

COMP1110/1140/6710

Week-11(A)

Tree ADT, Tree traversal and Tree applications

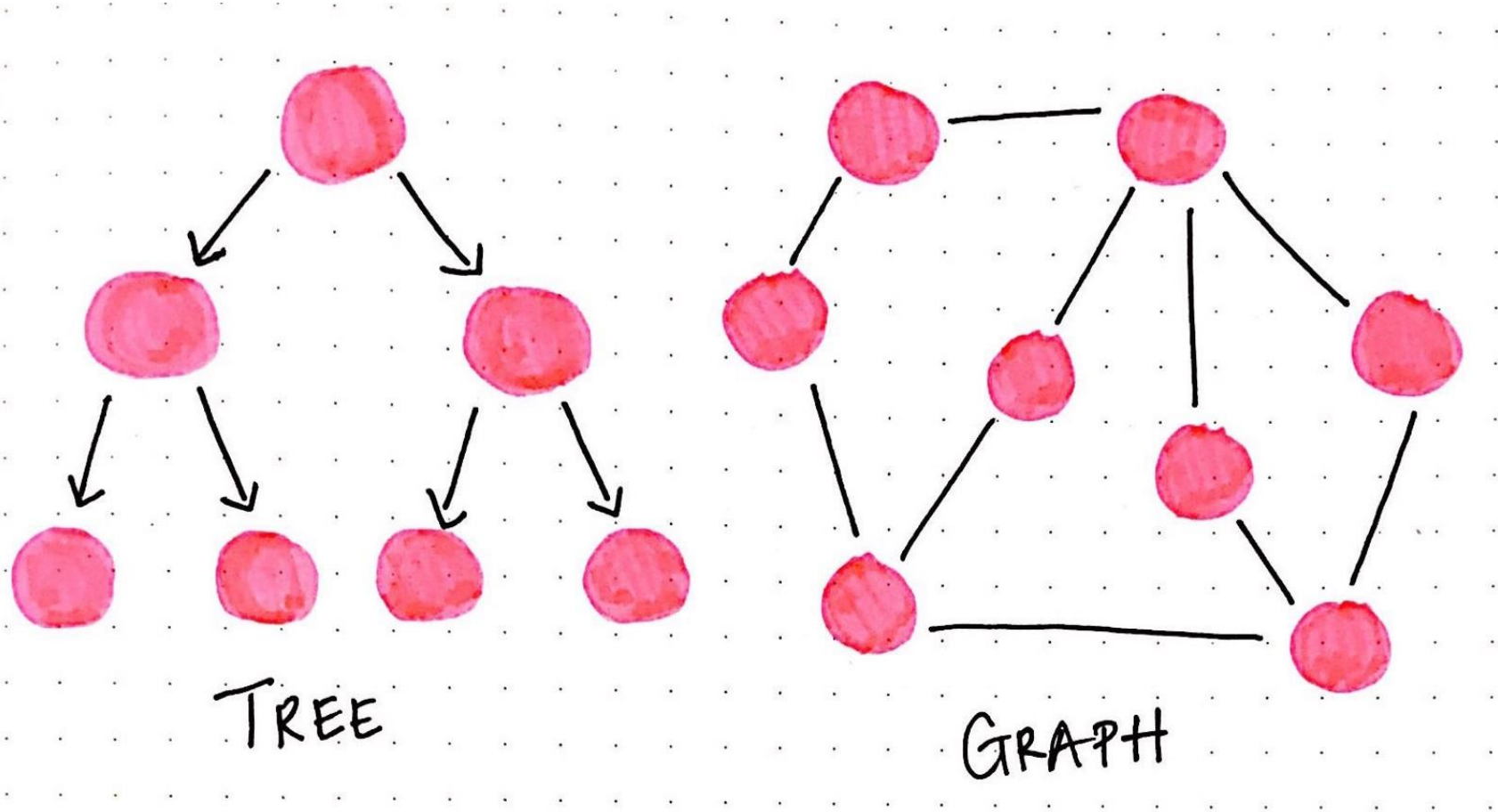


Many of the slides are adapted from online sources; they are used here for internal classroom teaching purpose only. The original copyright belong to their original authors. Please do not post it anywhere externally.

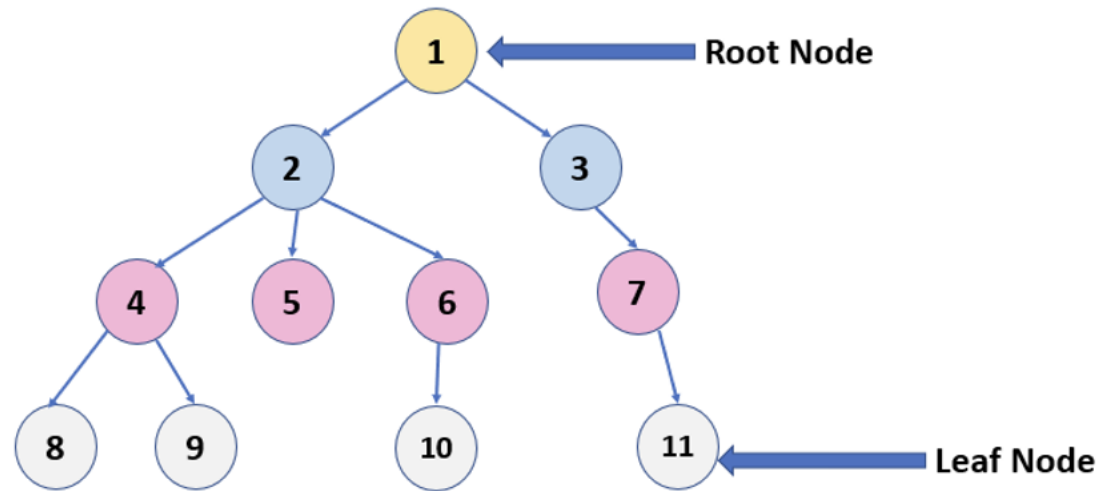
Study Plan for Week-11

- Week-11(A): (Monday)
 - **Tree ADT, and Tree Applications**
- Week-11(B): (Wednesday)
 - **Graph ADT, and Graph Algorithms**

Tree versus Graph

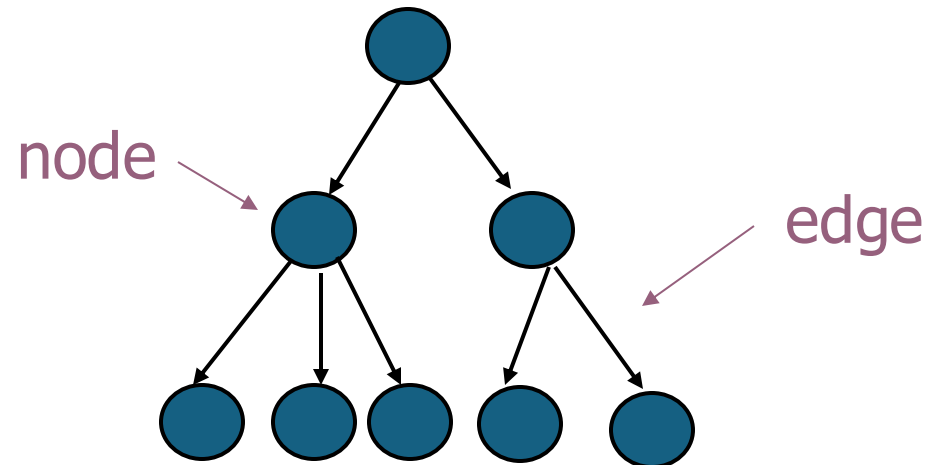


Tree ADT



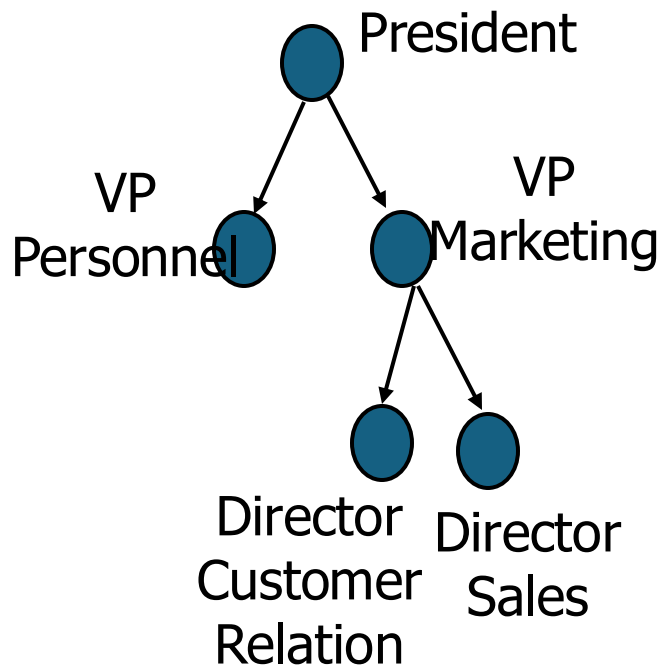
What is a tree?

- Trees are data type used to represent **hierarchical** relationship
- Each tree consists of nodes and edges
- Each node represents an object
- Each edge represents the relationship between two nodes.

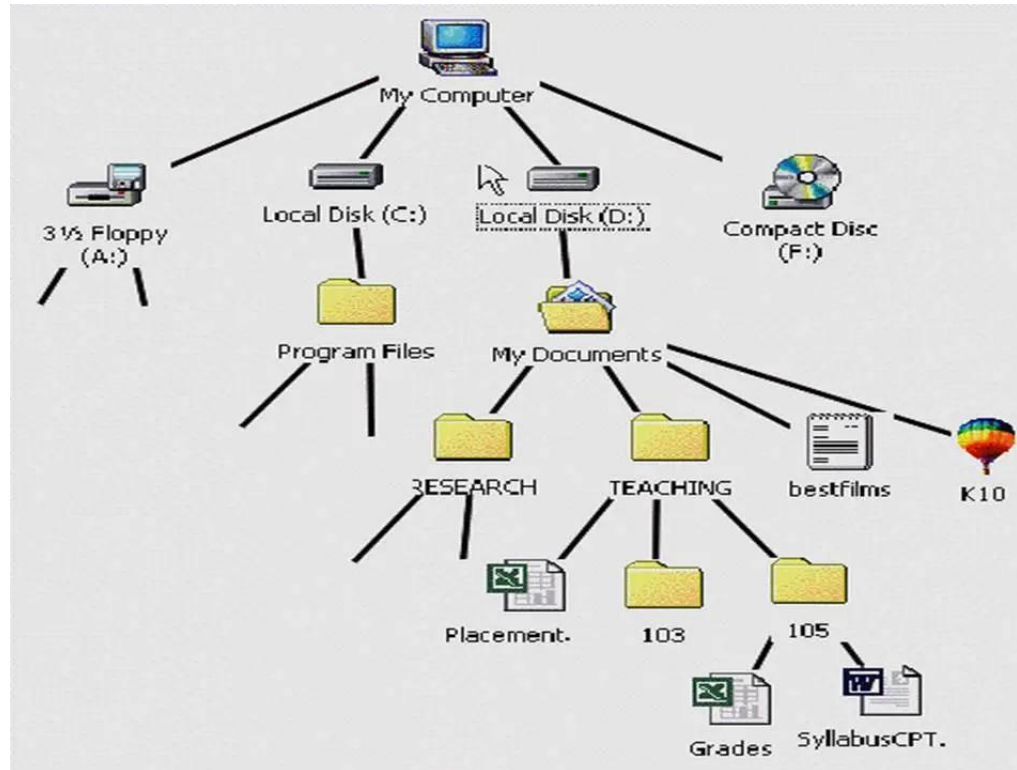


Some applications of Trees

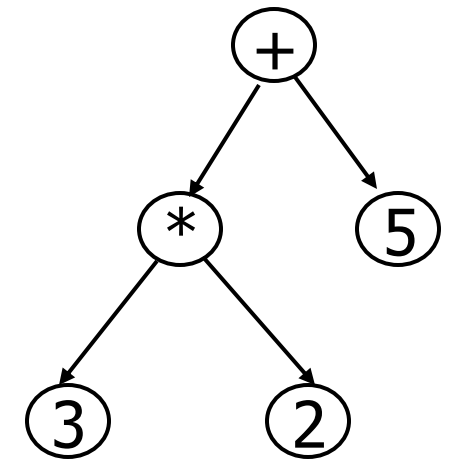
Organization Chart



File Directory tree

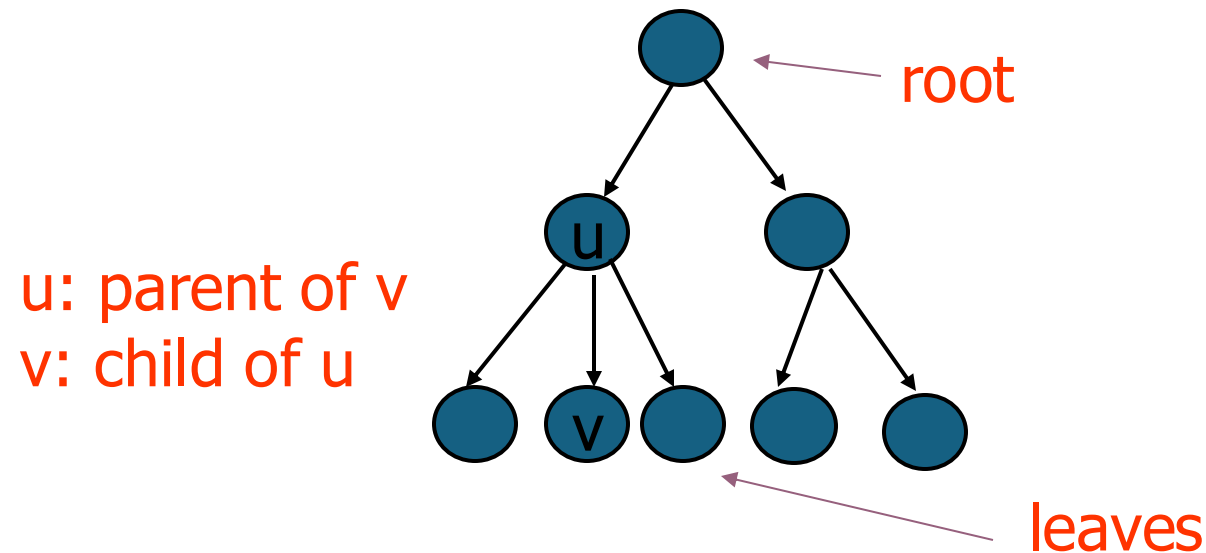


Expression Tree



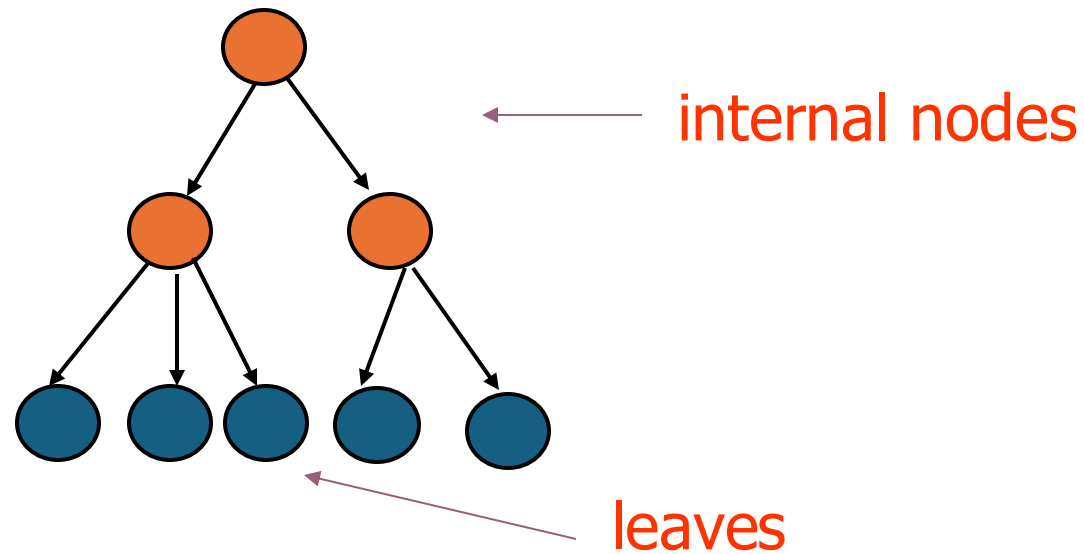
Terminology I

- For any two nodes u and v , if there is an edge pointing from u to v , u is called the **parent** of v while v is called the **child** of u . Such edge is denoted as (u, v) .
- In a tree, there is exactly one node without parent, which is called the **root**. The nodes without children are called **leaves**.



