Lecture 7
Binary search trees

EECS-214

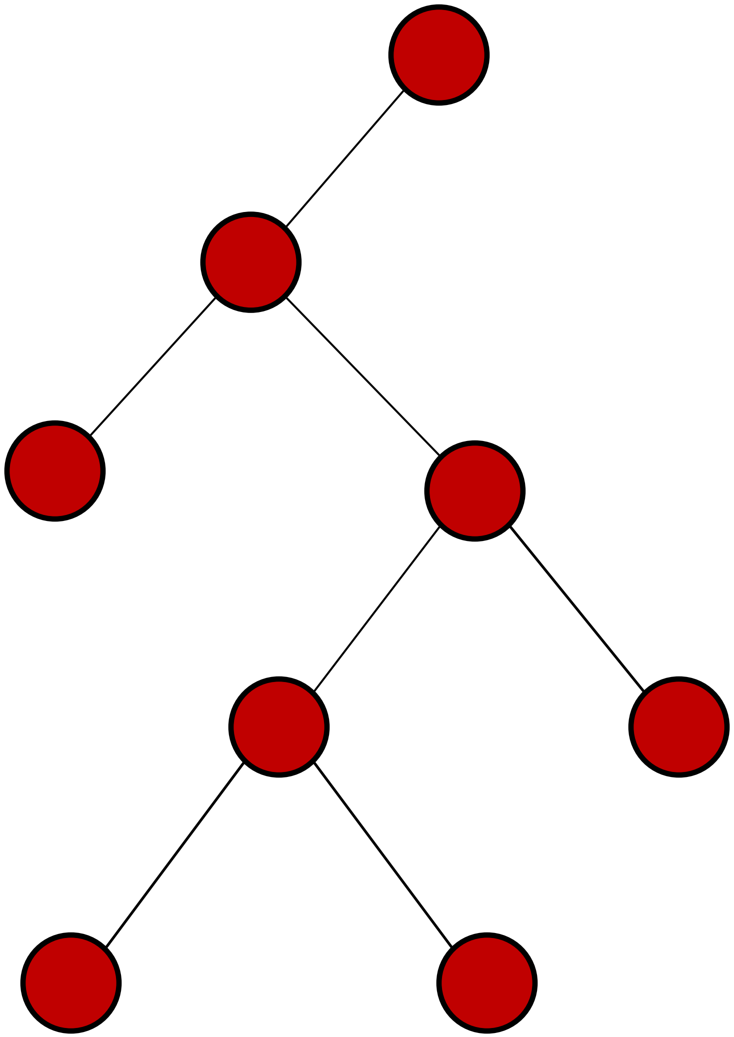
Representing collections of objects

We've been looking at **collection classes**

- **Store** a bunch of objects
- Different **flavors** support
  - Different kinds of **operations**
  - With different **performance** trade-offs
- Notice these are all **bad at searching**
  - All take
  - Or don't support it al all
- Can we do **better**?

- Dyanmic arrays
  - Get element at position: $O(1)$
  - Add and remove: $O(n)$
    - $O(1)$ amortized time if implemented with doubling
  - Search for an element: $O(n)$
- Linked lists
  - Get element at position: $O(n)$
  - Add and remove from beginning: $O(1)$
  - Add and remove from position specified by index: $O(n)$
  - Search for an element: $O(n)$
- Stacks and queues
  - Add/remove element: $O(1)$
    - If implemented with array and array can be expanded dynamically, then $O(1)$ amortized time (see lecture 17)
  - No other operations supported

Binary trees

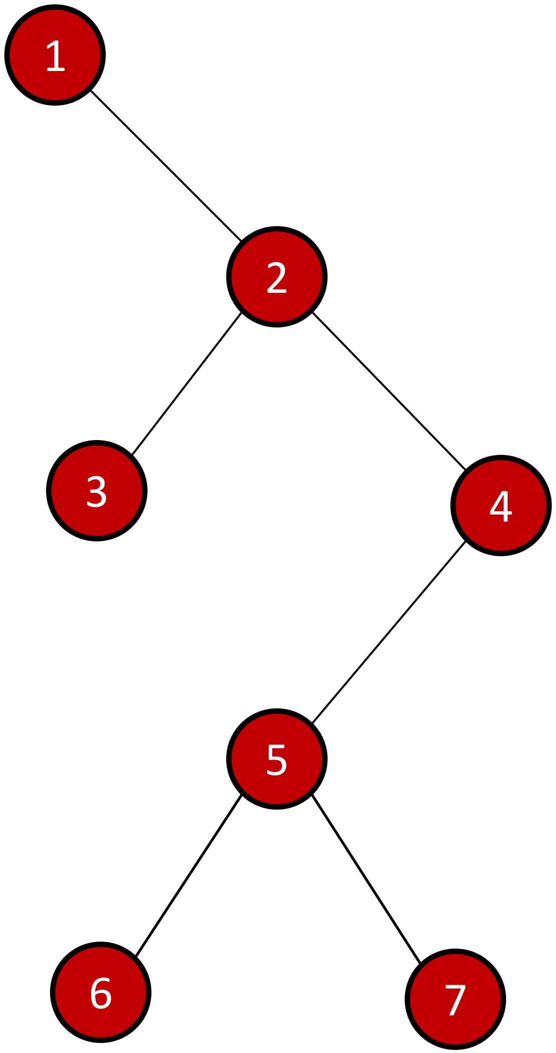


- Common case
- **Fixed branching factor** of 2
  - Every node has at most 2 children
  - Referred to as the **left** child and **right** child

Inorder traversal

```
Inorder(node) {
  Inorder(node.leftChild)
  print node
  Inorder(node.rightChild)
}
```

Output:

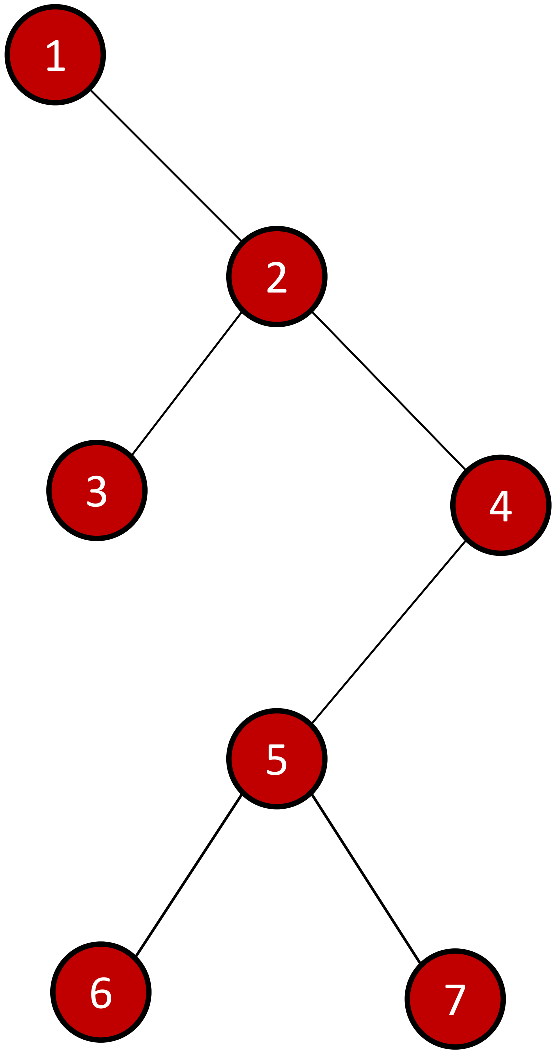


Inorder traversal

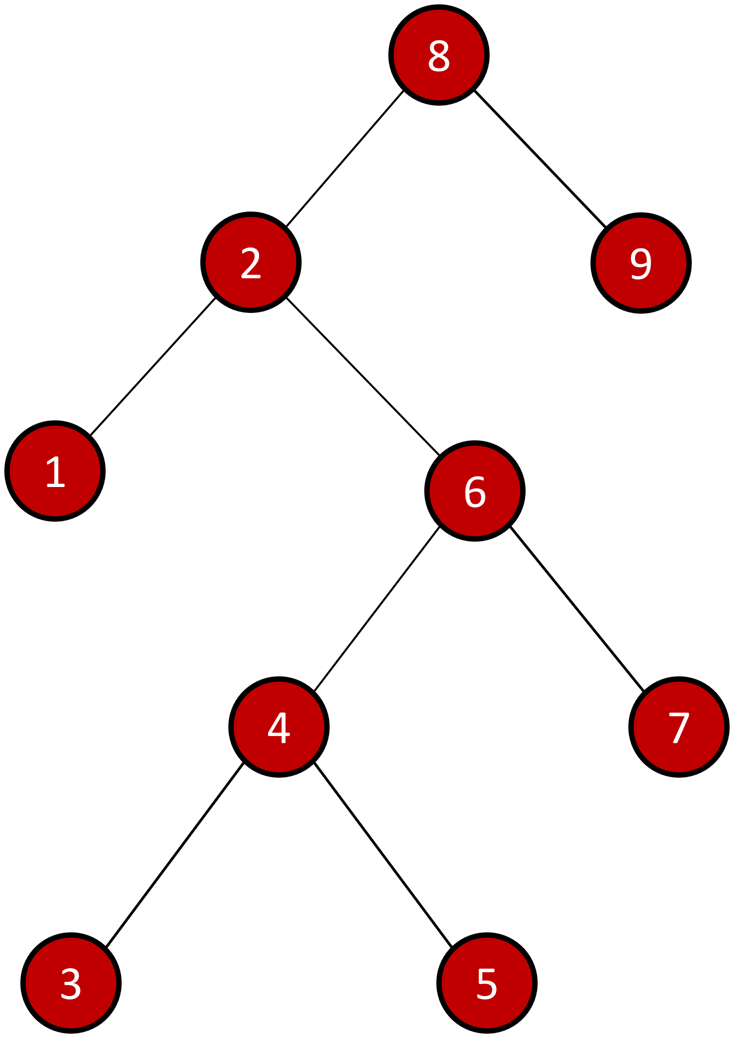
```
Inorder(node) {
  Inorder(node.leftChild)
  print node
  Inorder(node.rightChild)
}
```

Output:

1 3 2 6 5 7 4

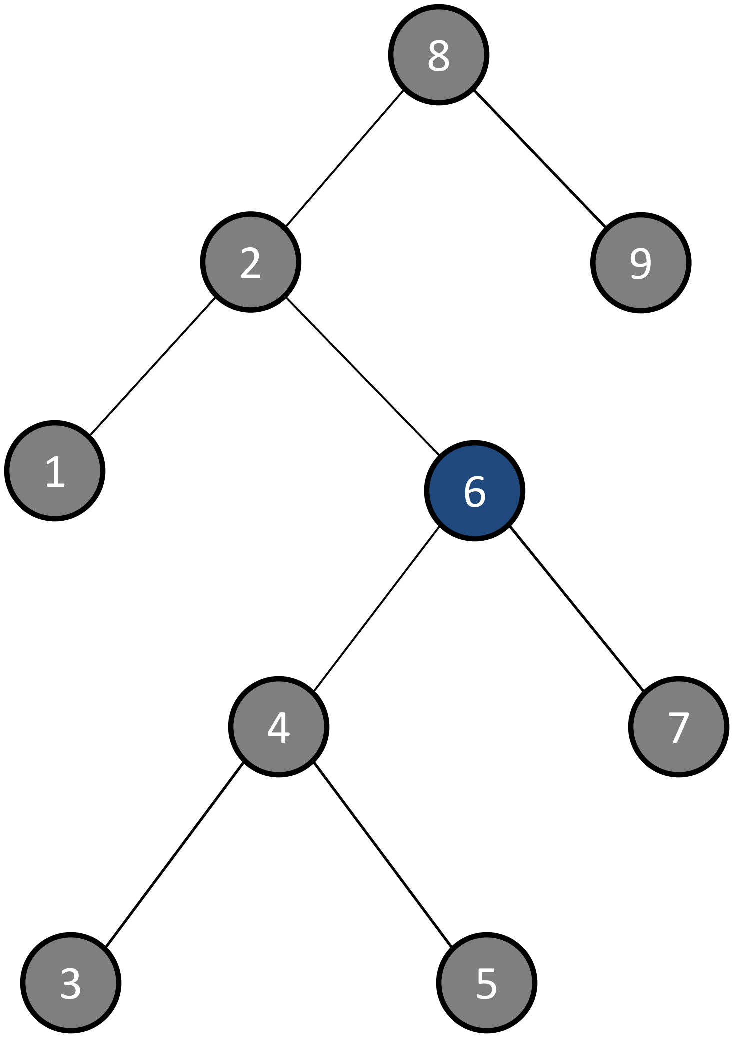


Binary **search** trees



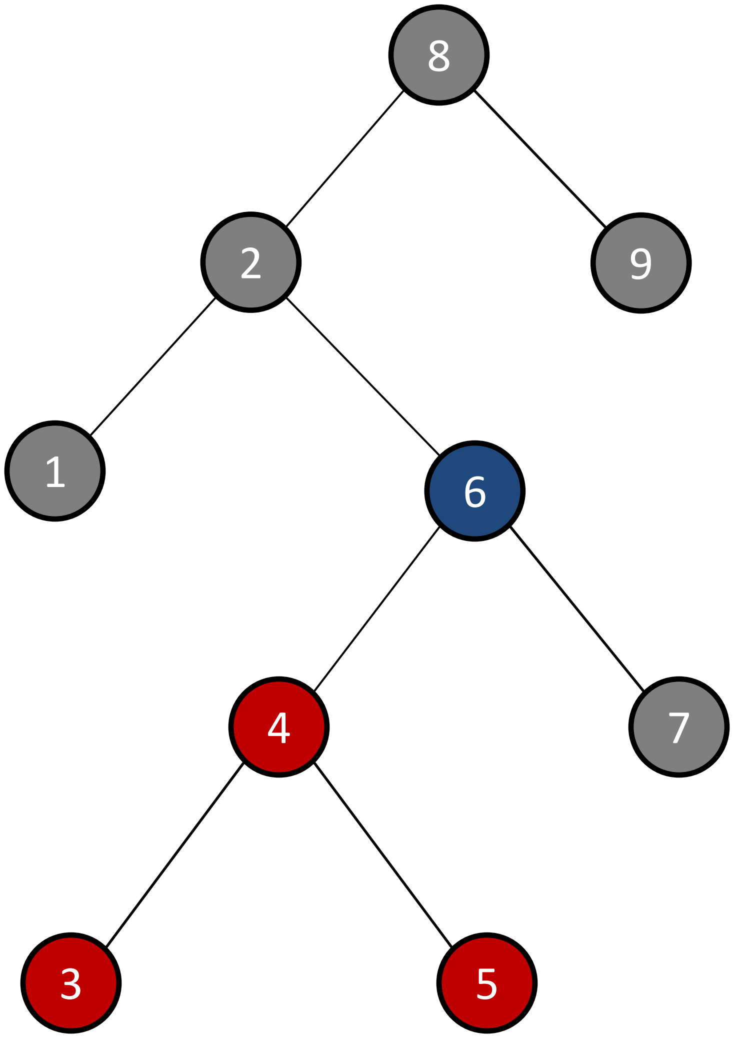
- Binary tree
- Each node **labeled** with a value
  - Number, string, or some other set that has a total order on it
- Has the magic **binary search tree property**

