COSE436

Lecture 18: Hierarchical Models

Nov 14, 2024

Won-Ki Jeong
(wkjeong@korea.ac.kr)

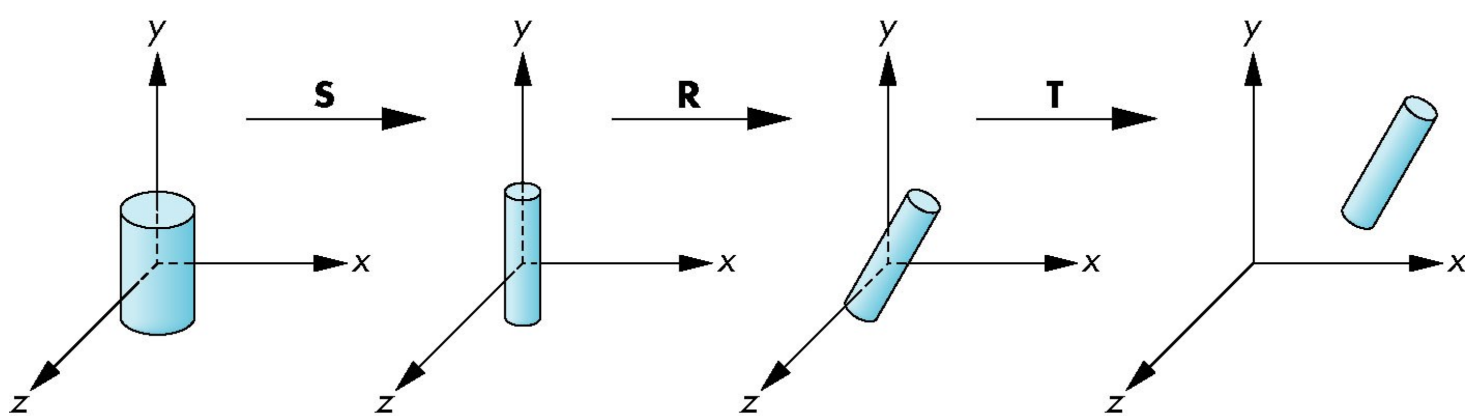
KOREA UNIVERSITY

Outline

- Limitations of linear modeling
- Hierarchical modeling
- Tree-structured modeling and traversal

Instance Transformation

- Start with a prototype object (a *symbol*)
- Each appearance of the object in the model is an *instance*
  - Can have different scale, orient, position
  - Defines instance transformation



Symbol-Instance Table

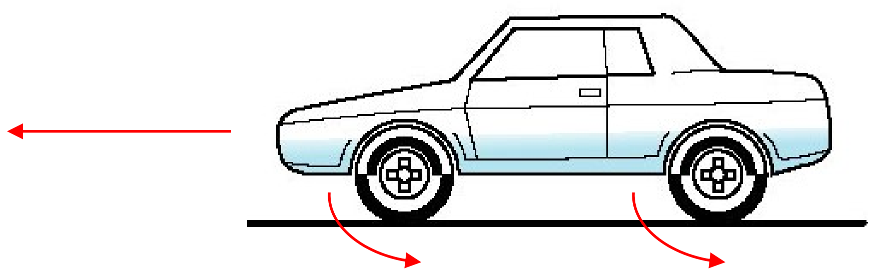
- Can store a model by assigning a number to each symbol and storing the parameters for the instance transformation

Symbol	Scale	Rotate	Translate
1	s_x, s_y, s_z	$\theta_x, \theta_y, \theta_z$	d_x, d_y, d_z
2			
3			
1			
1			
.			
.			

Same symbol
Different instance

Relationships in Car Model

- Symbol-instance table does not show relationships between parts of model
- Consider model of car
  - Chassis + 4 identical wheels
  - Two symbols



- Rate of forward motion determined by rotational speed of wheels

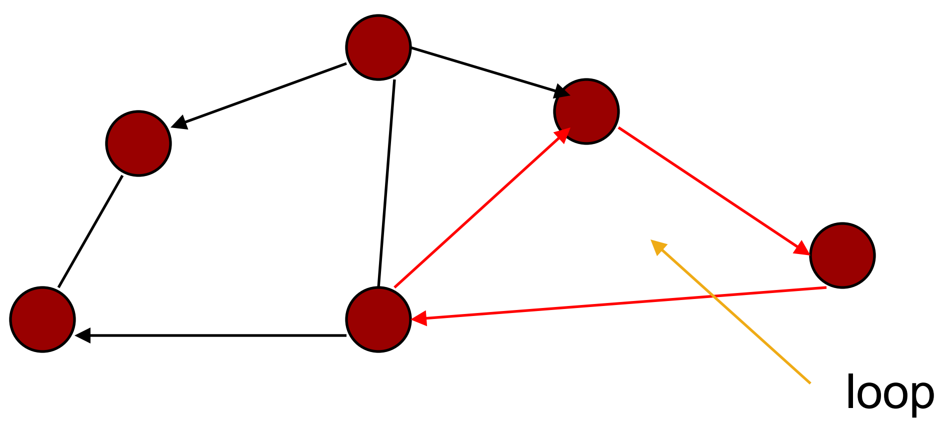
Structure Through Function Calls

```
car(speed)
{
    chassis()
    wheel(right_front);
    wheel(left_front);
    wheel(right_rear);
    wheel(left_rear);
}
```

- Fails to show relationships well
- Look at problem using a graph

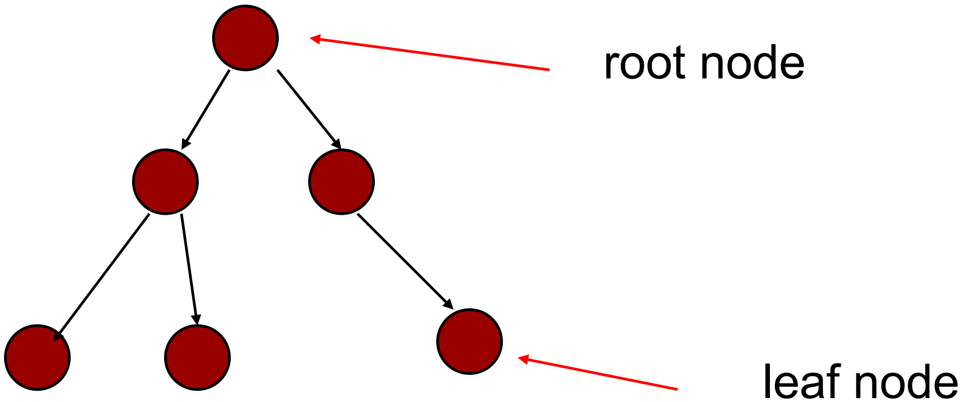
Graphs

- Set of *nodes* and *edges (links)*
- Edge connects a pair of nodes
  - Directed or undirected
- *Cycle*: directed path that is a loop

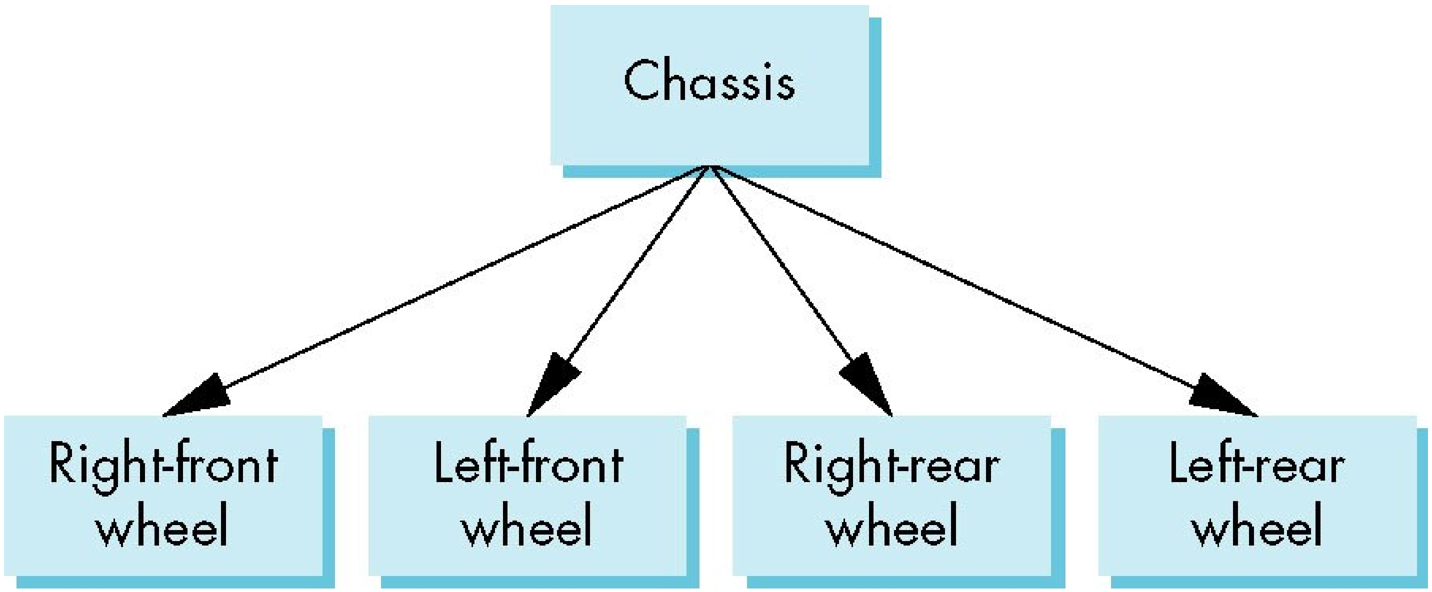


Tree

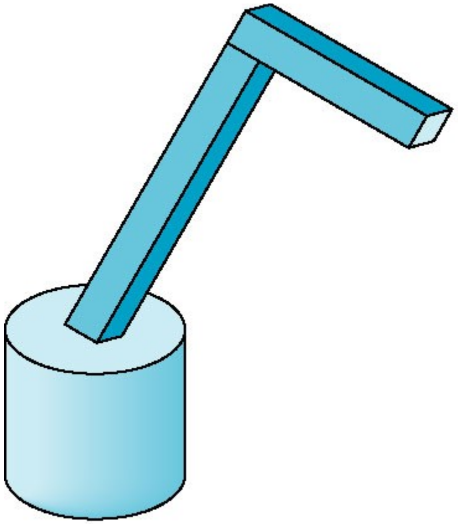
- Graph in which each node (except the root) has exactly one parent node
  - May have multiple children
  - Leaf or terminal node: no children