Terminology II

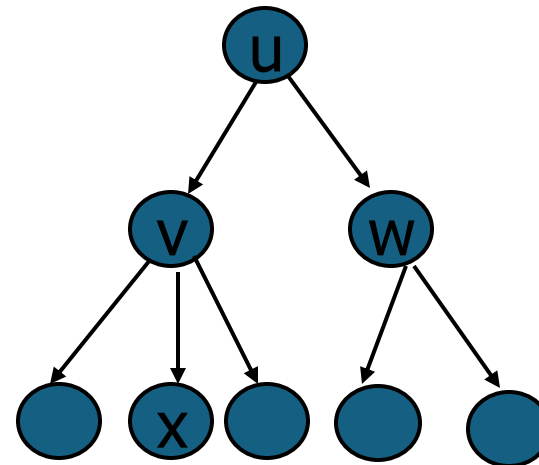
- In a tree, the nodes without children are called **leaves**. Otherwise, they are called **internal nodes**.



Terminology III

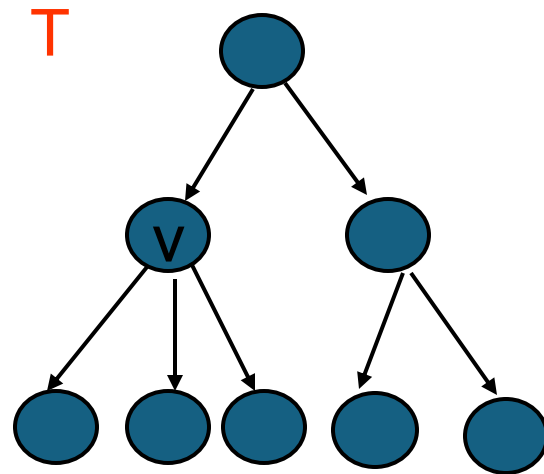
- If two nodes have the same parent, they are **siblings**.
- A node u is an **ancestor** of v if u is parent of v or parent of parent of v or ...
- A node v is a **descendent** of u if v is child of u or child of child of u or ...

v and w are siblings
 u and v are ancestors of x
 v and x are descendants of u

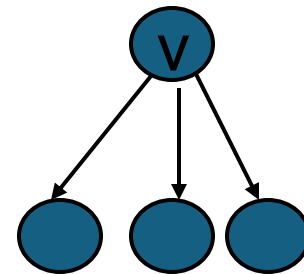


Terminology IV

- A **subtree** is any node together with all its descendants.

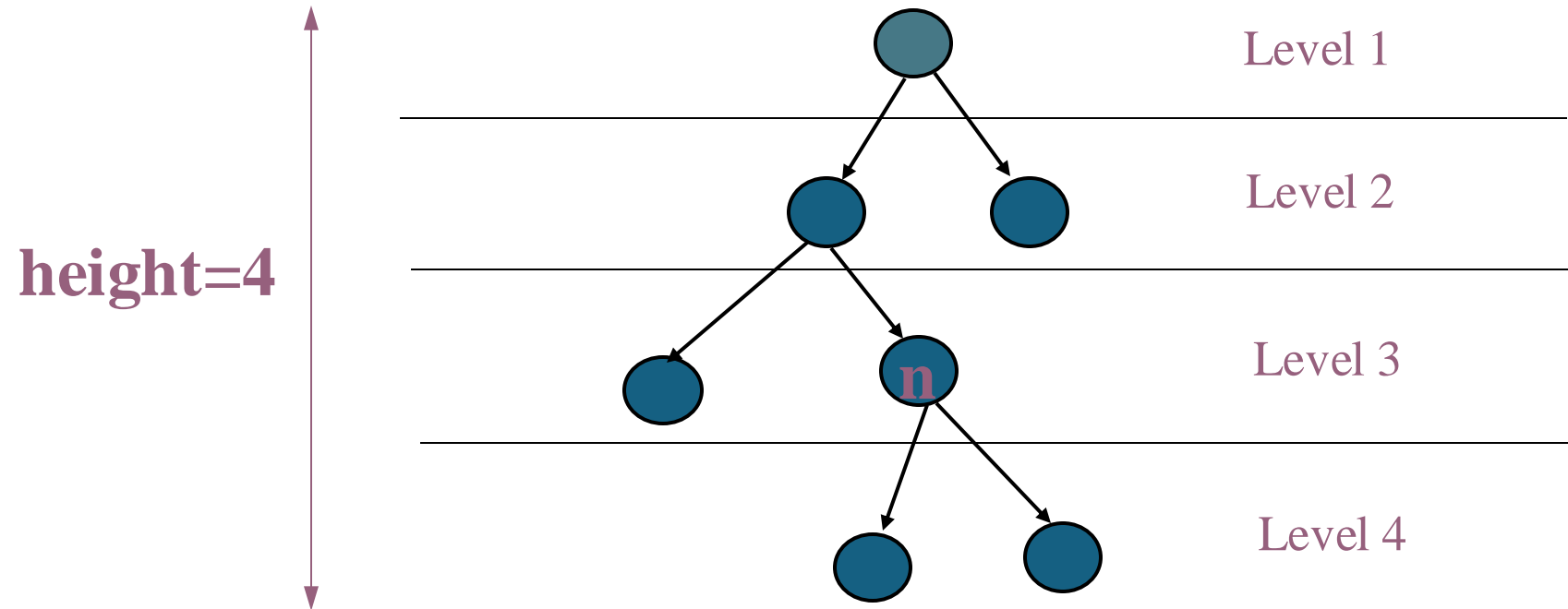


A subtree of T



Terminology V

- **Level of a node n:** number of nodes on the path from root to node n
- **Height of a tree:** maximum level among all of its node

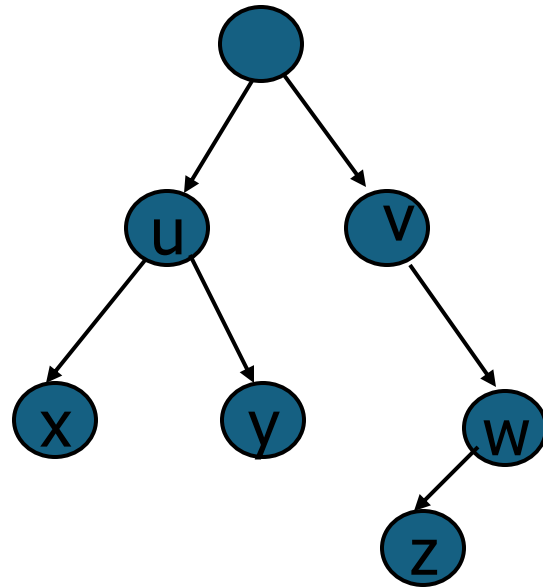


What is a tree ? -- According to Wikipedia

- In [computer science](#), a **tree** is a widely used ADT that represents a hierarchical [tree structure](#) with a set of connected [nodes](#).
- Each node in the tree can be connected to many children (depending on the type of tree), but must be connected to exactly one parent, except for the *root* node, which has no parent (i.e., the root node as the top-most node in the tree hierarchy).
- These constraints mean there are no cycles or "loops" (no node can be its own ancestor), and also that each child can be treated like the root node of its own subtree, making [recursion](#) a useful technique for [tree traversal](#).
- **[Binary Trees](#)** are a commonly used type, which constrain the number of children for each parent to at most two.

Binary Tree

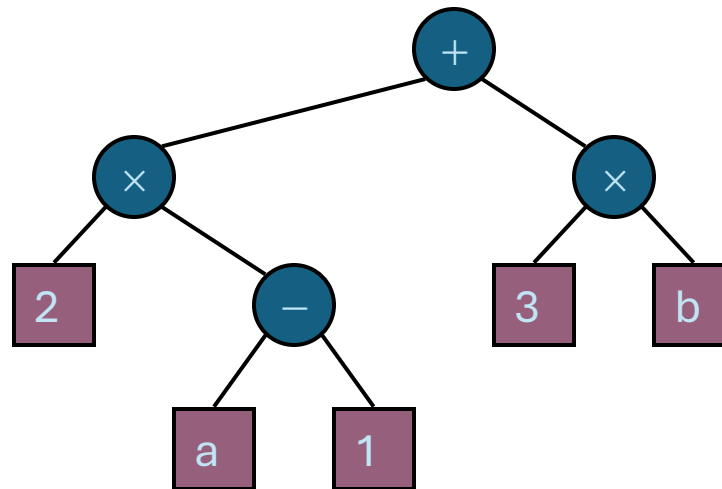
- Binary Tree: Tree in which every node has at most 2 children
- **Left child** of u: the child on the left of u
- **Right child** of u: the child on the right of u



x: left child of u
y: right child of u
w: right child of v
z: left child of w

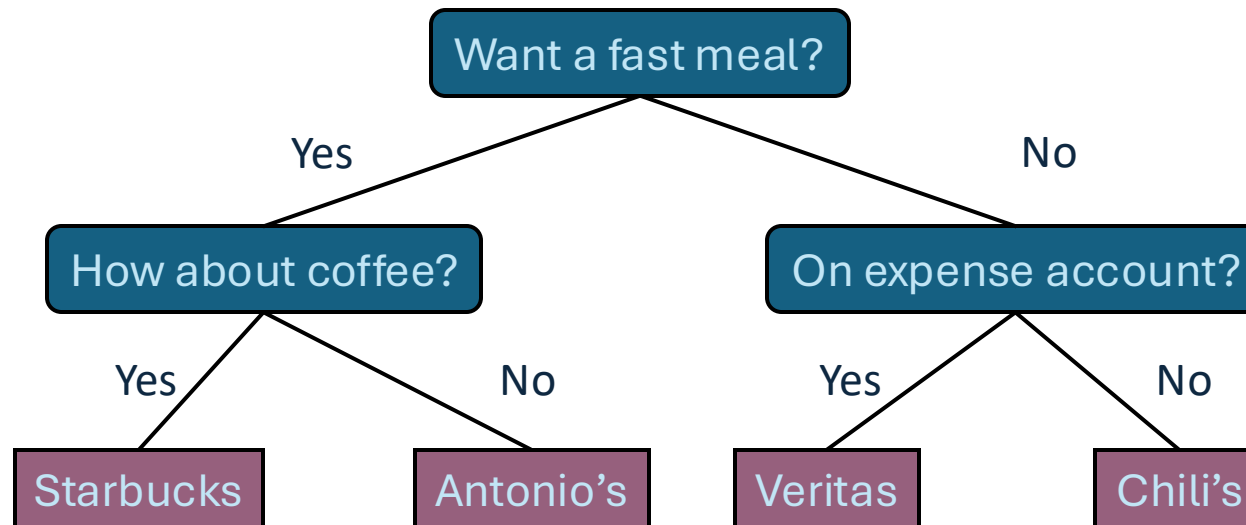
Example: Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
 - internal nodes: operators
 - leaves: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$



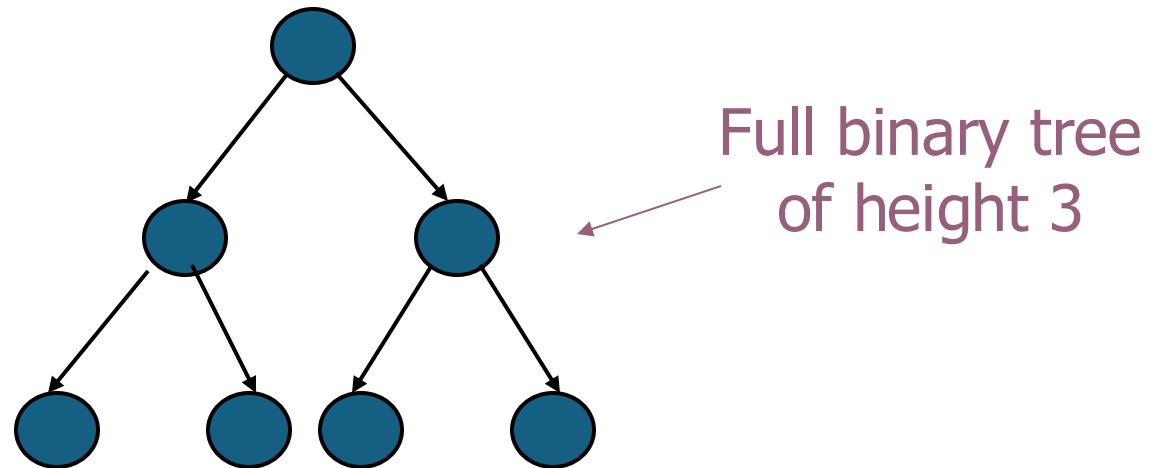
Example: Decision Tree

- Binary tree associated with a decision process
 - internal nodes: questions with yes/no answer
 - leaves: decisions
- Example: dining decision



Full binary tree

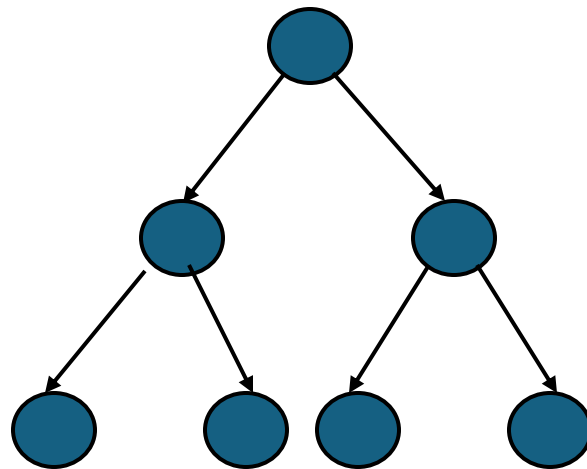
- If T is empty, T is a full binary tree of height 0.
- If T is not empty and of height $h > 0$, T is a full binary tree if both subtrees of the root of T are full binary trees of height $h-1$.



Property of binary tree (I)

- A full binary tree of height h has $2^h - 1$ nodes

No. of nodes $= 2^0 + 2^1 + \dots + 2^{(h-1)}$
 $= 2^h - 1$



Level 1: 2^0 nodes

Level 2: 2^1 nodes

Level 3: 2^2 nodes

Property of binary tree (II)

- Consider a binary tree T of height h . The number of nodes of $T \leq 2^h - 1$

Reason: you cannot have more nodes than a full binary tree of height h .