Binary **search** trees



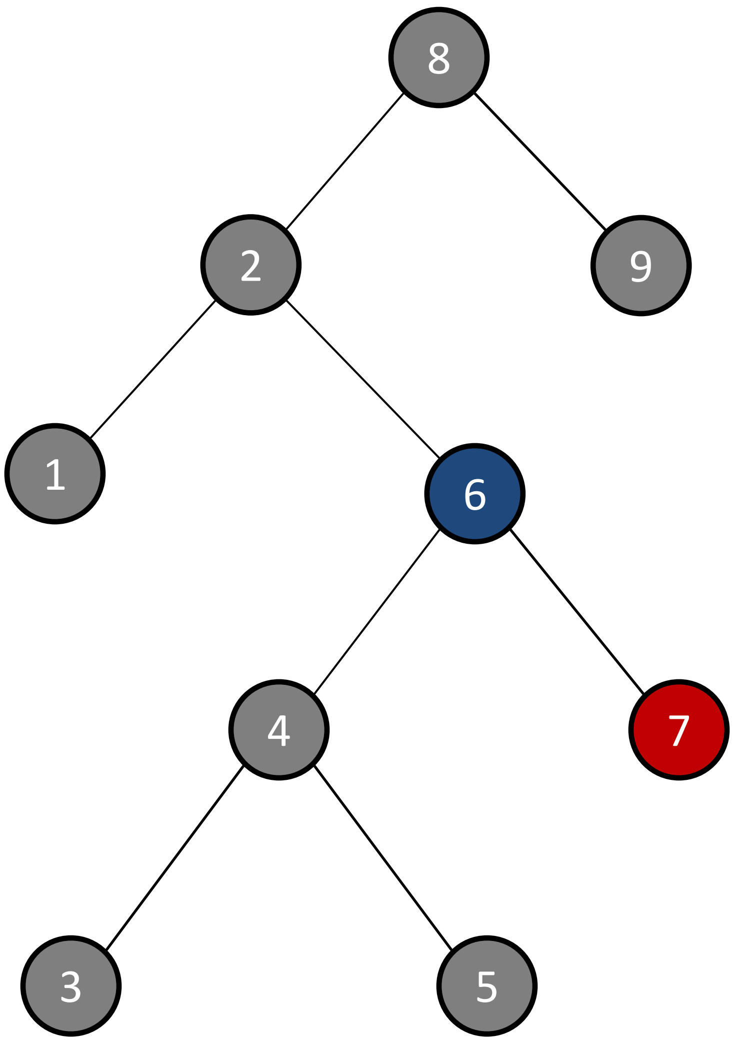
- Binary tree
- Each node **labeled** with a value
  - Number, string, or some other set that has a total order on it
- Has the magic **binary search tree property**
  - For any node

Binary **search** trees



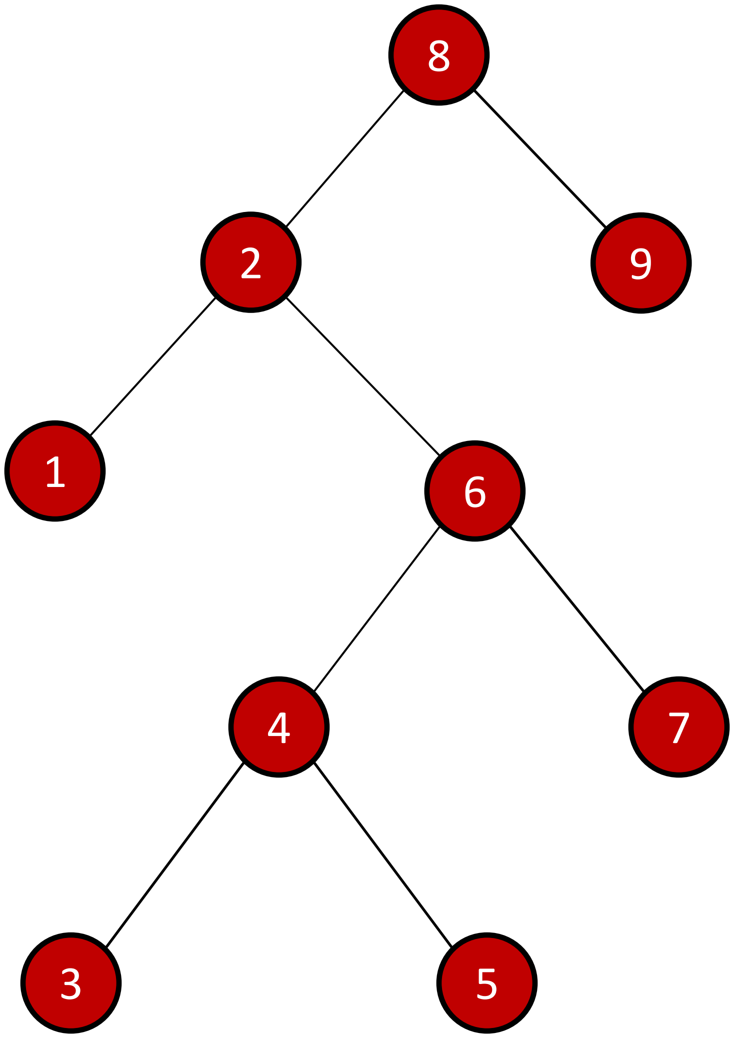
- Binary tree
- Each node **labeled** with a value
  - Number, string, or some other set that has a total order on it
- Has the magic **binary search tree property**
  - For any node
    - All the nodes in the left subtree have labels $\leq$ to its label

Binary **search** trees



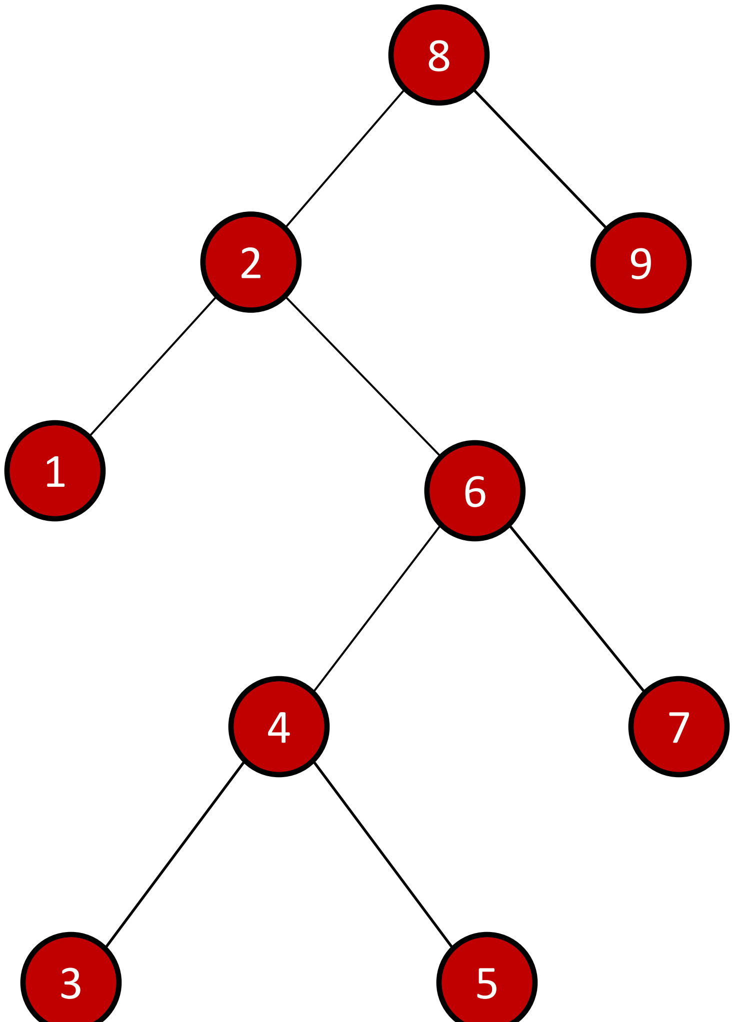
- Binary tree
- Each node **labeled** with a value
  - Number, string, or some other set that has a total order on it
- Has the magic **binary search tree property**
  - For any node
    - All the nodes in the left subtree have labels $\leq$ to its label
    - All the nodes in the right subtree have labels $\geq$ to its label

