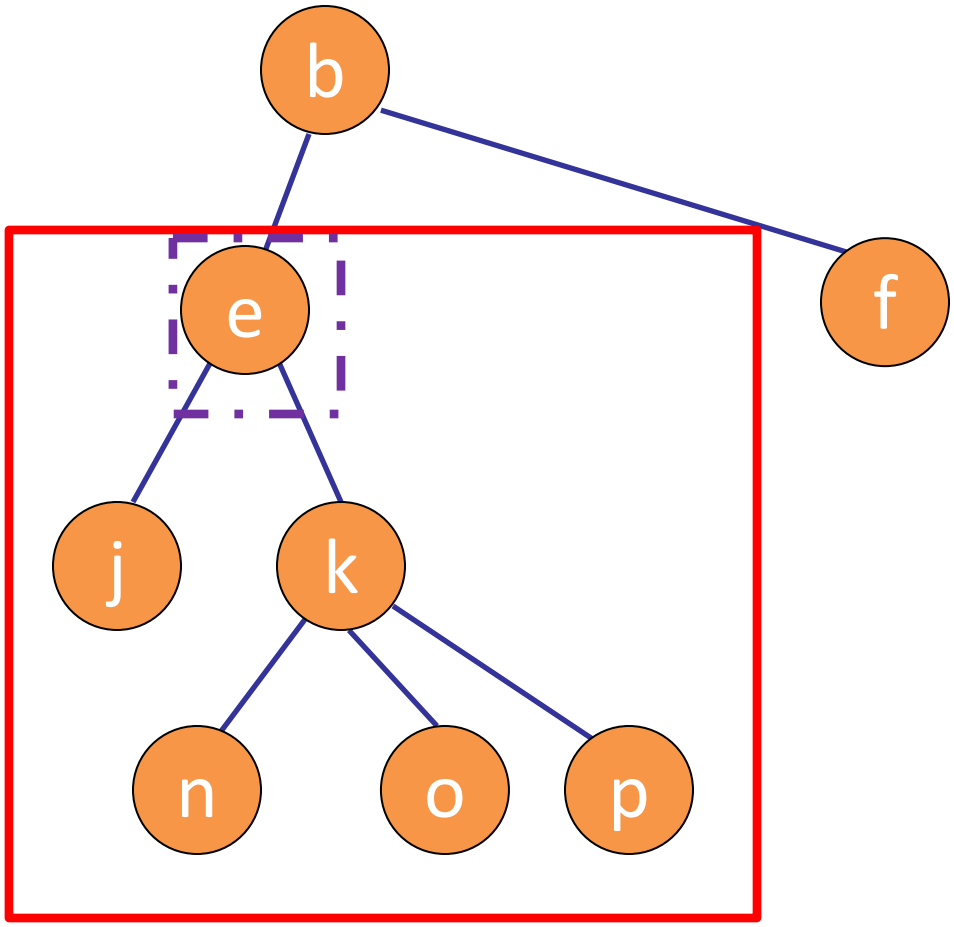
1. Visit leftmost subtree in Inorder
   1.1. Visit leftmost subtree in Inorder
   1.2. Visit root
   1.3. Visit remaining subtrees in Inorder
2. Visit root
3. Visit remaining subtrees in Inorder



j

InOrder

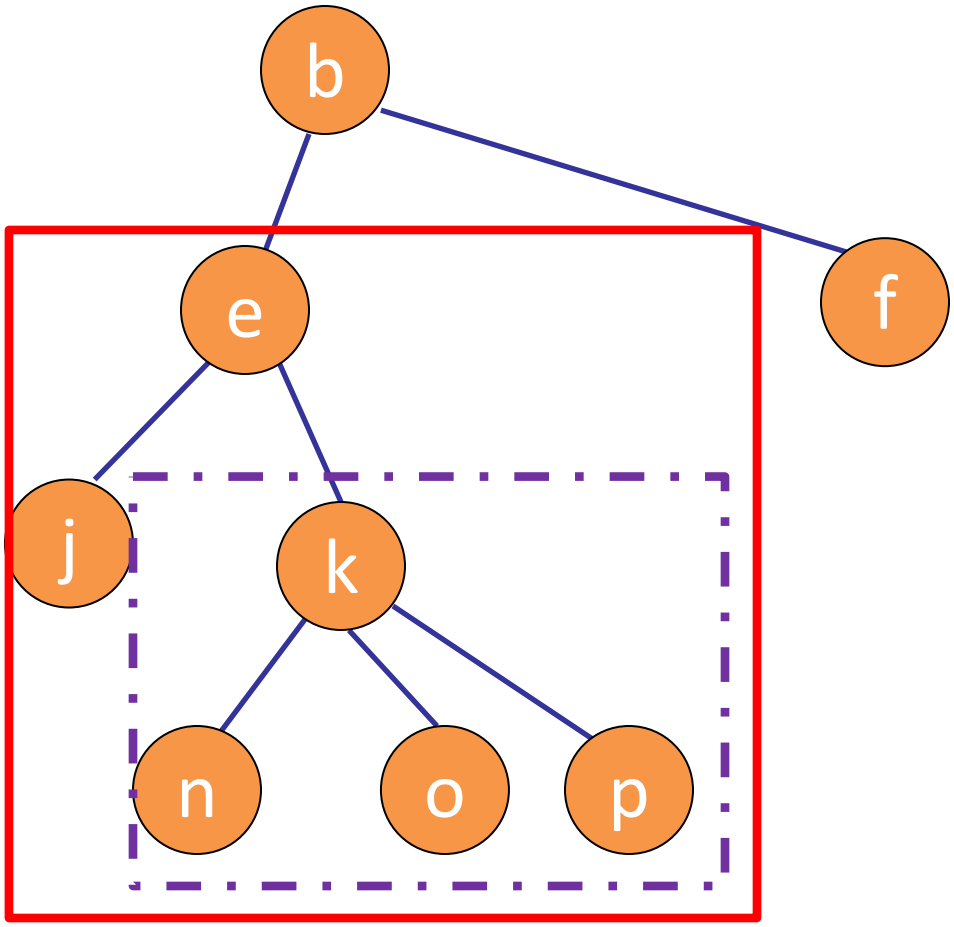
1. Visit leftmost subtree in Inorder
   1.1. Visit leftmost subtree in Inorder
   1.2. Visit root
   1.3. Visit remaining subtrees in Inorder
2. Visit root
3. Visit remaining subtrees in Inorder



j e

InOrder

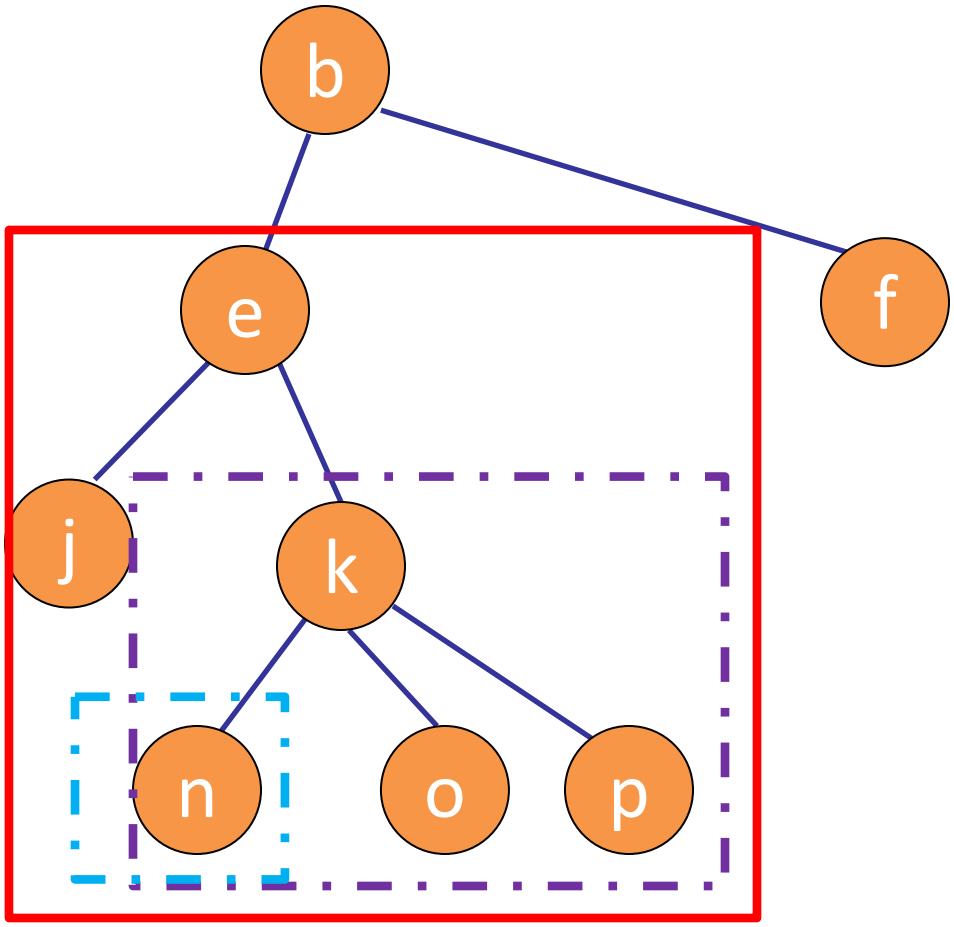
1. Visit leftmost subtree in Inorder
   1.1. Visit leftmost subtree in Inorder
   1.2. Visit root
   1.3. Visit remaining subtrees in Inorder
2. Visit root
3. Visit remaining subtrees in Inorder



j e

InOrder

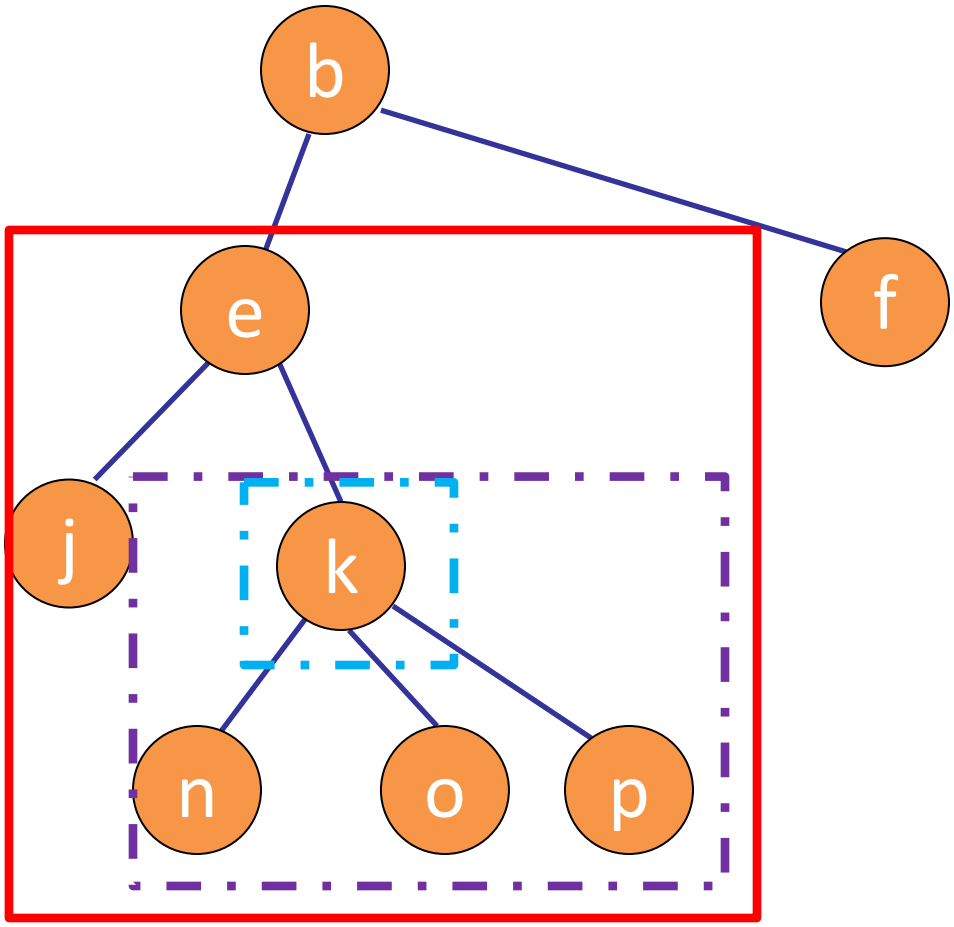
1. Visit leftmost subtree in Inorder
   1.1. Visit leftmost subtree in Inorder
   1.2. Visit root
   1.3. Visit remaining subtrees in Inorder
      1.3.1. Visit leftmost subtree in Inorder
      1.3.2. Visit root
      1.3.3. Visit remaining subtrees in Inorder
2. Visit root
3. Visit remaining subtrees in Inorder



j e n

InOrder

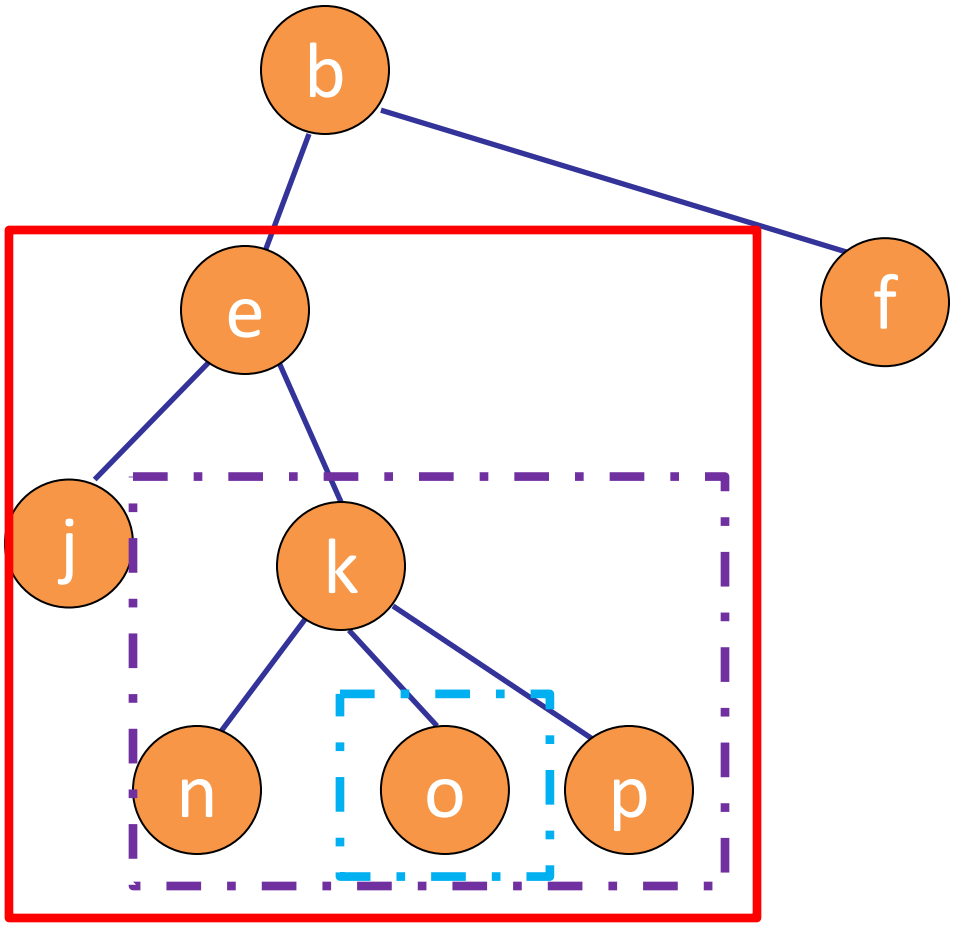
1. Visit leftmost subtree in Inorder
   1.1. Visit leftmost subtree in Inorder
   1.2. Visit root
   1.3. Visit remaining subtrees in Inorder
      1.3.1. Visit leftmost subtree in Inorder
      1.3.2. Visit root
      1.3.3. Visit remaining subtrees in Inorder
2. Visit root
3. Visit remaining subtrees in Inorder



j e n k

InOrder

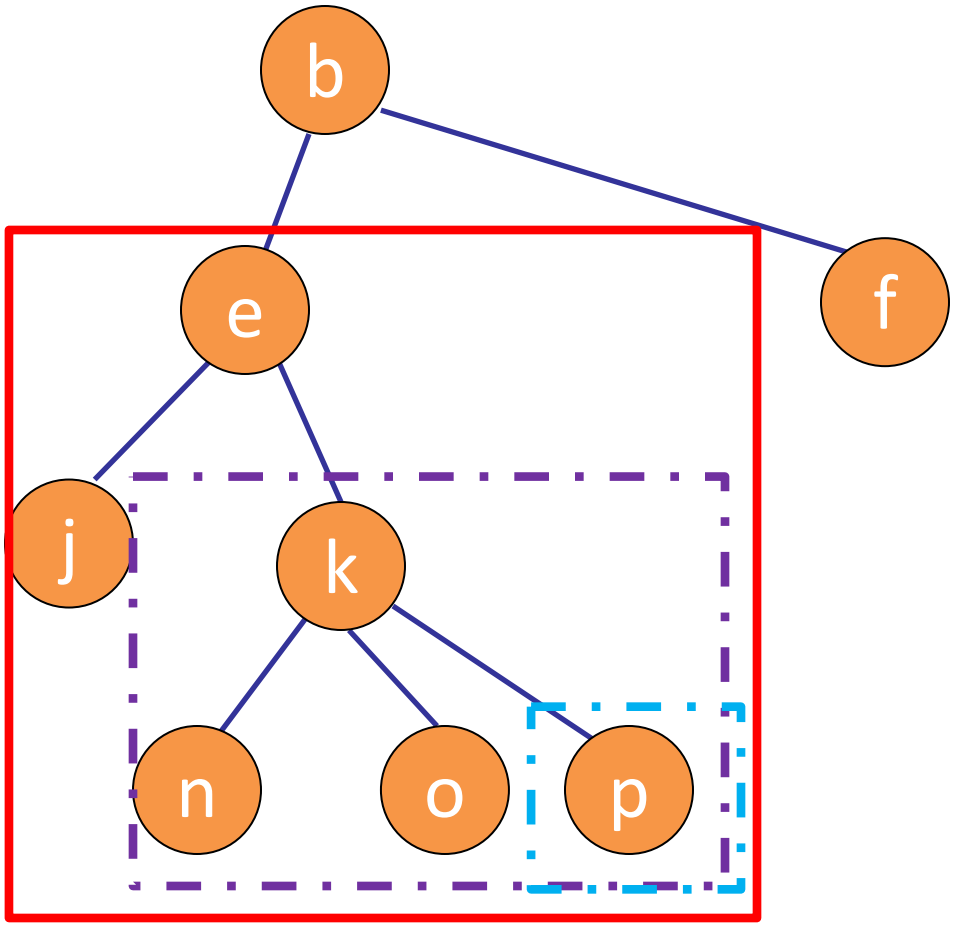
1. Visit leftmost subtree in Inorder
   1.1. Visit leftmost subtree in Inorder
   1.2. Visit root
   1.3. Visit remaining subtrees in Inorder
      1.3.1. Visit leftmost subtree in Inorder
      1.3.2. Visit root
      1.3.3. Visit remaining subtrees in Inorder
2. Visit root
3. Visit remaining subtrees in Inorder



j e n k o

InOrder

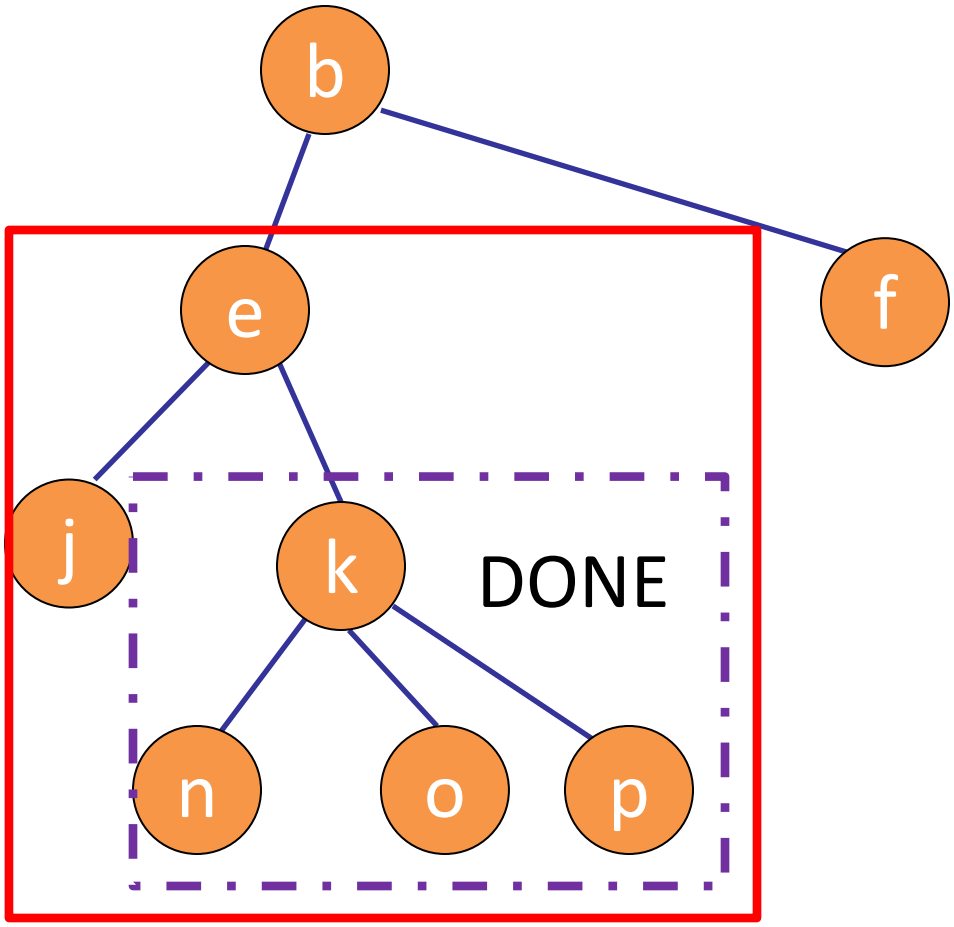
1. Visit leftmost subtree in Inorder
   1.1. Visit leftmost subtree in Inorder
   1.2. Visit root
   1.3. Visit remaining subtrees in Inorder
      1.3.1. Visit leftmost subtree in Inorder
      1.3.2. Visit root
      1.3.3. Visit remaining subtrees in Inorder
2. Visit root
3. Visit remaining subtrees in Inorder



j e n k o p

InOrder

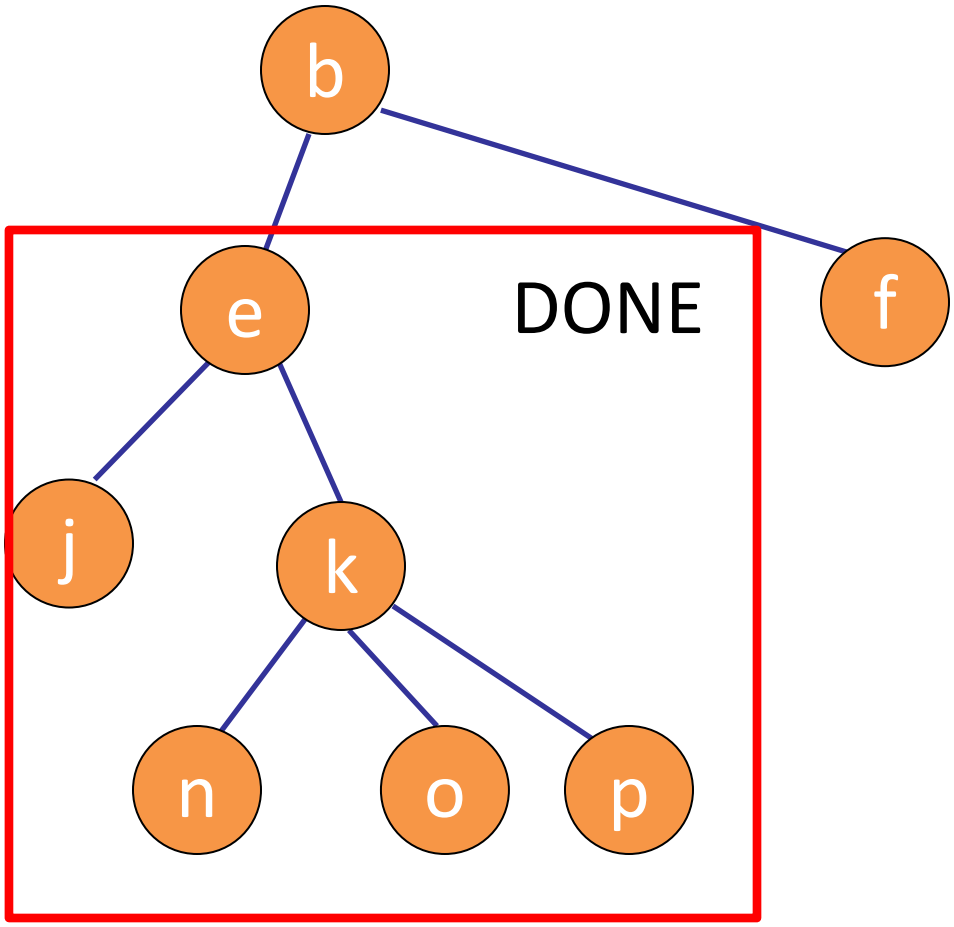
1. Visit leftmost subtree in Inorder
   1.1. Visit leftmost subtree in Inorder
   1.2. Visit root
   1.3. Visit remaining subtrees in Inorder
      1.3.1. Visit leftmost subtree in Inorder
      1.3.2. Visit root
      1.3.3. Visit remaining subtrees in Inorder
2. Visit root
3. Visit remaining subtrees in Inorder



j e n k o p

InOrder

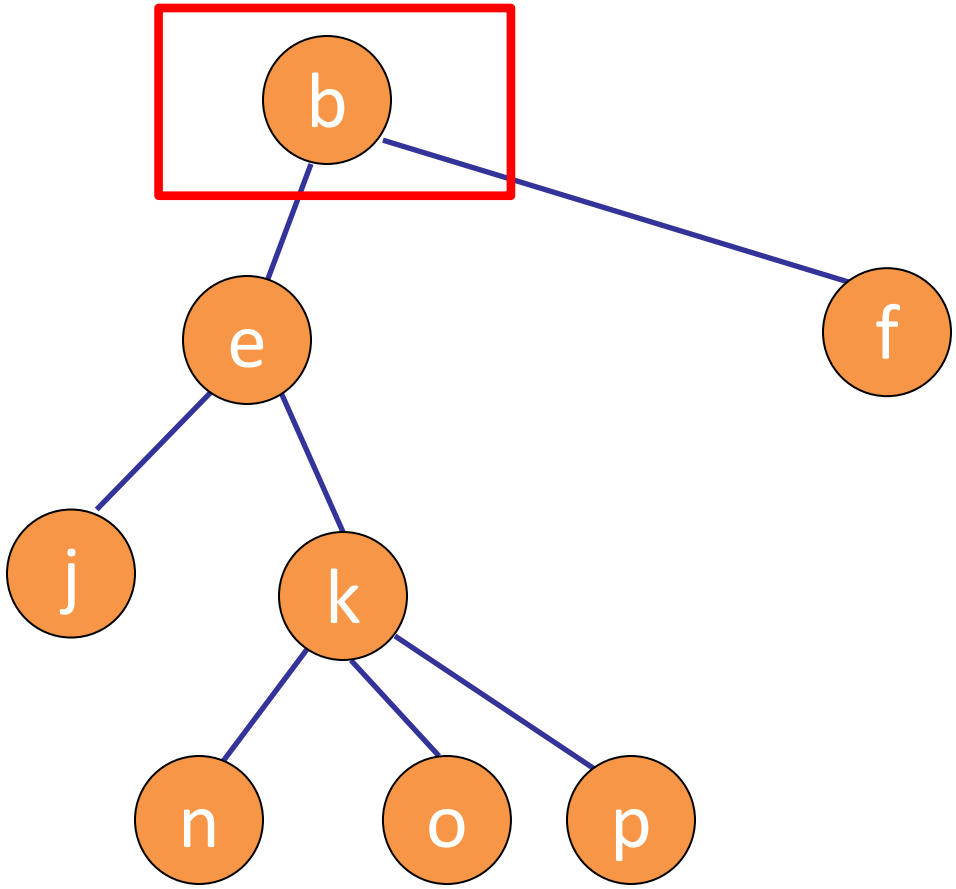
1. Visit leftmost subtree in Inorder
   1.1. Visit leftmost subtree in Inorder
   1.2. Visit root
   1.3. Visit remaining subtrees in Inorder
      1.3.1. Visit leftmost subtree in Inorder
      1.3.2. Visit root
      1.3.3. Visit remaining subtrees in Inorder
2. Visit root
3. Visit remaining subtrees in Inorder



j e n k o p

InOrder

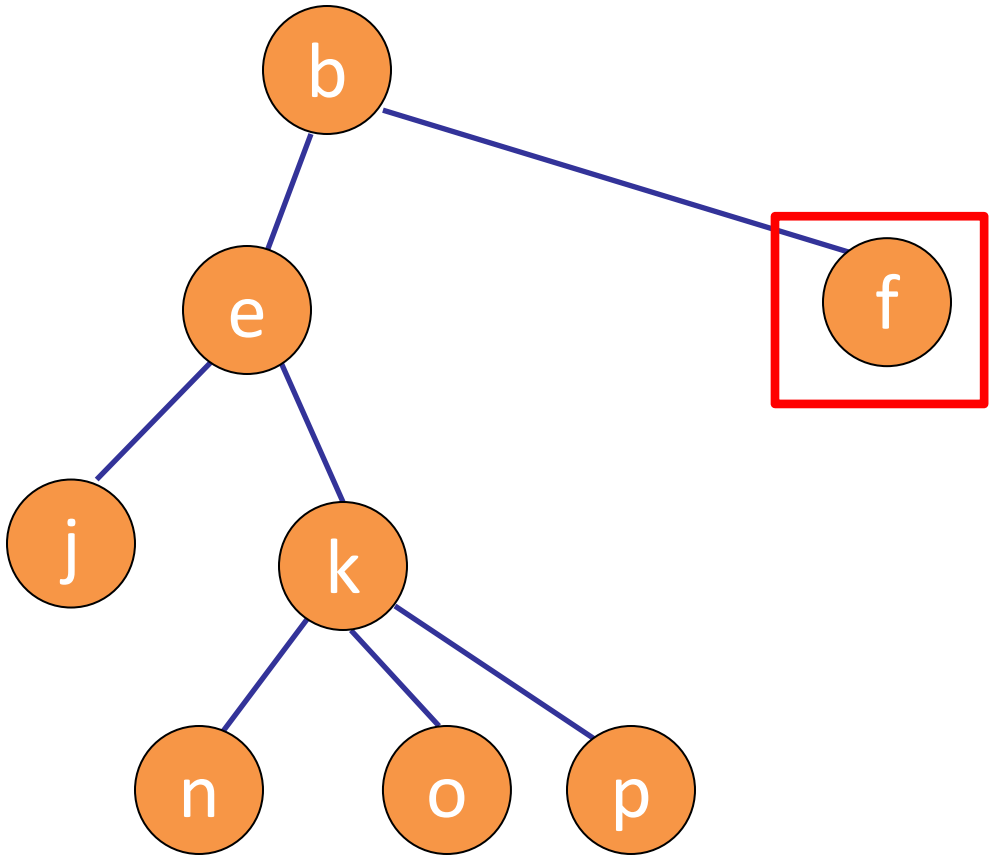
1. Visit leftmost subtree in Inorder
   1.1. Visit leftmost subtree in Inorder
   1.2. Visit root
   1.3. Visit remaining subtrees in Inorder
      1.3.1. Visit leftmost subtree in Inorder
      1.3.2. Visit root
      1.3.3. Visit remaining subtrees in Inorder
2. Visit root
3. Visit remaining subtrees in Inorder



j e n k o p b

InOrder

1. Visit leftmost subtree in Inorder
   1.1. Visit leftmost subtree in Inorder
   1.2. Visit root
   1.3. Visit remaining subtrees in Inorder
      1.3.1. Visit leftmost subtree in Inorder
      1.3.2. Visit root
      1.3.3. Visit remaining subtrees in Inorder
2. Visit root
3. Visit remaining subtrees in Inorder

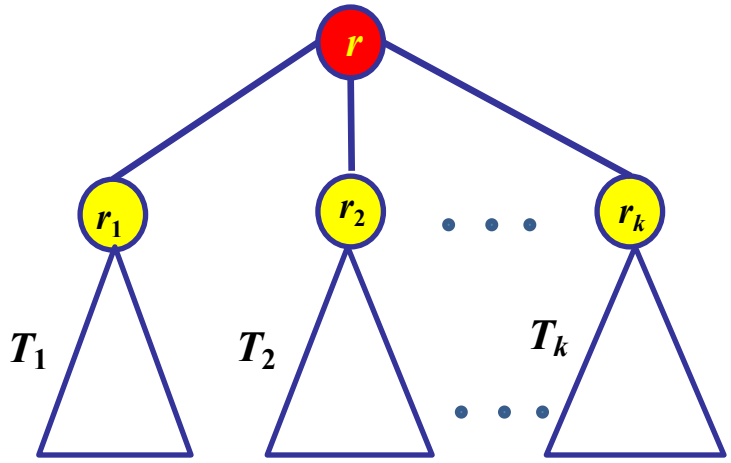


j e n k o p b f

Postorder Traversal

- In a postorder traversal, a node is visited after its descendants.