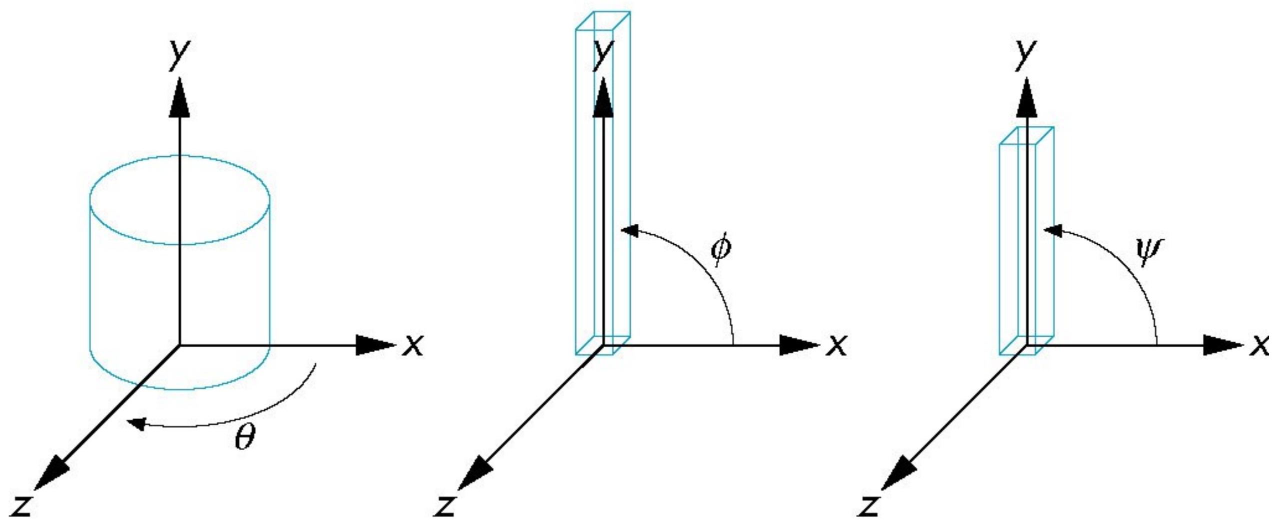
Tree Model of Car



Robot Arm



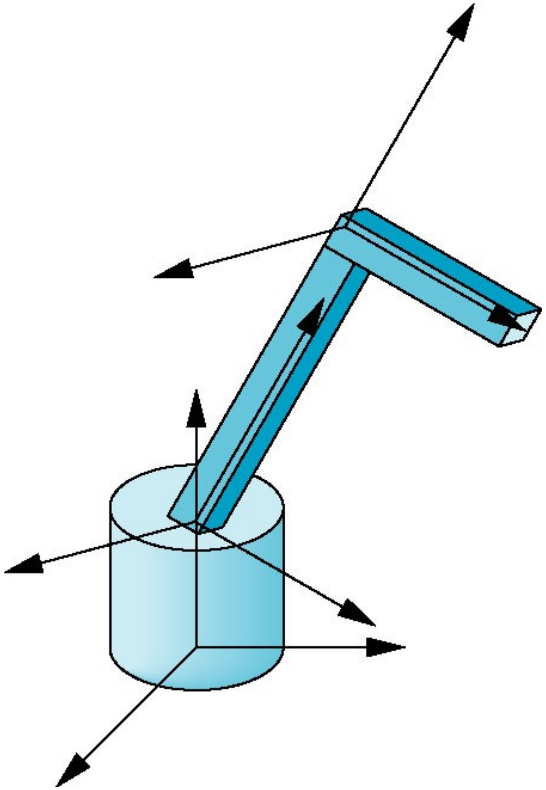
robot arm



parts in their own
coodinate systems

Articulated Models

- Robot arm is an example of an *articulated model*
  - Parts connected at joints
  - Can specify state of model by giving all joint angles



Relationships in Robot Arm

- Base rotates independently
  - Single angle determines position
- Lower arm attached to base
  - Its position depends on rotation of base
  - Must also translate relative to base and rotate about connecting joint
- Upper arm attached to lower arm
  - Its position depends on both base and lower arm
  - Must translate relative to lower arm and rotate about joint connecting to lower arm

Required Matrices

- Rotation of base: $\mathbf{R}_b$
  - Apply $\mathbf{M} = \mathbf{R}_b$ to base
- Translate lower arm relative to base: $\mathbf{T}_{lu}$
- Rotate lower arm around joint: $\mathbf{R}_{lu}$
  - Apply $\mathbf{M} = \mathbf{R}_b \mathbf{T}_{lu} \mathbf{R}_{lu}$ to lower arm
- Translate upper arm relative to lower arm: $\mathbf{T}_{uu}$
- Rotate upper arm around joint: $\mathbf{R}_{uu}$
  - Apply $\mathbf{M} = \mathbf{R}_b \mathbf{T}_{lu} \mathbf{R}_{lu} \mathbf{T}_{uu} \mathbf{R}_{uu}$ to upper arm

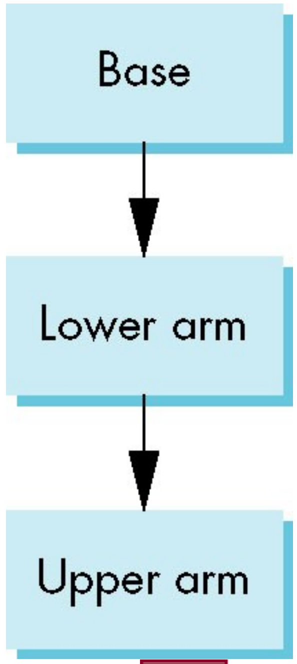
OpenGL Code for Robot Arm

```
mat4 ctm;
robot_arm()
{
    ctm = RotateY(theta);
    base();
    ctm *= Translate(0.0, h1, 0.0);
    ctm *= RotateZ(phi);
    lower_arm();
    ctm *= Translate(0.0, h2, 0.0);
    ctm *= RotateZ(psi);
    upper_arm();
}
```

Tree Model of Robot Arm

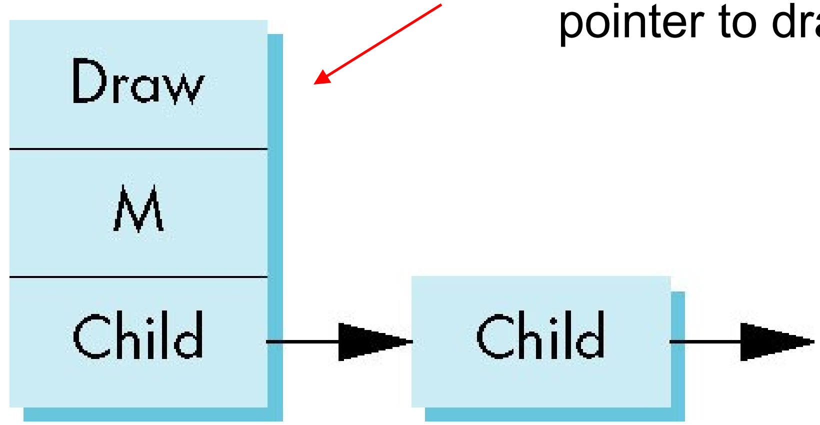
- Note code shows relationships between parts of model
  - Can change "look" of parts easily without altering relationships
- Simple example of tree model
- Want a general node structure

for nodes



Possible Node Structure

Code for drawing part or
pointer to drawing function



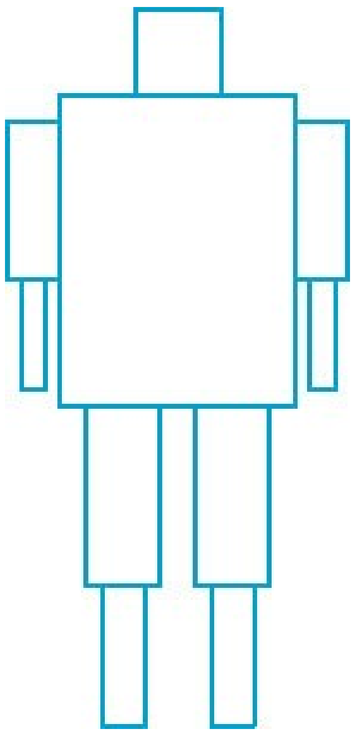
linked list of pointers to children

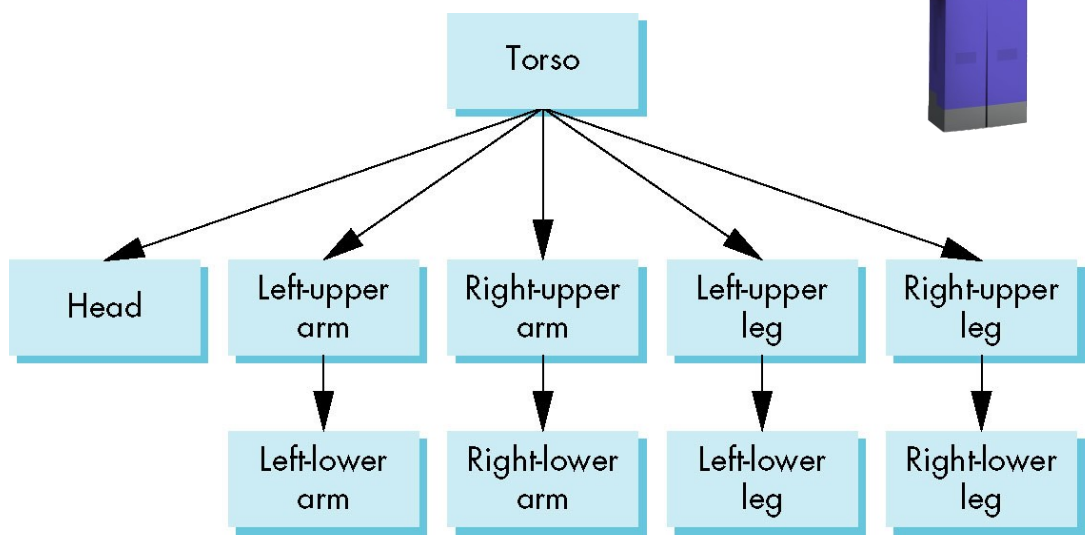
matrix relating node to parent

Generalizations

- Need to deal with multiple children
  - How do we represent a more general tree?
  - How do we traverse such a data structure?
- Animation
  - How to use dynamically?
  - Can we create and delete nodes during execution?