Proposition



- An **in-order** traversal of a binary search tree prints the nodes in **sorted order**

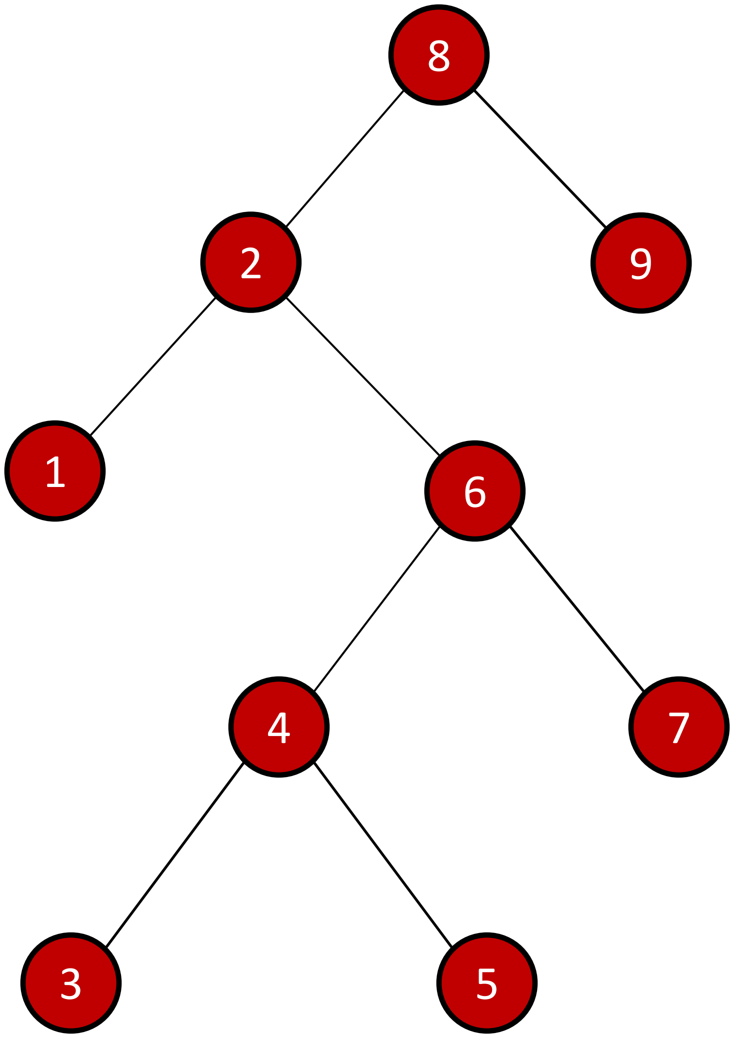
Proposition



Proof (by **induction**)

- True for trees of depth 1
- Assume it's true for trees of depth n (for some n)
- Consider a tree of depth n+1
  - A traversal of the tree will print:
    - An in-order traversal of the left subtree
    - The root
    - An in-order traversal of the right subtree
  - By assumption, the subtrees are printed in sorted order
  - And the root is
    - $\geq$ anything on the left
    - $\leq$ any on the right
  - So the whole thing is sorted
- So the proposition holds for a tree of any depth

Representing binary search trees



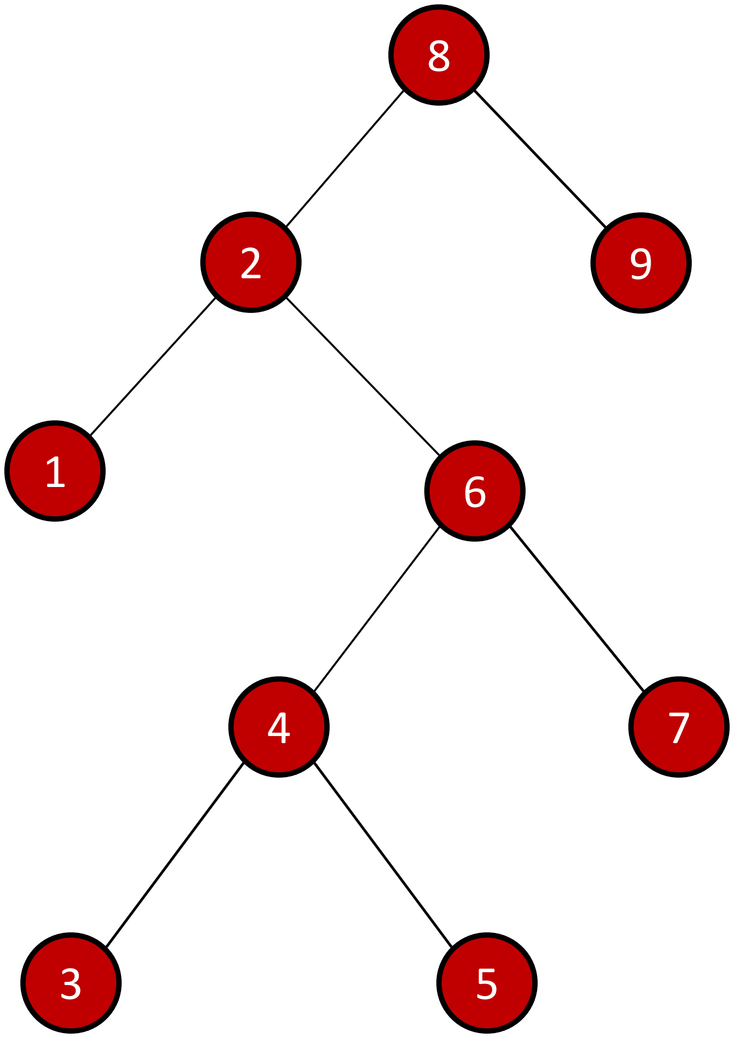

```
class BST {
  int key;
  BST parent;
  BST left;
  BST right;

  public BST(int k) {
    key = k;
    // other fields null
  }
}
```