Example: Postorder traversal on tree T:

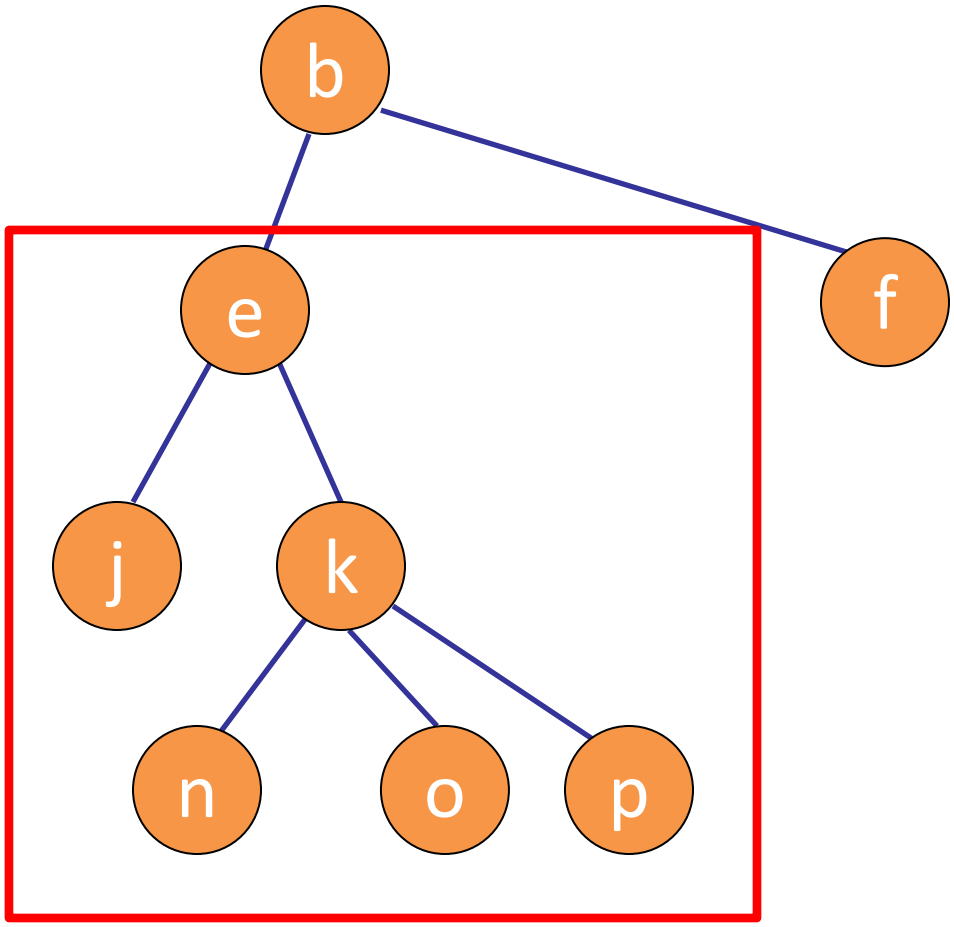



```
procedure postorder(T: ordered rooted tree)
   r := root of T
   for each child c of r from left to right
   begin
       T(c) := subtree with c as its root
       postorder(T(c))
   end
   visit r
```

- Step 1: visit T_1 in postorder,
- Step 2: visit T_2 in postorder,
- ……..
- Step k: visit T_k in postorder,
- Step k+1: visit root r

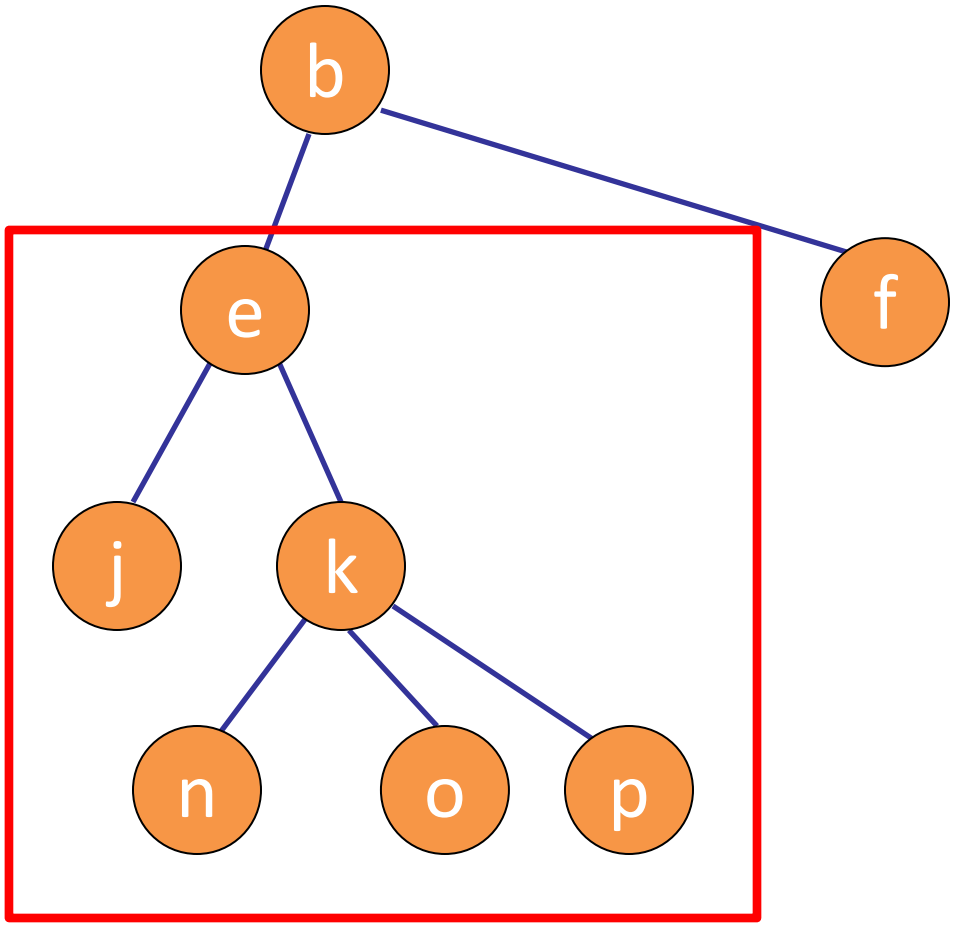
PostOrder

1. Visit leftmost subtree in Postorder
2. Visit remaining subtrees in Postorder
3. Visit root



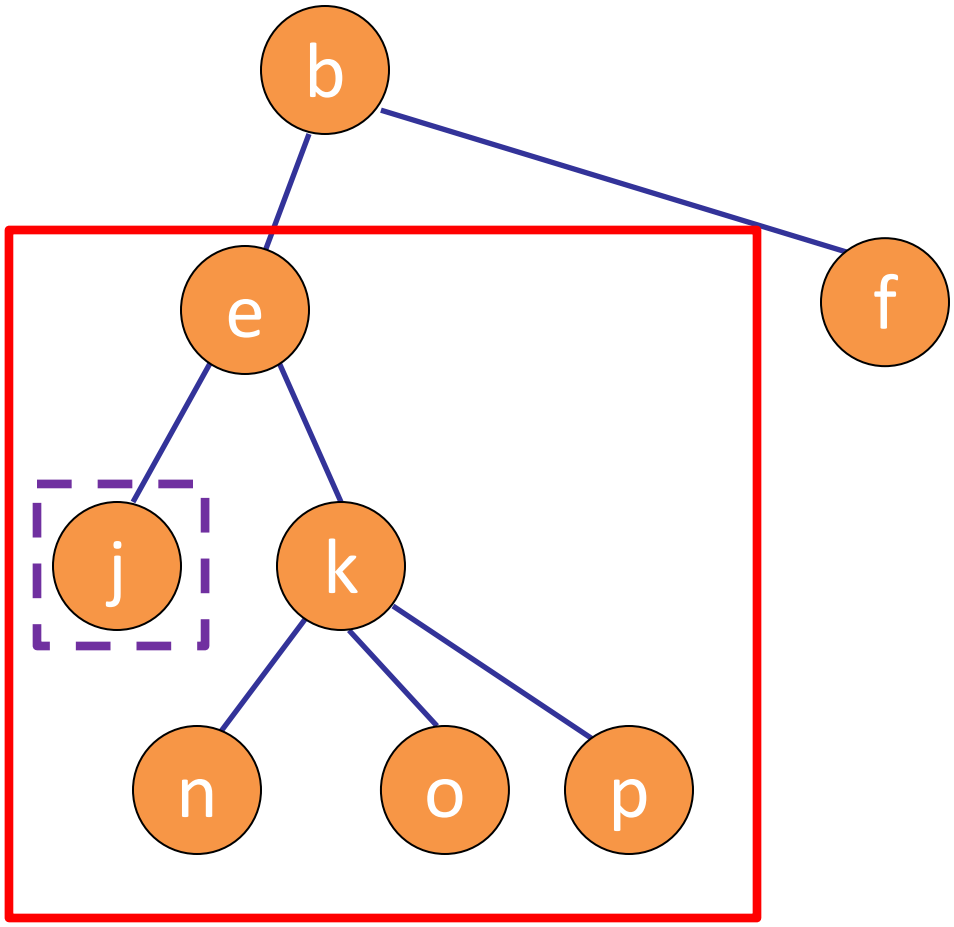
PostOrder

1. Visit leftmost subtree in Postorder
   1.1. Visit leftmost subtree in Postorder
   1.2. Visit remaining subtrees in Postorder
   1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root



PostOrder

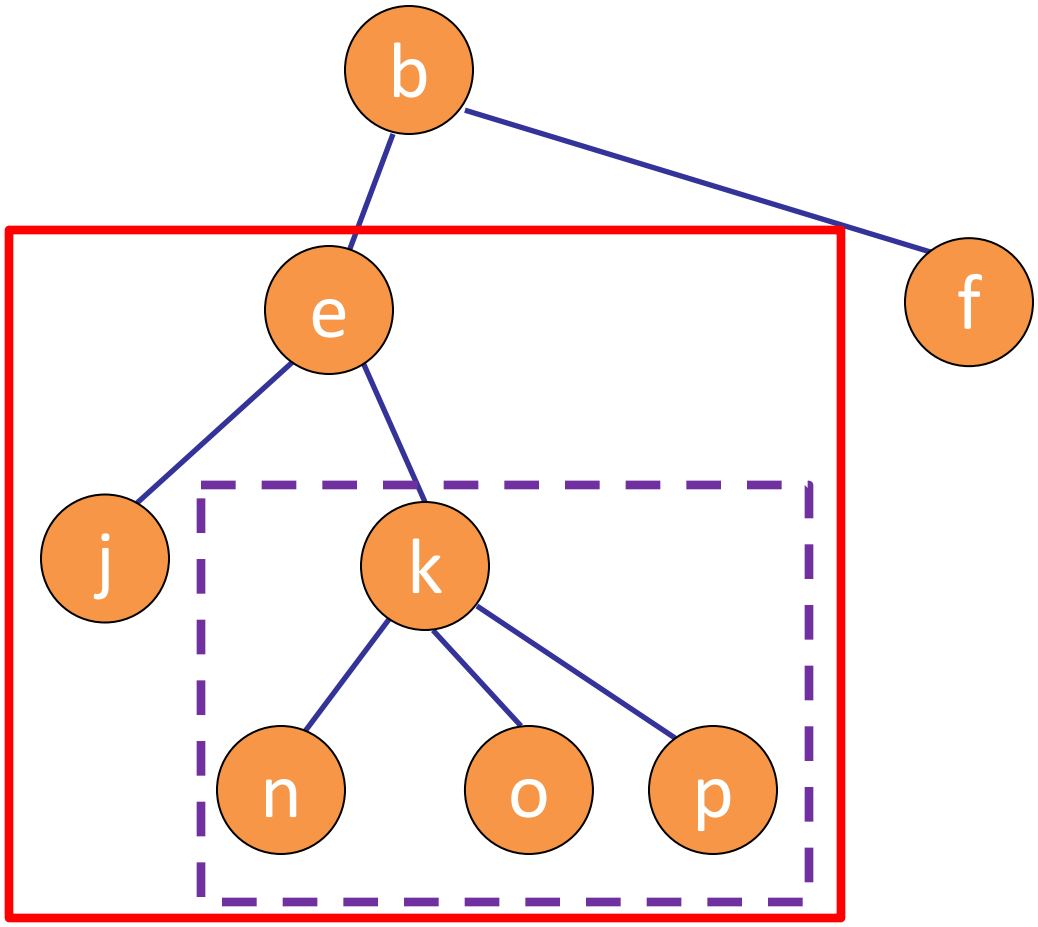
1. Visit leftmost subtree in Postorder
   1.1. Visit leftmost subtree in Postorder
   1.2. Visit remaining subtrees in Postorder
   1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root



j

PostOrder

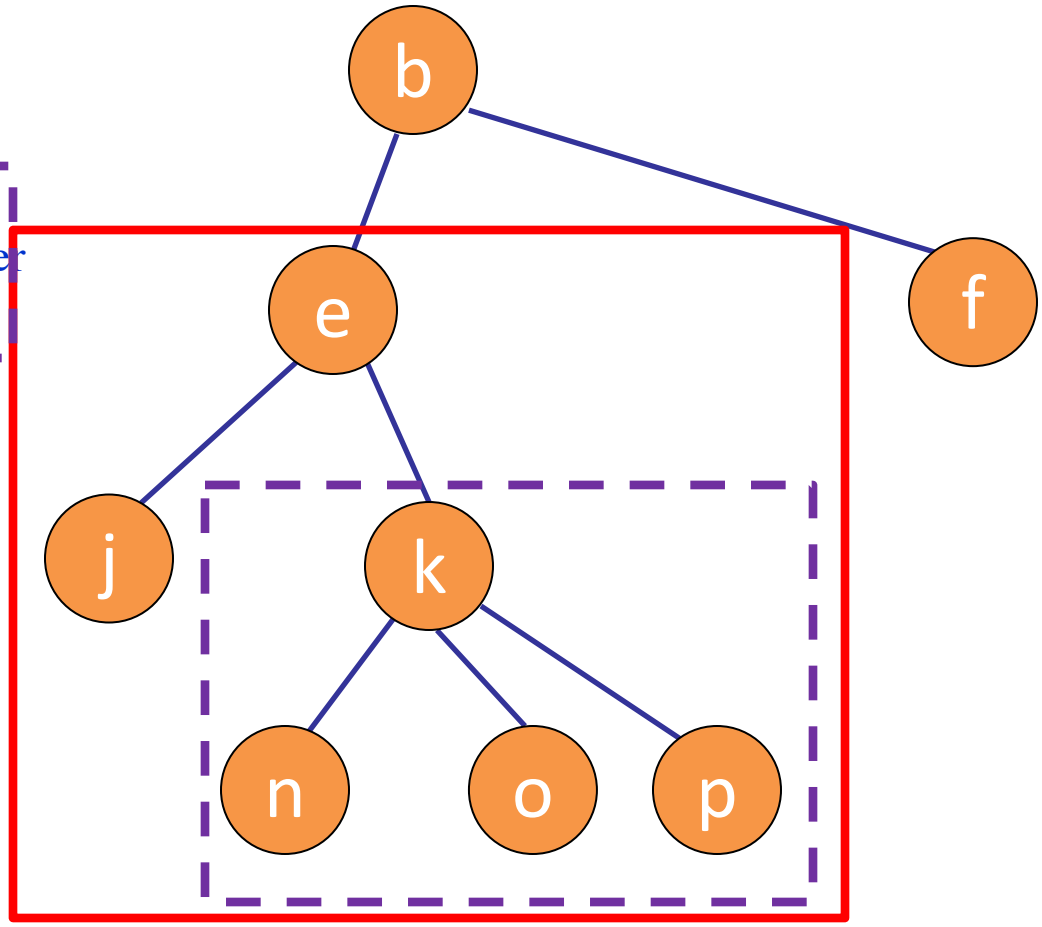
1. Visit leftmost subtree in Postorder
   1.1. Visit leftmost subtree in Postorder
   1.2. Visit remaining subtrees in Postorder
   1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root



j

PostOrder

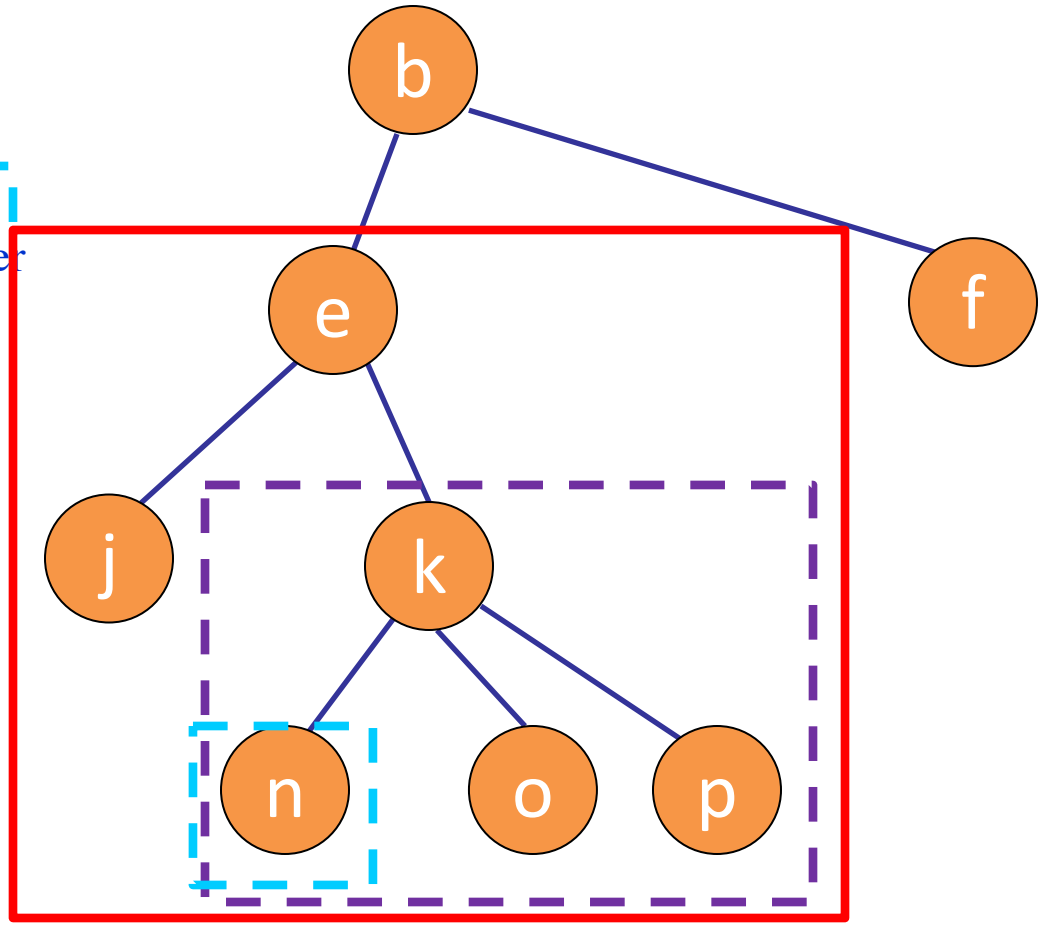
1. Visit leftmost subtree in Postorder
   1.1. Visit leftmost subtree in Postorder
   1.2. Visit remaining subtrees in Postorder
      1.2.1. Visit leftmost subtree in Postorder
      1.2.2. Visit remaining subtrees in Postorder
      1.2.3. Visit root
   1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root



j

PostOrder

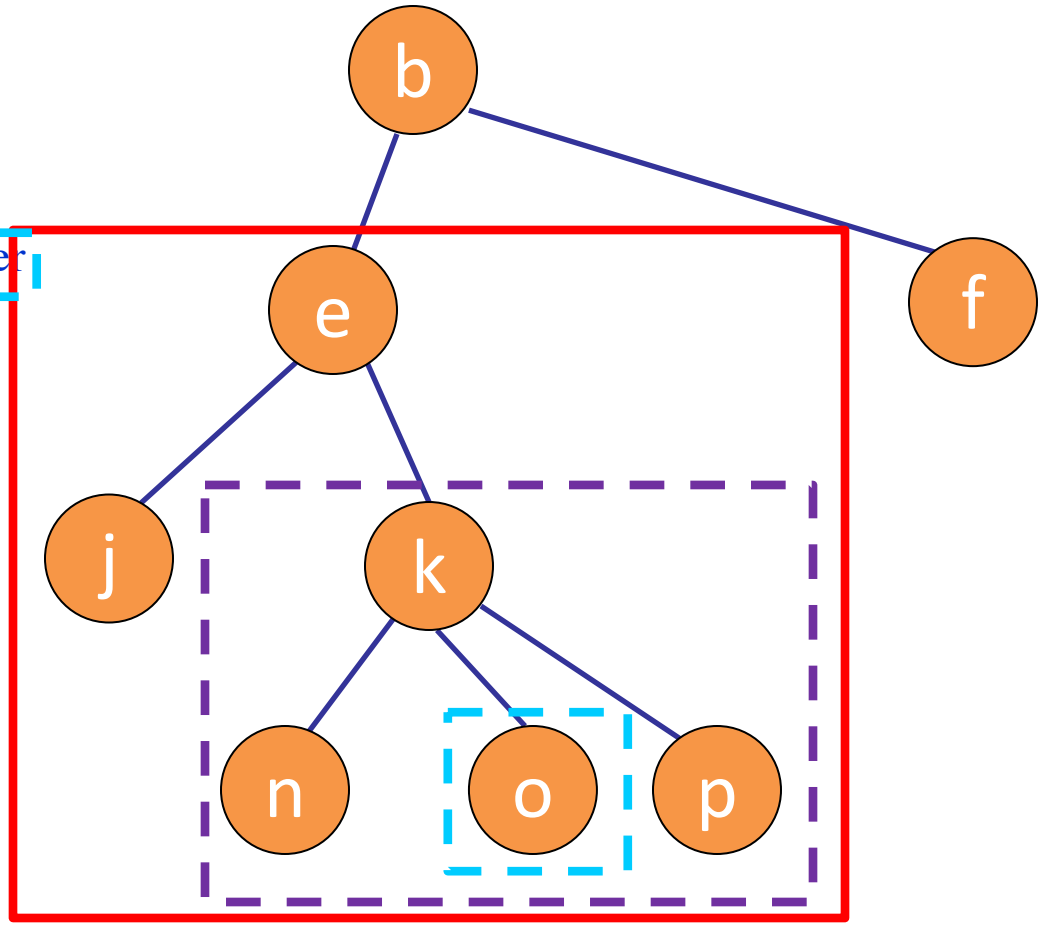
1. Visit leftmost subtree in Postorder
   1.1. Visit leftmost subtree in Postorder
   1.2. Visit remaining subtrees in Postorder
      1.2.1. Visit leftmost subtree in Postorder
      1.2.2. Visit remaining subtrees in Postorder
      1.2.3. Visit root
   1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root



j n

PostOrder

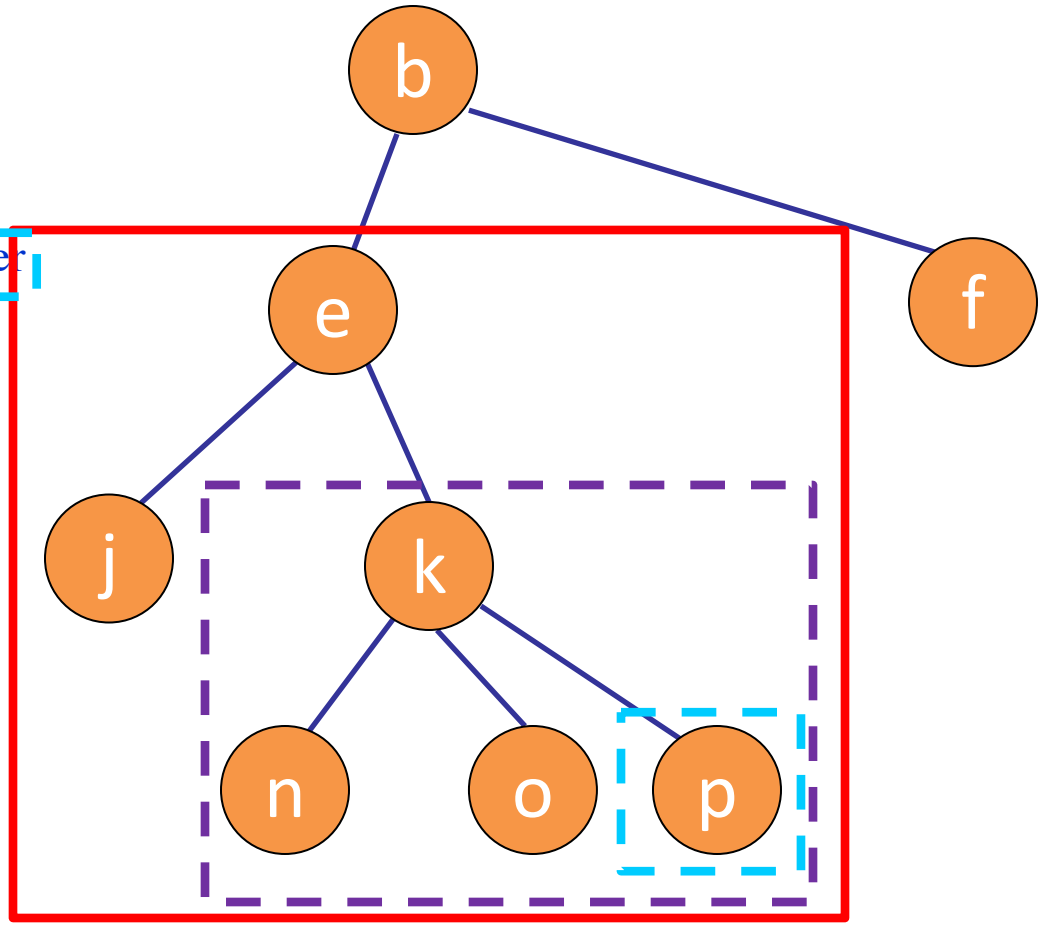
1. Visit leftmost subtree in Postorder
   1.1. Visit leftmost subtree in Postorder
   1.2. Visit remaining subtrees in Postorder
      1.2.1. Visit leftmost subtree in Postorder
      1.2.2. Visit remaining subtrees in Postorder
      1.2.3. Visit root
   1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root



