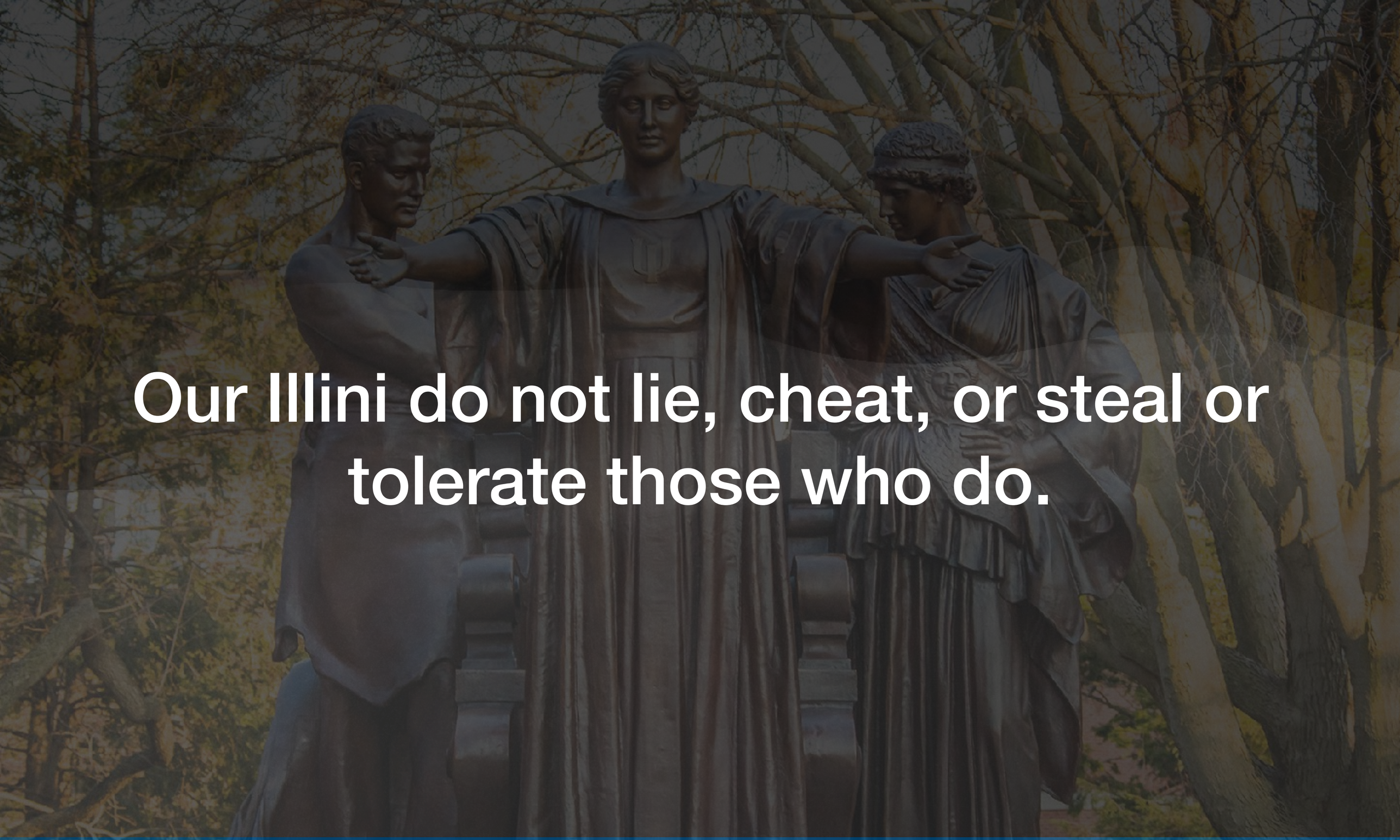
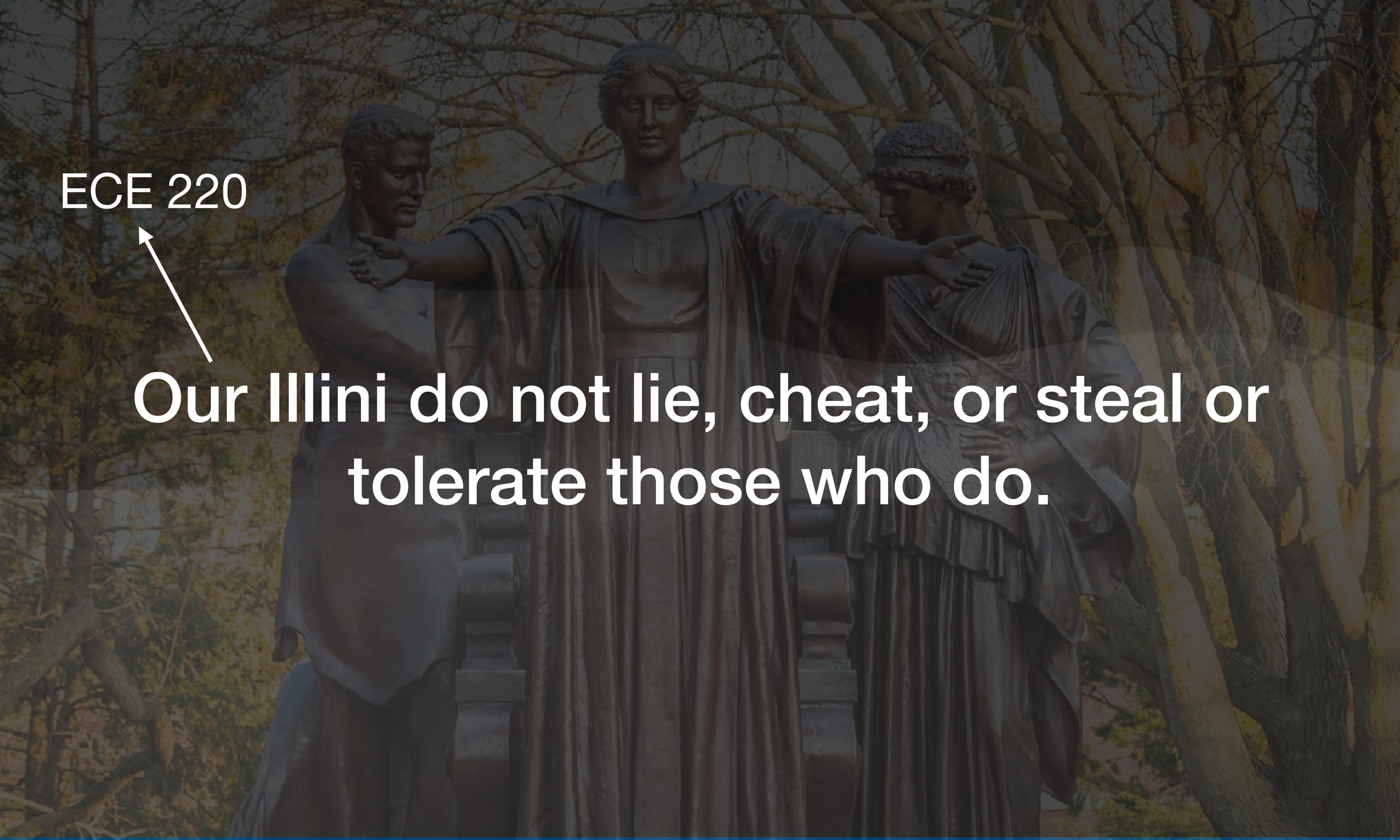
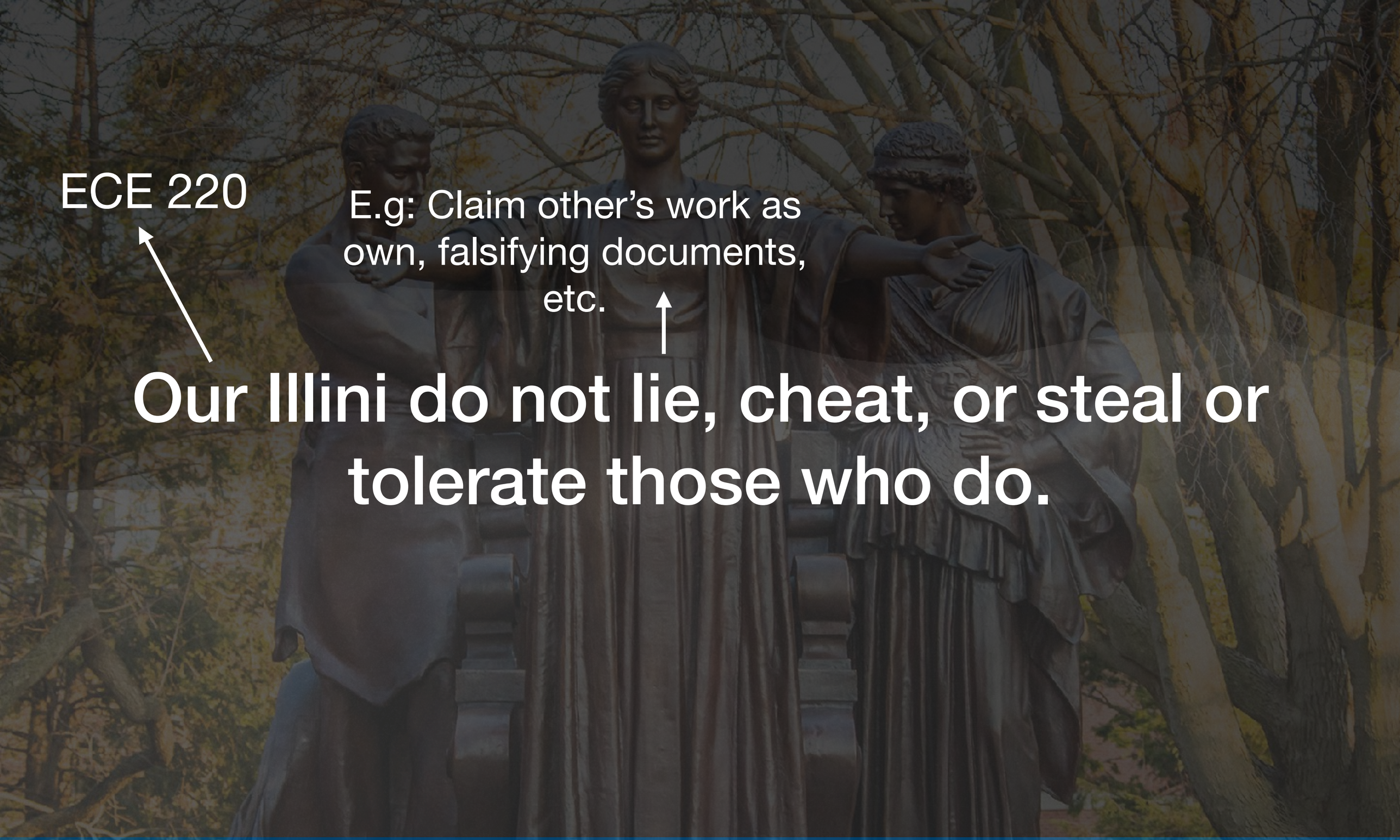
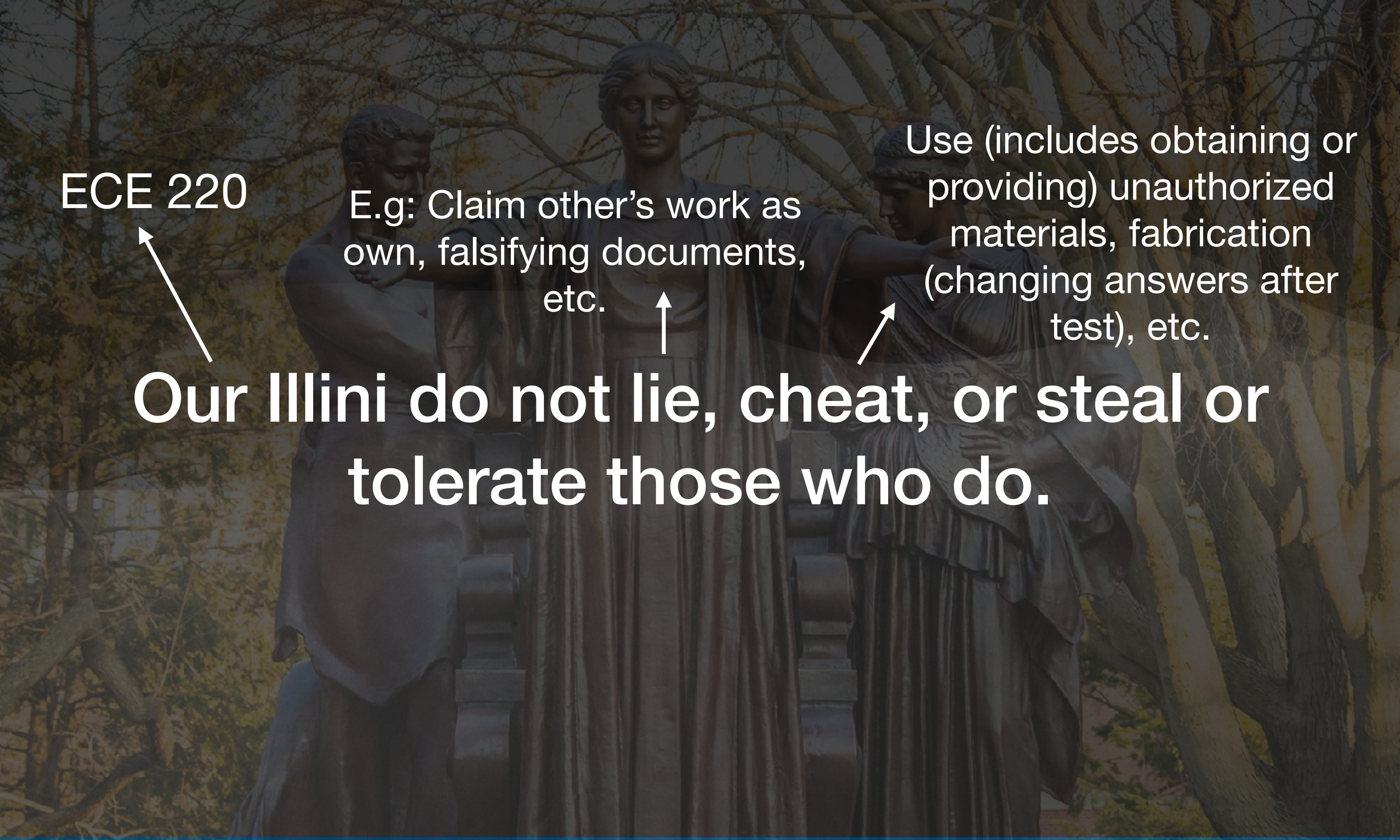
ECE 220

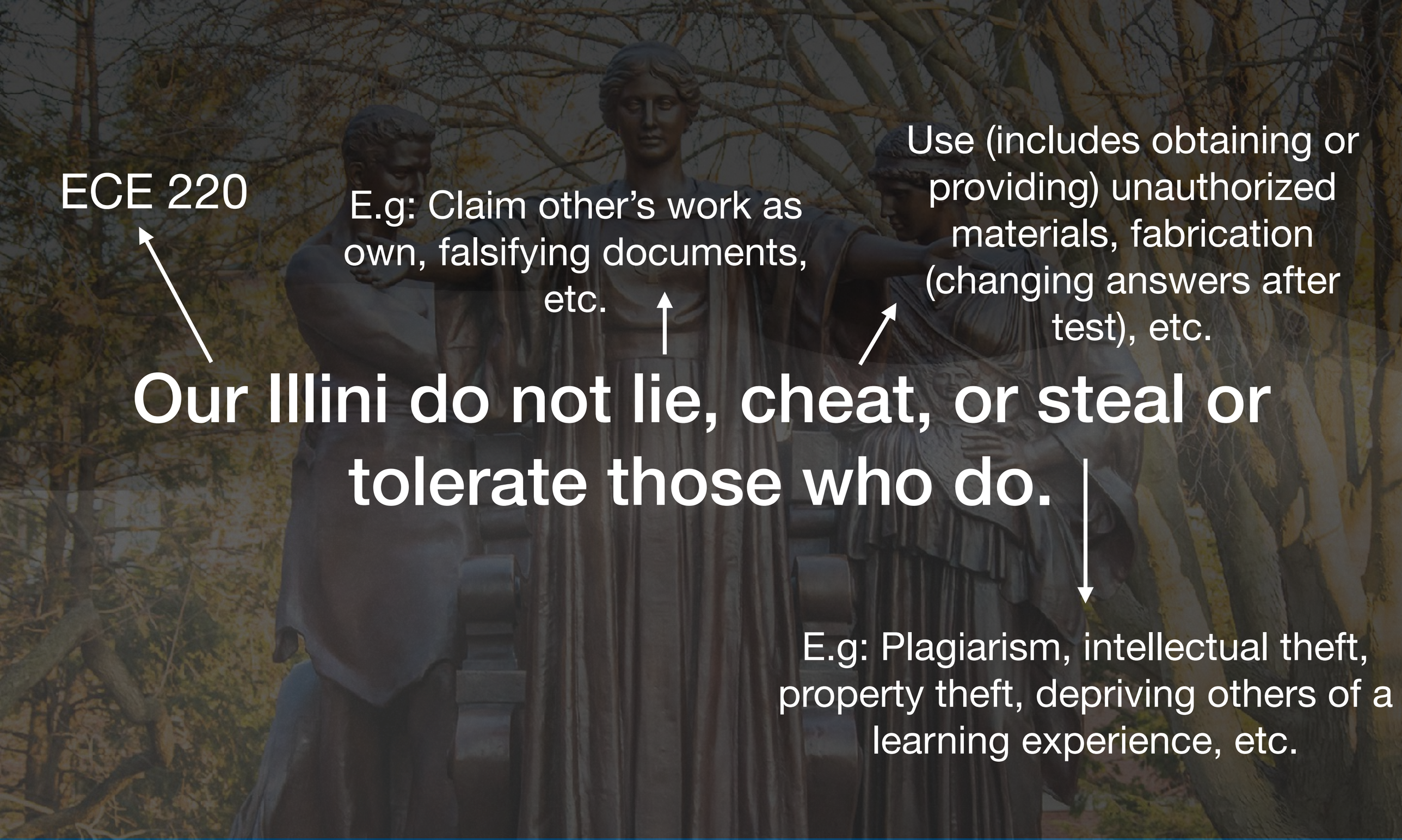
Lecture x0017 - 04/16
More trees (BST) and C++

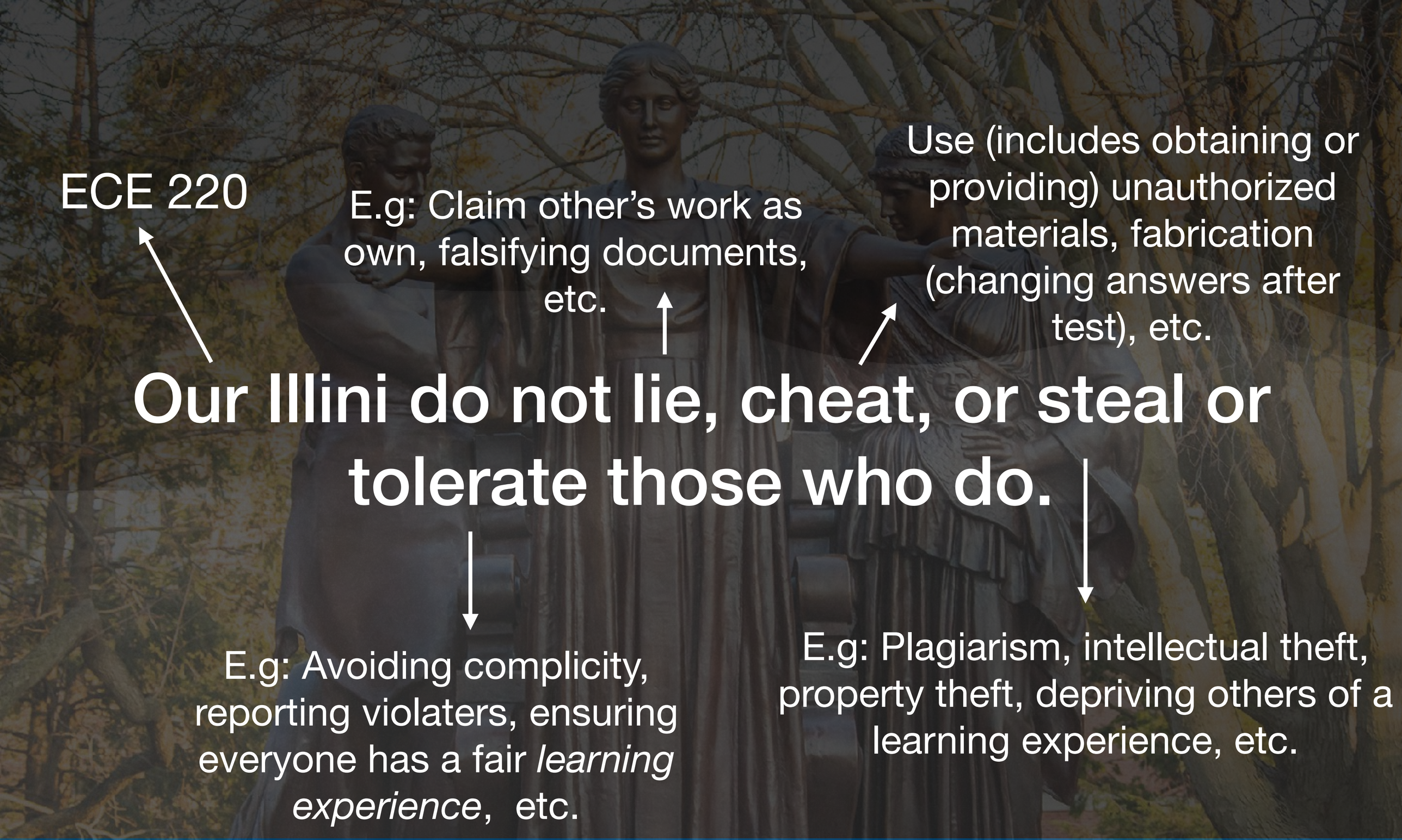

Our Illini do not lie, cheat, or steal or
tolerate those who do.


ECE 220
Our Illini do not lie, cheat, or steal or
tolerate those who do.


ECE 220
E.g: Claim other's work as own, falsifying documents, etc.
Our Illini do not lie, cheat, or steal or tolerate those who do.


ECE 220
E.g: Claim other's work as own, falsifying documents, etc.
Use (includes obtaining or providing) unauthorized materials, fabrication (changing answers after test), etc.
Our Illini do not lie, cheat, or steal or tolerate those who do.


ECE 220
E.g: Claim other's work as own, falsifying documents, etc.
Use (includes obtaining or providing) unauthorized materials, fabrication (changing answers after test), etc.
Our Illini do not lie, cheat, or steal or tolerate those who do.
E.g: Plagiarism, intellectual theft, property theft, depriving others of a learning experience, etc.


ECE 220
E.g: Claim other's work as own, falsifying documents, etc.
Use (includes obtaining or providing) unauthorized materials, fabrication (changing answers after test), etc.
Our Illini do not lie, cheat, or steal or tolerate those who do.
E.g: Avoiding complicity, reporting violaters, ensuring everyone has a fair *learning experience*, etc.
E.g: Plagiarism, intellectual theft, property theft, depriving others of a learning experience, etc.

Recap

Recap

- Previously

Recap

- Previously
  - Tree & tree concepts

Recap

- Previously
  - Tree & tree concepts
  - Trees

Recap

- Previously
  - Tree & tree concepts
  - Trees
    - Nodes - root/leaf

Recap

- Previously
  - Tree & tree concepts
- Trees
  - Nodes - root/leaf
  - Relations - child/parent/ sibling

Recap

- Previously
  - Tree & tree concepts
  - Trees
    - Nodes - root/leaf
    - Relations - child/parent/ sibling
    - Properties - depth/height

Recap

- Previously
  - Tree & tree concepts
  - Trees
    - Nodes - root/leaf
    - Relations - child/parent/ sibling
    - Properties - depth/height
    - Traversals - three kinds

Recap

- Previously
  - Tree & tree concepts
  - Trees
    - Nodes - root/leaf
    - Relations - child/parent/ sibling
    - Properties - depth/height
    - Traversals - three kinds
    - Using stacks/queues

Recap

- Previously
  - Tree & tree concepts
  - Trees
    - Nodes - root/leaf
    - Relations - child/parent/ sibling
    - Properties - depth/height
    - Traversals - three kinds
    - Using stacks/queues
    - Recursion

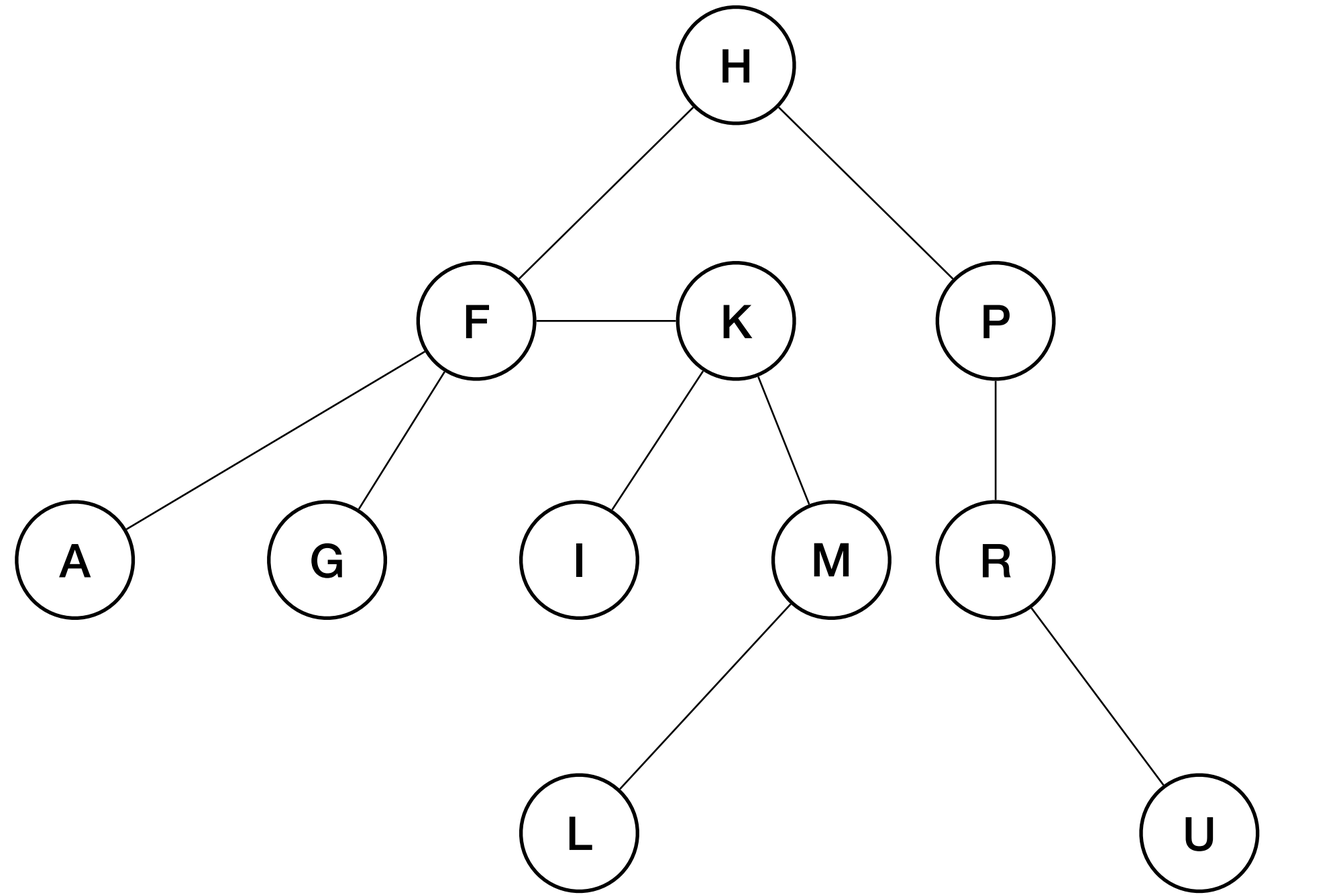
Recap

- Previously
  - Tree & tree concepts
  - Trees
    - Nodes - root/leaf
    - Relations - child/parent/ sibling
  - Properties - depth/height
  - Traversals - three kinds
  - Using stacks/queues
  - Recursion

Recap

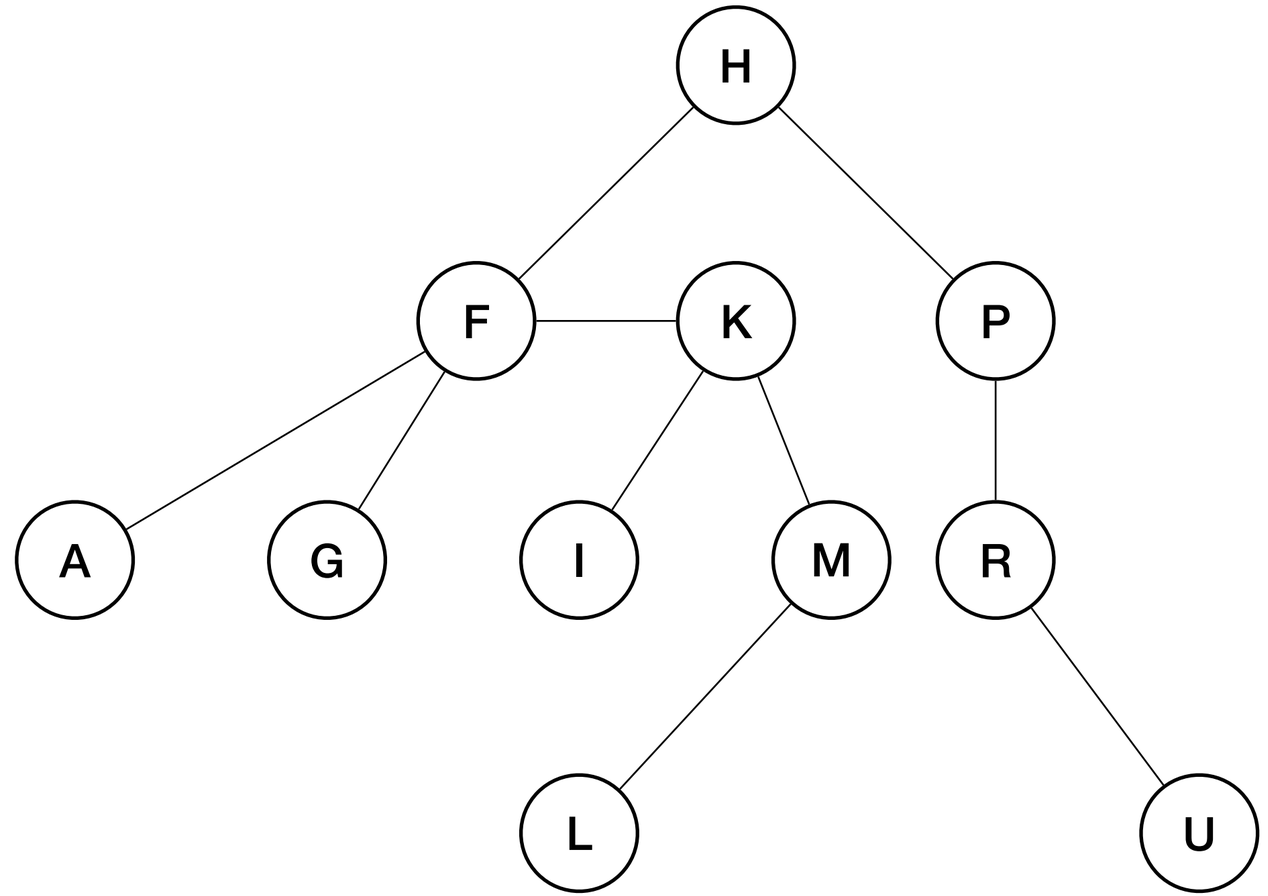
- Scan QR Code

Recap



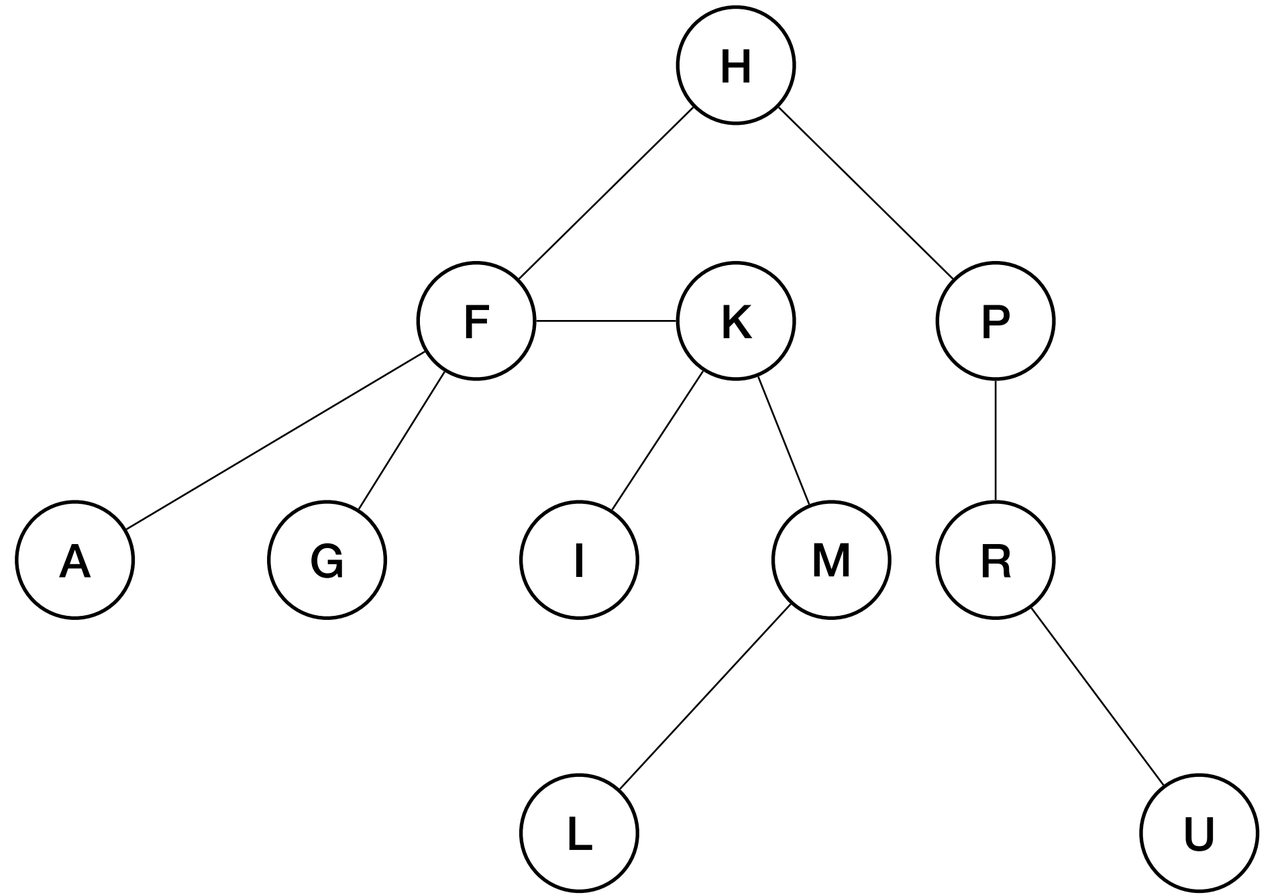
Recap

- Who are R's siblings?