Humanoid Figure

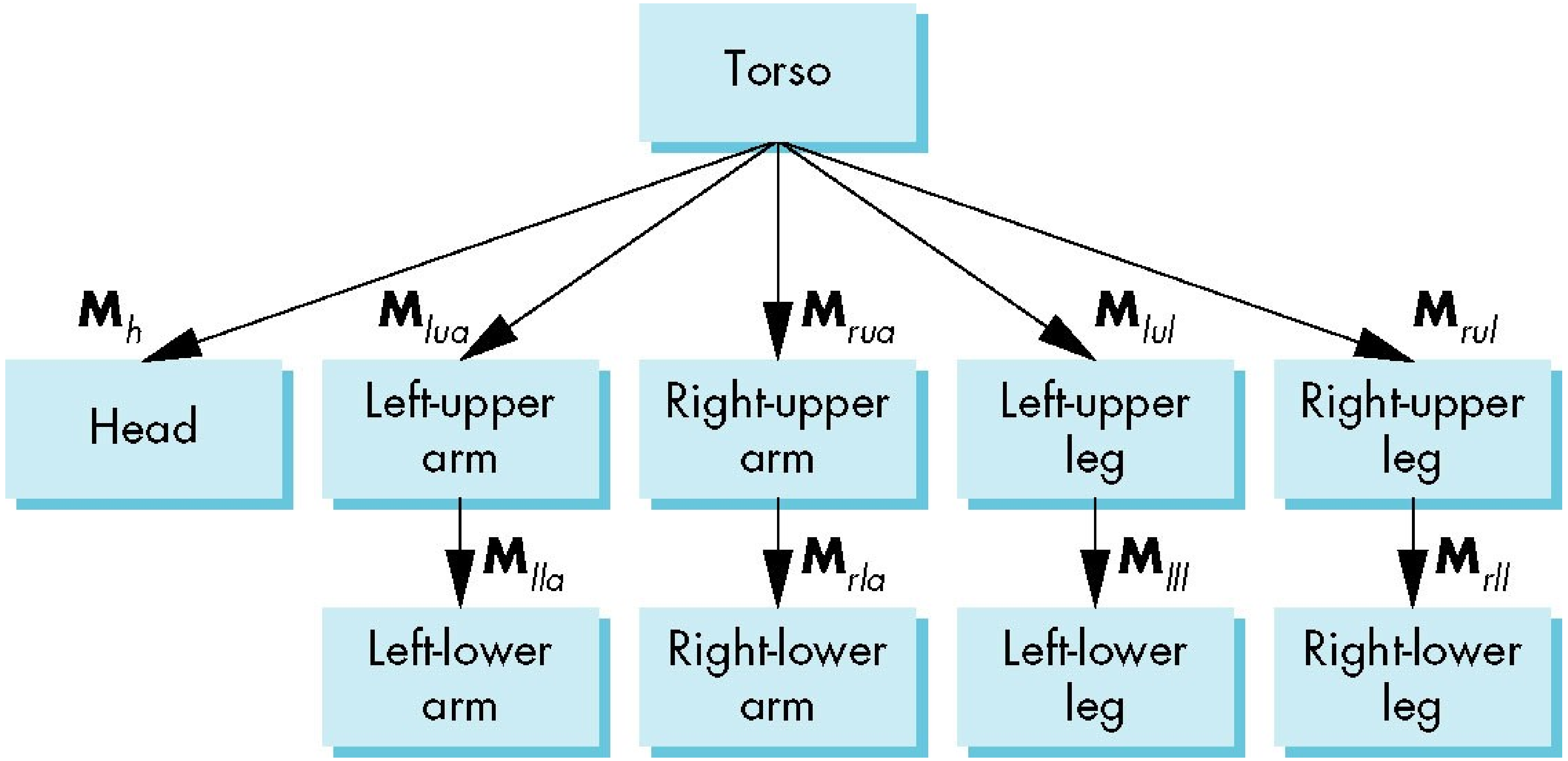




Building the Model

- Can build a simple implementation using quadrics: ellipsoids and cylinders
- Access parts through functions
  - `torso()`
  - `left_upper_arm()`
- Matrices describe position of node *with respect to* its parent
  - $\mathbf{M}_{\mathrm{lla}}$ positions left lower arm with respect to left upper arm

Tree with Matrices



Transformation Matrices

- There are 10 relevant matrices
  - $\mathbf{M}$ positions and orients entire figure through the torso which is the root node
  - $\mathbf{M}_{\mathrm{h}}$ positions head with respect to torso
  - $\mathbf{M}_{\mathrm{lua}}$, $\mathbf{M}_{\mathrm{rua}}$, $\mathbf{M}_{\mathrm{lul}}$, $\mathbf{M}_{\mathrm{rul}}$ position arms and legs with respect to torso
  - $\mathbf{M}_{\mathrm{lla}}$, $\mathbf{M}_{\mathrm{rla}}$, $\mathbf{M}_{\mathrm{lll}}$, $\mathbf{M}_{\mathrm{rll}}$ position lower parts of limbs with respect to corresponding upper limbs

Display and Traversal

- Display of the tree requires a *graph traversal*
  - Visit each node once
  - Display function at each node that describes the part associated with the node, applying the correct transformation matrix for position and orientation
- Traversal
  - Explicit traversal
  - Recursive tree traversal

Explicit Traversal

- Set model-view matrix to $\mathbf{M}$ and draw torso
- Set model-view matrix to $\mathbf{M}\mathbf{M}_{\mathrm{lua}}$ and draw left-upper arm
- For left-lower arm need $\mathbf{M}\mathbf{M}_{\mathrm{lua}}\mathbf{M}_{\mathrm{lla}}$ and so on
- We can use the matrix stack to store previous matrices as we traverse the tree

Traversal Code

```
figure() {
    PushMatrix()
    torso();
    Rotate (…);
    head();
    PopMatrix();
    PushMatrix();
    Translate(…);
    Rotate(…);
    left_upper_arm();
    PopMatrix();
    PushMatrix();
```

save present model-view matrix

update model-view matrix for head

recover original model-view matrix

save it again

update model-view matrix
for left upper arm

recover and save original
model-view matrix again

rest of code

Analysis

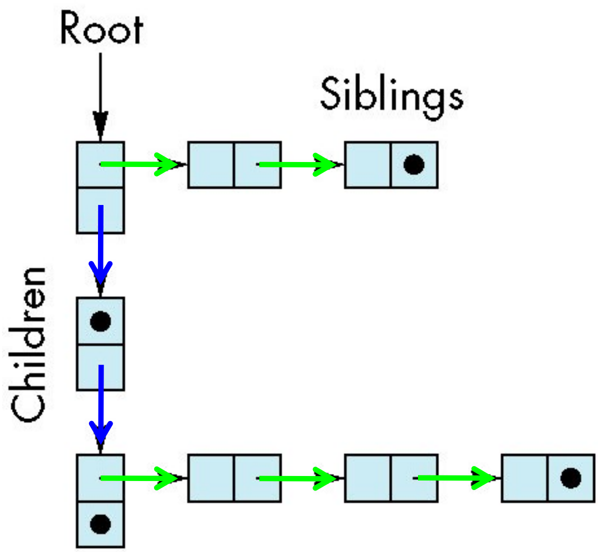
- The code describes a particular tree and a particular traversal strategy
  - Can we develop a more general approach?
- Note that the sample code does not include state changes, such as changes to colors
  - May also want to use a **`PushAttrib`** and **`PopAttrib`** to protect against unexpected state changes affecting later parts of the code

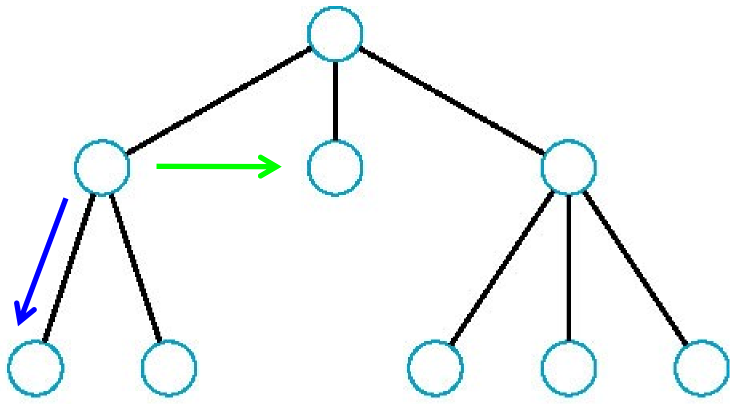
General Tree Data Structure

- Need a data structure to represent tree and an algorithm to traverse the tree
- We will use a left-child right sibling structure
  - Uses linked lists
  - Each node in data structure has two pointers
    - Left: next child node
    - Right: linked list of sibling children

Left-Child Right-Sibling Tree

- Blue: child, Green: sibling





Tree Node Structure

- At each node we need to store
  - Pointer to sibling
  - Pointer to child
  - Pointer to a function that draws the object represented by the node
  - Homogeneous coordinate matrix to multiply on the right of the current model-view matrix

Definition of treenode

```
typedef struct treenode
{
    mat4 m;
    void (*f)();
    struct treenode *sibling;
    struct treenode *child;
} treenode;
```

Example: torso and head nodes

```
treenode torso_node, head_node, lua_node, … ;

torso_node.m = RotateY(theta[0]);
torso_node.f = torso;
torso_node.sibling = NULL;
torso_node.child =  &head_node;

 head_node.m = translate(0.0,
   TORSO_HEIGHT+0.5*HEAD_HEIGHT,
   0.0)*RotateX(theta[1])*RotateY(theta[2]);
head_node.f = head;
head_node.sibling = &lua_node;
head_node.child = NULL;
```

Notes

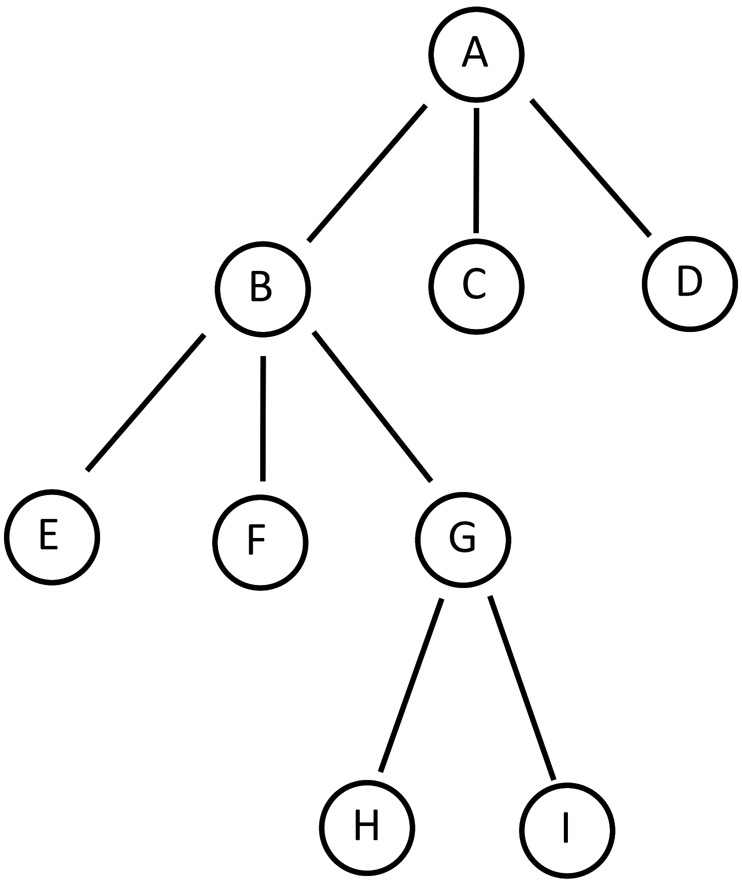
- The position of figure is determined by 11 joint angles stored in **`theta[11]`**
- Animate by changing the angles and redisplaying
- We form the required per-node matrix using **`Rotate`** and **`Translate`**
  - Because the matrix is formed using the model-view matrix, we may want to first push original model-view matrix on matrix stack

Recursive Traversal (Preorder)

```
void traverse(treenode* node)
{
    if(node==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*node->m;
    node->f();
    if(node->child!=NULL) traverse(node->child);
    model_view = mvstack.pop();
    if(node->sibling!=NULL) traverse(node->sibling);
}
```