j n o

PostOrder

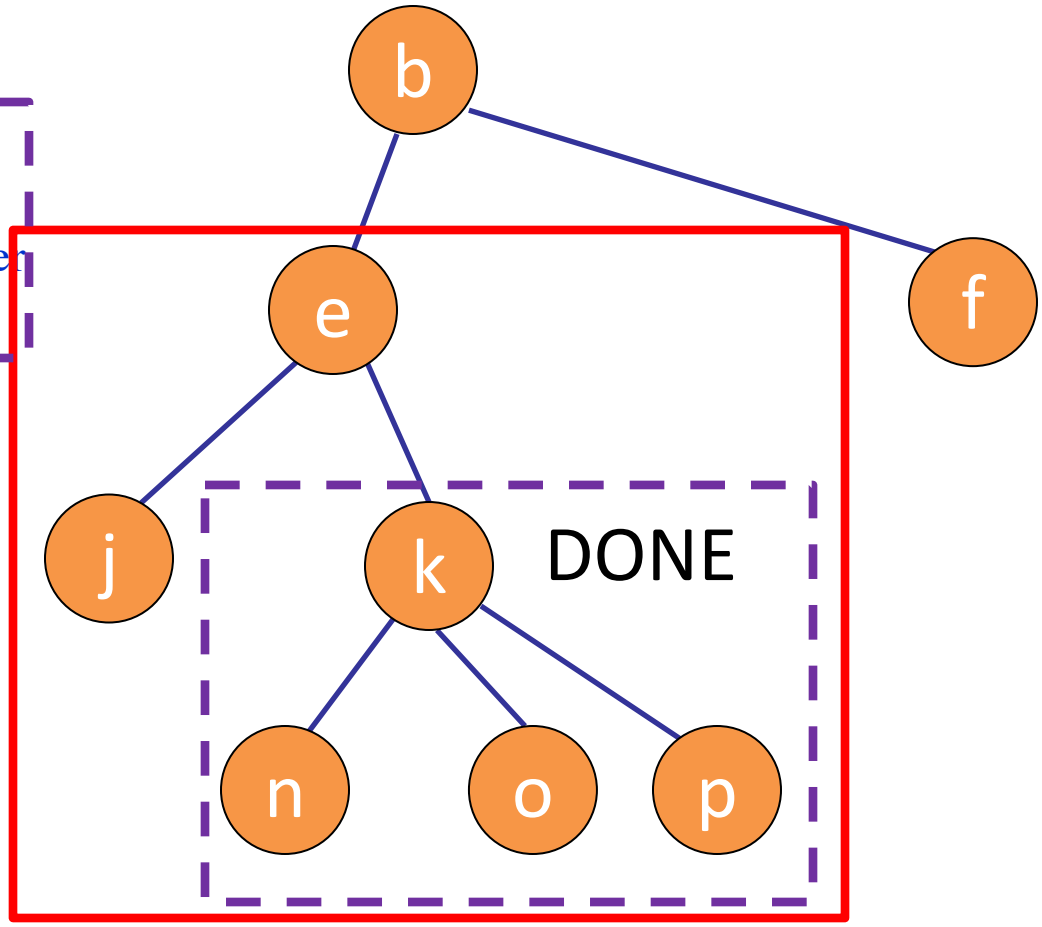
1. Visit leftmost subtree in Postorder
   1.1. Visit leftmost subtree in Postorder
   1.2. Visit remaining subtrees in Postorder
      1.2.1. Visit leftmost subtree in Postorder
      1.2.2. Visit remaining subtrees in Postorder
      1.2.3. Visit root
   1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root



j n o p

PostOrder

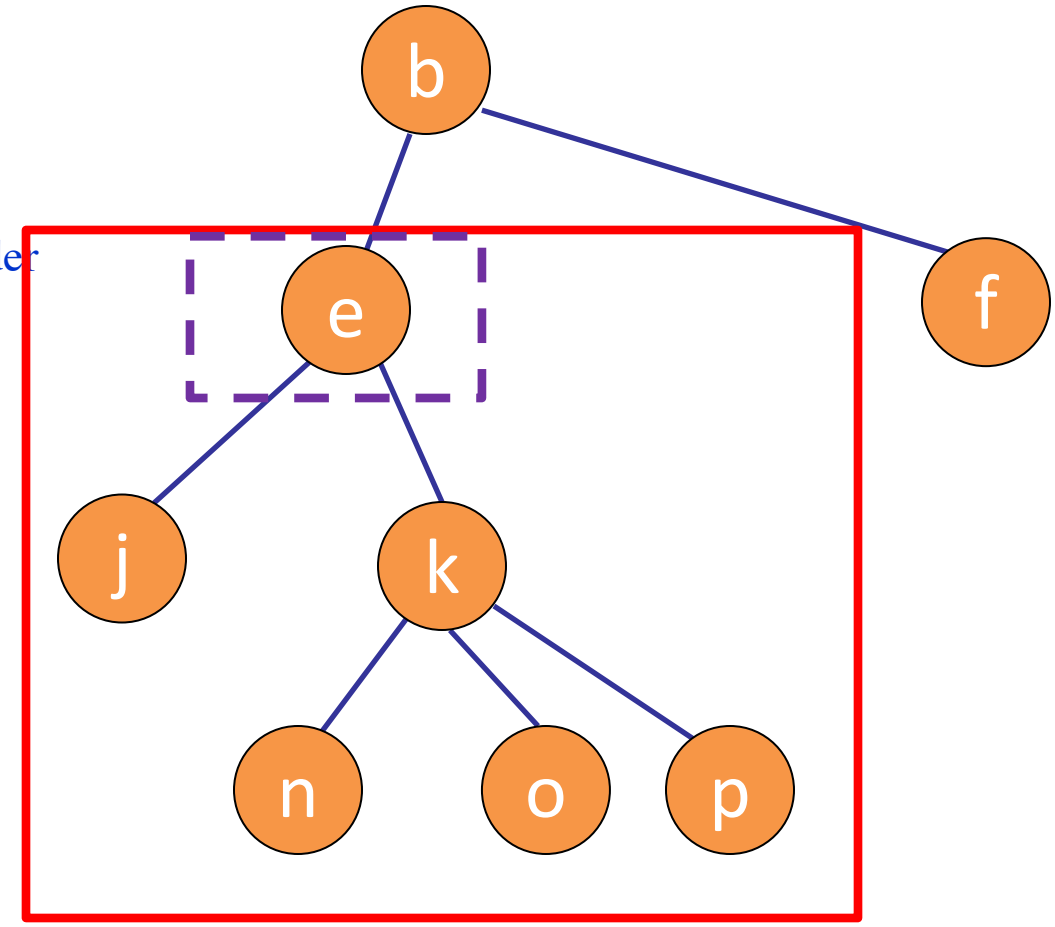
1. Visit leftmost subtree in Postorder
   1.1. Visit leftmost subtree in Postorder
   1.2. Visit remaining subtrees in Postorder
      1.2.1. Visit leftmost subtree in Postorder
      1.2.2. Visit remaining subtrees in Postorder
      1.2.3. Visit root
   1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root



j n o p k

PostOrder

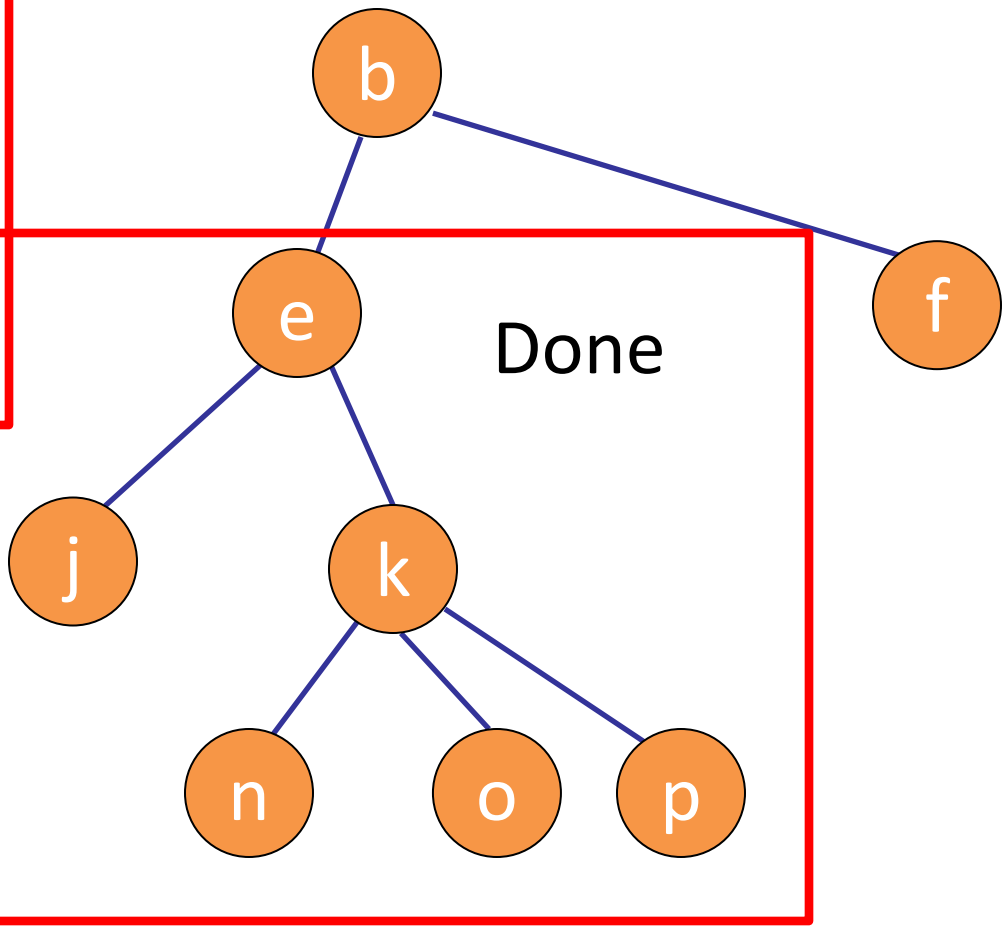
1. Visit leftmost subtree in Postorder
   1.1. Visit leftmost subtree in Postorder
   1.2. Visit remaining subtrees in Postorder
      1.2.1. Visit leftmost subtree in Postorder
      1.2.2. Visit remaining subtrees in Postorder
      1.2.3. Visit root
   1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root



j n o p k e

PostOrder

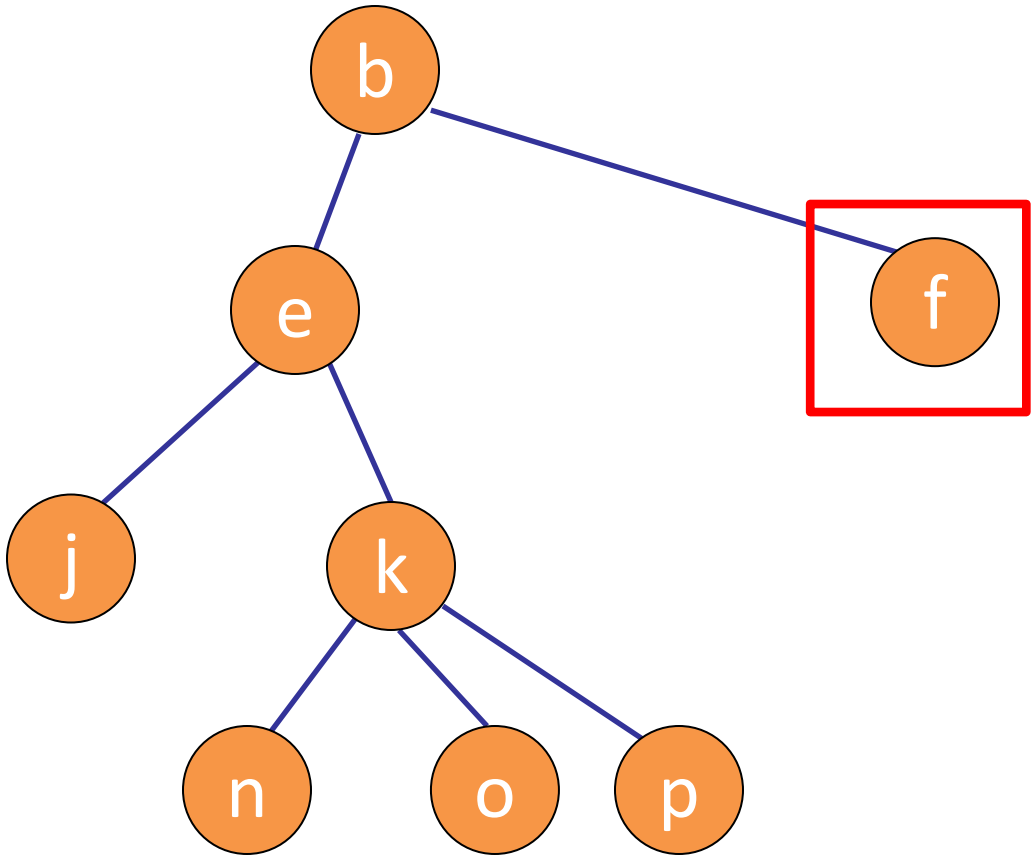
1. Visit leftmost subtree in Postorder
   1.1. Visit leftmost subtree in Postorder
   1.2. Visit remaining subtrees in Postorder
      1.2.1. Visit leftmost subtree in Postorder
      1.2.2. Visit remaining subtrees in Postorder
      1.2.3. Visit root
   1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root



j n o p k e

PostOrder

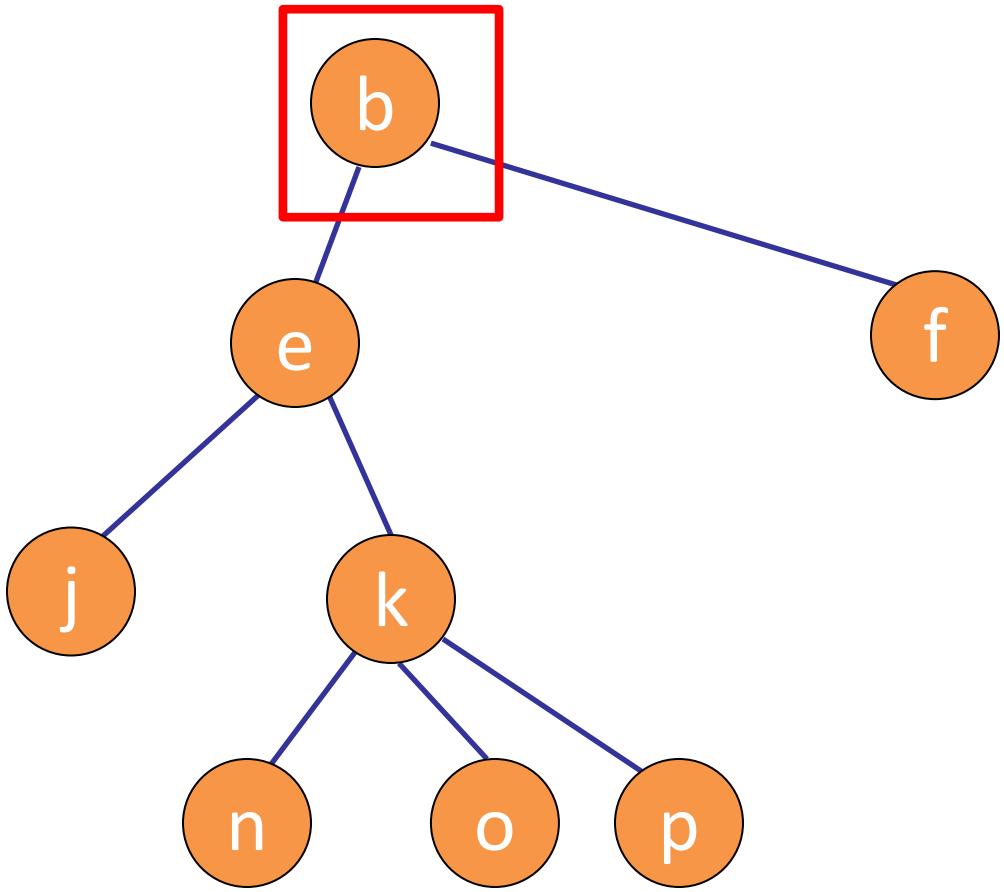
1. Visit leftmost subtree in Postorder
   1.1. Visit leftmost subtree in Postorder
   1.2. Visit remaining subtrees in Postorder
      1.2.1. Visit leftmost subtree in Postorder
      1.2.2. Visit remaining subtrees in Postorder
      1.2.3. Visit root
   1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root



j n o p k e f

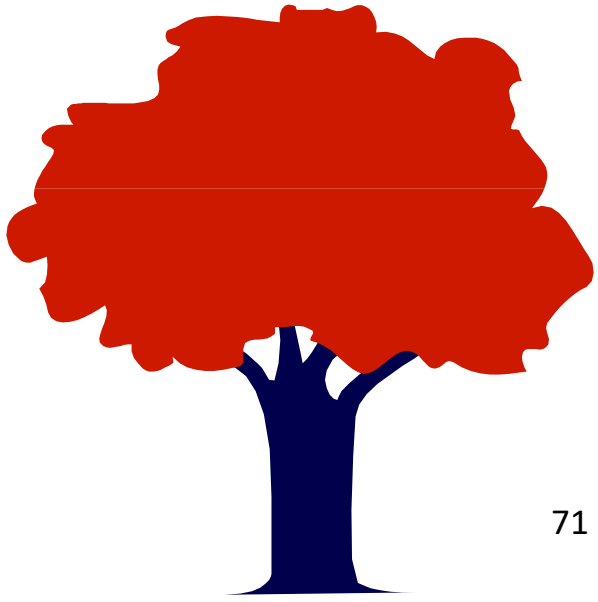
PostOrder

1. Visit leftmost subtree in Postorder
   1.1. Visit leftmost subtree in Postorder
   1.2. Visit remaining subtrees in Postorder
      1.2.1. Visit leftmost subtree in Postorder
      1.2.2. Visit remaining subtrees in Postorder
      1.2.3. Visit root
   1.3. Visit root
2. Visit remaining subtrees in Postorder
3. Visit root



j n o p k e f b

Contents



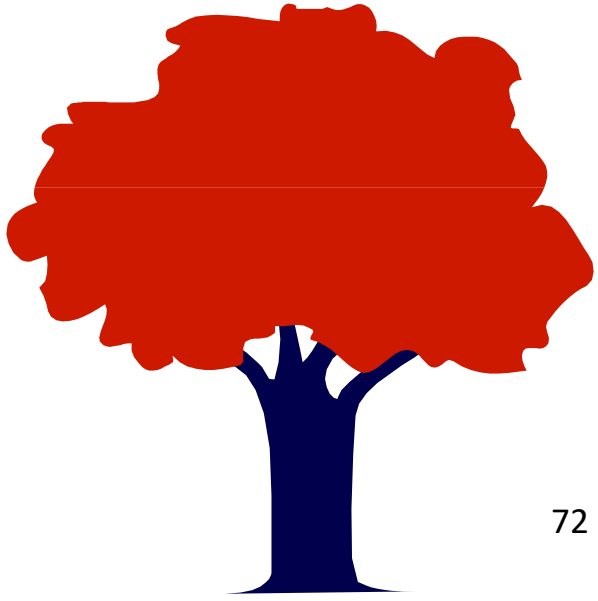
3.4. Binary tree

3.4.1. Definitions

3.4.2. Binary tree representation

3.4.3. Binary tree traversal

3.4.4. Some applications



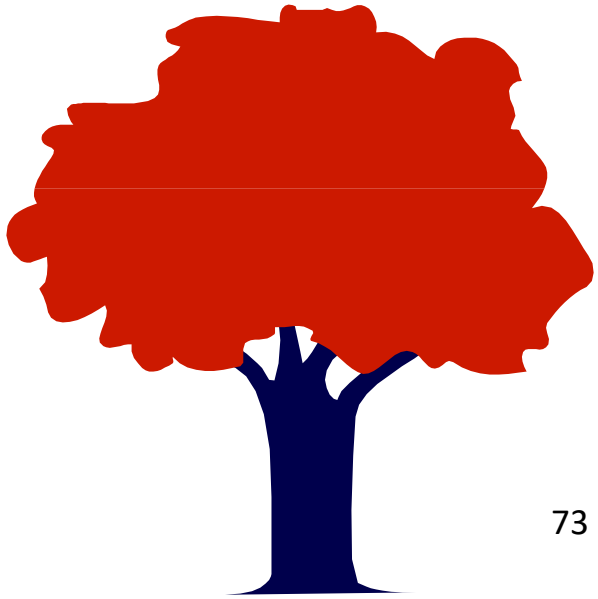
3.4. Binary tree

3.4.1. Definitions

3.4.2. Binary tree representation

3.4.3. Binary tree traversal

3.4.4. Some applications

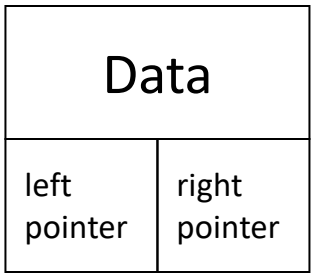


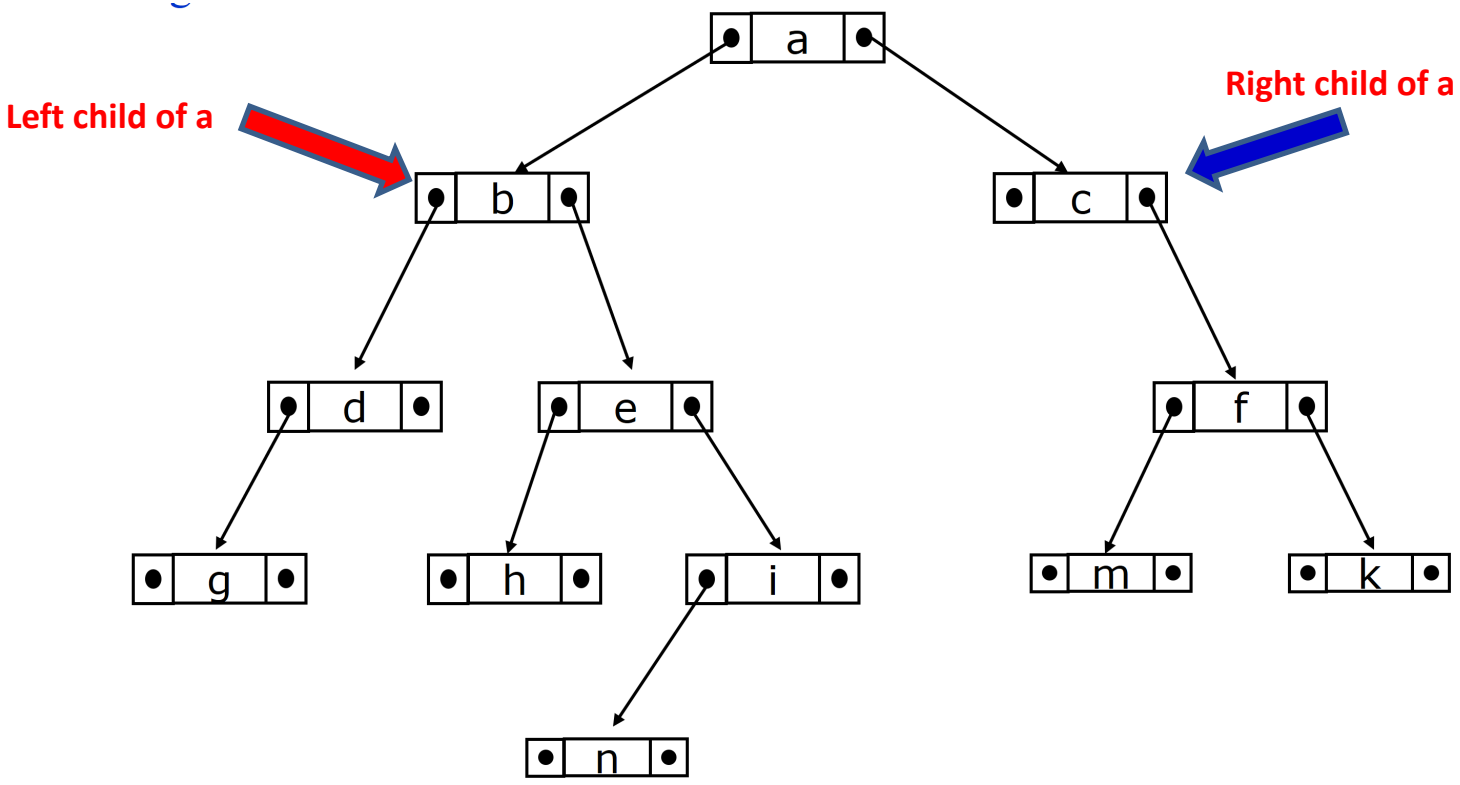
3.4.1. Definitions: Binary tree

- A binary tree is a tree in which no node can have more than two children.

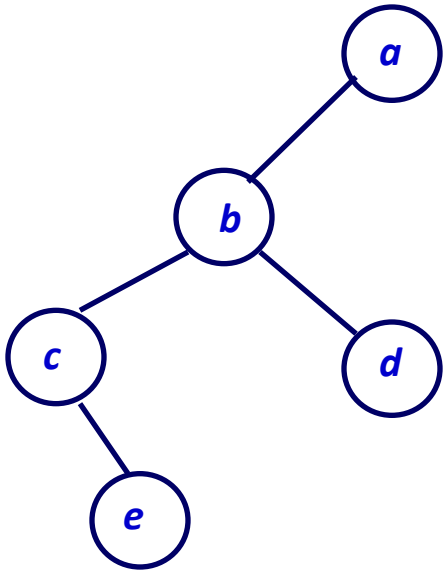
➔ Each node could either: (1) has no child, (2) has only left child, (3) has only right child, (4) has both left child and right child

- Each node has the data, a reference to a left child and a reference to a right child.

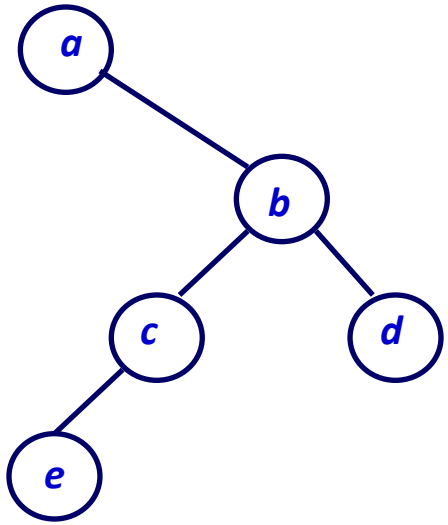




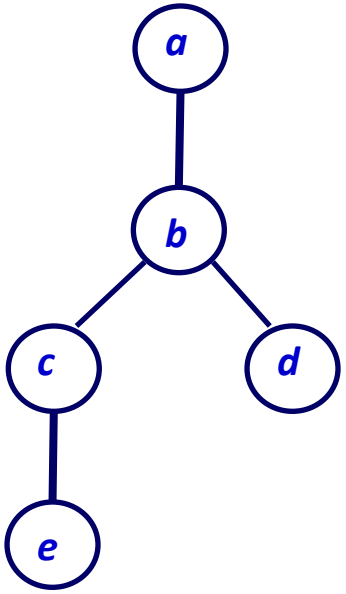
Note



Binary tree T_1



Binary tree T_2



General tree

$$T_1 \neq T_2$$

Given a binary tree of *n* nodes

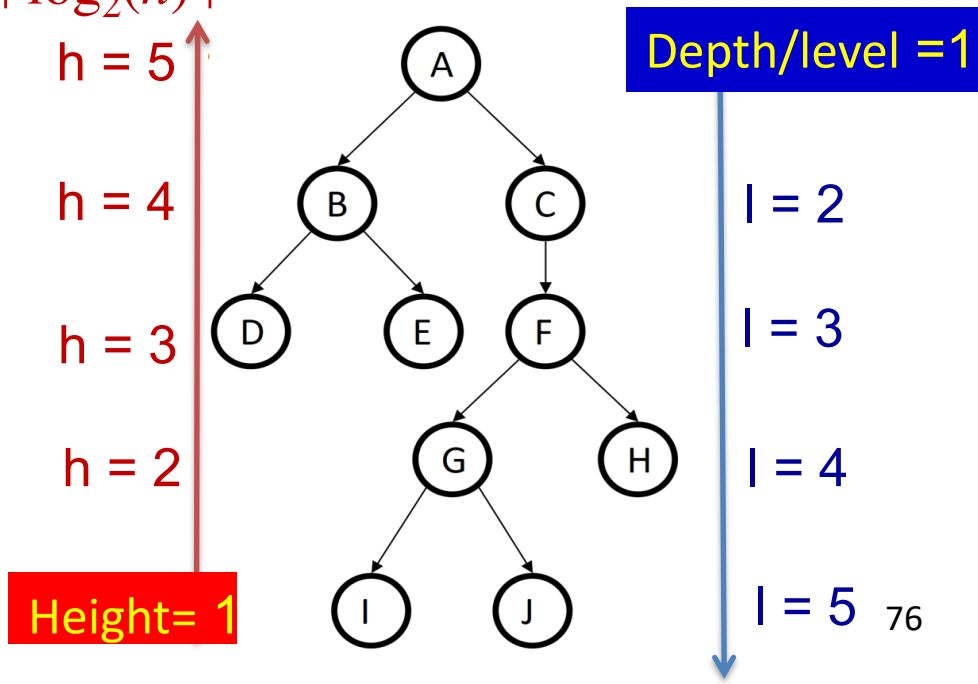
Properties:

$height(tree) = depth(tree)$

$= MAX\{depth(leaf)\} = height(root)$

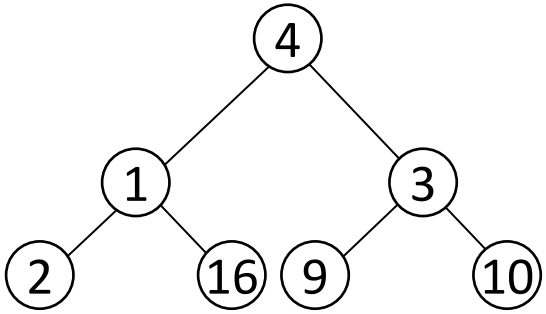
- max # of leaves = $2^{depth(tree)-1}$
- max # of nodes = $2^{depth(tree)} - 1$
- max # of nodes on level i^{th} = 2^{i-1}

Binary tree with n nodes: depth(tree) $\geq \lceil \log_2(n) \rceil$



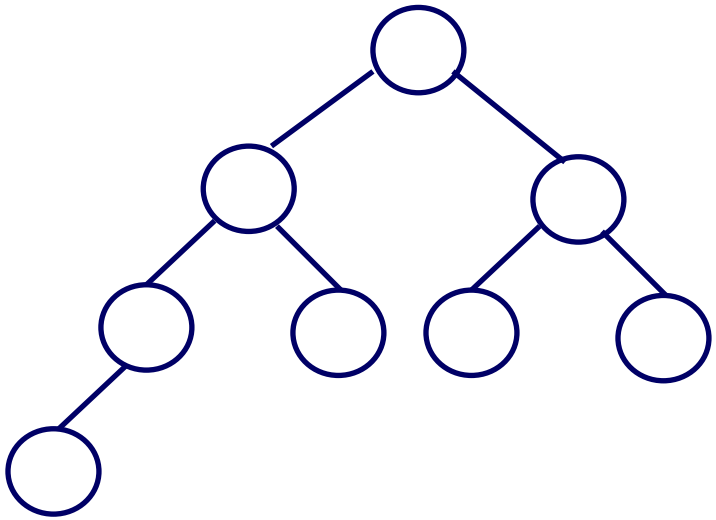
Full binary tree vs complete binary tree

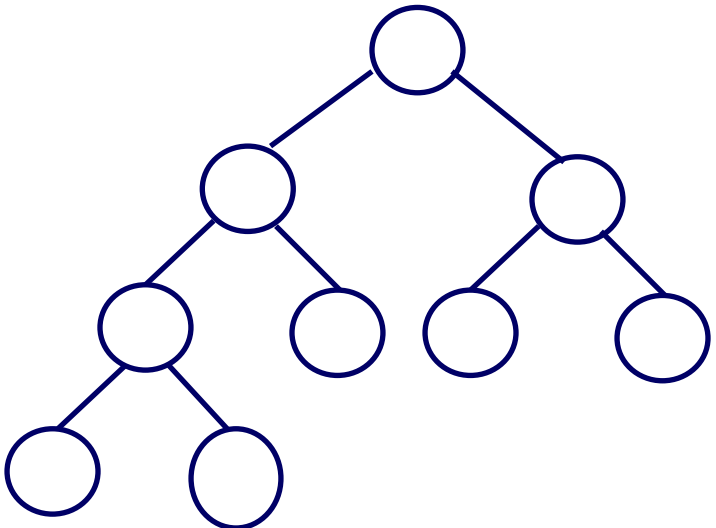
- ***Full*** binary tree: a binary tree in which
  - every parent has 2 children,
  - every leaf has equal depth



Full binary tree

- ***Complete*** binary tree: a binary tree in which
  - every level is full except possibly the deepest level
  - if the deepest level isn't full, leaf nodes are as far to the left as possible

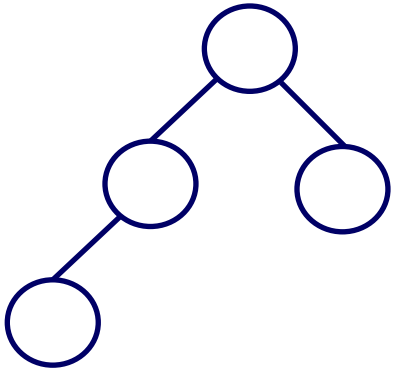




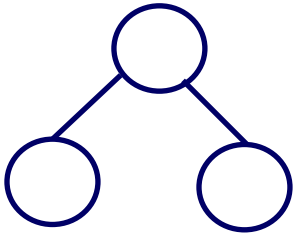
Which tree is full binary tree / complete binary tree?

