ECE 220

Lecture x0017
Trees, traversal, and BSTs intro

Recap

- Last week
  - OOP Concepts
    - Constructors, destructors, etc.
    - Inheritance, polymorphism, etc.
- Templates
  - Template functions
  - Template classes
  - Template library
    - Containers: lists vs. vectors
    - Iterators

New concept - trees

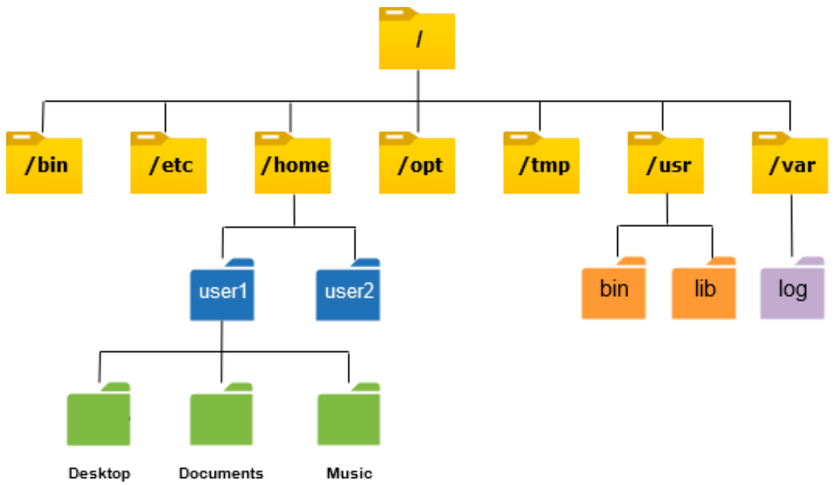
- Recall linked lists
  - Singly linked lists
  - Doubly linked lists
- Linked lists, queues, stacks: *linear* data structures

- Trees - are *nonlinear* & hierarchical
  - Think family trees or organizational charts
  - Basic unit ~ node in DLL
  - Difference - functions.

Lesson objectives

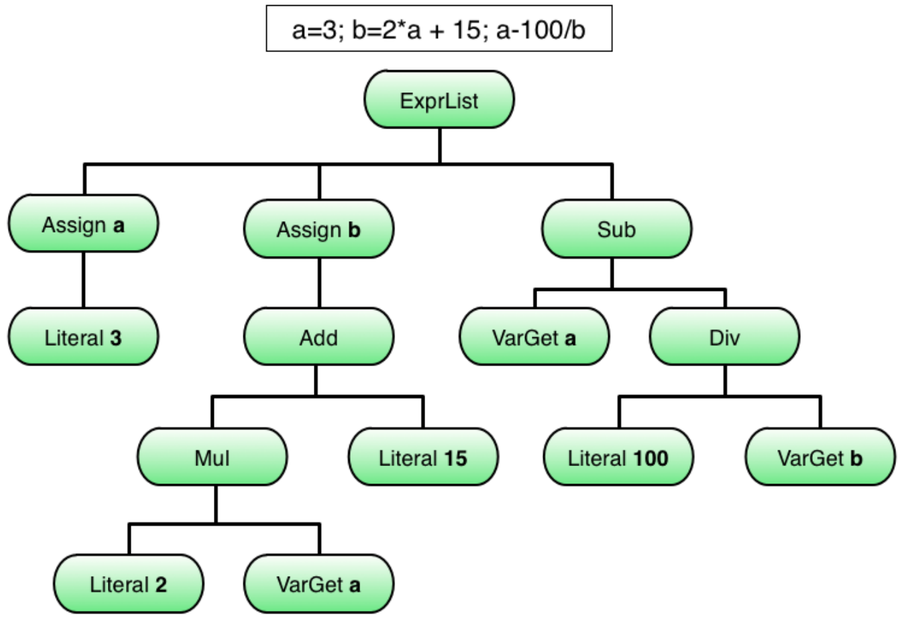
- Understand the concept of Trees as an ADT and their use in applications
- Be able to use structs in C to implement trees; specifically binary trees
- Understand and be able to implement different types of tree traversals
- Understand and be able to implement binary search trees
- Understand and be able to perform common operations on trees

Why trees?



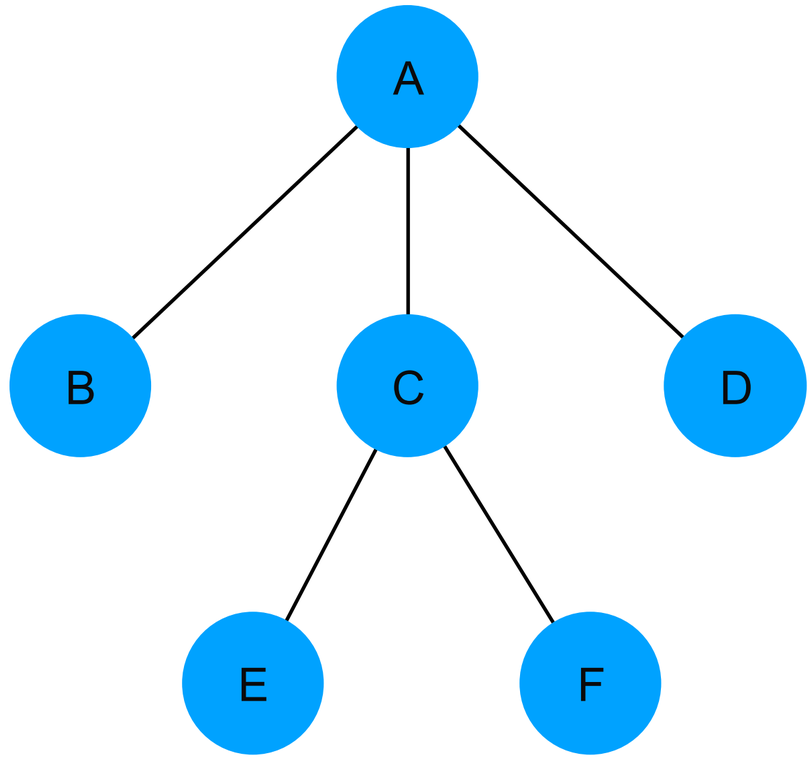
Filesystems, computer graphics, programming languages, taxonomic classifictaion, etc.

QuadTree: https://en.wikipedia.org/wiki/Quadtree



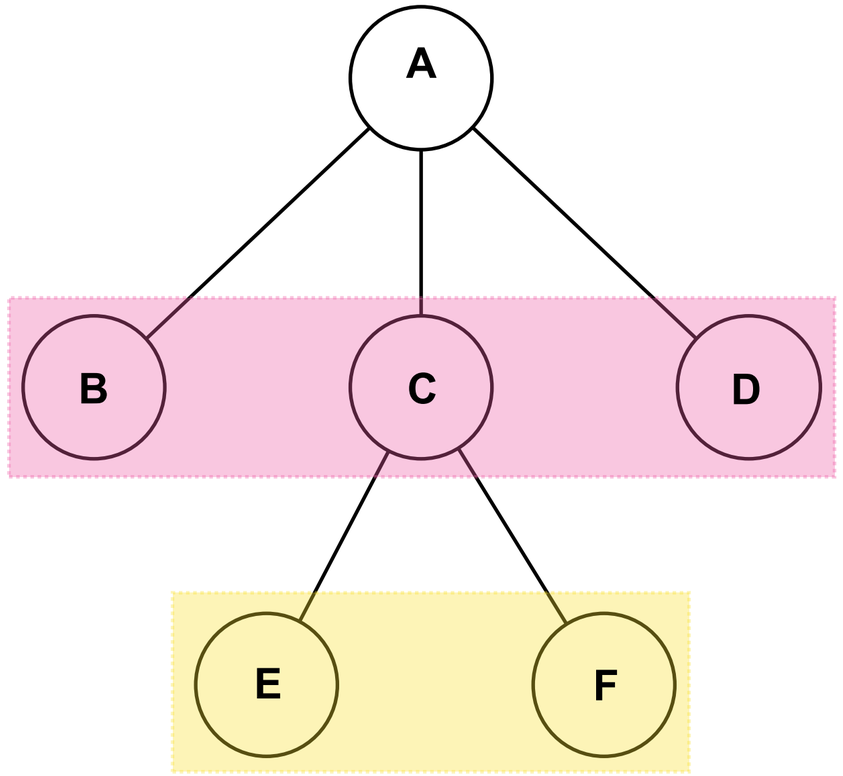
Concepts related to trees

- Root
  - Top most node, no parent.
- Leaf
  - Outermost nodes, no children
- Inner node(s)
  - Has atleast one child



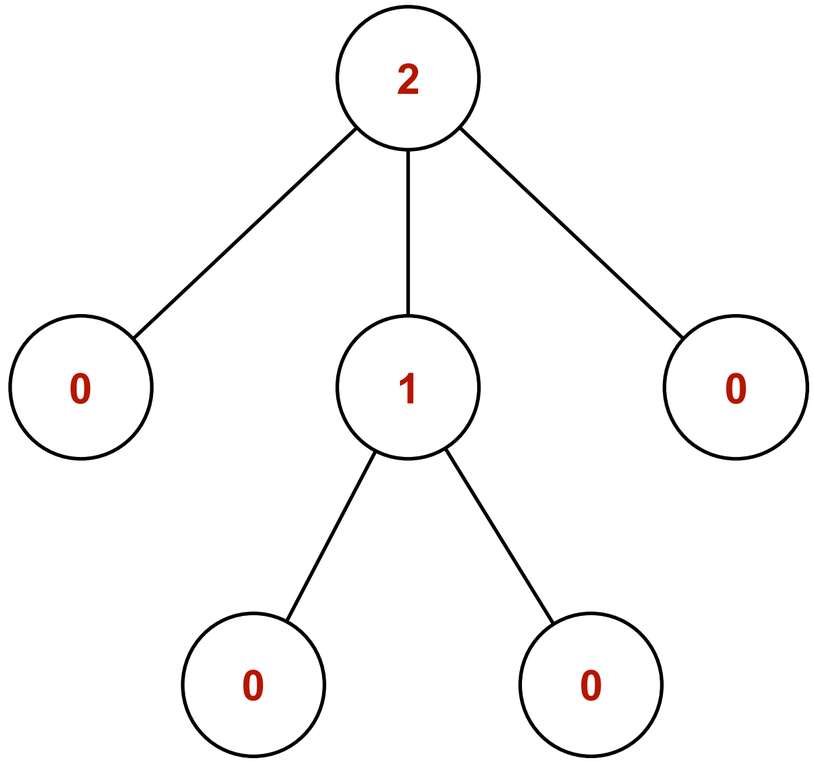
Siblings

- Siblings



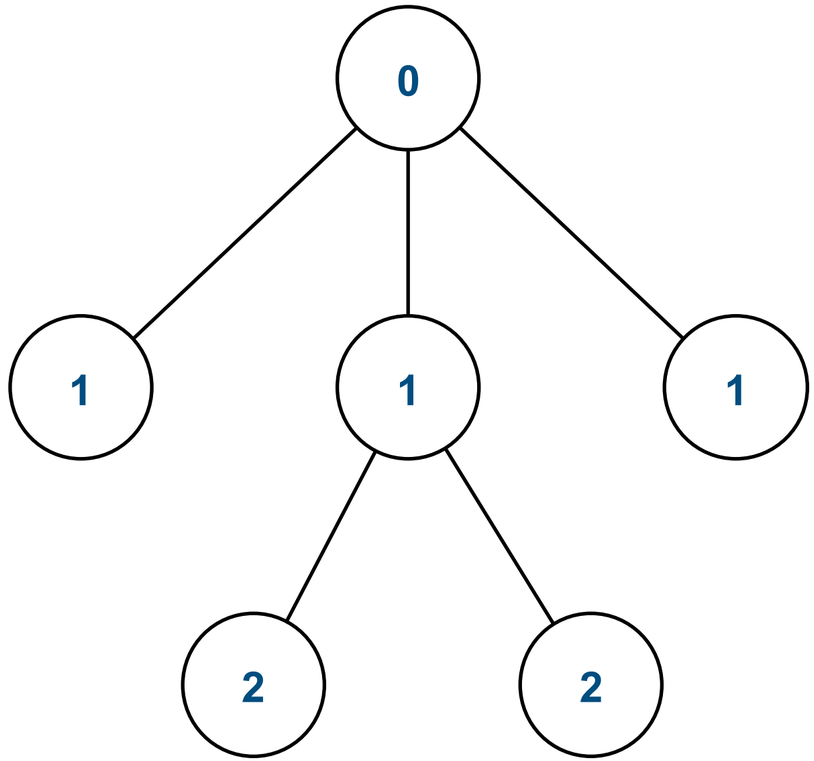
Height

- Siblings
- Height (of a node)
  - Length of *longest* path from given node to a leaf