1. Depth first traversal
2. Sibling’s transformation only depends on its parent

Example

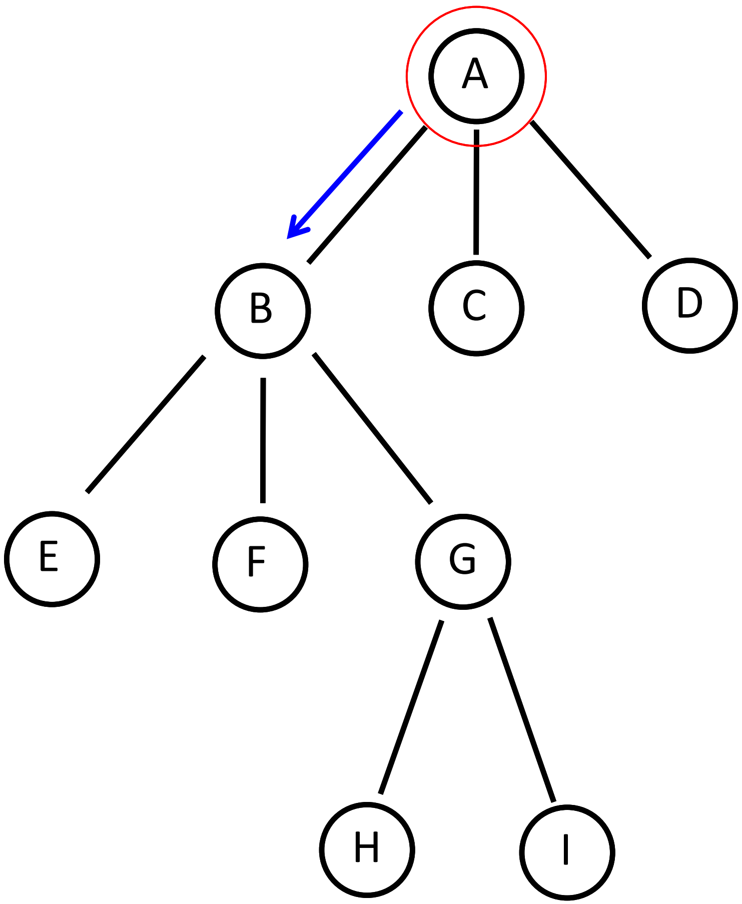



```
void traverse(treenode* node)
{
    if(node==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*node->m;
    node->f();
    if(node->child!=NULL) traverse(node->child);
    model_view = mvstack.pop();
    if(node->sibling!=NULL) traverse(node->sibling);
}
```

Current Matrix: I (Identity)

Matrix Stack:

Example

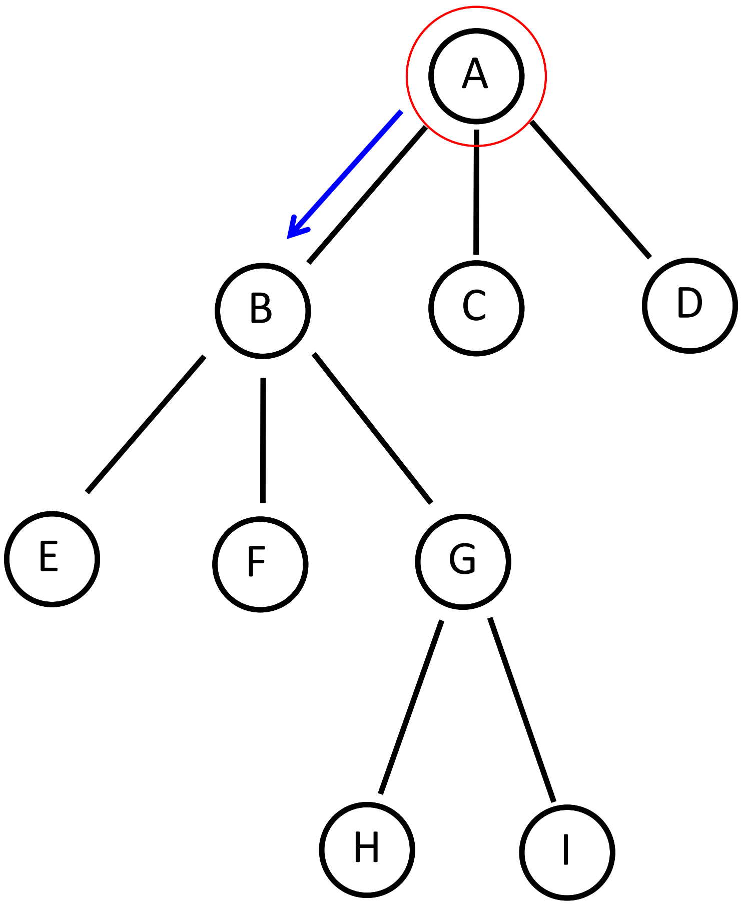



```
void traverse(treenode* node)
{
    if(node==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*node->m;
    node->f();
    if(node->child!=NULL) traverse(node->child);
    model_view = mvstack.pop();
    if(node->sibling!=NULL) traverse(node->sibling);
}
```

Current Matrix: A

Matrix Stack: I

Example

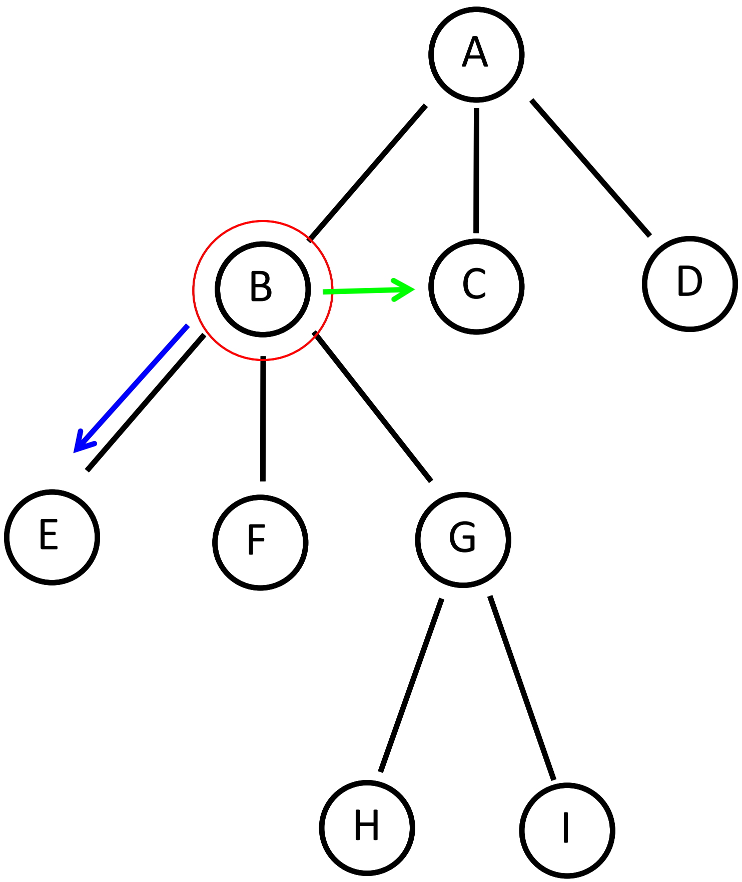



```
void traverse(treenode* node)
{
    if(node==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*node->m;
    node->f();
    if(node->child!=NULL) traverse(node->child);
    model_view = mvstack.pop();
    if(node->sibling!=NULL) traverse(node->sibling);
}
```

Current Matrix: A

Matrix Stack: I

Example

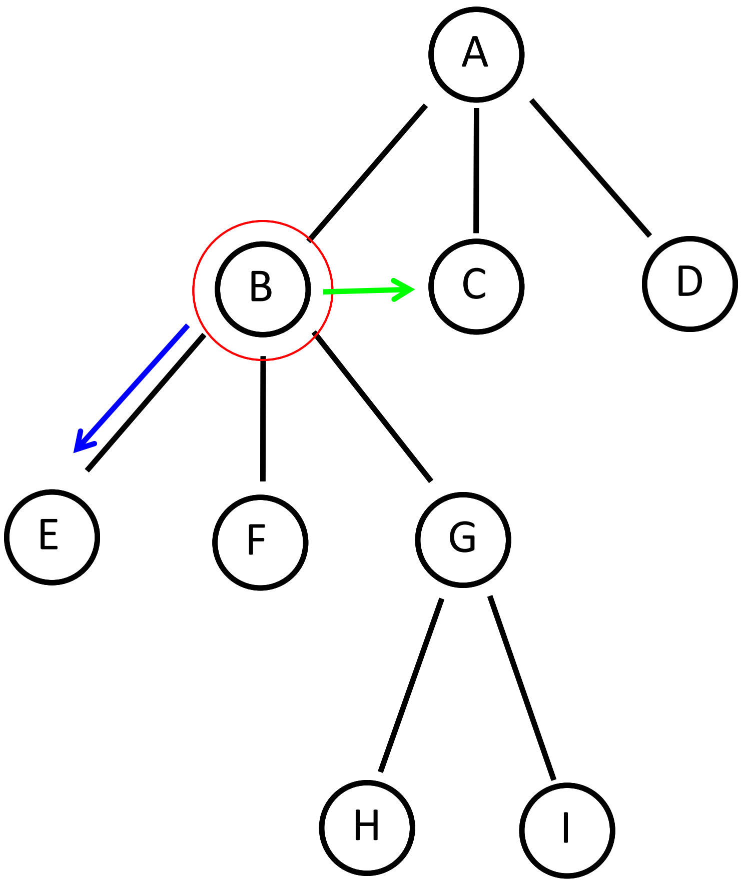



```
void traverse(treenode* node)
{
    if(node==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*node->m;
    node->f();
    if(node->child!=NULL) traverse(node->child);
    model_view = mvstack.pop();
    if(node->sibling!=NULL) traverse(node->sibling);
}
```

Current Matrix: A

Matrix Stack: I, A

Example

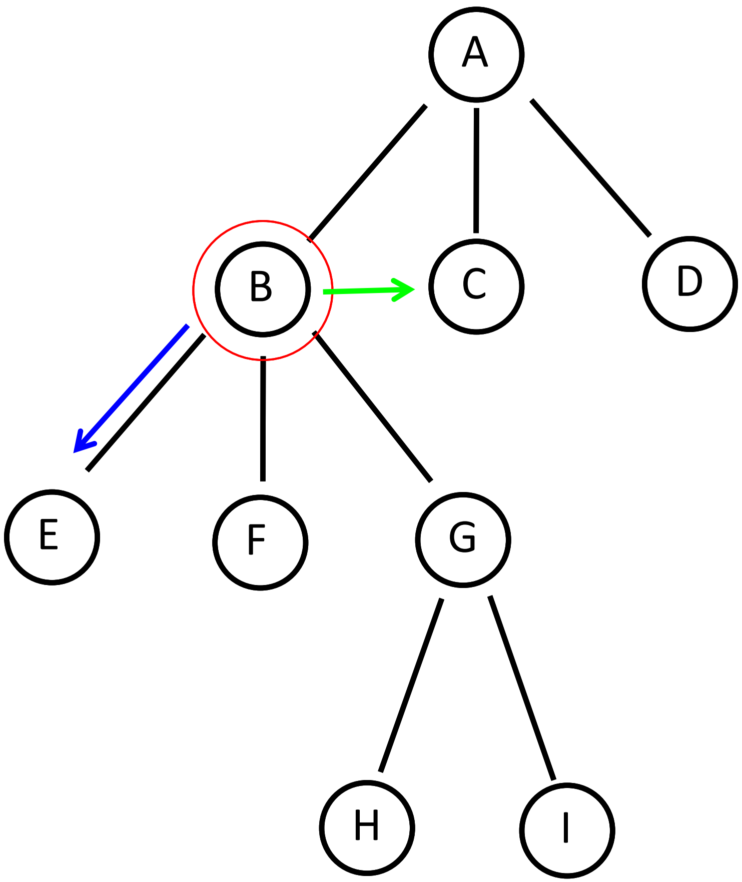



```
void traverse(treenode* node)
{
    if(node==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*node->m;
    node->f();
    if(node->child!=NULL) traverse(node->child);
    model_view = mvstack.pop();
    if(node->sibling!=NULL) traverse(node->sibling);
}
```