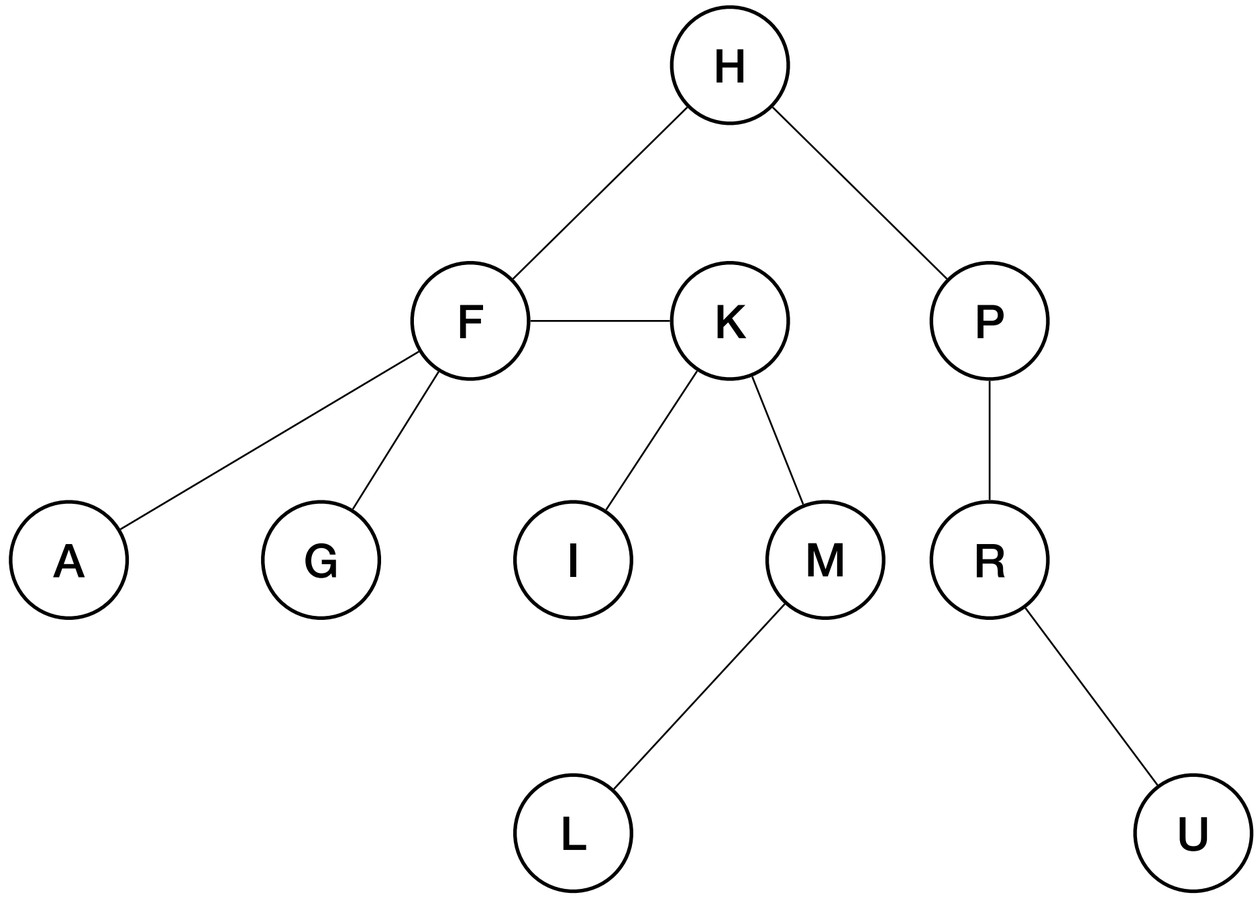
Recap

- Who are R's siblings?
- What is the depth of node I?



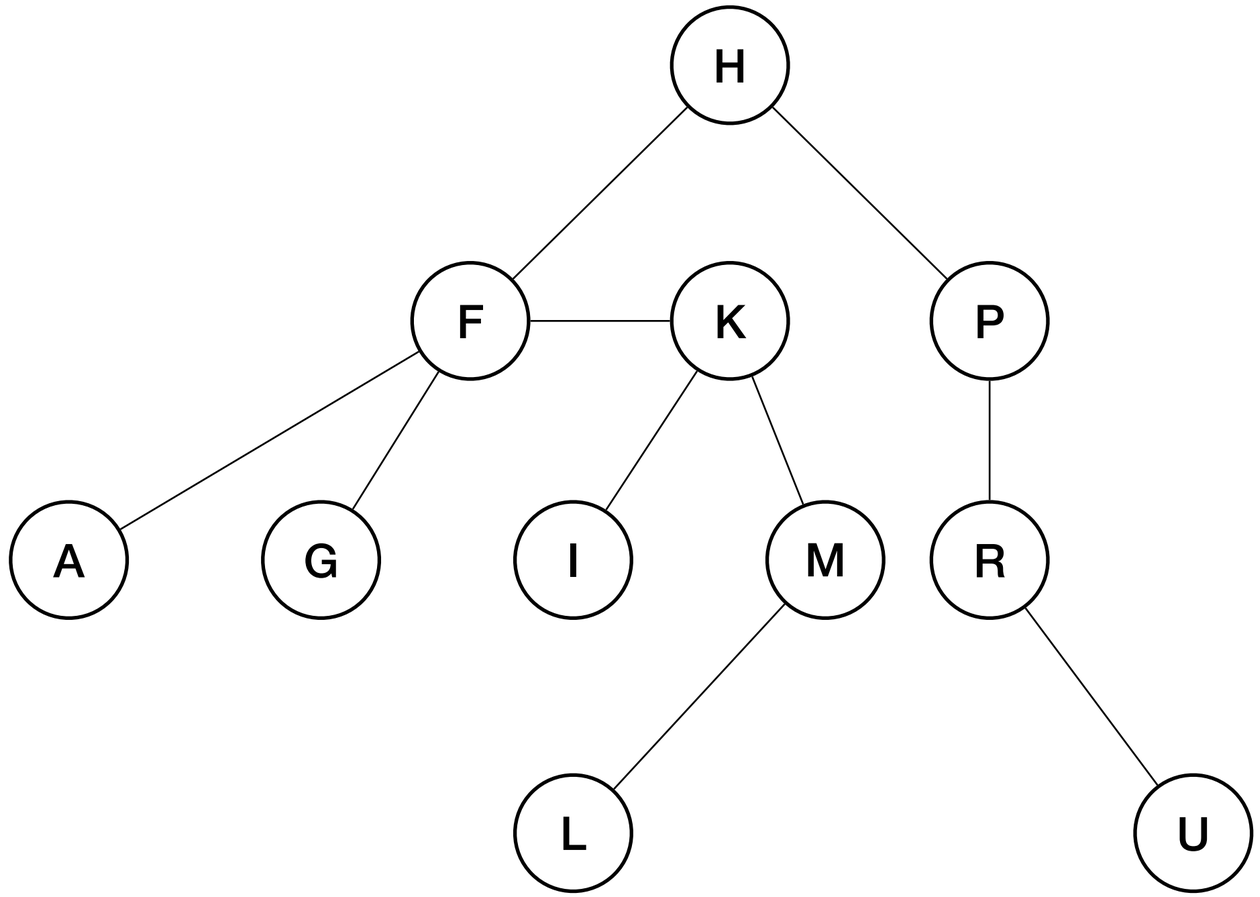
Recap

- Who are R's siblings?
- What is the depth of node I?
- Who are the leaf nodes?



Recap

- Who are R's siblings?
- What is the depth of node I?
- Who are the leaf nodes?
- What is the height of the tree?



Binary Search Trees

Binary Search Trees

- Binary trees that have a particular *sorted* property are called binary search trees (BST)

Binary Search Trees

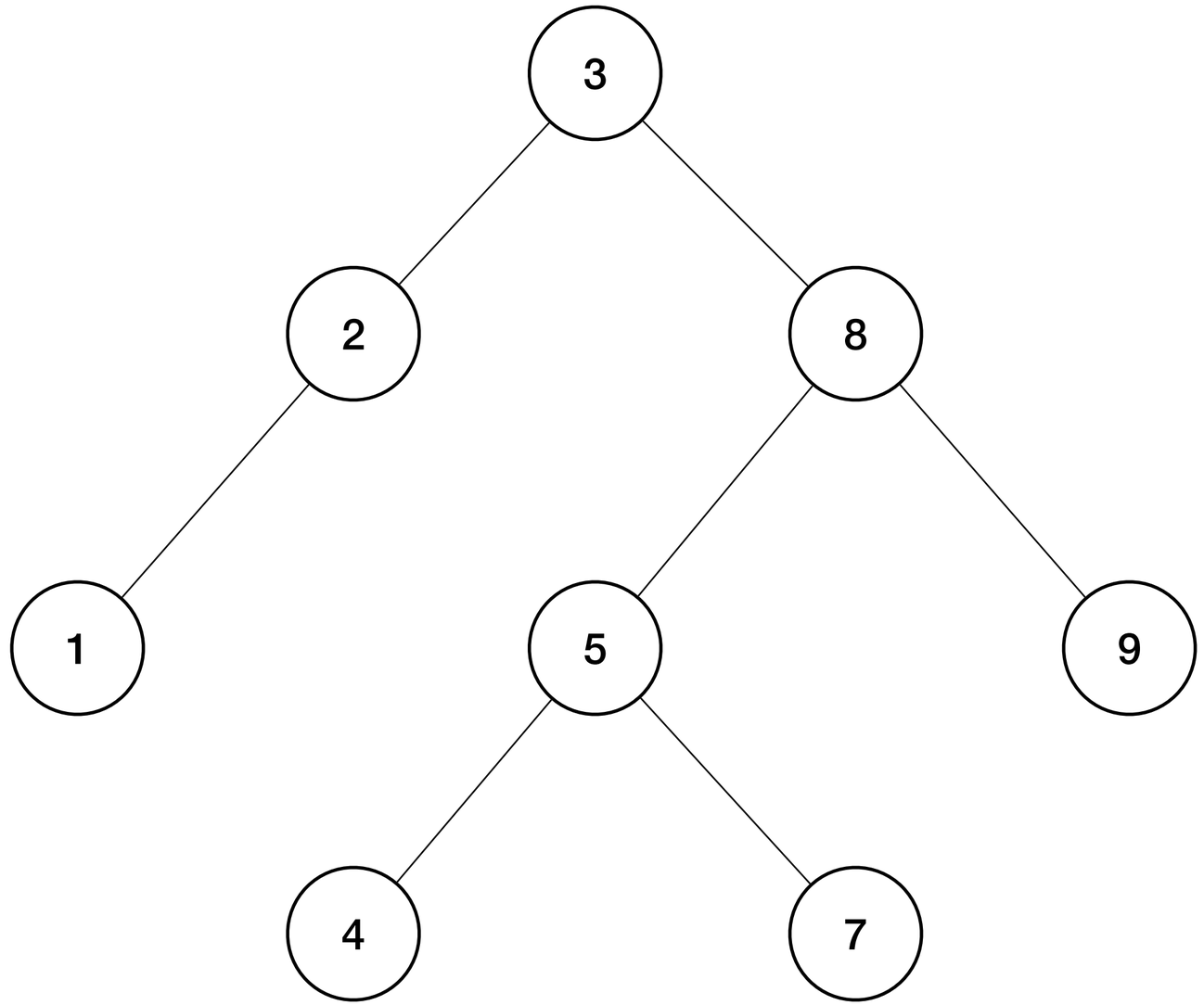
- Binary trees that have a particular *sorted* property are called binary search trees (BST)
  - All nodes in the **left subtree** of a given node are *lesser than or equal* to the node

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

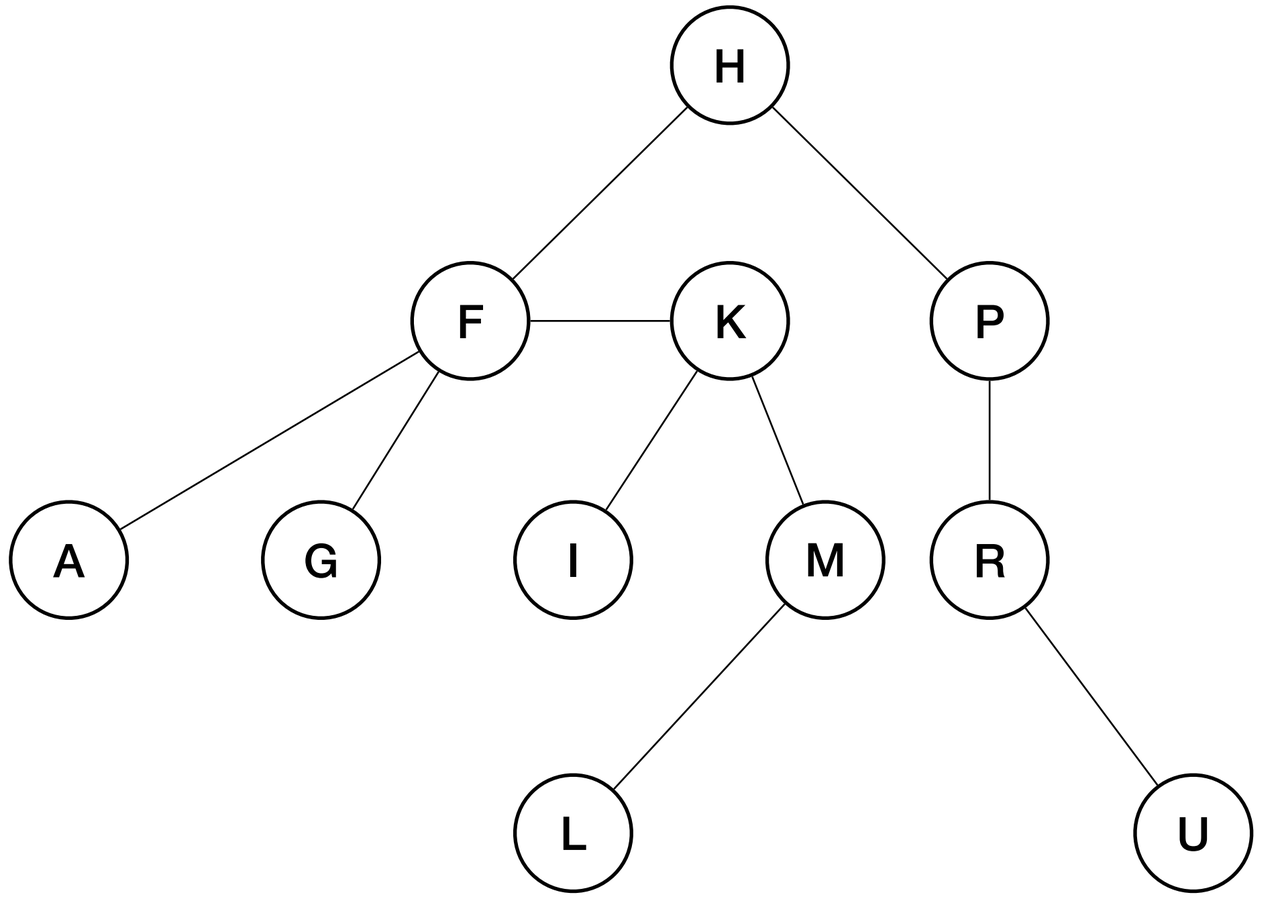
Binary Search Trees

- 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



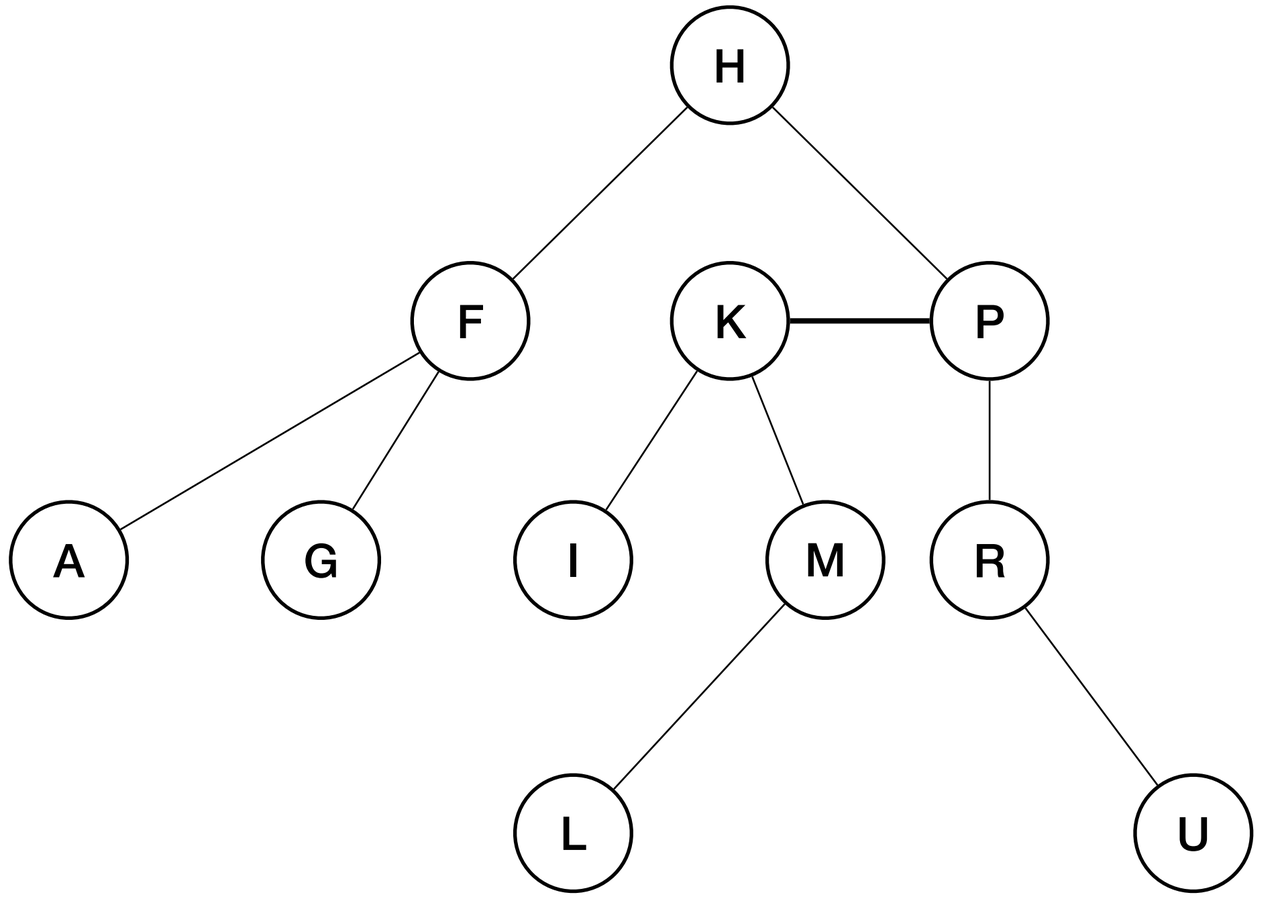
Recap

- Who are the siblings of L?
- What is the depth of node I?
- List the leaf nodes?
- What is the height of the tree?
- Is this a Binary Search Tree?



Recap

- Who are the siblings of L?
- What is the depth of node I?
- List the leaf nodes?
- What is the height of the tree?
- Is this a Binary Search Tree?



Exercises with BST

Exercises with BST

- How can you find the minimum or maximum element in a BST?

Exercises with BST

- How can you find the minimum or maximum element in a BST?
- How can we search a BST for a 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?

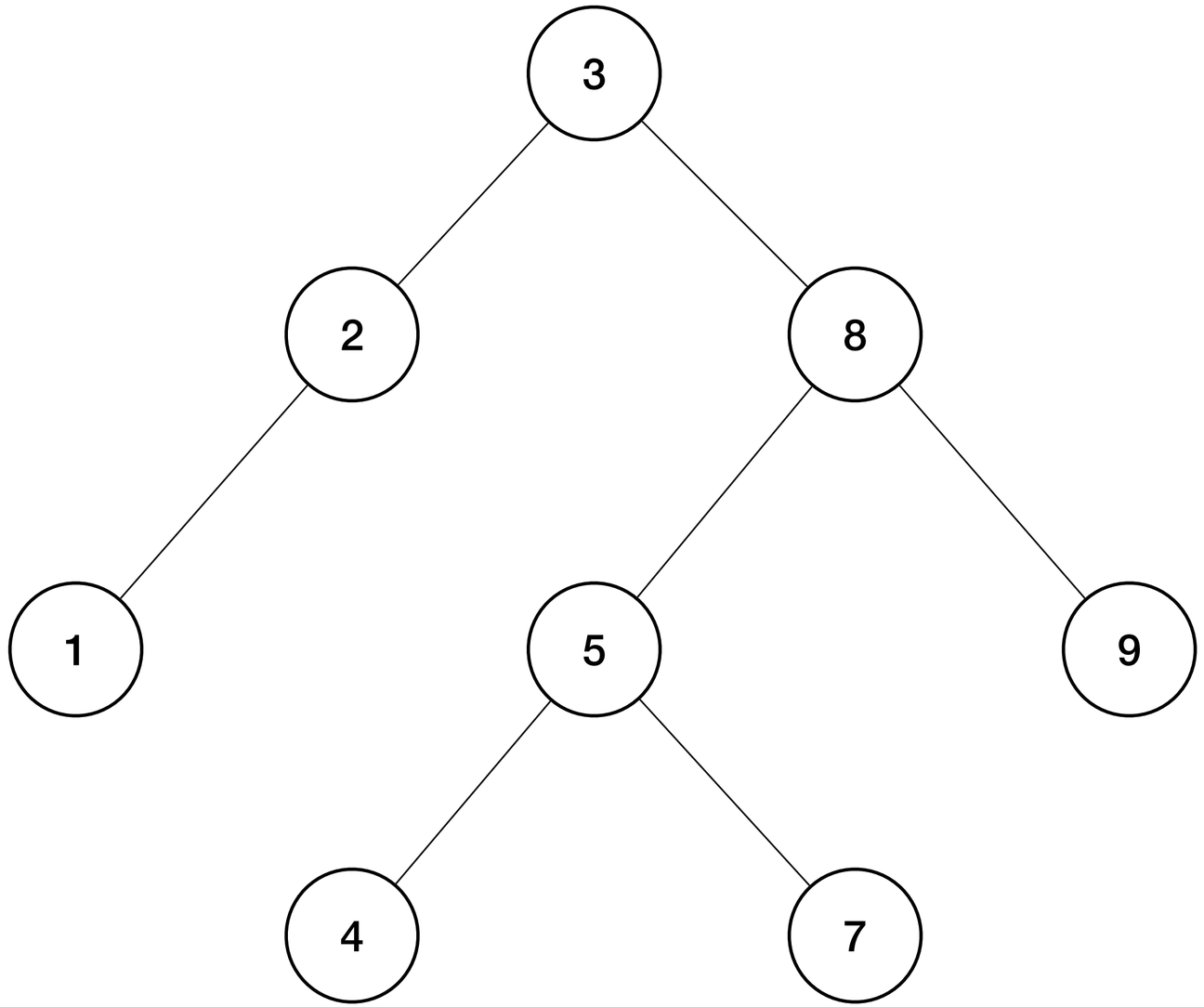
Exercises with BST

- 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)?