Full binary tree
- every parent has 2 children,
- every leaf has equal depth

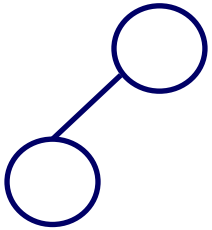
Complete binary tree
- every level is full except possibly the deepest level
- if the deepest level isn't full, leaf nodes are as far to the left as possible



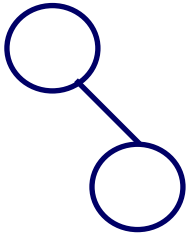
Complete binary tree



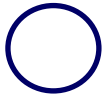
Full and complete



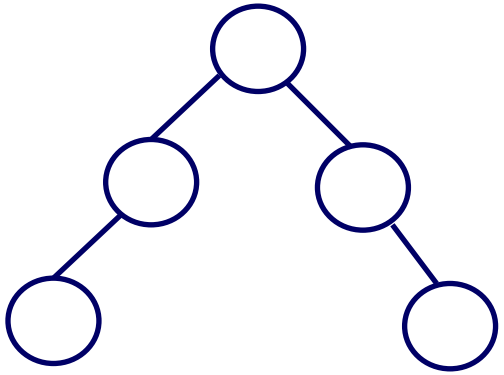
Complete



Neither full nor complete



Full and complete



Neither full nor complete

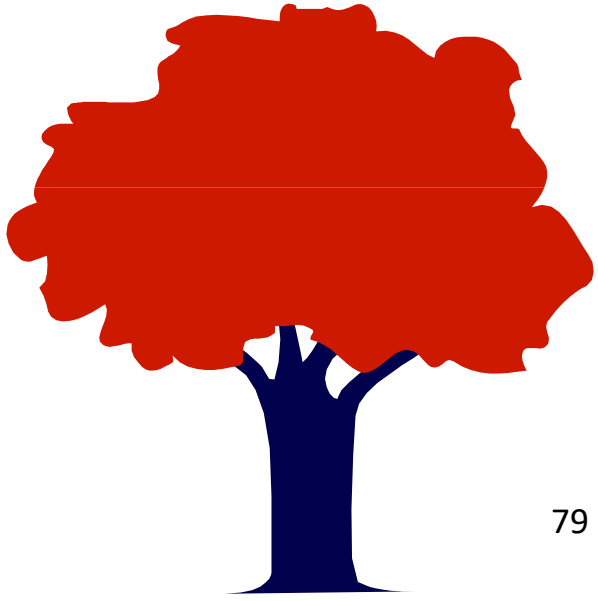
3.4. Binary tree

3.4.1. Definitions

3.4.2. Binary tree representation

3.4.3. Binary tree traversal

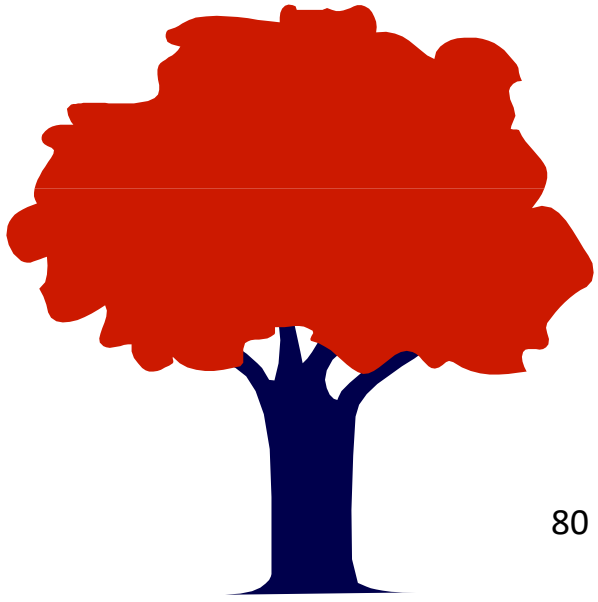
3.4.4. Some applications



3.4.2. Binary tree representation

3.4.2.1. Array

3.4.2.2. Pointer

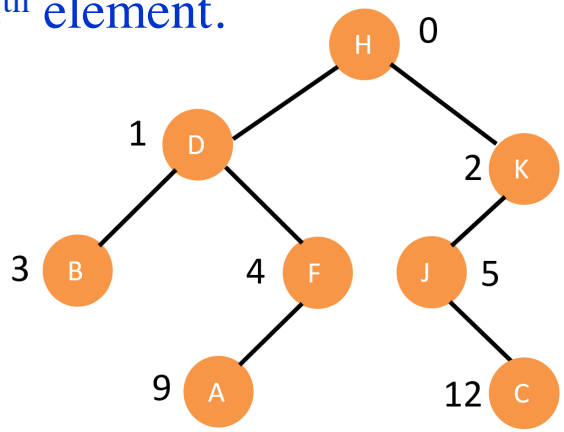


1D array representation of a binary tree

Nodes are numbered / indexed as following:

- giving 0 to root,
- then all the nodes are numbered from left to right level by level from top to bottom. Empty nodes are also numbered.

Then each node having an index i is put into the array as its i^{th} element.



➔ An 1D array ***A*** can be used to represent a binary tree:

- Root is $A[0]$
- for any node at $A[i]$:
  - its parent node is $A[(i\text{-}1)/2]$
  - its left child $A[2i + 1]$ and its right child is $A[2i + 2]$
  - If any node does not have any of its child, then null value is stored at the corresponding index of the array.

A

H	D	K	B	F	J	NULL	NULL	NULL	A	NULL	NULL	C	NULL
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	.....	25	26	.....	30

Right child of K

Left child of B

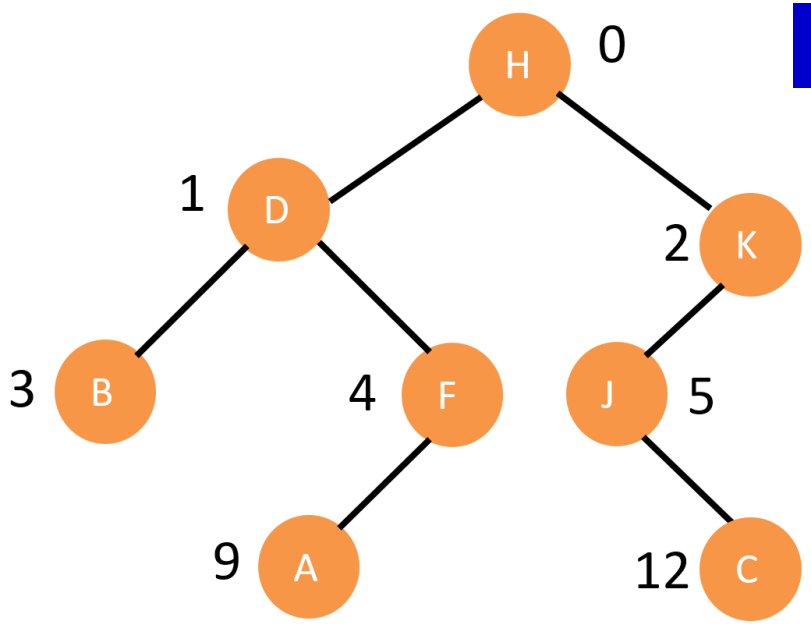
Right child of B

left child of J

Left child of C

Right child of C

1D array representation of a binary tree



Depth/level =1 — Max #nodes = 2^0=1

l = 2 — Max #nodes = 2^1=2

l = 3 — Max #nodes = 2^2=4

l = 4 — Max #nodes = 2^3=8

Max #nodes = 2^4=16

$\sum max\#nodes = 31$

for any node at $A[i]$:

- its parent node is $A[(i\text{-}1)/2]$
- its left child $A[2i + 1]$ and its right child is $A[2i + 2]$

	H	D	K	B	F	J	NULL	NULL	NULL	A	NULL	NULL	C	NULL
A	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	.....	25	26	.....	30

Right child of K

Left child of B

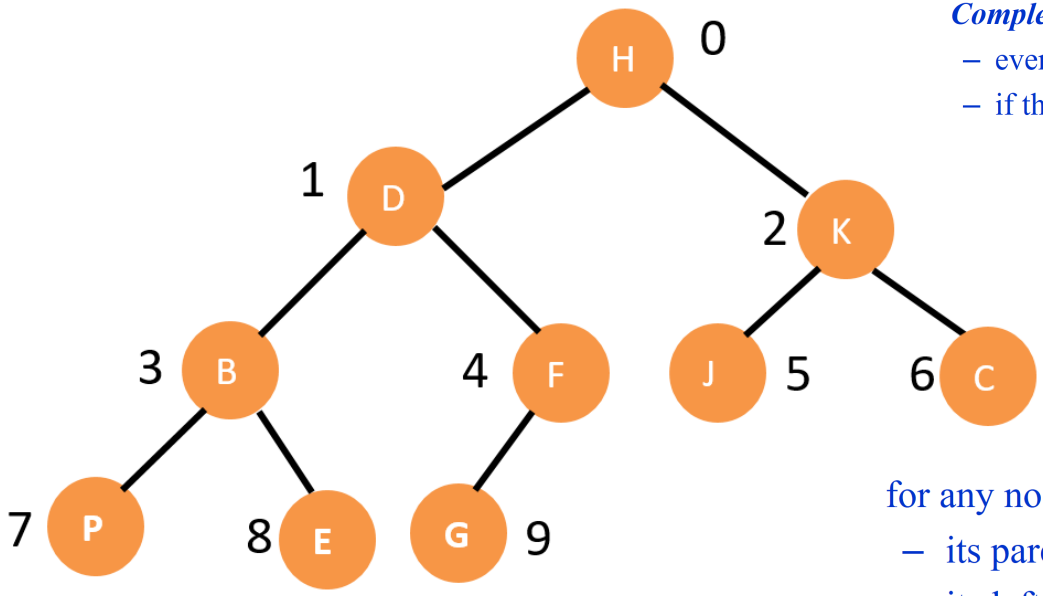
Right child of B

left child of J

Left child of C

Right child of C

1D array representation of a complete binary tree



Complete binary tree

- every level is full except possibly the deepest level
- if the deepest level isn't full, leaf nodes are as far to the left as possible

for any node at $A[i]$:

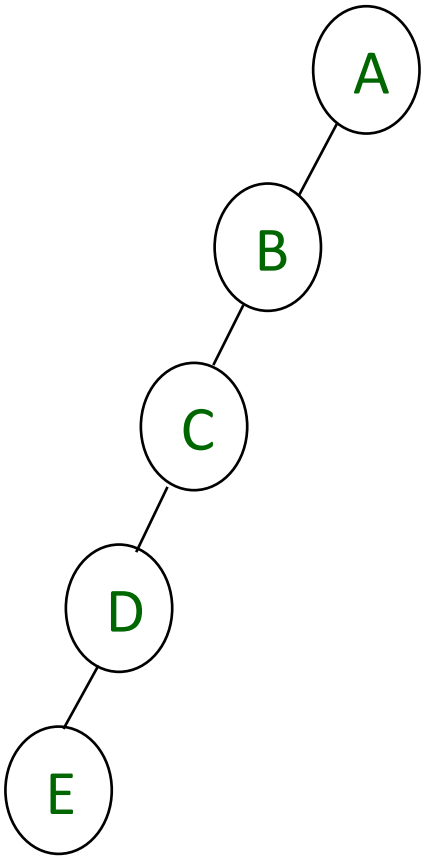
- its parent node is $A[(i\text{-}1)/2]$
- its left child $A[2i + 1]$ and its right child is $A[2i + 2]$

H	D	K	B	F	J	C	P	E	G
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

If using 1D array to represent a complete binary tree of n nodes:

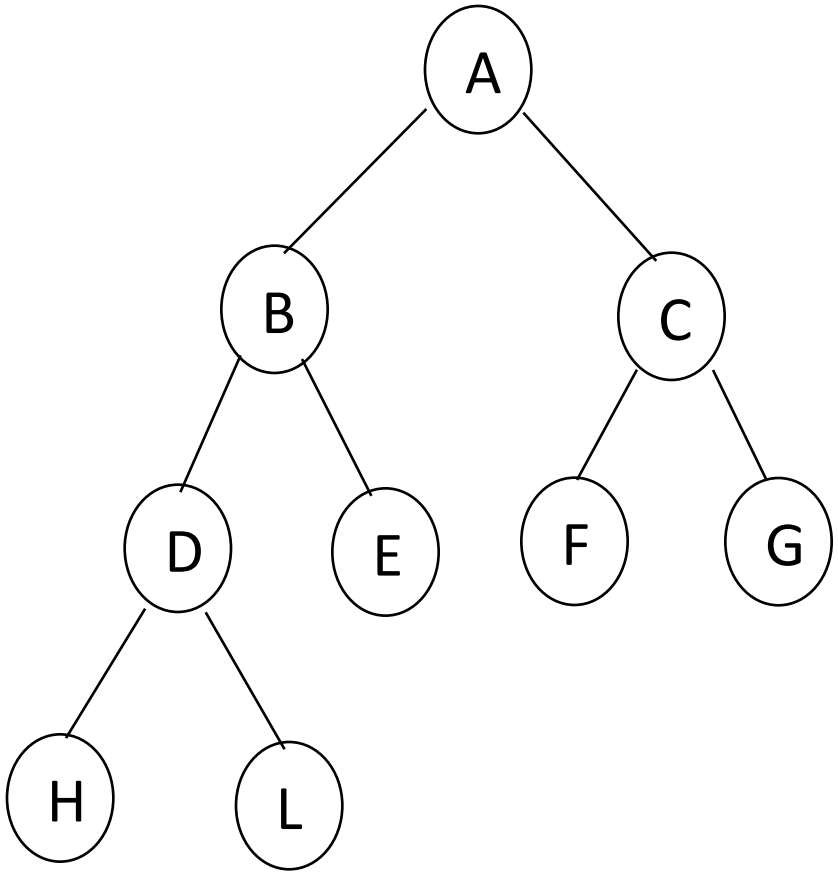
- We only need the array of n elements
- To check a node $A[i]$ is leaf or not ?

1D array representation of a binary tree



[0]	A
[1]	B
[2]	NULL
[3]	C
[4]	NULL
[5]	NULL
[6]	NULL
[7]	D
	NULL
[15]	E

(1) waste space
(2) insertion/deletion problem

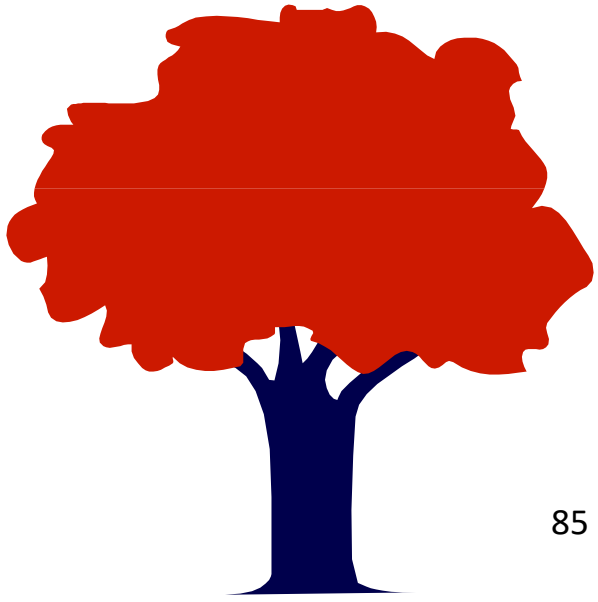


[0]	A
[1]	B
[2]	C
[3]	D
[4]	E
[5]	F
[6]	G
[7]	H
[8]	L

3.4.2. Binary tree representation

3.4.2.1. Array

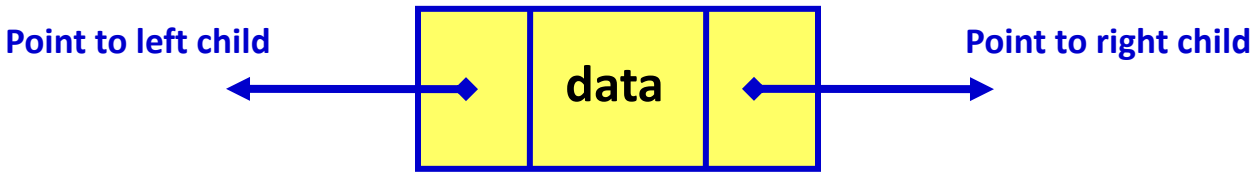
3.4.2.2. Pointer



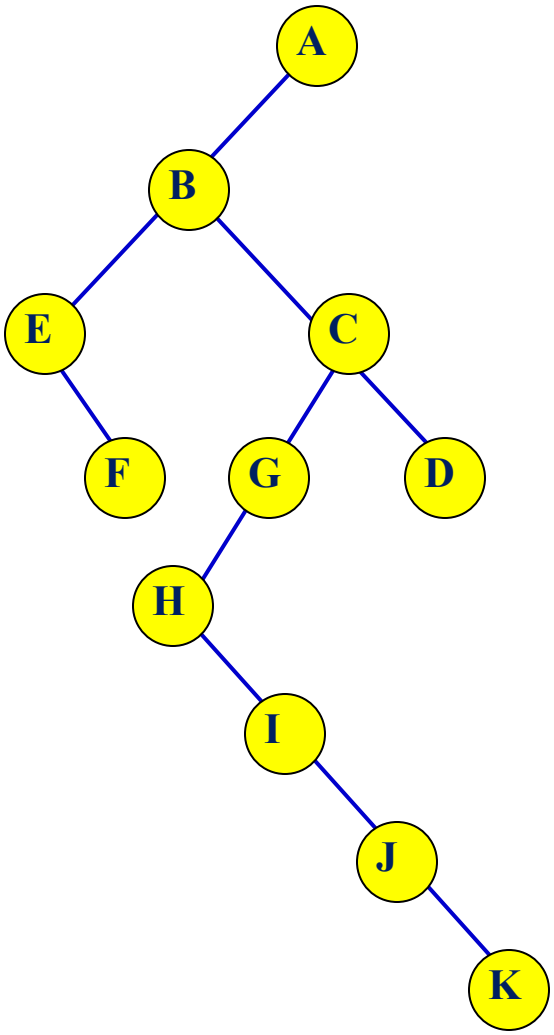
Pointer representation of a binary tree

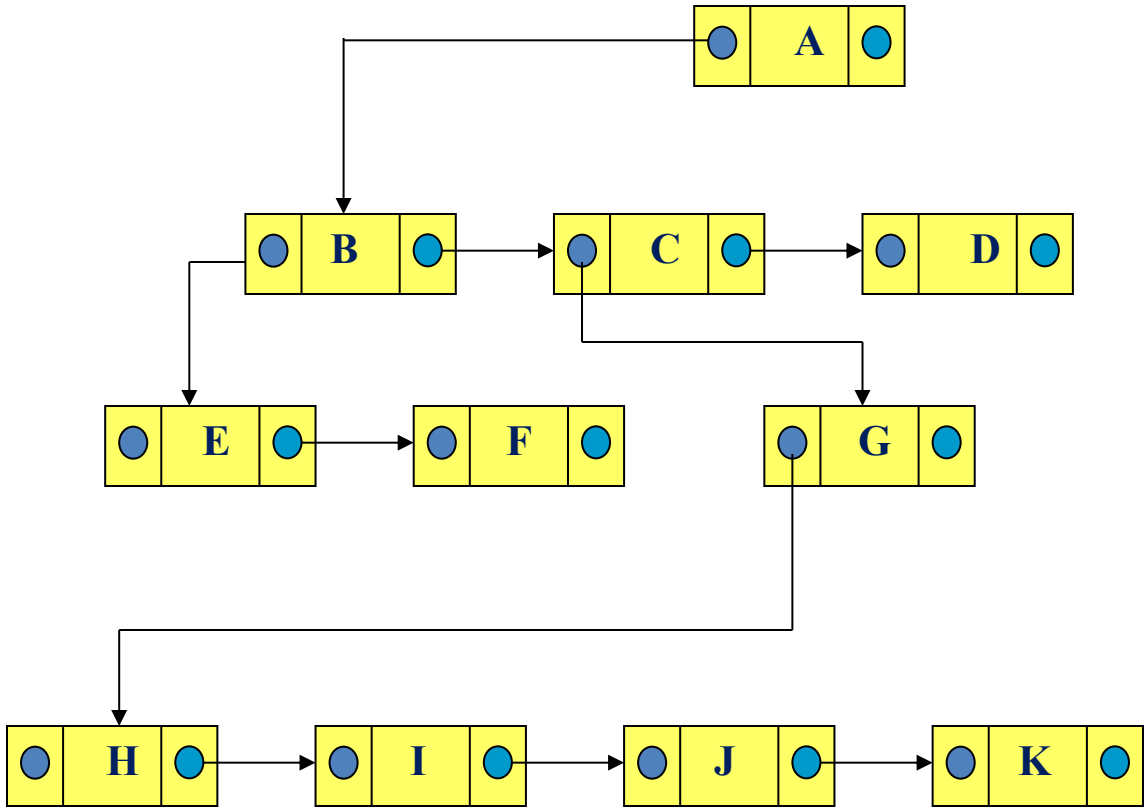
- Each node contains the address of the left child and the right child.
- If any node has its left or right child empty then it will have in its respective link field, a null value.
- A leaf node has null value in both of its links.
- The structure defining a node of binary tree in C is as follows:

```
typedef struct
{
   DataType data; /*data of node; DataType: int, char, double..*/
   struct node *left ;   /* points to the left child */
   struct node *right;   /* points to the right child */
}node;
```



Pointer representation of a binary tree: Example





Basic operations on binary tree

```
typedef struct
{
    char word[20];  // Data of node
     struct node * left;
     struct node *right;
}node;


node* makeTreeNode(char *word);
node *insertNode(node* root, char *word, bool LEFT);
int countNodes(node *root);
int depth(node *root);
void freeTree(node *root);
void printPreorder(node *root);
void printPostorder(node *root);
void printInorder(node *root);
```

Basic operations on binary tree: MakeNode

```
node* makeTreeNode(char *word);
```

- Input parameter: data of the inserted node
- Output: Return the pointer to (address of) the new node

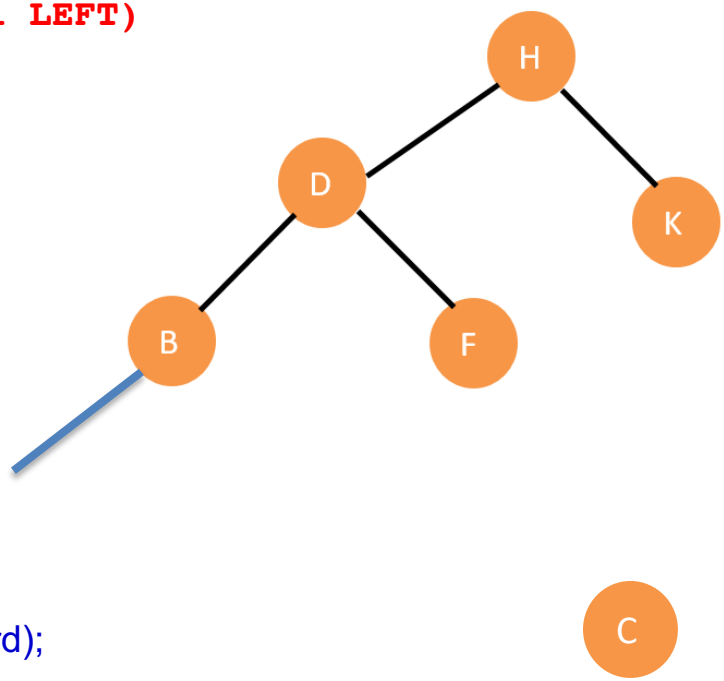
Steps:

- Allocate memory for new node. Check whether the allocation is successful?
  - If Yes: assign data of new node = word
    pointer of left child = NULL
    pointer to right child = NULL
- Return the pointer to (address of) the new node

```
node* makeTreeNode(char *word)
{
   node* newNode = NULL;
   newNode = (node*)malloc(sizeof(node));
   if (newNode == NULL){ //Memory allocation is unsucessful
      printf("Out of memory\n");
      exit(1);
   }
   else {
      strcpy(newNode->word,word);
      newNode->left = NULL;
      newNode->right= NULL;
   }
   return newNode;
}
```

Basic operations on binary tree: insert new node into the tree

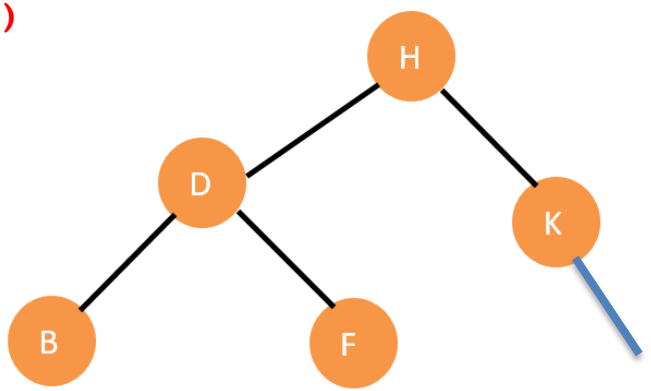
```
node *insertNode(node *tree, char *word, bool LEFT)
{
    node *newNode, *p;
    newNode = makeTreeNode(word);
    if ( tree == NULL ) return newNode;
    if (LEFT) //insert node on the leftmost of the tree
    {
        p = tree;
        while (p->left !=NULL) p = p->left;
        p->left = newNode;
        printf("Node %s  is left child of  %s \n",word,(*p).word);
    }
    else  { //insert node on the rightmost of the tree
        p = tree;
        while (p->right !=NULL) p = p->right;
        p->right = newNode;
        printf("Node %s  is right child of  %s \n", word,(*p).word);
    }
    return tree;
}
```



Call: `RandomInsert(H, "C", 1);`

Basic operations on binary tree: insert new node into the tree

```
node *insertNode(node *tree, char *word, bool LEFT)
{
    node *newNode, *p;
    newNode = makeTreeNode(word);
    if ( tree == NULL ) return newNode;
    if (LEFT) //insert node on the leftmost of the tree
    {
        p = tree;
        while (p->left !=NULL) p = p->left;
        p->left = newNode;
        printf("Node %s  is left child of  %s \n",word,(*p).word);
    }
    else  { //insert node on the rightmost of the tree
        p = tree;
        while (p->right !=NULL) p = p->right;
        p->right = newNode;
        printf("Node %s  is right child of  %s \n", word,(*p).word);
    }
    return tree;
}
```