Current Matrix: AB

Matrix Stack: I, A

Example

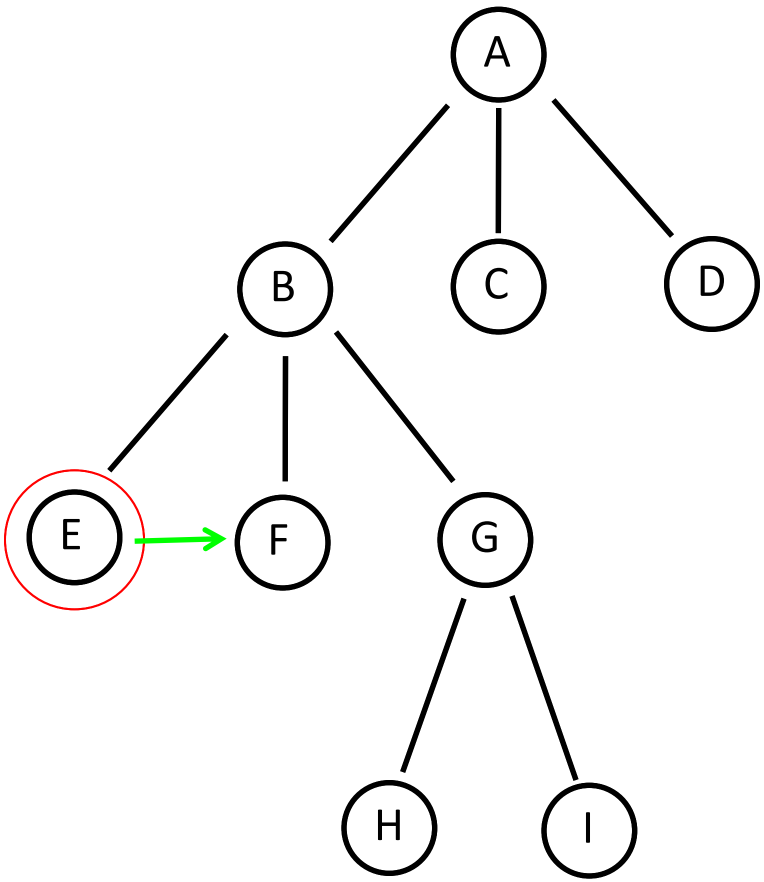



```
void traverse(treenode* node)
{
    if(node==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*node->m;
    node->f();
    if(node->child!=NULL) traverse(node->child);
    model_view = mvstack.pop();
    if(node->sibling!=NULL) traverse(node->sibling);
}
```

Current Matrix: AB

Matrix Stack: I, A

Example

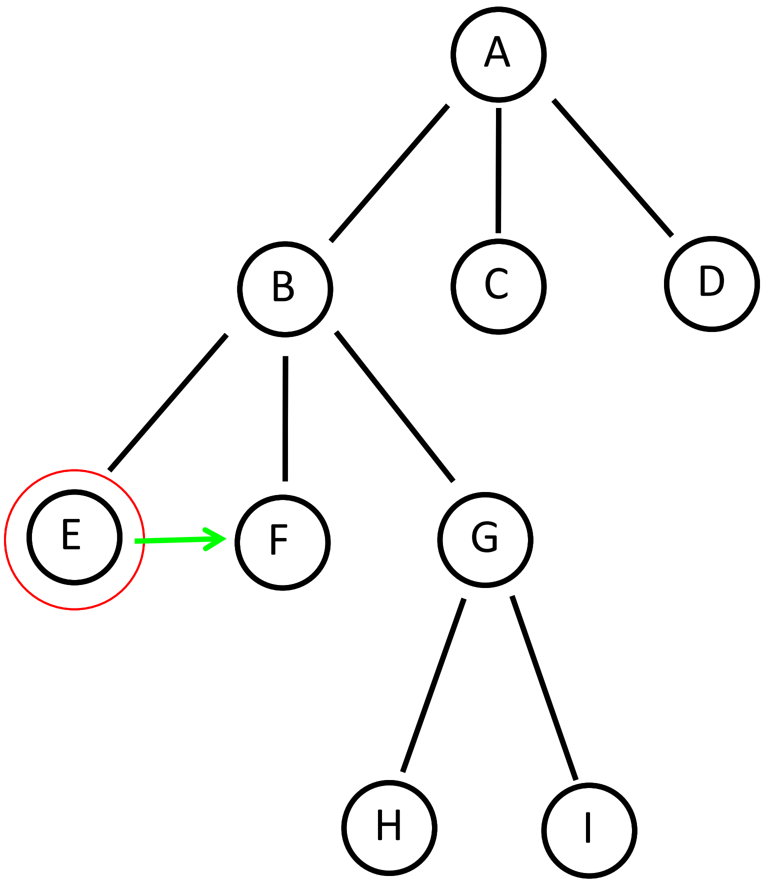



```
void traverse(treenode* node)
{
    if(node==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*node->m;
    node->f();
    if(node->child!=NULL) traverse(node->child);
    model_view = mvstack.pop();
    if(node->sibling!=NULL) traverse(node->sibling);
}
```

Current Matrix: AB

Matrix Stack: I, A, AB

Example

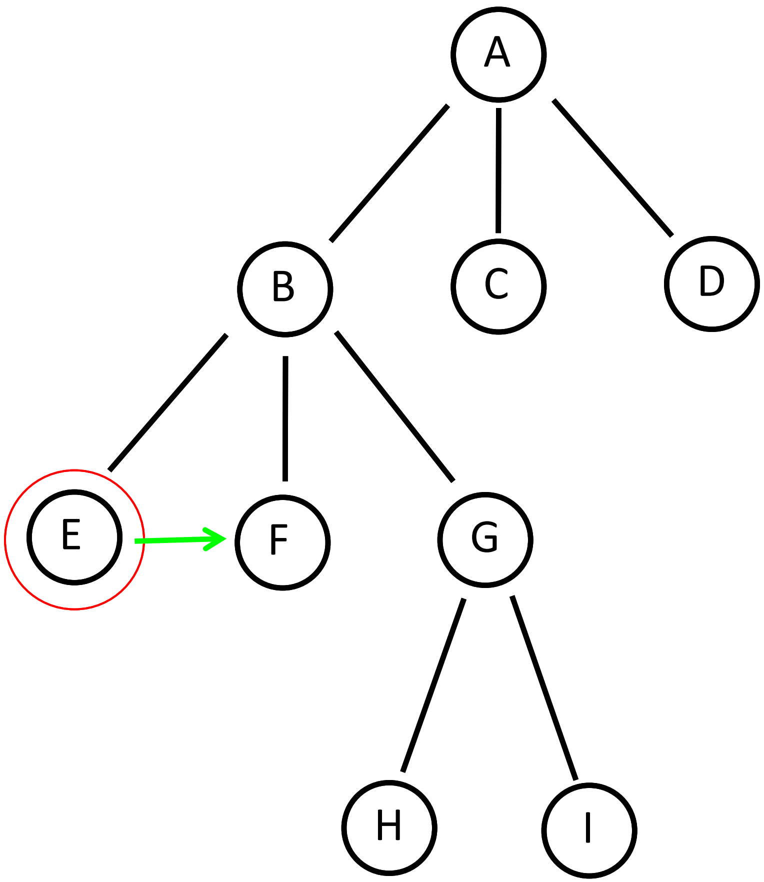



```
void traverse(treenode* node)
{
    if(node==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*node->m;
    node->f();
    if(node->child!=NULL) traverse(node->child);
    model_view = mvstack.pop();
    if(node->sibling!=NULL) traverse(node->sibling);
}
```

Current Matrix: ABE

Matrix Stack: I, A, AB

Example



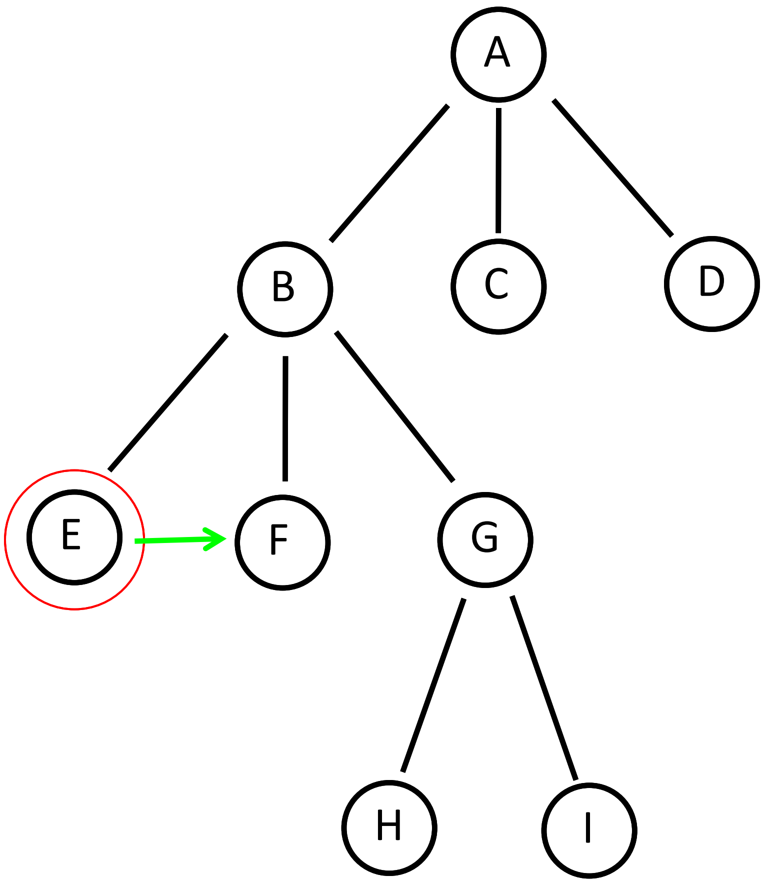

```
void traverse(treenode* node)
{
    if(node==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*node->m;
    node->f();
    if(node->child!=NULL) traverse(node->child);
    model_view = mvstack.pop();
    if(node->sibling!=NULL) traverse(node->sibling);
}
```

no child!