Property of binary tree (III)

- The minimum height of a binary tree with n nodes is $\log(n+1)$

By property (II), $n \leq 2^h - 1$

Thus, $2^h \geq n + 1$

That is, $h \geq \log_2(n + 1)$

Recap: Properties of Binary Trees

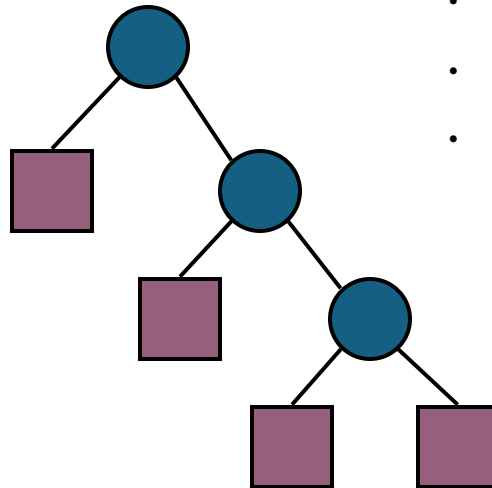
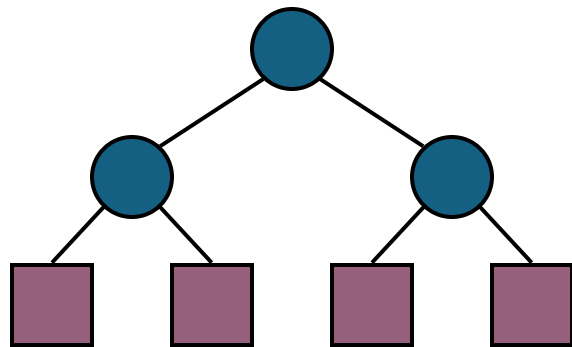
- Notation

n number of nodes

l number of leaves

i number of internal nodes

h height



- Properties:

- $l = i + 1$

- $n = 2l - 1$

- $h \leq i$

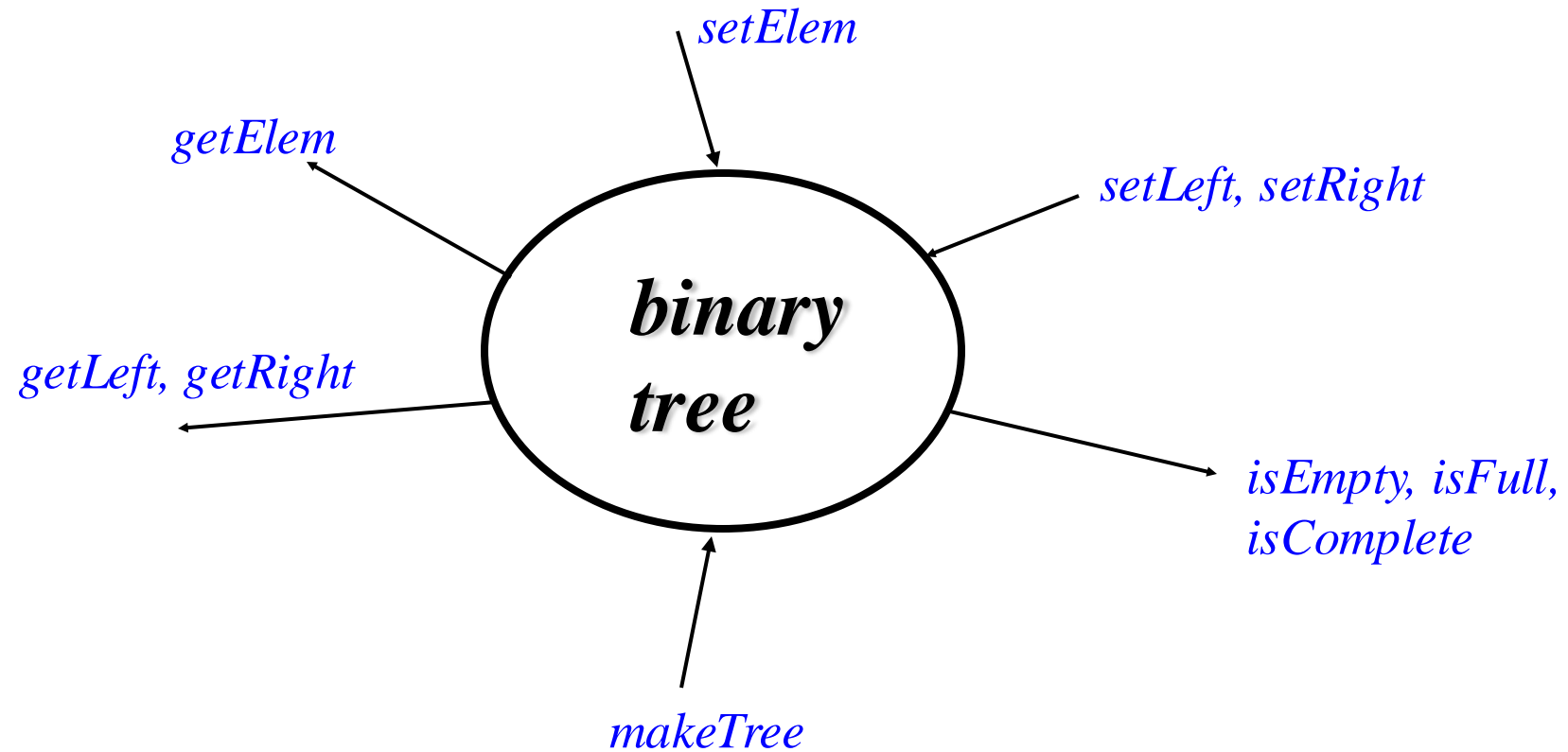
- $h \leq (n - 1)/2$

- $l \leq 2^h$

- $h \geq \log_2 l$

- $h \geq \log_2 (n + 1) - 1$

Binary Tree ADT

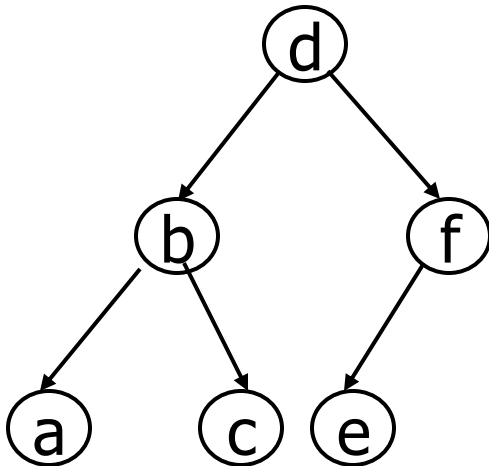


Representation (implementation) of a Binary Tree

- An array-based implementation
- A reference-based implementation

An array-based representation

-1: empty tree



nodeNum	item	leftChild	rightChild
0	d	1	2
1	b	3	4
2	f	5	-1
3	a	-1	-1
4	c	-1	-1
5	e	-1	-1
6	?	?	?
7	?	?	?
8	?	?	?
9	?	?	?
...

root

0

free

6

A reference-based Representation

NULL: empty tree

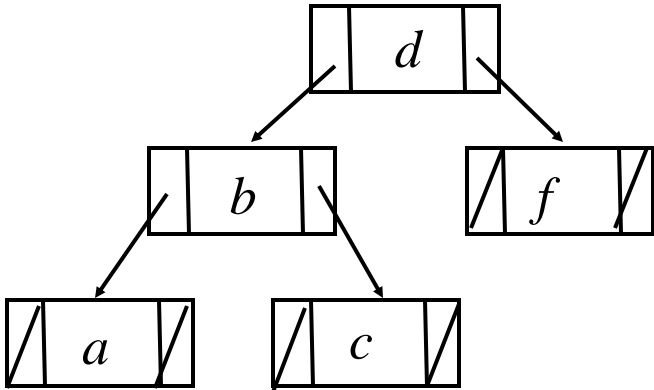
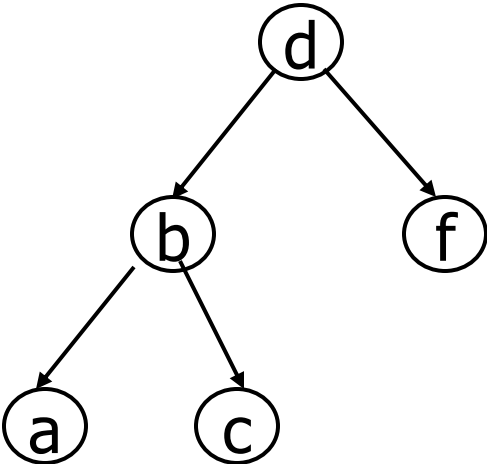
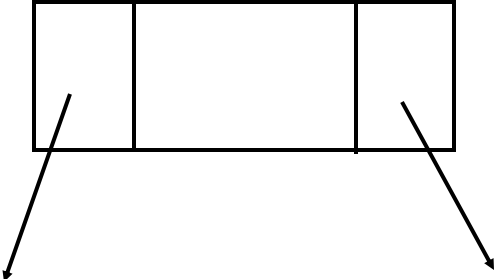
You can code this with a class of three fields:

Object element;

BinaryNode left;

BinaryNode right;

left element right



Tree Traversal

- Given a binary tree, we may like to do some operations on all nodes in a binary tree. For example, we may want to double the value in every node in a binary tree.
- To do this, we need a **tree traversal** algorithm which visits every node in the binary tree.

Ways to traverse a tree

- There are four main ways to traverse a tree:
 - Pre-order:
 - (1) visit node, (2) recursively visit left subtree, (3) recursively visit right subtree
 - In-order:
 - (1) recursively visit left subtree, (2) visit node, (3) recursively right subtree
 - Post-order:
 - (1) recursively visit left subtree, (2) recursively visit right subtree, (3) visit node
 - Level-order:
 - Traverse the nodes level by level
- In different situations, we use different traversal algorithm.

Examples for expression tree

- By pre-order, (prefix)

+ * 2 3 / 8 4

- By in-order, (infix)

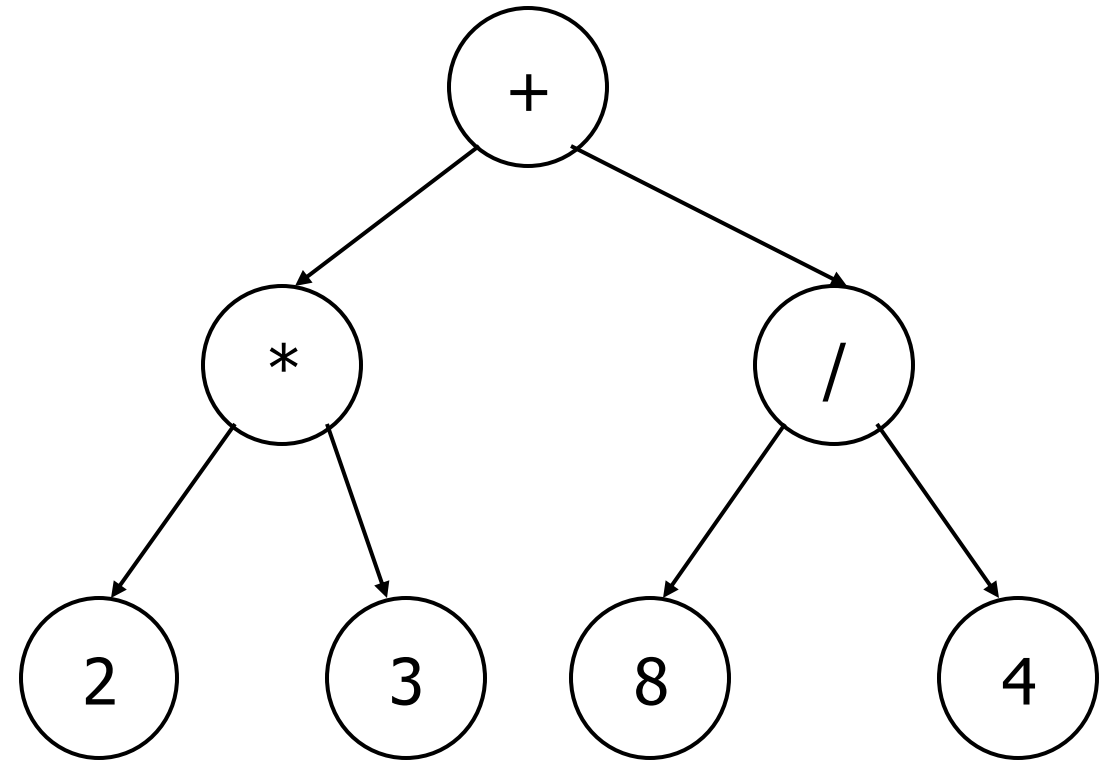
2 * 3 + 8 / 4

- By post-order, (postfix)

2 3 * 8 4 / +

- By level-order,

+ * / 2 3 8 4



- Note 1: Infix is what we read!

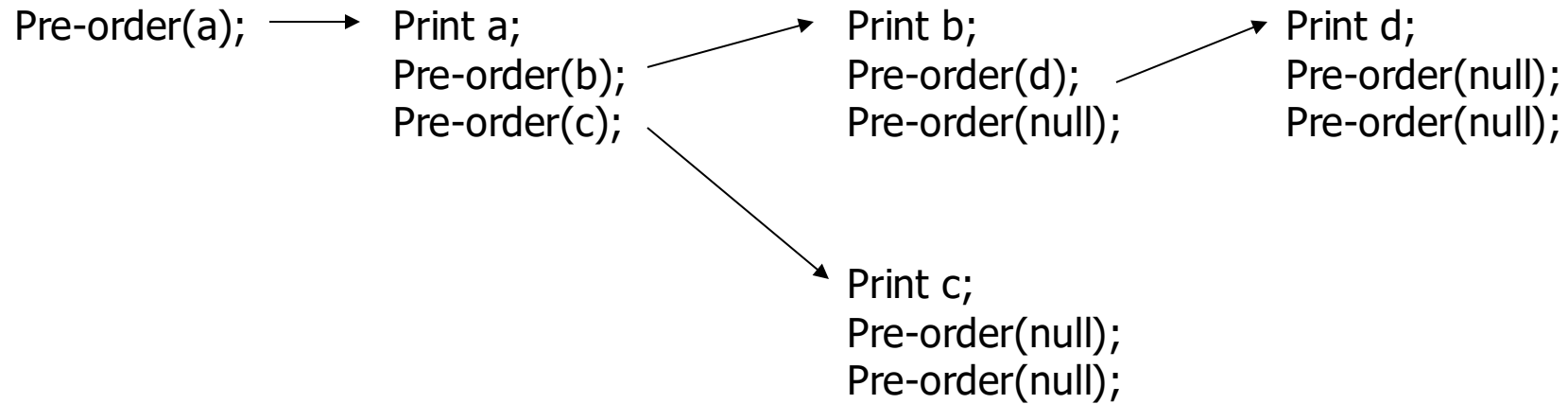
- Note 2: Postfix expression can be computed efficiently using stack ADT

Pseudo code: Pre-order

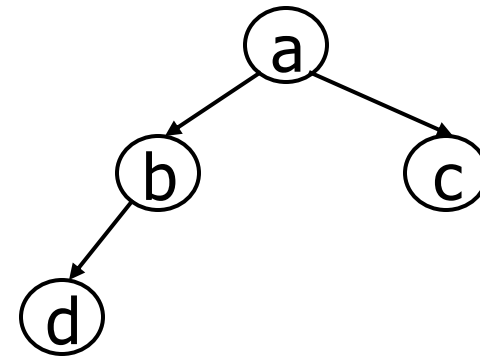
Algorithm pre-order(BTree x)

```
If (x is not empty) {  
    print x.getItem();           // you can do other things!  
    pre-order(x.getLeftChild());  
    pre-order(x.getRightChild());  
}
```

Pre-order example



a b d c



Time complexity of Pre-order Traversal

- For every node x , we will call $\text{pre-order}(x)$ one time, which performs $O(1)$ operations.
- Thus, the total time = $O(n)$.