Depth

- Siblings
- Height (of a node)
  - Length of *longest* path from given node to a leaf
- Depth (of a node)
  - Length of path from root to given node

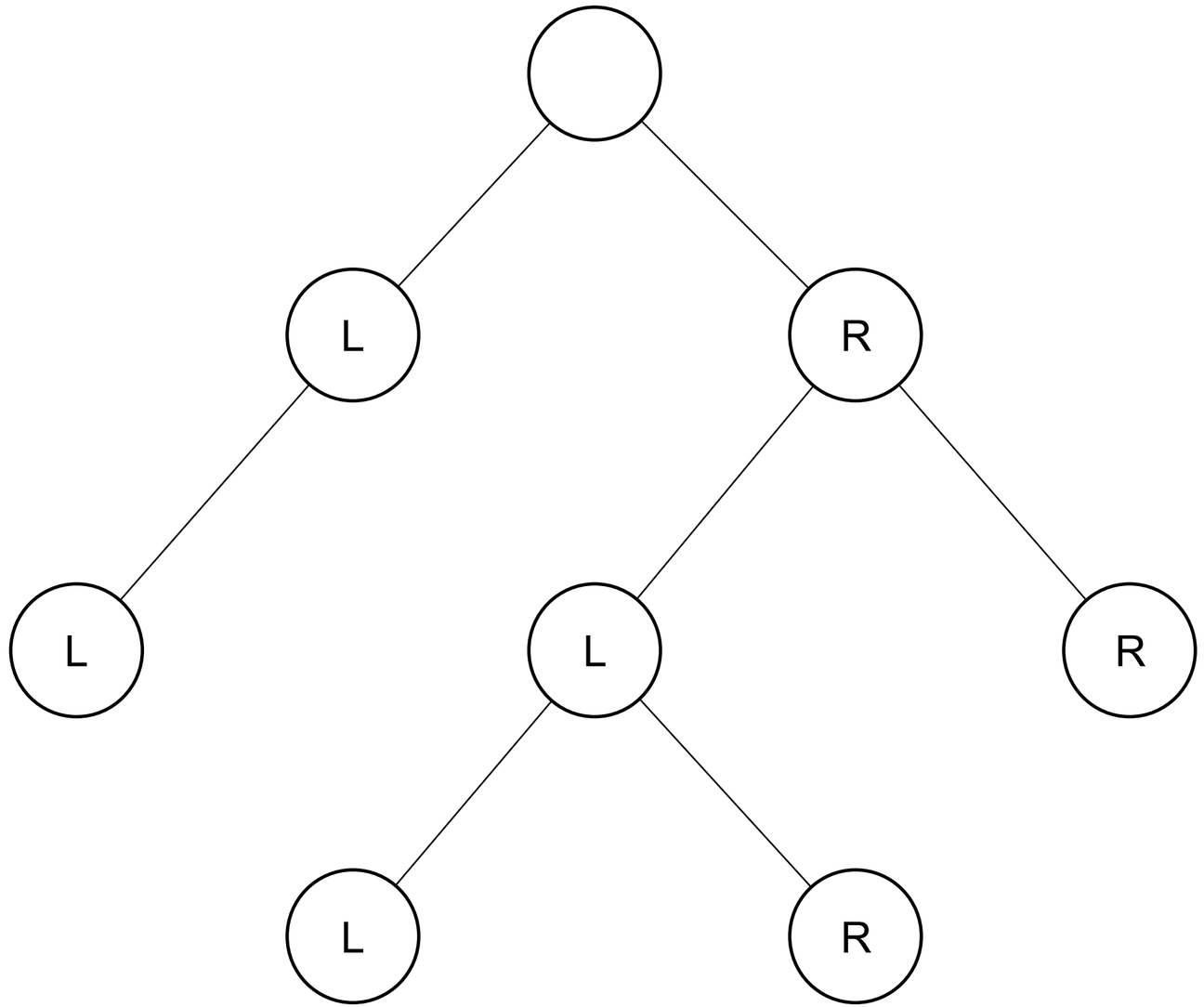


Binary trees

- Trees where every node has *at most* two children.

```
typedef struct person node;
struct person{
   char *name;
   node *next;
   node *prev;
}node;

typedef struct node treeNode;
struct node{
   int data;
   treeNode *left;
   treeNode *right;
};
```



Traversing trees

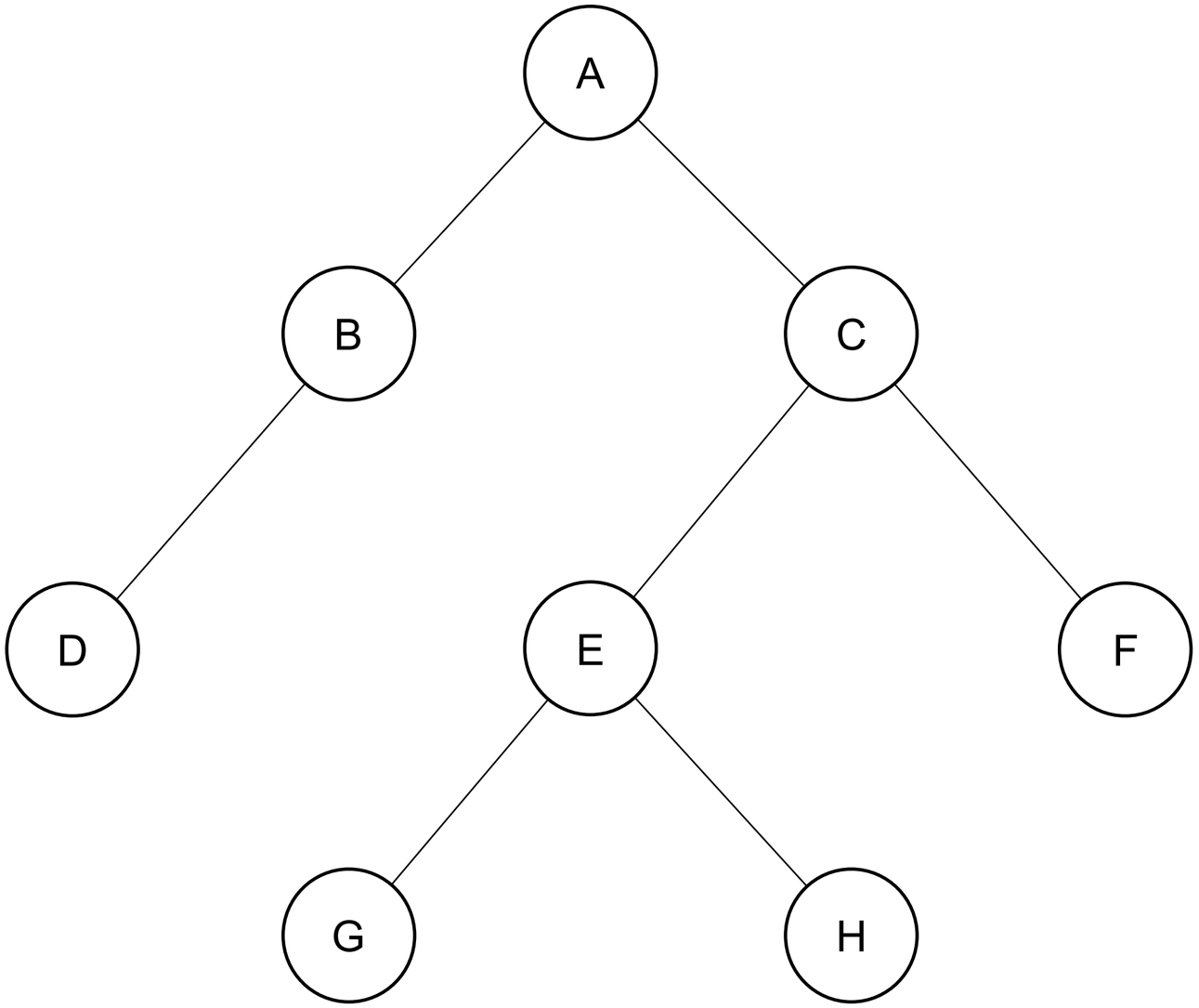
- You can traverse trees in three ways
  - Pre-order
    - Root, Left, Right
  - In-order
    - Left, Root, Right
  - Post-order
    - Left, Right, Root

For each node, read the data of the **node**, then visit the **left** subtree and then the **right** subtree.

For each node, visit the **left** subtree, then read the data of the **node**, then visit the **right** subtree.

For each node, visit the **left** subtree, then visit the **right** subtree, then read the data of the **node**.

Types of traversals



For each node, read the data of the **node**, then visit the **left** subtree and then the **right** subtree.