Finding extremals in a BST

Minimum - keep going left

Minimum - keep going right

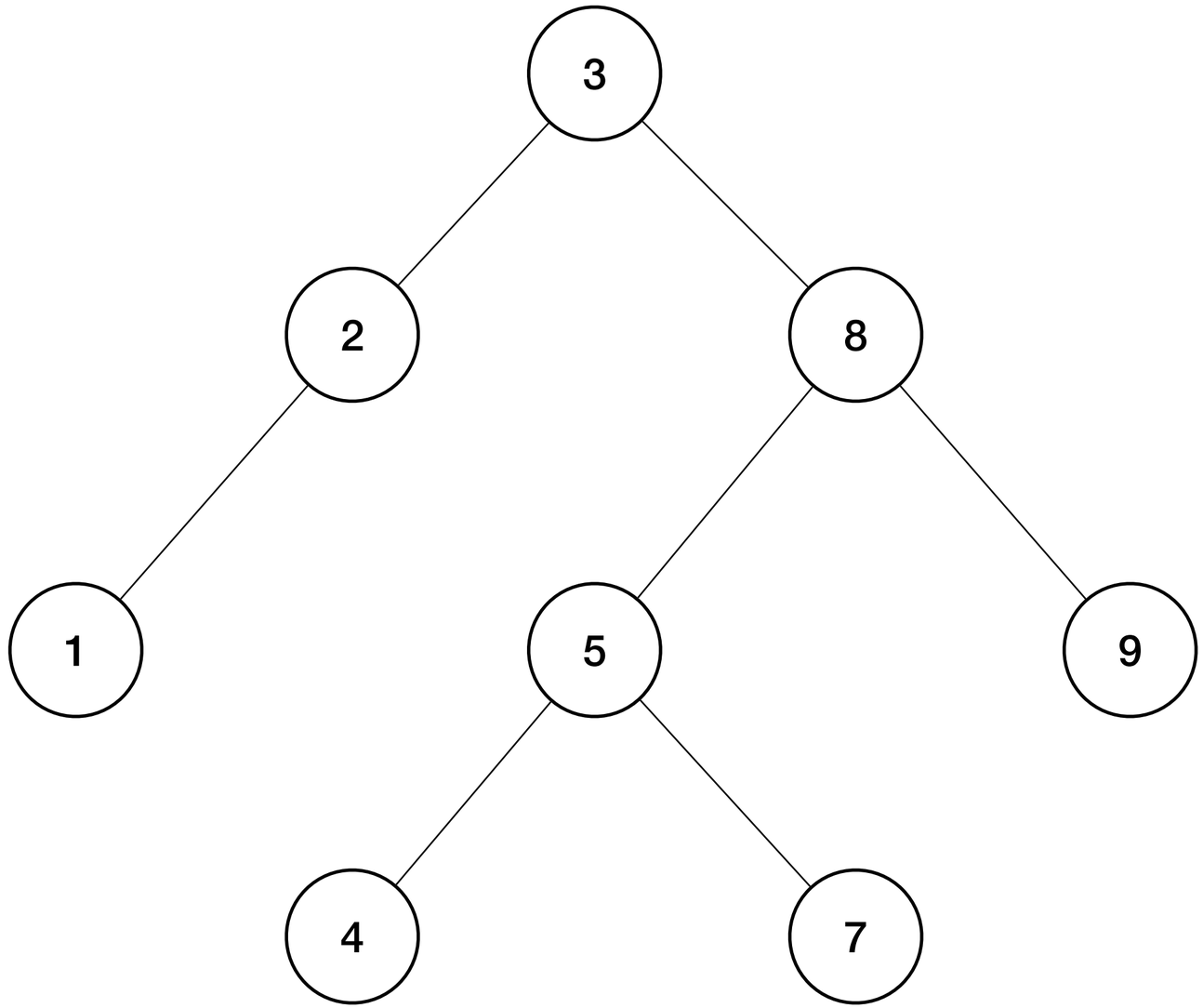


Finding extremals in a BST

Minimum - keep going left

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

Minimum - keep going right

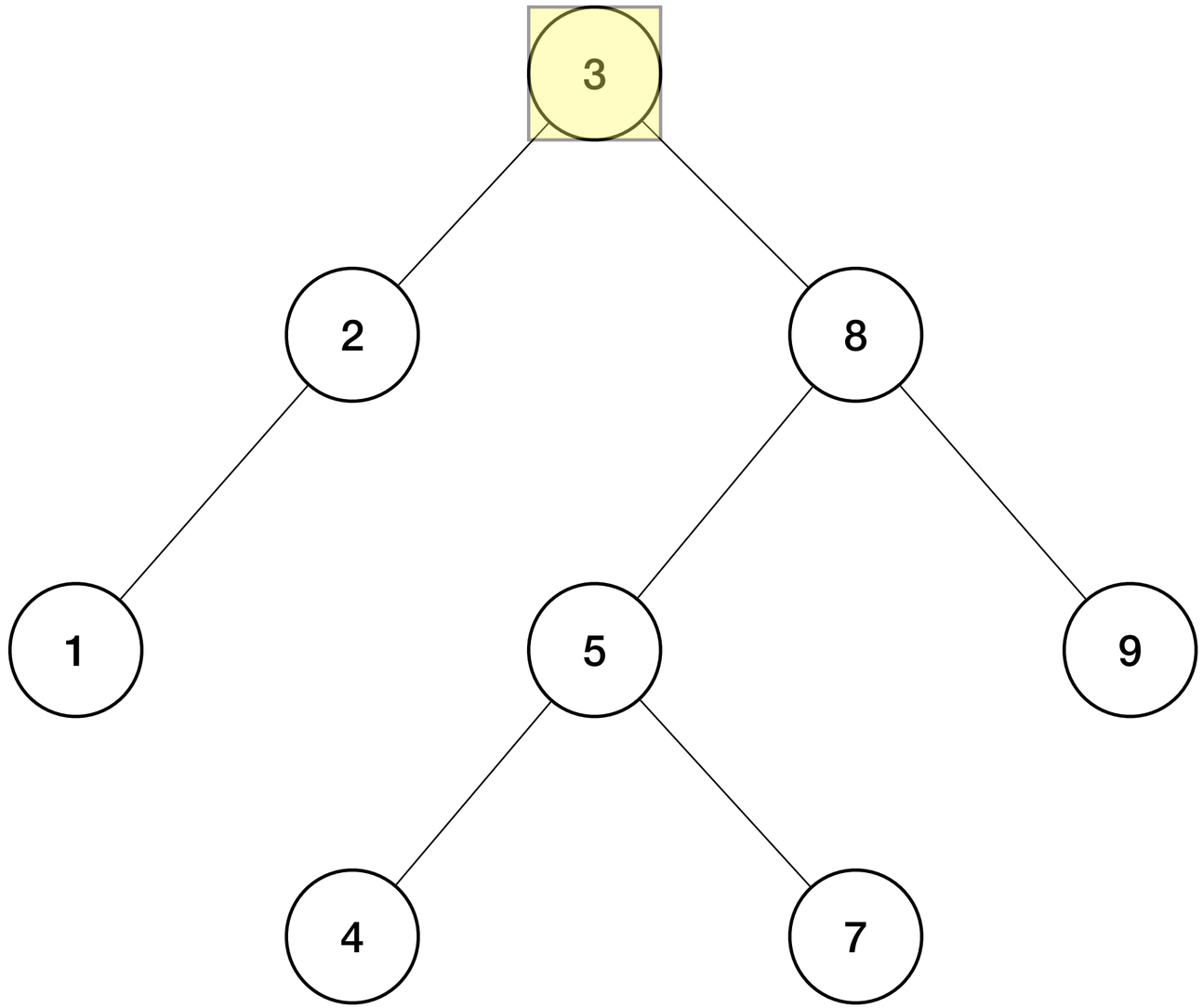


Finding extremals in a BST

Minimum - keep going left

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

Minimum - keep going right

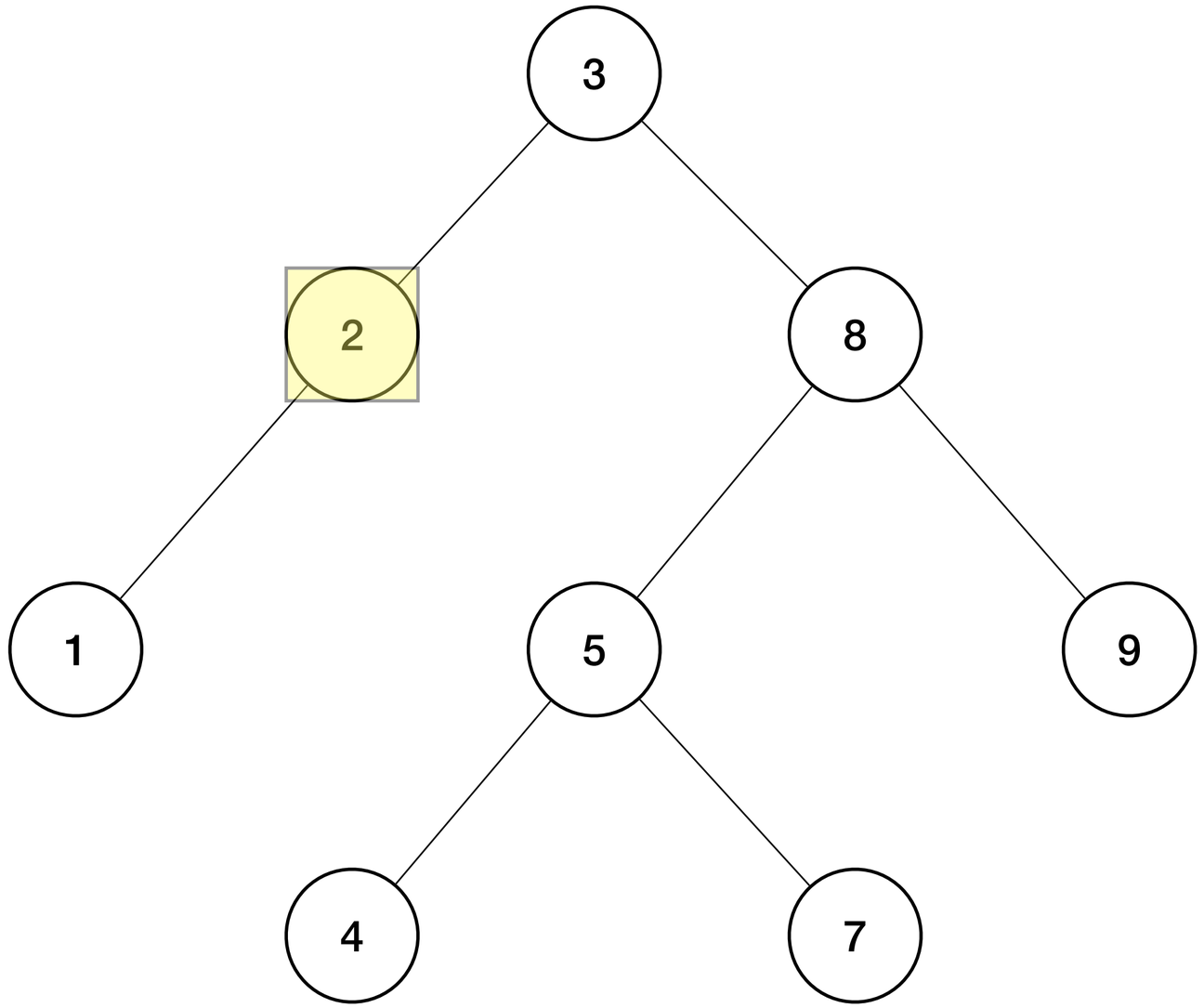


Finding extremals in a BST

Minimum - keep going left

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

Minimum - keep going right

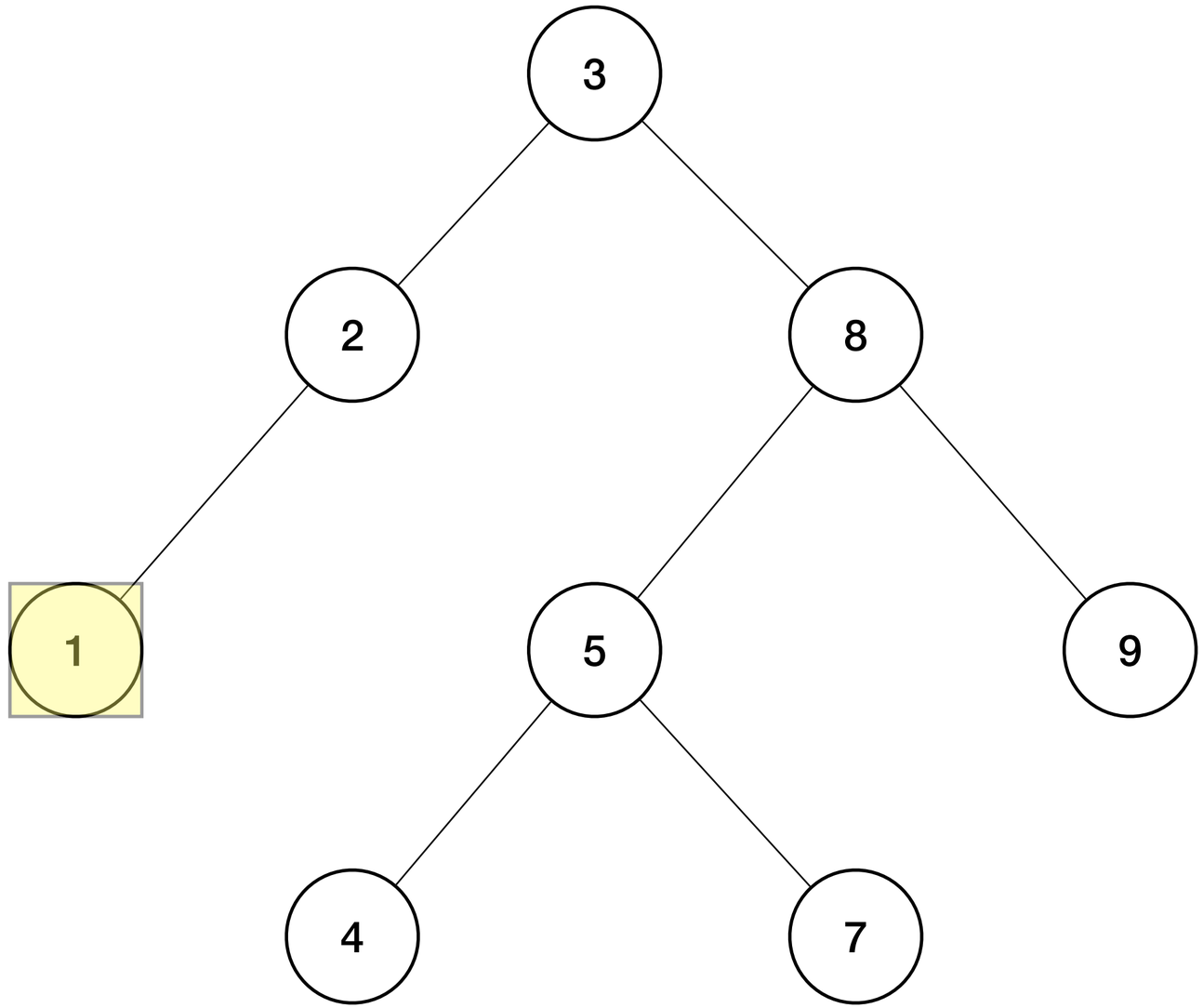


Finding extremals in a BST

Minimum - keep going left

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

Minimum - keep going right

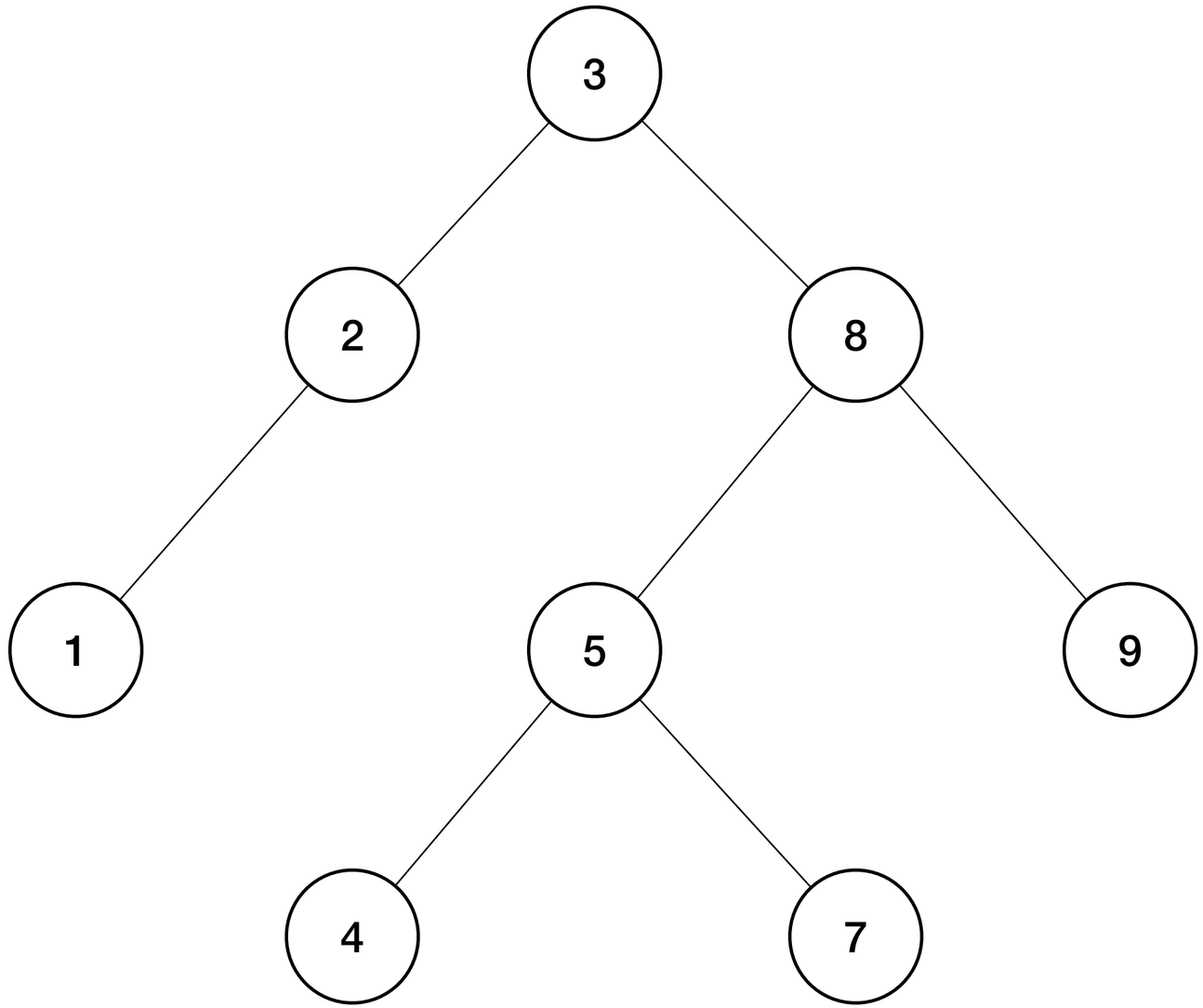


Finding extremals in a BST

Minimum - keep going left

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

Minimum - keep going right



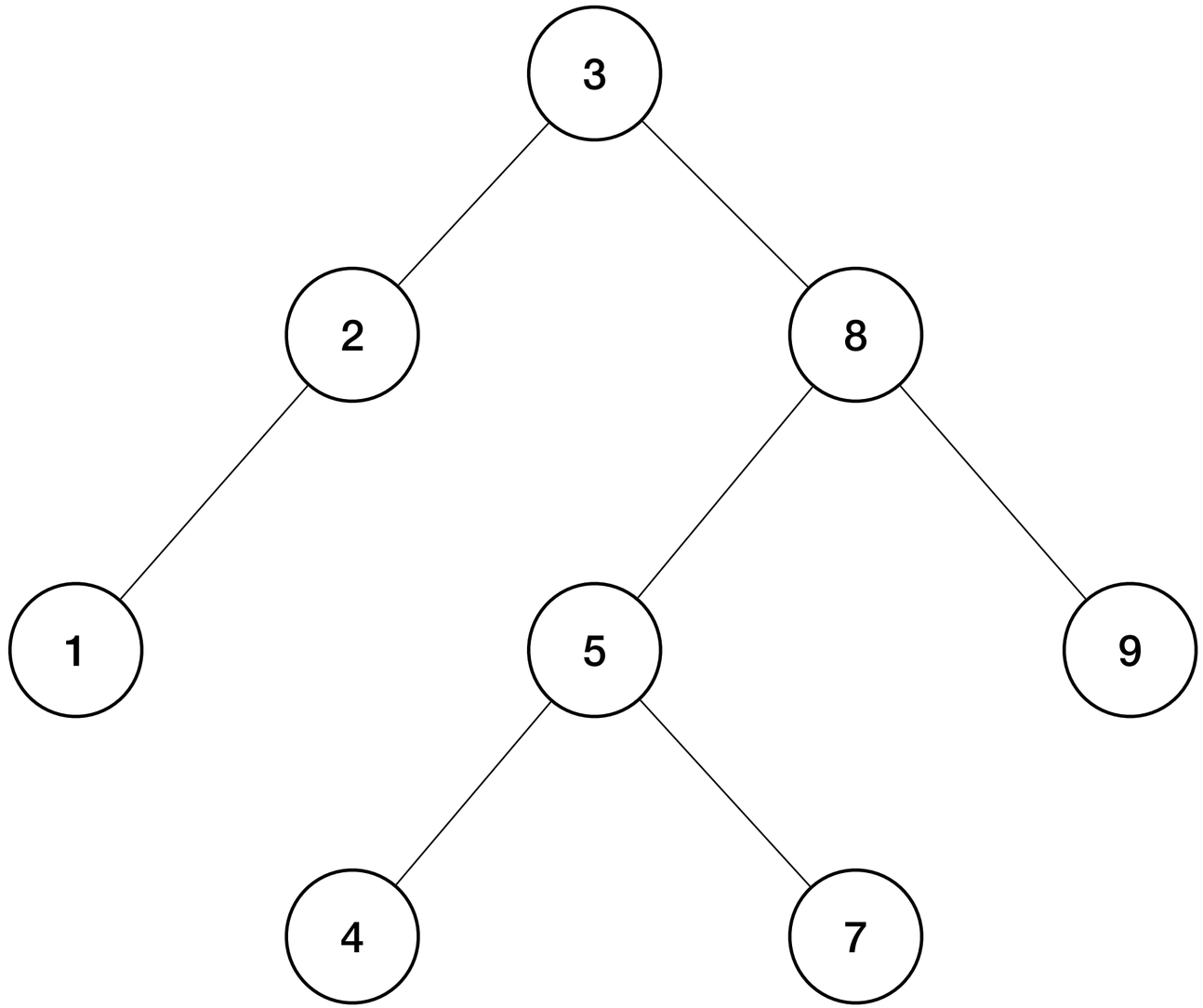
Finding extremals in a BST

Minimum - keep going left

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

Minimum - keep going right

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



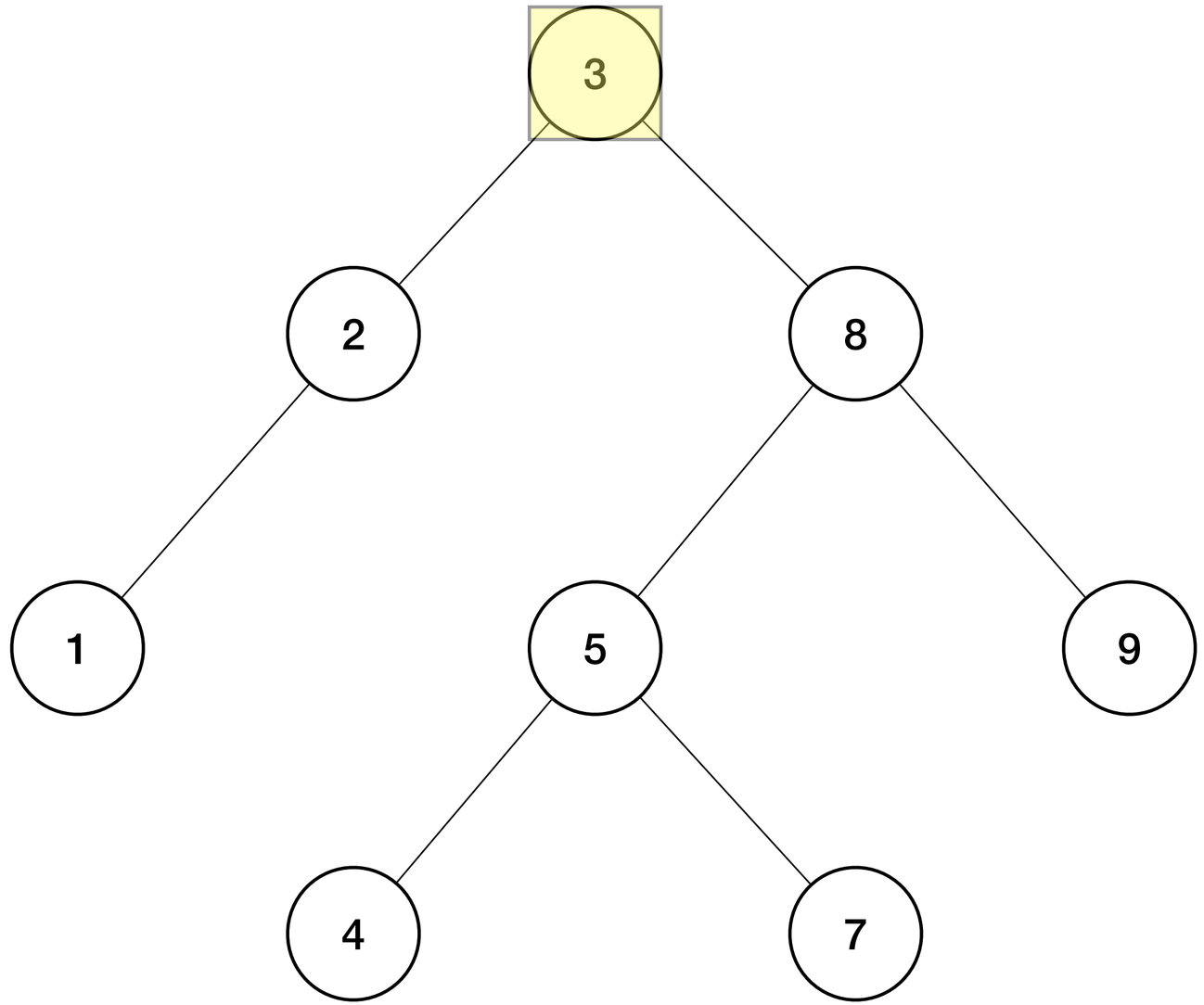
Finding extremals in a BST

Minimum - keep going left

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

Minimum - keep going right

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



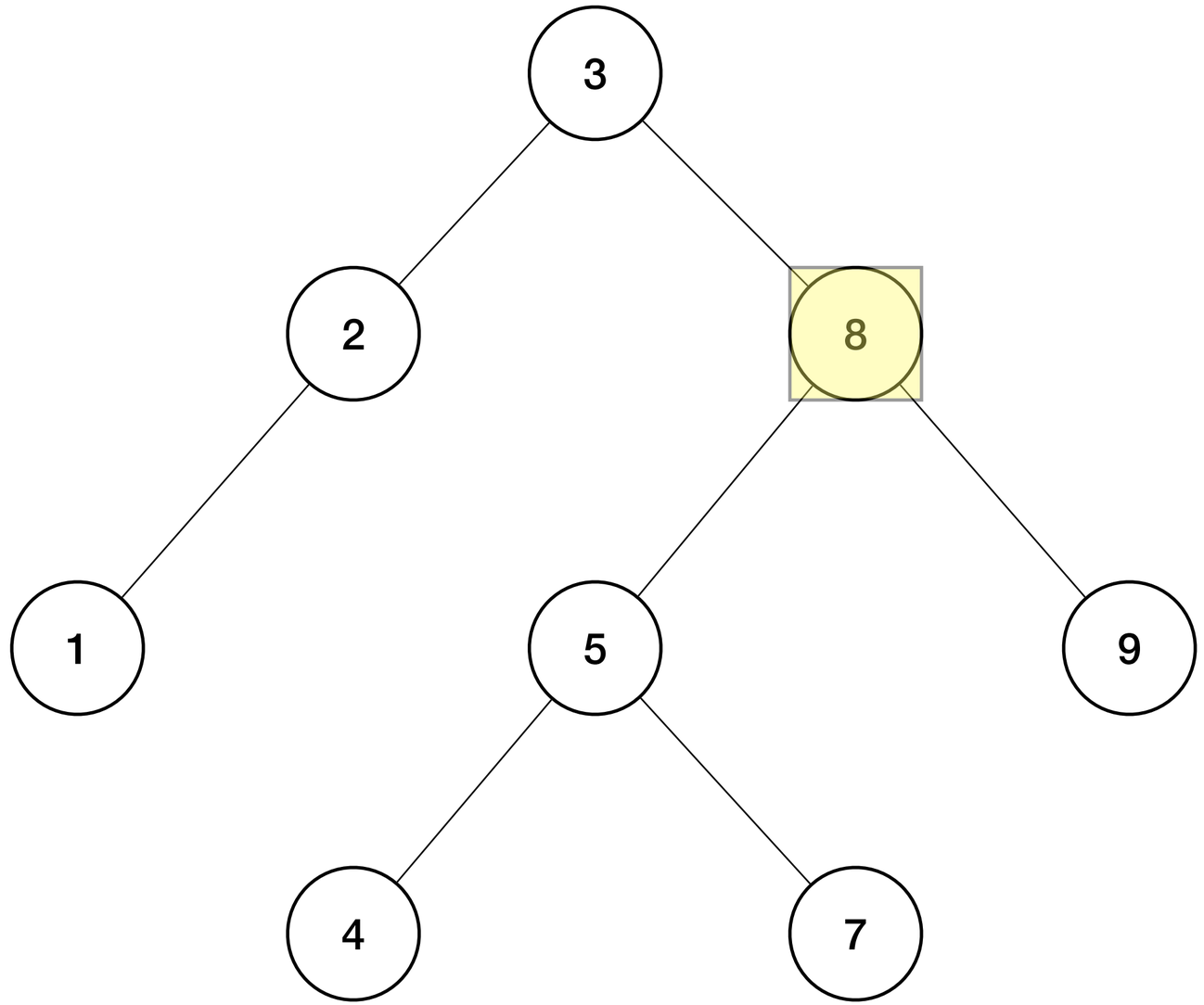
Finding extremals in a BST

Minimum - keep going left

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

Minimum - keep going right

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



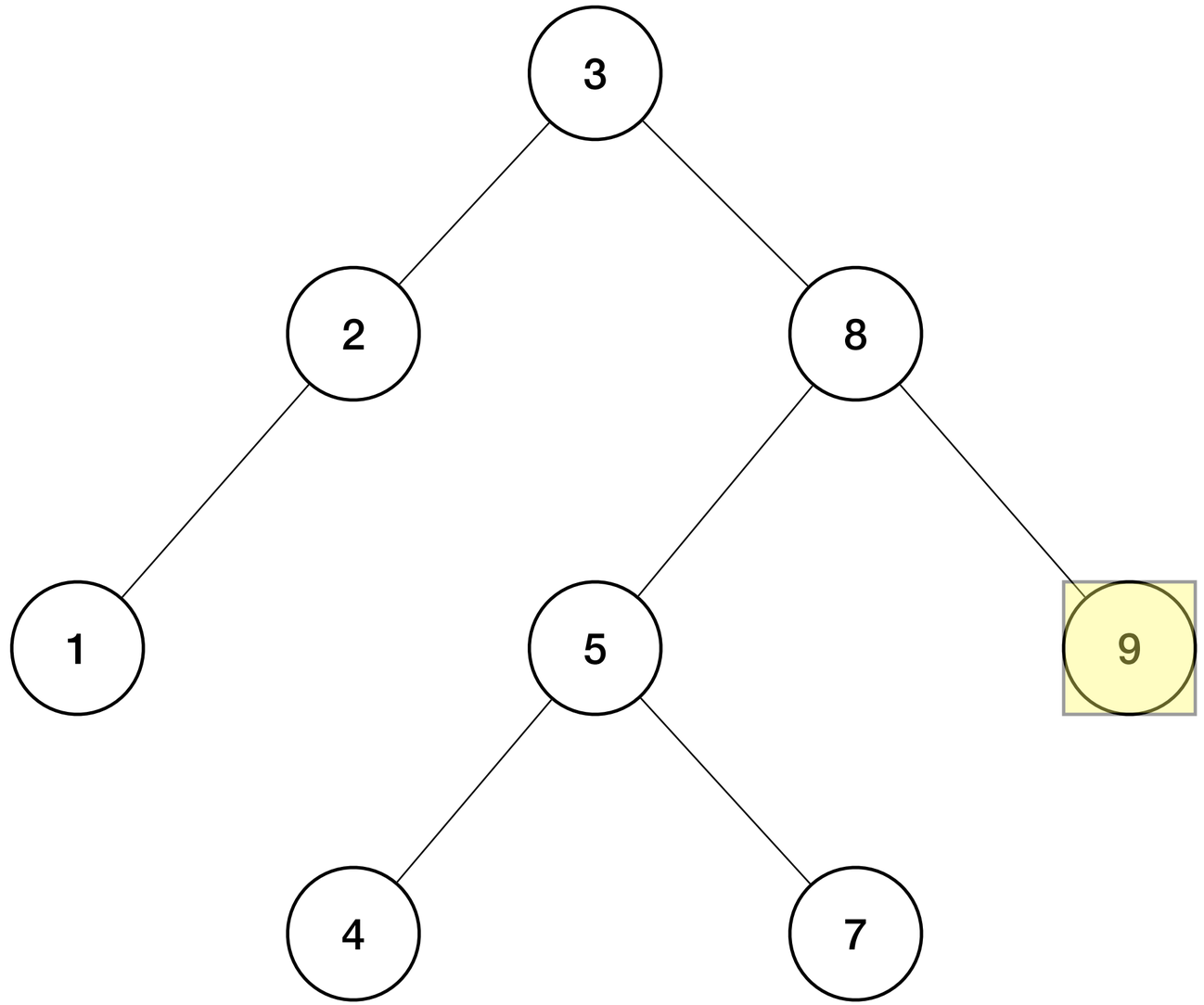