tree search

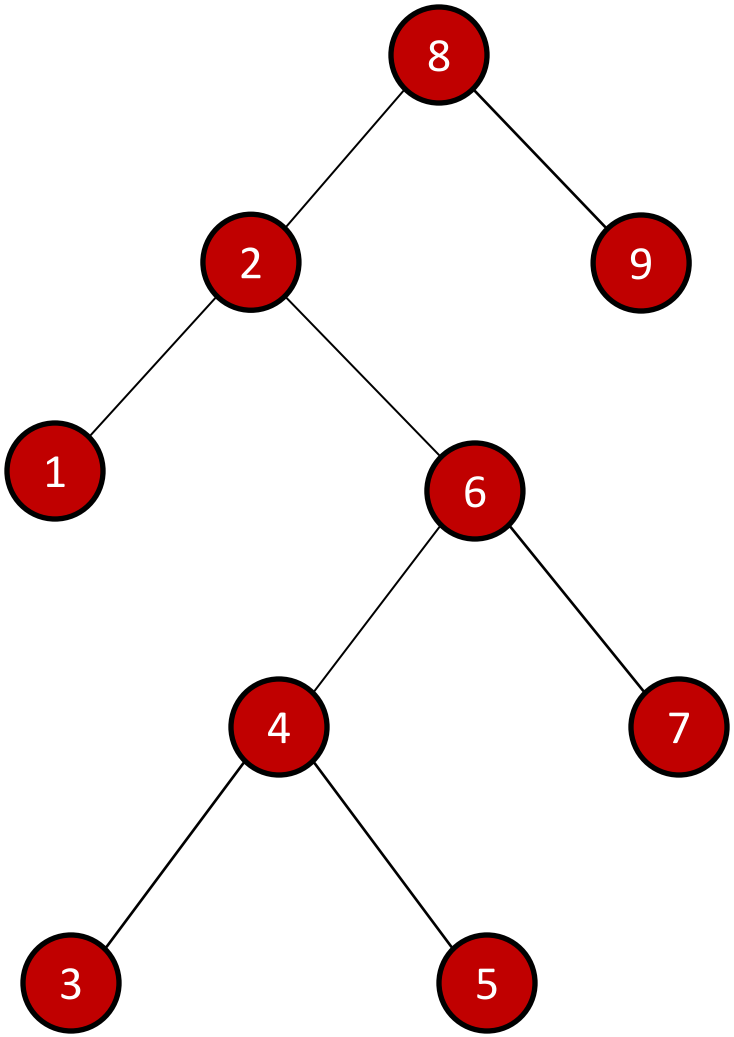
Searching a binary search tree

(pseudocode)




```
// Search tree starting at node
// Return node containing key k
// Or null if k missing from tree
Search(node, int k) {
    if (node == null)
        // Failure: not in tree
        return null;
    else if (k == node.key)
        // Success: found it
        return node;
    else if (k < node.key)
        return Search(node.left, k);
    else
        return Search(node.right, k);
}
```

111 students:
What do we call this kind of recursion?




```
Search(node, int k) {
  if (node == null)
    // Failure: not in tree
    return null;
  else if (k == node.key)
    // Success: found it
    return node;
  else if (k < node.key)
    return Search(node.left, k);
  else
    return Search(node.right, k);
}
```

Tail recursion

- Tail recursions are where all the recursive calls are of the form “**return Search(…)**”
  - All the procedure will do when it gets the result
  - Is **forward it on** to its caller
- Tail-call optimization
  - Don’t bother making a new stack frame for the new call
  - Just **reuse the existing stack frame**
  - And **jump back** to the beginning of the procedure

```
Search(node, int k) {
  if (node == null)
    // Failure: not in tree
    return null;
  else if (k == node.key)
    // Success: found it
    return node;
  else if (k < node.key)
    return Search(node.left, k);
  else
    return Search(node.right, k);
}
```

Tail recursion

- Tail recursions are where all the recursive calls are of the form “return Search()”
  - All the procedure will do when it gets the result
  - Is forward it on to its caller

- Tail-call optimization
  - Don’t bother making a new stack frame for the new call
  - Just reuse the existing stack frame
  - And jump back to the beginning of the procedure

```
Search(node, int k) {
   start: if (node == null)
      // Failure: not in tree
      return null;
   else if (k == node.key)
      // Success: found it
      return node;
   else if (k < node.key)
   { node = node.left; goto start; }
   else
   { node = node.right; goto start; }
}
```

Tail recursion

- But of course, goto is evil
  - Makes code hard to read
  - Hard to maintain
  - Your coworkers will put not be pleased with you

```
Search(node, int k) {
  start: if (node == null)
    // Failure: not in tree
    return null;
  else if (k == node.key)
    // Success: found it
    return node;
  else if (k < node.key)
  { node = node.left; goto start; }
  else
  { node = node.right; goto start; }
}
```

Tail recursion

- But of course, goto is evil
  - Makes code hard to read
  - Hard to maintain
  - Your coworkers will put not be pleased with you
- So we just rewrite it as a while loop
- This is why 111 called tail recursions “iterations”
  - They’re the set of iterations that can be rewritten as while loops
  - Good C compilers, like gcc do this automatically

```
Search(node, int k) {
  while (node != null) {
    if (k == node.key)
      // Success: found it
      return node;
    else if (k < node.key)
      node = node.left;
    else
      node = node.right;
  }
  // Failure: not in tree
  return null;
}
```

Tail recursion

- We can even simplify it a little more

```
Search(node, int k) {
  while (node != null
          && node.key != k)
      if (k < node.key)
        node = node.left;
      else
        node = node.right;
  return node;
}
```

performance

Analysis

- How do we analyze the **running time** of this algorithm?

```
Search(node, int k) {
  while (node != null
          && node.key != k)
    if (k < node.key)
      node = node.left;
    else
      node = node.right;
  return node;
}
```