A → B → D → C → E → G → H → F

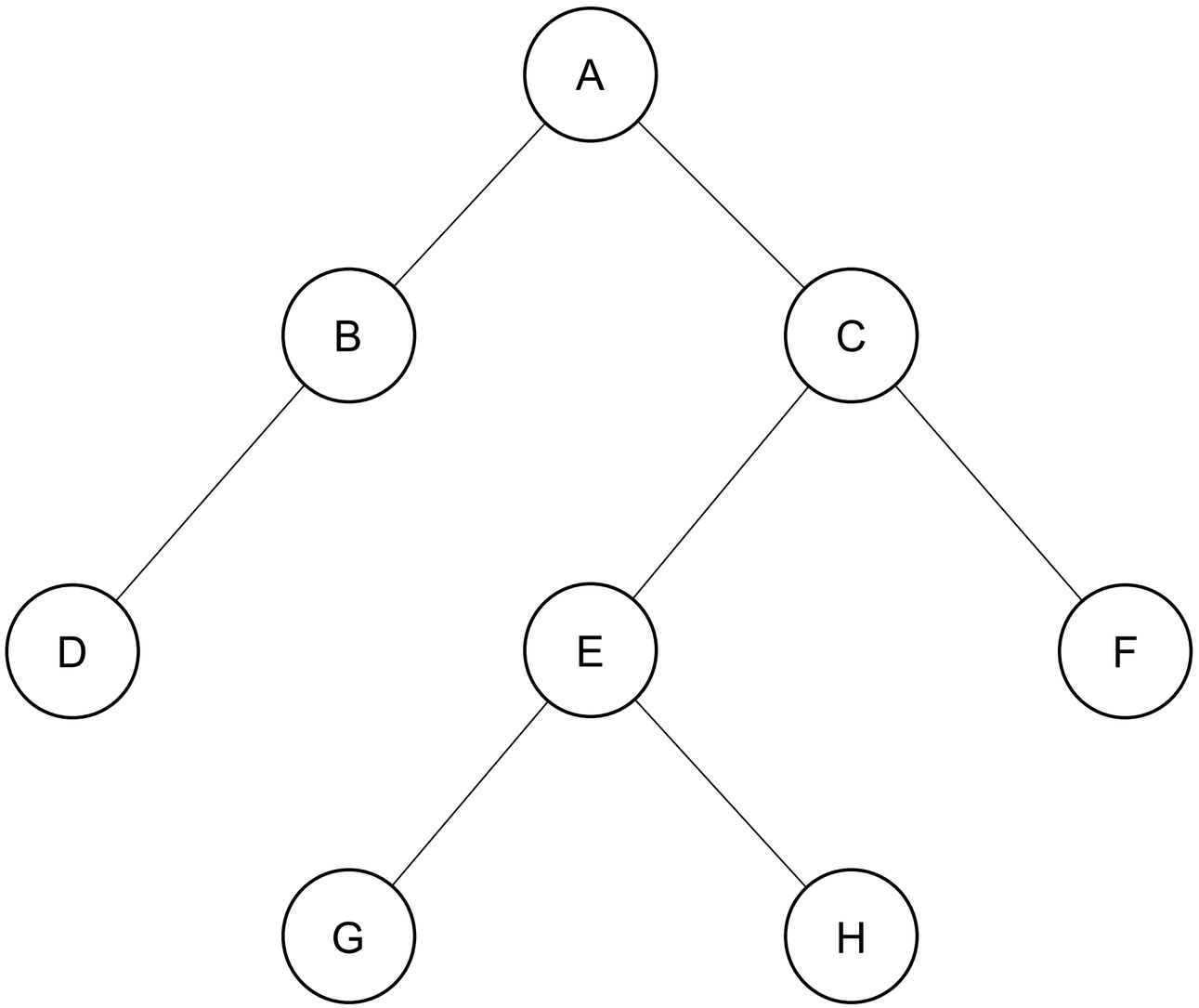
For each node, visit the **left** subtree, then read the data of the **node**, then visit the **right** subtree.

D → B → A → G → E → H → C → F

For each node, visit the **left** subtree, then visit the **right** subtree, then read the data of the **node**.

D → B → G → H → E → F → C → A

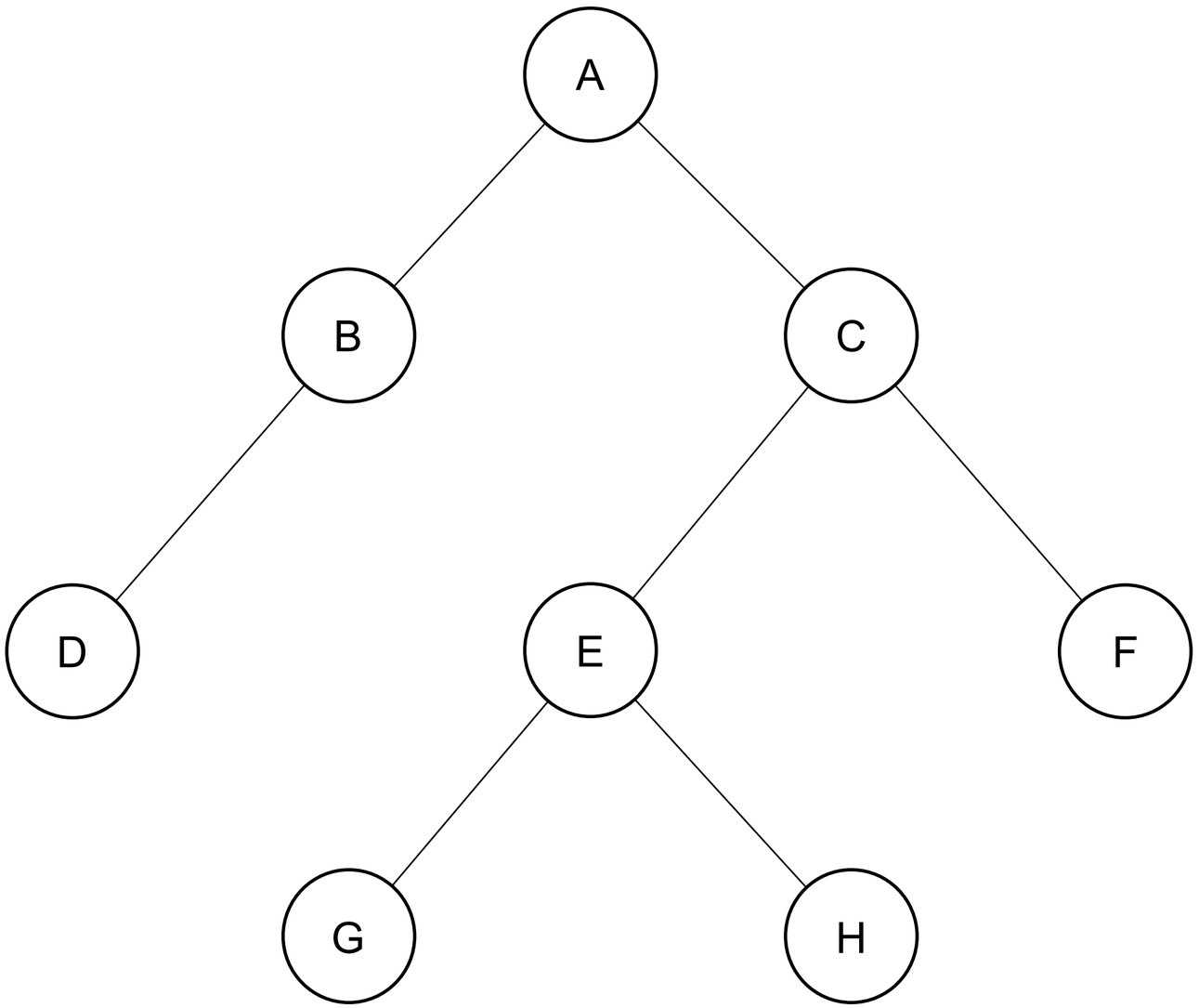
Breadth-first traversal



- The previous are called **depth-first** traversals. Could also do a **breadth-first** traversal.
  - Traverse through all the children of a node, then visit the grandchildren.

$A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G \rightarrow H$

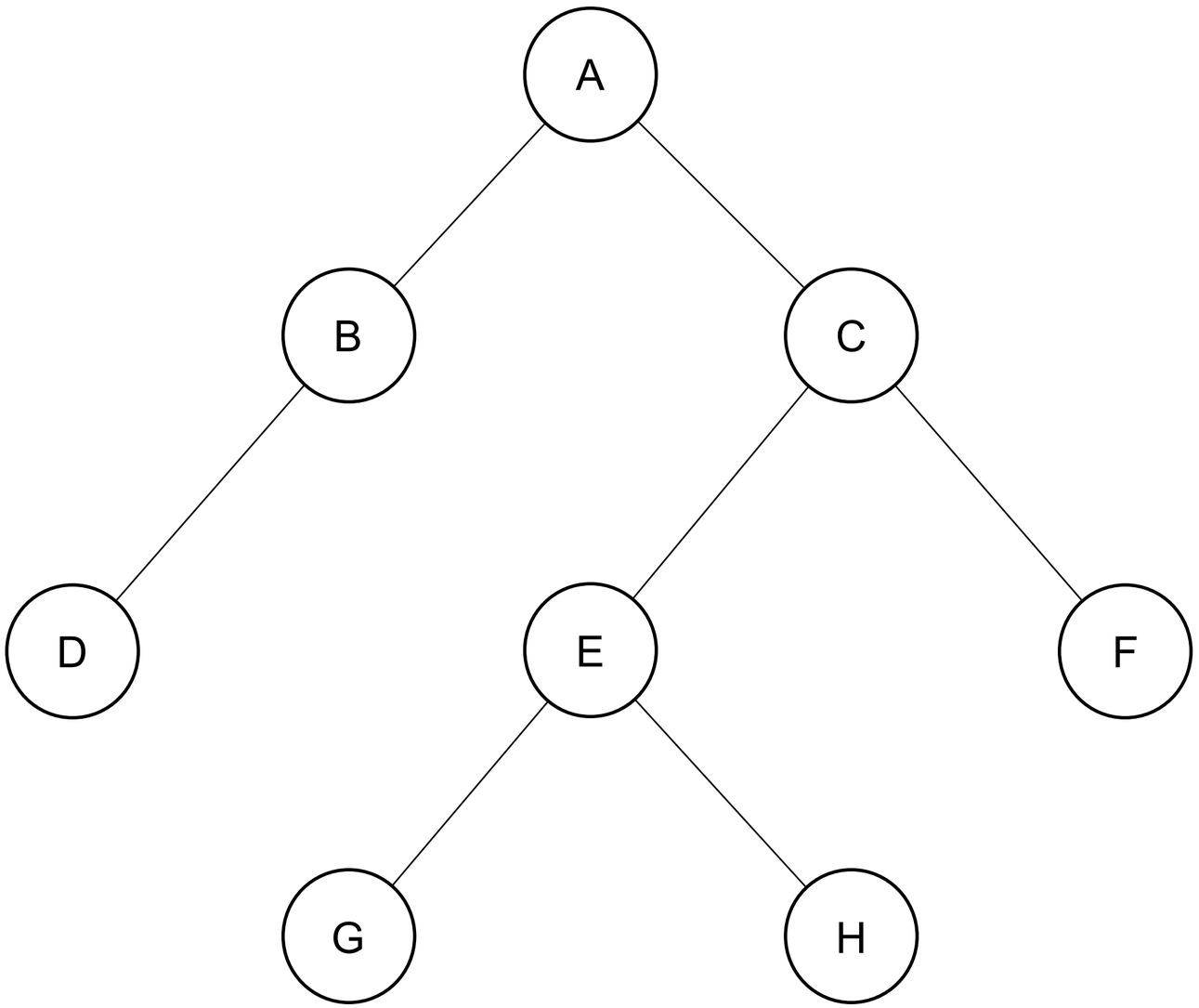
Traversal using stack




```
Stack myStack
myStack.push(root)
while(myStack){
    cursor = myStack.pop()
    cursor.print()
    if (cursor->right)
     myStack.push(cursor->right)
    if (cursor->left)
     myStack.push(cursor->left)
}
```

What does this algorithm do?