Finding extremals in a BST

Minimum - keep going left

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

Minimum - keep going right

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



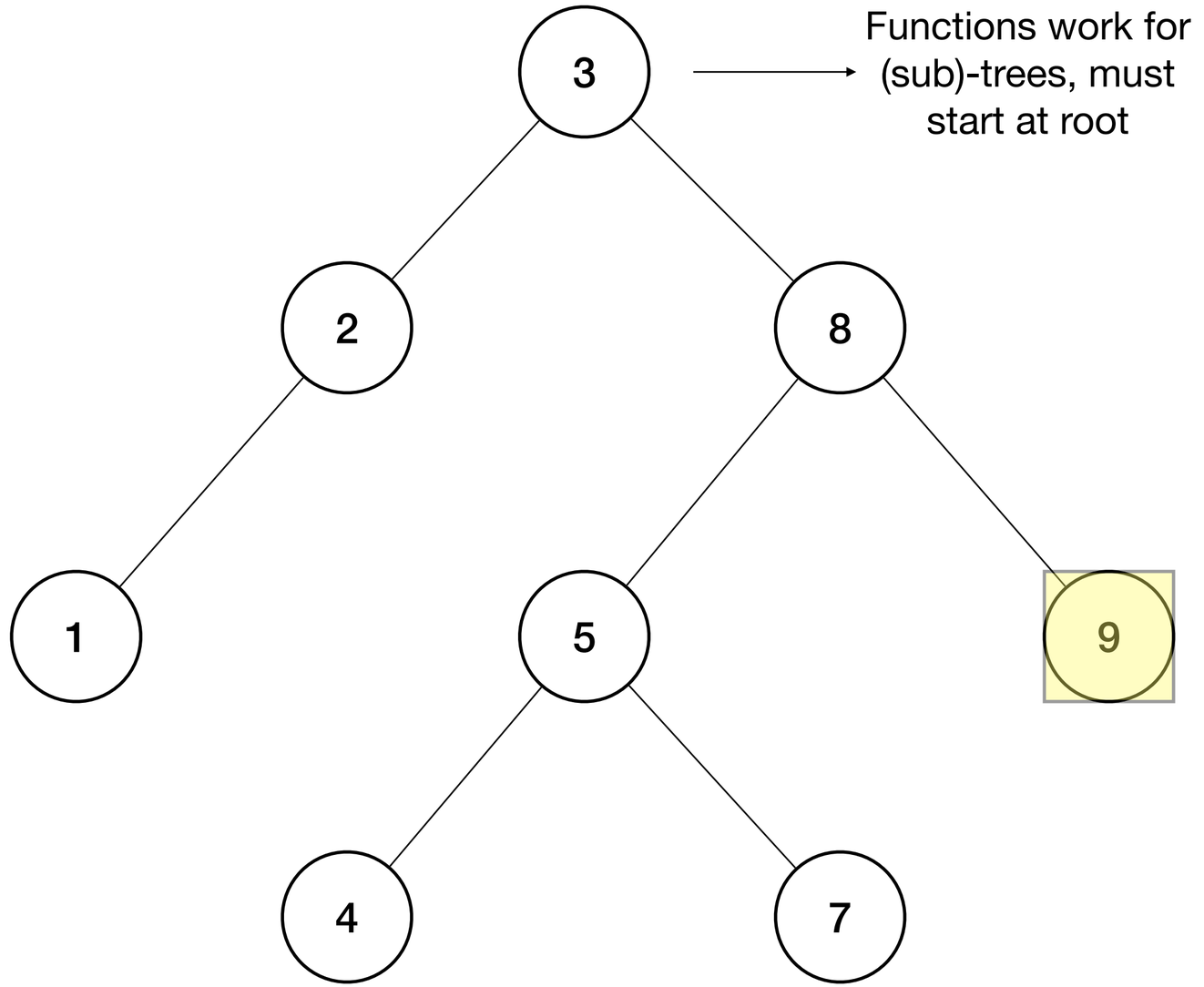
Finding extremals in a BST

Minimum - keep going left

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

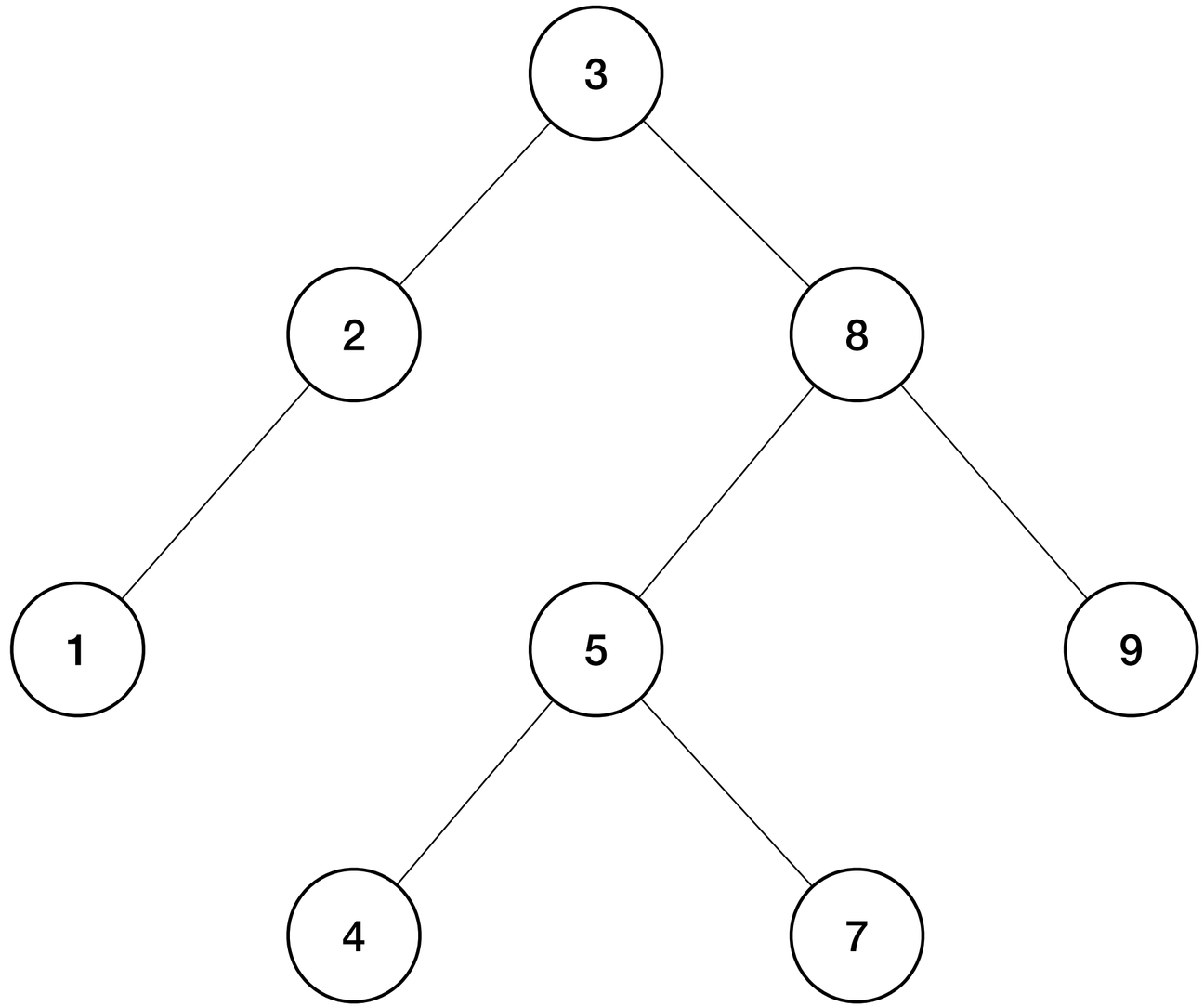
Minimum - keep going right

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



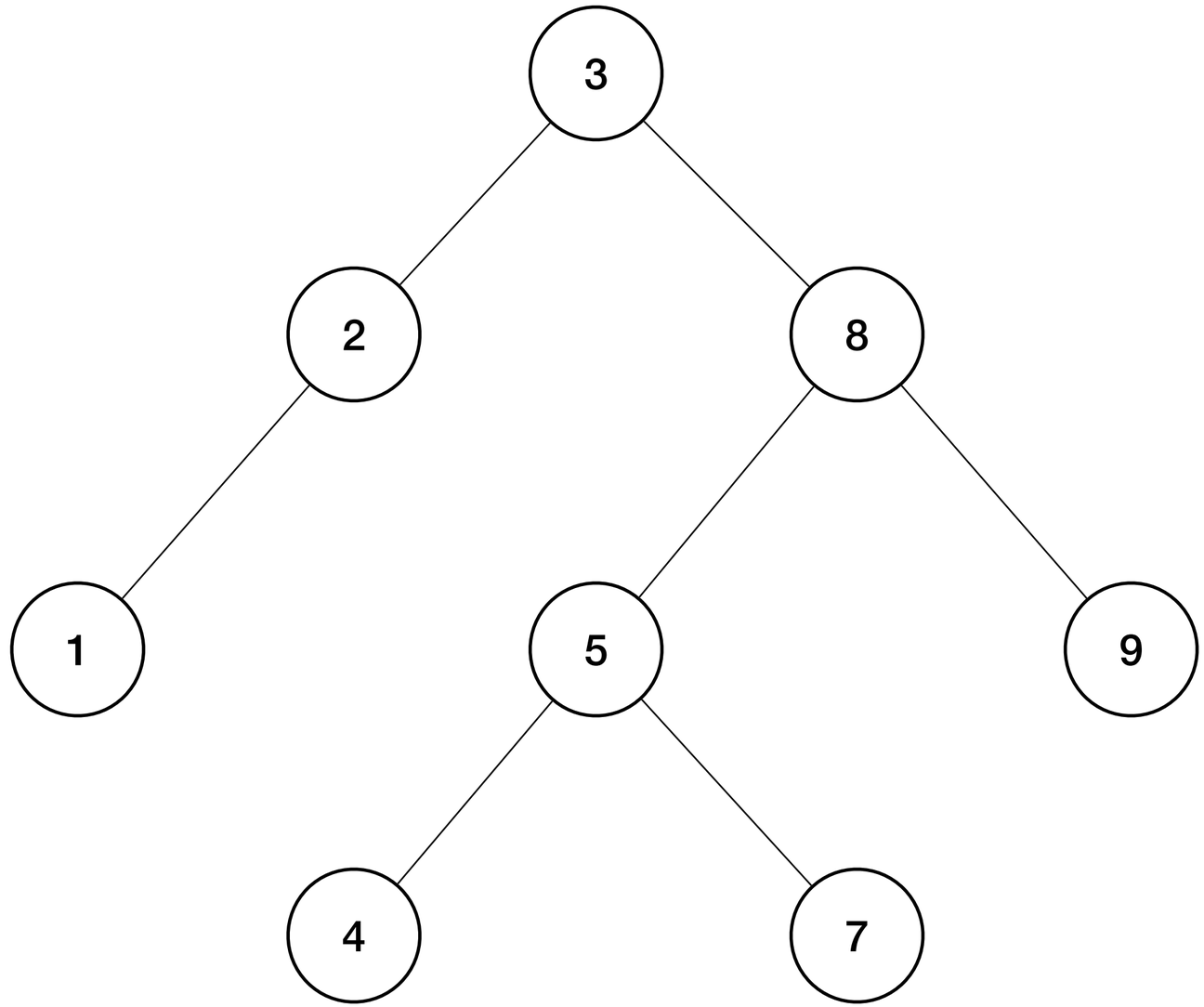
Searching in a BST

```
node * find_elem(node *cursor, int key){
  if (cursor==NULL) // Key not found

  if (cursor->data == key)
                    // Found key
  if (cursor->data < key)

  else

}
```



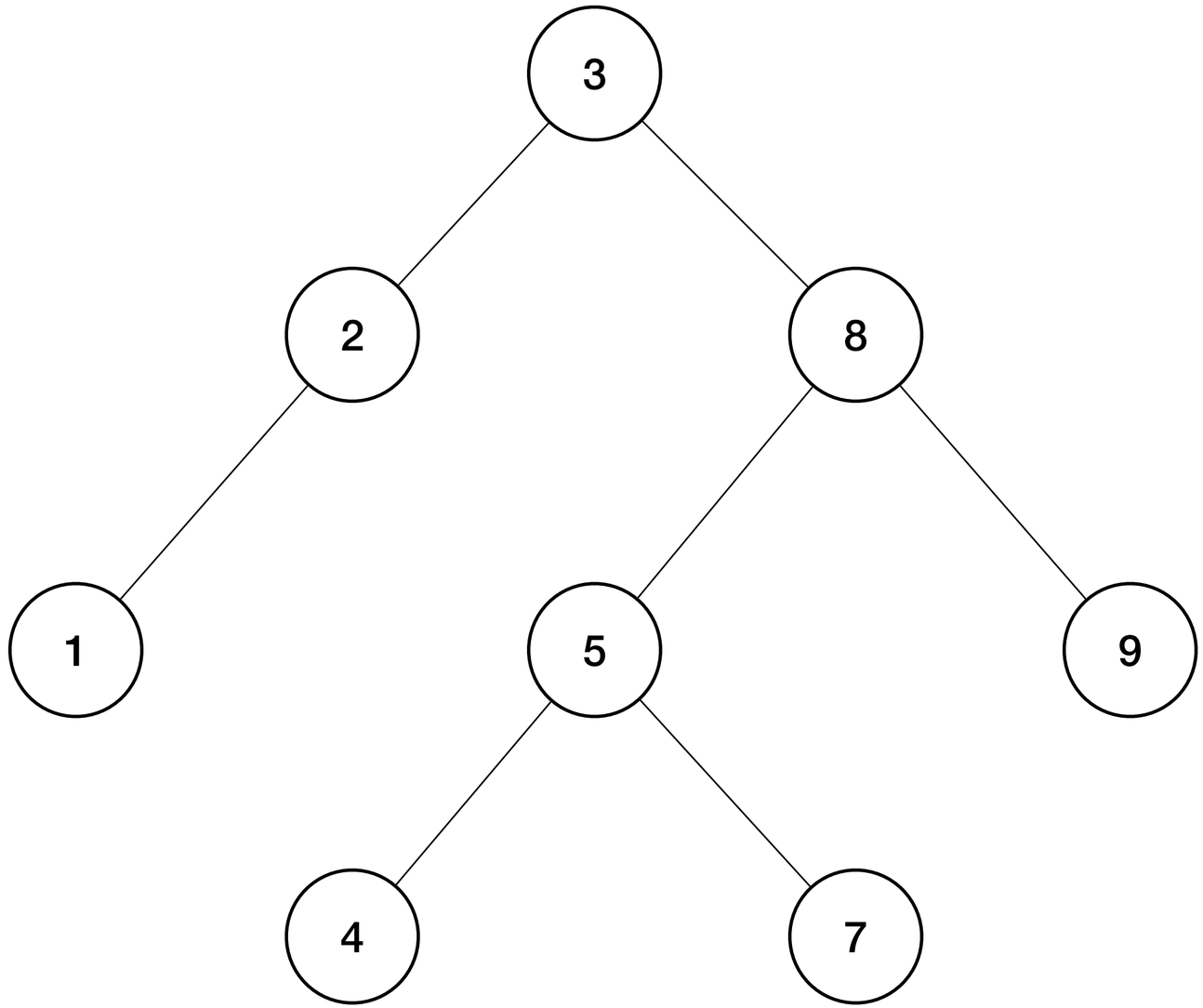
Searching in a BST

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

    else

}
```



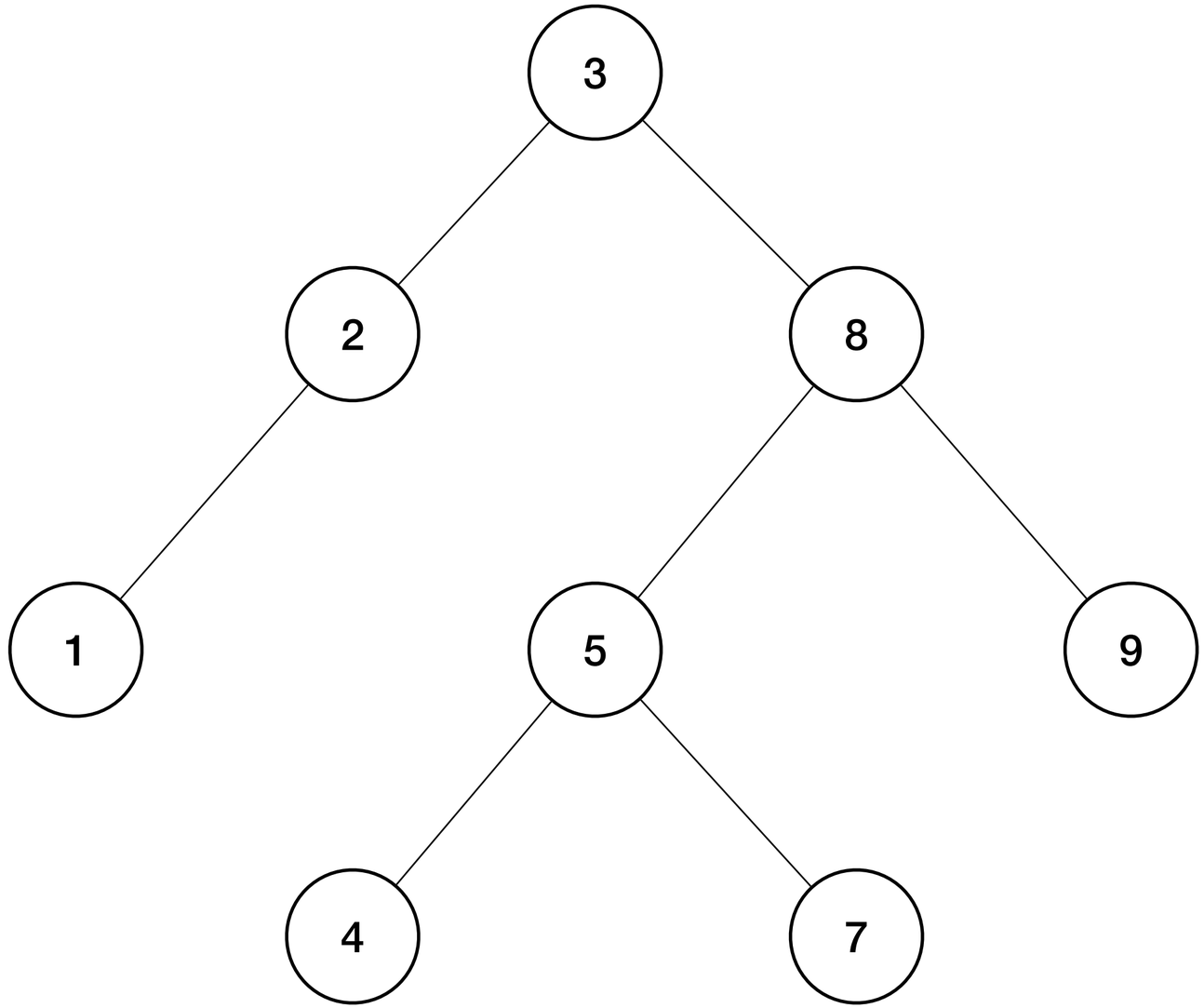
Searching in a BST

```
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)

  else

}
```



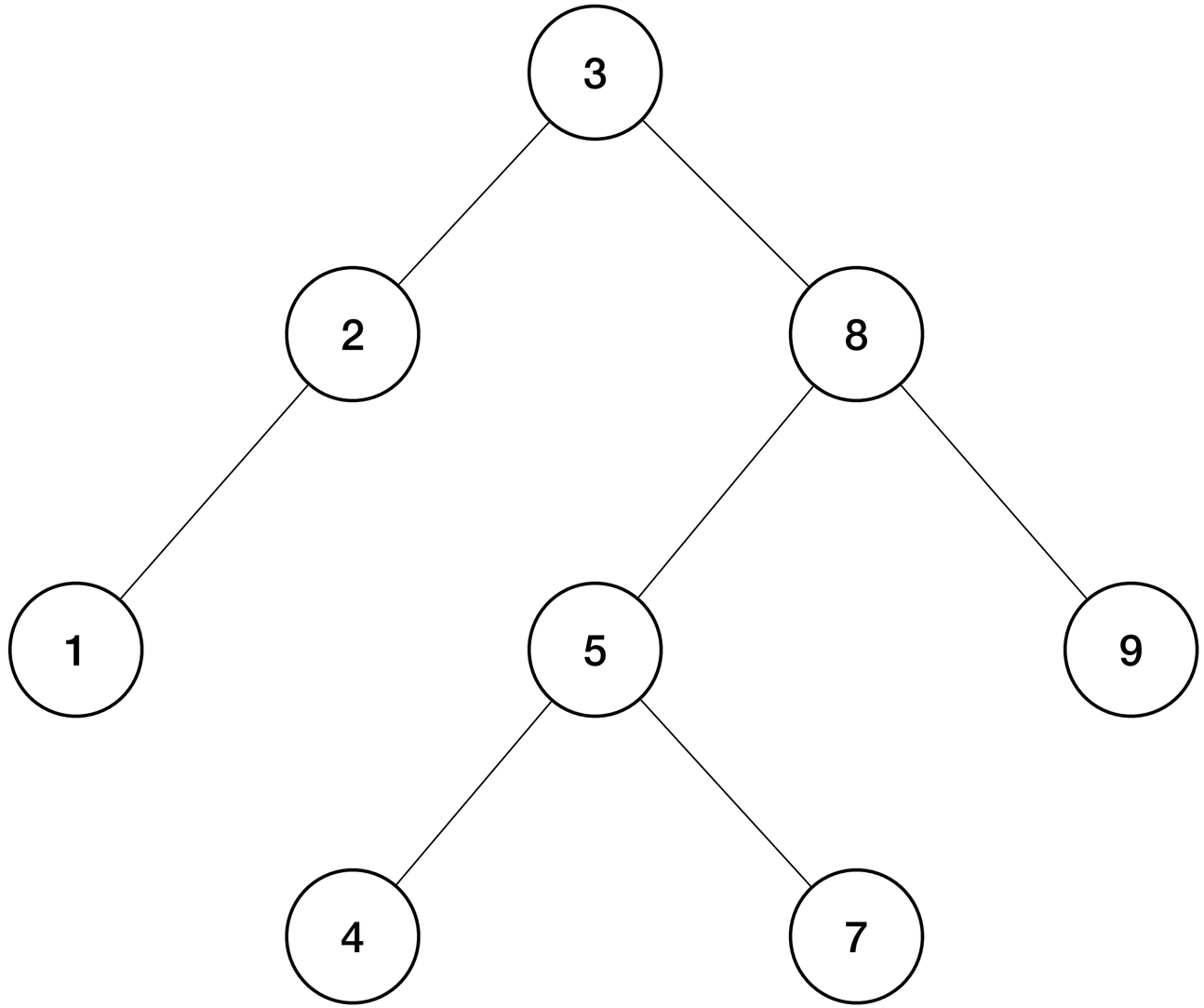
Searching in a BST

```
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


}
```



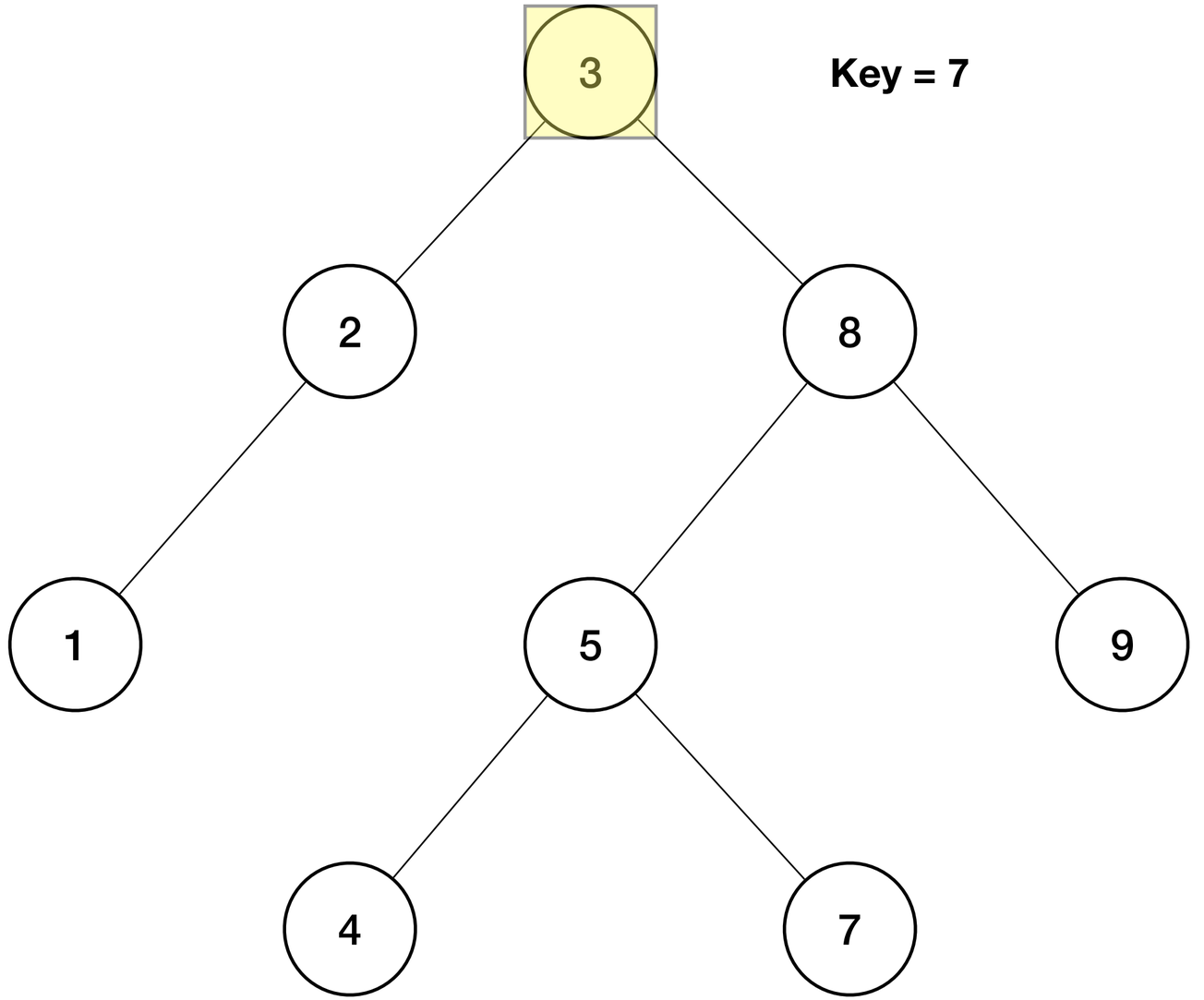
Searching in a BST

```
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);
}
```



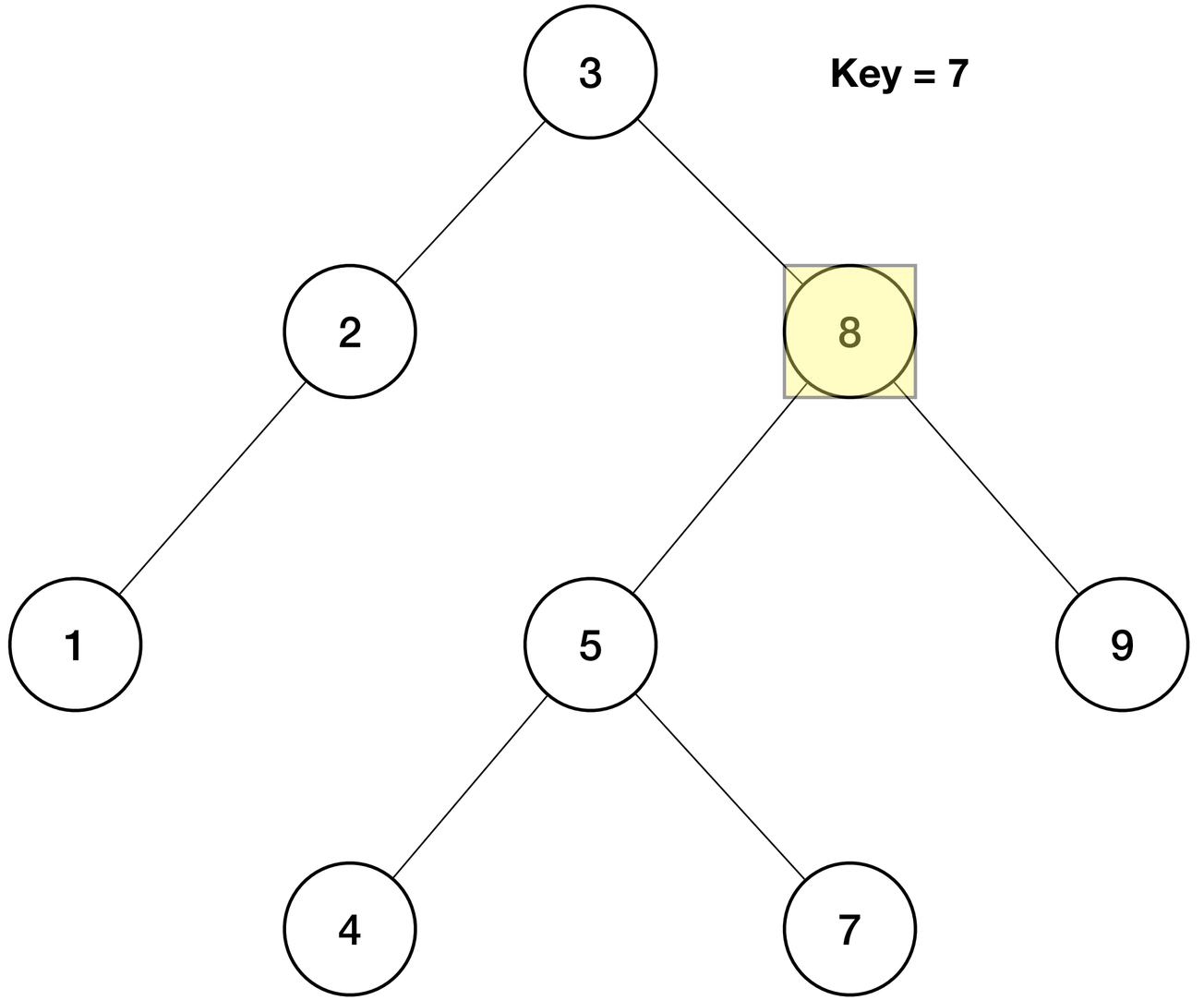
Searching in a BST

```
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);
}
```



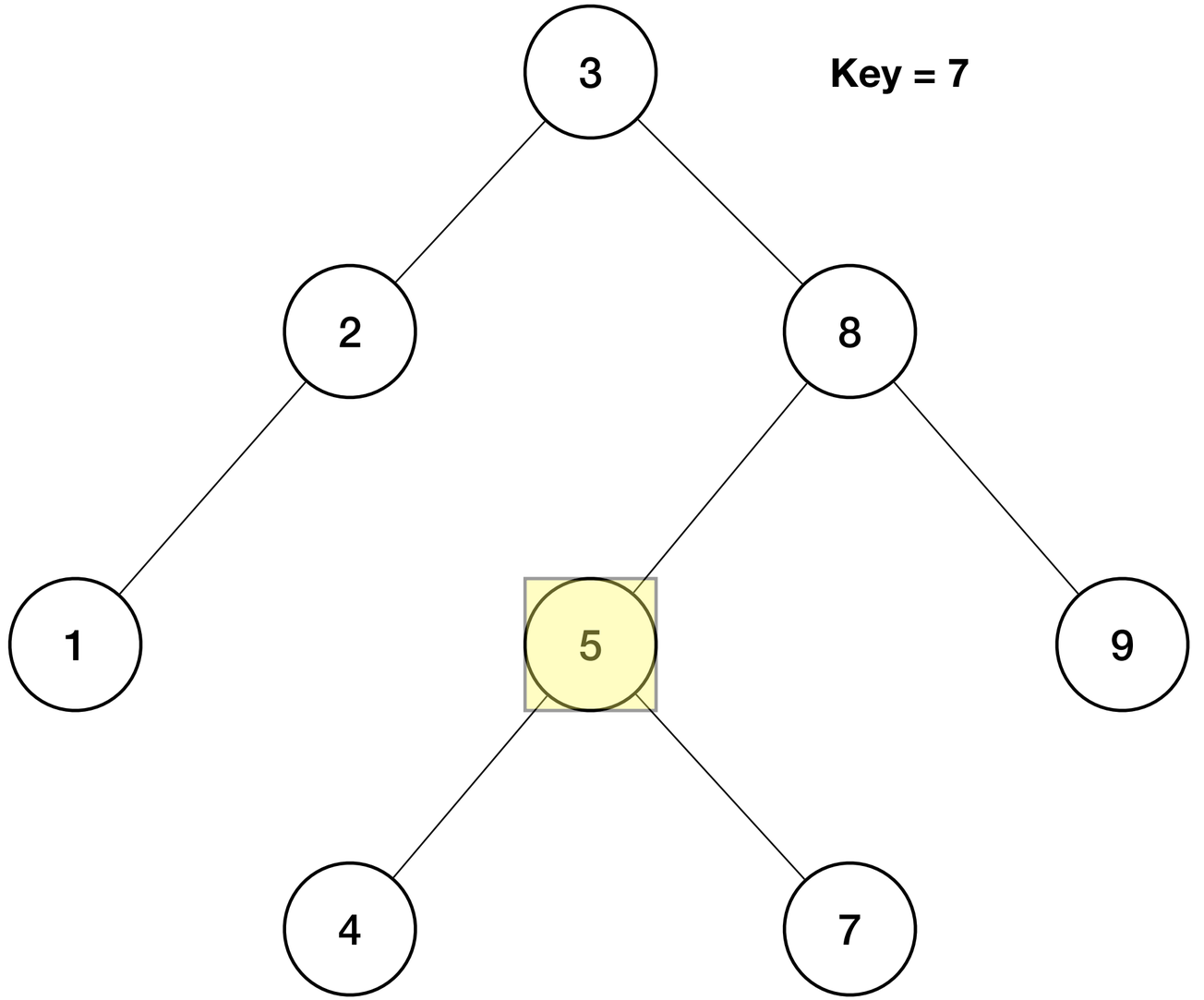
Searching in a BST

```
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);
}
```



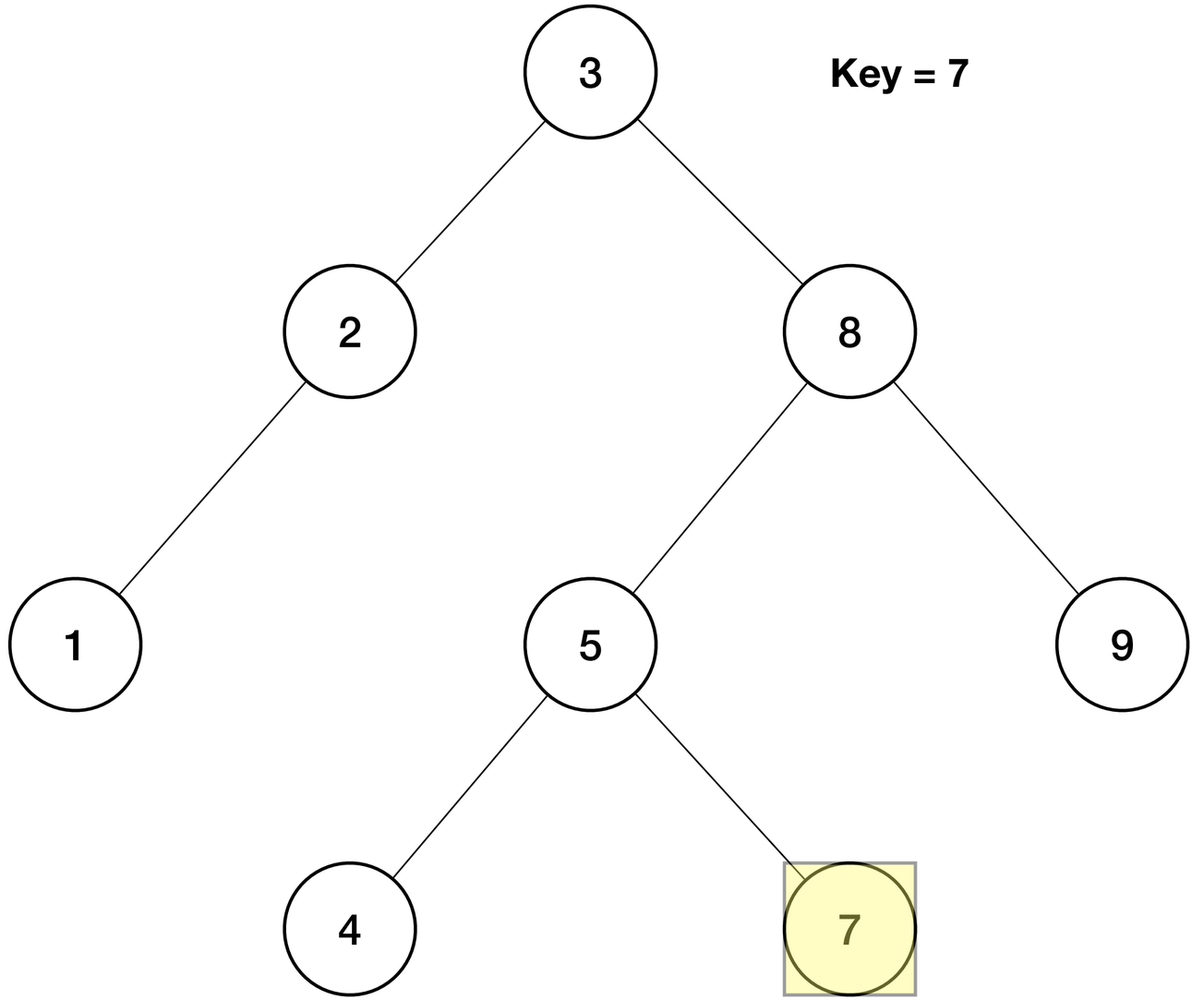
Searching in a BST

```
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);
}
```

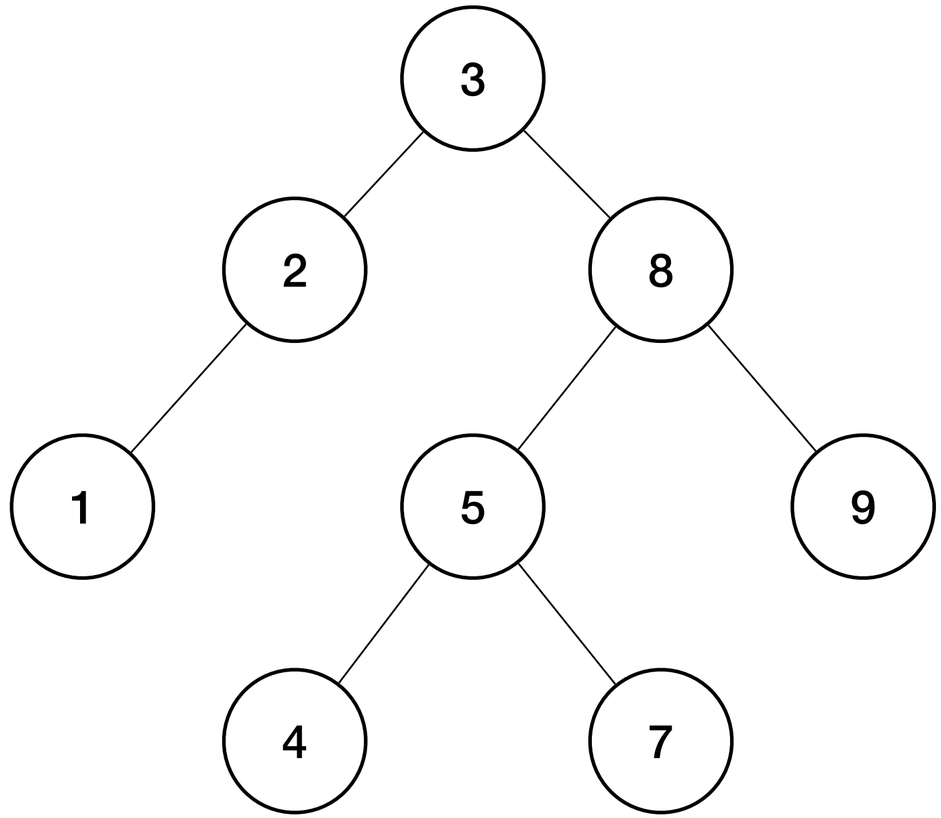


Searching in a BST