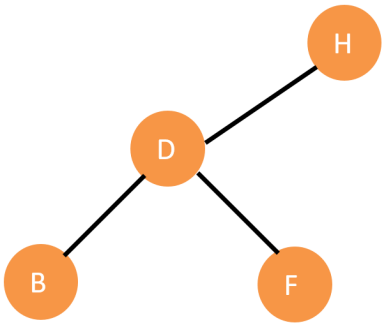


C

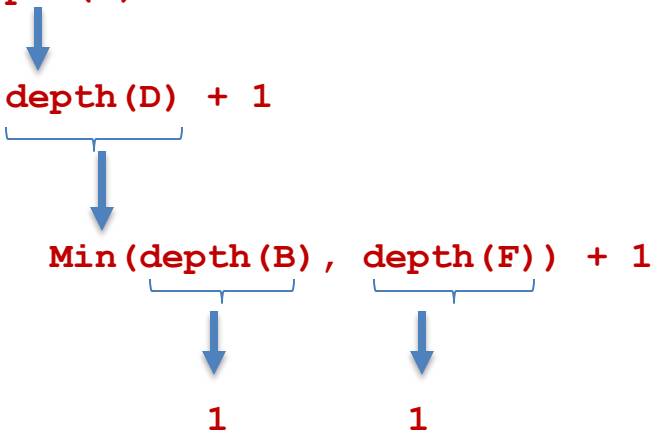
Call: `RandomInsert(H, "C", 0);`

Basic operations on binary tree: calculate depth of tree

```
int depth(node *root)  {  /* the function computes the depth of a tree */
    if( root == NULL ) return 0;
     // Base case : Leaf Node. This accounts for height = 1.
     if (root->left == NULL && root->right == NULL) return 1;

     // If left subtree is NULL, recur for right subtree
     if (root->left==NULL) return depth(root->right) + 1;

     // If right subtree is NULL, recur for left subtree
     if (root->right==NULL) return depth(root->left) + 1;

     return min(depth(root->left), depth(root->right)) + 1;
}
```

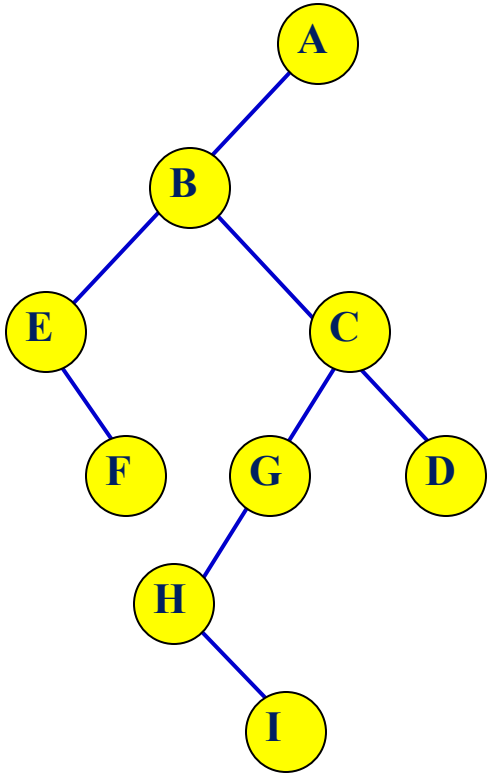


Call depth(H):



Basic operations on binary tree: count number of nodes on the tree

```
int countNodes(node *root) {
/* the function counts the number of nodes of a tree*/
   if( root == NULL ) return 0;
   else {
      int ld = countNodes(root->left);
      int rd = countNodes(root->right);
      return 1+ld+rd;
   }
}
```



#nodes on the tree = 1 + #nodes on left subtree + #nodes on right subtree

Call countNodes(A);

Basic operations on binary tree: free the tree

```
void freeTree(node *root)
{
     if( root == NULL ) return;
     freeTree(root->left);
     freeTree(root->right);
     free(root);
}
```

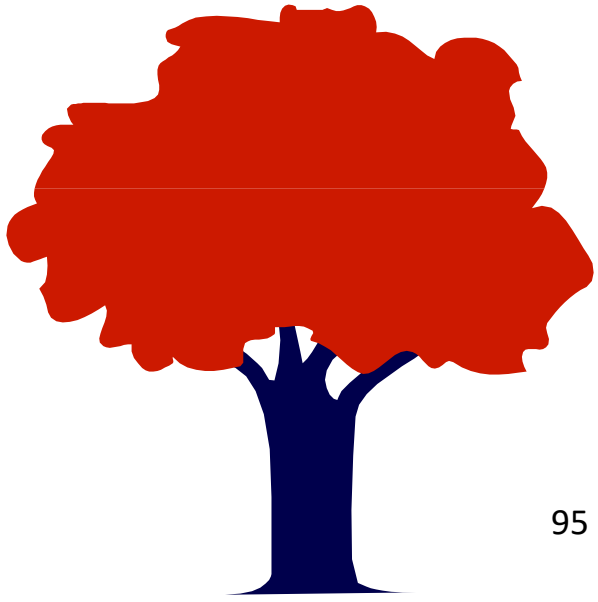
3.4. Binary tree

3.4.1. Definitions

3.4.2. Binary tree representation

3.4.3. Binary tree traversals

3.4.4. Some applications

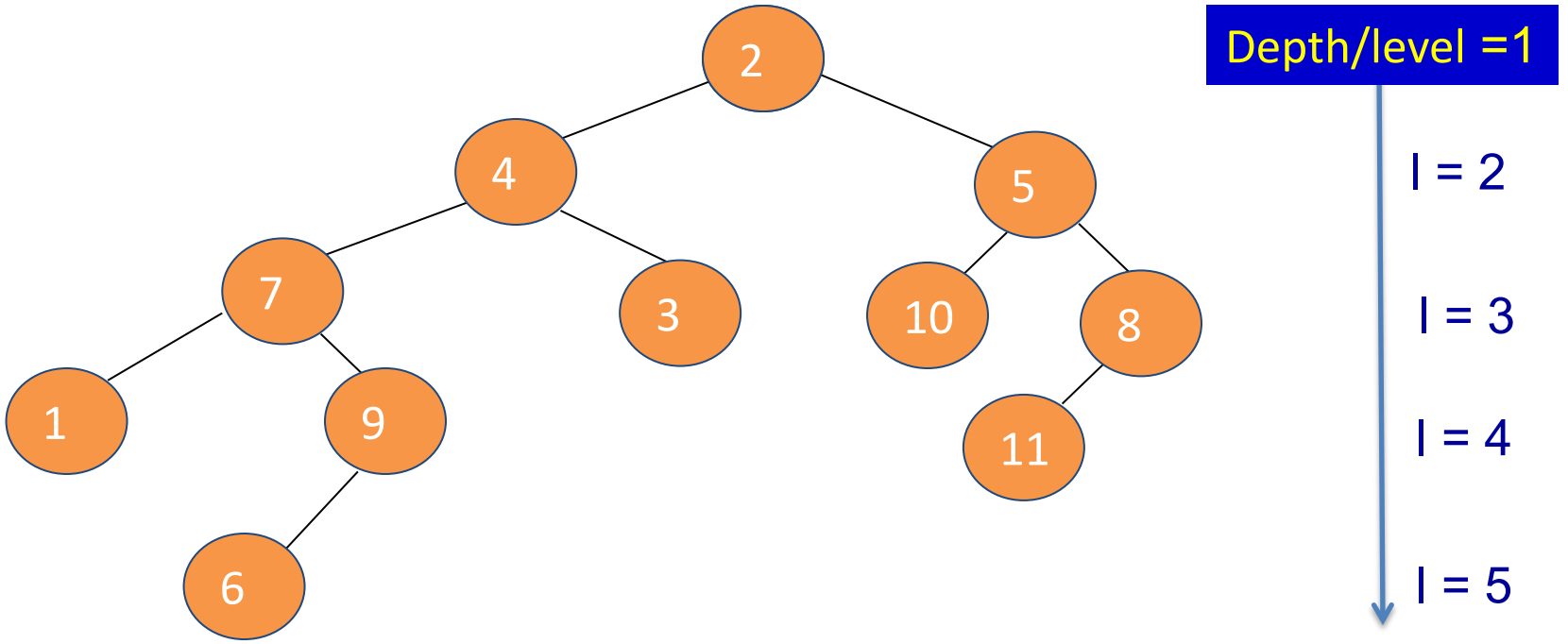


3.4.3. Binary Tree Traversals

- A traversal is where each node in a tree is visited and visited once
- There are two very common traversals
  - Breadth First
  - Depth First

3.4.3. Binary Tree Traversals: Breadth First

- In a breadth first traversal all of the nodes on a given level are visited and then all of the nodes on the next level are visited.
- Usually in a left to right fashion



- Breadth_First_Search(2): 2, 4, 5, 7, 3, 10, 8, 1, 9, 11, 6

3.4.3. Binary Tree Traversals: Depth First

- In a depth first traversal all the nodes on a branch are visited before any others are visited
- There are three common depth first traversals
  - Inorder
  - Preorder
  - Postorder
- Each type has its use and specific application

3.4.3. Binary Tree Traversals: Depth First

- ***Preorder* NLR**
  - Visit a **n**ode,
  - Visit **l**eft subtree in preorder,
  - Visit **r**ight subtree in preorder
- ***Inorder* LNR**
  - Visit **l**eft subtree in inorder,
  - Visit a **n**ode,
  - Visit **r**ight subtree in inorder
- ***Postorder* LRN**
  - Visit **l**eft subtree in postorder,
  - Visit **r**ight subtree in postorder,
  - Visit a **n**ode

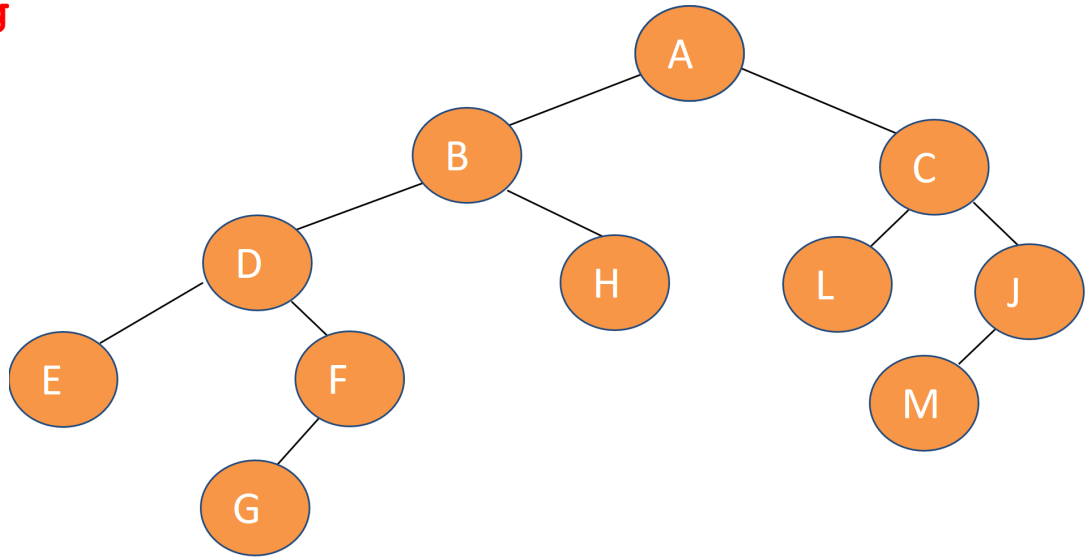
Preorder traversal - NLR

- Visit the node.
- Traverse the left subtree.
- Traverse the right subtree.

```
void printPreorder(node *root)
{    if( root != NULL )
     {
       printf("%s  ", root->word);
       printPreorder(root->left);
       printPreorder(root->rig
     }
}
```

```
Call: printPreorder(A);
A B D E F G H C L J M
```

left subtree right subtree



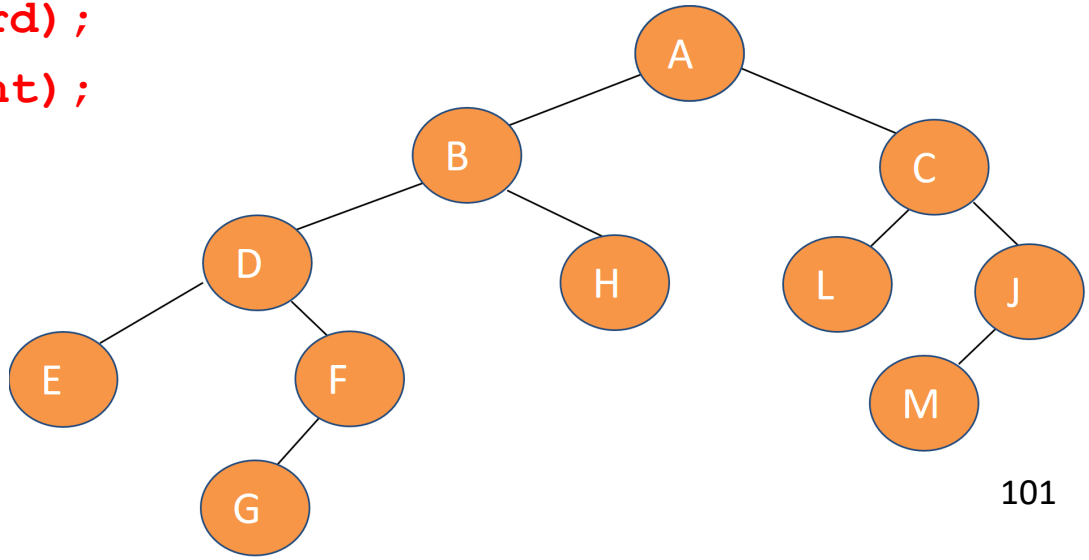
Inorder traversal - LNR

- Traverse the left subtree
- Visit the node
- Traverse the right subtree

```
void printInorder(node *tree)
{
   if( tree != NULL )
     {
       printInorder(tree->left);
       printf("%s ", tree->word);
       printInorder(tree->right);
   }
}
```

```
Call: printInorder(A);
E D G F B H A L C M J
```

left subtree right subtree



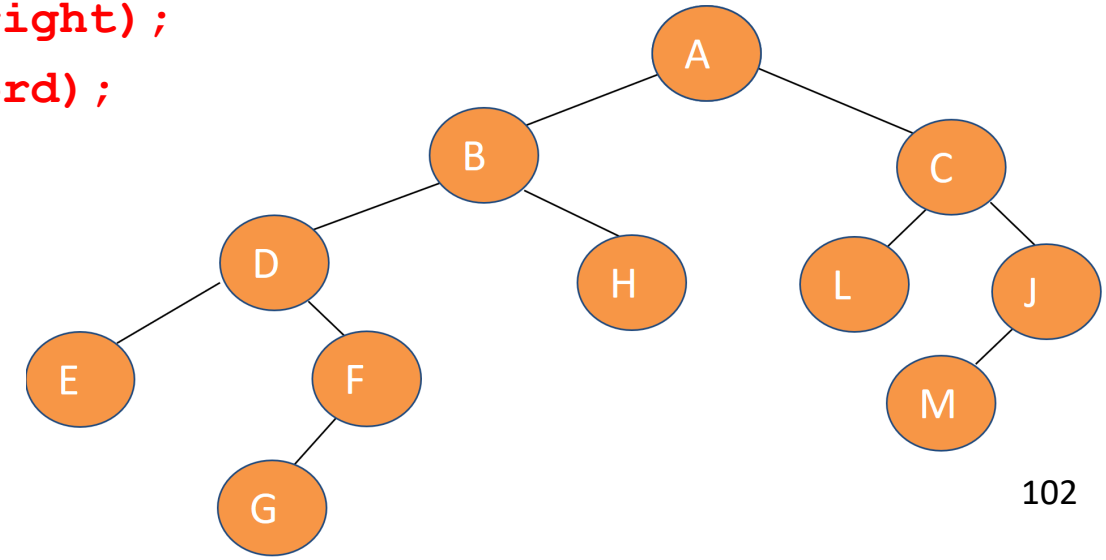
Postorder traversal - LRN

- Traverse the left subtree
- Traverse the right subtree
- Visit the node

```
void printPostorder(node *tree)
{
   if( tree != NULL )
     {
       printPostorder(tree->left);
       printPostorder(tree->right);
       printf("%s ", tree->word);
   }
}
```

```
Call: printPostorder(A);
E G F D H B L M J C A
```

left subtree right subtree



Program in C

```
/* The program for testing binary tree traversal */
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>
typedef struct {
   char word[20];
   struct node * left;
   struct node *right;
} node;
node *makeTreeNode(char *word);
node *insertNode(node* tree,char *word);
void freeTree(node *tree);
void printPreorder(node *tree);
void printPostorder(node *tree);
void printInorder(node *tree);
int countNodes(node *tree);
int depth(node *tree);
```

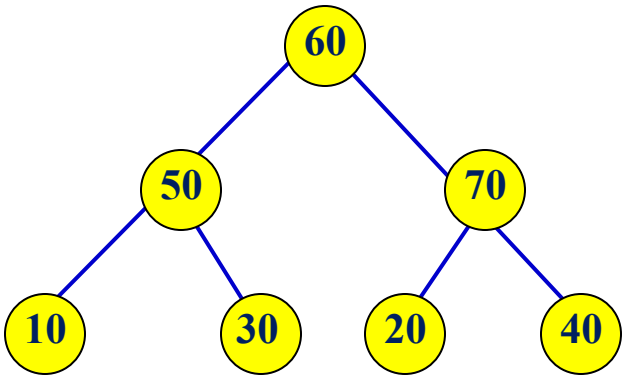
Program in C

```
void main() {
    node *randomTree=NULL;
    char word[20] = "a";
     while( strcmp(word,"~")!=0)
    {   printf("\nEnter item (~ to finish): ");
        scanf("%s", word);
        if (strcmp(word,"~")==0) printf("Finish to input data for node... \n");
        else randomTree  = insertNode(randomTree,word, rand()%2);
    }
    printf("The tree in preorder:\n"); printPreorder(randomTree);
    printf("The tree in postorder:\n"); printPostorder(randomTree);
    printf("The tree in inorder:\n"); printInorder(randomTree);
    printf("The number of nodes is: %d\n",countNodes(randomTree));
    printf("The depth of the tree is: %d\n", depth(randomTree));
    freeTree(randomTree);
}
```

Exercise

- Draw a binary tree of height 3, whose postorder traversal is (10, 30, 50, 20, 40, 70, 60)

Postorder: Left_Right_Root



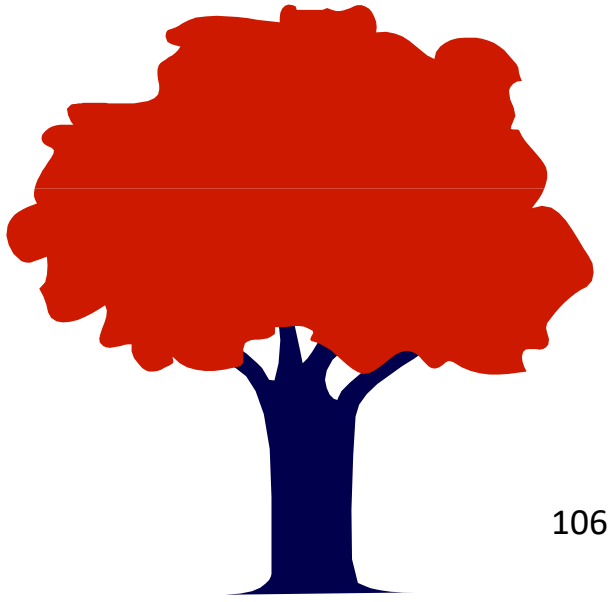
3.4. Binary tree

3.4.1. Definitions

3.4.2. Binary tree representation

3.4.3. Binary tree traversals

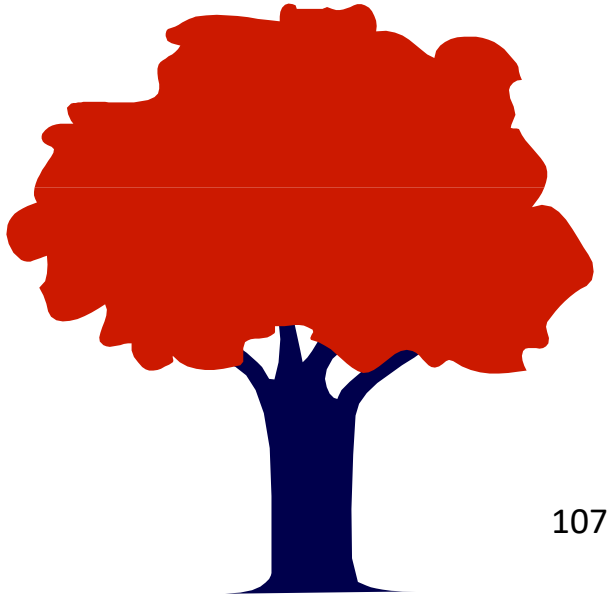
3.4.4. Some applications



3.4.4. Some applications

3.4.4.1. Arithmetic expression

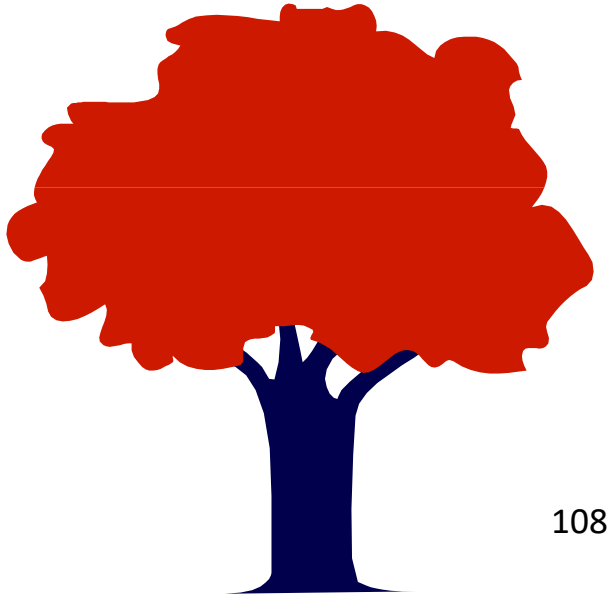
3.4.4.2. Huffman code



3.4.4. Some applications

3.4.4.1. Arithmetic expression

3.4.4.2. Huffman code

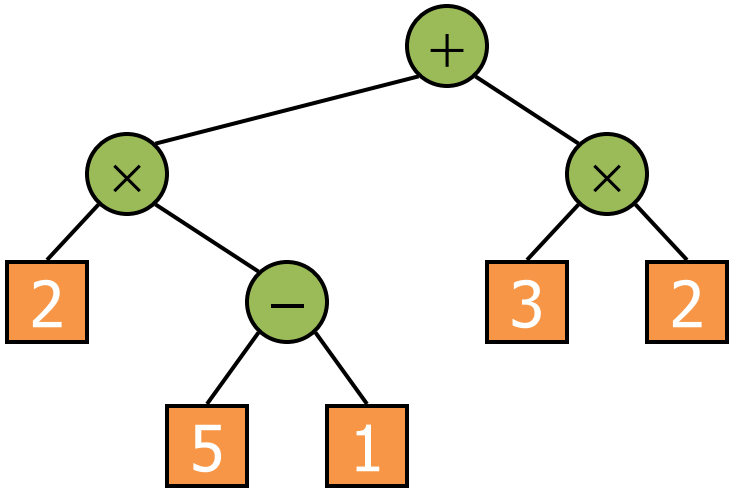


3.4.4.1. Arithmetic expression

- Binary tree are used to represent for an arithmetic expression:
  - internal nodes: operators
  - leaves: operands

Example 1: arithmetic expression tree for the expression

$$((2 \times (5 - 1)) + (3 \times 2))$$



Traversing expression tree:

1. preorder traversal

+ x 2 – 5 1 x 3 2

give us: prefix expression

2. inorder traversal

2 x 5 – 1 + 3 x 2

give us: infix expression

3. postorder traversal

2 5 1 – x 3 2 x +

give us: postfix expression

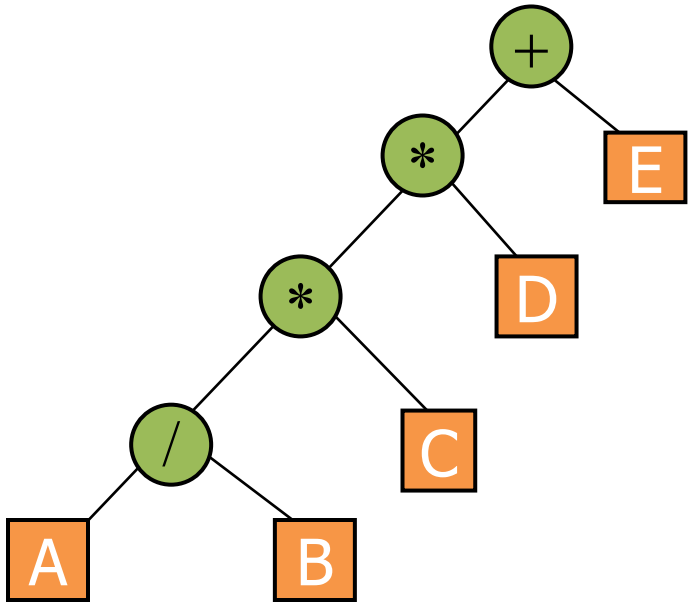
4. level order traversal (breath first traveral)

+ x x 2 – 3 2 5 1

3.4.4.1. Arithmetic expression

- Binary tree are used to represent for an arithmetic expression:
  - internal nodes: operators
  - leaves: operands

Example 2:



Traversing expression tree:

1. preorder traversal

+ * * / A B C D E

give us: prefix expression

2. inorder traversal

A / B * C * D + E

give us: infix expression

3. postorder traversal

A B / C * D * E +

give us: postfix expression

4. level order traversal (breath first traveral)

+ * E * D / C A B

Infix/Postfix/Prefix Expressions

- **INFIX**: the expressions in which operands surround the operator

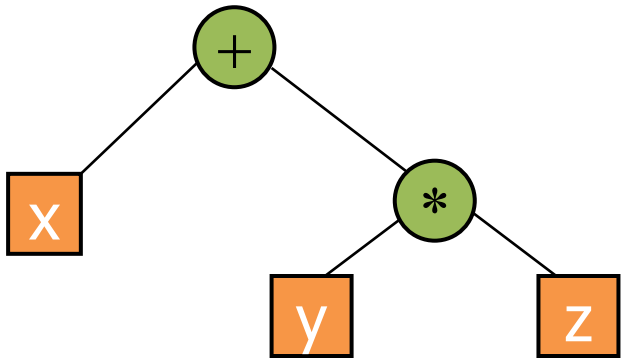
Example: x+y, x+y*z

- **POSTFIX** (also known as Reverse Polish Notation): operator comes after the operands

Example: xy+ , xyz*+

- **PREFIX** (also Known as Polish notation): operator comes before the operands

Example: +xy, +x*yz



Infix Expressions

- **INFIX**: the expressions in which operands surround the operator

How do you evaluate :

a * b + c / d

1) This is done by assigning operator priorities:

priority(*) **=** priority(/) **>** priority(+) **=** priority(-)

2) When an operand lies between two operators, the operand associates with the operator that has higher priority.