Pseudo codes: In-order and post-order

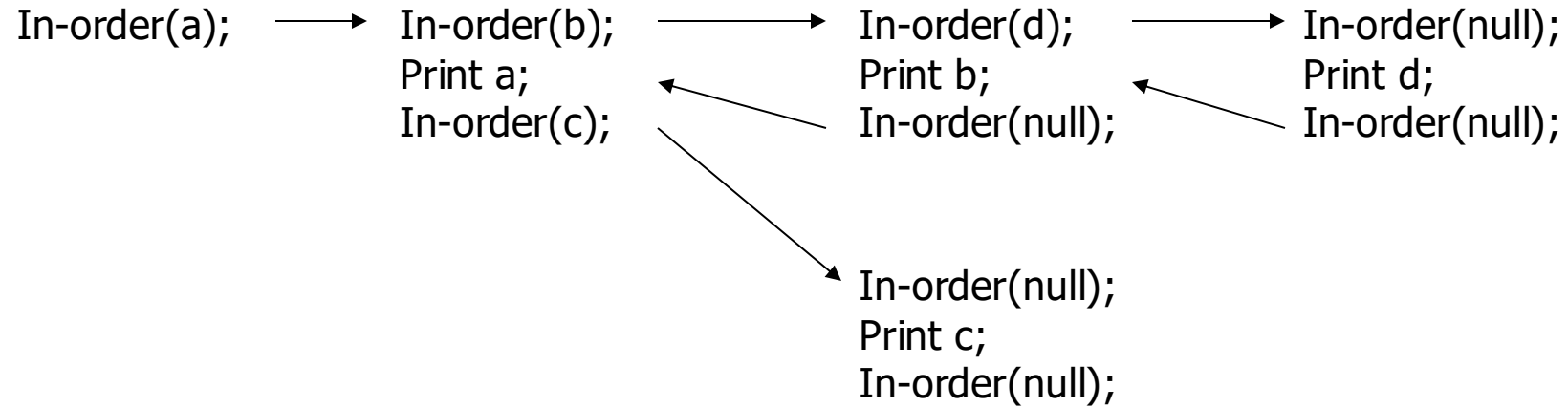
Algorithm in-order(BTree x)

```
If (x is not empty) {  
    in-order(x.getLeftChild());  
    print x.getItem(); // you can do other things!  
    in-order(x.getRightChild());  
}
```

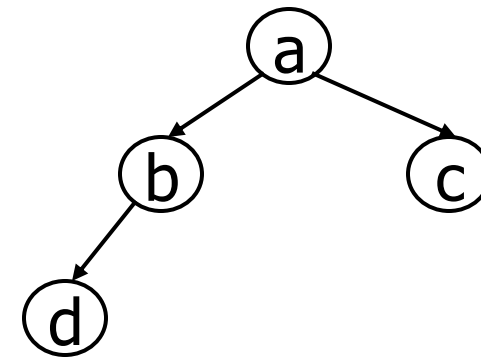
Algorithm post-order(BTree x)

```
If (x is not empty) {  
    post-order(x.getLeftChild());  
    post-order(x.getRightChild());  
    print x.getItem(); // you can do other things!  
}
```

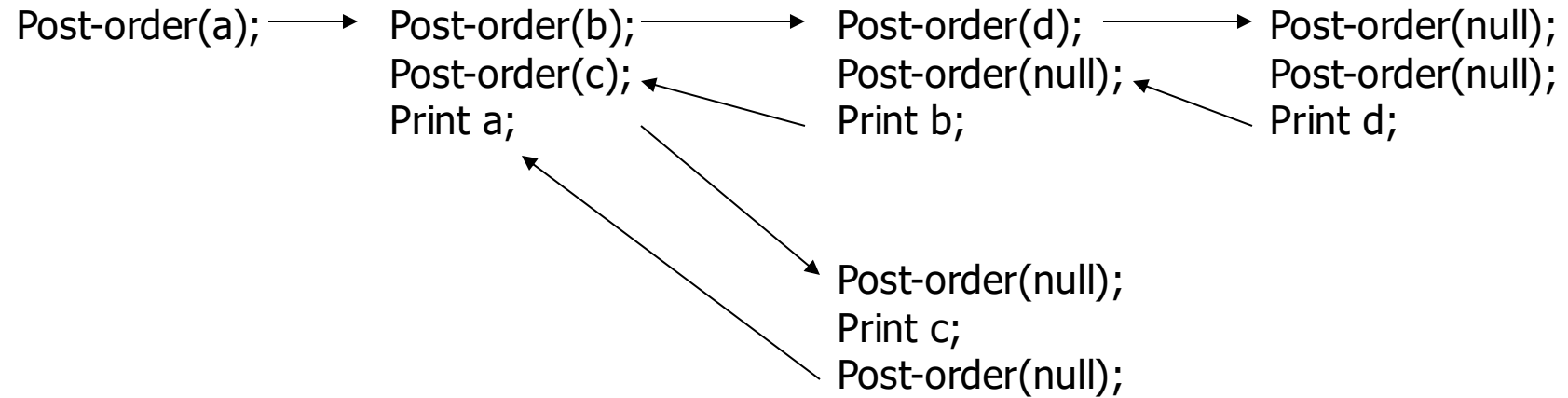
In-order example



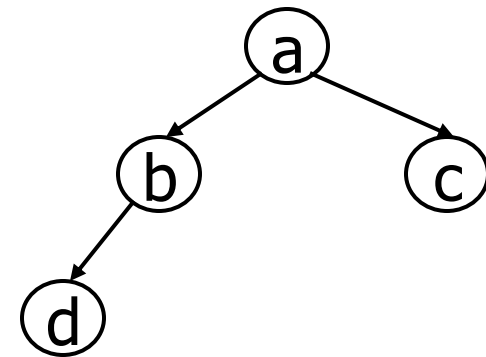
d b a c



Post-order example



d b c a



Time complexity for in-order and post-order

- Similar to pre-order traversal, the time complexity is $O(n)$.

Level-order

- Level-order traversal requires a queue!

Algorithm level-order(BTree t)

```
Queue Q = new Queue();  
BTree n;
```

```
Q.enqueue(t); // insert pointer t into Q
```

```
while (!Q.empty()){  
    n = Q.dequeue(); //remove next node from the front of Q
```

```
    if (!n.isEmpty()){  
        print n.getItem(); // you can do other things  
        Q.enqueue(n.getLeft()); // enqueue left subtree on rear of Q  
        Q.enqueue(n.getRight()); // enqueue right subtree on rear of Q  
    };  
};
```

Time complexity of Level-order traversal

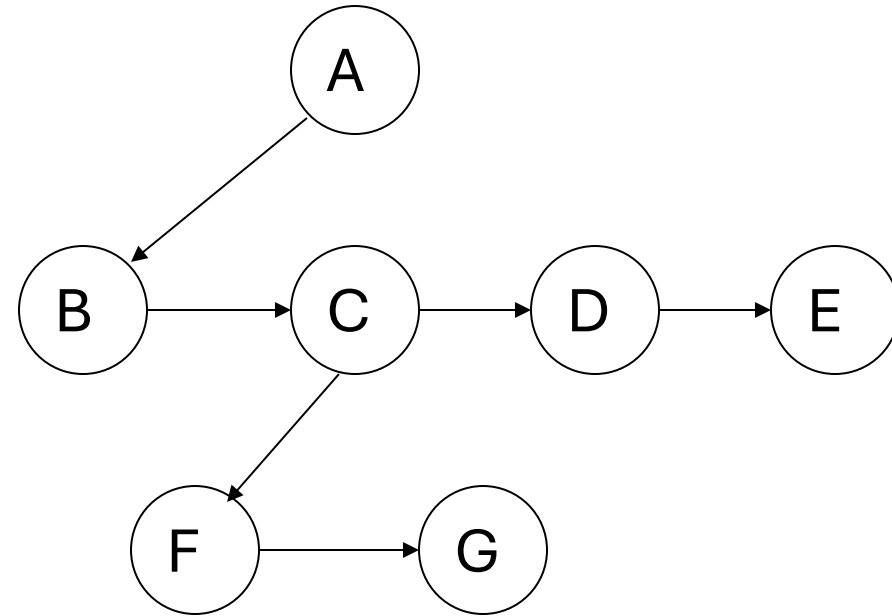

- Each node will enqueue and dequeue one time.
- For each node dequeued, it only does one print operation!
- Thus, the time complexity is $O(n)$.

General Tree ADT

- We use positions to abstract nodes
- Generic methods:
 - integer `size()`
 - boolean `isEmpty()`
 - objectIterator `elements()`
 - positionIterator `positions()`
- Accessor methods:
 - position `root()`
 - position `parent(p)`
 - positionIterator `children(p)`
- Query methods:
 - boolean `isInternal(p)`
 - boolean `isLeaf (p)`
 - boolean `isRoot(p)`
- Update methods:
 - `swapElements(p, q)`
 - object `replaceElement(p, o)`
- Additional update methods may be defined by data structures implementing the Tree ADT

General tree implementation

```
Public Class TreeNode
{
    Object    Element
    TreeNode FirstChild
    TreeNode * Nextsiblings
}
```



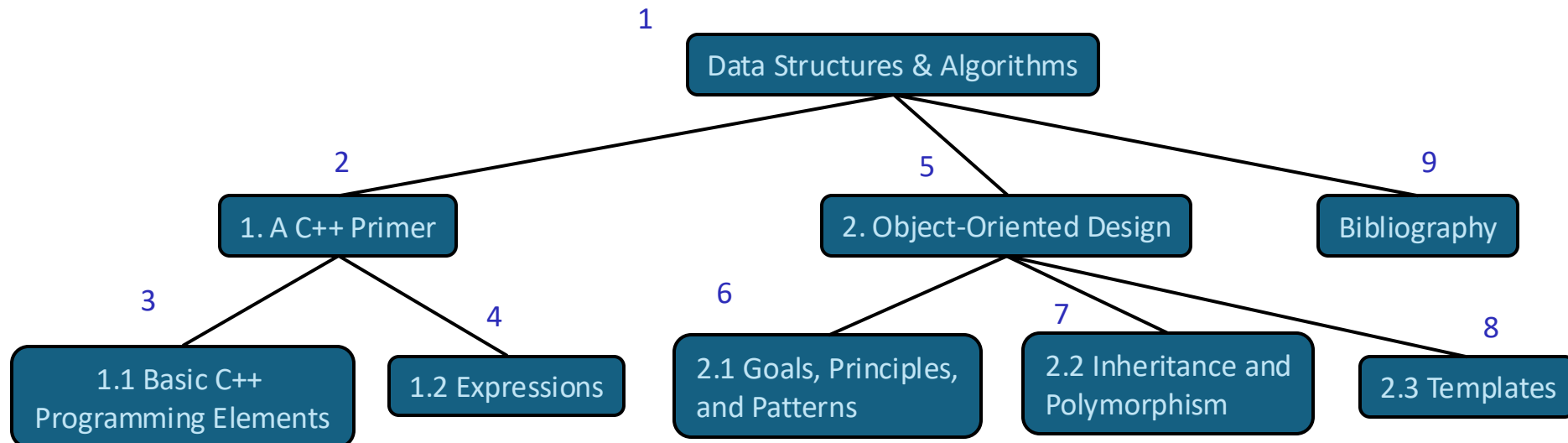
because we do not know how many children a node has in advance.

- Traversing a general tree is similar to traversing a binary tree

General tree preorder traversal

- A *traversal* visits the nodes of a tree in a systematic manner
- In a *preorder traversal*, a node is visited before its descendants
- Application: print a structured document

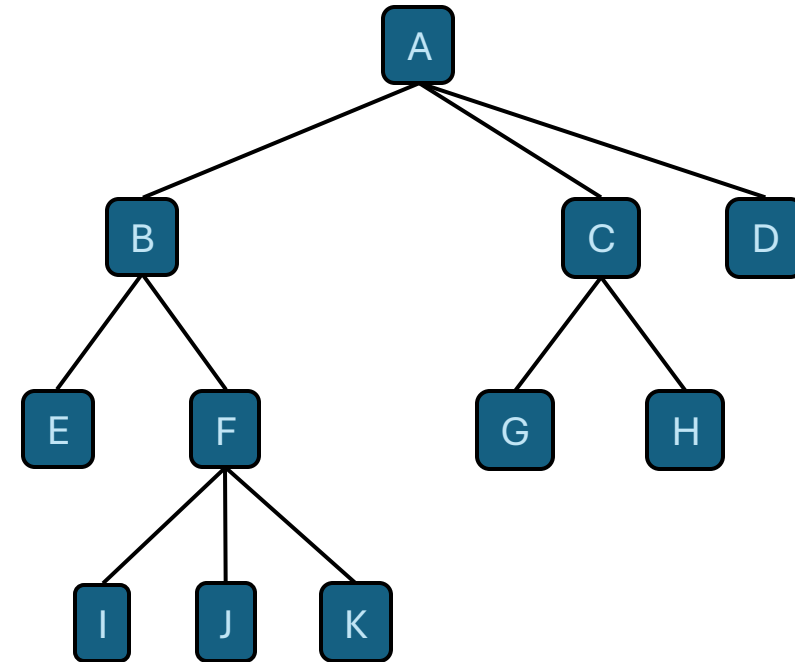
```
Algorithm preOrder(v)
  visit(v)
  for each child w of v
    preOrder(w)
```



Exercise: Preorder Traversal

- In a *preorder traversal*, a node is visited before its descendants
- List the nodes of this tree in preorder traversal order.

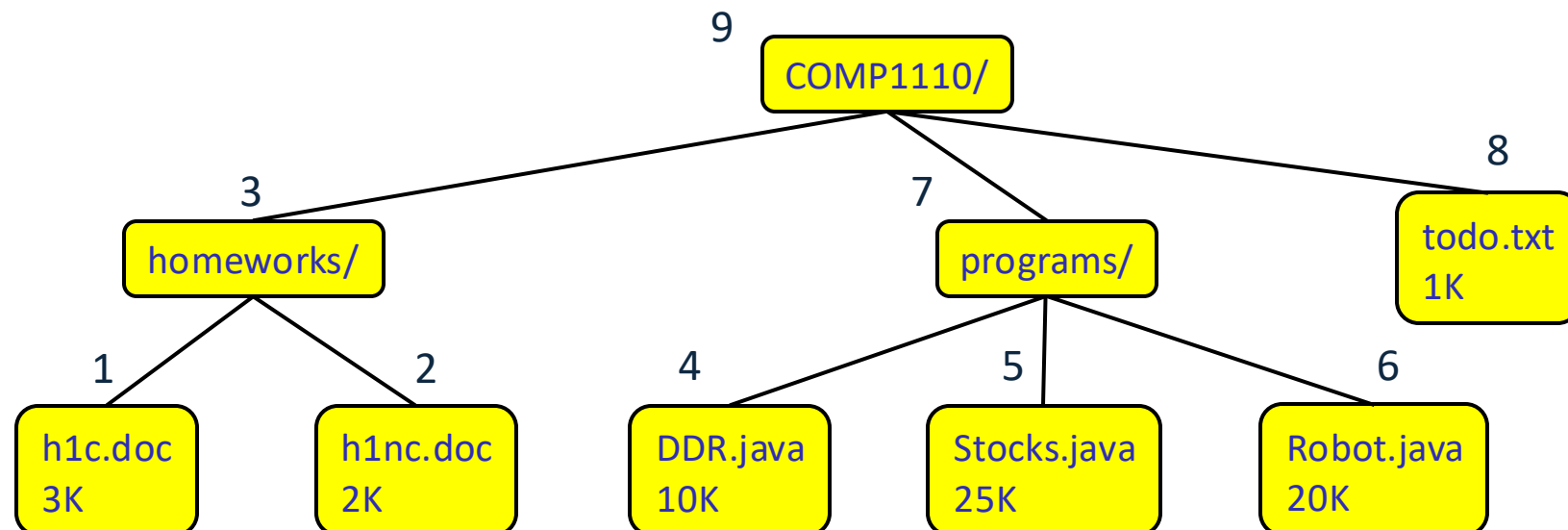
```
Algorithm preOrder(v)
  visit(v)
  for each child w of v
    preOrder(w)
```