Traversal using queue




```
Queue myQueue
myQueue.enqueue(root)
while(myQueue){
     cursor = myQueue.dequeue()
     cursor.print()
     if (cursor->left)
          myQueue.enqueue(cursor->left)
     if (cursor->right)
          myQueue.enqueue(cursor->right)
}
```

What does this algorithm do?

```
typedef struct node{
    int data;
    struct node *left;
    struct node *right;
} node;
```

Practice time

- Write functions to:
  - Add to the left and right of a node.
- Implement preorder, inorder, and postorder traversals.
- Delete a tree.

```
int main(){
  int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  node * root = (node *) malloc(sizeof(node));
  root->data = arr[0];
  root->left = NULL;
  root->right=NULL;
  node * cursor = root;
  for (int j=0, i=1; i<11; i=i+2, j++){
    add_left(&cursor, arr[i]);
    add_right(&cursor, arr[i+1]);
    cursor = (j%2==0) ? cursor->right : cursor->left;
  }
  print_preorder(root);
  print_inorder(root);
  print_postorder(root);
  delete_tree(root);
}
```

Printing a tree

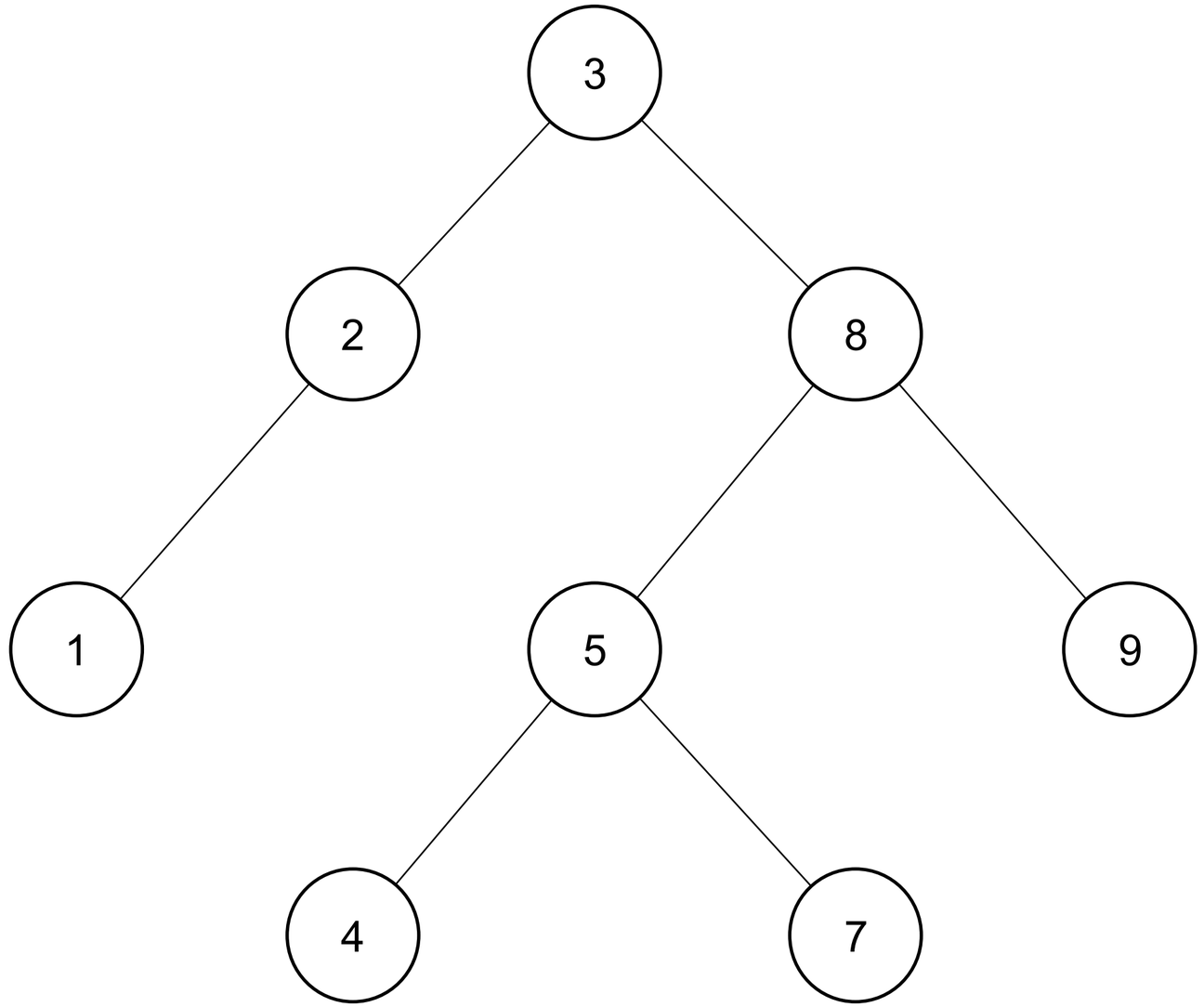
- Can we print a tree in a human readable way?
  - Focus on pre-order traversal
    - Print node, then go left, then go right
- Use *depth* to print right amount of indentation

```
void treeprint(node *cursor, int depth){
  if (cursor == NULL)
    return;
  for (int i = 0; i < depth; i++)
    printf(i == depth - 1 ? "|-" : "  ");
  printf("%d\n", cursor->data);
  treeprint(cursor->left, depth + 1);
  treeprint(cursor->right, depth + 1);
}
```

Let us check if we got previous slide right …

Binary Search Trees

- Binary trees that have a particular *sorted* property are called **binary search trees** (BST)
  - All nodes in the **left subtree** of a given node are *lesser than or equal* to the node
  - All nodes in the **right subtree** of a given node are *greater* than that node



Exercises with BST

- How can you find the minimum or maximum element in a BST?
- How can we search a BST for a node?
- How should you insert a new node in a BST?
- How can you find the height of a *general* tree (can also be BST)?

```
typedef struct node{
    int data;
    struct node *left;
    struct node *right;
} node;
```

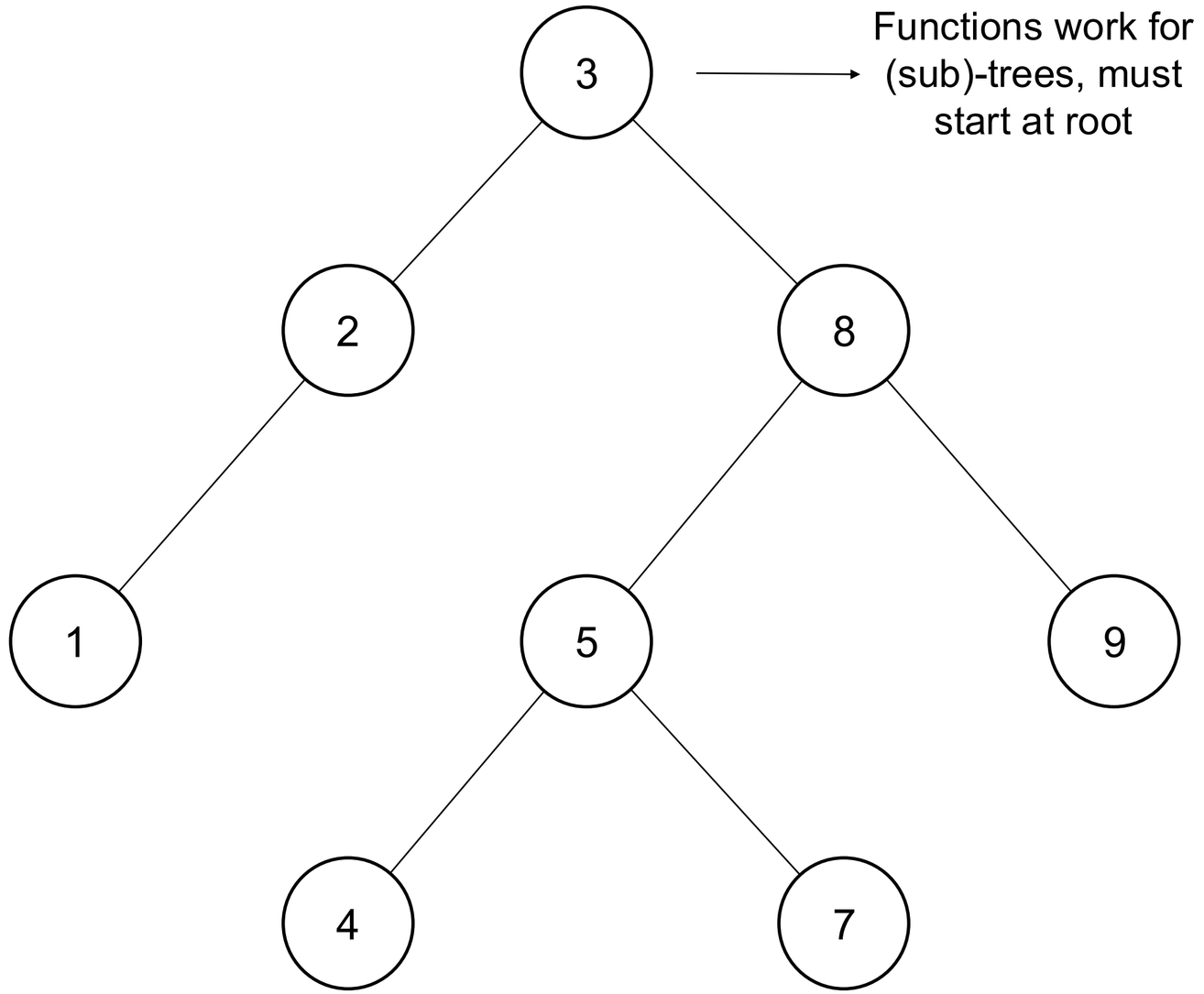
Finding extremals in a BST

Minimum - keep going left

```
node * findmin(node *cursor){
  if (cursor->left==NULL)
    return cursor;
  else
    return findmin(cursor->left);
}
```

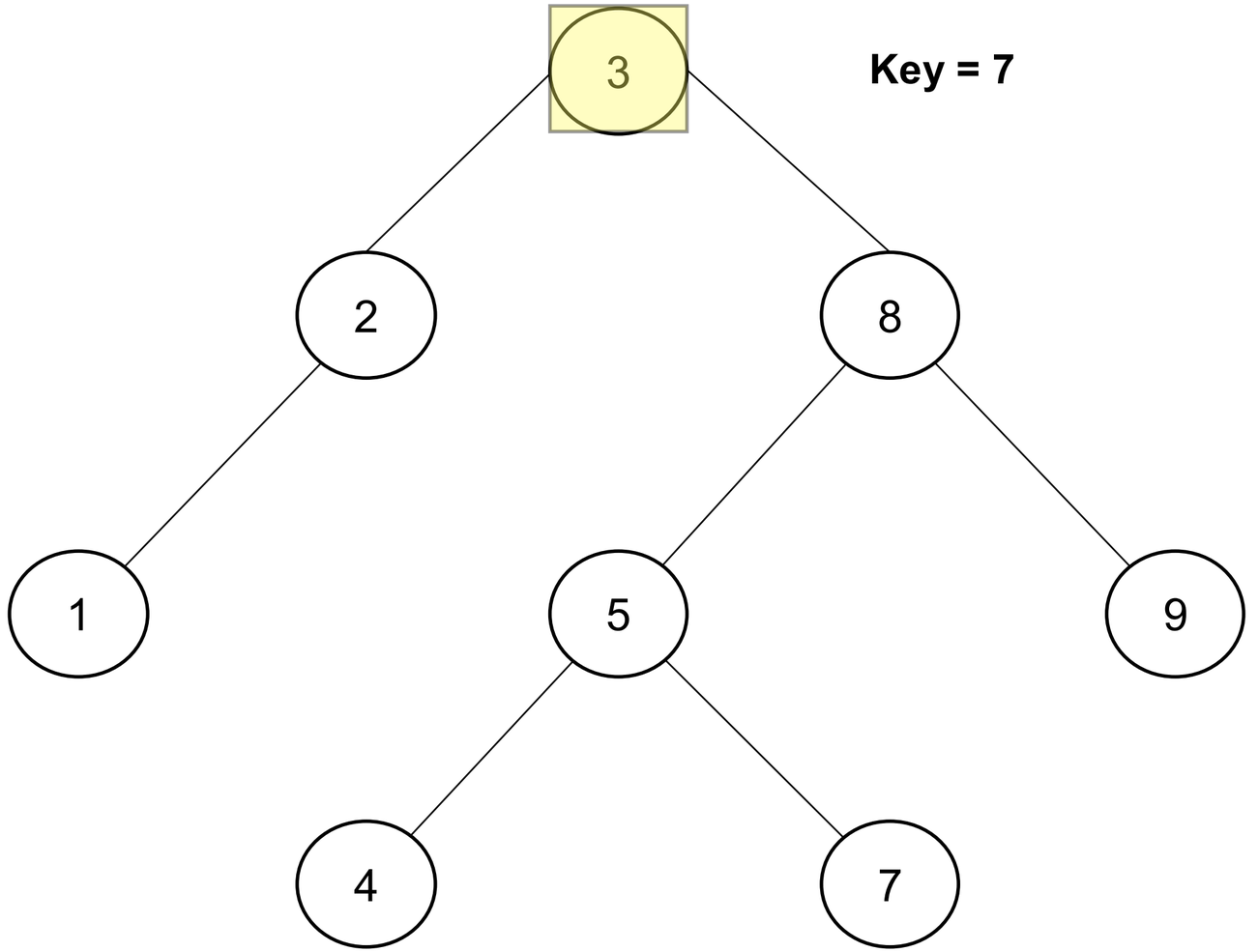
Maximum - keep going right

```
node * findmax(node *cursor){
  if (cursor->right==NULL)
    return cursor;
  else
    return findmax(cursor->right);
}
```