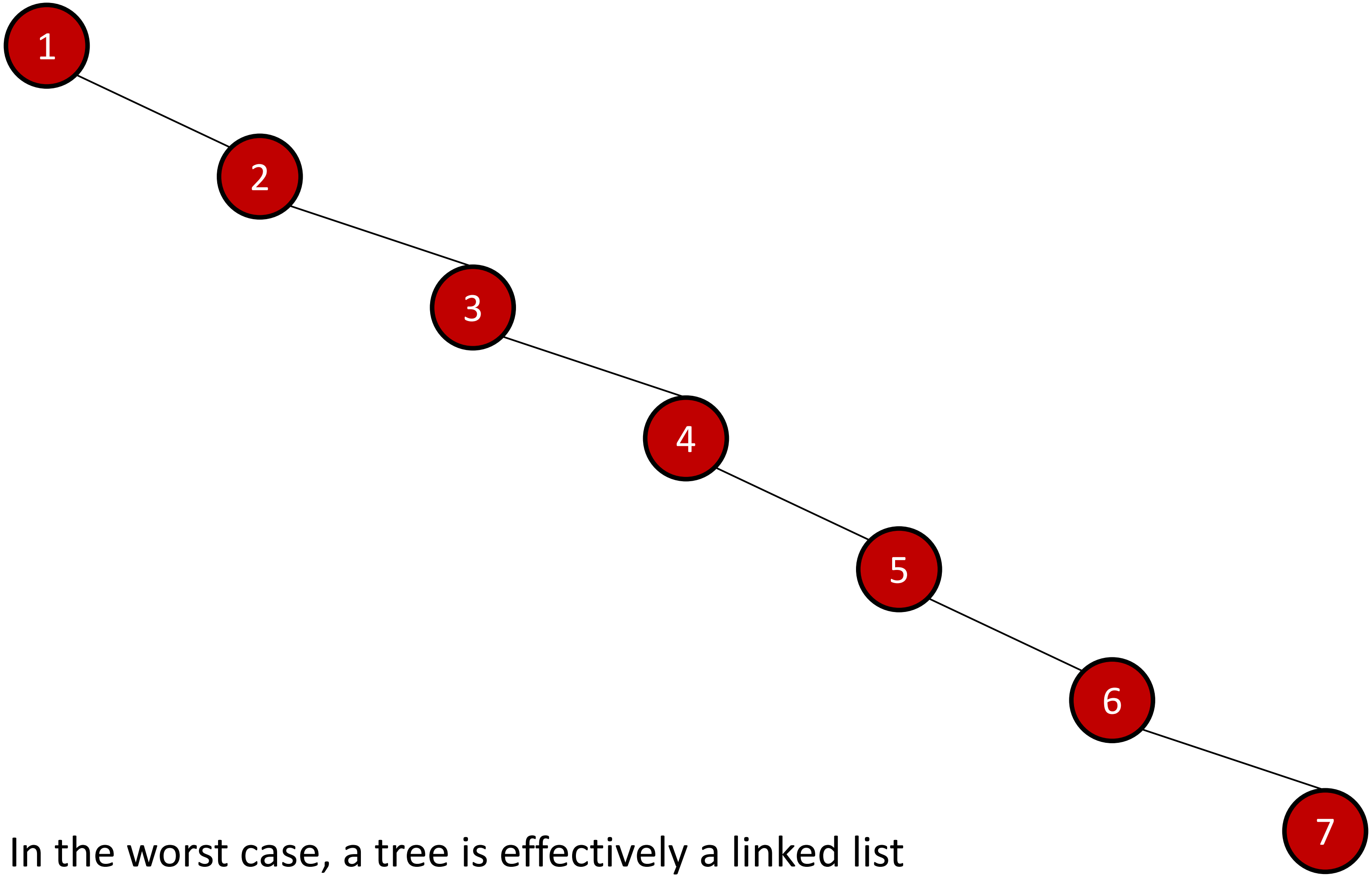
Analysis

- How do we analyze the running time of this algorithm?
  - Each iteration replaces node with one of its children
  - So on each iteration, the **depth** of node **increases by 1**
  - But the depth is **bounded by the height** of the tree (number of levels in the tree)
  - So the loop can’t run for more iterations than the height

- So the running time is $\boldsymbol{O(h)}$ where h is the height of the tree

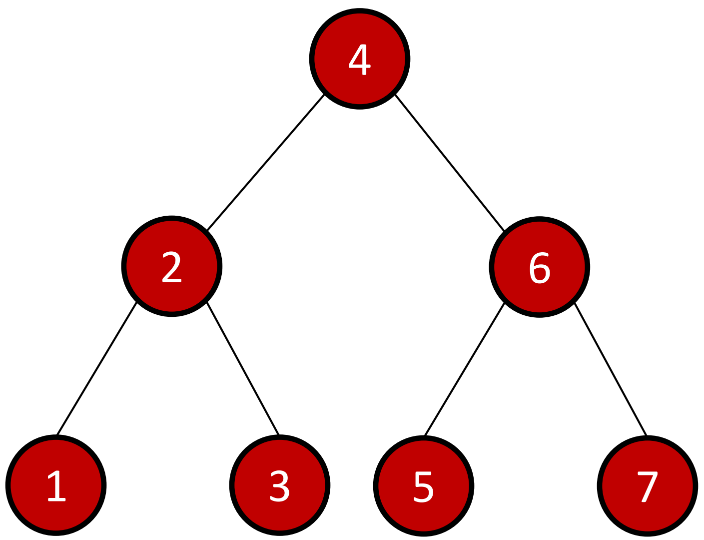
```
Search(node, int k) {
  while (node != null
          && node.key != k)
     if (k < node.key)
        node = node.left;
     else
        node = node.right;
  return node;
}
```

A bad tree to search



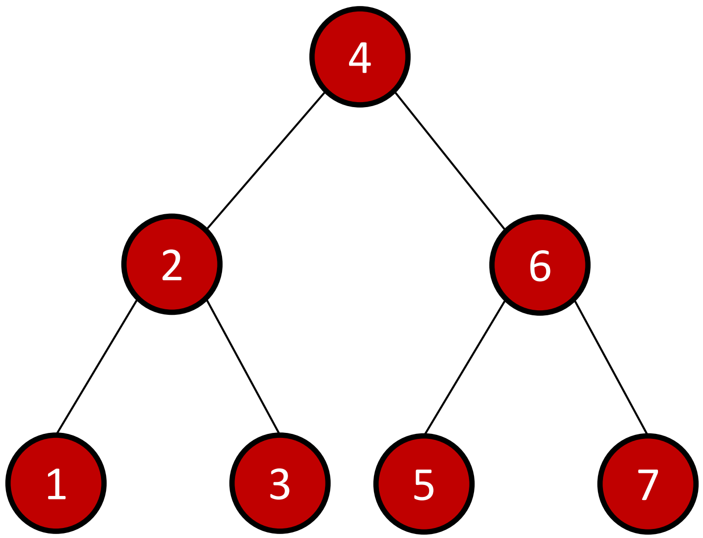
In the worst case, a tree is effectively a linked list

A good tree to search



- Informally, a **balanced** binary tree is one where the different branches are the approximately **same depth**
- A balanced search tree has a **small height** for its given number of nodes
  - $O(\log n)$
- So tree search **runs fast** if the tree is balanced

A good tree to search



- The most balanced tree is a **complete** binary tree
  - All leaves at same depth
  - All other nodes have both left and right children
- We'll talk next time about strategies for automatically balancing trees

other operations

Special-case searches

```
// Get the node holding
// the minimum value
// in the tree
Minimum(n) {
  while (n.left != null)
    n = n.left;
  return n;
}
```

```
// Get the node holding
// the maximum value
// in the tree
Maximum(n) {
  while (n.right != null)
    n = n.right;
  return n;
}
```

Successor

- Sometimes you want to find **the node that would come after this one** in an in-order traversal
- Simulate what an in-order walk would do at this point

```
Successor(n) {
  if (n.right != null)
    return Minimum(n.right);

  p = n.parent;
  while (p != null
            && n == p.right) {
    n = p;
    p = n.parent;
  }
  return p;
}
```

Successor

Two cases:

1. If **we have a right child**
   - In-order would go to the right child
   - Which would go to its left child
   - Which would go to its left child
   - Etc., until we get to a leaf
   - We can do all the with Minimum

```
Successor(n) {
  if (n.right != null)
    return Minimum(n.right);

  p = n.parent;
  while (p != null
         && n = p.right) {
    n = p;
    p = n.parent;
  }
  return p;
}
```

Successor

Two cases:

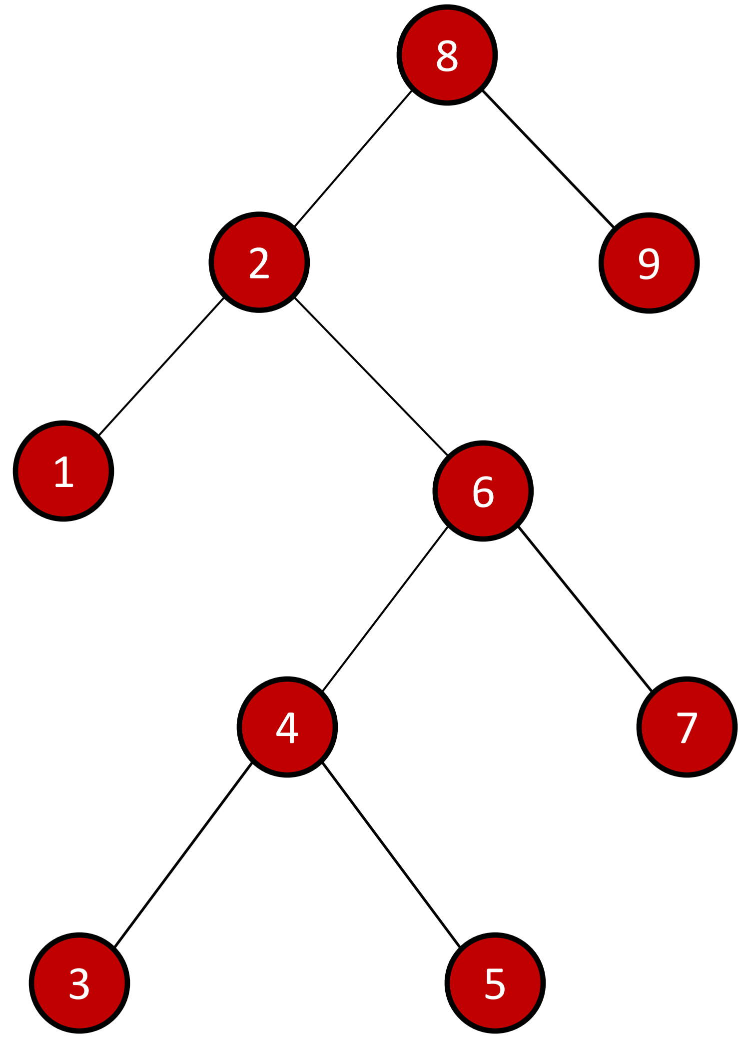
1. If we have a right child
2. **No right child**
   - Have to **move up**
   - But how far?
   - Until we find an **ancestor** node **for whom we are a left descendant**, not a right descendant

```
Successor(n) {
  if (n.right != null)
    return Minimum(n.right);

  p = n.parent;
  while (p != null
         && n == p.right) {
    n = p;
    p = n.parent;
  }
  return p;
}
```