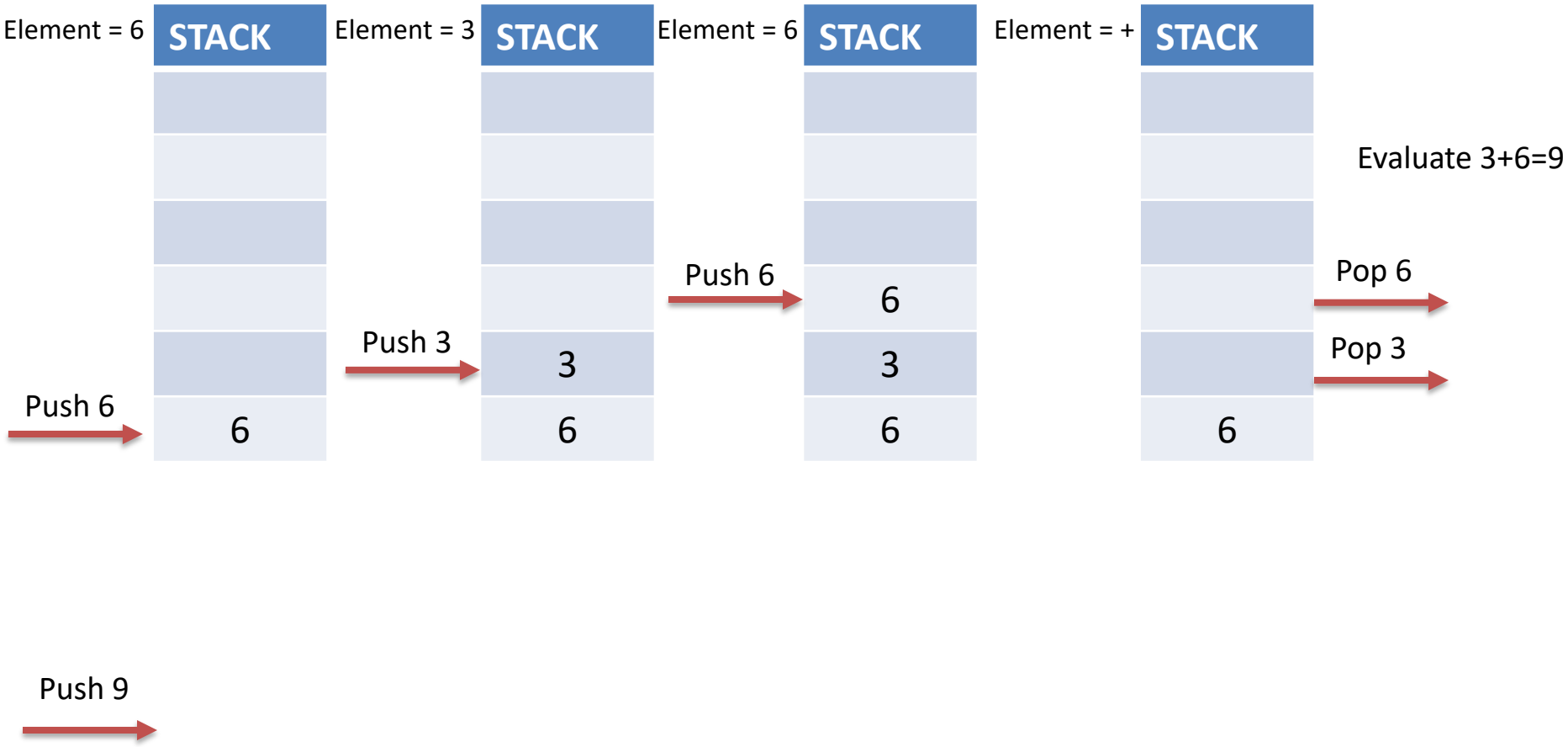
➔Operand b lies between * and + ==> b associates with *

➔Operand c lies between + and / ==> c associates with /

Therefore: a* b + c / d = (a*b) + (c/d)

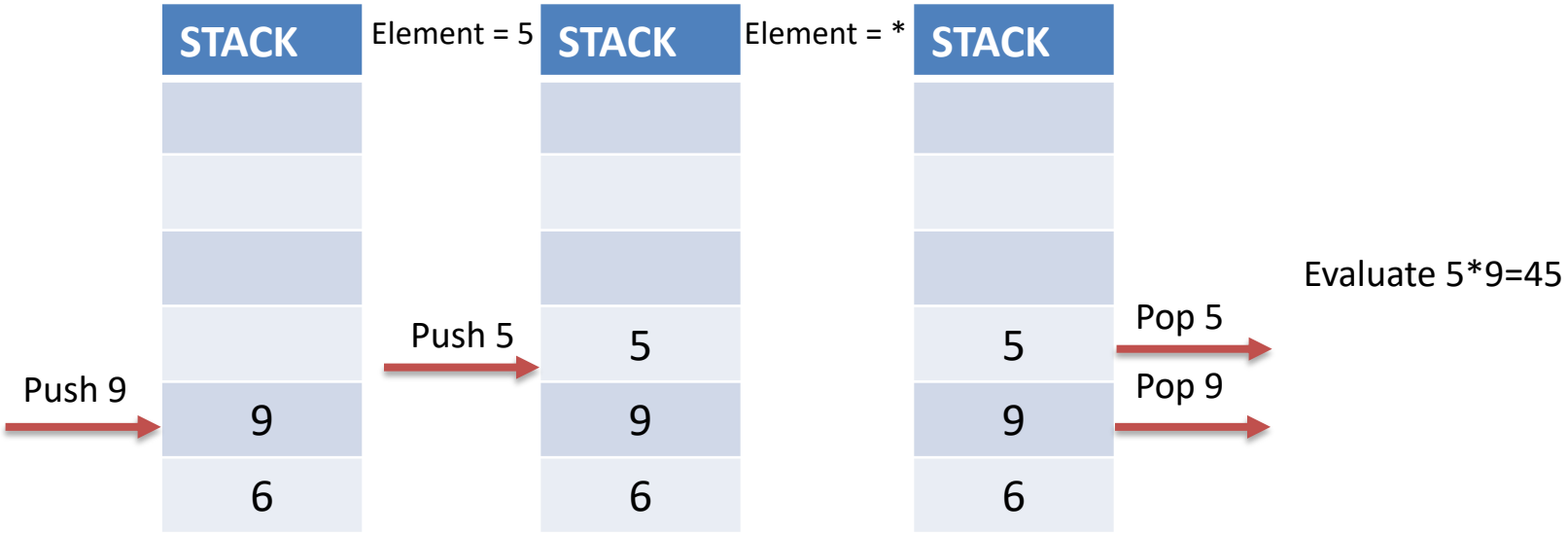
Example 1: Evaluate postfix expressions: using stack

- Evaluate the following postfix expression using stacks: **6 3 6 + 5 * 9 / -**
- The expression is evaluated by using stack as follows:

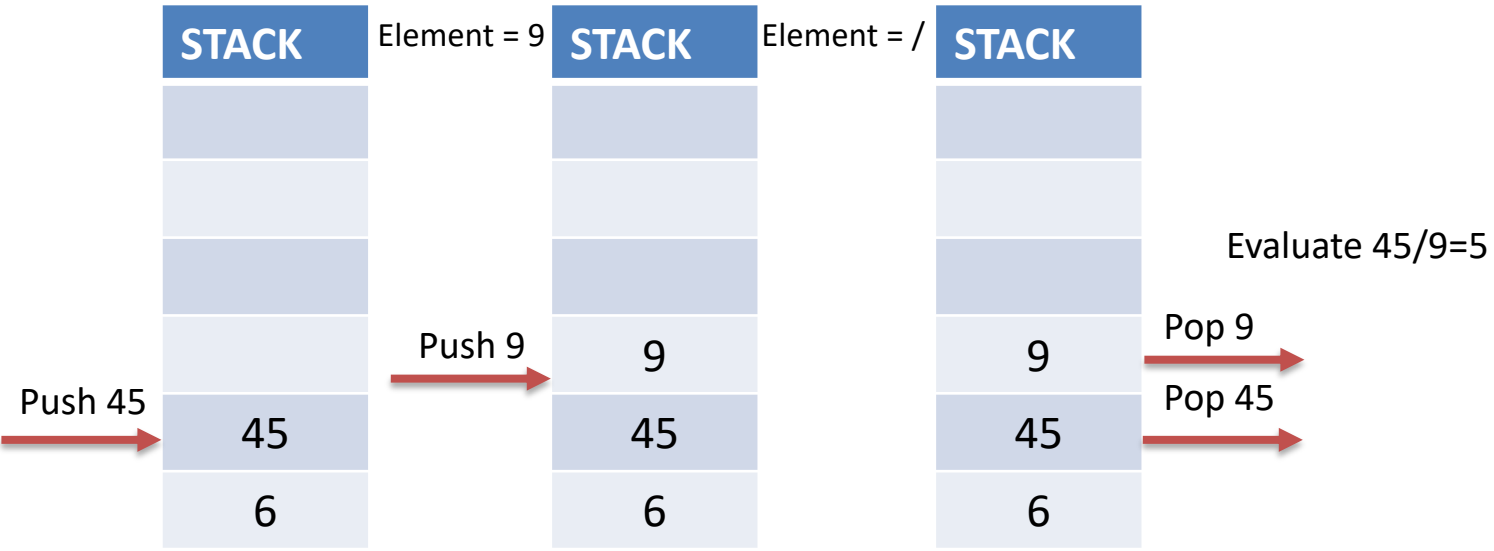


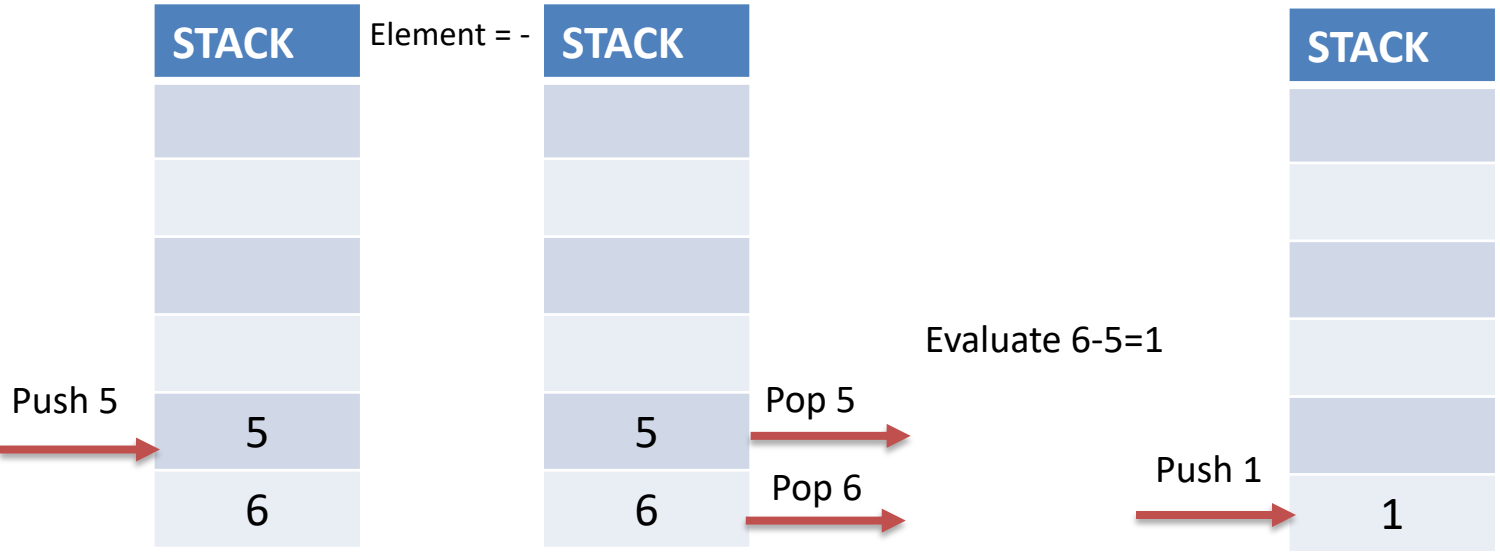
Example 1: Evaluate postfix expressions: using stack

- Evaluate the following postfix expression using stacks: **6 3 6 + 5 * 9 / -**
- The expression is evaluated by using stack as follows:



Evaluate postfix expressions using stack: 6 3 6 + 5 * 9 / -





Final state of stack with result

Evaluate postfix expressions: using stack

```
evaluationPostfix(stack S, postfixExpression E)
{ //Expression is stored in array E
  //Read expression from left to right:
   i = 0;
   while ( i < number of characters_in_postfix_expression)
   {
      if (E[i] is an operand)
             push E[i] onto stack
      else if (E[i] is an operator)
      {
        1.  Pop the top element from stack and store it in operand2
        2.  Pop the next top element from stack and store it in operand2
        3.  Evaluate: operand2 op operand1 and push the result onto stack
      }
      i = i + 1;
   }//end while
    Pop the top element, store it in Result;
    return Result; //which is value of expression
}
```

Example 1:Evaluate postfix expression:

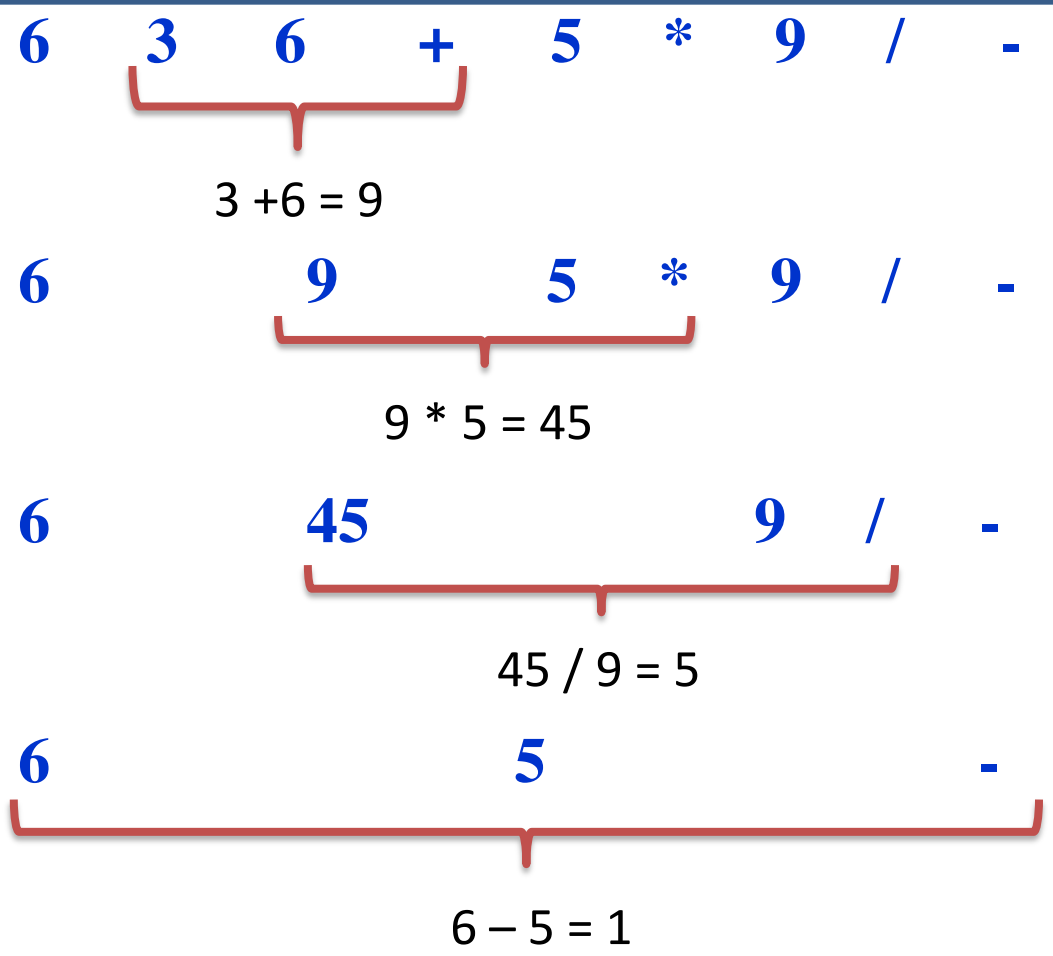
636+5*9/-

Value of expression = 1

2nd way to display the steps of algorithm

	Element of expression	Action performed	Stack status
1	6	Push 6 to stack	6
2	3	Push 3 to stack	6 3
3	6	Push 6 to stack	6 3 6
4	+	Pop 6	6 3
5		Pop 3	6
6		Evaluate 3 + 6 = 9	6
7		Push 9 to stack	6 9
8	5	Push 5 to stack	6 9 5
9	*	Pop 5	6 9
10		Pop 9	6
11		Evaluate 9*5=45	6
12		Push 45 to stack	6 45
13	9	Push 9 to stack	6 45 9
14	/	Pop 9	6 45
15		Pop 45	6
16		Evaluate 45/9=5	6
17		Push 5 to stack	6 5
18	-	Pop 5	6
19		Pop 6	Empty
20		Evaluate 6-5=1	Empty
21		Push 1 to stack	1
22		Pop value = 1	Empty

Example 1: Evaluate postfix expressions: using stack

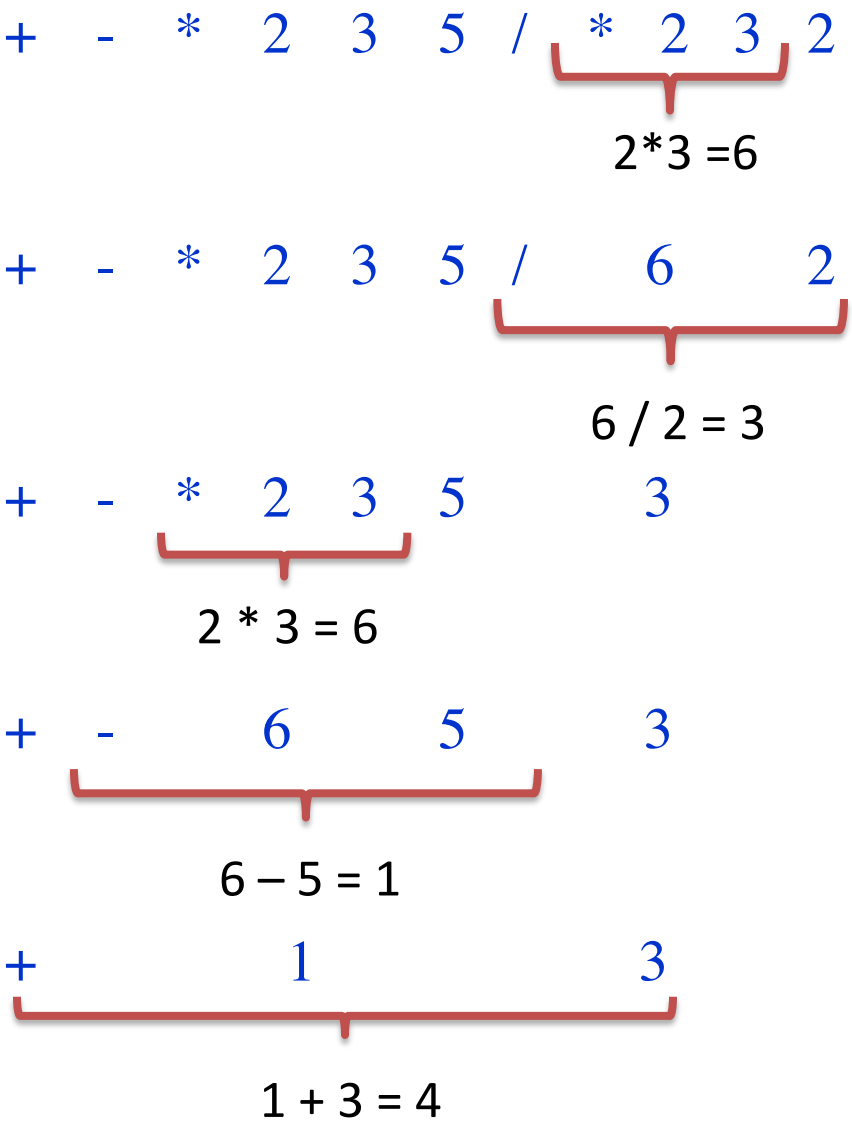


Read expression from left to right

Value of expression = 1

3rd way to display the steps of algorithm

Example 2: Evaluate prefix expressions: using stack



Read expression from right to left

Value of expression = 4

Example 2: Evaluate prefix expression:

+-*235/*232

	Element of expression	Action performed	Stack status
1	2	Push 2 to stack	2
2	3	Push 3 to stack	2 3
3	2	Push 2 to stack	2 3 2
4	*	Pop 2	2 3
5		Pop 3	2
6		Evaluate 2 * 3 = 6	2
7		Push 6 to stack	2 6
8	/	Pop 6 to stack	2
9		Pop 2	Empty
10		Evaluate: 6/2=3	Empty
11		Push 3 to stack	3
12	5	Push 5 to stack	3 5
13	3	Push 3 to stack	3 5 3
14	2	Push 2 to stack	3 5 3 2

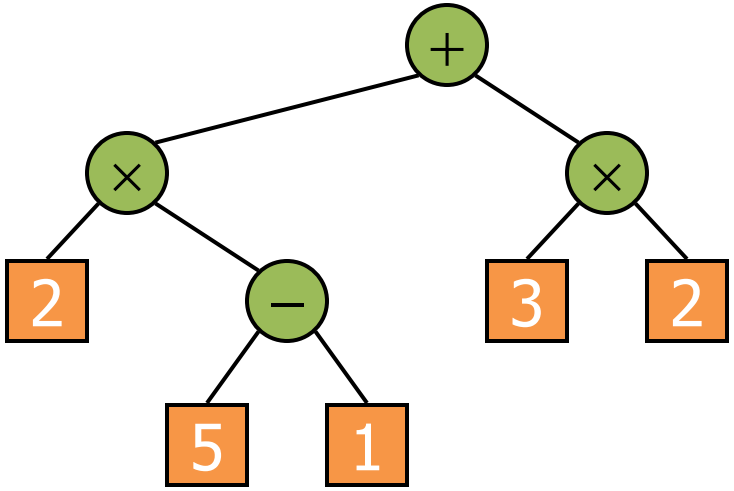
	Element of expression	Action performed	Stack status
15	*	Pop 2	3 5 3
16		Pop 3	3 5
17		Evaluate 2*3=6	3 5
18		Push 6 to stack	3 5 6
19	-	Pop 6	3 5
20		Pop 5	3
21		Evaluate 6-5 = 1	3
22		Push 1 to stack	3 1
23	+	Pop 1	3
24		Pop 3	Empty
25		Evaluate 1 + 3 = 4	Empty
26		Push 4 to stack	4
27		Pop value = 4	Empty

Value of expression = 4

Print Arithmetic Expressions

Given a binary tree having root pointer pointed to the root of the tree, and this tree represents an arithmetic expression. Write a function to print this expression.

- Inorder traversal:
  - print "(" before traversing left subtree
  - print operand or operator when visiting node
  - print ")" after traversing right subtree



Call `printTree(+);`

The expression is:

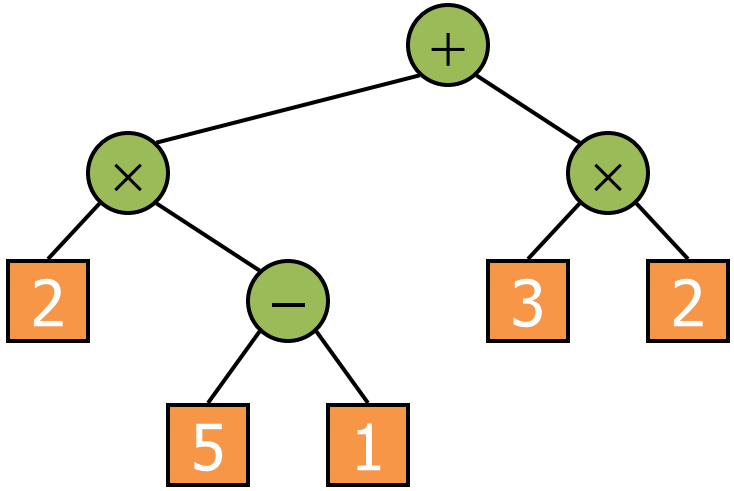
((2 × (5 - 1)) + (3 × 2))

```
void printTree(node *root)
    if (root.left != NULL)
    {
        print("(");
        printTree (root.left);
    }
    print(root.data);
    if (root.right != NULL)
    {
        printTree (root.right);
        print (")");
    }
```

Evaluate Arithmetic Expressions

Given a binary tree having root pointer pointed to the root of the tree, and this tree represents an arithmetic expression. Write a function to evaluate this expression.

- Postorder traversal:
  - Recursively evaluate subtrees
  - Apply the operator after subtrees are evaluated



Call `A = evaluate(+);`
The value of A is:

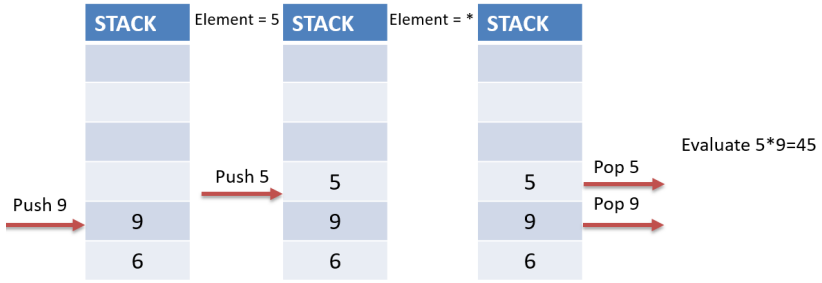
```
int evaluate (node *root)
  if (root.left == NULL)//external node
    return root.data;
   else  //internal node
     x = evaluate (root.left);
     y = evaluate (root.right);
     let o be the operator root.data
     z = apply o to x and y
     return z;
```

Exercise: Evaluate expression using stack

Exercise 1: evaluate postfix expression using stack

1) 2 3 + 4 * 5 *
2) 27 3 2 ^ / 3 17 * + 27 2 * -
3) 7 6 + 4 * 410 - 5 ^
4) 7 5 – 9 2 / *

1st way to display the steps of algorithm:



Exercise 2: evaluate prefix expression using stack

1) - + 7 * 4 5 + 2 10
2) - * 9 + 2 3 / 27 3

2nd way to display the steps of algorithm:

	Element of expression	Action performed	Stack status
1	6	Push 6 to stack	6
2	3	Push 3 to stack	6 3
3	6	Push 6 to stack	6 3 6

3rd way to display the steps of algorithm:

6 9 5 * 9 / -

9 * 5 = 45

6 45 9 / -

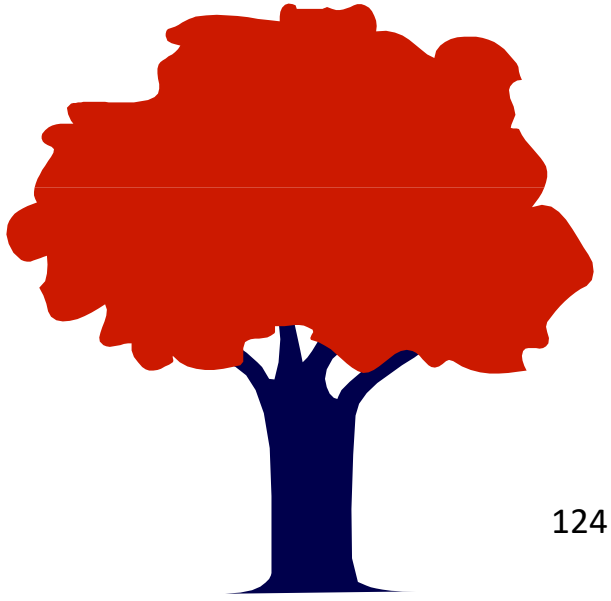
45 / 9 = 5

Read expression from left to right

3.4.4. Some applications

3.4.4.1. Arithmetic expression

3.4.4.2. Huffman code



Exercise

- Suppose that we have a 100,000 character data file that we wish to store. The file contains only 6 characters, appearing with the following frequencies:

	a	b	c	d	e	f
Frequency	45000	13000	12000	16000	9000	5000

- We encode each character as a binary string (called **codeword**). Thus, we need to find a binary code that encodes the file using as few bits as possible, i.e., compresses it as much as possible.
- There are two possible ways to encode:
  - A fixed-length code: each codeword has the same length.
  - A variable-length code: codewords may have different lengths.
- Example: if using fixed-length code for our problem, we need at least 3 bits per codeword as there are 6 characters ($2^2 = 4 < 6$, $2^3 = 8$)

	a	b	c	d	e	f
Frequency	45000	13000	12000	16000	9000	5000
A fixed-length	000	001	010	011	100	101
A variable-length	0	101	100	111	1101	1100

Exercise

- Example: if using fixed-length code for our problem, we need at least 3 bits per codeword as there are 6 characters ($2^2 = 4 < 6, 2^3 = 8$)

	a	b	c	d	e	f
Frequency	45000	13000	12000	16000	9000	5000
A fixed-length	000	001	010	011	100	101
A variable-length	0	101	100	111	1101	1100

- The fixed length-code requires:

$$3 * 100000 = 300000 \text{ bits to store the file}$$

3 bits for each character Number of characters

- The variable-length code uses only:

$$1 * 45000 + 3*13000 + 3*12000 + 3 * 16000 + 4 * 9000 + 4 * 5000 = 224000 \text{ bits}$$

#bits used to encode character "a"

#bits used to encode character "f"

➔ saving a lot of space: 25%

Exercise

- There are two possible ways to encode:
  - A fixed-length code: each codeword has the same length

    ➔ easy to encode and decode, but requires larger memory.
  - A variable-length code: codewords may have different lengths
    - Assign the longest codes to the most infrequent events.
    - Assign the shortest codes to the most frequent events.
    - Each code word must be uniquely identifiable regardless of length➔ no codeword is a prefix of another one (example: {a = 1, b = 110, c = 10, d = 111}, then when encoding, what is "1101111"?)

    ➔ Prefix code: a code is called a prefix code if no codeword is a prefix of another one (Example: {a = 0, b = 110, c = 10, d = 111} is a prefix code)

	a	b	c	d	e	f
Frequency	45000	13000	12000	16000	9000	5000
A fixed-length	000	001	010	011	100	101
A variable-length	0	101	100	111	1101	1100

Optimum source coding problem

- The variable-length code uses only 224000 bits rather than 300000 bits of fixed-length code ➔ save 25%. Question: Could we do better ? Or is this already the optimum (lowest cost) prefix code ?

	a	b	c	d	e	f
Frequency	45000	13000	12000	16000	9000	5000
A fixed-length	000	001	010	011	100	101
A variable-length	0	101	100	111	1101	1100

- The optimum source coding problem: Given an alphabet $A = \{a_1, \ldots, a_n\}$ with frequency distribution $f(a_i)$, find a binary prefix code C for A that minimizes the number of bits

$$B(C) = \sum_{a=1}^{n} f(a_i)L(c(a_i))$$

needed to encode a message of $\sum_{a=1}^{n} f(a)$ characters, where:

  - $c(a_i)$ is the codeword for encoding a_i ,
  - $L(c(a_i))$ is the length of the codeword $c(a_i)$.

Optimum source coding problem

- The optimum source coding problem: Given an alphabet $A = \{a_1, \ldots, a_n\}$ with frequency distribution $f(a_i)$, find a binary prefix code C for A that **minimizes** the number of bits

$$B(C) = \sum_{a=1}^{n} f(a_i) L(c(a_i))$$

needed to encode a message of $\sum_{a=1}^{n} f(a)$ characters, where:

  - $c(a_i)$ is the codeword for encoding a_i ,
  - $L(c(a_i))$ is the length of the codeword $c(a_i)$.

Remark: Huffman developed a nice greedy algorithm for solving this problem and producing a minimum cost (optimum) prefix code. The code that it produces is called a **Huffman code**.

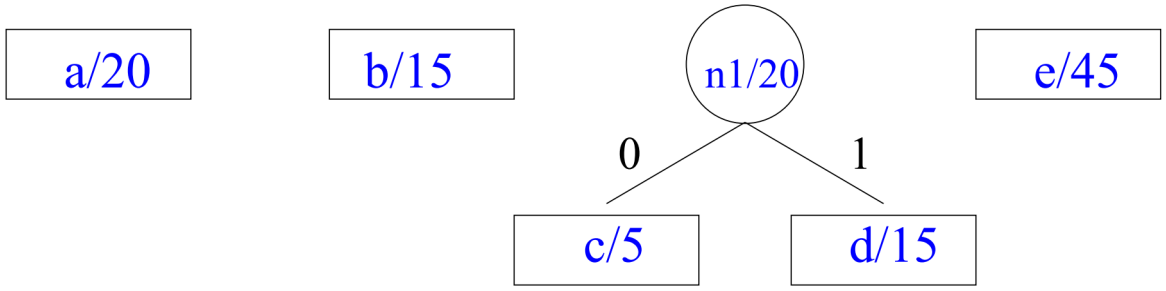
3.4.4.2 Huffman code

- Step 1: Pick two letters x, y from alphabet A with the smallest frequencies and create a subtree that has these two characters as leaves. (greedy idea) Label the root of this subtree as z.
- Step 2: Set frequency f(z) = f(x) + f(y). Remove x, y and add z creating new alphabet: $A' = A \cup \{z\} - \{x, y\}$, note that $|A'| = |A| - 1$
- Repeat this procedure, called **merge**, with new alphabet A' until an alphabet with only one symbol is left.