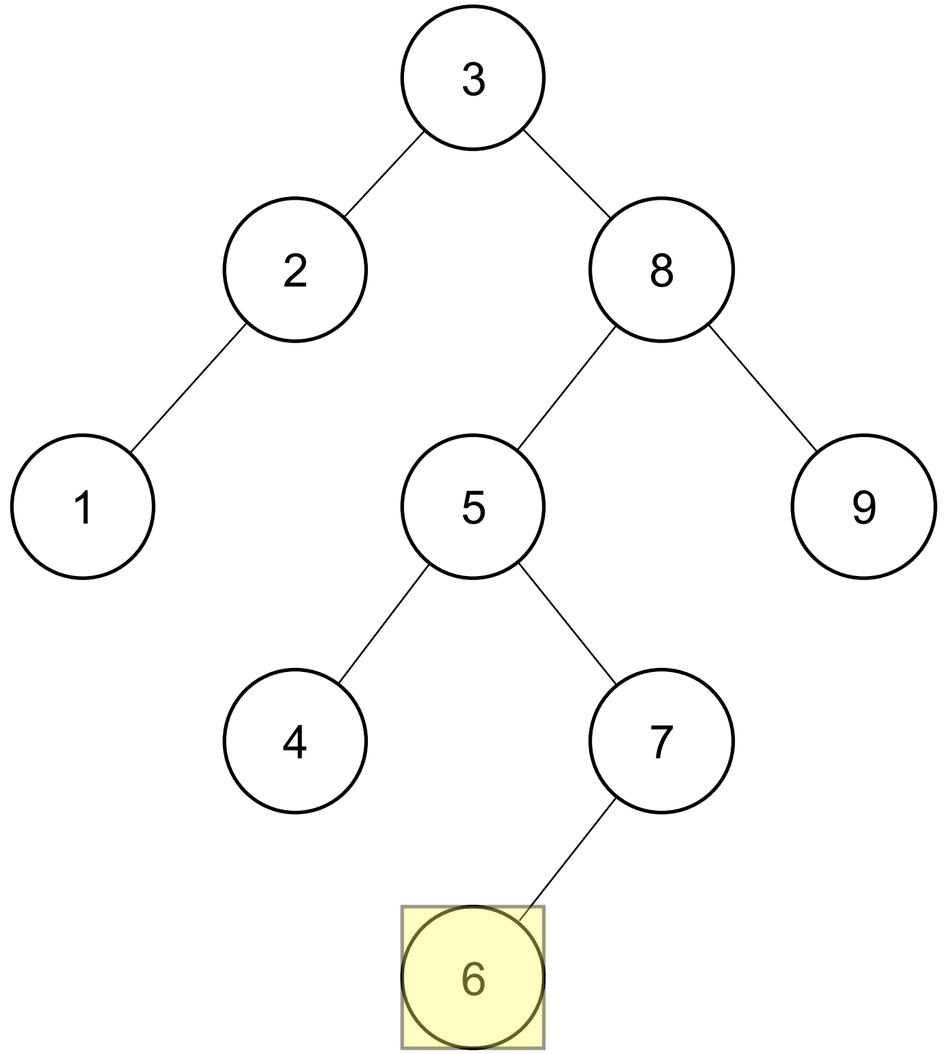
Searching in a BST

```
node * find_elem(node *cursor, int key){
  if (cursor==NULL) // Key not found
    return NULL;
  if (cursor->data == key)
    return cursor;  // Found key
  if (cursor->data < key)
    // Go right
    return find_elem(cursor->right, key);
  else
    // Go left
    return find_elem(cursor->left, key);
}
```



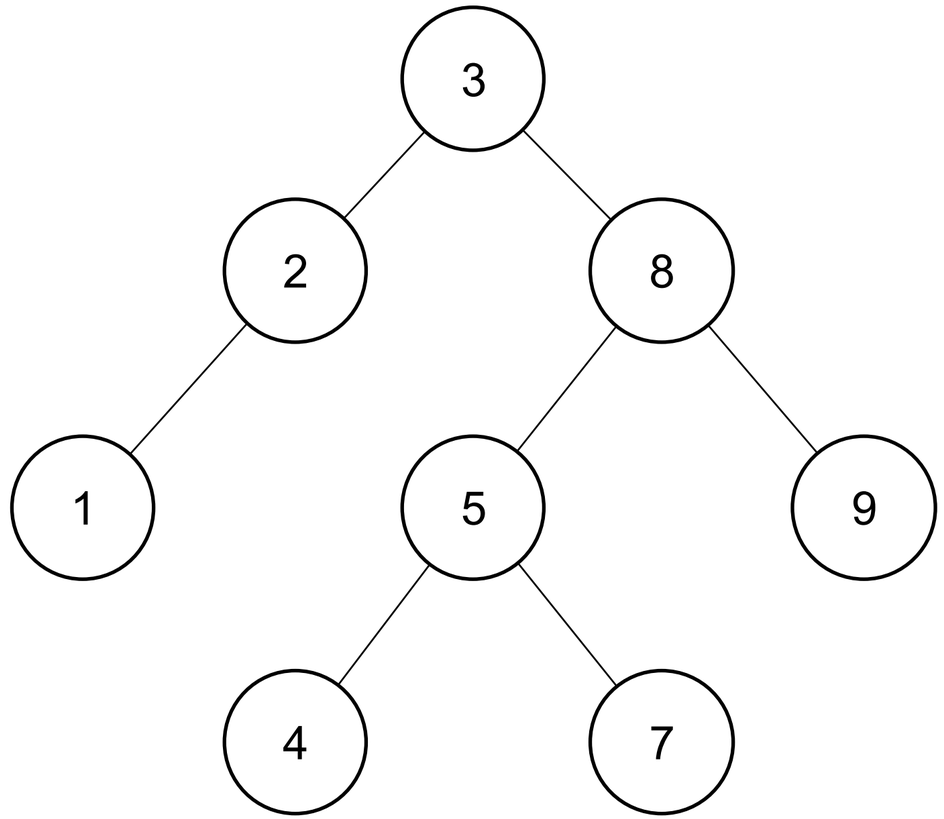
Insertion in a BST

- Insertions need to preserve the BST property
- Add new nodes only as leaf nodes
- Consider inserting 6 in the BST on the right …



Insertion in a BST - code

```
node * insert(node *cursor, int data){
if (cursor==NULL)
  return newNode(data);
else{
    if (data < cursor->data)
      cursor->left = insert(cursor->left, data);
    else
      cursor->right = insert(cursor->right, data);
    return cursor;
  }
}
```

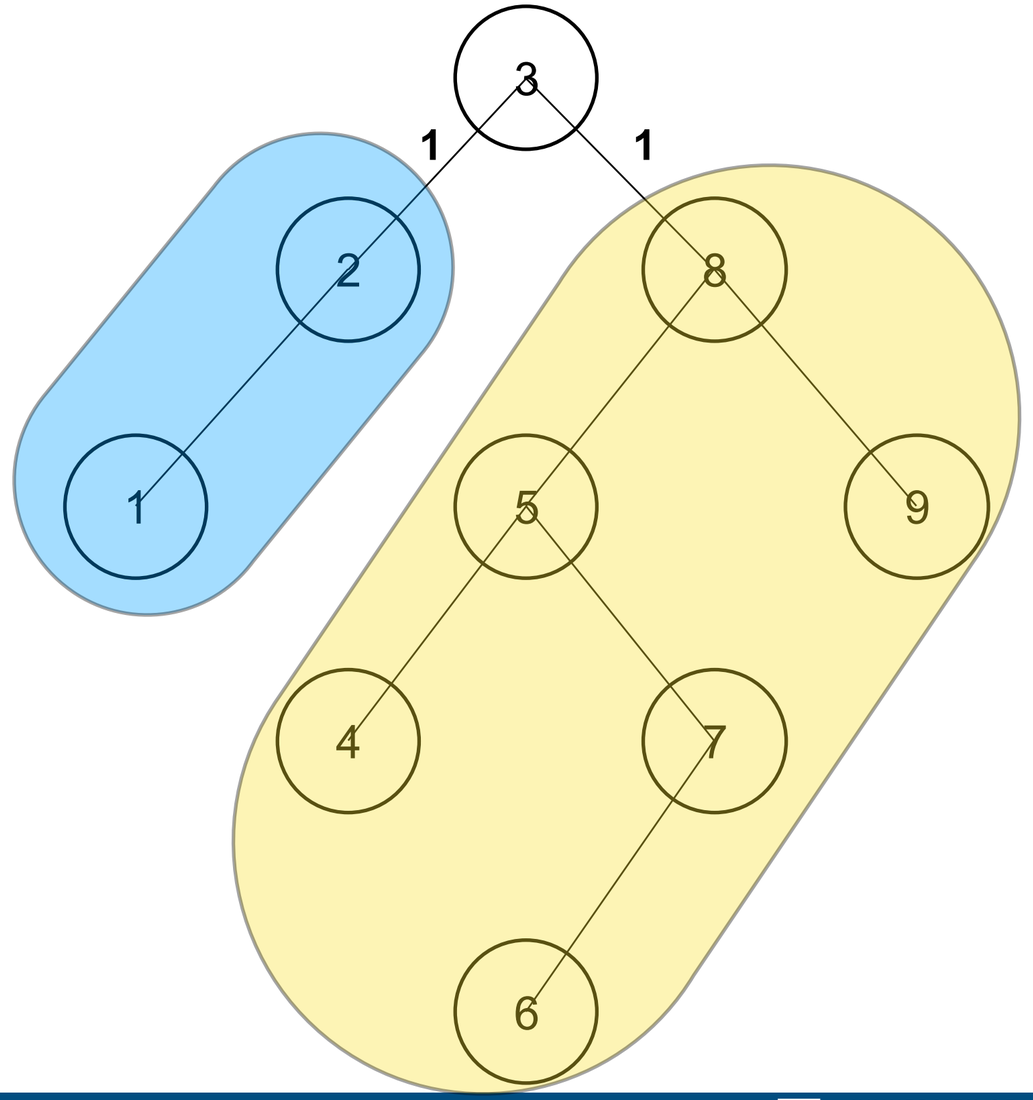


Finding height of a tree

- Height is length of *longest* path from root to leaf(s)