Current Matrix: ABE

Matrix Stack: I, A, AB

Example

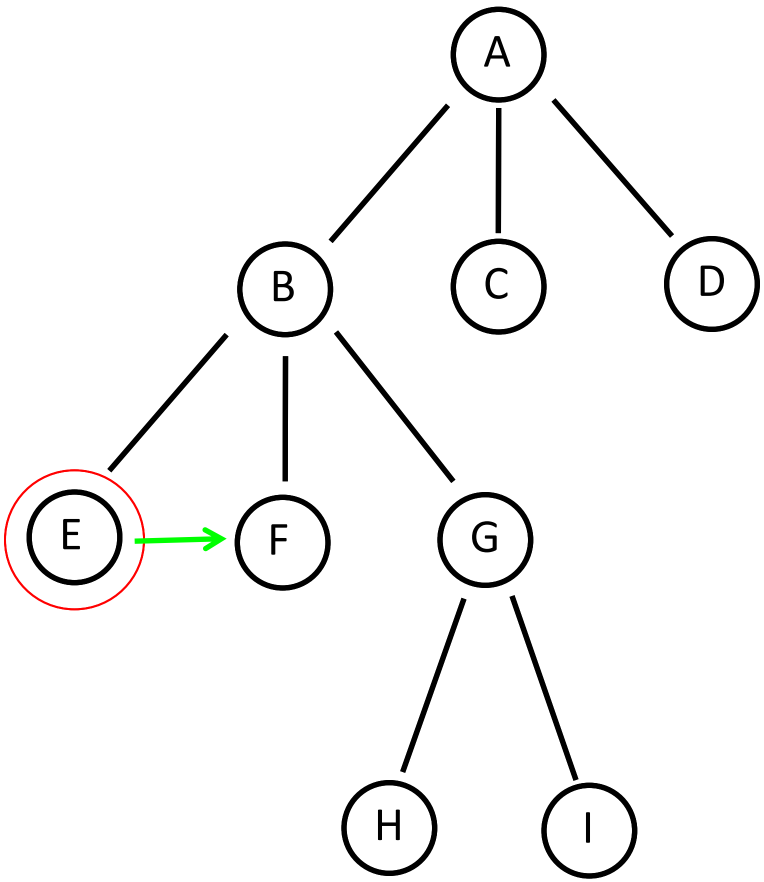



```
void traverse(treenode* node)
{
    if(node==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*node->m;
    node->f();
    if(node->child!=NULL) traverse(node->child);
    model_view = mvstack.pop();
    if(node->sibling!=NULL) traverse(node->sibling);
}
```

Current Matrix: AB

Matrix Stack: I, A

Example

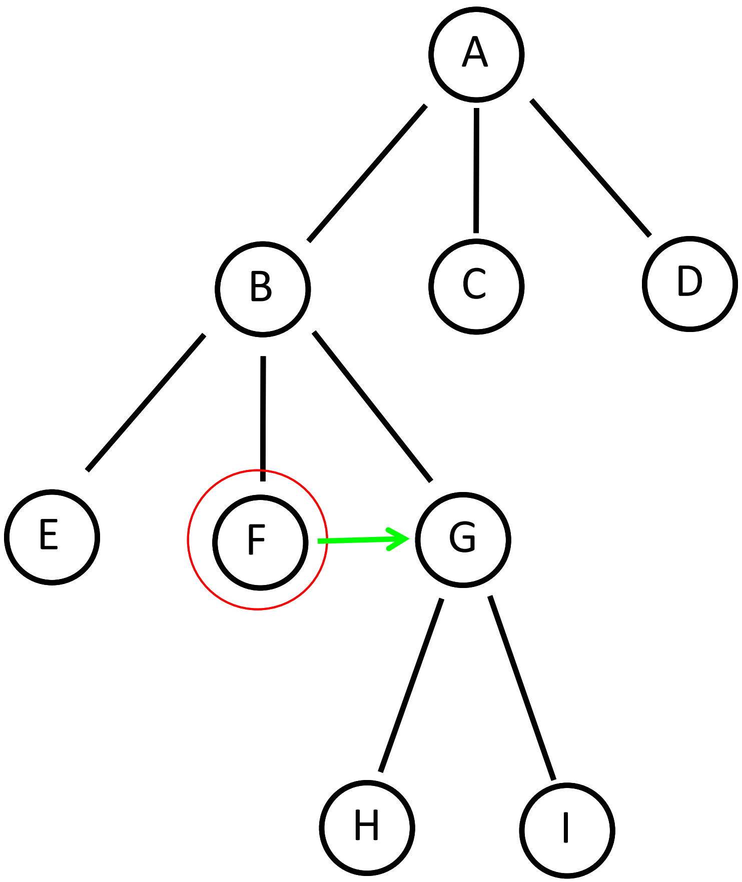



```
void traverse(treenode* node)
{
    if(node==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*node->m;
    node->f();
    if(node->child!=NULL) traverse(node->child);
    model_view = mvstack.pop();
    if(node->sibling!=NULL) traverse(node->sibling);
}
```

Current Matrix: AB

Matrix Stack: I, A

Example

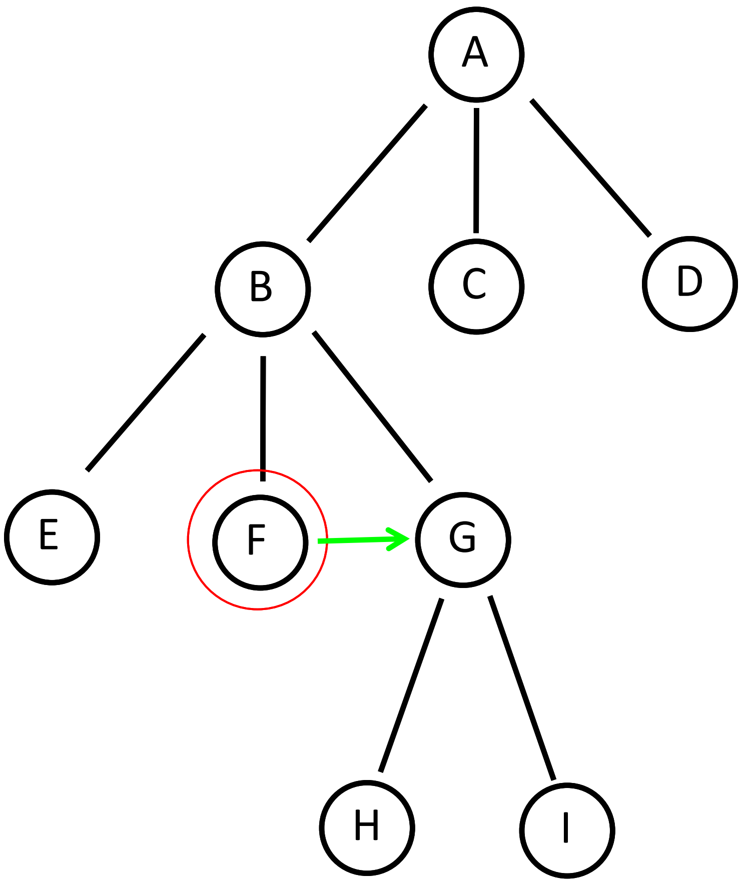



```
void traverse(treenode* node)
{
    if(node==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*node->m;
    node->f();
    if(node->child!=NULL) traverse(node->child);
    model_view = mvstack.pop();
    if(node->sibling!=NULL) traverse(node->sibling);
}
```

Current Matrix: AB

Matrix Stack: I, A, AB

Example

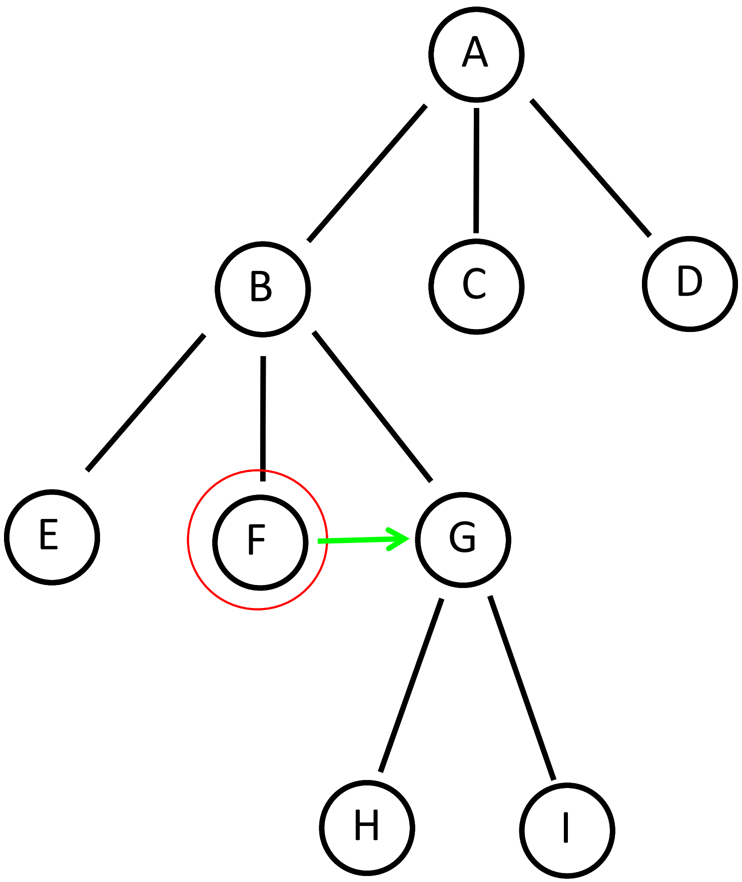



```
void traverse(treenode* node)
{
    if(node==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*node->m;
    node->f();
    if(node->child!=NULL) traverse(node->child);
    model_view = mvstack.pop();
    if(node->sibling!=NULL) traverse(node->sibling);
}
```

Current Matrix: ABF

Matrix Stack: I, A, AB

Example



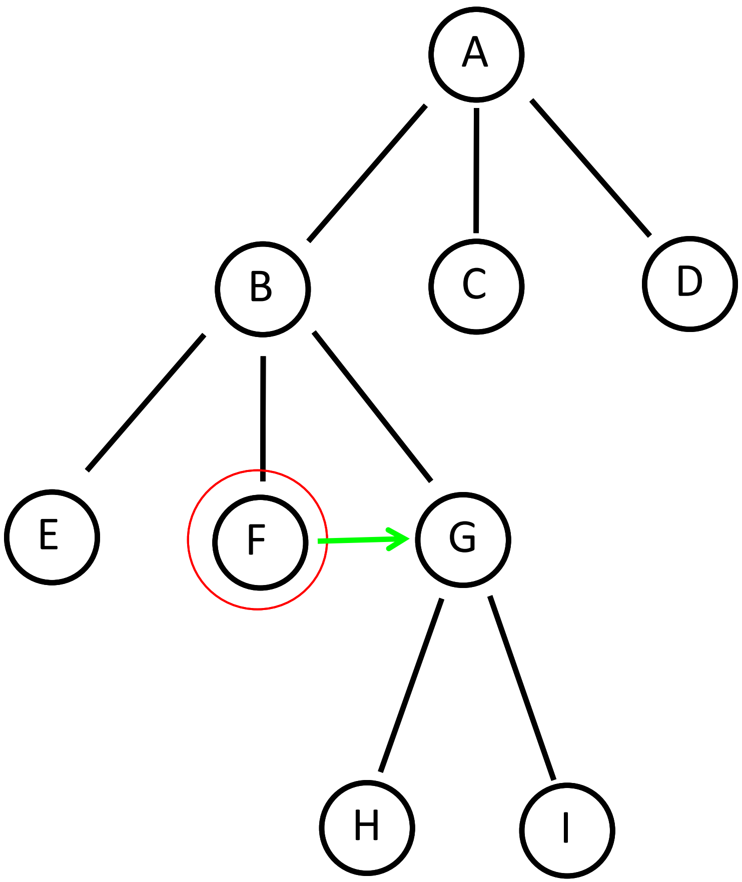

```
void traverse(treenode* node)
{
    if(node==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*node->m;
    node->f();
    if(node->child!=NULL) traverse(node->child);
    model_view = mvstack.pop();
    if(node->sibling!=NULL) traverse(node->sibling);
}
```