General Tree Postorder Traversal

- In a *postorder traversal*, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

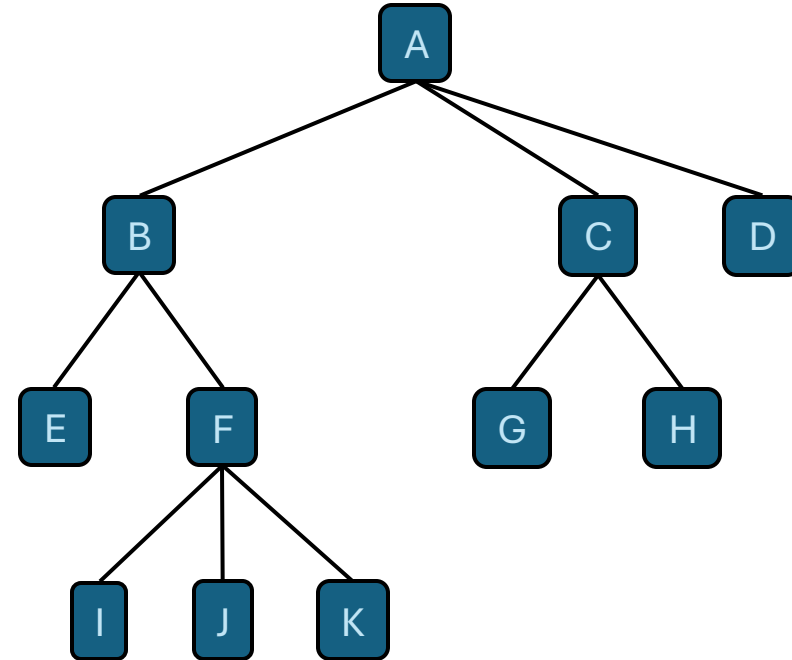
```
Algorithm postOrder(v)
  for each child w of v
    postOrder(w)
  visit(v)
```



In-class exercise: Postorder Traversal

- In a *postorder traversal*, a node is visited after its descendants
- List the nodes of this tree in postorder traversal order.

```
Algorithm postOrder(v)
  for each child w of v
    postOrder(w)
  visit(v)
```

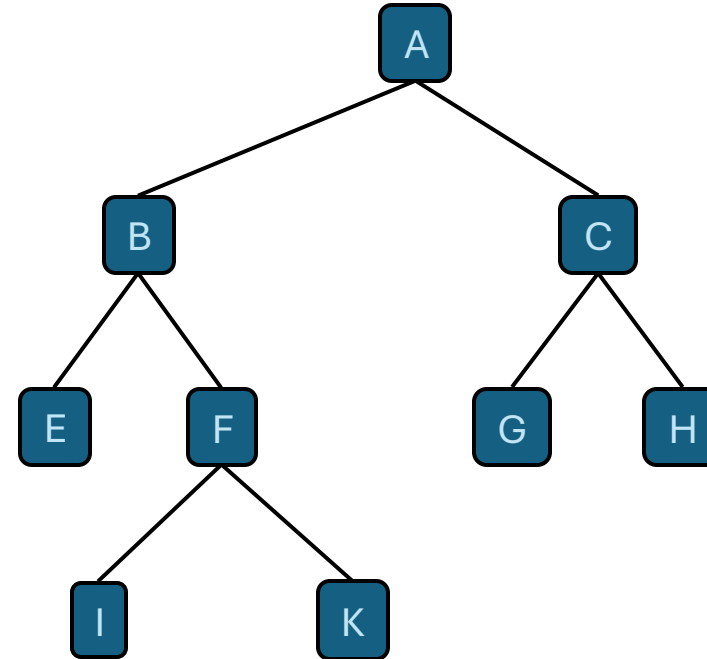


In-class Exercise:

Inorder Traversal of a binary tree

- In an *inorder traversal* a node is visited after its left subtree and before its right subtree
- List the nodes of this tree in inorder traversal order.

```
Algorithm inOrder(v)
  if isInternal(v)
    inOrder(leftChild(v))
  visit(v)
  if isInternal(v)
    inOrder(rightChild(v))
```



In-class exercise:

Preorder & InOrder Traversal of a binary tree

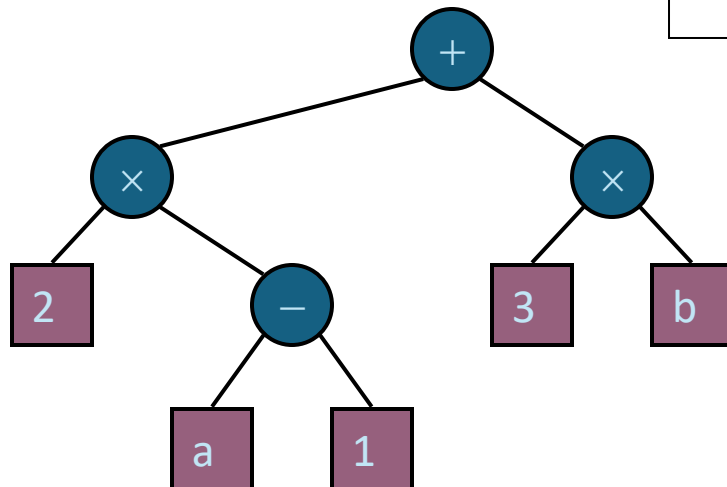
- Draw a (single) binary tree T, such that
 - Each internal node of T stores a single character
 - A preorder traversal of T yields EXAMFUN
 - An inorder traversal of T yields MAFXUEN

In-class exercise:

Print Arithmetic Expressions

- Specialization of an inorder traversal
 - print operand or operator when visiting node
 - print "(" before traversing left subtree
 - print ")" after traversing right subtree

```
Algorithm printExpression(v)
    if isInternal(v)
        print "("
        printExpression(leftChild(v))
    print (v.element())
    if isInternal(v)
        printExpression(rightChild(v))
    print (")")
```

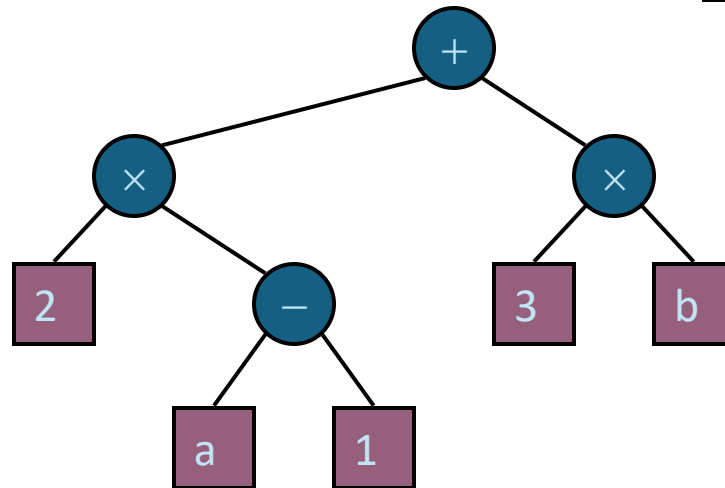


$((2 \times (a - 1)) + (3 \times b))$

In-class exercise: Evaluate Arithmetic Expressions

- [Specialization of a postorder traversal](#)
 - recursive method returning the value of a subtree
 - when visiting an internal node, combine the values of the subtrees

```
Algorithm evalExpr(v)
  if isLeaf(v)
    return v.element()
  else
    x ← evalExpr(leftChild(v))
    y ← evalExpr(rightChild(v))
    ◇ ← operator stored at v
    return x ◇ y
```

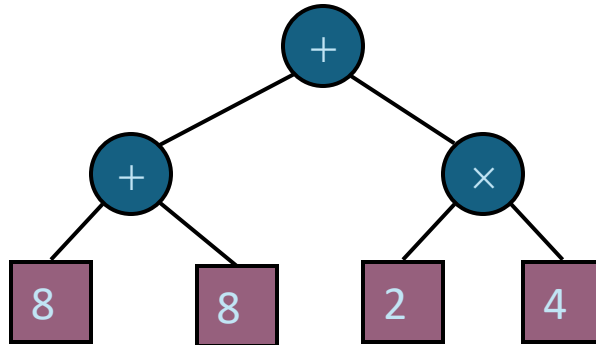


In-class Exercise: Arithmetic Expressions

- Draw an expression tree that has
 - Four leaves, storing the values 2, 4, 8, and 8
 - 3 internal nodes, storing operations +, -, *, / (operators can be used more than once, but each internal node stores only one)
 - The value of the root is 24

Solution: Arithmetic Expressions

- Draw an expression tree that has
 - Four leaves, storing the values 2, 4, 8, and 8
 - 3 internal nodes, storing operations +, -, *, / (operators can be used more than once, but each internal node stores only one)
 - The value of the root is 24



Take-home Exercise: Arithmetic Expressions

- Draw an expression tree that has
 - Four leaves, storing the values 1, 3, 4, and 8
 - 3 internal nodes, storing operations +, -, *, / (operators can be used more than once, but each internal node stores only one)
 - The value of the root is 24

Another example:

Huffman Coding

Take a break..

Coding theory

- Conversion, Encryption, Compression
- Binary coding
- Variable length coding

A	0	0	1
B	0	1	0
C	...		
D			
E			
F			

Encoding Messages

- Encode a message composed of a string of characters
- Codes used by computer systems
 - ASCII
 - uses 8 bits per character
 - can encode 256 characters
 - Unicode
 - 16 bits per character
 - can encode 65536 characters
 - includes all characters encoded by ASCII
- ASCII and Unicode are *fixed-length codes*
 - all characters represented by same number of bits

Problems

- Suppose that we want to encode a message constructed from the symbols **A**, **B**, **C**, **D**, and **E** using a fixed-length code
 - How many bits are required to encode each symbol?
 - ◆ at least **3 bits** are required
 - ◆ 2 bits are not enough (can only encode four symbols)
 - ◆ How many bits are required to encode the message **DEAACAAAABA**?
 - ◆ there are twelve symbols, each requires 3 bits;
 - ◆ $12 \cdot 3 =$ **36 bits** are required

Drawbacks of fixed-length codes

- Wasted space
 - Unicode uses twice as much space as ASCII
 - inefficient for plain-text messages containing only ASCII characters
- Same number of bits used to represent all characters
 - 'a' and 'e' occur more frequently than 'q' and 'z'
- **Potential solution:** use variable-length codes
 - variable number of bits to represent characters when frequency of occurrence is known
 - **shorter codes for characters that occur more frequently**

Advantages of variable-length codes

- The advantage of variable-length codes over fixed-length is short codes can be given to characters that occur frequently
 - on average, the length of the encoded message is less than fixed-length encoding
- **Potential problem:** how do we know where one character ends and another begins?
 - not a problem if number of bits is fixed!

A = 00
B = 01
C = 10
D = 11

0010110111001111111111

A C D B A D D D D D

Decode the following

E	0
T	11
N	100
I	1010
S	1011

11010010010101011

E	0
T	10
N	100
I	0111
S	1010

100100101010

Problem

- Design a variable-length prefix-free code such that the message **DEAACAAAAABA** can be encoded using 22 bits
- Possible solution:
 - **A** occurs eight times while **B**, **C**, **D**, and **E** each occur once
 - represent **A** with a one bit code, say 0
 - remaining codes cannot start with 0
 - represent **B** with the two bit code 10
 - remaining codes cannot start with 0 or 10
 - represent **C** with 110
 - represent **D** with 1110
 - represent **E** with 11110

Prefix(-free) property

- A code has the **prefix-free property** if no character code is the prefix (start of the code) for another character
- Example:

Symbol	Code
P	000
Q	11
R	01
S	001
T	10

01001101100010

RSTQPT

- 000 is not a prefix of 11, 01, 001, or 10
- 11 is not a prefix of 000, 01, 001, or 10 ...

Code without prefix-free property

- The following code does **not** have prefix-free property

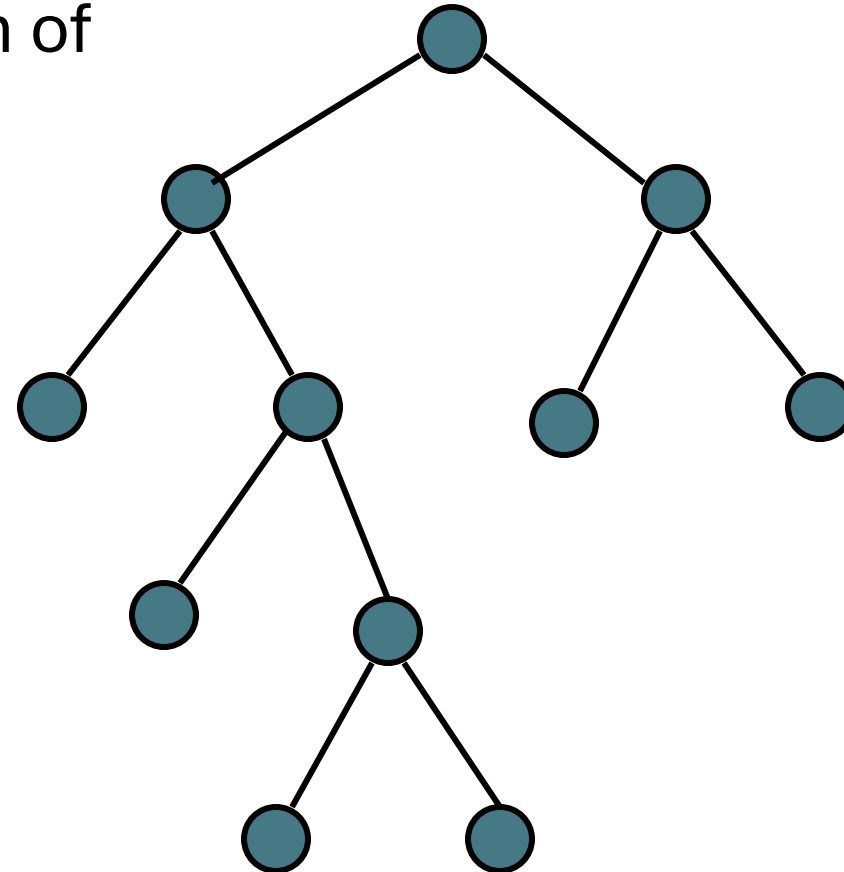
Symbol	Code
P	0
Q	1
R	01
S	10
T	11

- The pattern **1110** can be decoded as **QQQP**, **QTP**, **QQS**, or **TS**

Prefix codes and binary trees

- Tree representation of prefix codes

A	00
B	010
C	0110
D	0111
E	10
F	11



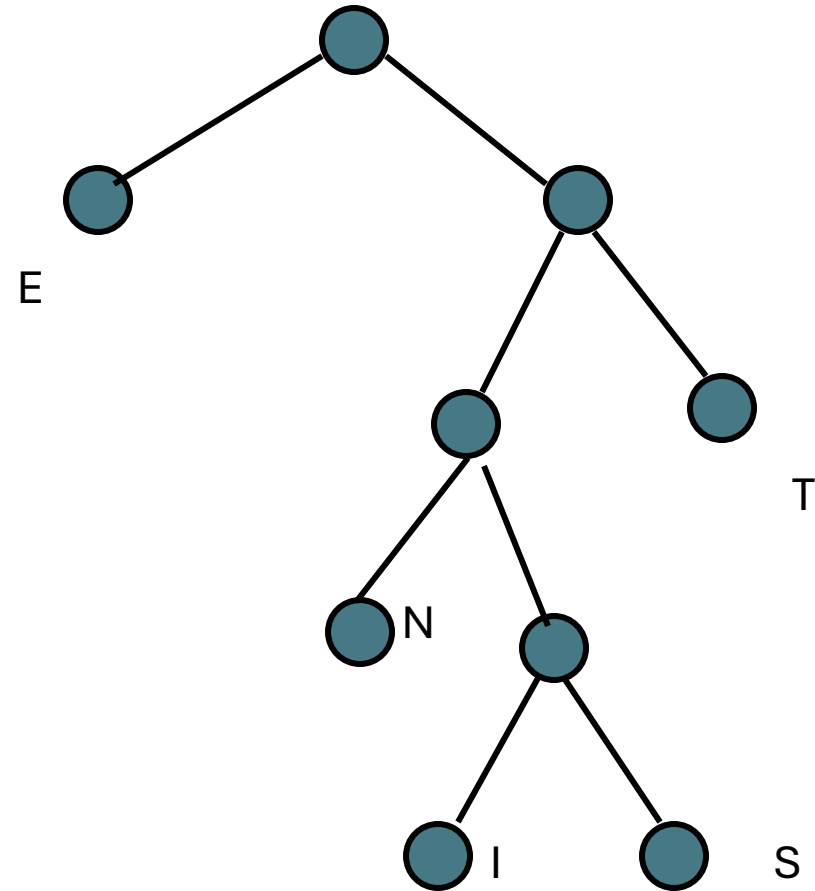
Construct the tree for the following code ?