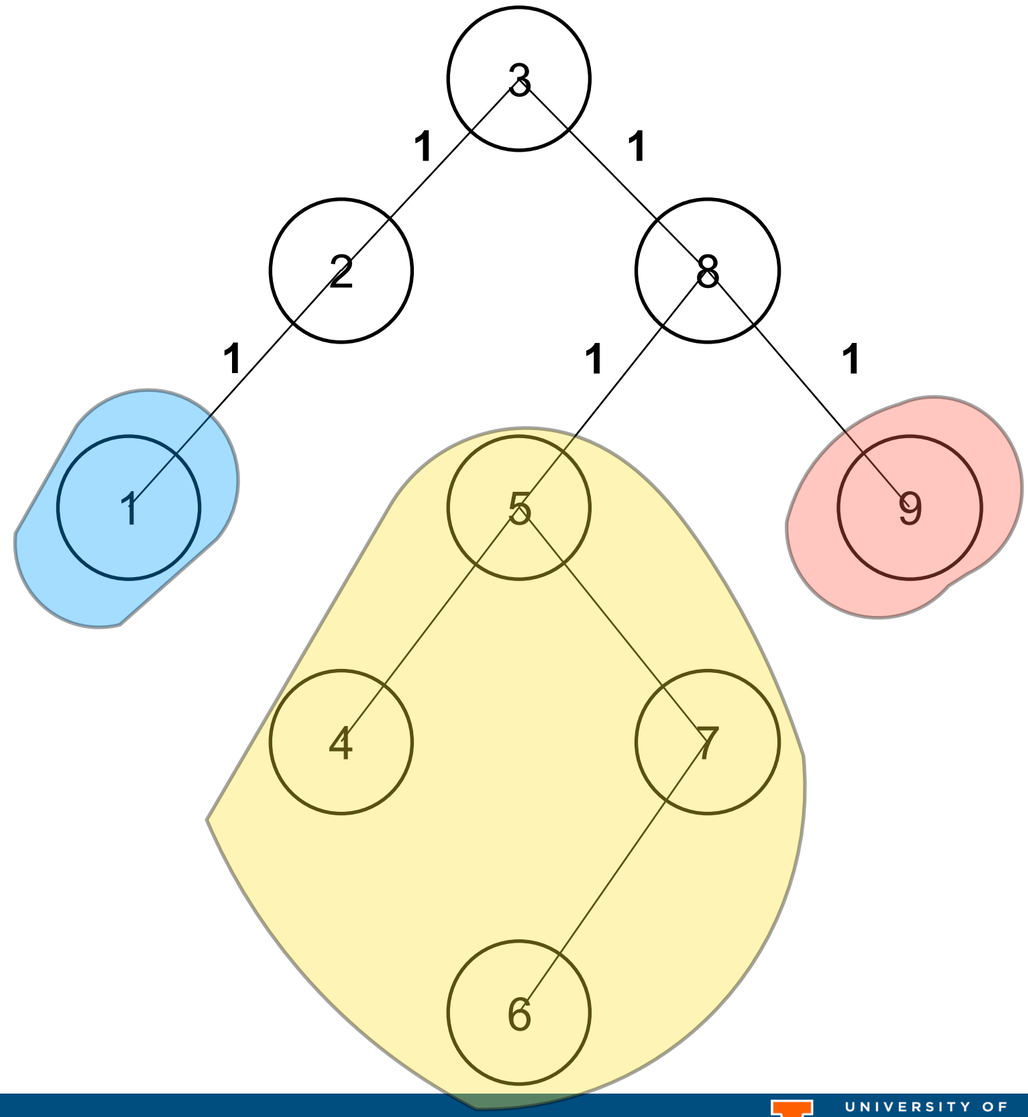
- Recursively calculate: 1 + height of L/R subtree(s)

- Take maximum at each step



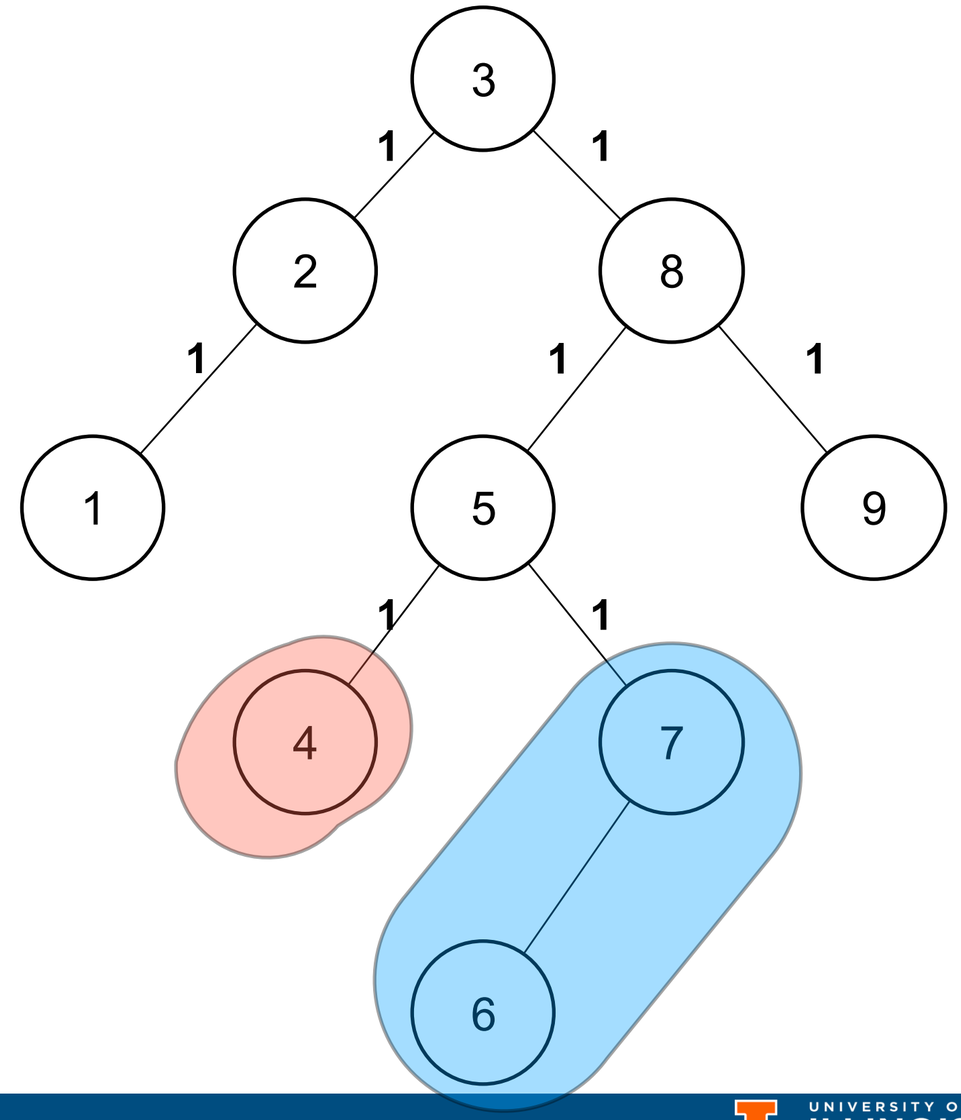
Finding height of a tree

- Height is length of *longest* path from root to leaf(s)
  - Recursively calculate: 1 + height of L/R subtree(s)
  - Take maximum at each step



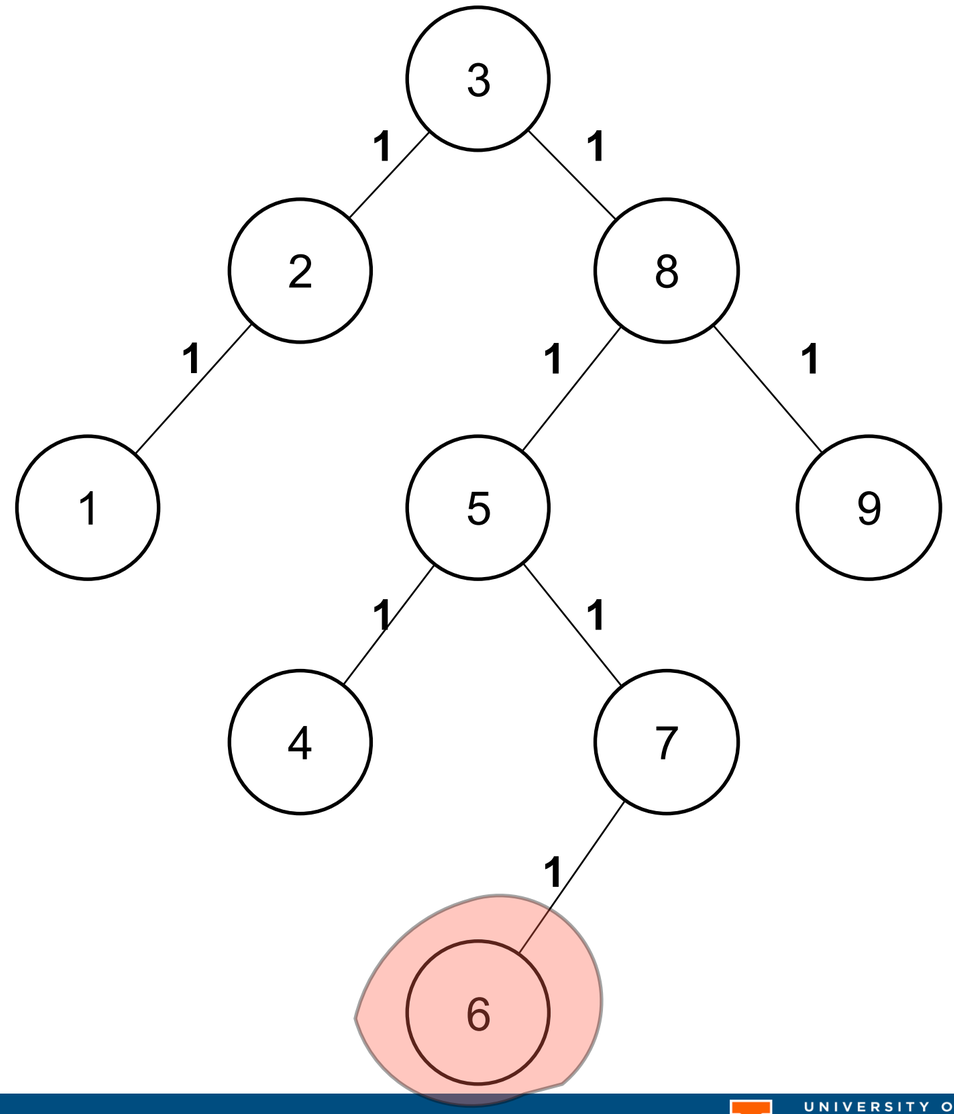
Finding height of a tree

- Height is length of *longest* path from root to leaf(s)
  - Recursively calculate: 1 + height of L/R subtree(s)
  - Take maximum at each step