Current Matrix: AB

Matrix Stack: I, A

KOREA UNIVERSITY

Example

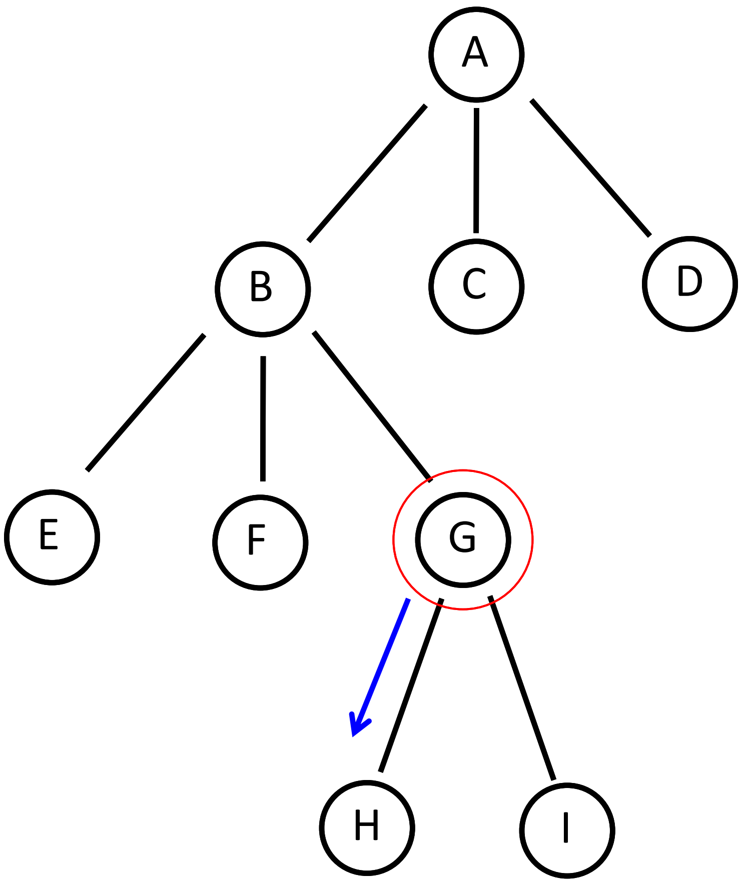



```
void traverse(treenode* node)
{
    if(node==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*node->m;
    node->f();
    if(node->child!=NULL) traverse(node->child);
    model_view = mvstack.pop();
    if(node->sibling!=NULL) traverse(node->sibling);
}
```

Current Matrix: AB

Matrix Stack: I, A

Example

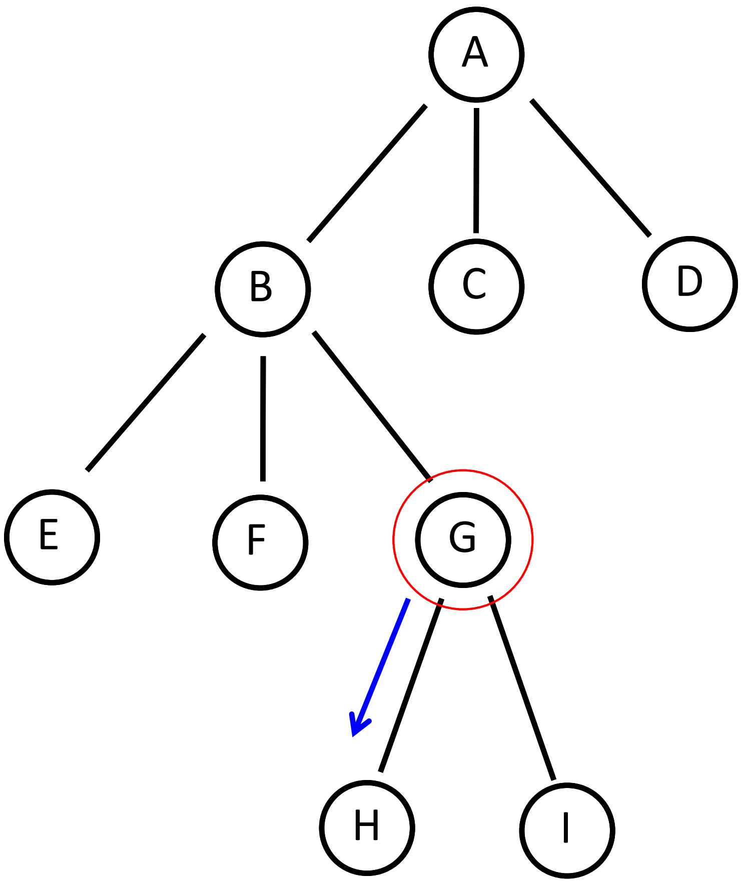



```
void traverse(treenode* node)
{
    if(node==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*node->m;
    node->f();
    if(node->child!=NULL) traverse(node->child);
    model_view = mvstack.pop();
    if(node->sibling!=NULL) traverse(node->sibling);
}
```

Current Matrix: AB

Matrix Stack: I, A, AB

Example

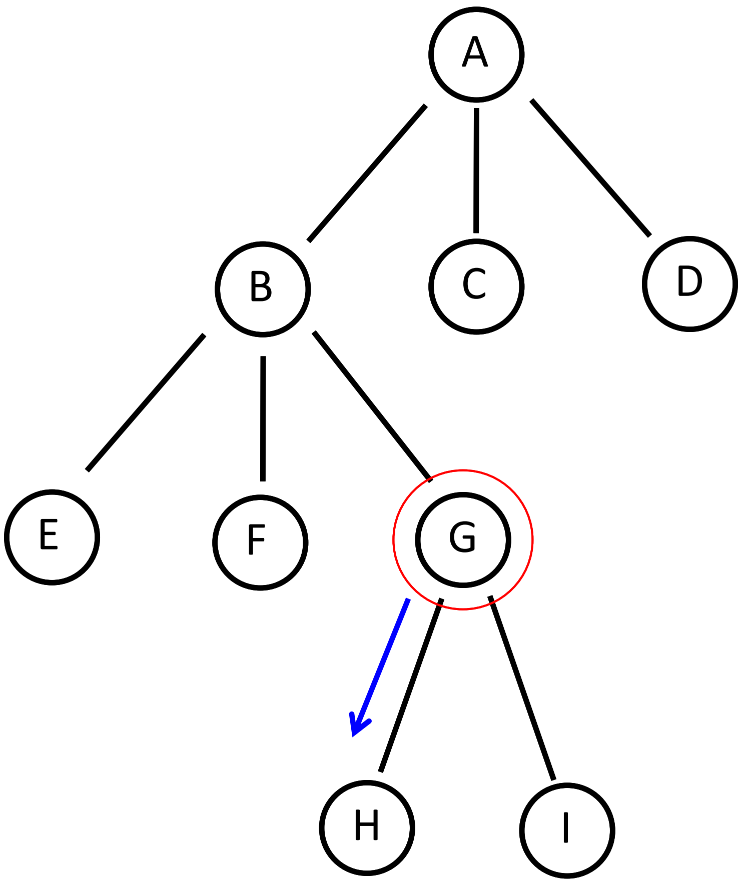



```
void traverse(treenode* node)
{
    if(node==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*node->m;
    node->f();
    if(node->child!=NULL) traverse(node->child);
    model_view = mvstack.pop();
    if(node->sibling!=NULL) traverse(node->sibling);
}
```

Current Matrix: ABG

Matrix Stack: I, A, AB

Example

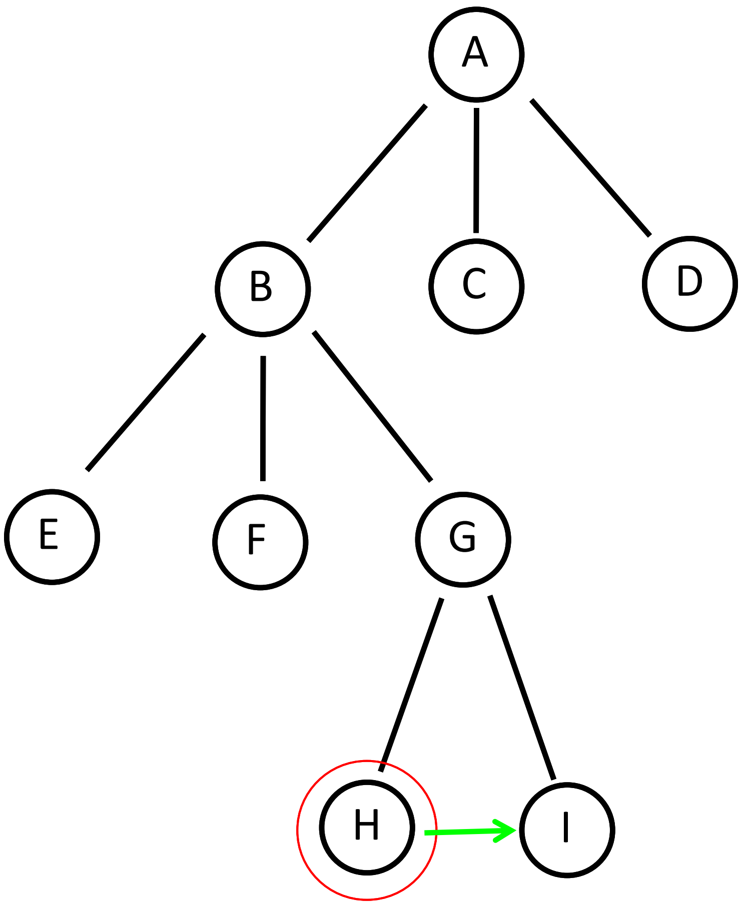



```
void traverse(treenode* node)
{
    if(node==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*node->m;
    node->f();
    if(node->child!=NULL) traverse(node->child);
    model_view = mvstack.pop();
    if(node->sibling!=NULL) traverse(node->sibling);
}
```