Modifying binary search trees

- So BSTs are good for searching
  - $O(\log n)$ for balanced trees
- But how do we **add and delete** elements?
  - Today: adding without worry about balancing
  - Tomorrow: adding in self-balancing trees



Insertion

Two cases

- Adding to **empty tree**
  - New node becomes root
- Non-empty tree
  - Node added as **child of some leaf node**

Basic idea

- **Search for new key** as if you were expecting to find it
- You'll **fail** (if you don't, no need to add it!)
- Add the node as a leaf of the **last node examined** before failing

Insertion code (returns root of tree)

```
Insert(root, int k) {
  node = new BST(k);
  if (root==null)
    return node;

  parent = FindInsertionPoint(root, k);
  node.parent = parent;
  if (node.key<parent.key)
    parent.left = node;
  else
    parent.right = node;

  return root
}
```

```
FindInsertionPoint(n, int k) {
   parent = null;

   while (n != null) {
      parent = n;
      if (k<n.key)
         n = n.left;
      else
         n = n.right;
   }

   return parent;
}
```

Analysis: how long does it take?

```
Insert(root, int k) {
  node = new BST(k);
  if (root==null)
    return node;

  parent = FindInsertionPoint(root, k);
  node.parent = parent;
  if (node.key<parent.key)
    parent.left = node;
  else
    parent.right = node;

  return root
}
```

```
FindInsertionPoint(n, int k) {
  parent = null;

  while (n != null) {
    parent = n;
    if (k<n.key)
      n = n.left;
    else
      n = n.right;
  }

  return parent;
}
```

Analysis: how long does it take?

```
Insert(root, int k) {
  node = new BST(k);
  if (root==null)
    return node;

  parent = FindInsertionPoint(root, k);
  node.parent = parent;
  if (node.key<parent.key)
    parent.left = node;
  else
    parent.right = node;

  return root
}
```

$$O(h)$$

```
FindInsertionPoint(n, int k) {
  parent = null;

  while (n != null)
    parent = n;
    if (k<n.key)
      n = n.left;
    else
      n = n.right;
  }

  return parent;
}
```