E	0
T	11
N	100
I	1010
S	1011

Solution:

- Tree representation of prefix codes

E	0
T	11
N	100
I	1010
S	1011



Encoded message

DEAACAAAAABA

Symbol	Code
A	0
B	10
C	110
D	1110
E	11110

1110111100011000000100

22 bits

Another possible code

DEAACAAAAABA

Symbol	Code
A	0
B	100
C	101
D	1101
E	1111

1101111100101000001000

22 bits

Better code

DEAACAAAAABA

Symbol	Code
A	0
B	100
C	101
D	110
E	111

11011100101000001000

20 bits

What code to use?

- Question: Is there a variable-length code that makes the most efficient use of space?

Answer: Yes!

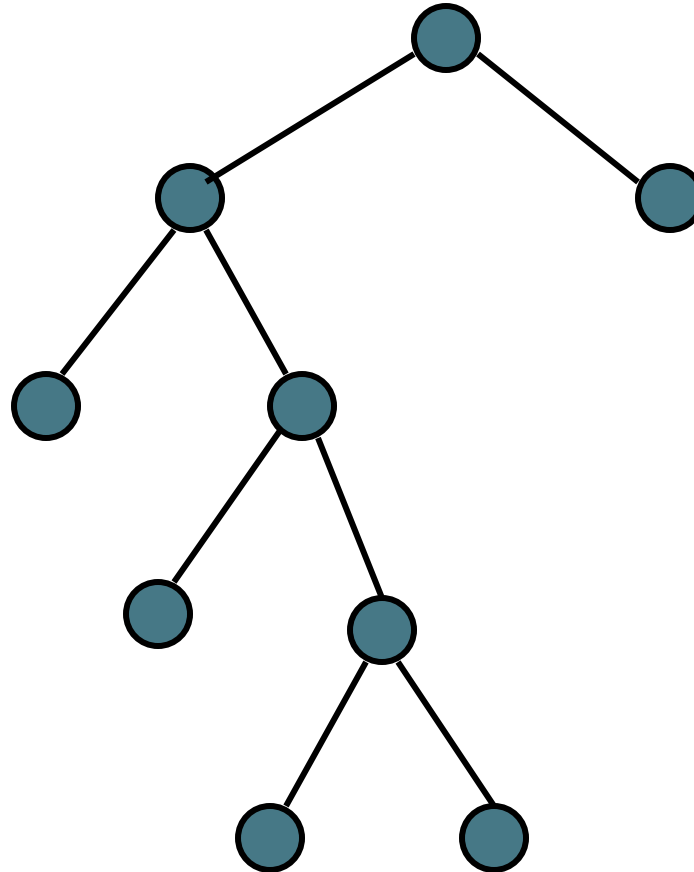
Minimum weighted length code

- Average cost
- Average leaf depth

- Huffman tree – tree with minimum weighted path length
- $C(T)$ – weighted path length

Compute average leaf depth

A	00	1/4
B	010	1/8
C	0110	1/16
D	0111	1/16
E	1	1/2



Huffman coding tree

- Binary tree
 - each leaf contains symbol (character)
 - label edge from node to left child with 0
 - label edge from node to right child with 1
- Code for any symbol obtained by following path from root to the leaf containing symbol
- Code has prefix-free property
 - leaf node cannot appear on path to another leaf
 - *note*: fixed-length codes are represented by a complete Huffman tree and clearly have the prefix-free property

Building a Huffman tree

- Find frequencies of each symbol occurring in message
- Begin with a forest of single node trees
 - each contain symbol and its frequency
- Do recursively
 - select two trees with smallest frequency at the root
 - produce a new binary tree with the selected trees as children and store the sum of their frequencies in the root
- Recursion ends when there is one tree
 - this is the Huffman coding tree

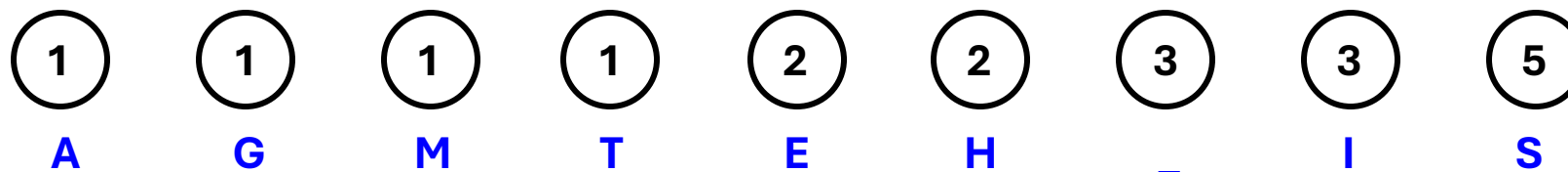
Example

- Build the Huffman coding tree for the message

“This is his message”