```
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);
}
```

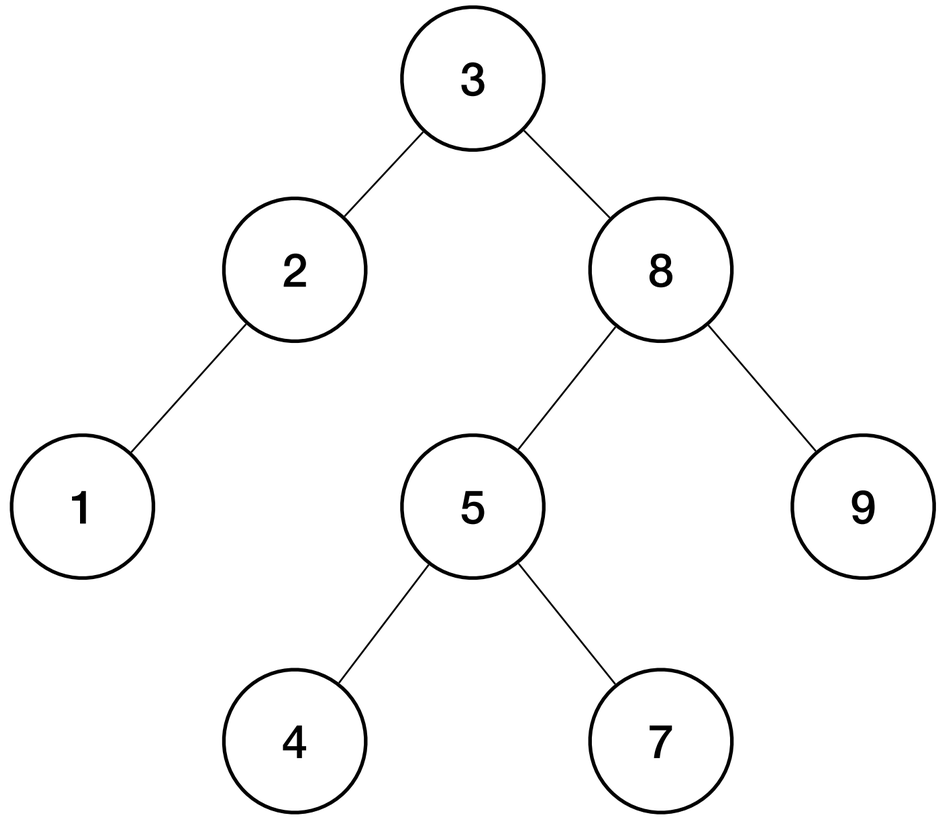


Insertion in a BST



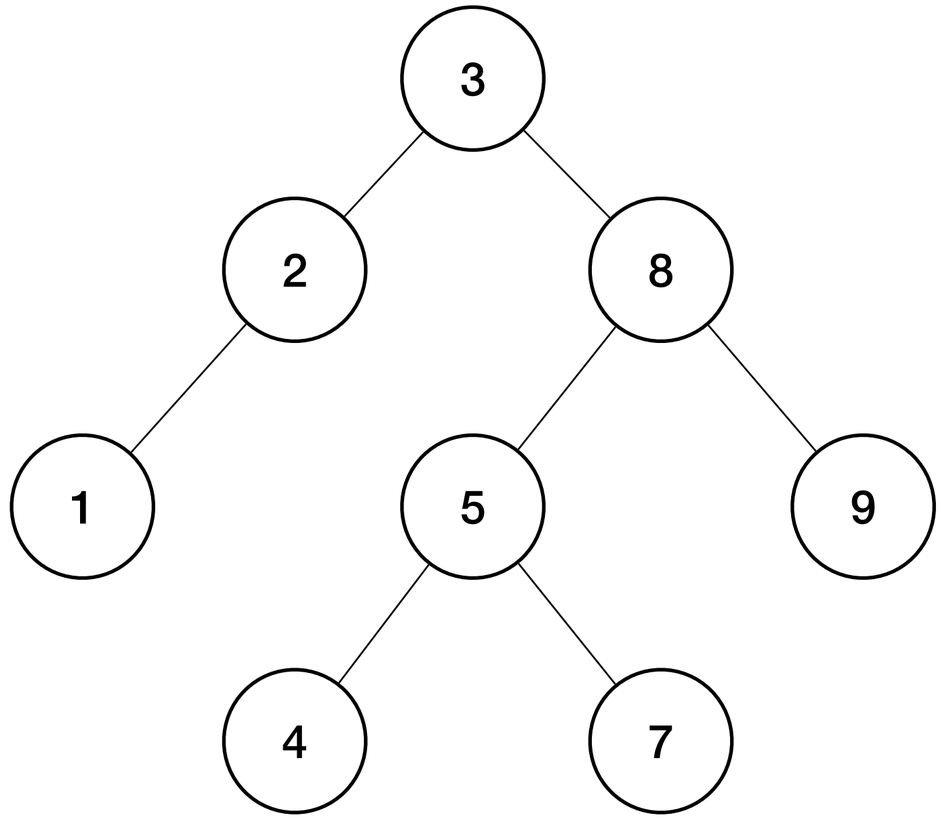
Insertion in a BST

- Insertions need to preserve the BST property



Insertion in a BST

- Insertions need to preserve the BST property
- Add new nodes only as leaf nodes

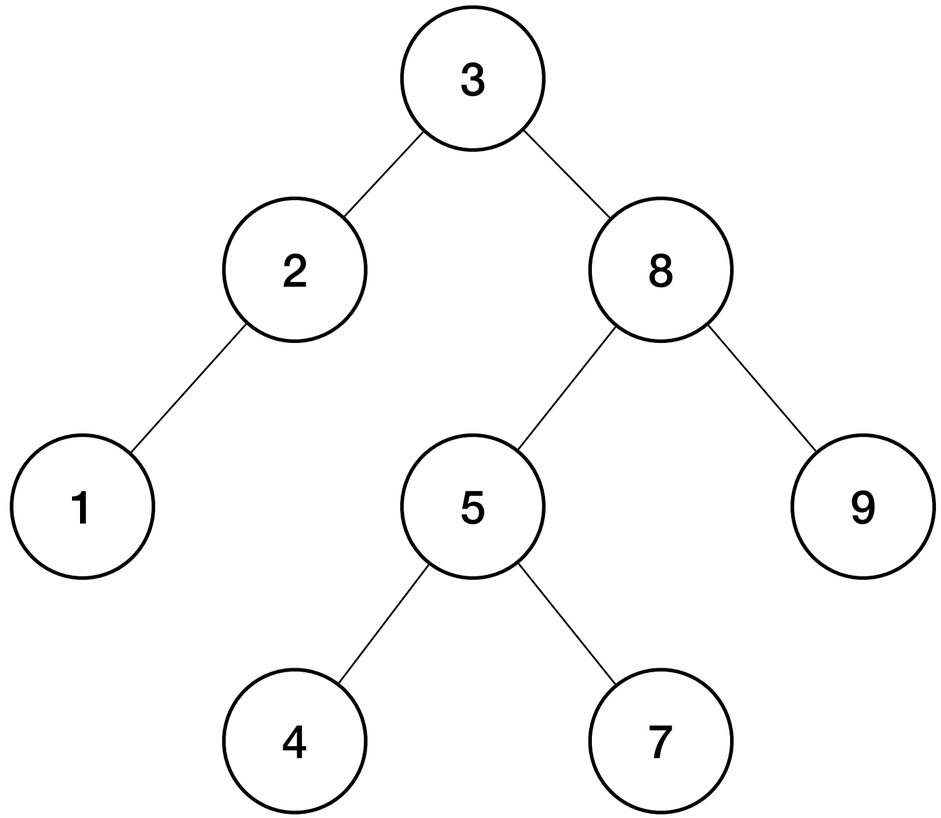


Insertion in a BST

- Insertions need to preserve the BST property

- Add new nodes only as leaf nodes

- Consider inserting 6 in the BST on the right …

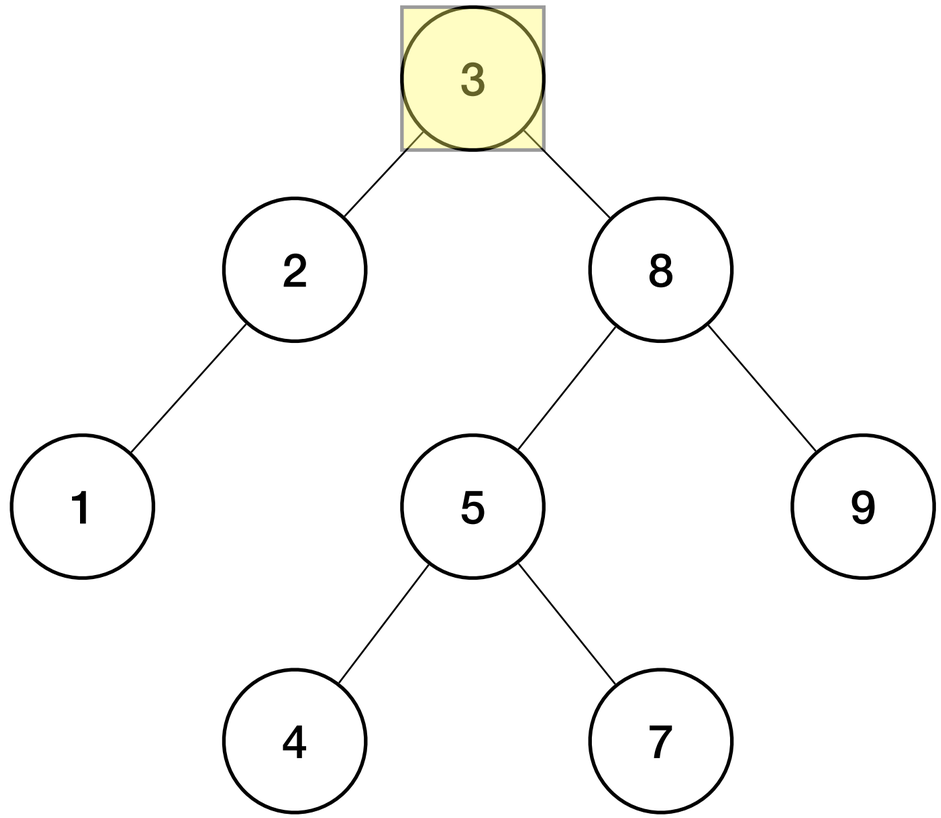


Insertion in a BST

- Insertions need to preserve the BST property

- Add new nodes only as leaf nodes

- Consider inserting 6 in the BST on the right …

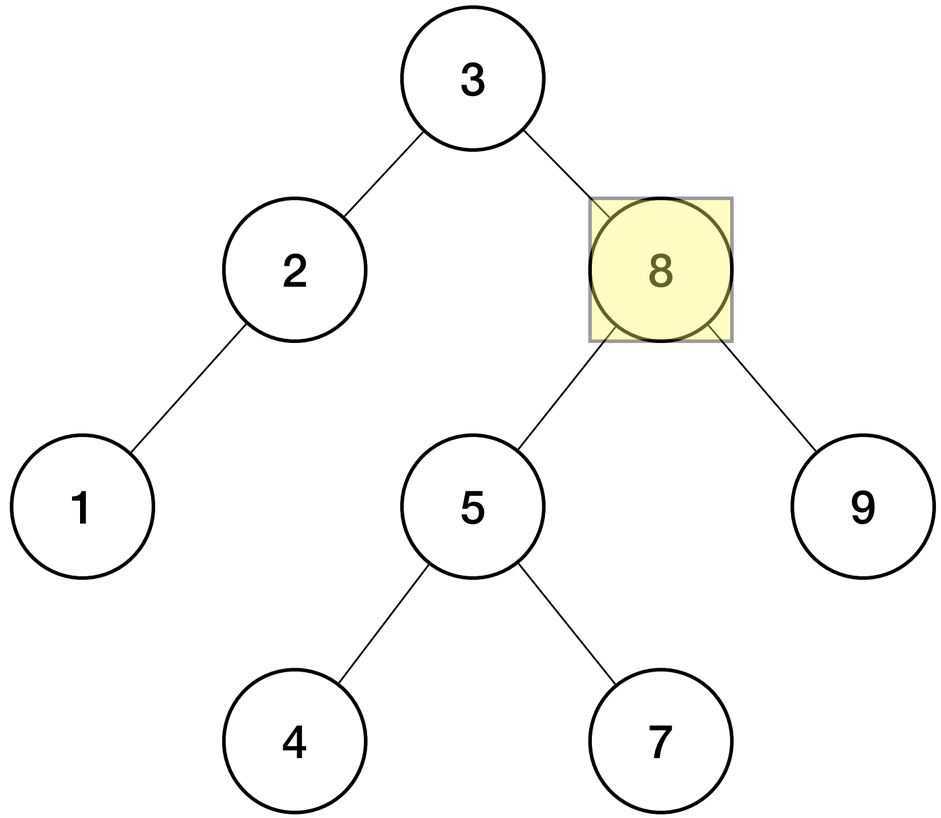


Insertion in a BST

- Insertions need to preserve the BST property

- Add new nodes only as leaf nodes

- Consider inserting 6 in the BST on the right …

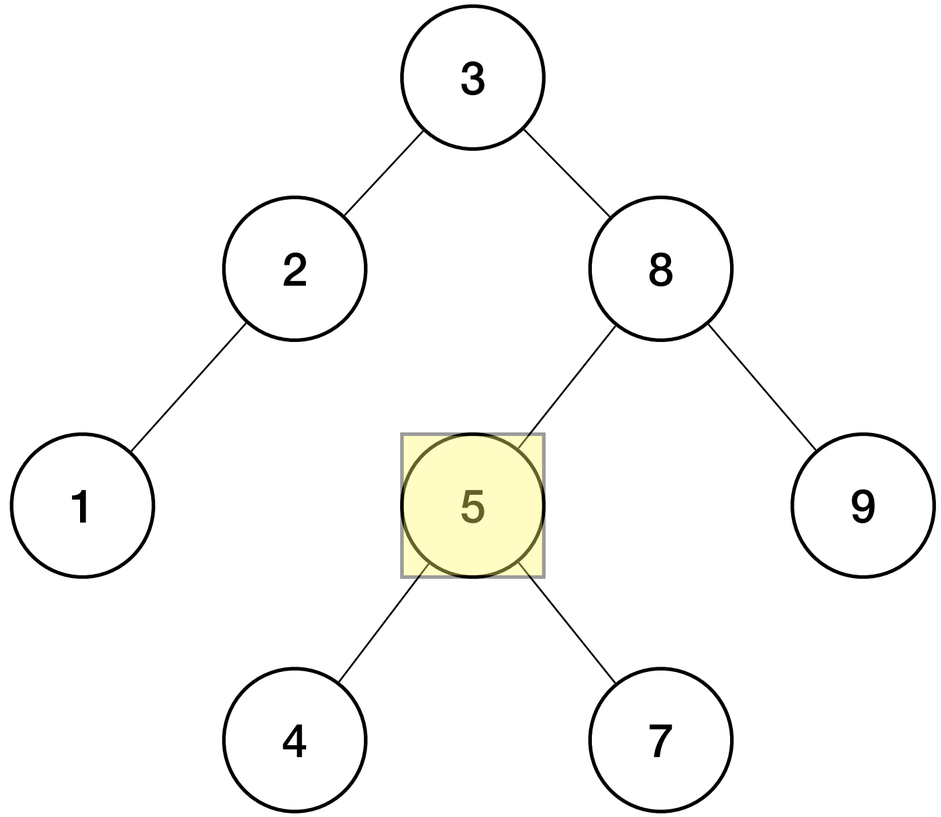


Insertion in a BST

- Insertions need to preserve the BST property

- Add new nodes only as leaf nodes

- Consider inserting 6 in the BST on the right …

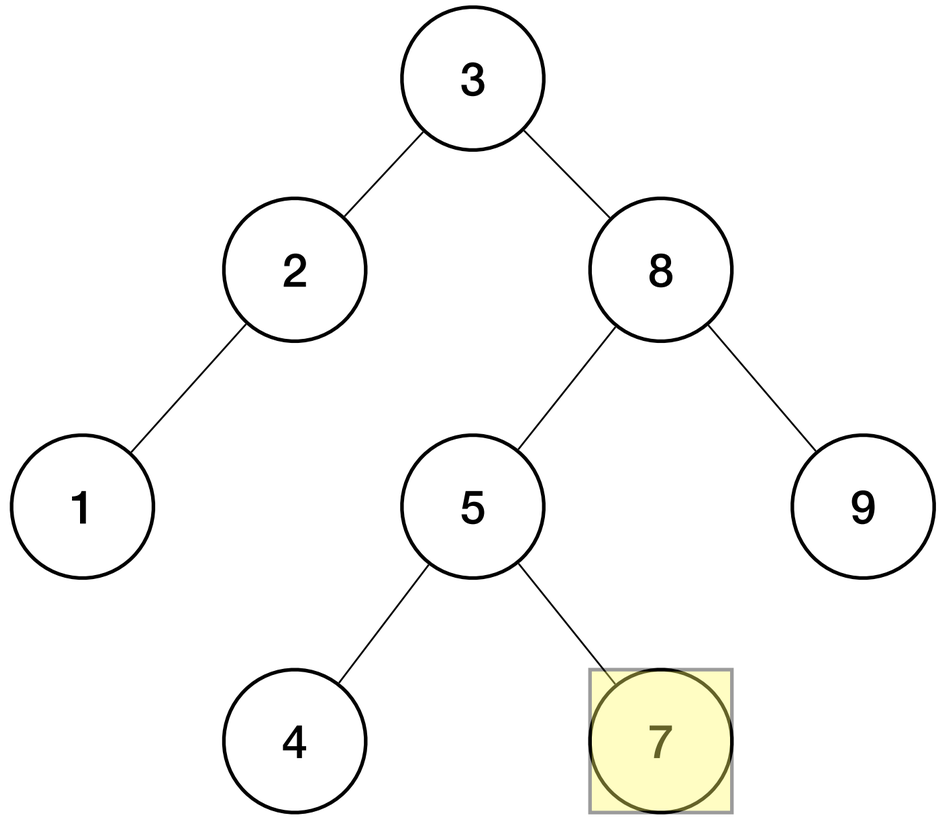


Insertion in a BST

- Insertions need to preserve the BST property

- Add new nodes only as leaf nodes

- Consider inserting 6 in the BST on the right …

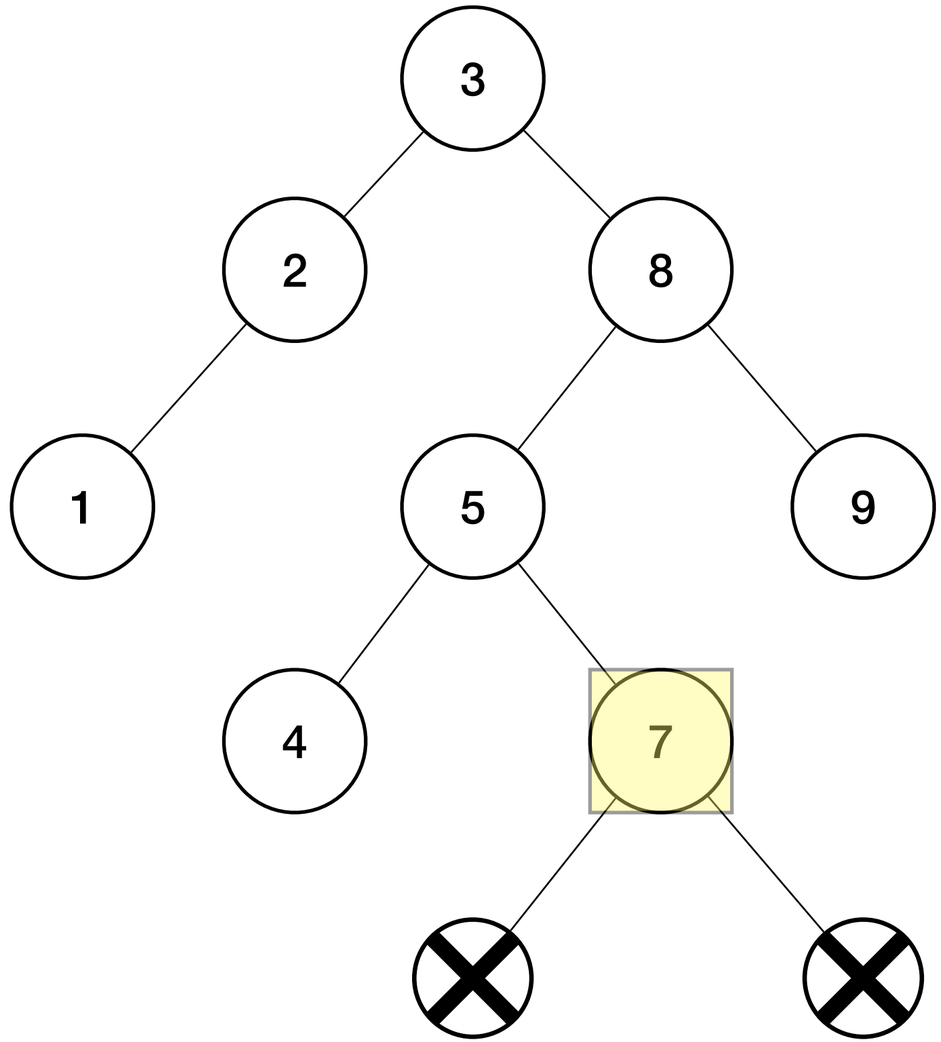


Insertion in a BST

- Insertions need to preserve the BST property

- Add new nodes only as leaf nodes

- Consider inserting 6 in the BST on the right …

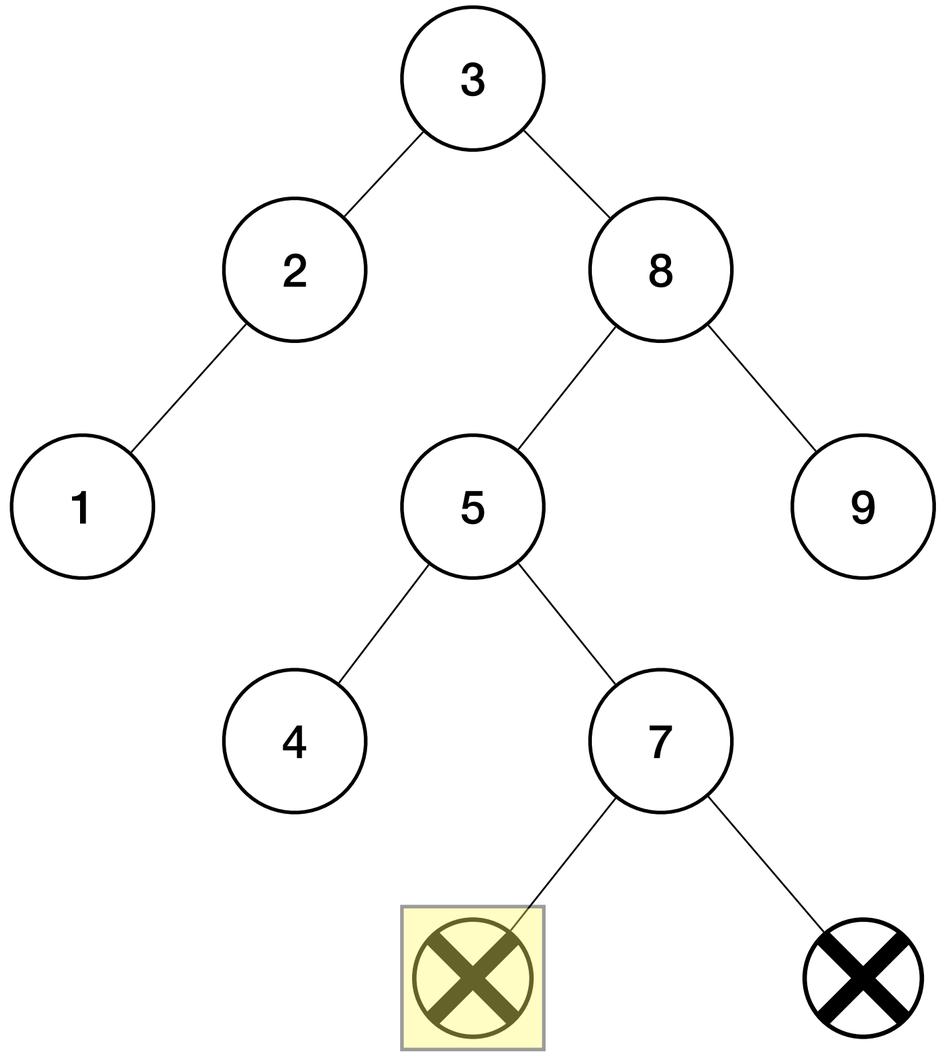


Insertion in a BST

- Insertions need to preserve the BST property

- Add new nodes only as leaf nodes

- Consider inserting 6 in the BST on the right …

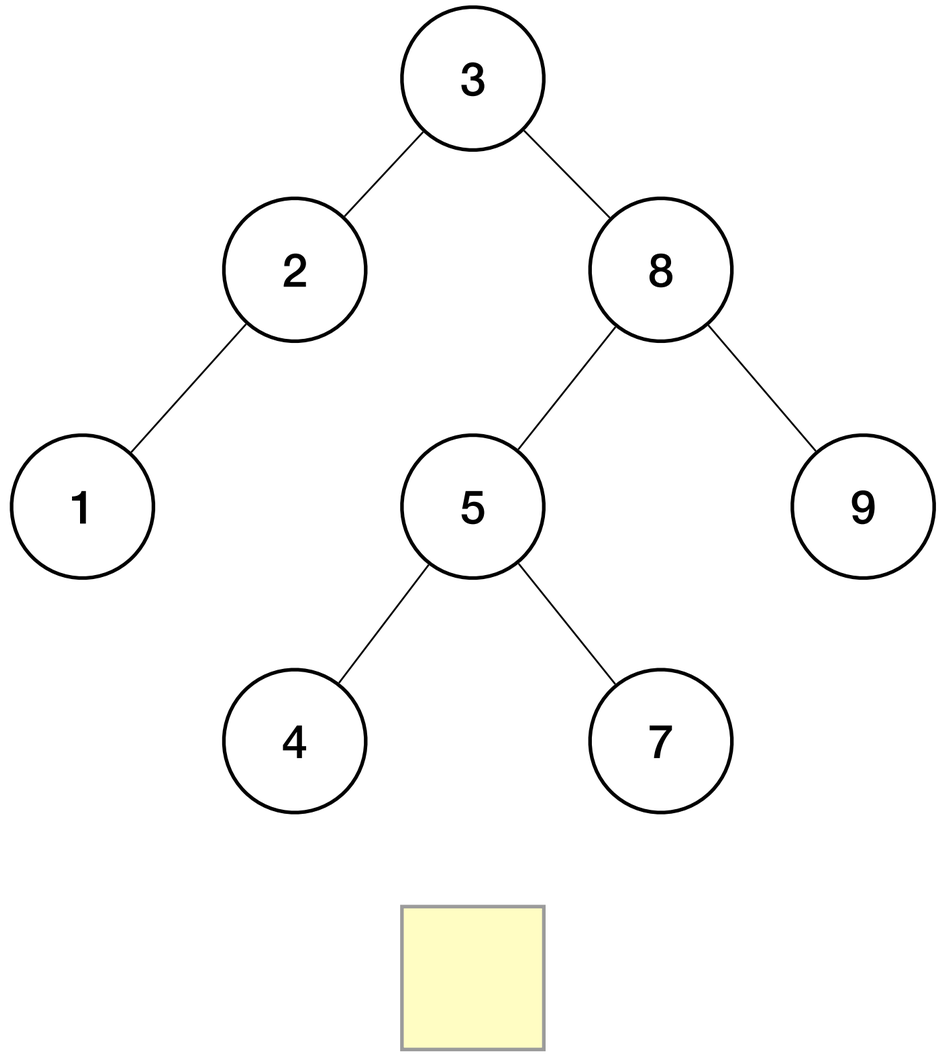


Insertion in a BST

- Insertions need to preserve the BST property

- Add new nodes only as leaf nodes

- Consider inserting 6 in the BST on the right …

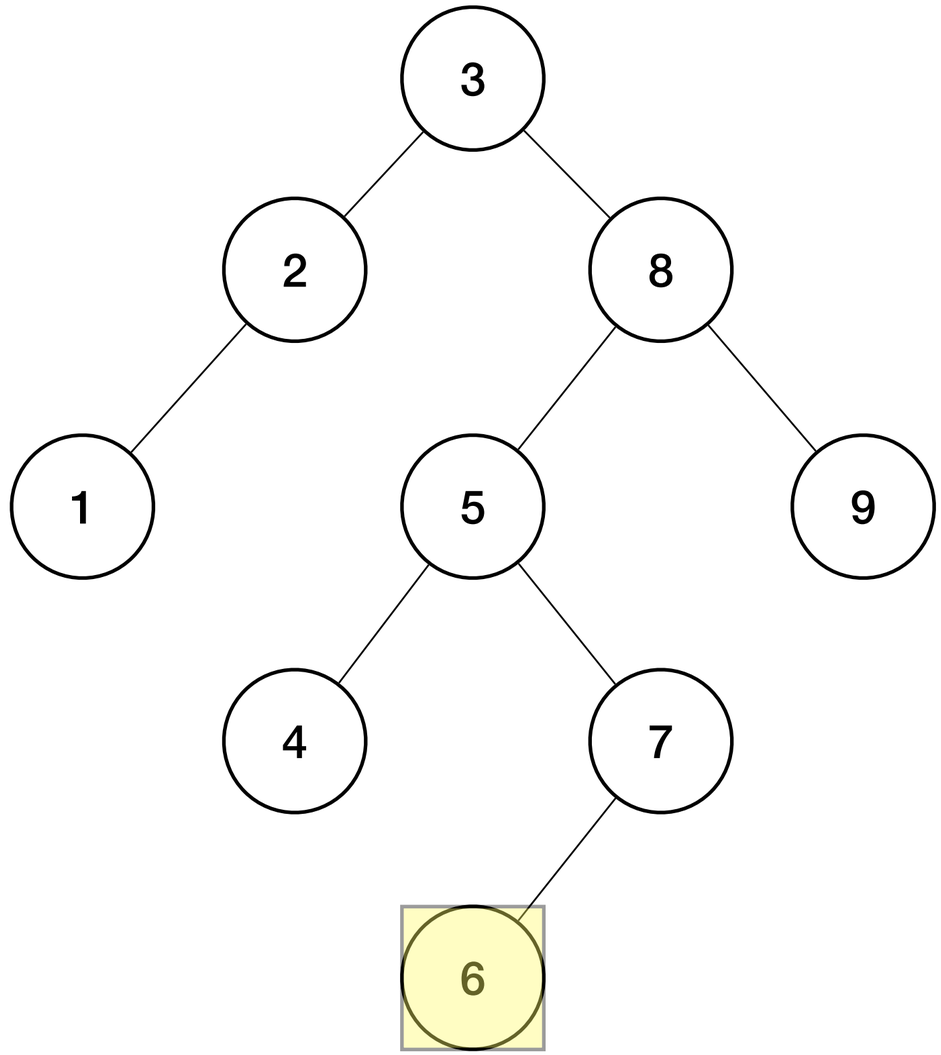


Insertion in a BST

- Insertions need to preserve the BST property

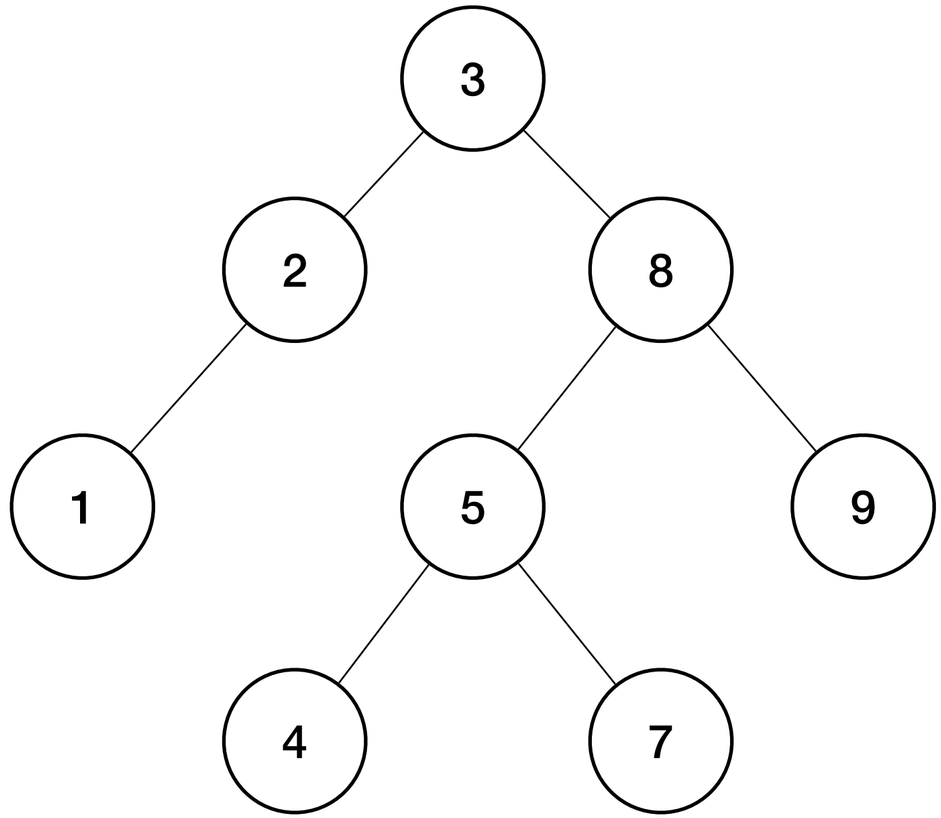
- Add new nodes only as leaf nodes

- Consider inserting 6 in the BST on the right …



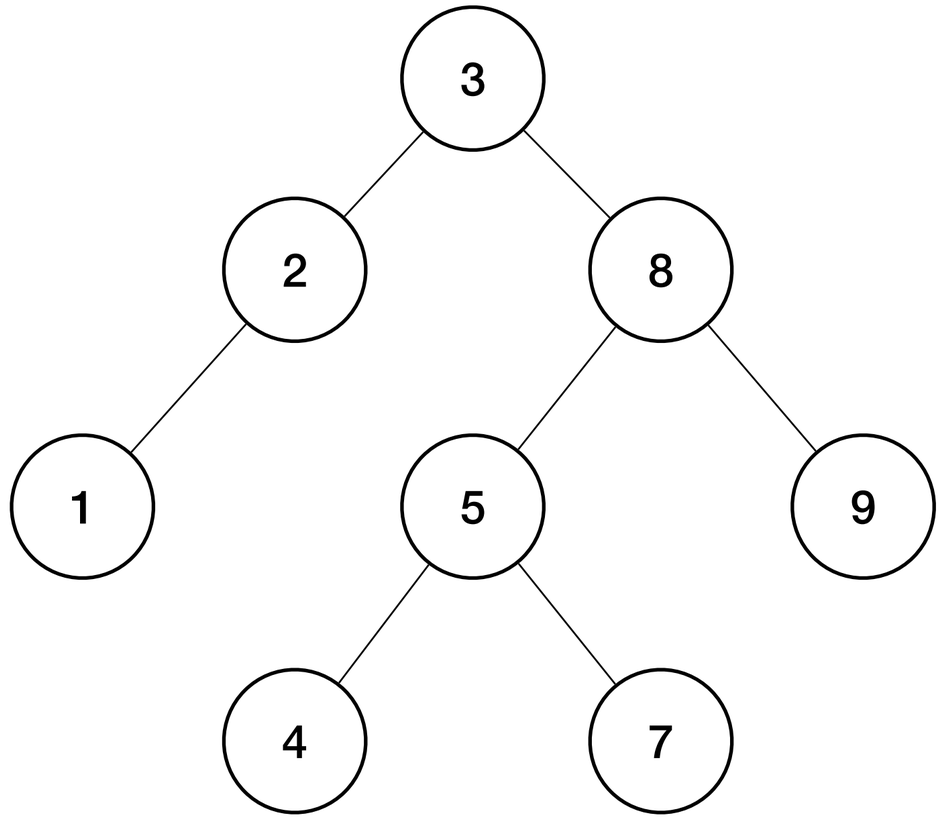
Insertion in a BST

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



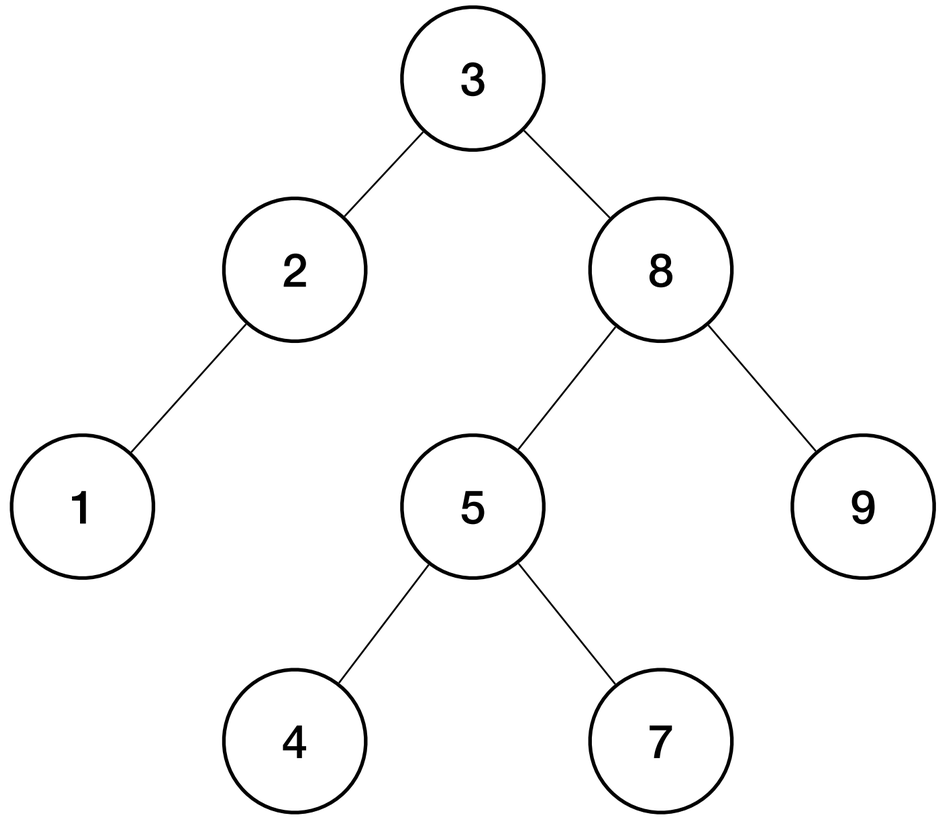
Insertion in a BST