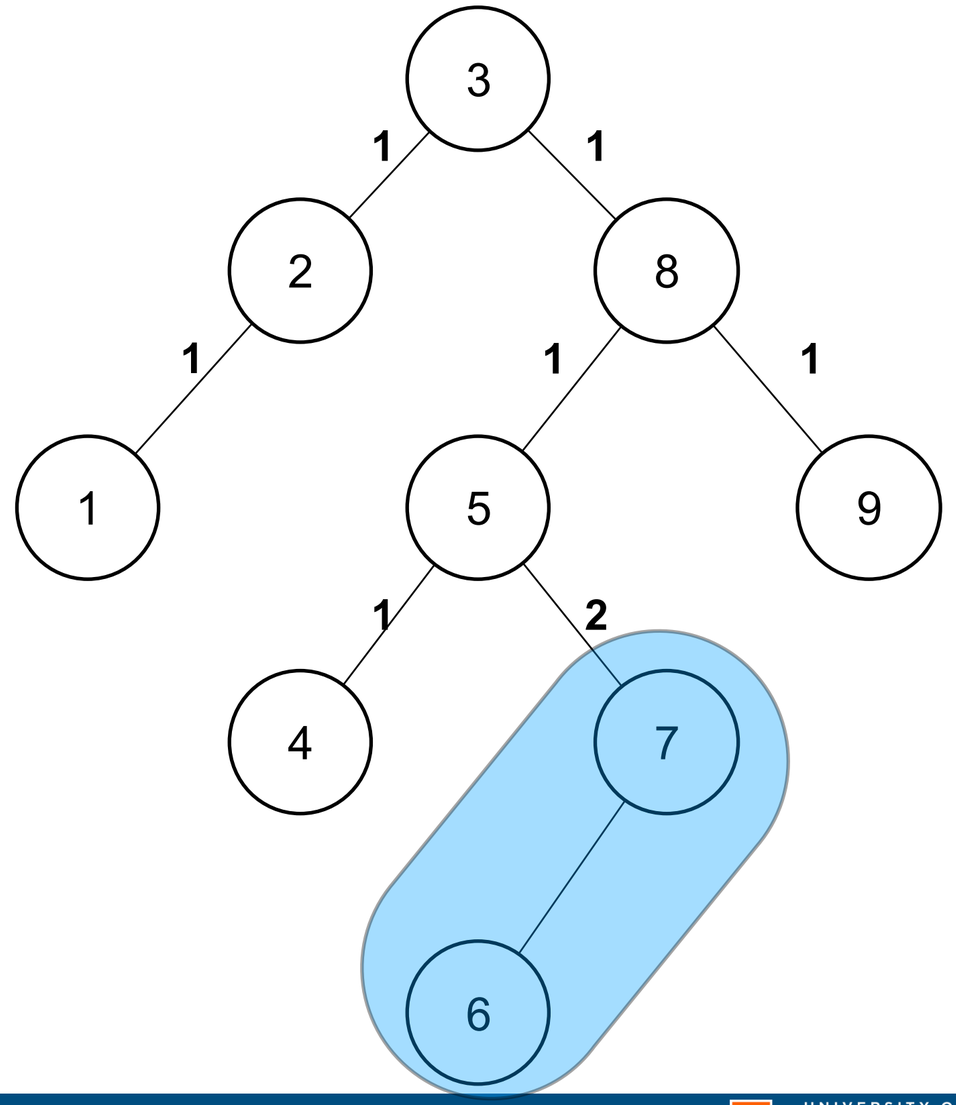
Finding height of a tree

- Height is length of *longest* path from root to leaf(s)
  - Recursively calculate: 1 + height of L/R subtree(s)
  - Take maximum at each step



Finding height of a tree

- Height is length of *longest* path from root to leaf(s)
  - Recursively calculate: 1 + height of L/R subtree(s)
  - Take maximum at each step

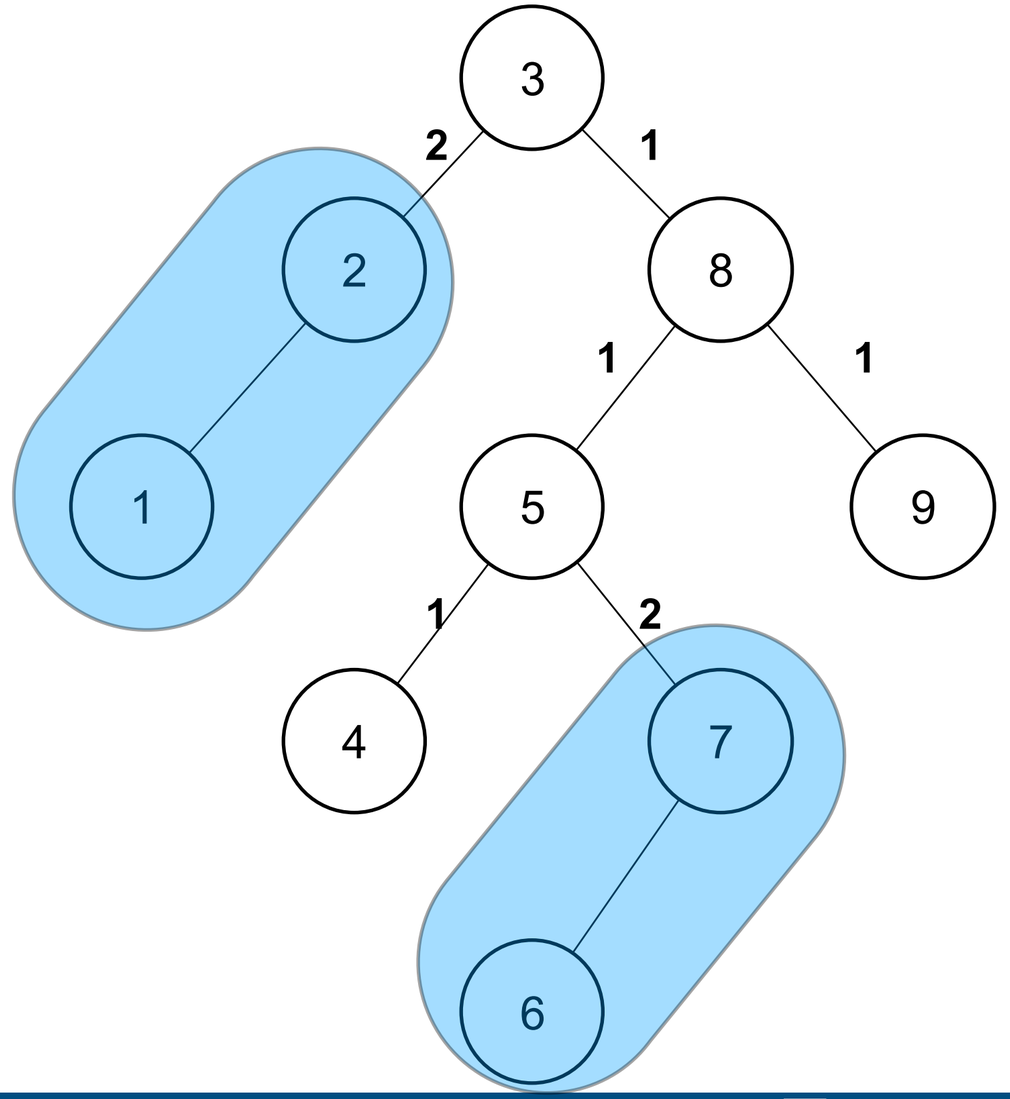


Finding height of a tree

- Height is length of *longest* path from root to leaf(s)

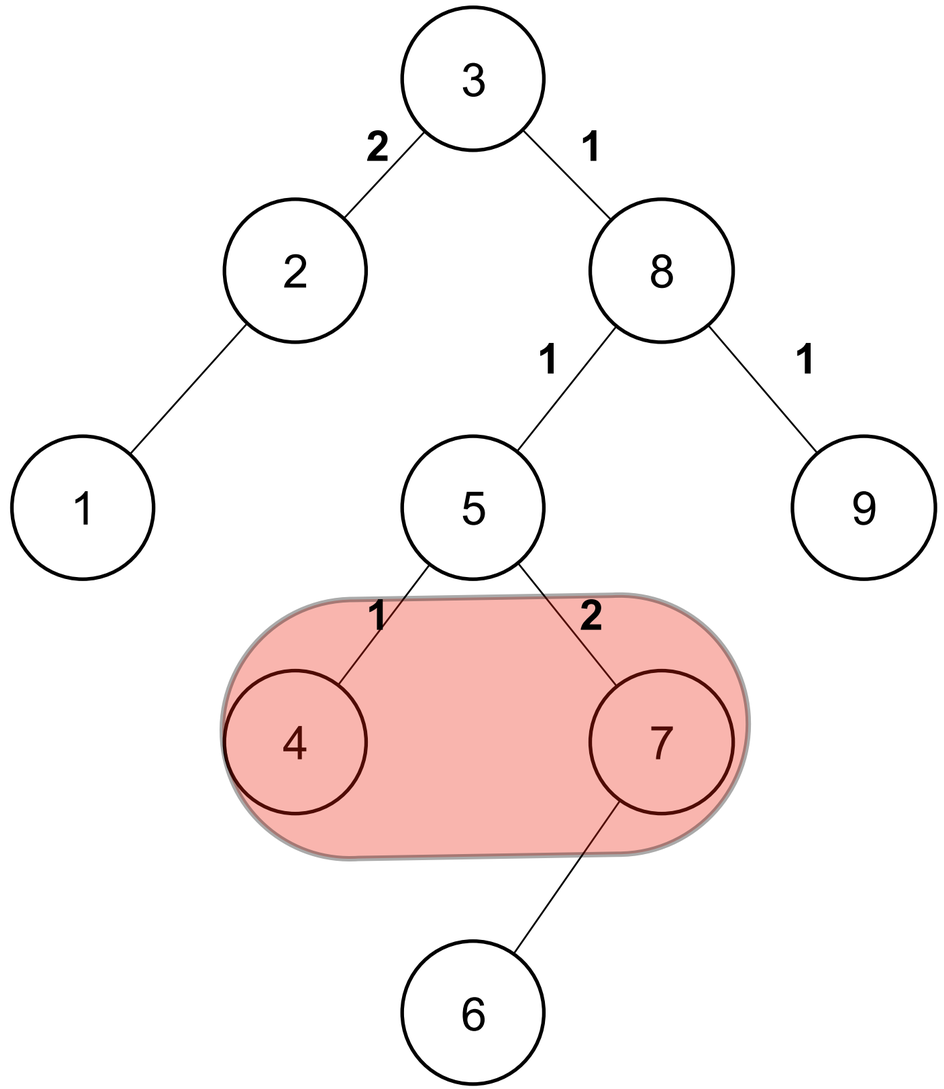
- Recursively calculate: 1 + height of L/R subtree(s)