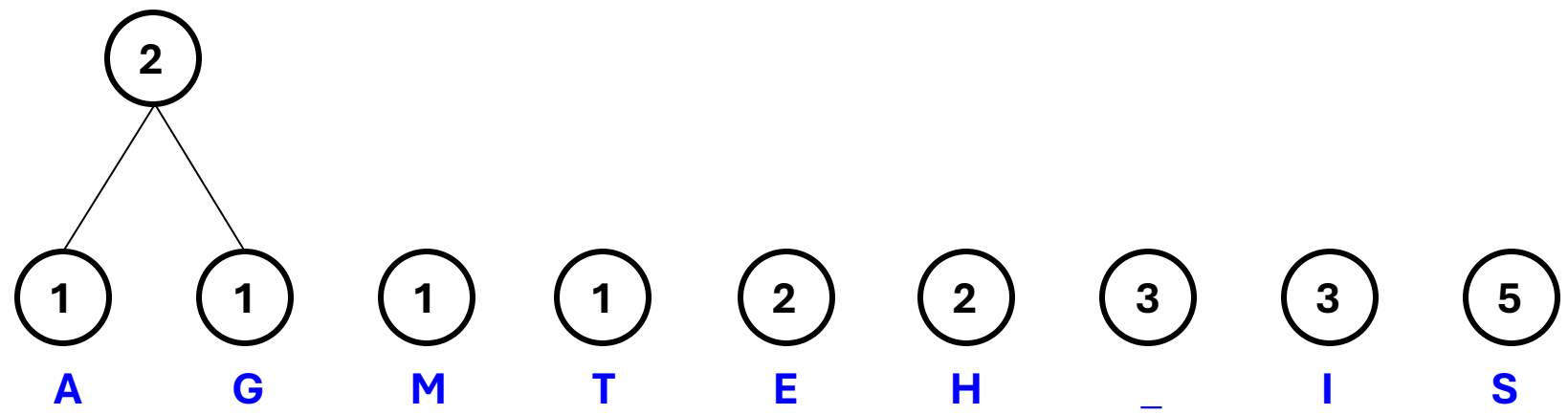
- Character frequencies

A	G	M	T	E	H	_	I	S
1	1	1	1	2	2	3	3	5

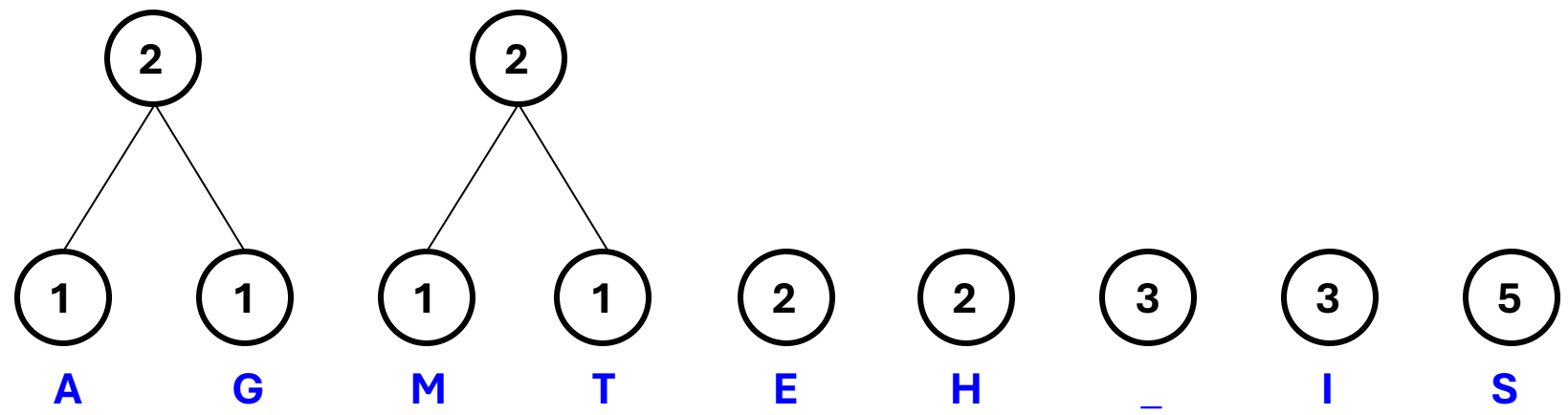


- Begin with a forest of single-node trees

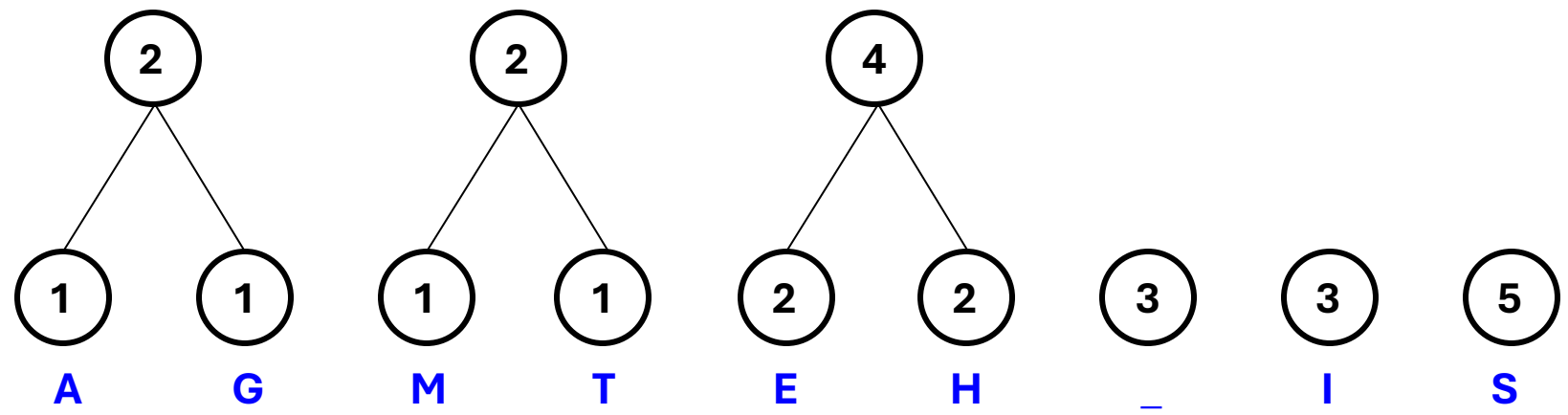
Step 1



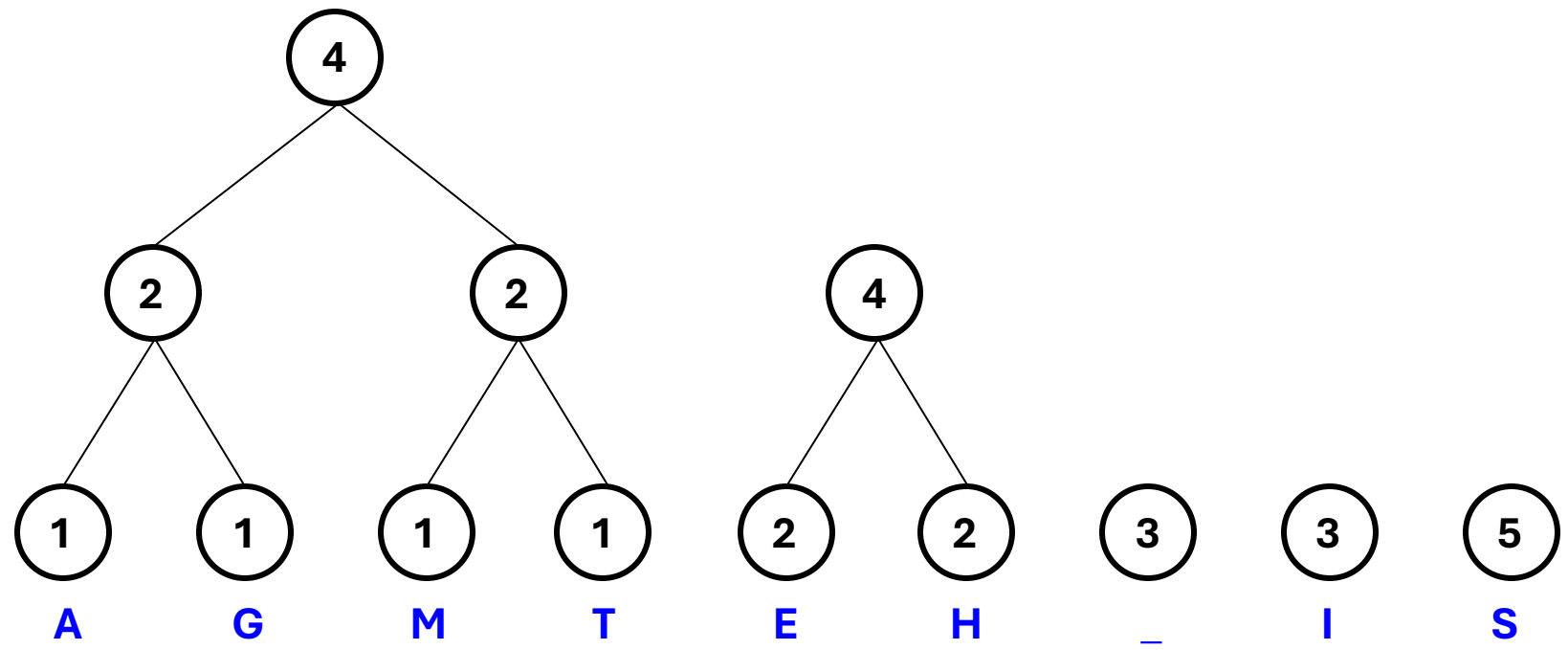
Step 2



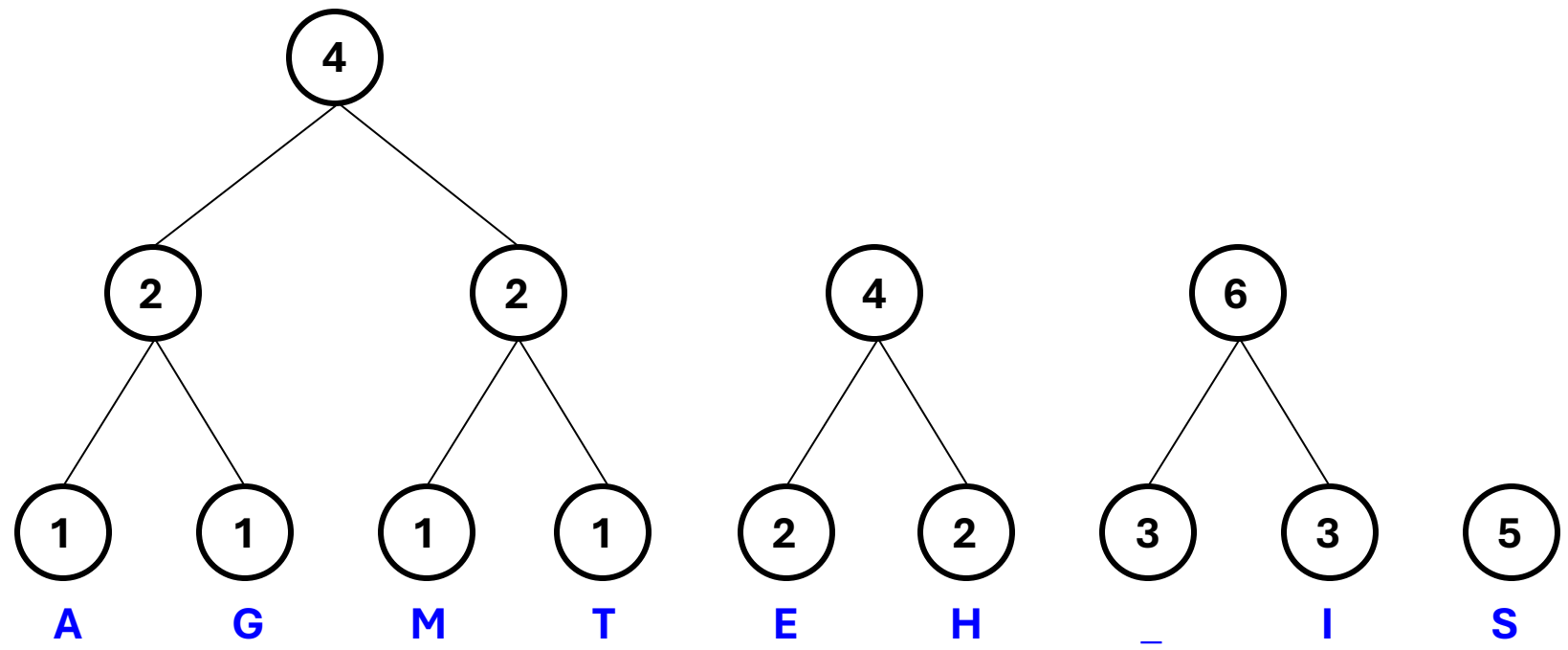
Step 3



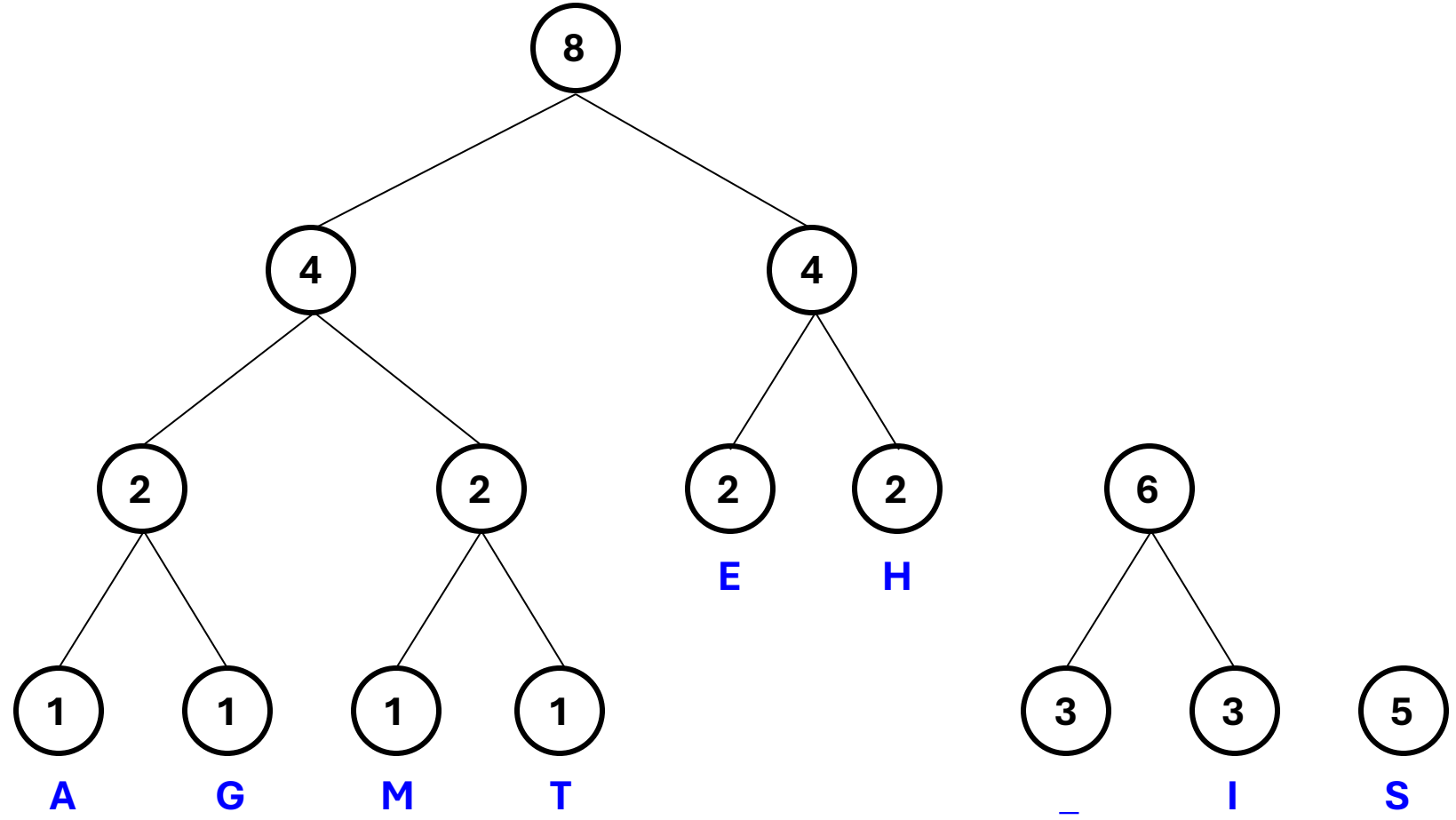
Step 4



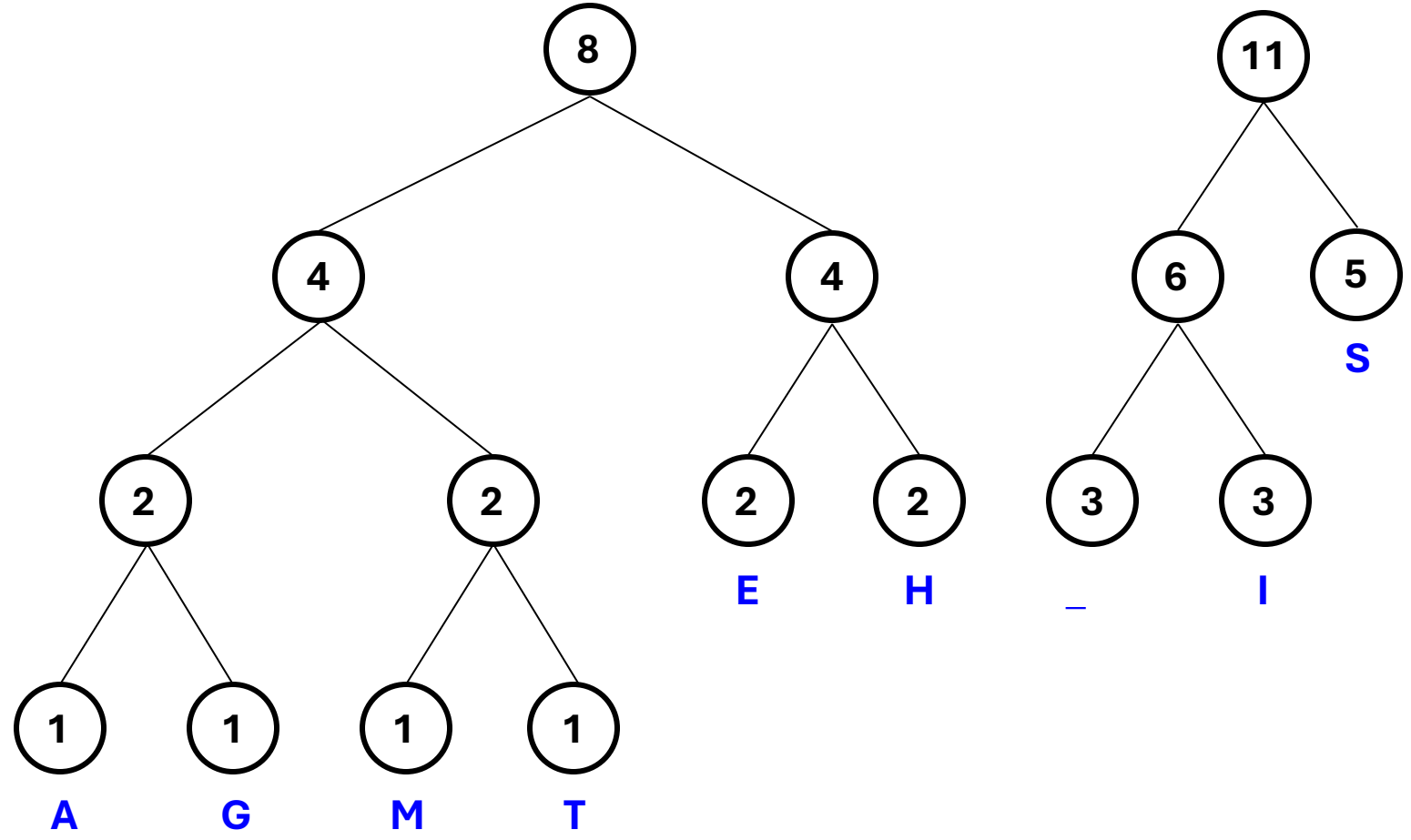
Step 5



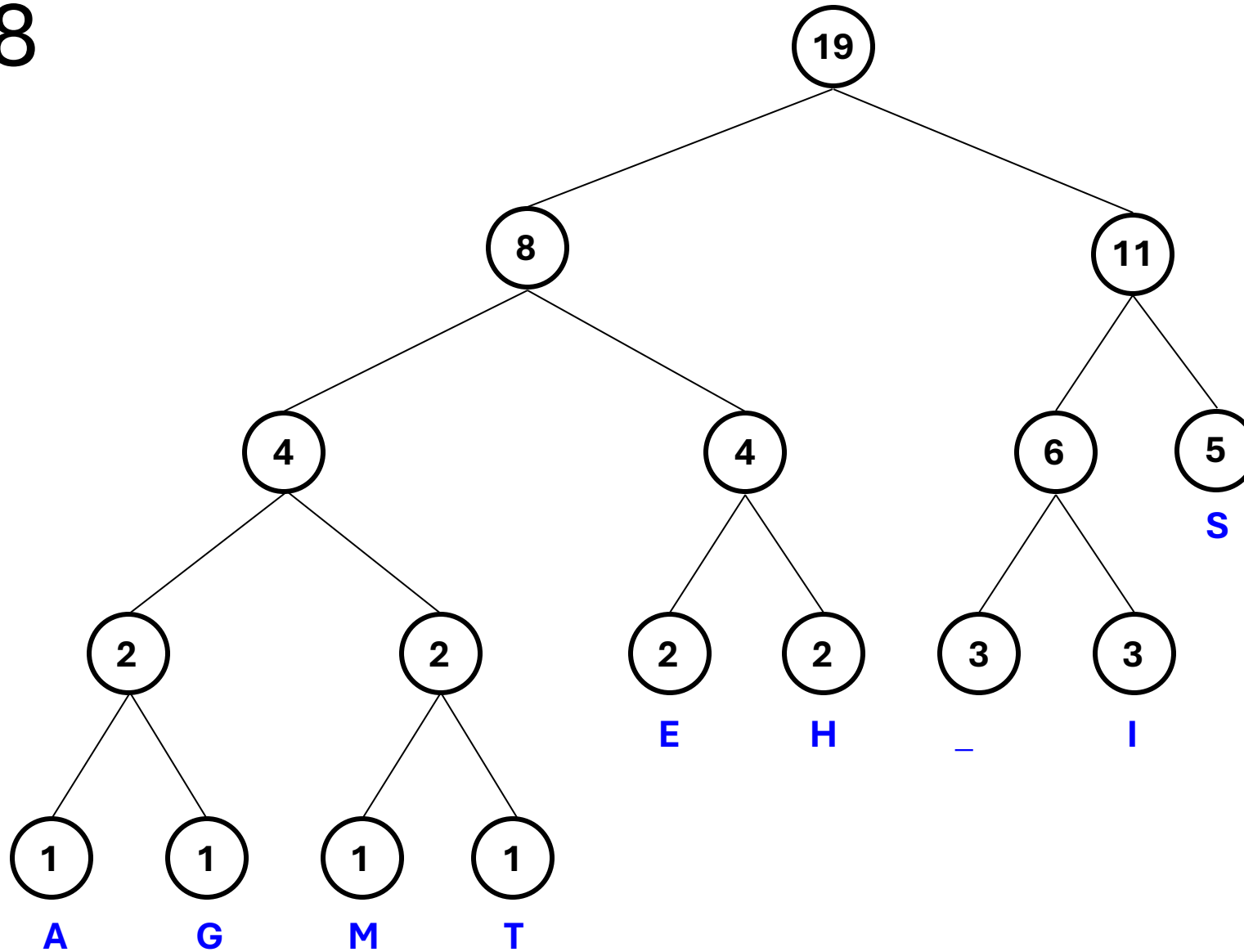
Step 6



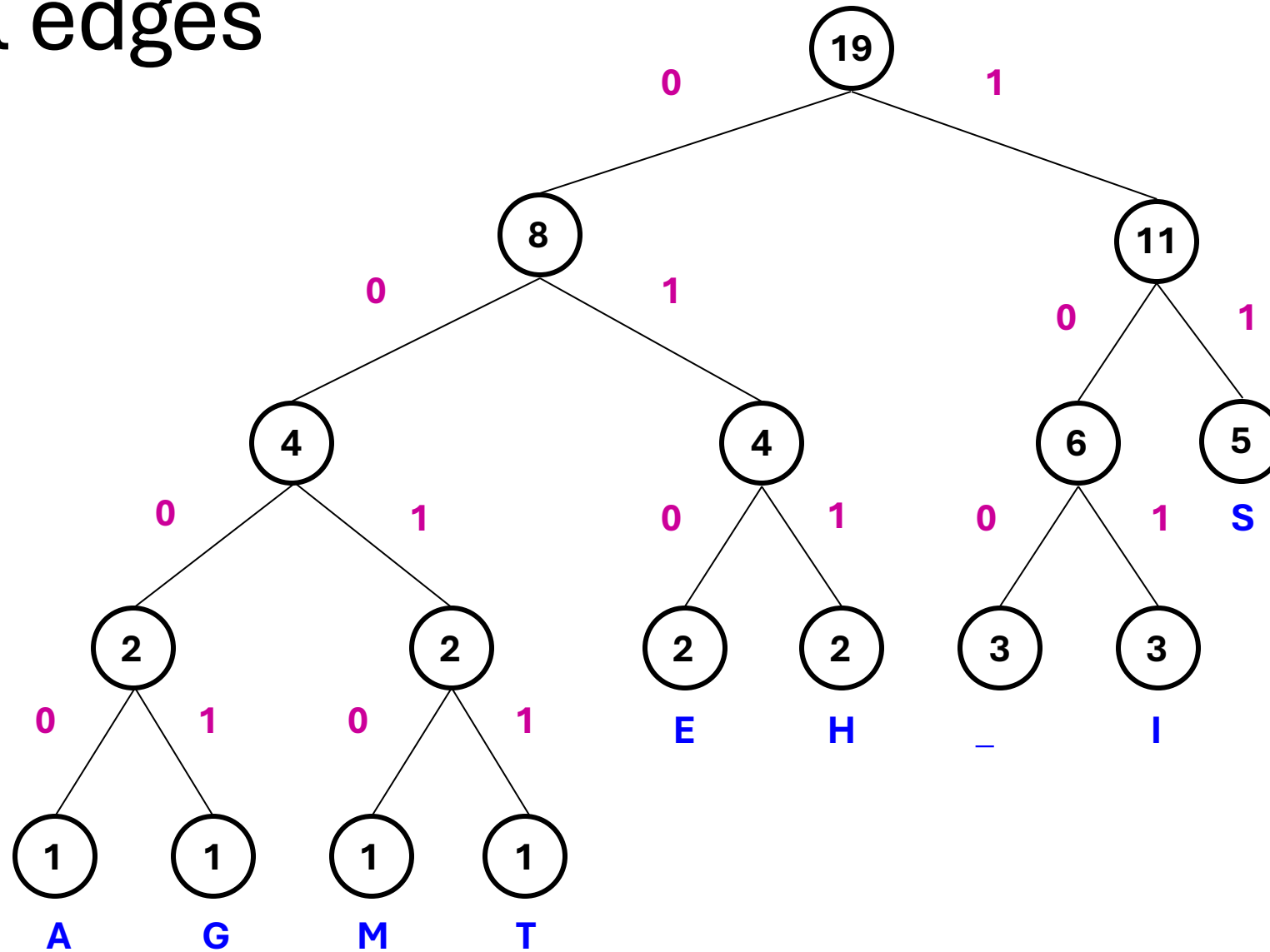
Step 7



Step 8



Label edges



Huffman code & encoded message

This is his message

S	11
E	010
H	011
-	100
I	101
A	0000
G	0001
M	0010
T	0011

001101110111100101111000111011111000010010111100000001010

Huffman Code

Greedy template. [Huffman, 1952]

Create tree **bottom-up**.

a) Make *two leaves* for **two lowest-frequency** letters y and z.

b) Recursively build tree for the rest using a **meta-letter** for yz.

Codes: Huffman Encoding

```
Huffman(S) {  
  if |S|=2 {  
    return tree with root and 2 leaves  
  } else {  
    let y and z be lowest-frequency letters in S  
    S' = S  
    remove y and z from S'  
    insert new letter  $\omega$  in S' with  $f_{\omega}=f_y+f_z$   
    T' = Huffman(S')  
    T = add two children y and z to leaf  $\omega$  from T'  
    return T  
  }  
}
```

Q. What is the *time complexity*?