Current Matrix: ABG

Matrix Stack: I, A, AB

Example

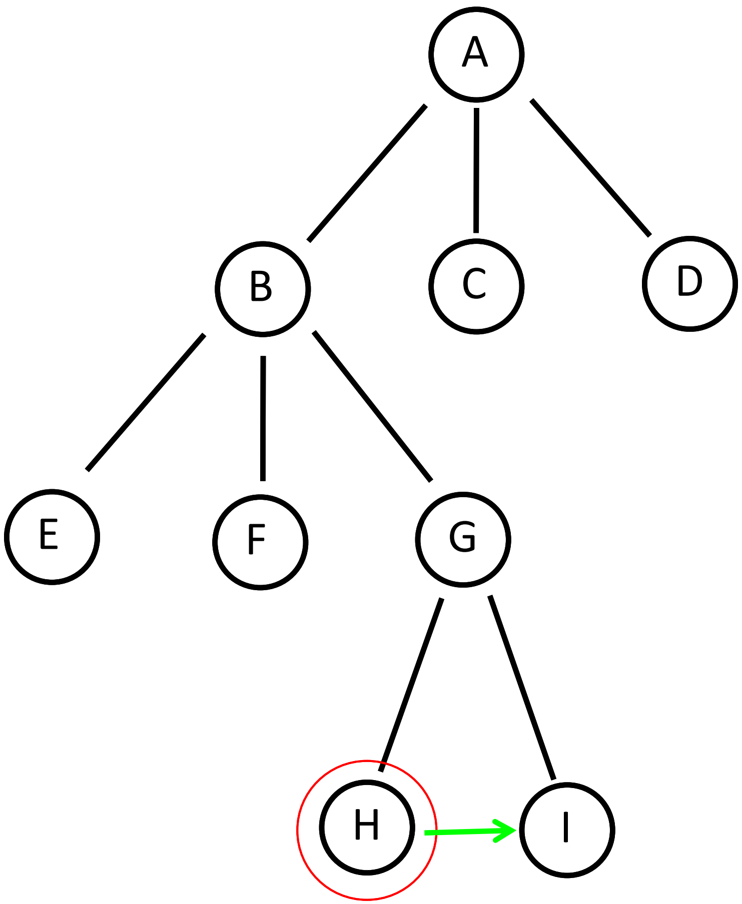



```
void traverse(treenode* node)
{
    if(node==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*node->m;
    node->f();
    if(node->child!=NULL) traverse(node->child);
    model_view = mvstack.pop();
    if(node->sibling!=NULL) traverse(node->sibling);
}
```

Current Matrix: ABG

Matrix Stack: I, A, AB, ABG

Example




```
void traverse(treenode* node)
{
    if(node==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*node->m;
    node->f();
    if(node->child!=NULL) traverse(node->child);
    model_view = mvstack.pop();
    if(node->sibling!=NULL) traverse(node->sibling);
}
```

Current Matrix: ABGH

Matrix Stack: I, A, AB, ABG

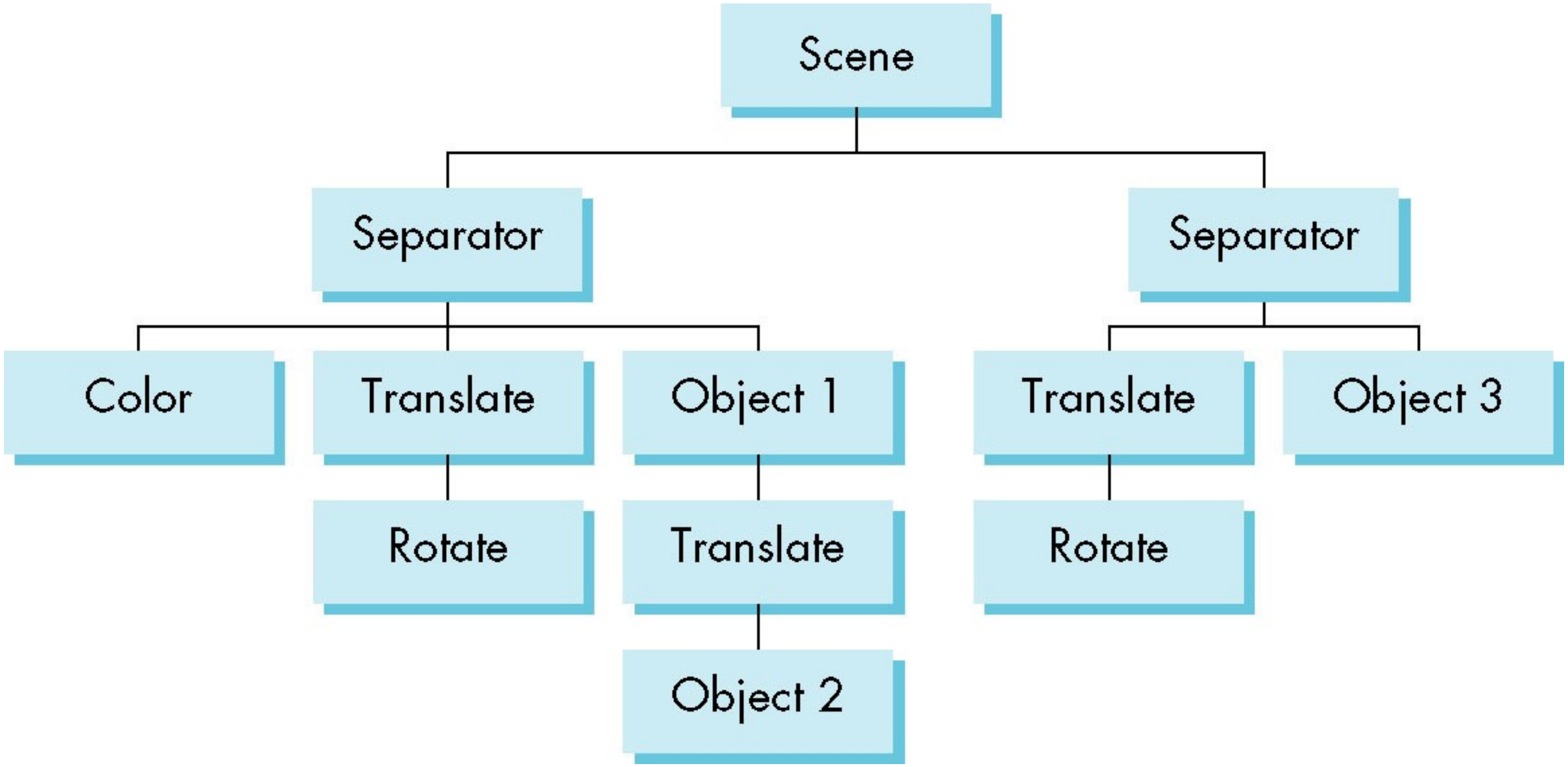
Note

- Current matrix must be stored in a stack before updated by current node's matrix
- Current matrix is concatenation of parent node's matrices with mine
- Current matrix is only applied to children, not siblings
- We can push/pop attributes as well

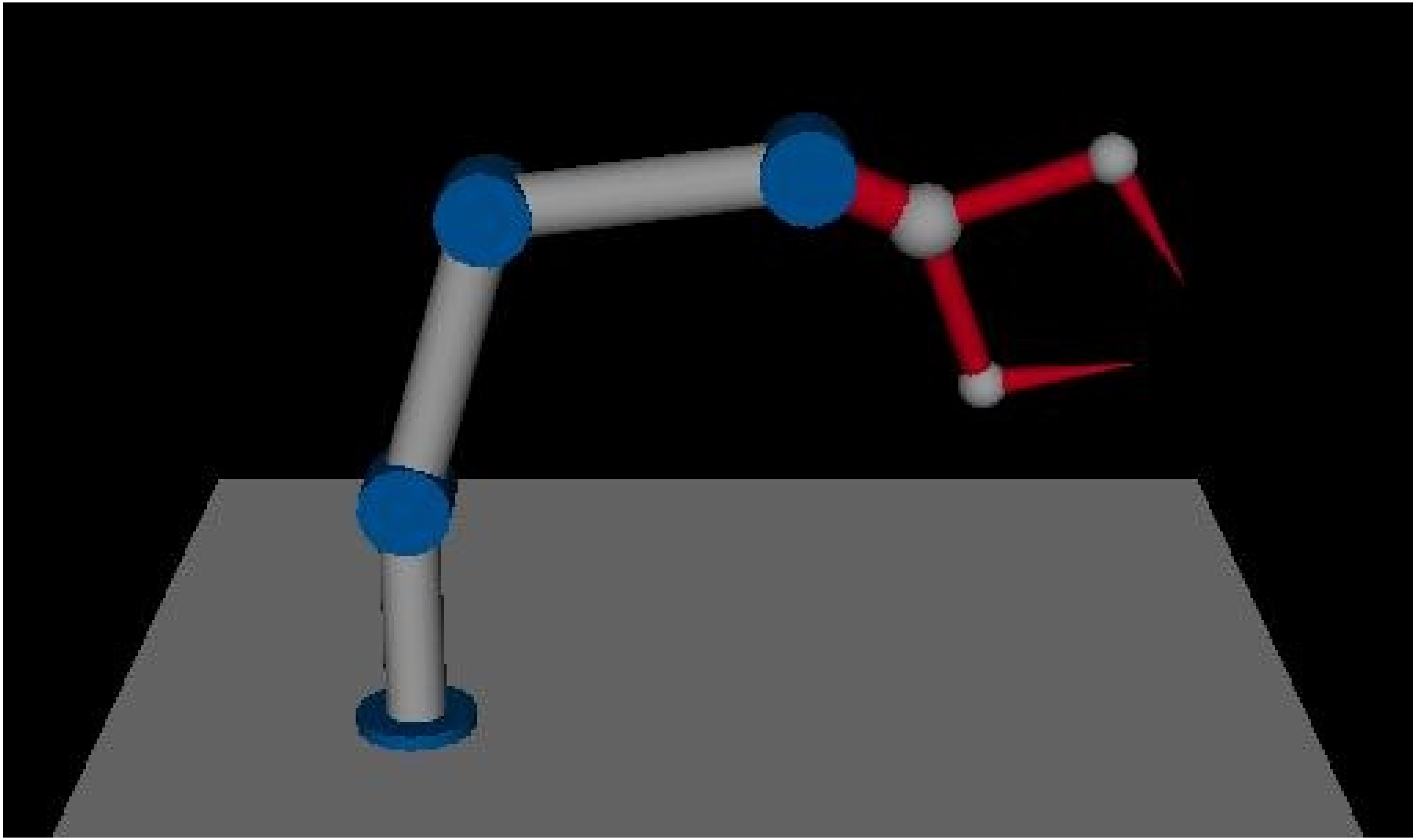
General Scene Descriptions

- In Humanoid figure model, we saw that
  - We could describe model either by tree or by equivalent code
  - We could write a generic traversal to display
- If we can represent all the elements of a scene (cameras, lights, materials, geometry) as nodes, we should be able to show them in a tree
  - Render scene by traversing this tree

Scene Graph


Scene
Separator
Separator
Color
Translate
Object 1
Translate
Object 3
Rotate
Translate
Rotate
Object 2

Questions?



Questions?