```
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 =
    return cursor;
  }
}
```

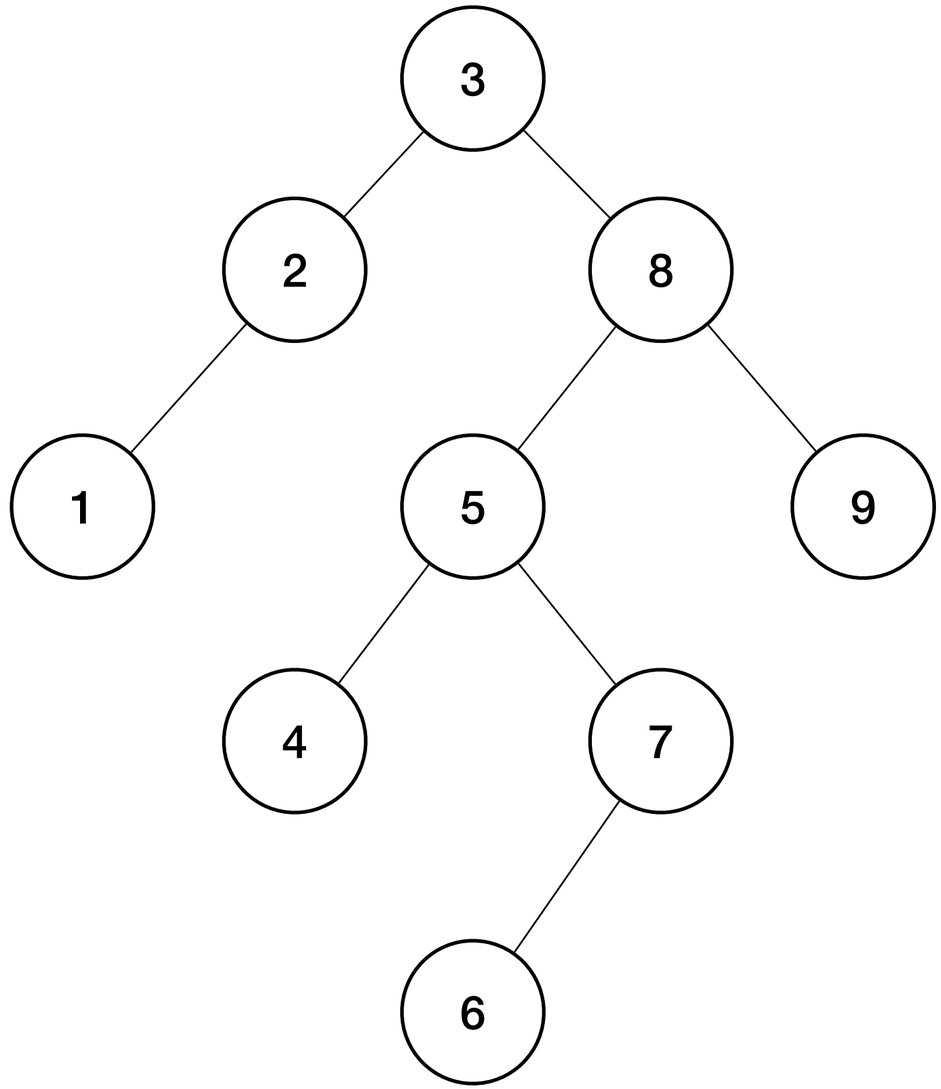


Insertion in a BST