- Take maximum at each step



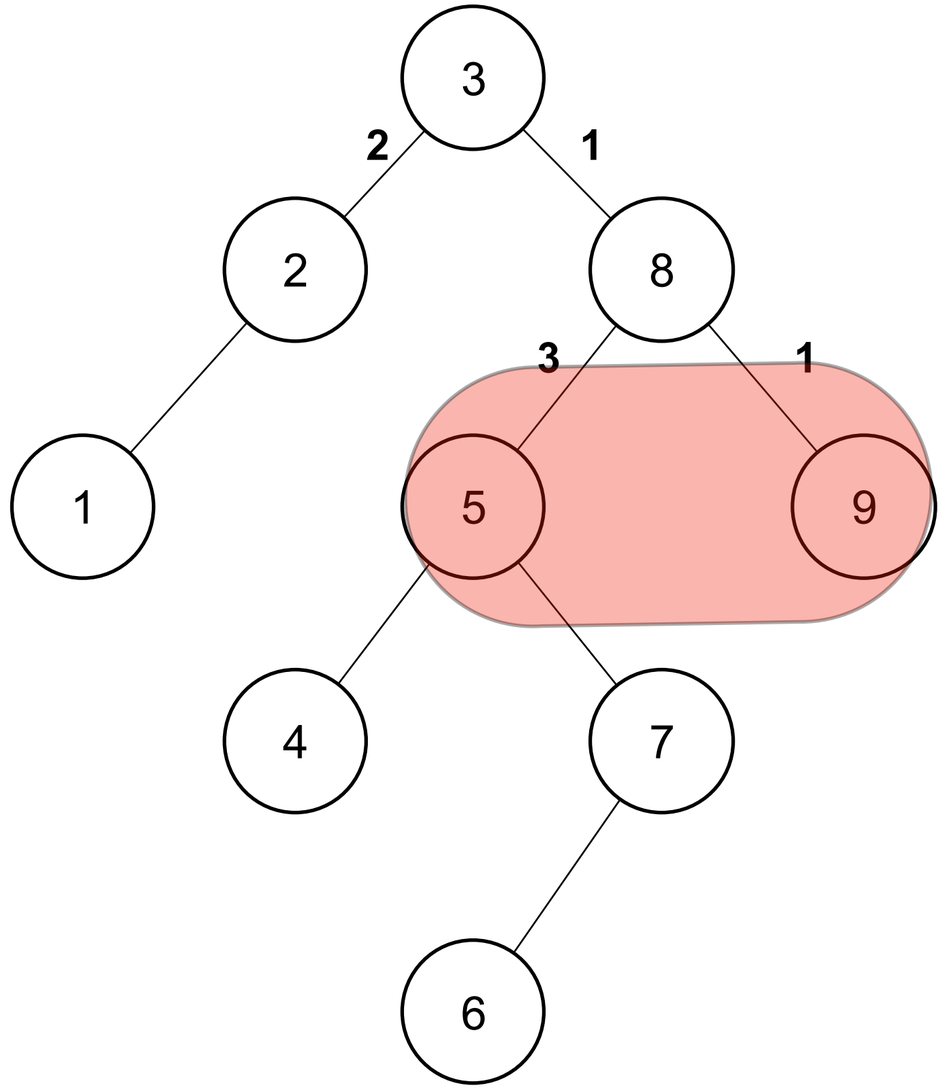
Finding height of a tree

- Height is length of *longest* path from root to leaf(s)
  - Recursively calculate: 1 + height of L/R subtree(s)
  - Take maximum at each step



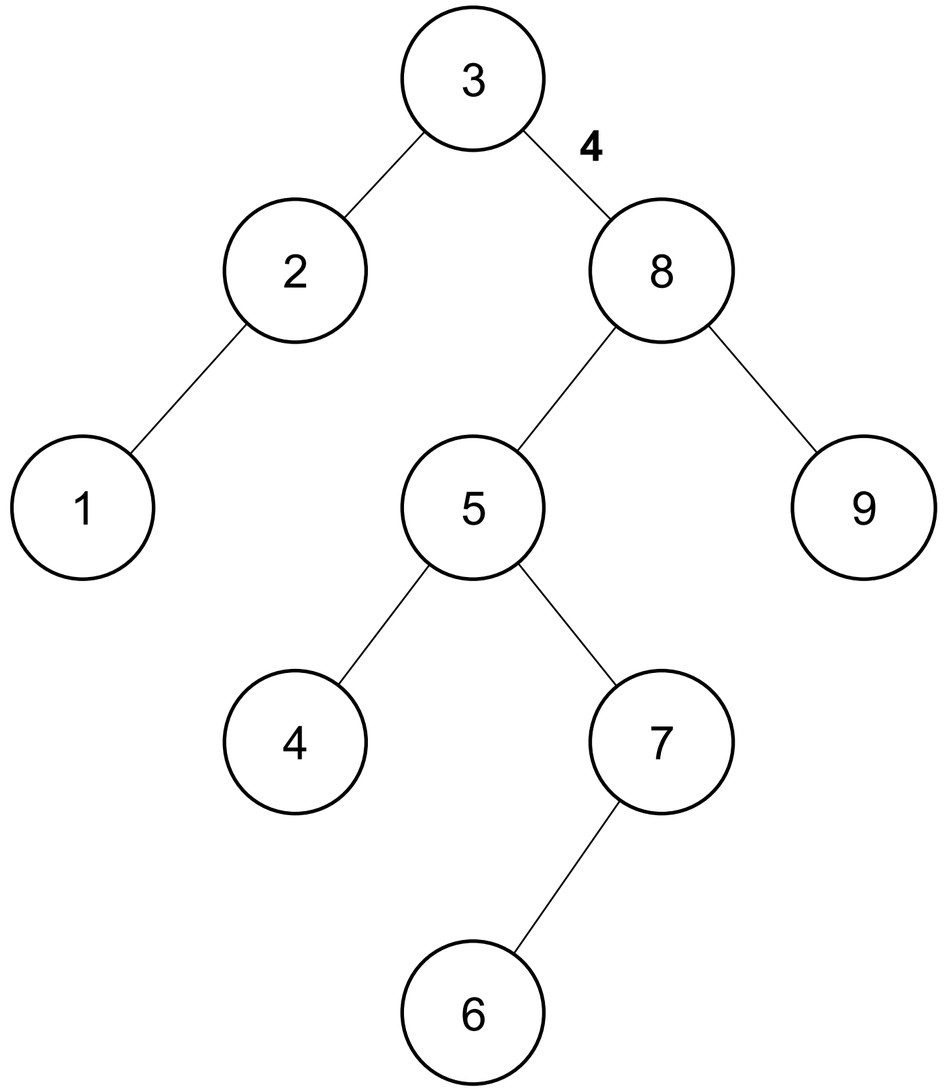
Finding height of a tree

- Height is length of *longest* path from root to leaf(s)
  - Recursively calculate: 1 + height of L/R subtree(s)
  - Take maximum at each step



Finding height of a tree

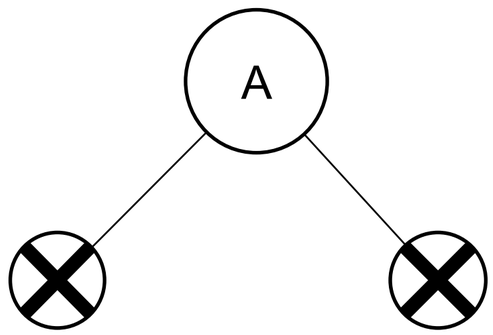
- Height is length of *longest* path from root to leaf(s)
  - Recursively calculate: 1 + height of L/R subtree(s)
  - Take maximum at each step

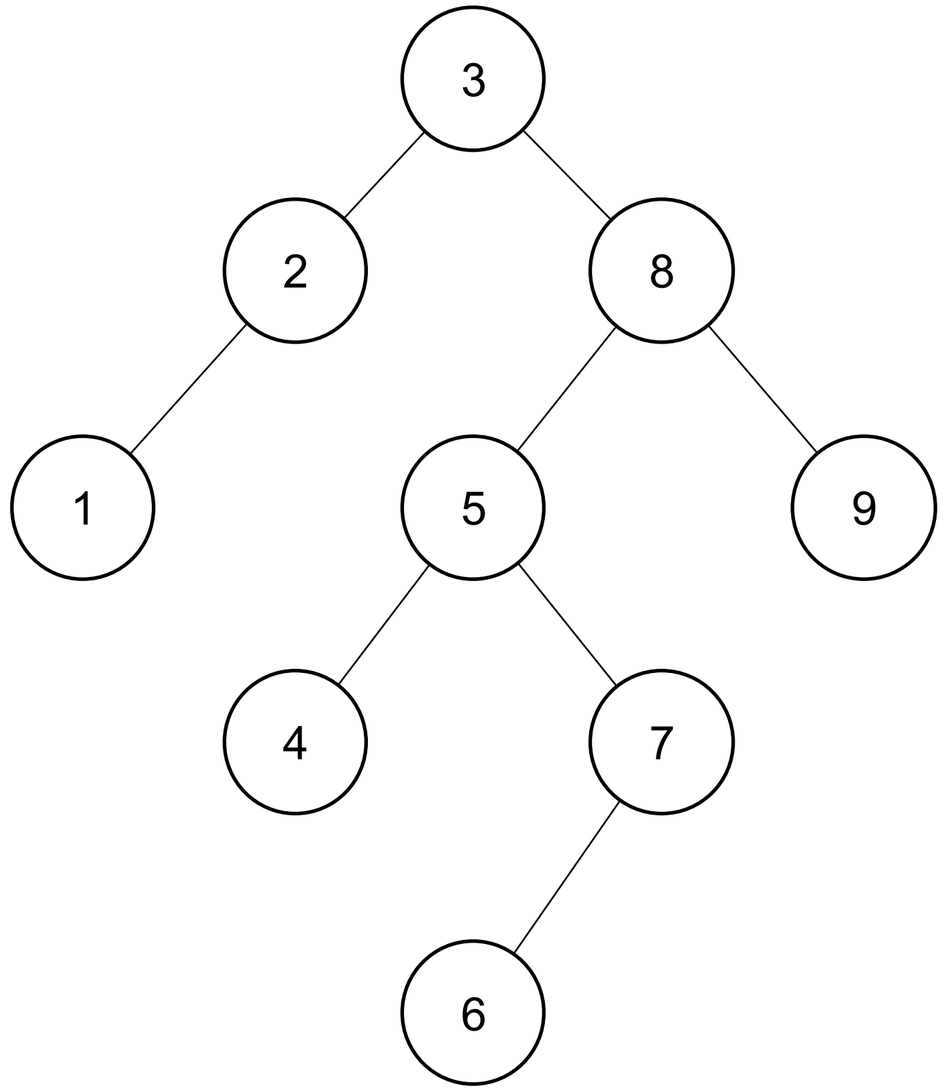


Finding height of a tree

```
int tree_height(node *cursor){
  int lh, rh;
  if (cursor==NULL)
    return -1;
  else{
    lh = 1 + tree_height(cursor->left);
    rh = 1 + tree_height(cursor->right);
    return (lh > rh ? lh : rh);
  }
}
```

What should be height of single node?





```
template <typename T>
struct treenode{
  T data;
  treenode *left;
  treenode *right;
};
```

Next class …

- Using classes in C++, create a ***templated*** BST class and perform or find:
  - Insertion
  - Searching
  - Traversal
  - Vectorization
  - Size of tree (# of nodes)
  - Find height of the tree
  - Deletion of tree

```
template <class N>
class bst{
private:
  …
  …
  …
public:
  bst();
  void insert(N data);
  treenode<N> *search(N data);
  void inorder();
  vector<N> vectorize();
  int node_count();
  int height();
  void print();
  ~bst();
};
```