Codes: Huffman Encoding

```
Huffman(S) {  
  if |S|=2 {  
    return tree with root and 2 leaves  
  } else {  
    let y and z be lowest-frequency letters in S  
    S' = S  
    remove y and z from S'  
    insert new letter ω in S' with fω=fy+fz  
    T' = Huffman(S')  
    T = add two children y and z to leaf ω from T'  
    return T  
  }  
}
```

Q. What is the time complexity?

A. $T(n) = T(n-1) + O(n) \rightarrow O(n^2)$

Q. How to implement finding *lowest-frequency letters* efficiently?

A. Use *priority queue* for S: $T(n) = T(n-1) + O(\log n) \rightarrow O(n \log n)$

Example:

Weights 4, 5, 6, 7, 11, 14, 21;

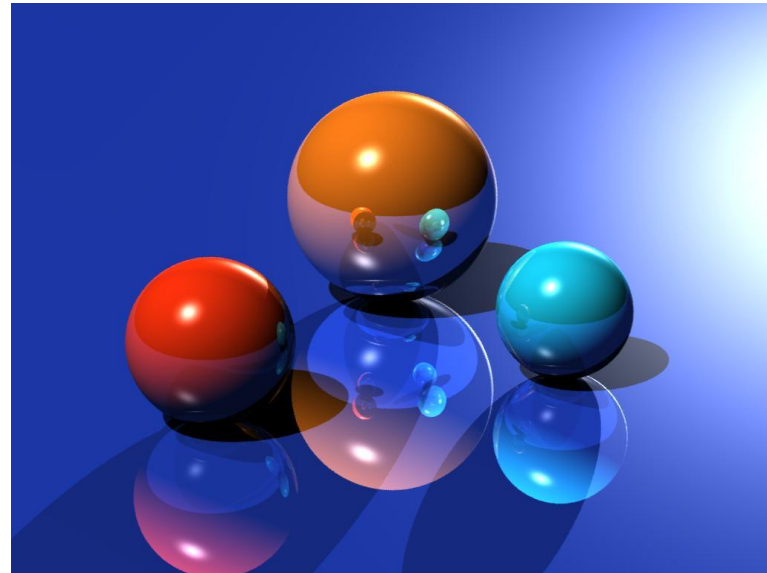
Draw a Huffman tree for the following data values
and show internal weights:

3, 5, 9, 14, 16, 35

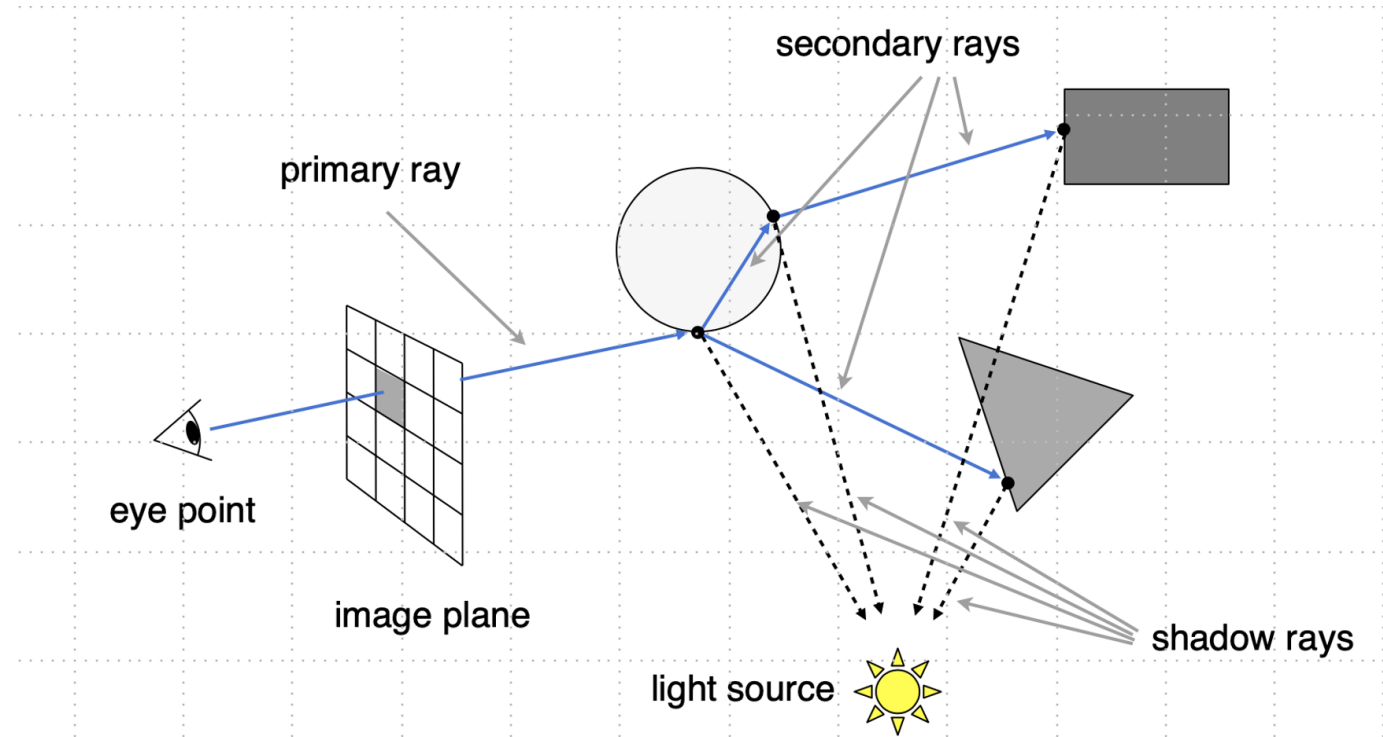
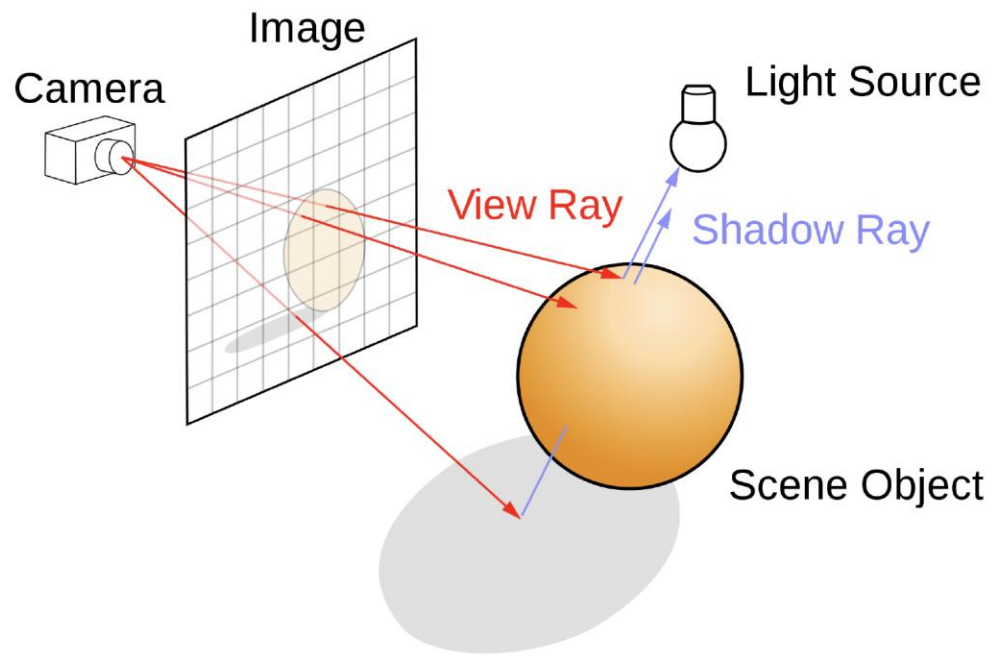
Summary

- Huffman coding is a technique used to compress files for transmission
- Uses statistical coding
 - more frequently used symbols have shorter code words
- Works well for text and fax transmissions
- An application that uses several data structures

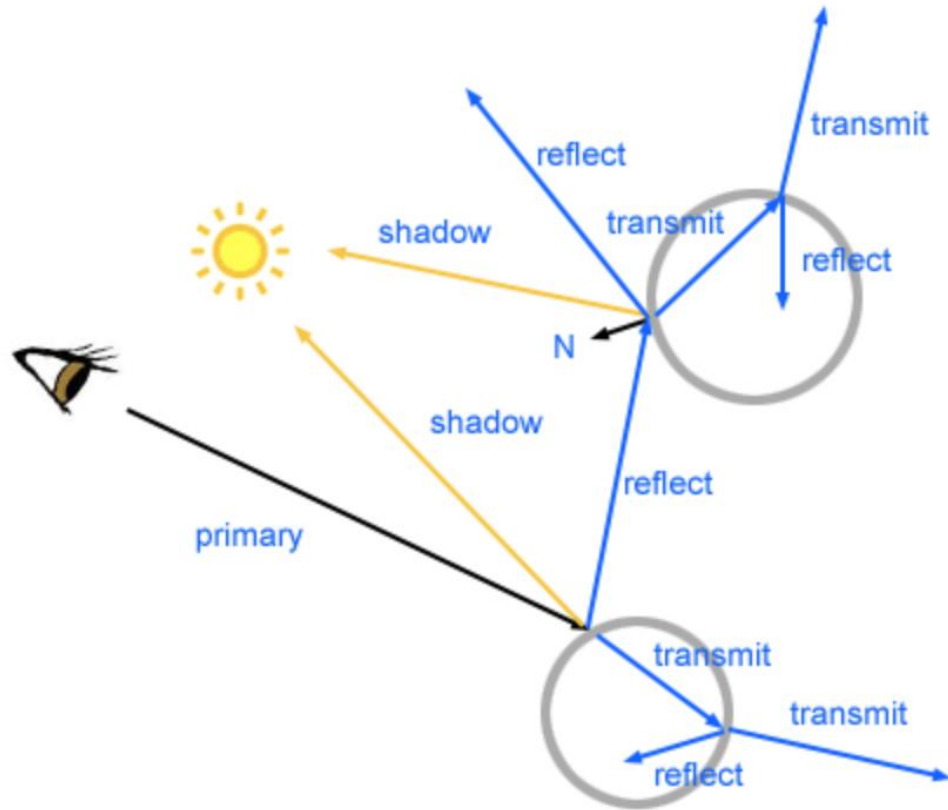
Tree ADT for photo-realistic rendering using Ray-tracing in CG



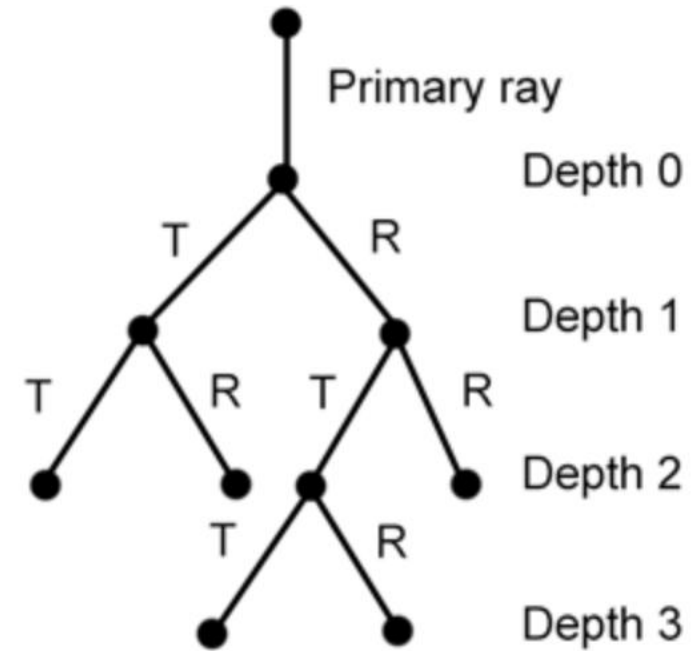
Ray_tracing tree



Ray_tracing tree



© www.scratchapixel.com



© www.scratchapixel.com

Next: Week-11(B)

- Graph ADT, and Graph Algorithms.