```
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;
  }
}
```

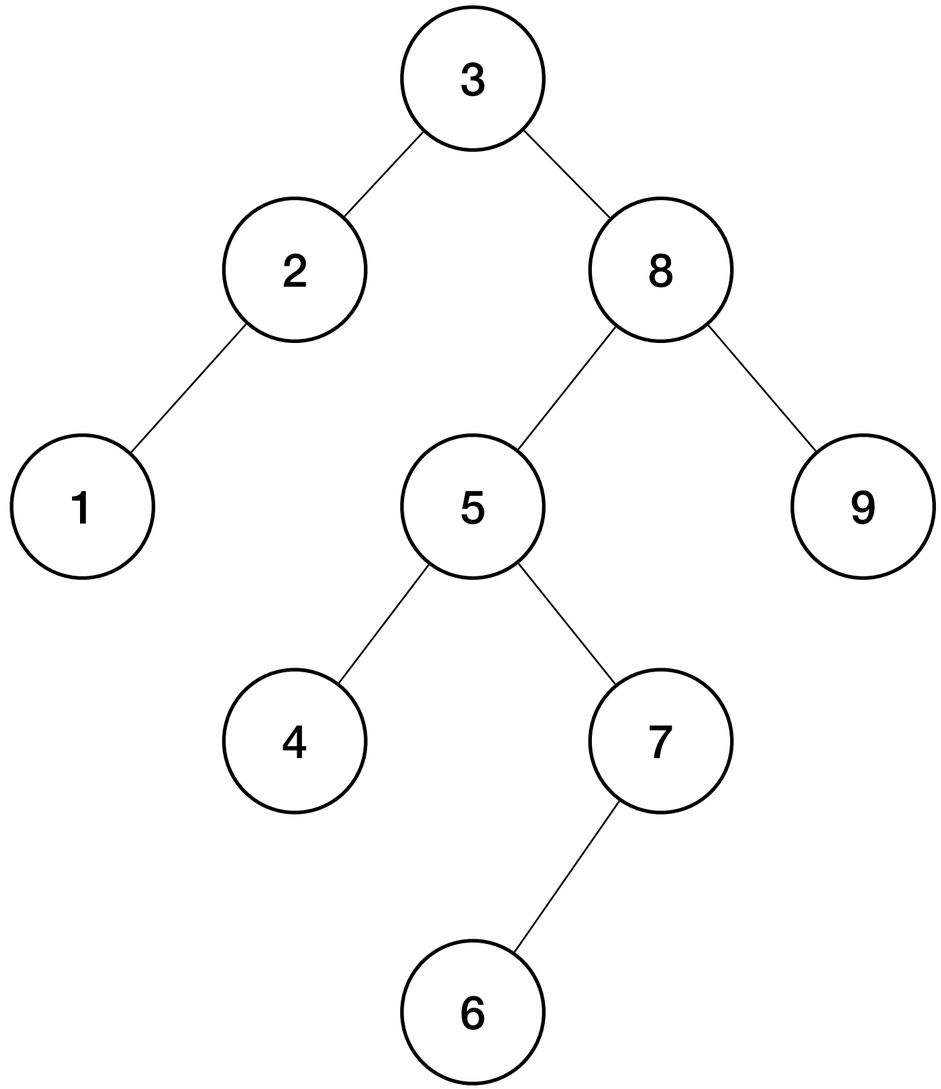


Finding height of a tree



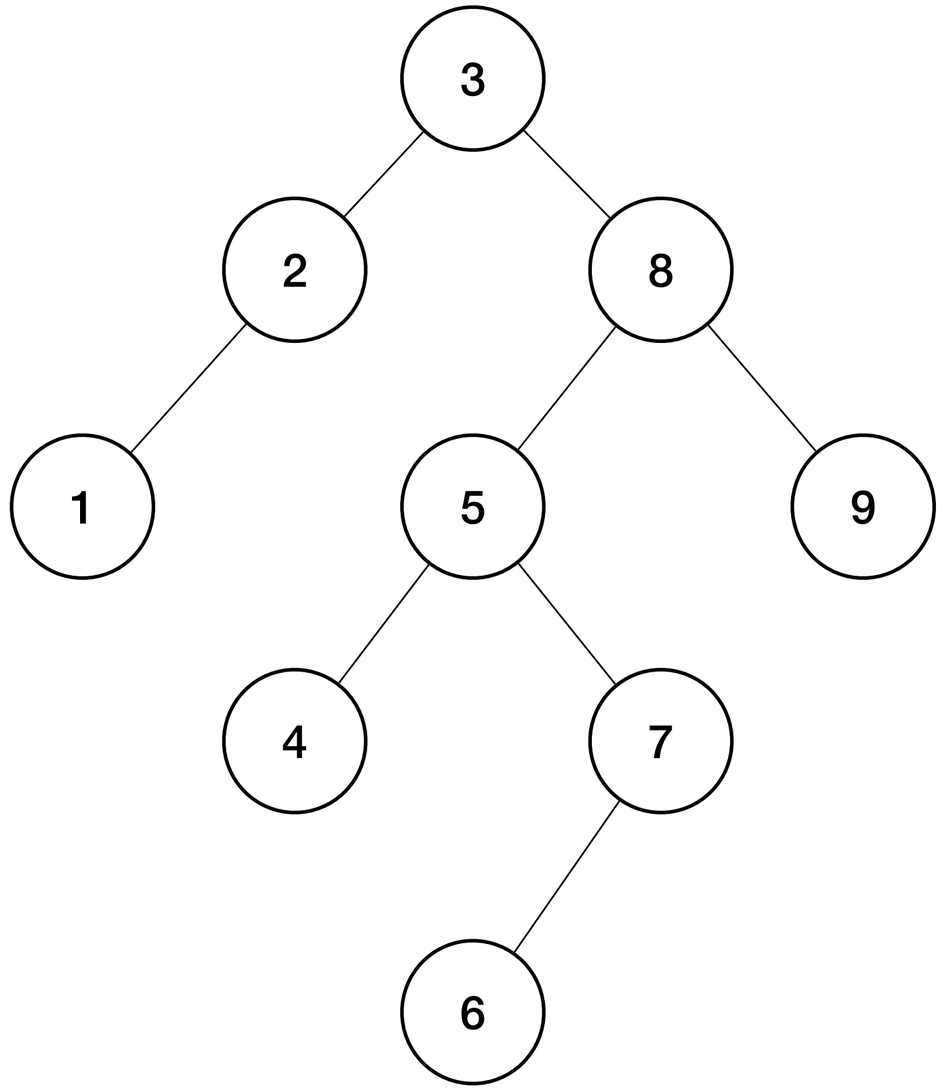
Finding height of a tree

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



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)

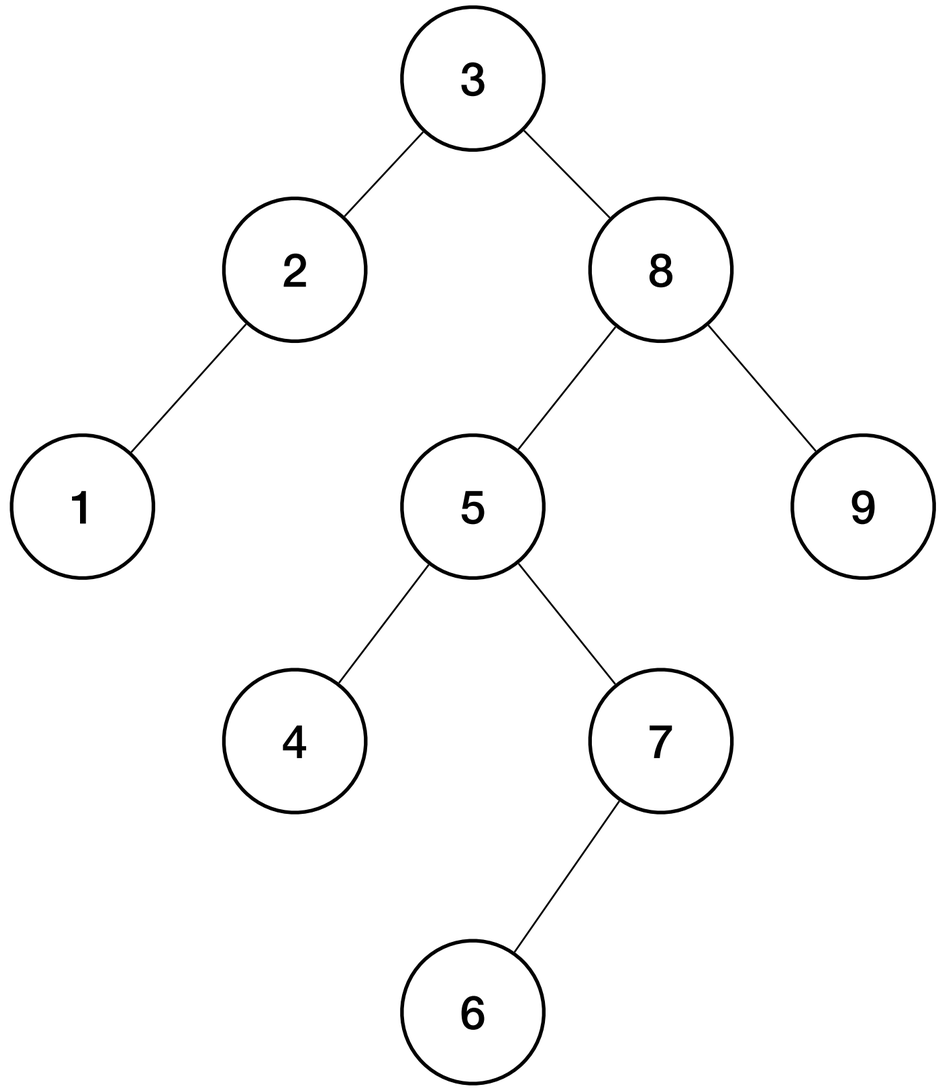


Finding height of a tree

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

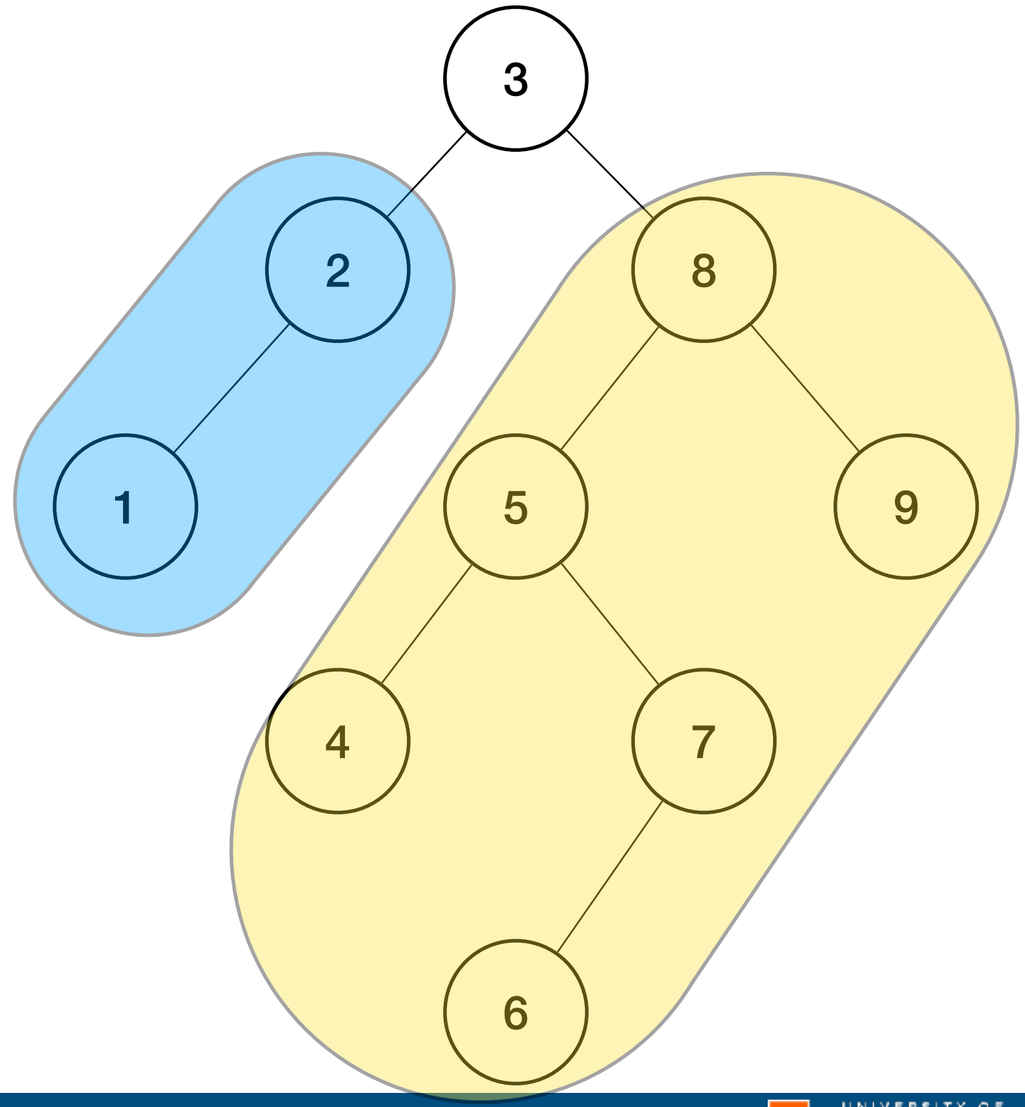
- Recursively calculate: 1 + height of L/R subtree(s)

- Take maximum at each step



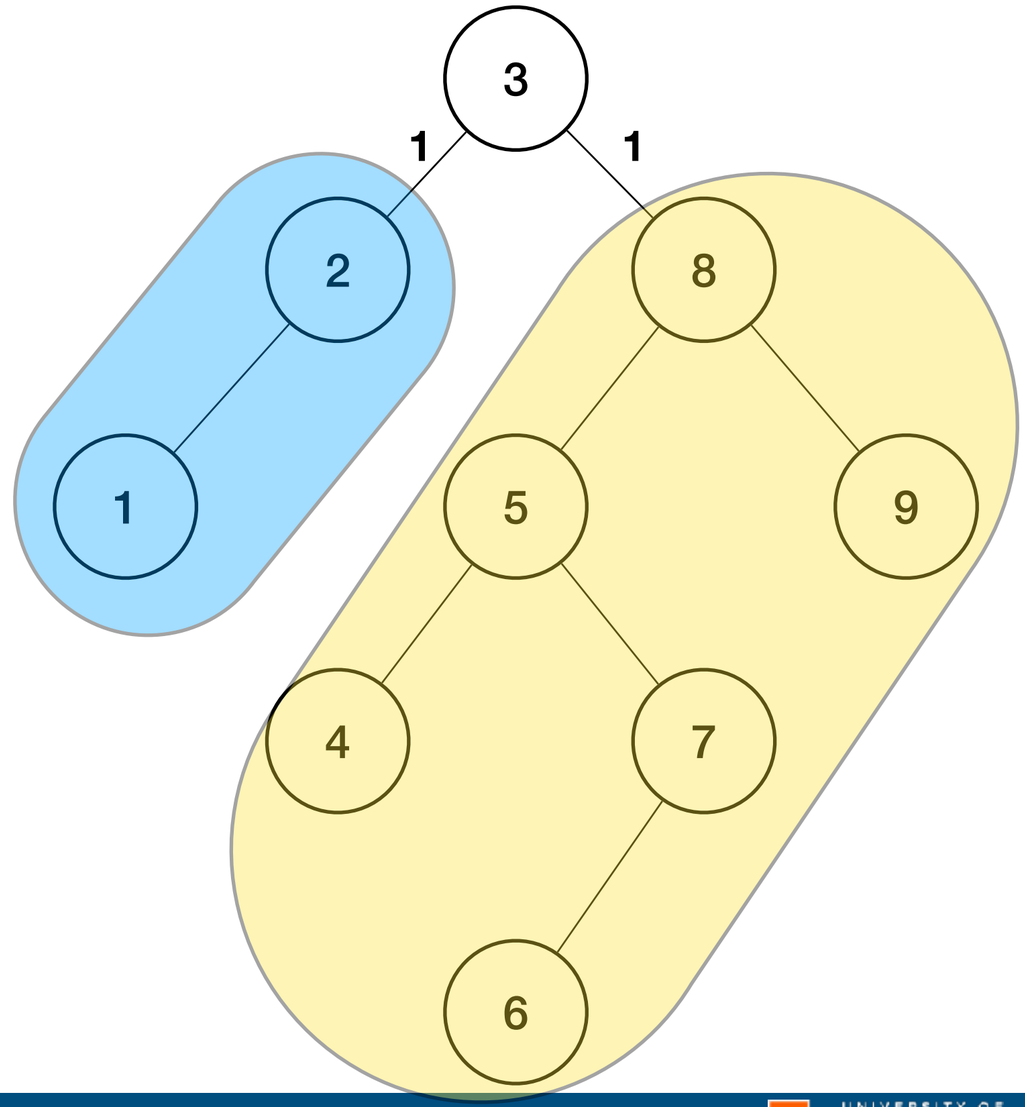
Finding height of a tree

- 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



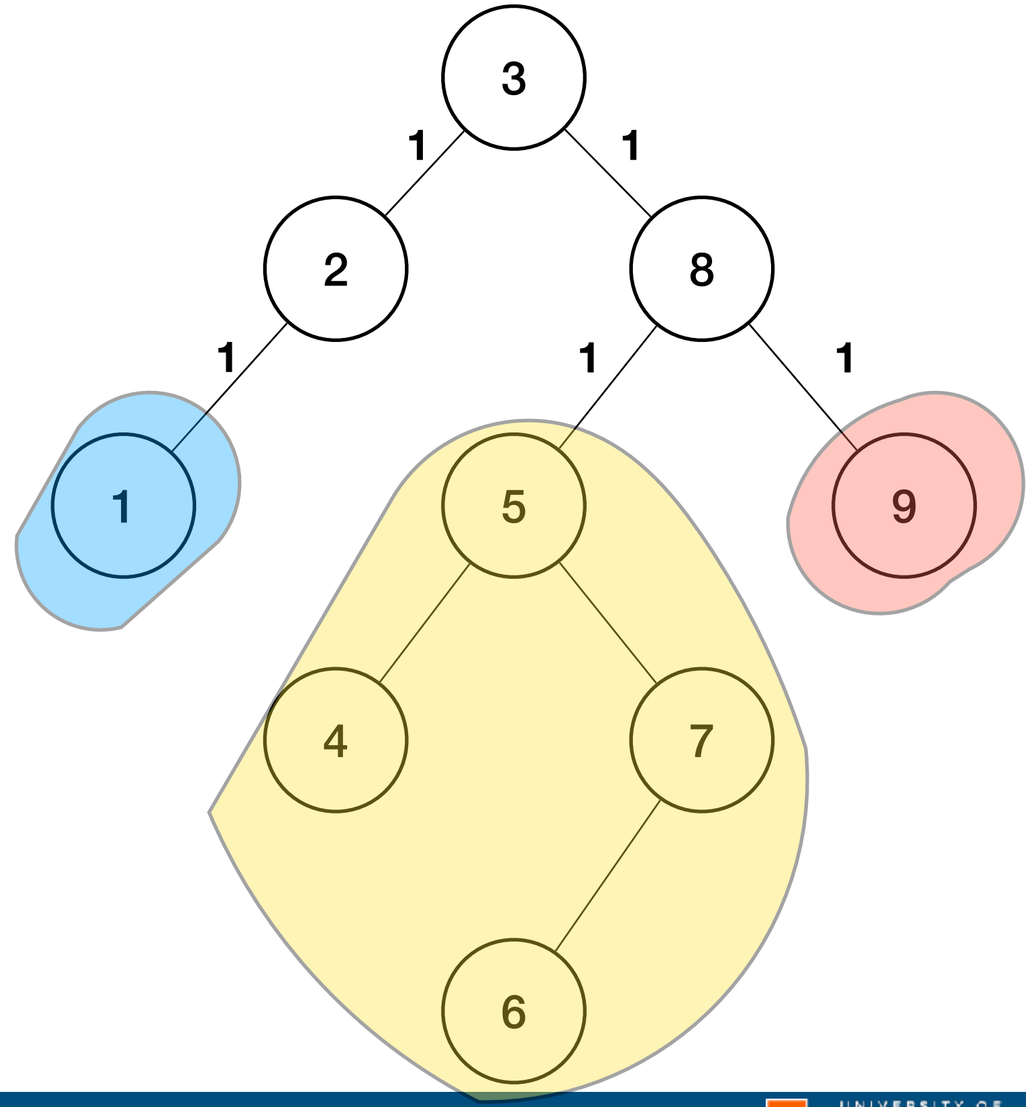
Finding height of a tree

- 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

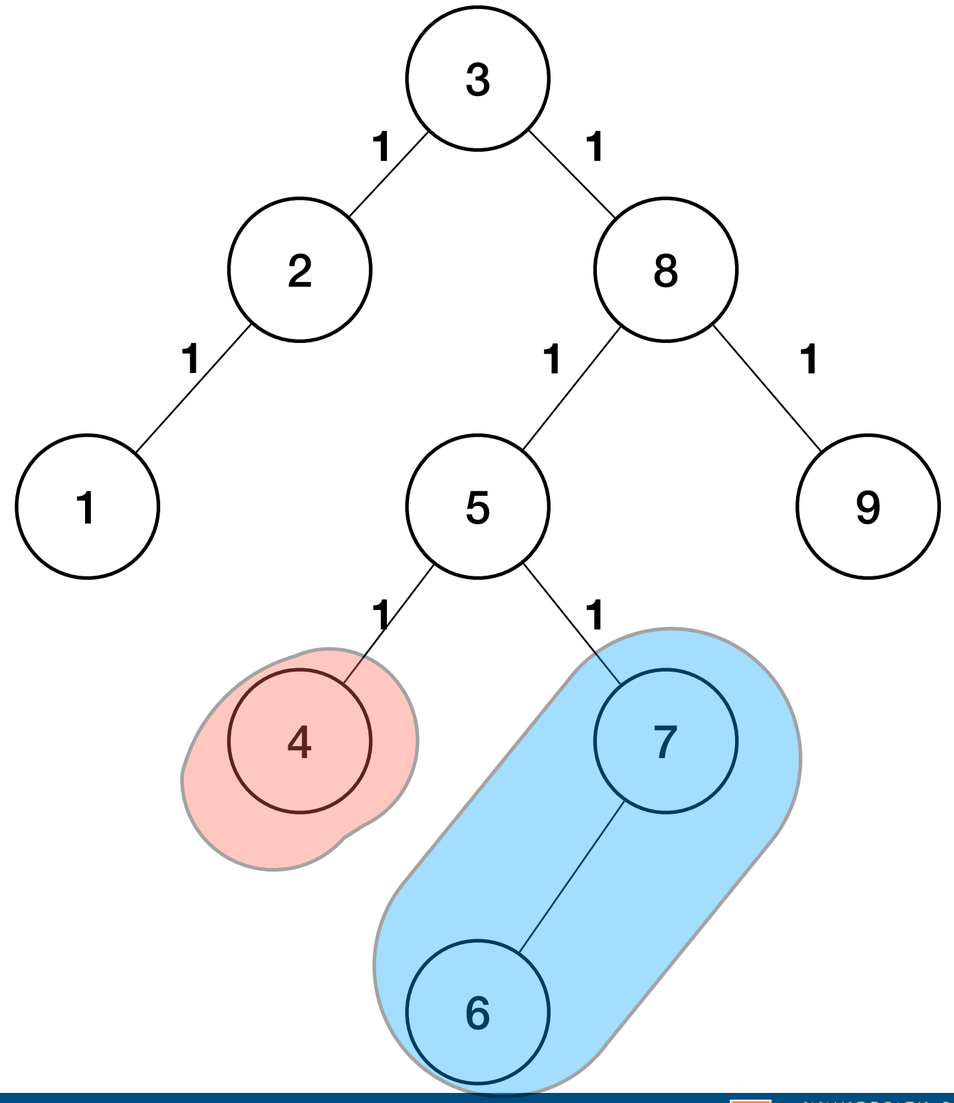


Finding height of a tree

- 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

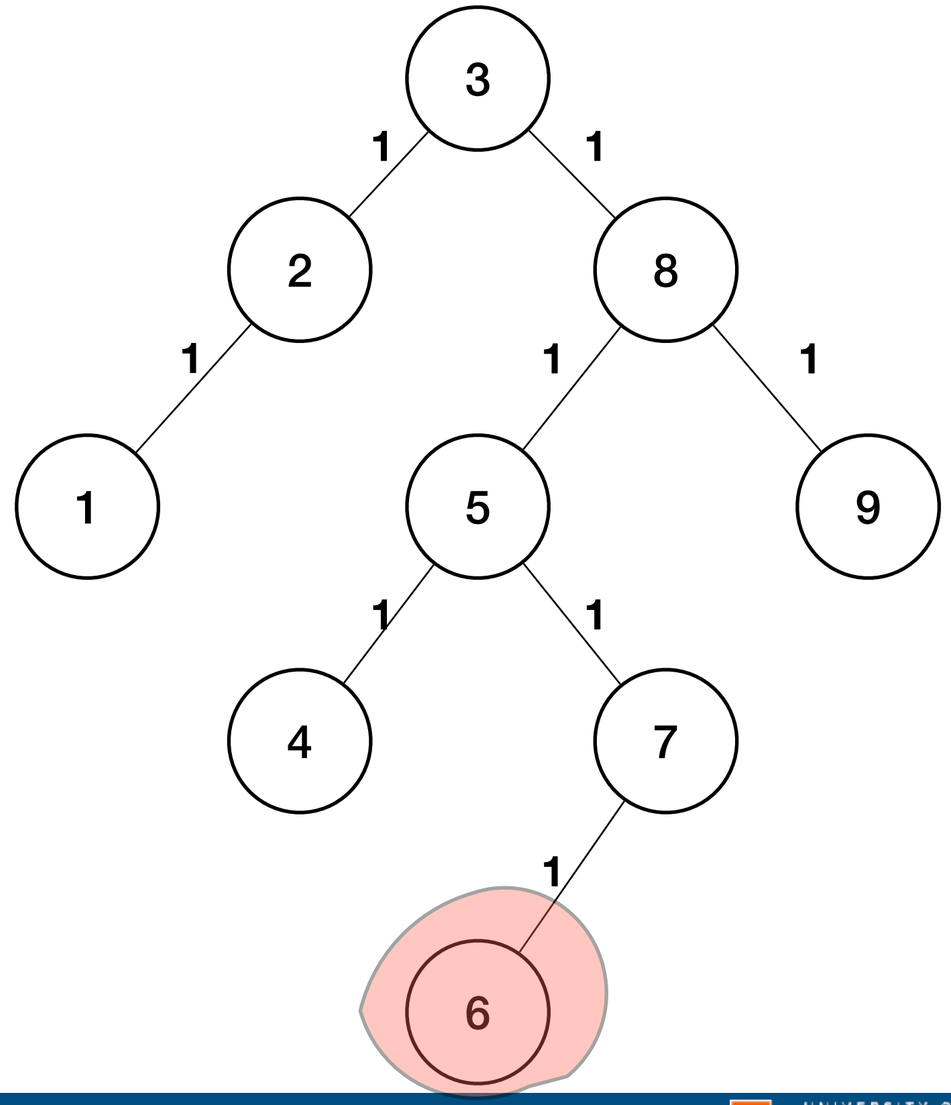


Finding height of a tree

- 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

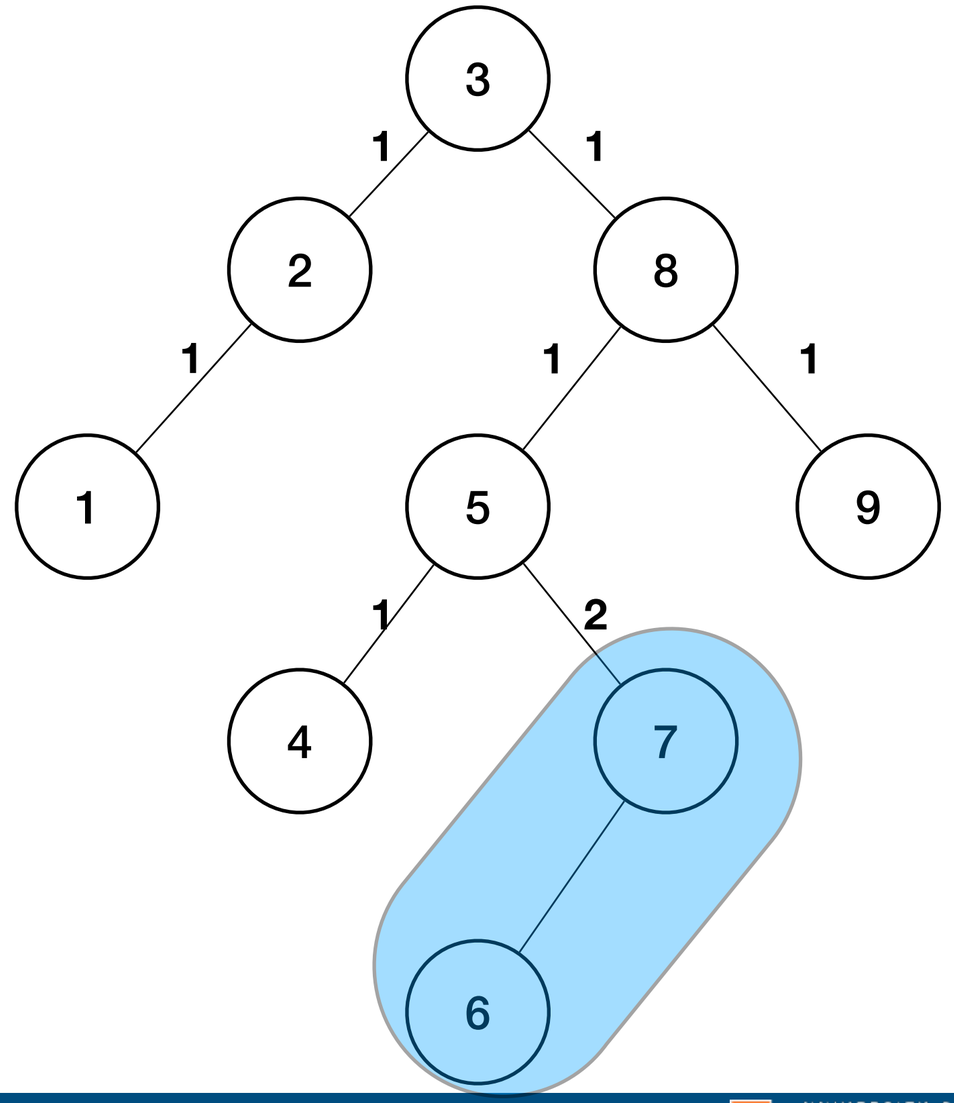


Finding height of a tree

- 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

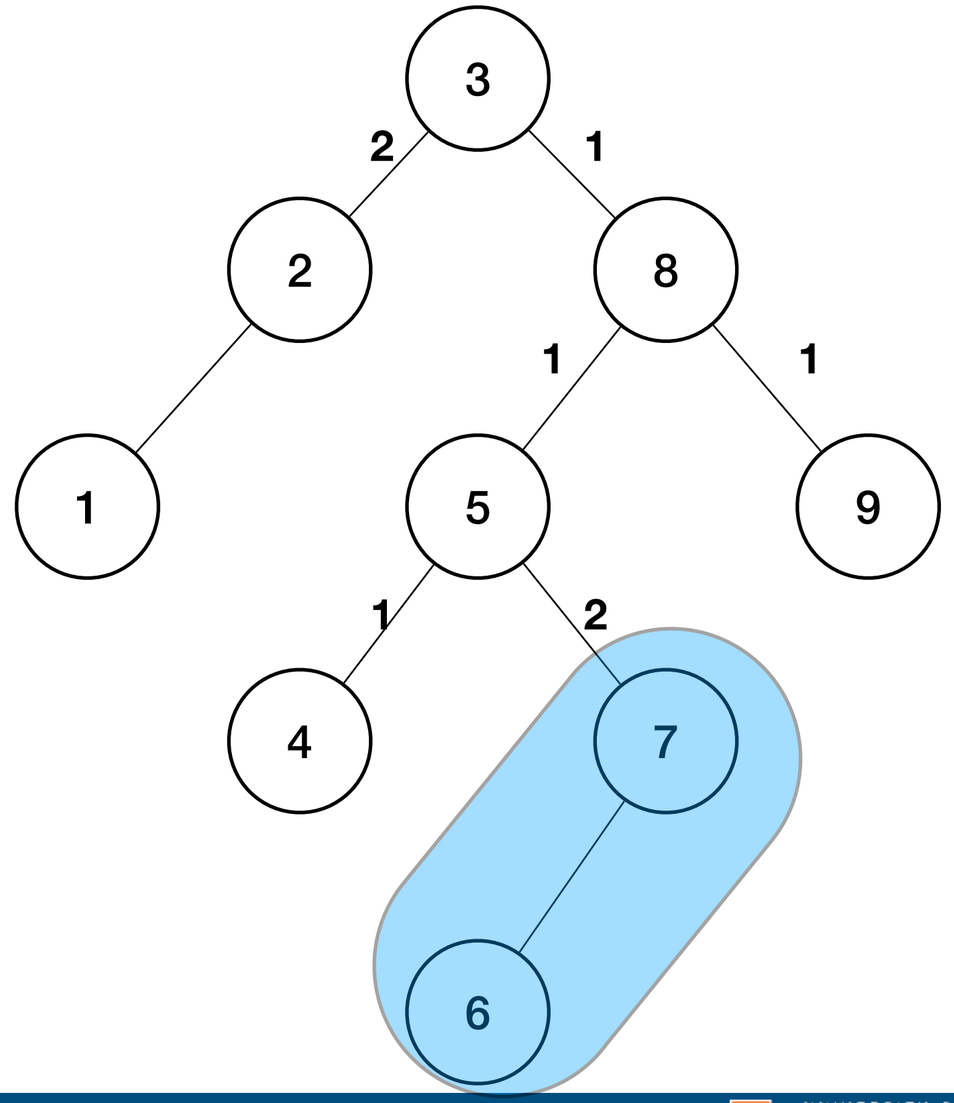


Finding height of a tree

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

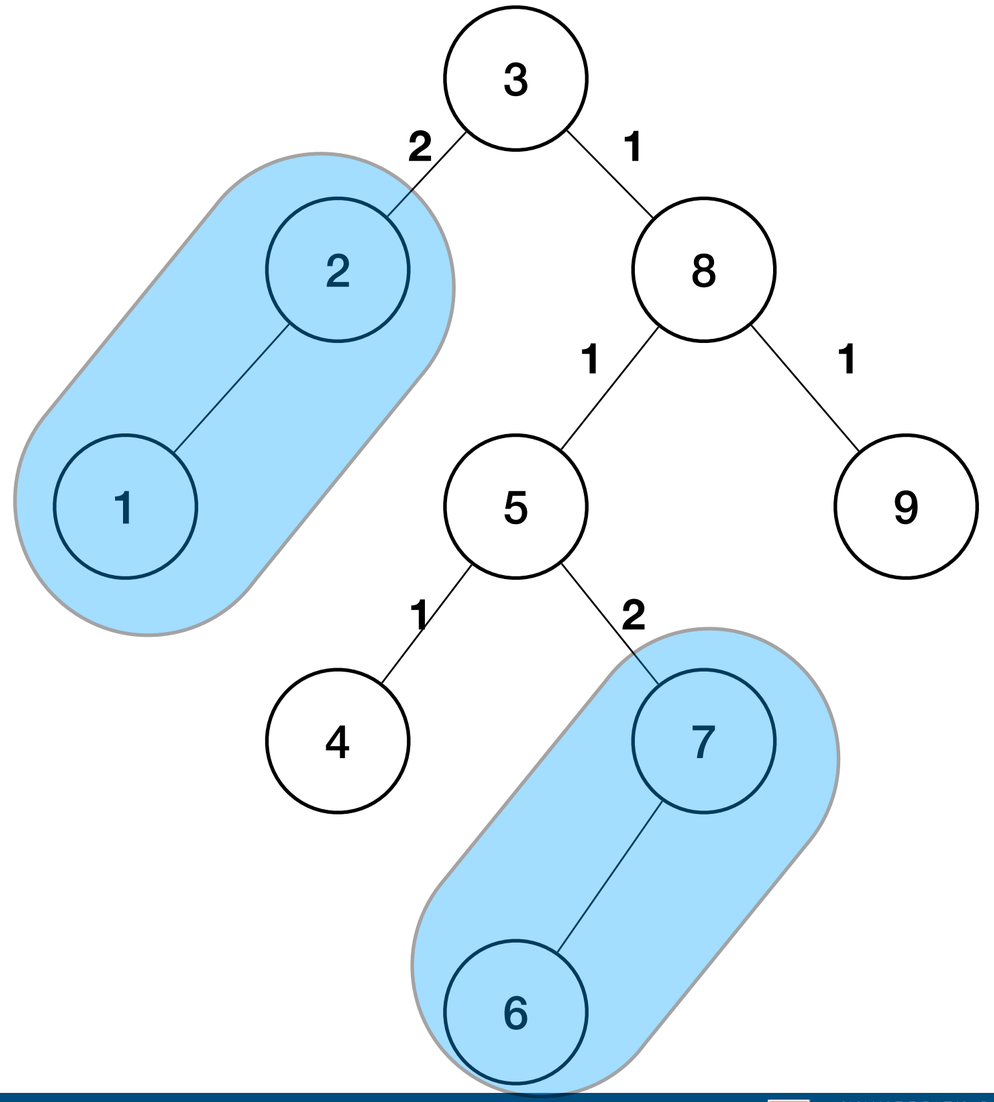
- Recursively calculate: 1 + height of L/R subtree(s)

- 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

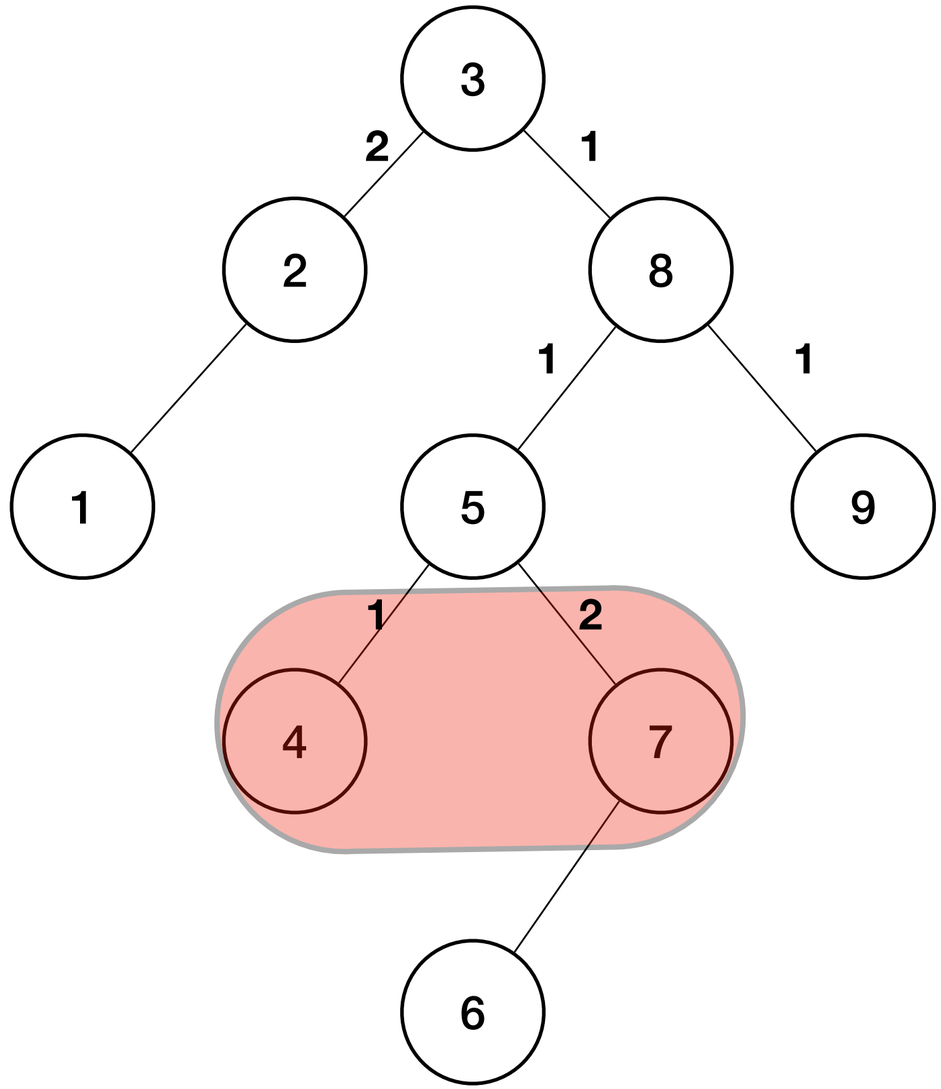


Finding height of a tree

- 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

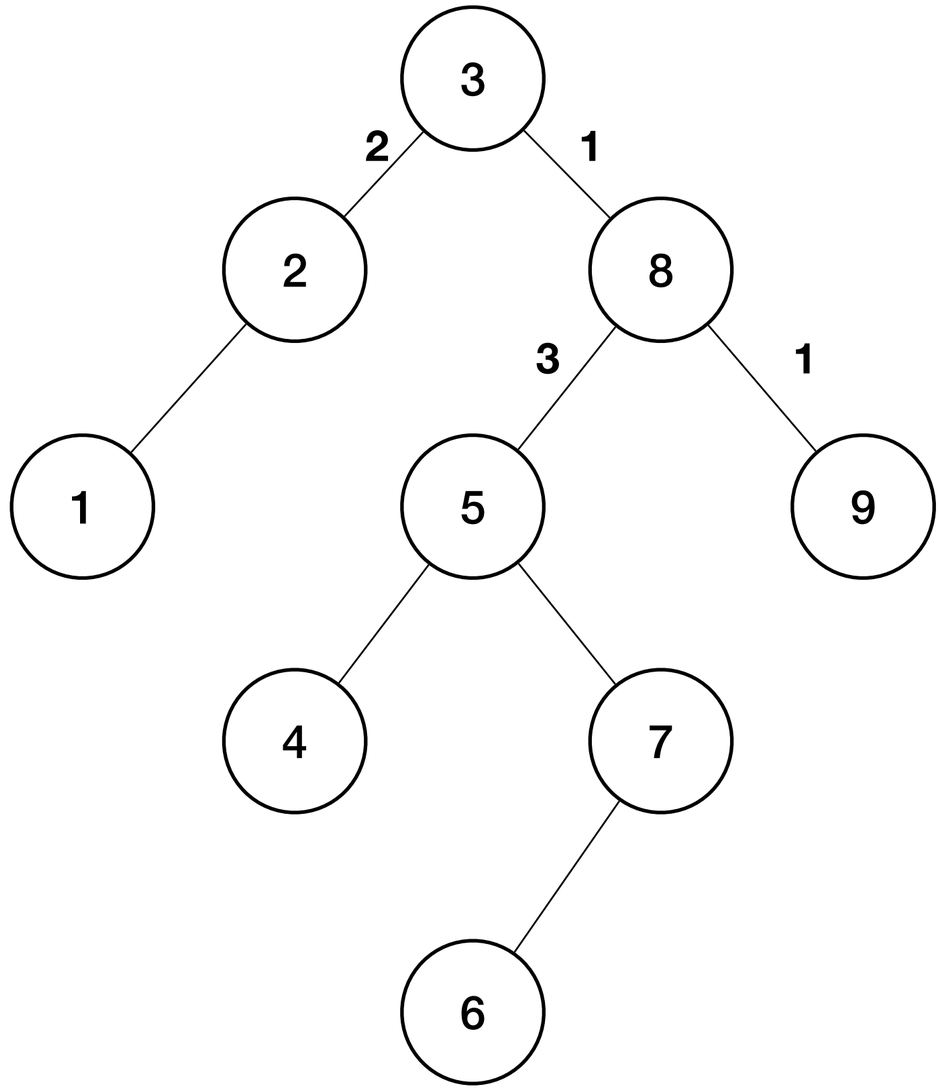


Finding height of a tree

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

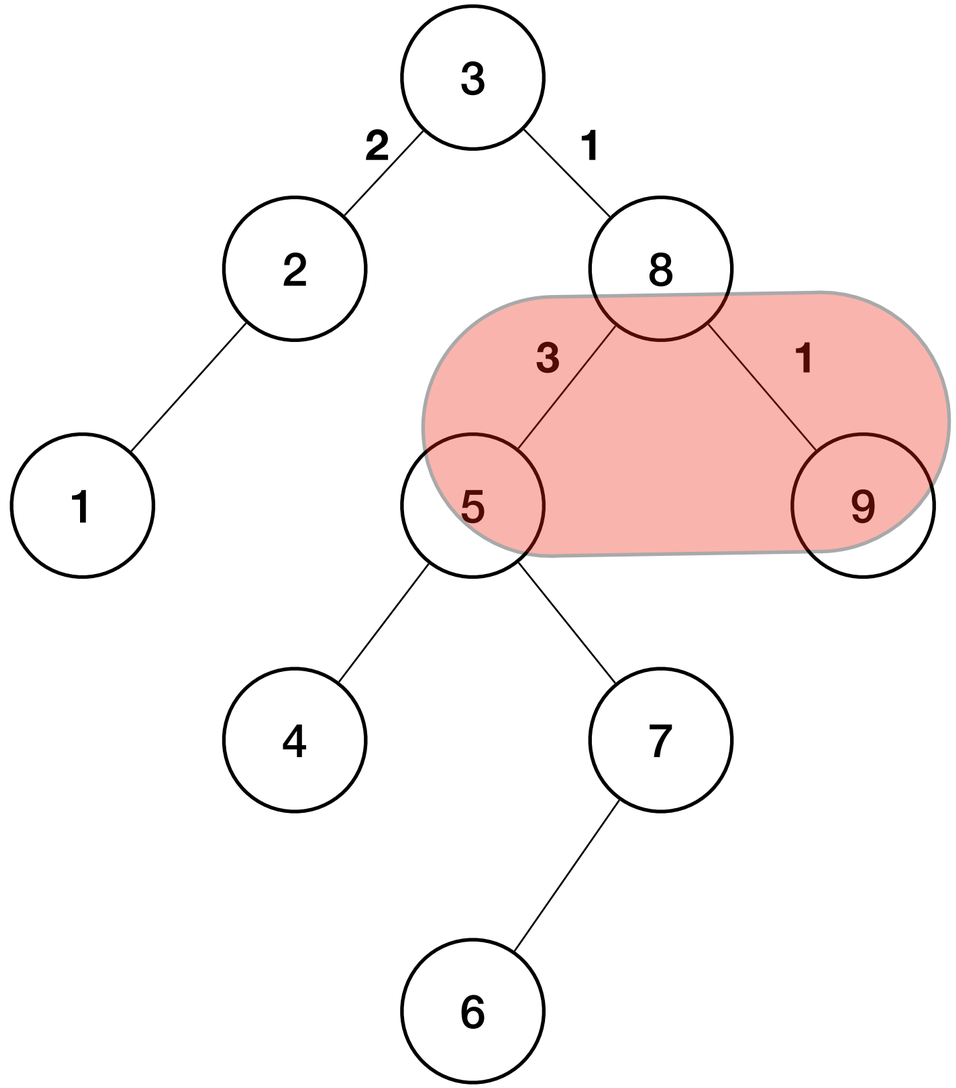
- Recursively calculate: 1 + height of L/R subtree(s)

- 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

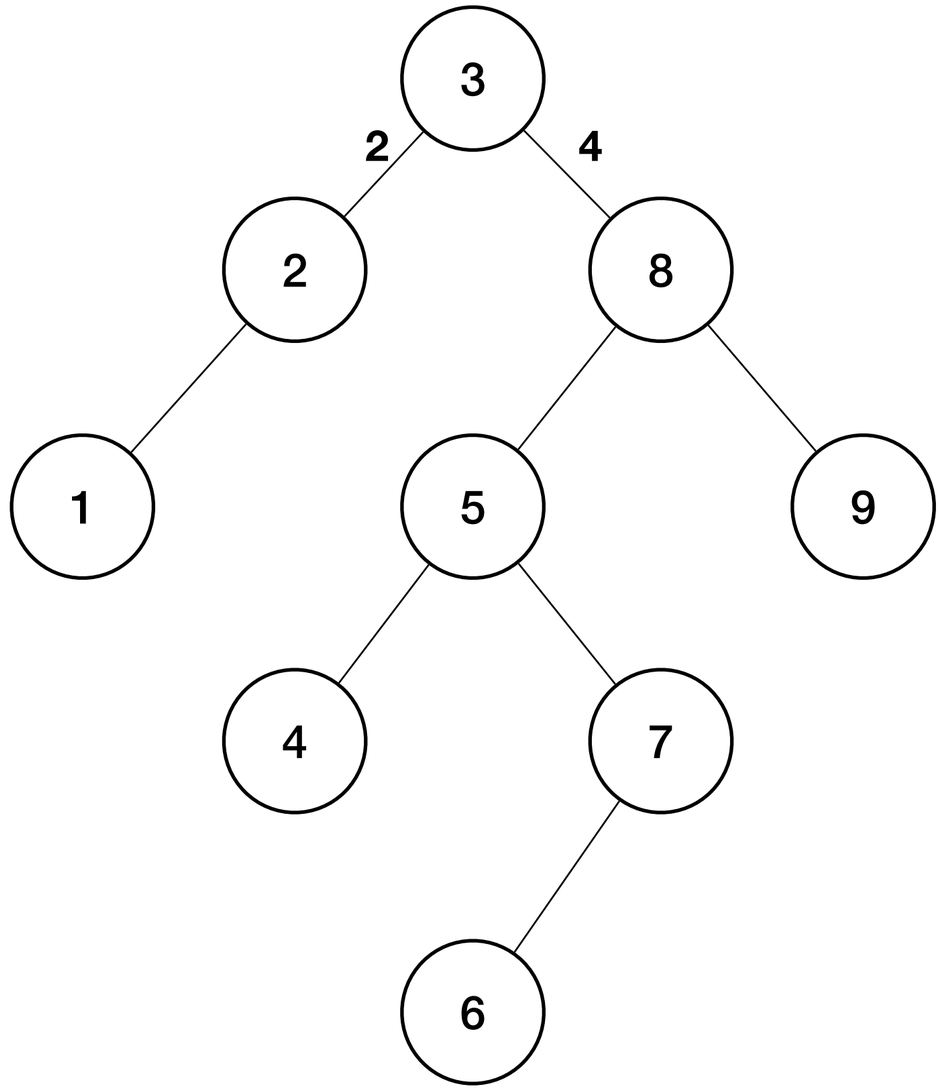


Finding height of a tree

- 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

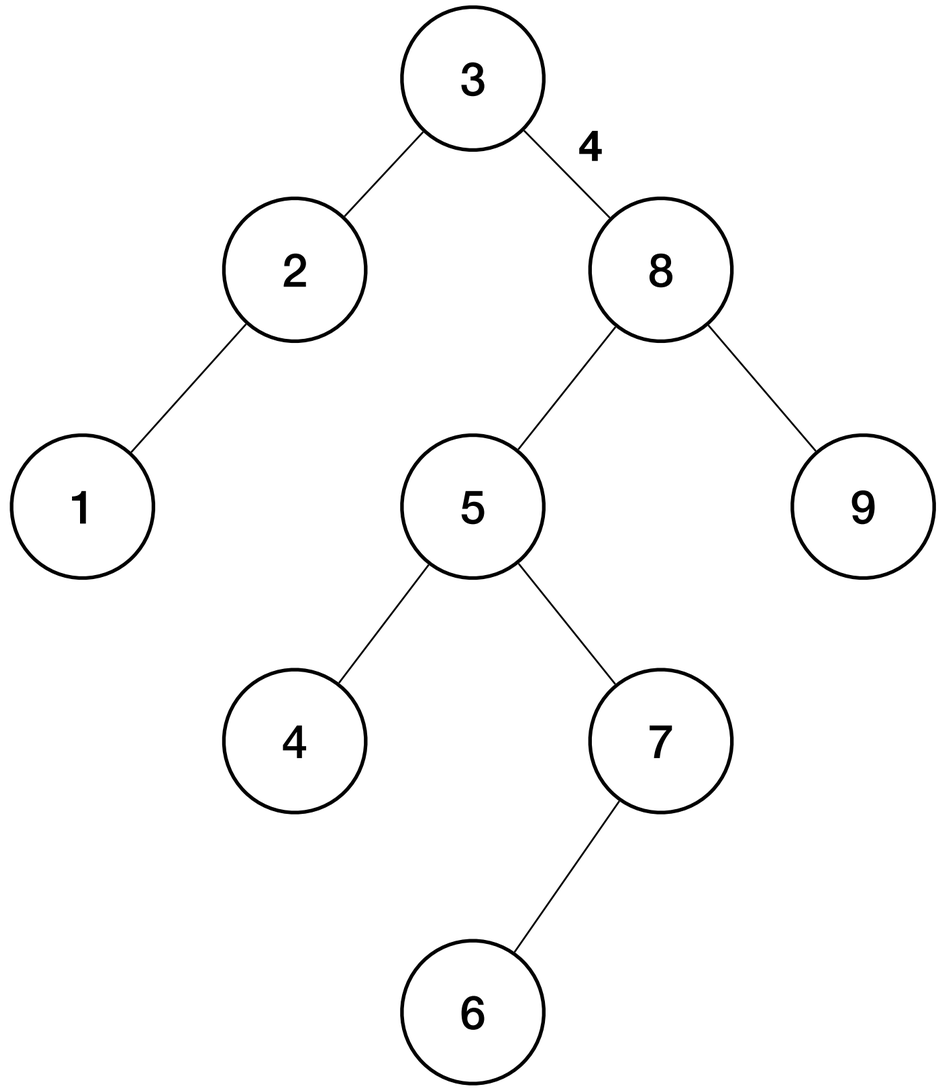


Finding height of a tree

- 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

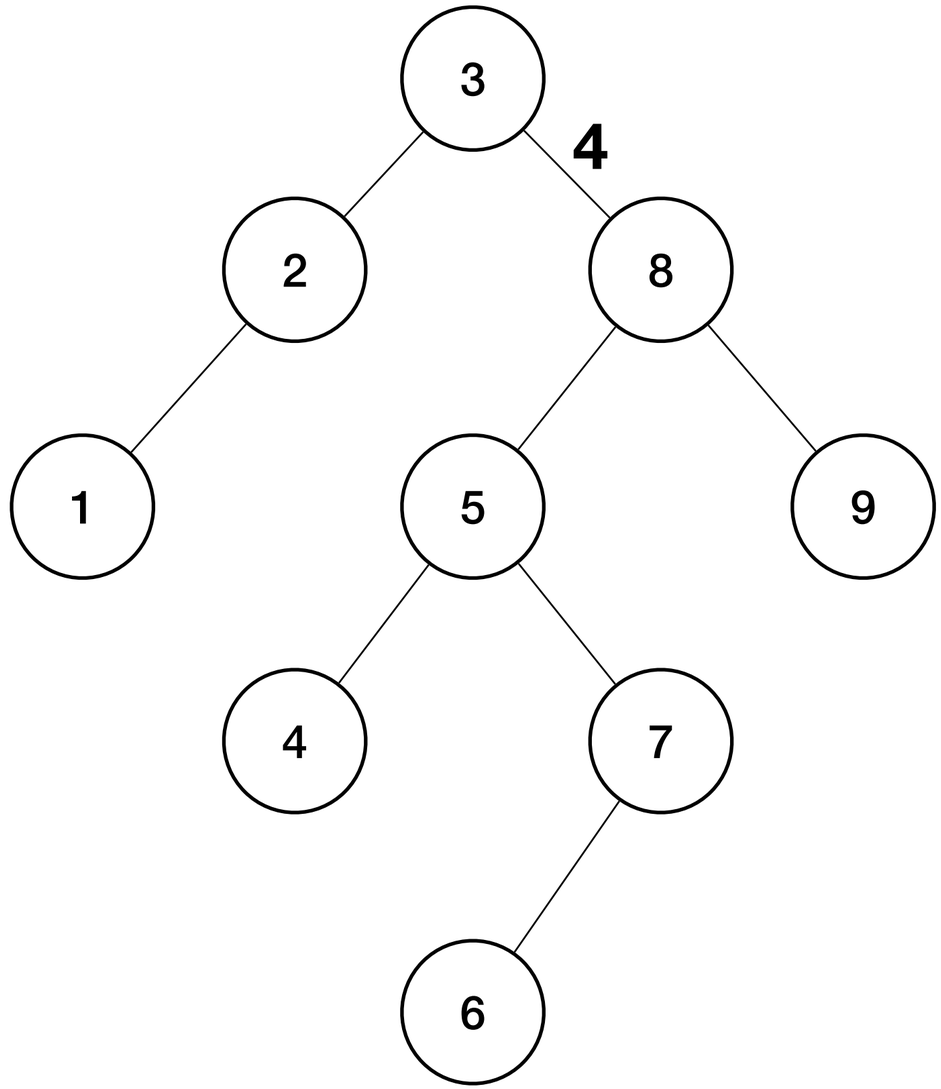


Finding height of a tree

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

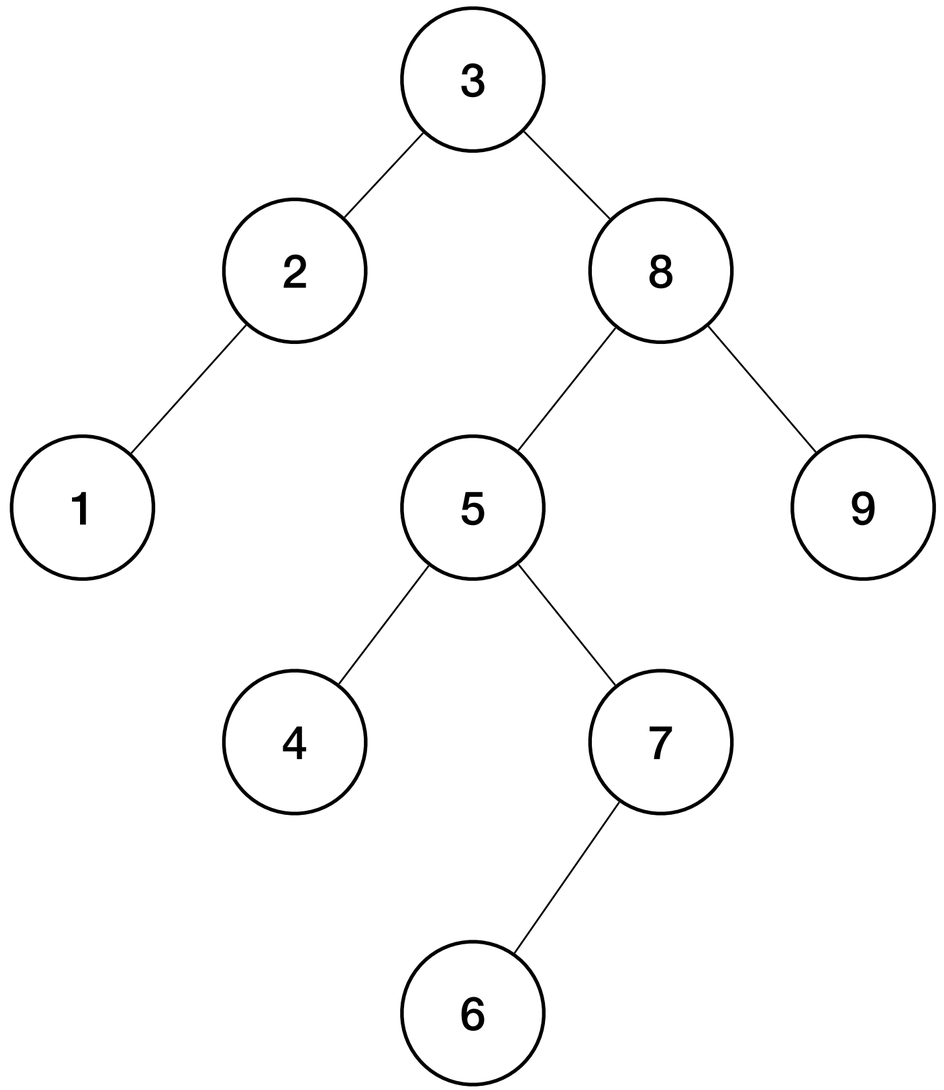
- Recursively calculate: 1 + height of L/R subtree(s)

- Take maximum at each step



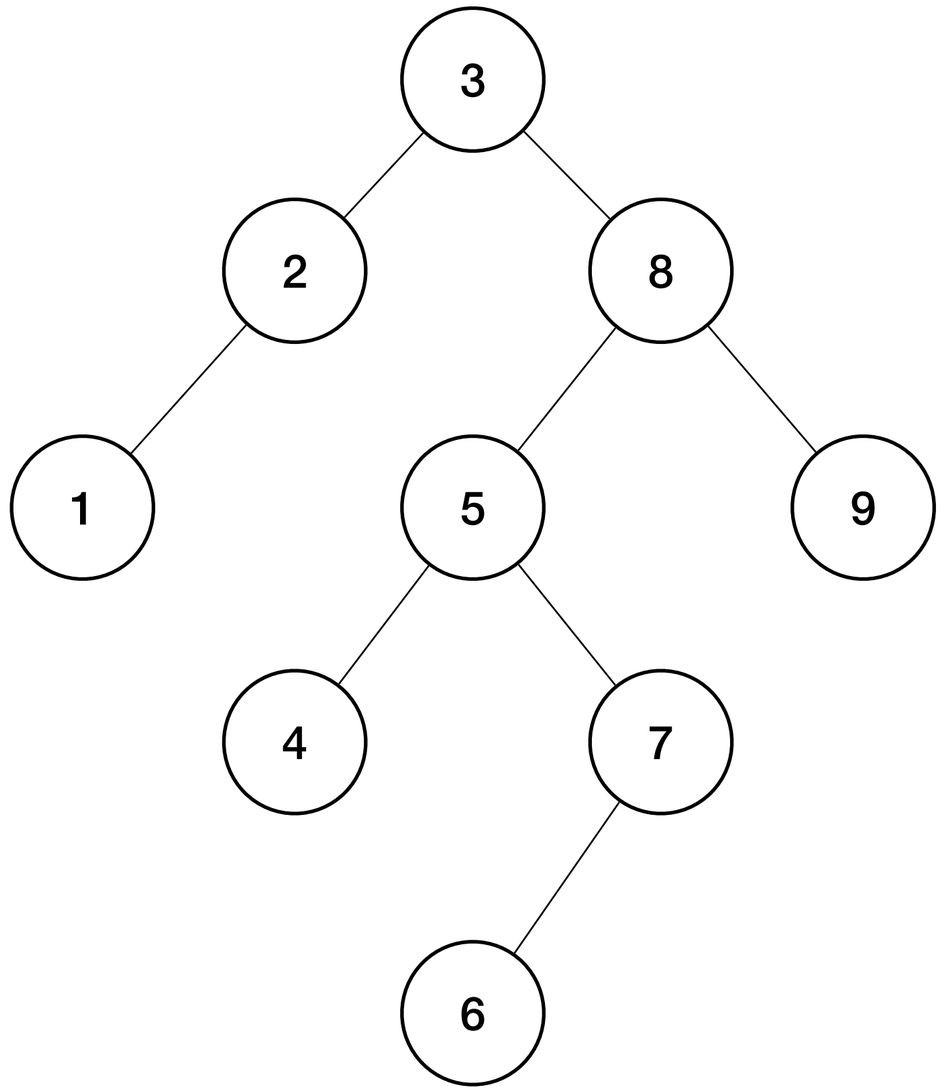
Find height of a tree