The resulting tree is the Huffman code

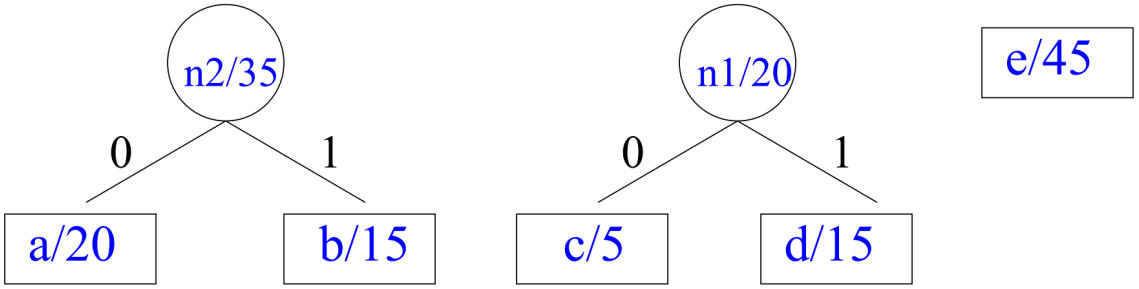
3.4.4.2 Huffman code

- Let A = {**a** / 20, **b** / 15, **c** / 5, **d** /15, **e** / 45} be the alphabet and its frequency distribution.
- In the first step Huffman coding merges c and d



Alphabet is now A_1 = {**a** / 20, **b** / 15, **n1** / 20, **e** / 45}

- Algorithm merges a and b (could also b and n1)

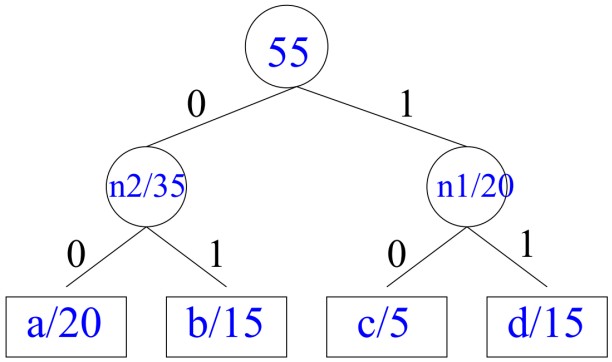


Alphabet is now A_2 = {**n2** / 35, **n1**: / 20, **e** / 45}

3.4.4.2 Huffman code

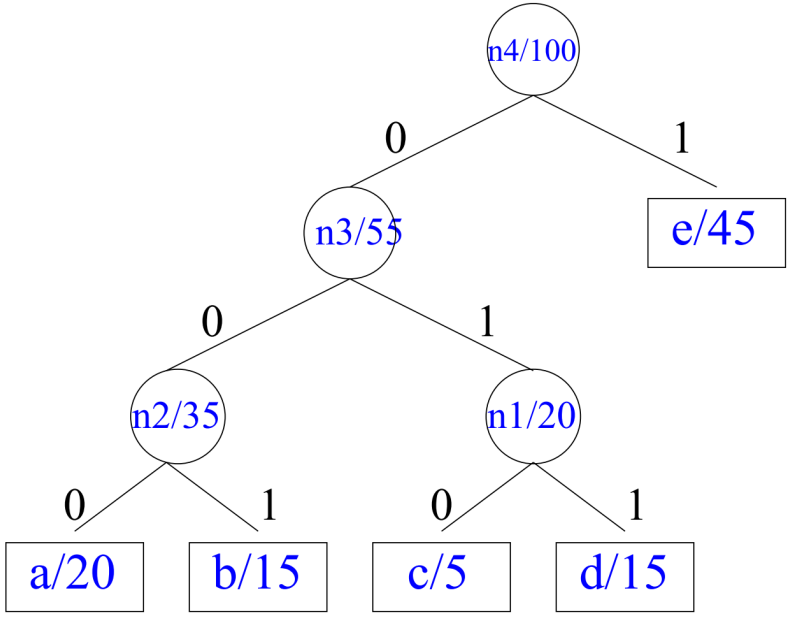
Alphabet is now A_2 = {**n2** / 35, **n1** / 20, **e** / 45}

Algorithm merges n1 and n2:



Alphabet is now A_2 = {**n3** / 55, **e** / 45}

Algorithm merges e and n3 and finishes



Huffman code is:

a = 000, b = 001, c = 010, d = 011, e = 1

This is the optimum (minimum-cost) prefix code for this distribution.

3.4.4.2 Huffman code: Creating Tree

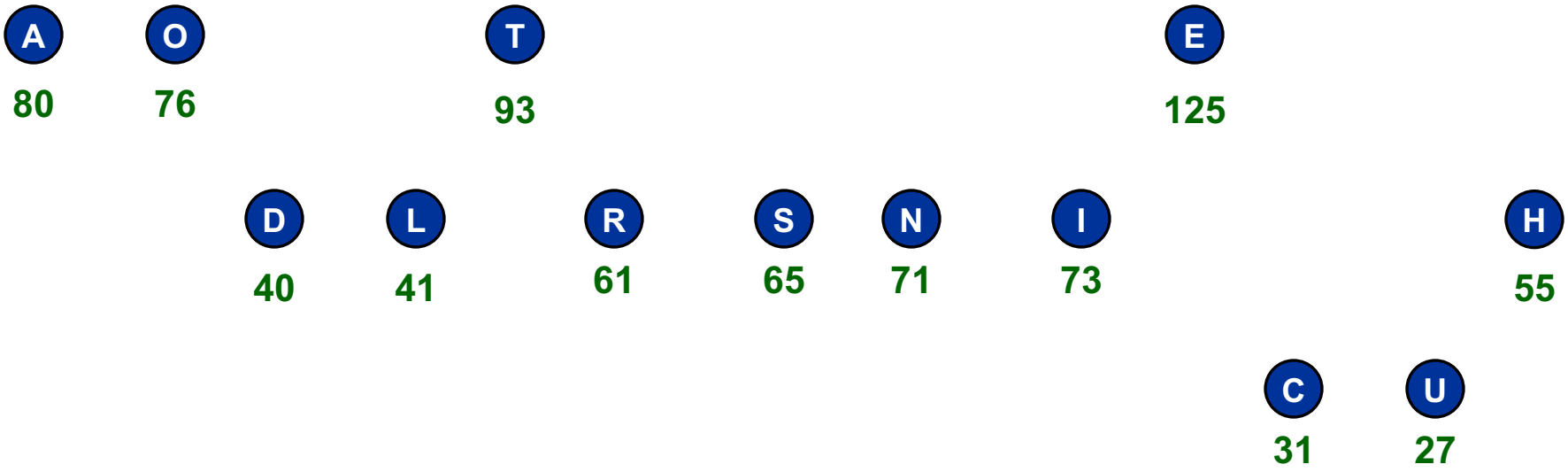
- Algorithm:
  - Place each symbol in leaf
    - Weight of leaf = symbol frequency
  - Select two trees L and R (initially leafs)
    - Such that L, R have lowest frequencies in tree
  - Create new (internal) node
    - Left child $\Rightarrow$ L
    - Right child $\Rightarrow$ R
    - New frequency $\Rightarrow$ frequency(L) + frequency(R)
  - Repeat until all nodes merged into one tree

3.4.4.2 Huffman code

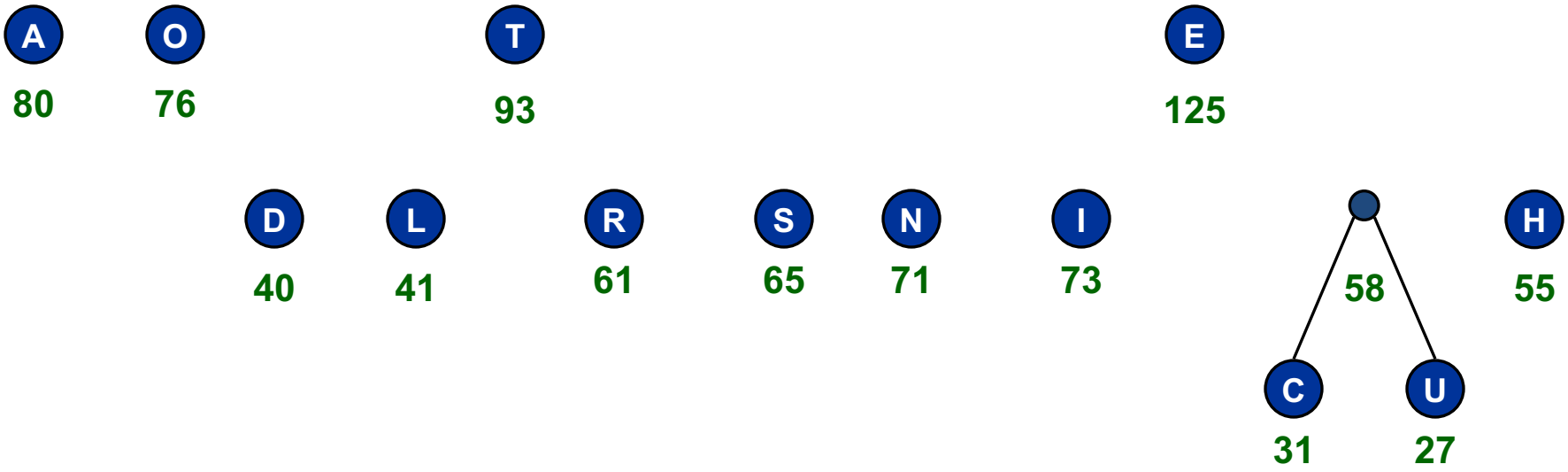
- Frequency of characters used in a text file:

Char	Freq
E	125
T	93
A	80
O	76
I	72
N	71
S	65
R	61
H	55
L	41
D	40
C	31
U	27

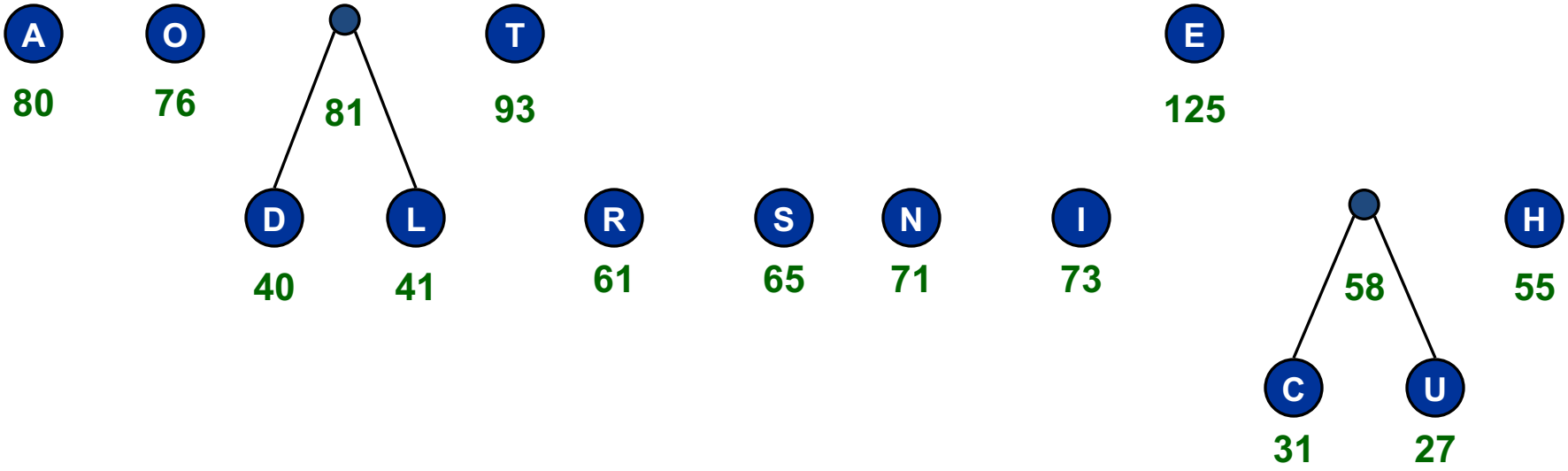
3.4.4.2 Huffman code



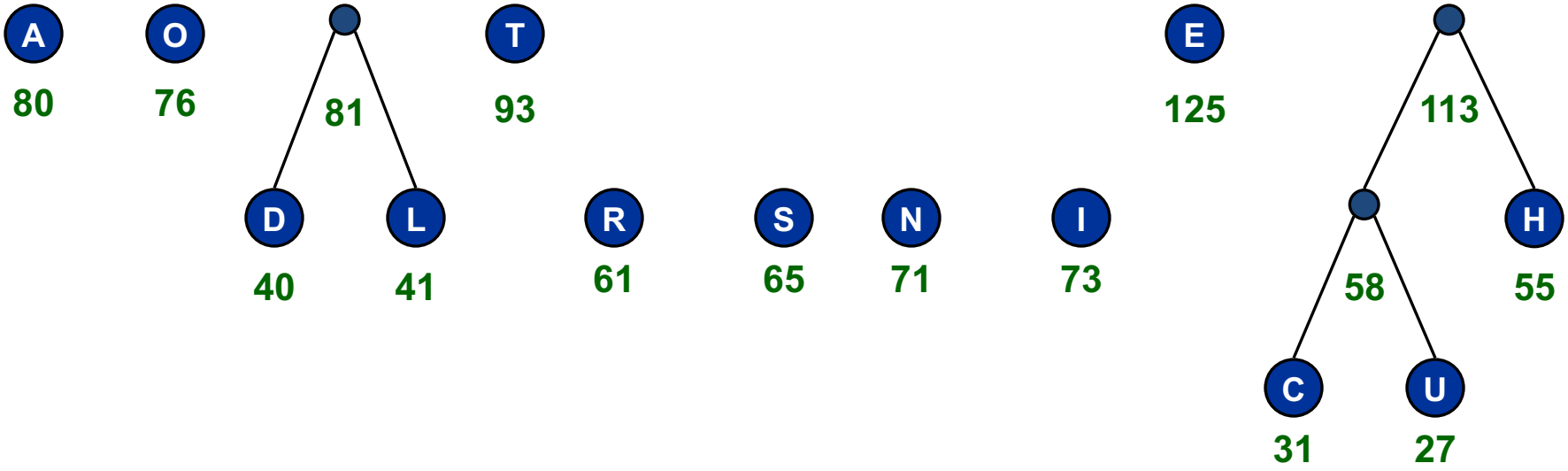
3.4.4.2 Huffman code



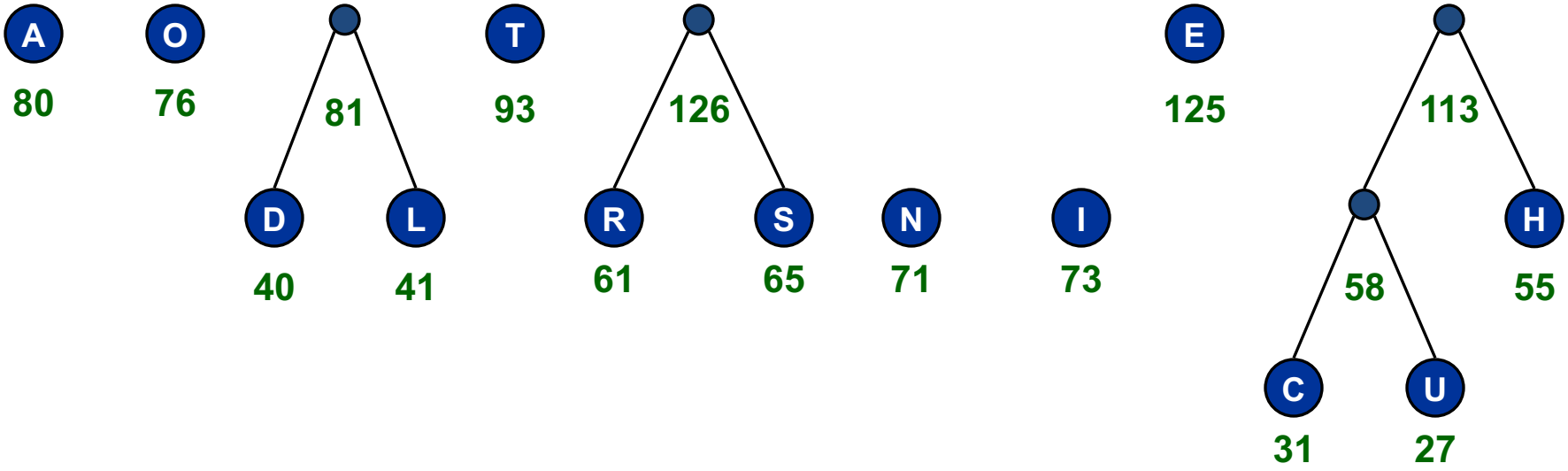
3.4.4.2 Huffman code



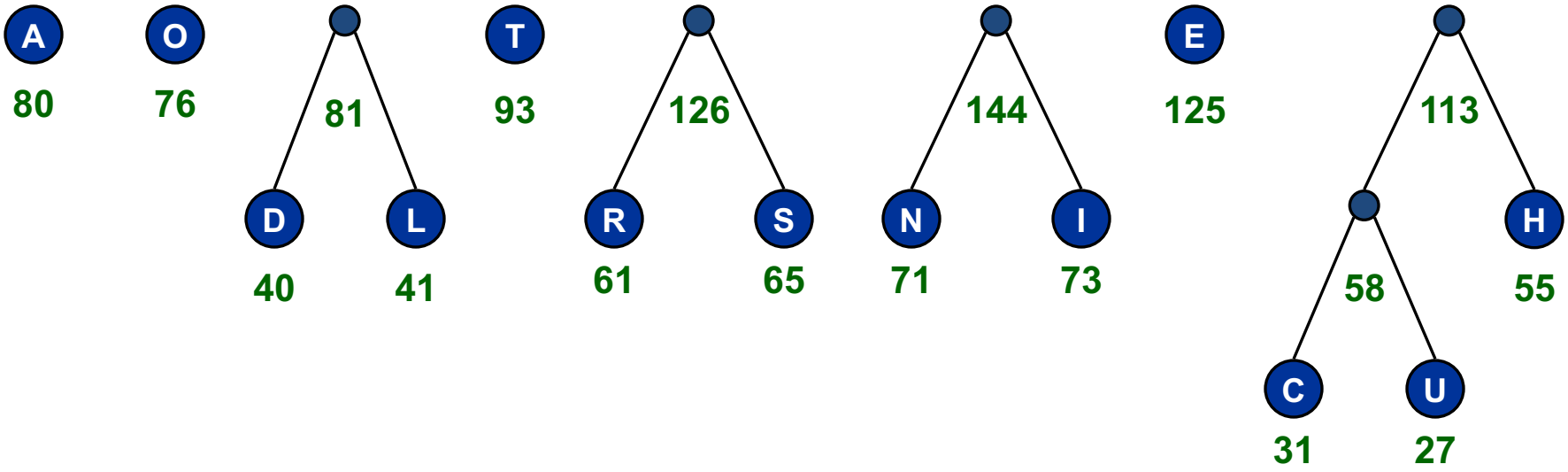
3.4.4.2 Huffman code



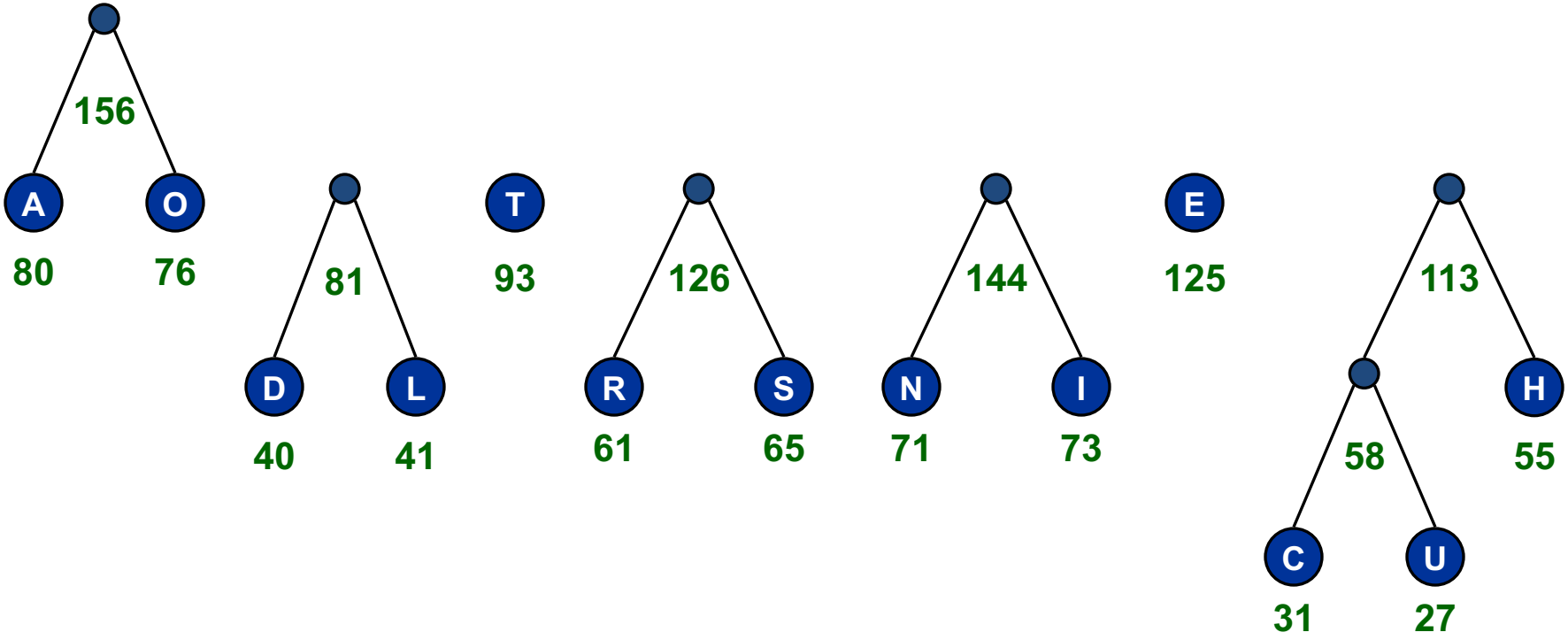
3.4.4.2 Huffman code



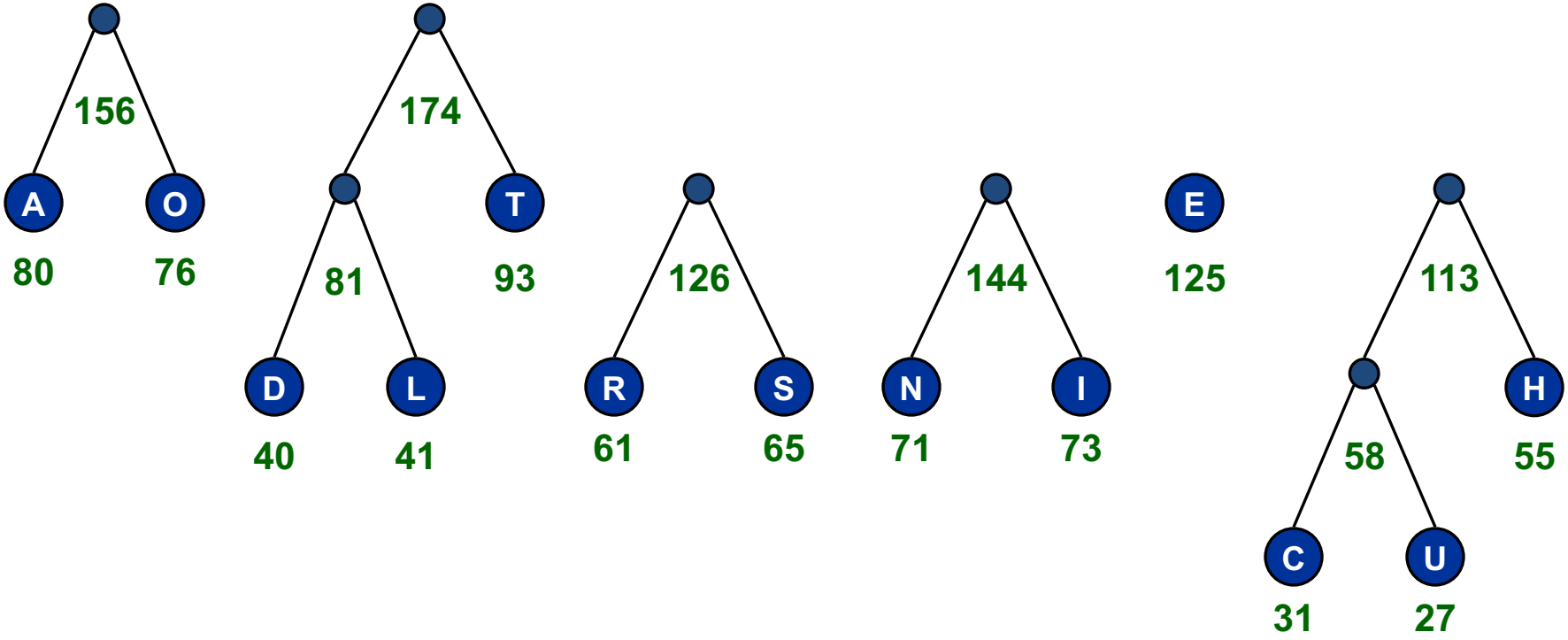
3.4.4.2 Huffman code



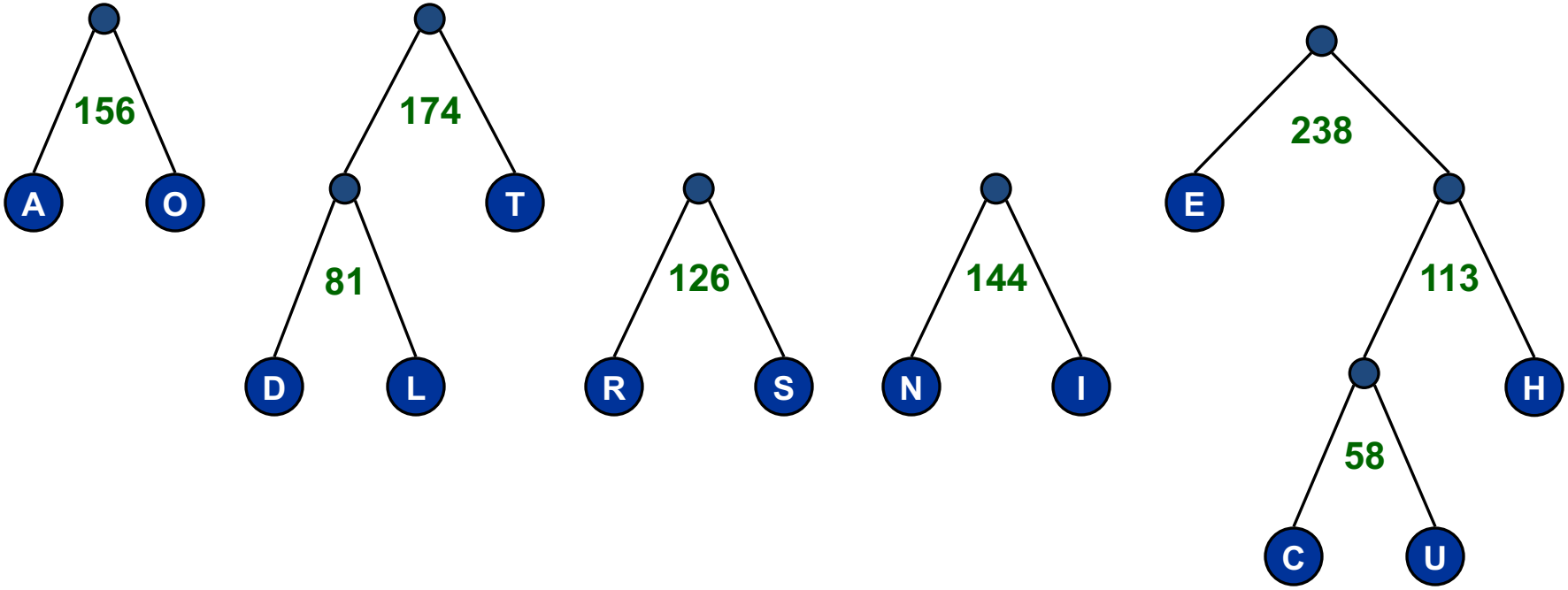
3.4.4.2 Huffman code



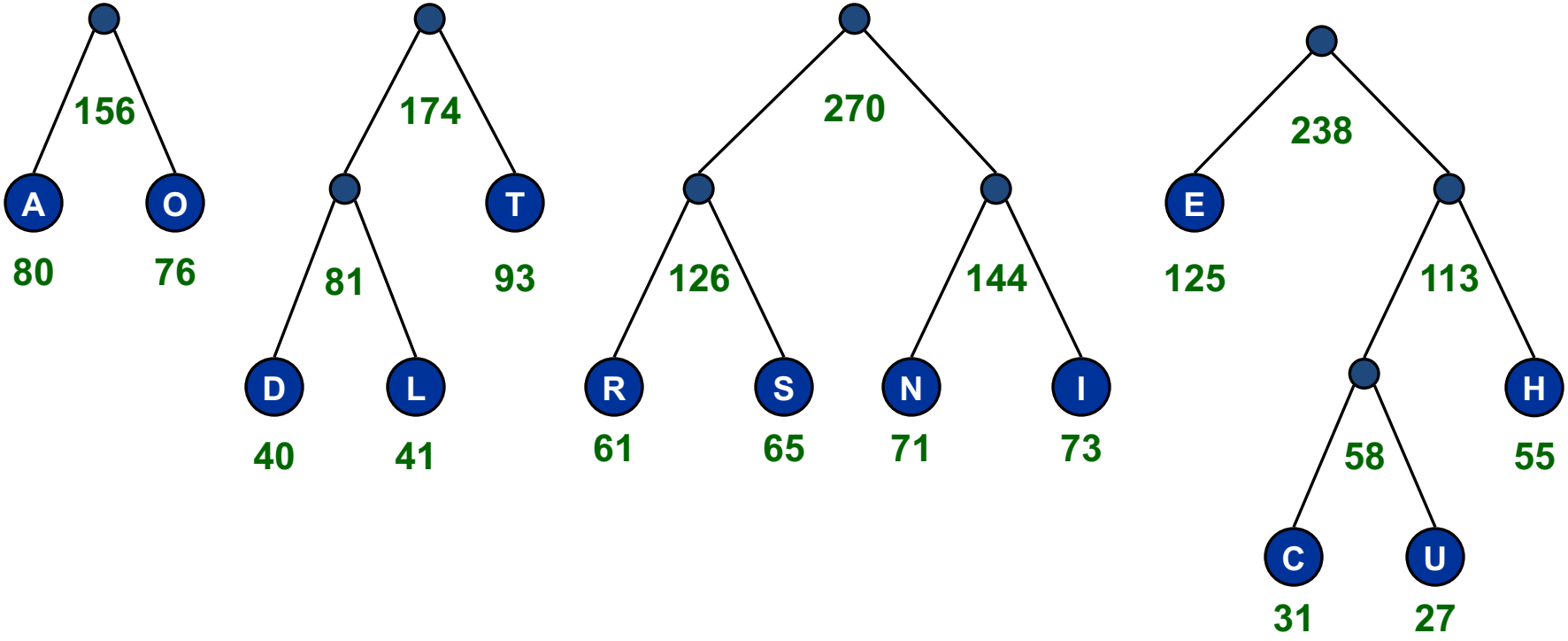
3.4.4.2 Huffman code



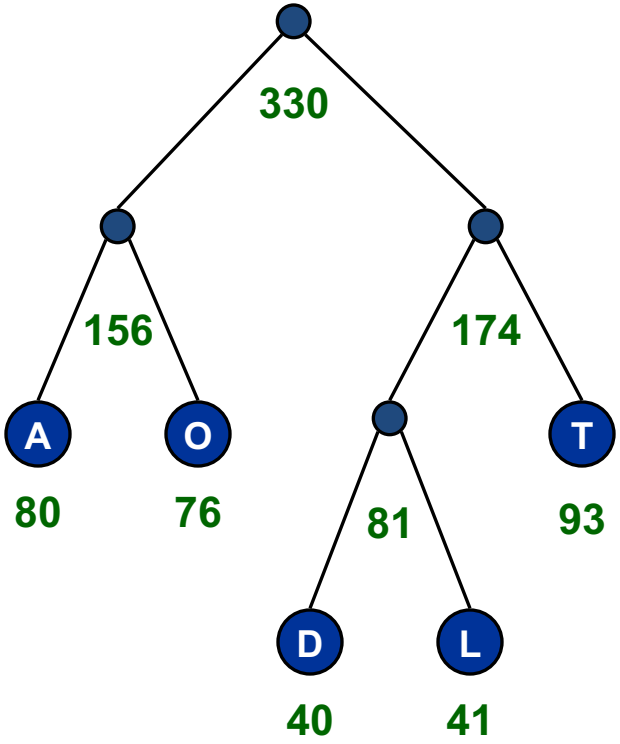
3.4.4.2 Huffman code

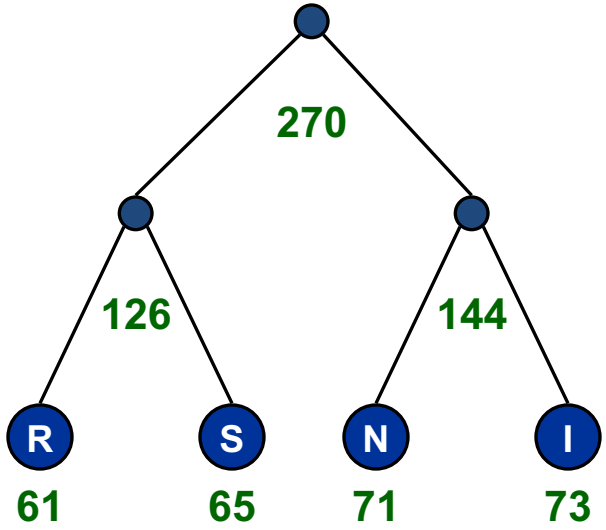


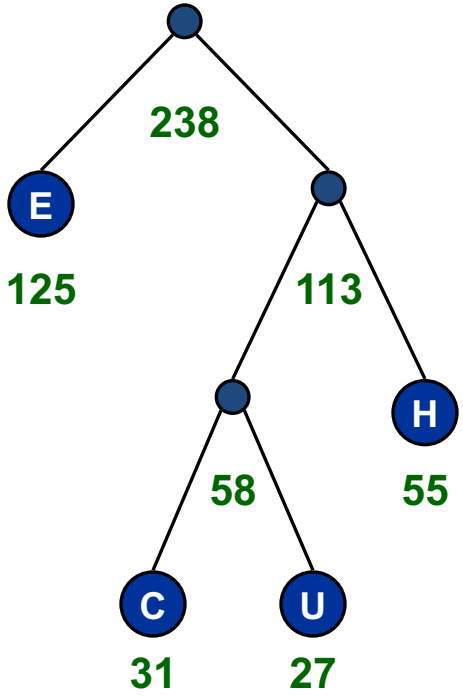
3.4.4.2 Huffman code



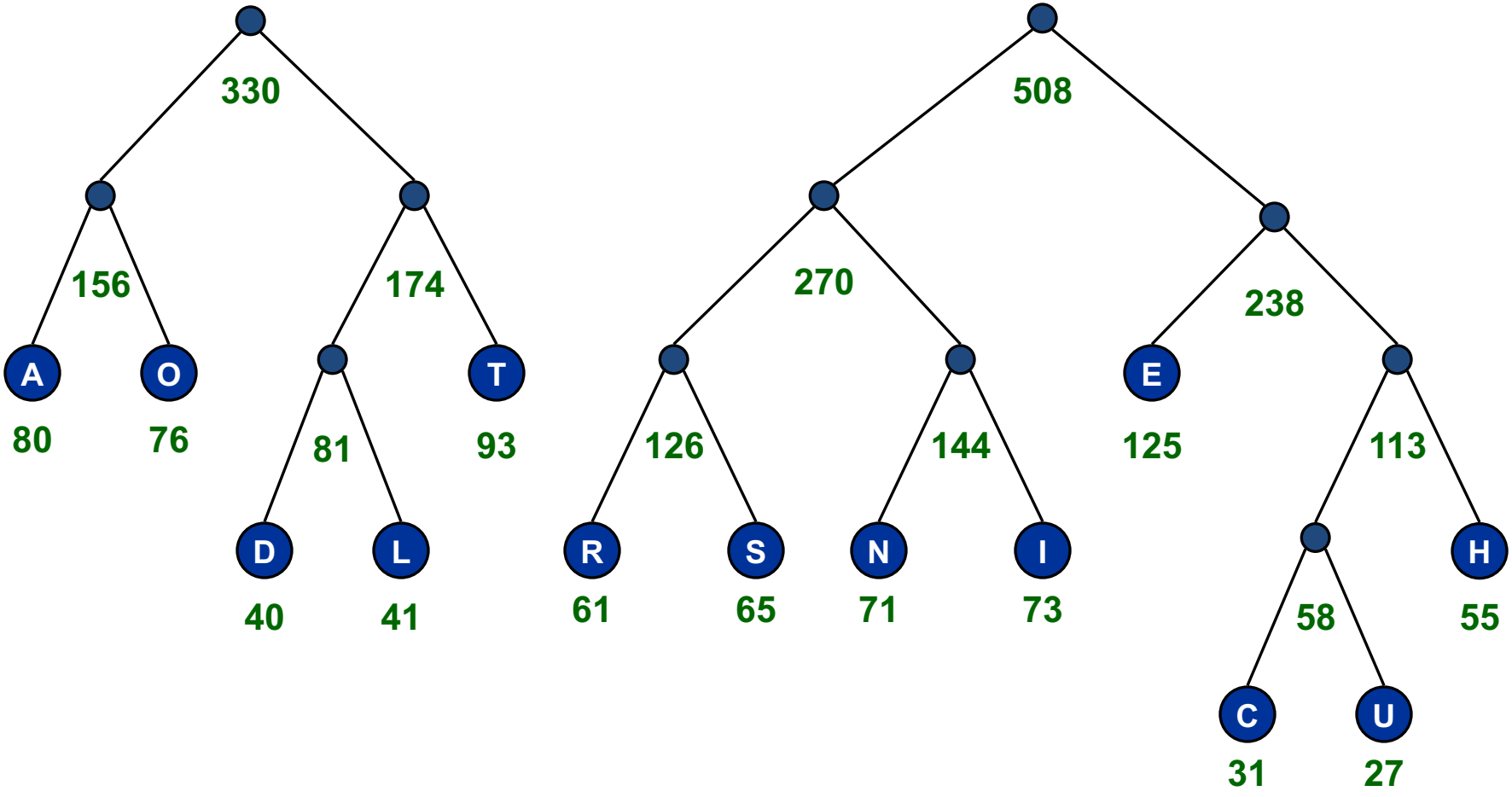
3.4.4.2 Huffman code



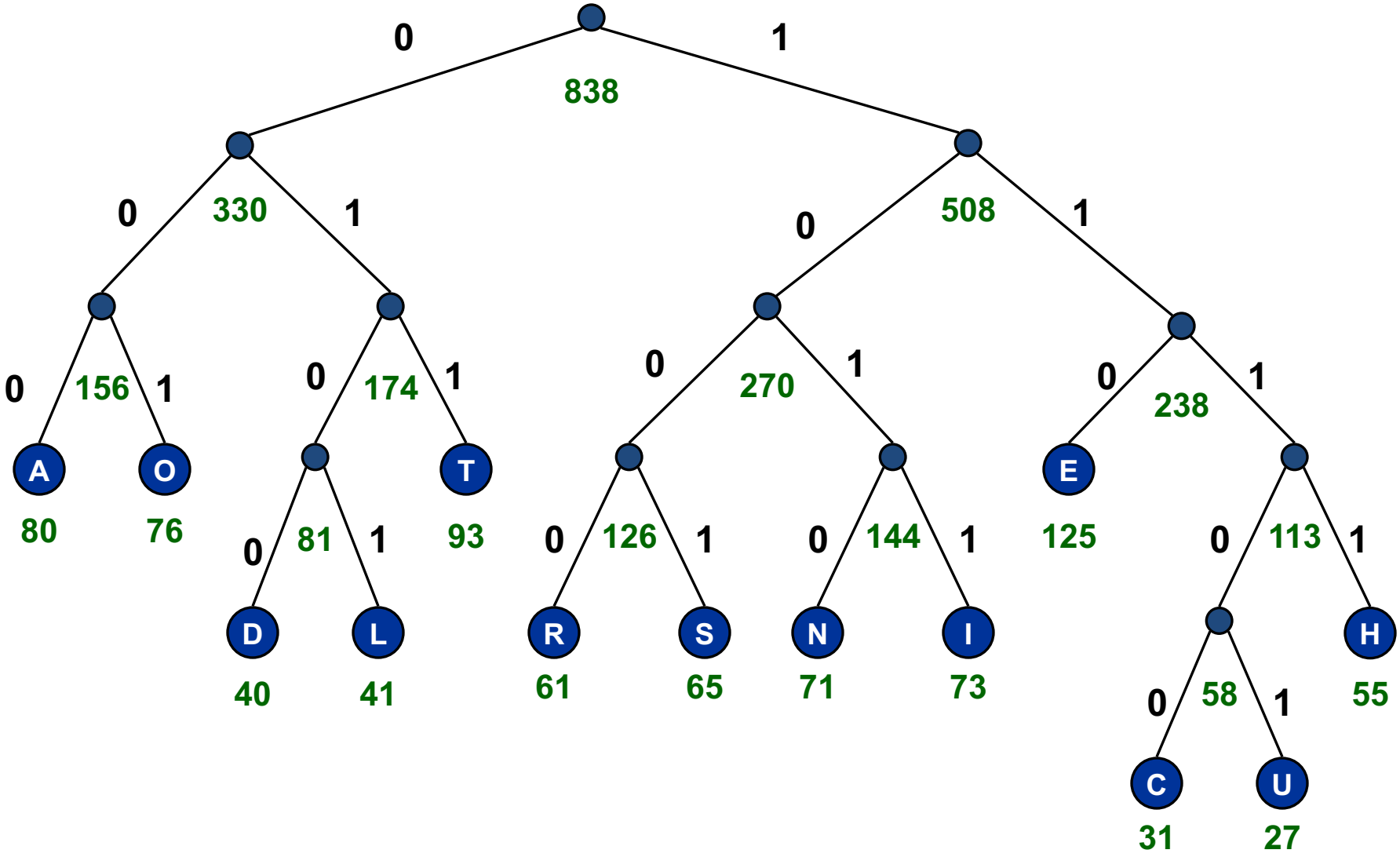




3.4.4.2 Huffman code



3.4.4.2 Huffman code



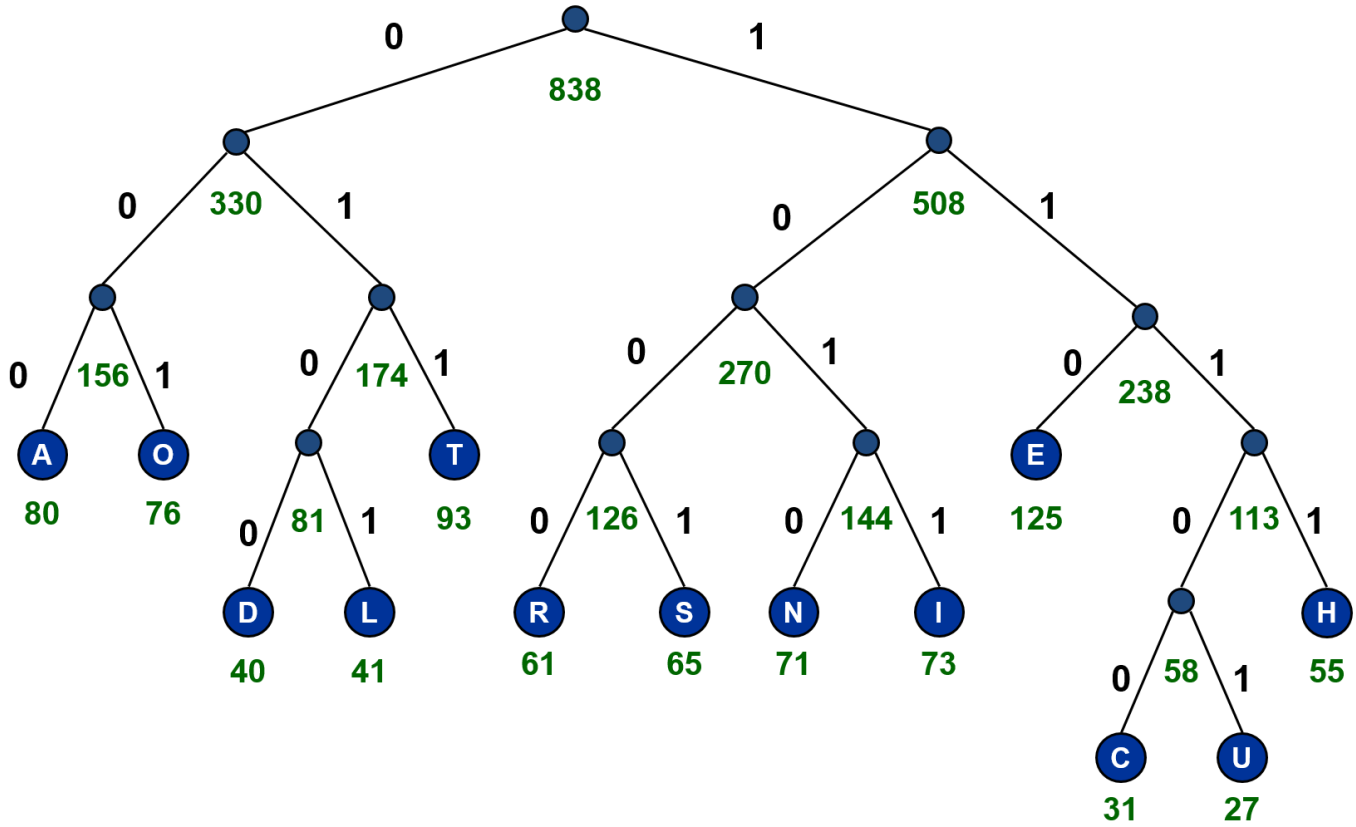
3.4.4.2 Huffman code

Char	Frequency	Fixed-length code	Huffman code
E	125	0000	110
T	93	0001	011
A	80	0010	000
O	76	0011	001
I	73	0100	1011
N	71	0101	1010
S	65	0110	1001
R	61	0111	1000
H	55	1000	1111
L	41	1001	0101
D	40	1010	0100
C	31	1011	11100
U	27	1100	11101
Sum	838	3352	3036

- Use Huffman code: save 10%
- Huffman encoding is a simple example of **data compression**: representing data in fewer bits than it would otherwise need

3.4.4.2 Huffman code: Decoding

- Algorithm decoding:
  - Read compressed file & binary tree
  - Use binary tree to decode file
    - Follow path from root to leaf
- Example: Decode “11100000011”



Data compression

- Huffman encoding is a simple example of data compression: representing data in fewer bits than it would otherwise need
- A more sophisticated method is GIF (Graphics Interchange Format) compression, for .gif files
- Another is JPEG (Joint Photographic Experts Group), for .jpg files
  - Unlike the others, JPEG is lossy—it loses information
  - Generally OK for photographs (if you don't compress them *too* much), because decompression adds "fake" data very similar to the original

Infix Expressions

- Used in arithmetic
- Operator is placed between the operands
- Parentheses are used to denote the priority of the operation
- Examples
  - a + b
  - (a + b) * (a - b)
  - (a ^ b * (c + (d * e) - f)) / g

Postfix Expressions

- Mathematical Notation
- Operator is placed after all its operands
- No need of parentheses
- Easy to parse as compared to infix
- Examples:

Infix Expression	Postfix Expressions
a + b	a b +
(a + b) * (a - b)	a b + a b - *
(a ^ b * (b + (d * e) - f)) / g	a b ^ c d e * + f - * g /

Prefix Expressions

- Mathematical Notation
- Operator is placed before all its operands
- No need of parentheses
- Easy to parse as compared to infix
- A valid prefix expression always starts with an operator and ends with an operand.
- Examples:

Infix Expression	Prefix Expressions
a + b	+ a b
(a + b) * (a - b)	* + a b - a b
(a ^ b * (c + (d * e) - f)) / g	/ * ^ a b - + * d e c f g

Evaluating Prefix Expressions

Example 1 (a=5, b=3)	Example 2(a=4, b=2, c=5, d=6, e=7, f=8, g=3)
* + a b - a b - * **+ 5 3** **- 5 3** * **8** **2** ***** **8** **2** **16 (Answer)**	/ * ^ a b - + * d e c f g / * **^ 4 2** - + *** 6 7** 5 8 3 / * **16** - + **42** 5 8 3 / * 16 - **+ 42 5** 8 3 / * 16 - **47** 8 3 / * 16 **-** **47** **8** 3 / * 16 **39** 3 / ***** **16** **39** 3 / **624** 3 **/** **624** **3** **208 (Answer)**

Evaluating Postfix Expressions

Example 1 (a=5, b=3)	Example 2(a=4, b=2, c=5, d=6, e=7, f=8, g=3)
a b + a b - * **5 3 +** **5 3 -** * **8 2** * **8 2 *** **16** (Answer)	a b ^ c d e * + f - * g / **4 2 ^** 5 **6 7 *** + 8 - * 3 / **16** 5 **42** + 8 - * 3 / 16 **5 42 +** 8 - * 3 / 16 **47** 8 - * 3 / 16 **47** **8 -** * 3 / 16 **39** * 3 / **16** **39** ***** 3 / **624** 3 / **624** **3 /** **208 (Answer)**

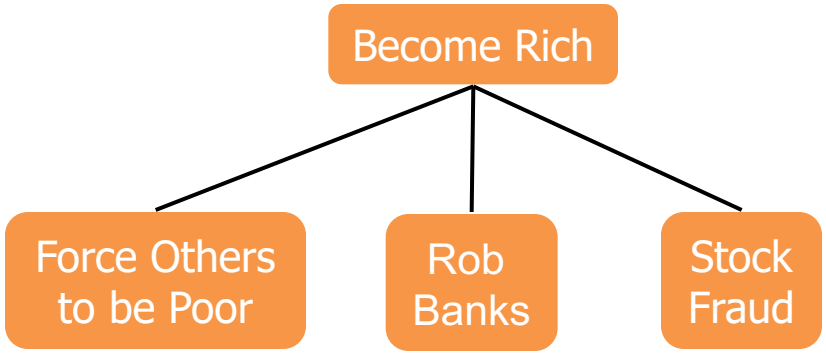
Why trees?

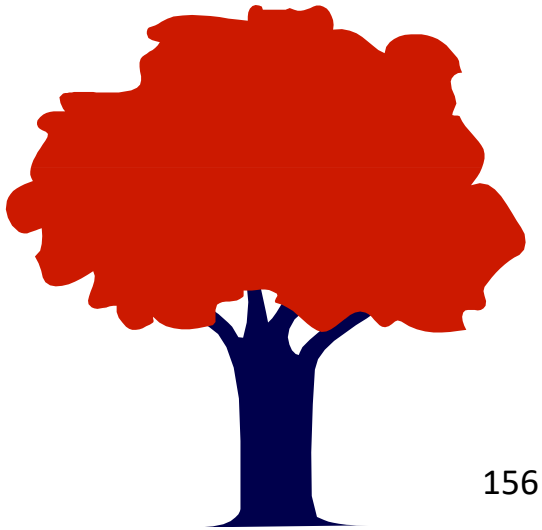
Upfront cost ➔ improved searching

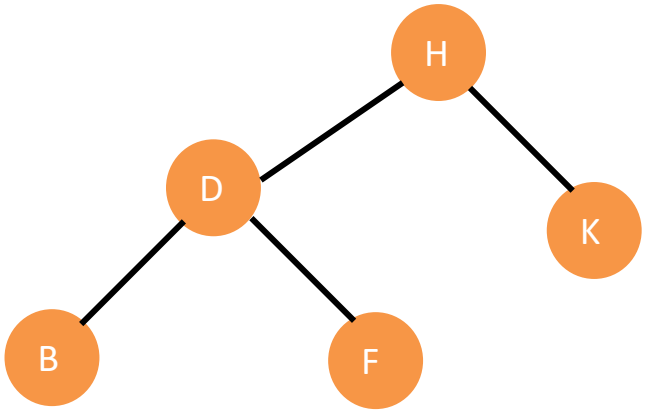
Searching array/linked list: O(N)

Binary search tree: longer setup, shorter search

Reason for shortened search is that nodes on other branches are not iterated through







H	D	K	B	F	J	NULL	NULL	NULL	A	NULL	NULL	C	NULL			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	.....	25	26

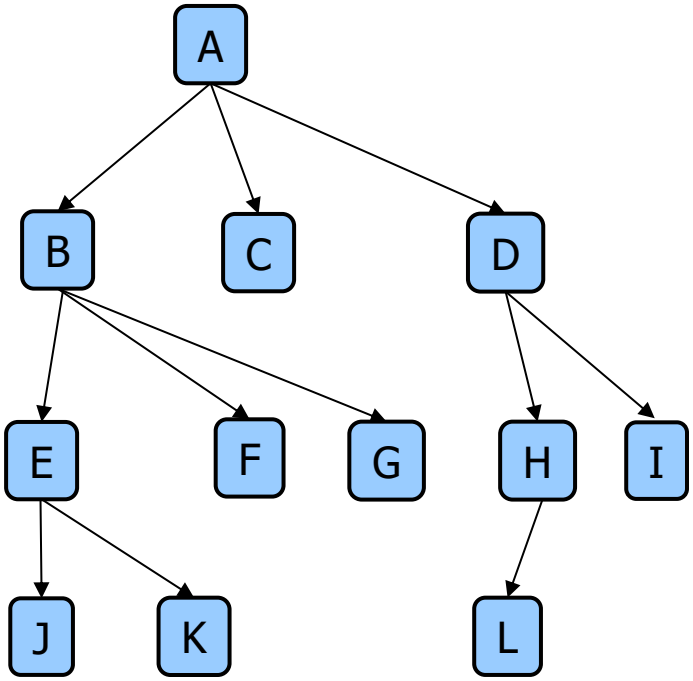
Right child of K

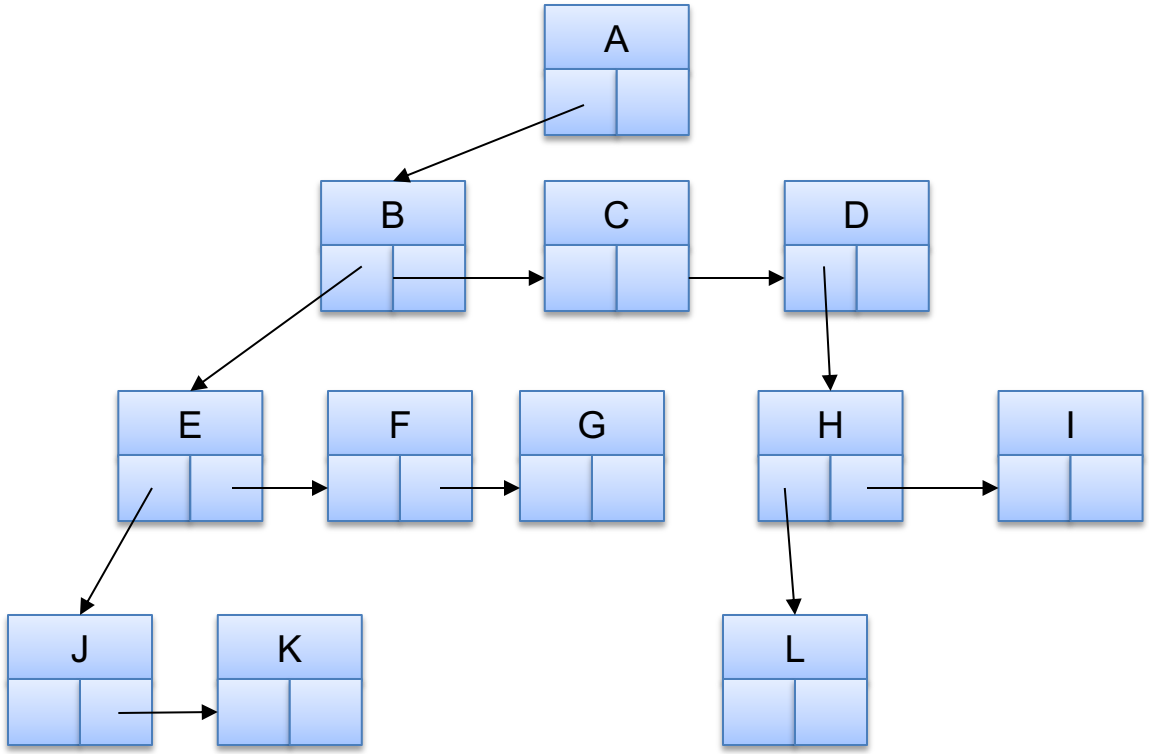
left child of J

Left child of C

Right child of C

3.2. Tree representation : Leftmost-Child, Right-Sibling representation





Infix, Prefix, and Postfix Notation

represent complicated expressions using an ordered rooted tree (typically binary)

Algebraic expressions

- preorder – Polish notation
- inorder – infix notation
- postorder – reverse Polish notation

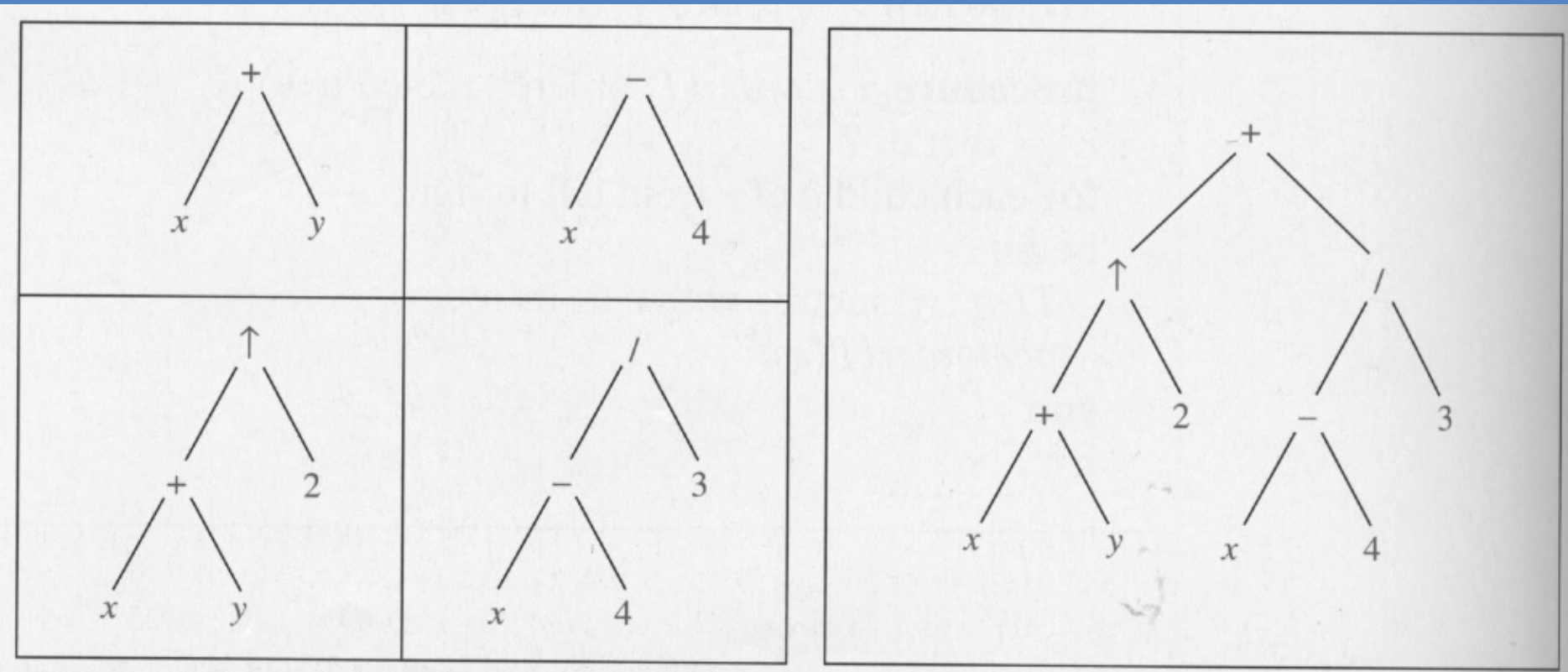


FIGURE 10 **A Binary Tree Representing $((x + y) \uparrow 2) + ((x - 4)/3)$.**

Express in:

a. Polish
b. infix
c. reverse Polish

Evaluate postfix expression 7 2 3 * - 4 ^ 9 e / +