```
int tree_height(node *cursor){
  int lh, rh;
  if (cursor==NULL)

  else{
    lh =
    rh =
    return
  }
}
```



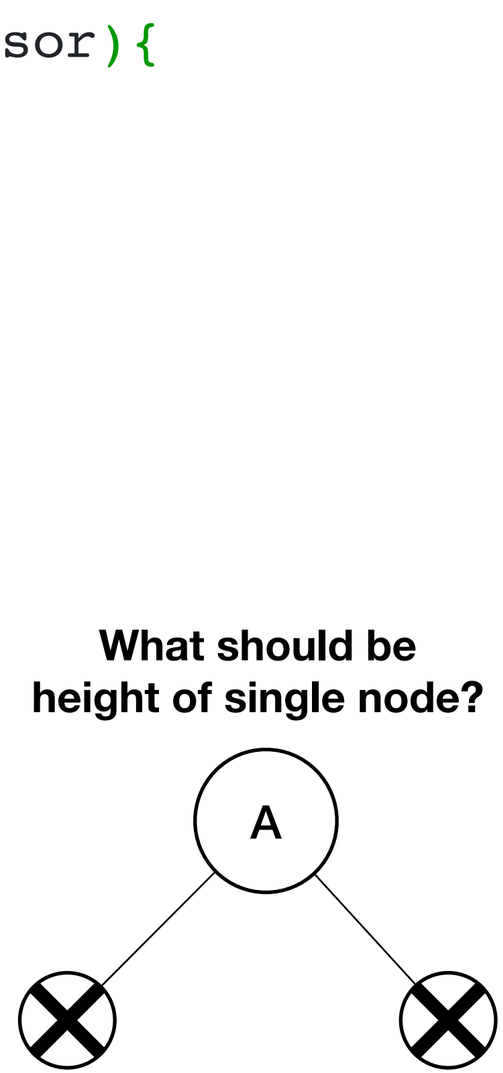
Find height of a tree

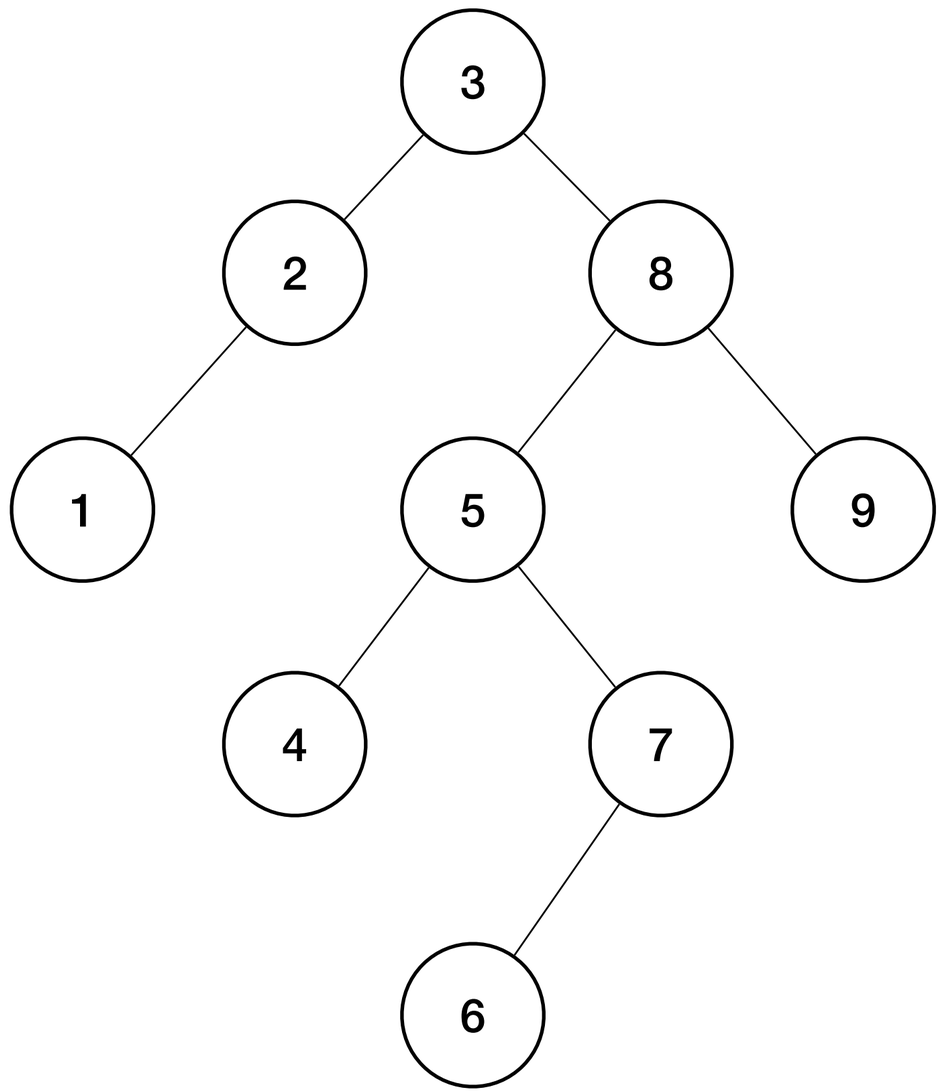
```
int tree_height(node *cursor){
  int lh, rh;
  if (cursor==NULL)
    return -1;
  else{
    lh =
    rh =
    return
  }
}
```



Find height of a tree

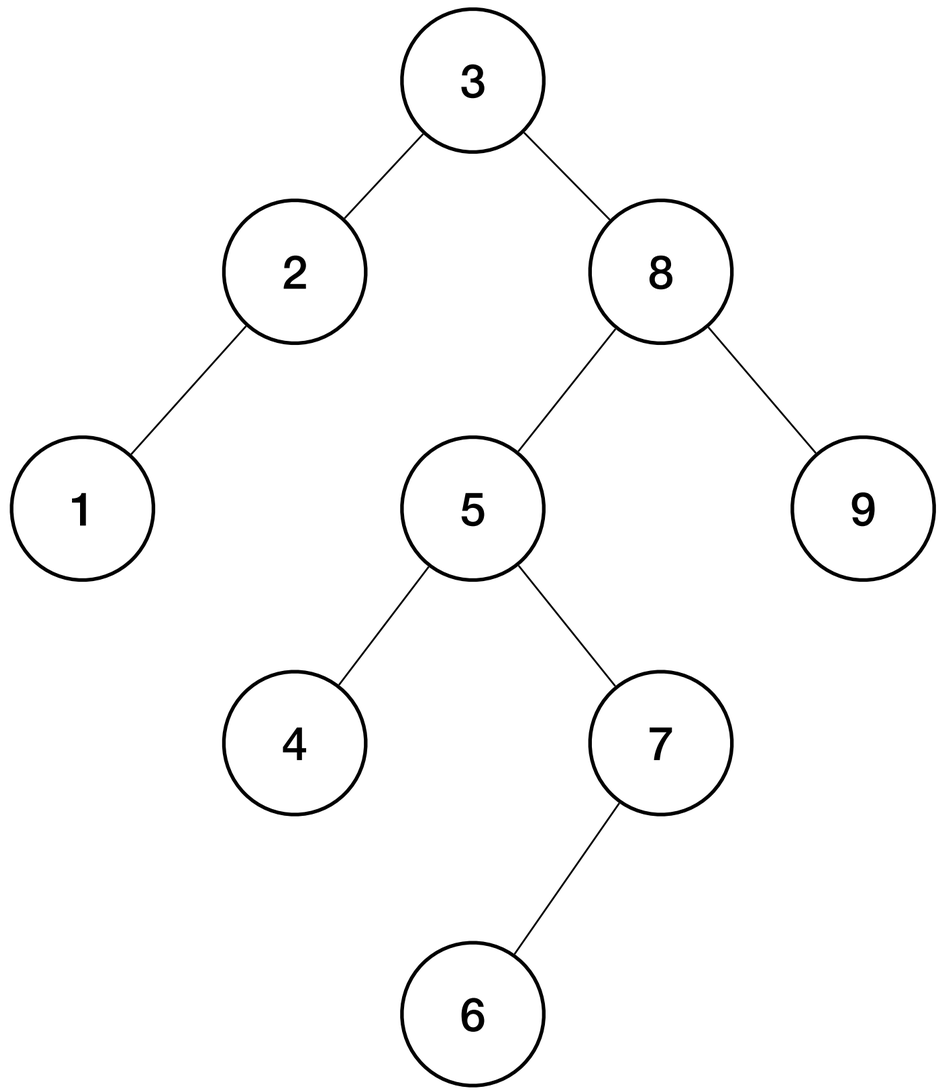
```
int tree_height(node *cursor){
  int lh, rh;
  if (cursor==NULL)
    return -1;
  else{
    lh =
    rh =
    return
  }
}
```



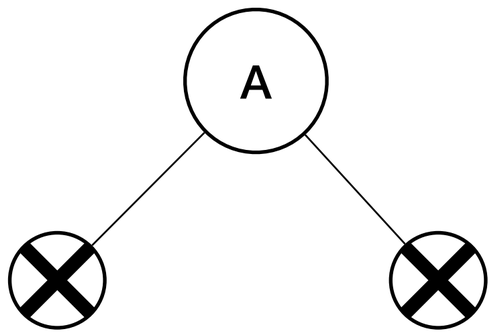


Find height of a tree

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



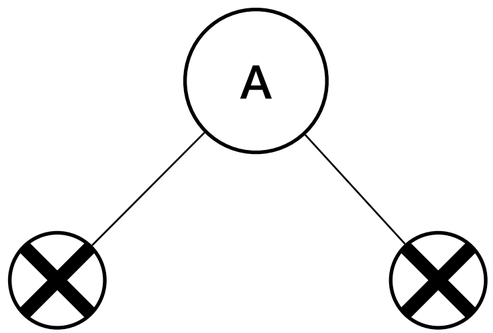
What should be height of single node?

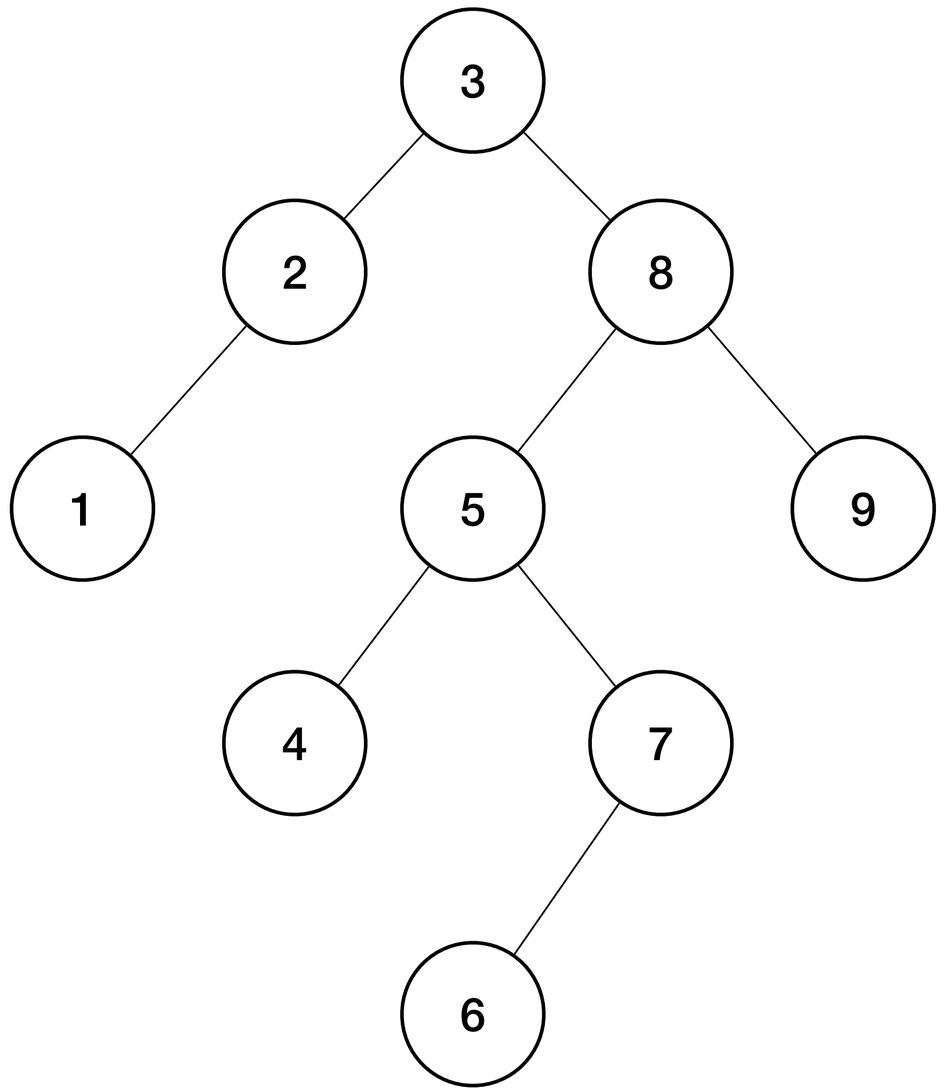


Find height of a tree

```
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
  }
}
```

What should be height of single node?

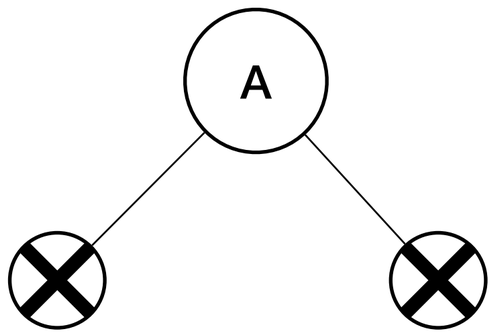


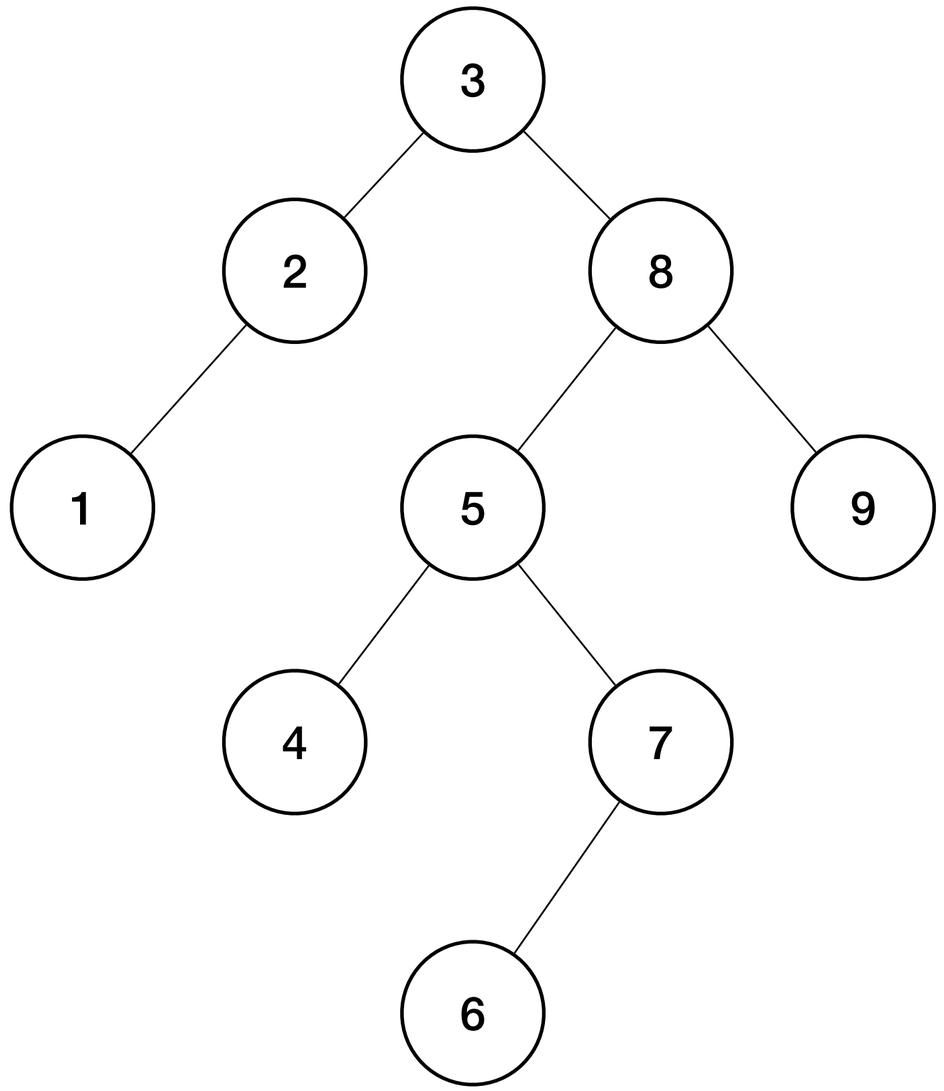


Find height of a tree

```
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;
};
```

C++ Example

- Using classes in C++, create a BST class and perform or find:

```
template <class N>
class bst{
private:
  …
  …
  …
public:
  bst();
```

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

C++ Example

- Using classes in C++, create a BST class and perform or find:
  - Insertion

```
template <class N>
class bst{
private:
  …
  …
  …
public:
  bst();
  void insert(N data);
```

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

C++ Example

- Using classes in C++, create a BST class and perform or find:
  - Insertion
  - Searching

```
template <class N>
class bst{
private:
  …
  …
  …
public:
  bst();
  void insert(N data);
  treenode<N> *search(N data);
```

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

C++ Example

- Using classes in C++, create a BST class and perform or find:
  - Insertion
  - Searching
  - Traversal

```
template <class N>
class bst{
private:
  …
  …
  …
public:
  bst();
  void insert(N data);
  treenode<N> *search(N data);
  void inorder();
```

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

C++ Example

- Using classes in C++, create a BST class and perform or find:
  - Insertion
  - Searching
  - Traversal
  - Vectorization

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

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

C++ Example

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

```
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();
```

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

C++ Example

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

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

C++ Example

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

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

C++ Example

- Using classes in C++, create a 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();
};
```

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

C++ Example

- Using classes in C++, create a BST class and perform or find:

```
template <class N>
class bst{
private:
  typedef treenode<N> node;
  node *root;
```

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

C++ Example

- Using classes in C++, create a BST class and perform or find:
  - Insertion

```
template <class N>
class bst{
private:
  typedef treenode<N> node;
  node *root;

  void insert(N data, node **cursor);
```

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

C++ Example

- Using classes in C++, create a BST class and perform or find:
  - Insertion
  - Searching

```
template <class N>
class bst{
private:
  typedef treenode<N> node;
  node *root;
```

```
void insert(N data, node **cursor);
node *search(N key, node *cursor);
```

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

C++ Example

- Using classes in C++, create a BST class and perform or find:
  - Insertion
  - Searching
- Traversal

```
template <class N>
class bst{
private:
  typedef treenode<N> node;
  node *root;
```

```
void insert(N data, node **cursor);
node *search(N key, node *cursor);
void inorder(node *cursor);
```

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

C++ Example

- Using classes in C++, create a BST class and perform or find:
  - Insertion
  - Searching
  - Traversal
  - Vectorization

```
template <class N>
class bst{
private:
  typedef treenode<N> node;
  node *root;
```

```
void insert(N data, node **cursor);
node *search(N key, node *cursor);
void inorder(node *cursor);
vector<N> vectorize(node *cursor, vector<N> &v);
```

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

C++ Example

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

```
template <class N>
class bst{
private:
  typedef treenode<N> node;
  node *root;
void insert(N data, node **cursor);
node *search(N key, node *cursor);
void inorder(node *cursor);
vector<N> vectorize(node *cursor, vector<N> &v);
int countnodes(node *cursor);
void print(node *cursor, int depth);
```

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

C++ Example

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

```
template <class N>
class bst{
private:
  typedef treenode<N> node;
  node *root;

void insert(N data, node **cursor);
node *search(N key, node *cursor);
void inorder(node *cursor);
vector<N> vectorize(node *cursor, vector<N> &v);
int countnodes(node *cursor);
void print(node *cursor, int depth);
int height(node *cursor);
```

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

C++ Example

- Using classes in C++, create a 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:
  typedef treenode<N> node;
  node *root;

void insert(N data, node **cursor);
node *search(N key, node *cursor);
void inorder(node *cursor);
vector<N> vectorize(node *cursor, vector<N> &v);
int countnodes(node *cursor);
void print(node *cursor, int depth);
int height(node *cursor);
void delete_tree(node *cursor);
```

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

C++ Example

- Using classes in C++, create a 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:
  typedef treenode<N> node;
  node *root;

void insert(N data, node **cursor);
node *search(N key, node *cursor);
void inorder(node *cursor);
vector<N> vectorize(node *cursor, vector<N> &v);
int countnodes(node *cursor);
void print(node *cursor, int depth);
int height(node *cursor);
void delete_tree(node *cursor);
```

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

C++ Example

- Using classes in C++, create a 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:
  typedef treenode<N> node;
  node *root;
  void insert(N data, node **cursor);
  node *search(N key, node *cursor);
  void inorder(node *cursor);
  vector<N> vectorize(node *cursor, vector<N> &v);
  int countnodes(node *cursor);
  void print(node *cursor, int depth);
  int height(node *cursor);
  void delete_tree(node *cursor);

public:
```

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

C++ Example

- Using classes in C++, create a 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:
  typedef treenode<N> node;
  node *root;
  void insert(N data, node **cursor);
  node *search(N key, node *cursor);
  void inorder(node *cursor);
  vector<N> vectorize(node *cursor, vector<N> &v);
  int countnodes(node *cursor);
  void print(node *cursor, int depth);
  int height(node *cursor);
  void delete_tree(node *cursor);

public:
  …
```

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

C++ Example

- Using classes in C++, create a 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:
  typedef treenode<N> node;
  node *root;
  void insert(N data, node **cursor);
  node *search(N key, node *cursor);
  void inorder(node *cursor);
  vector<N> vectorize(node *cursor, vector<N> &v);
  int countnodes(node *cursor);
  void print(node *cursor, int depth);
  int height(node *cursor);
  void delete_tree(node *cursor);

public:
  …
  …
```

C++ Example

C++ Example

```cpp
#include <iostream>
#include "bst.hpp"

using namespace std;

int main(){
  bst <int> tree1;
  cout<<"Building a Binary Search Tree"<<endl;

  tree1.insert(45);
  tree1.insert(50);
  tree1.insert(35);
  tree1.insert(30);
  tree1.insert(70);
  tree1.insert(20);
  tree1.insert(40);
  tree1.insert(80);
  tree1.insert(60);

  cout<<"Total number of nodes in this tree: ";
  cout<<tree1.node_count()<<endl;
  tree1.inorder();
  cout<<endl;
  tree1.print();
  cout<<"The tree height is: "<<tree1.height();
  cout<<endl;

  vector <int> v = tree1.vectorize();
  cout<<"Vectorized in order this is:"<<endl;
  for (auto it= v.begin(); it != v.end(); ++it)
     cout<<*it<<", ";
  return 0;
  }
```

Some food for thought

Some food for thought

- Can you write a function to determine if a binary tree is a valid BST?

Some food for thought

- Can you write a function to determine if a binary tree is a valid BST?

- Can you reconstruct a BST from its vectorized version?

Some food for thought

- Can you write a function to determine if a binary tree is a valid BST?

- Can you reconstruct a BST from its vectorized version?

  - *Traversals matter*: pre-order, in-order, post-order

Inorder: 1,2,3,4,5,6,

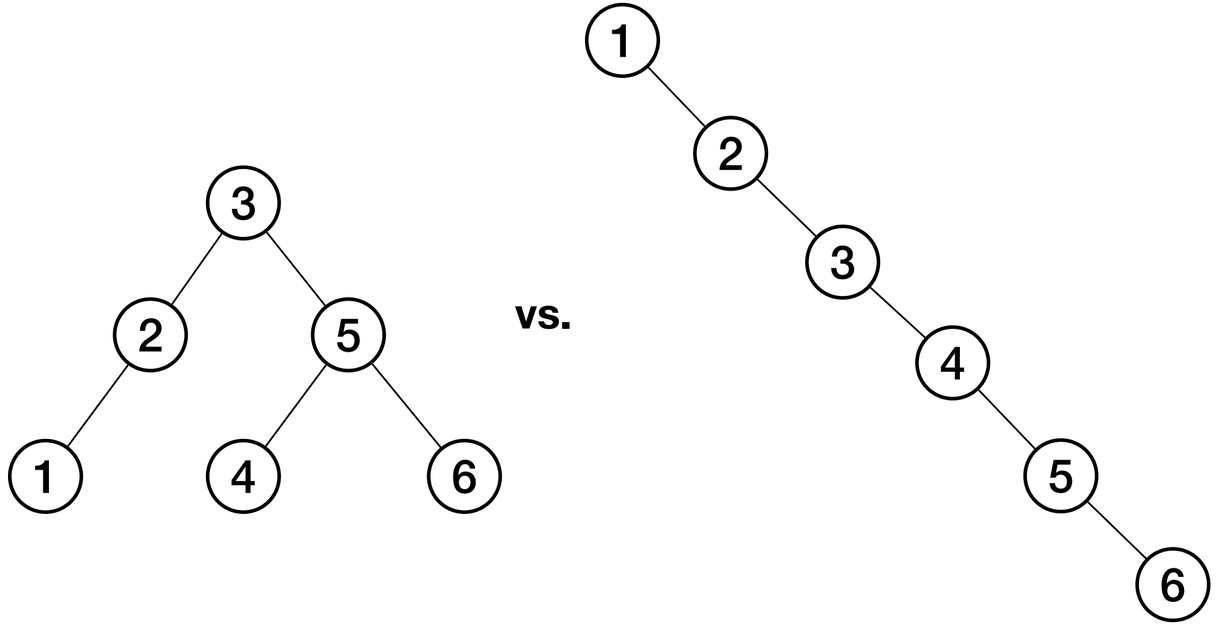
Some food for thought

- Can you write a function to determine if a binary tree is a valid BST?

- Can you reconstruct a BST from its vectorized version?

  - *Traversals matter*: pre-order, in-order, post-order

Inorder: 1,2,3,4,5,6,



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

Some food for thought

- Can you write a function to determine if a binary tree is a valid BST?

- Can you reconstruct a BST from its vectorized version?

  - *Traversals matter*: pre-order, in-order, post-order

- How to modify struct definition if tree is no longer binary?

Inorder: 1,2,3,4,5,6,