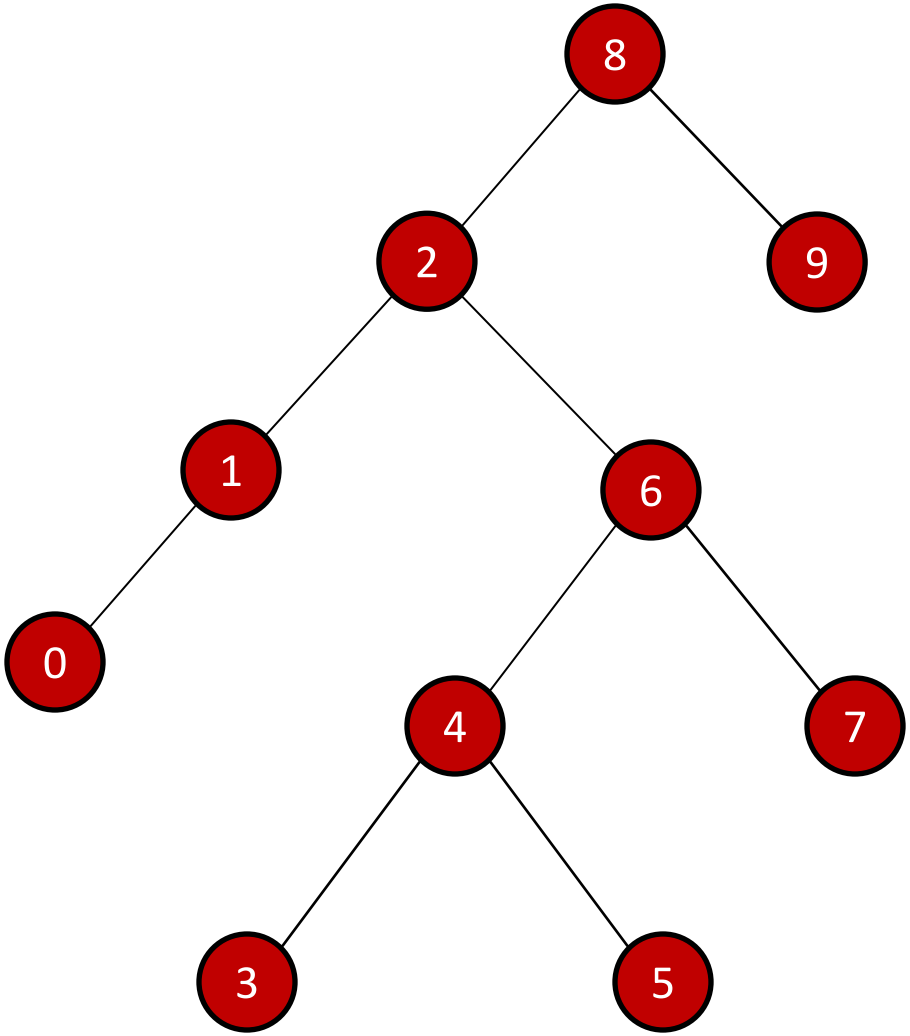
$$O(h)$$

Deletion

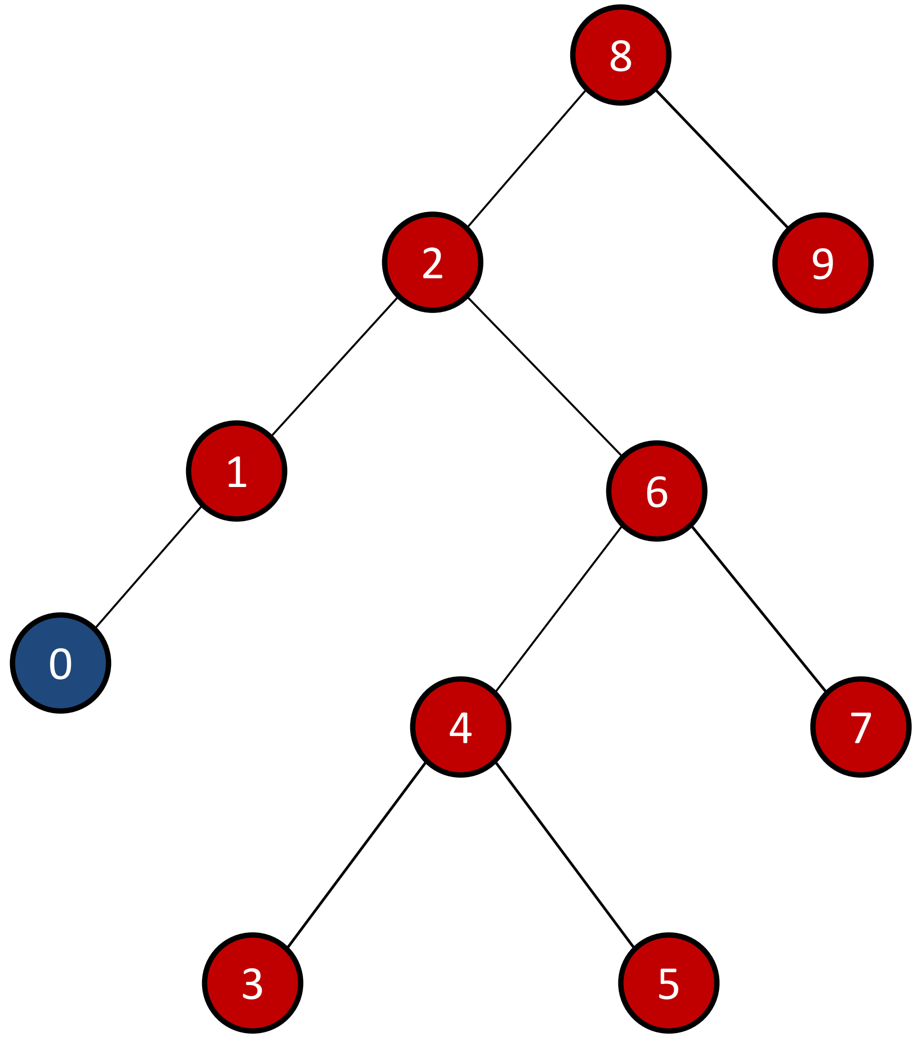
Lots of **case analysis**

- Node has **no children**
  - Set parent's child pointer to **null**
- Node has **one child**
  - **Replace** parent's child pointer with node's child pointer
- Node has **two children**
  - "**Replace**" node with its **successor**
    - Find its successor
    - **Delete** its successor
    - **Change label** of node to label of old successor

Case 1: Node has no children

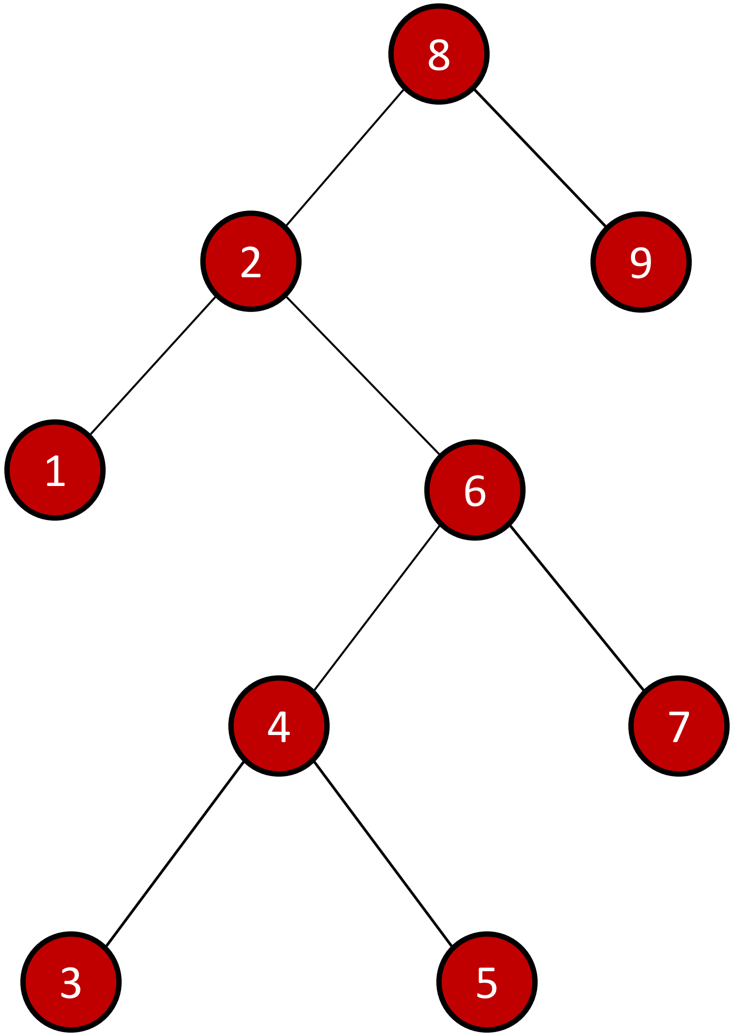


Case 1: Node has no children

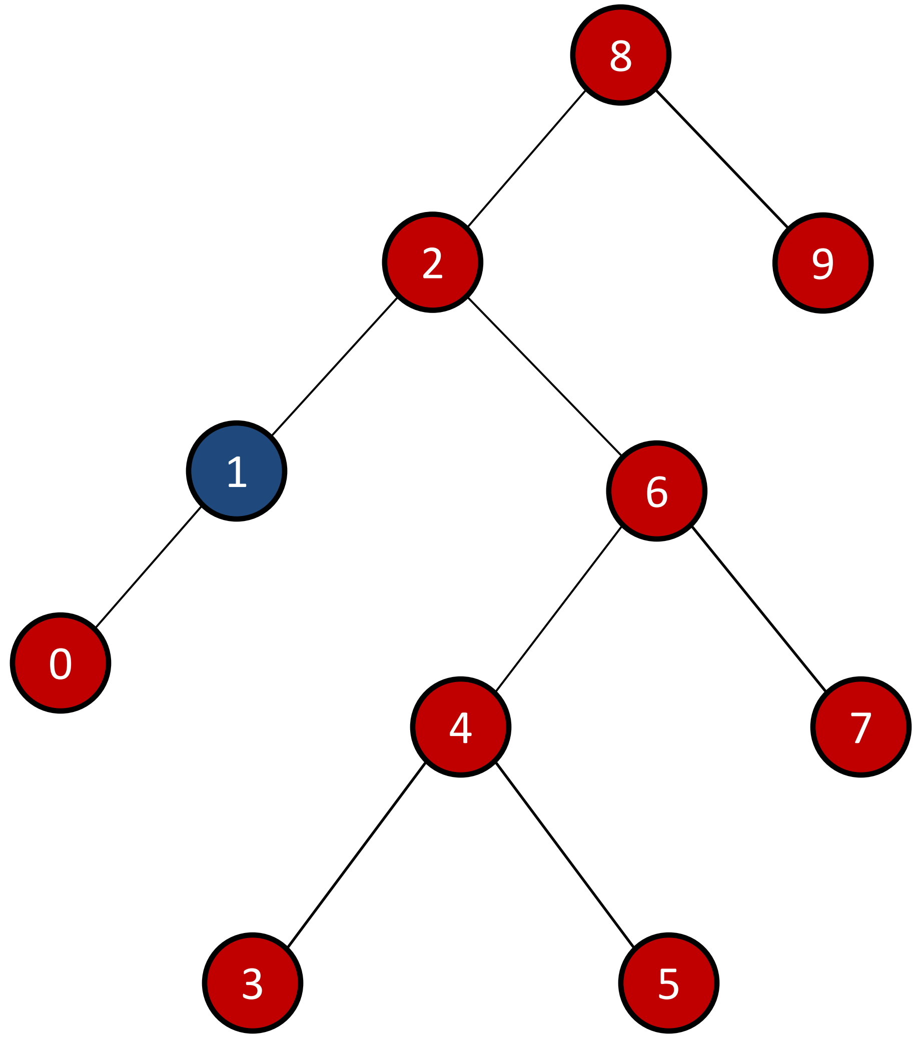


Case 1: Node has no children

- Set parent's child pointer to null
  - And call delete on node, if using C++

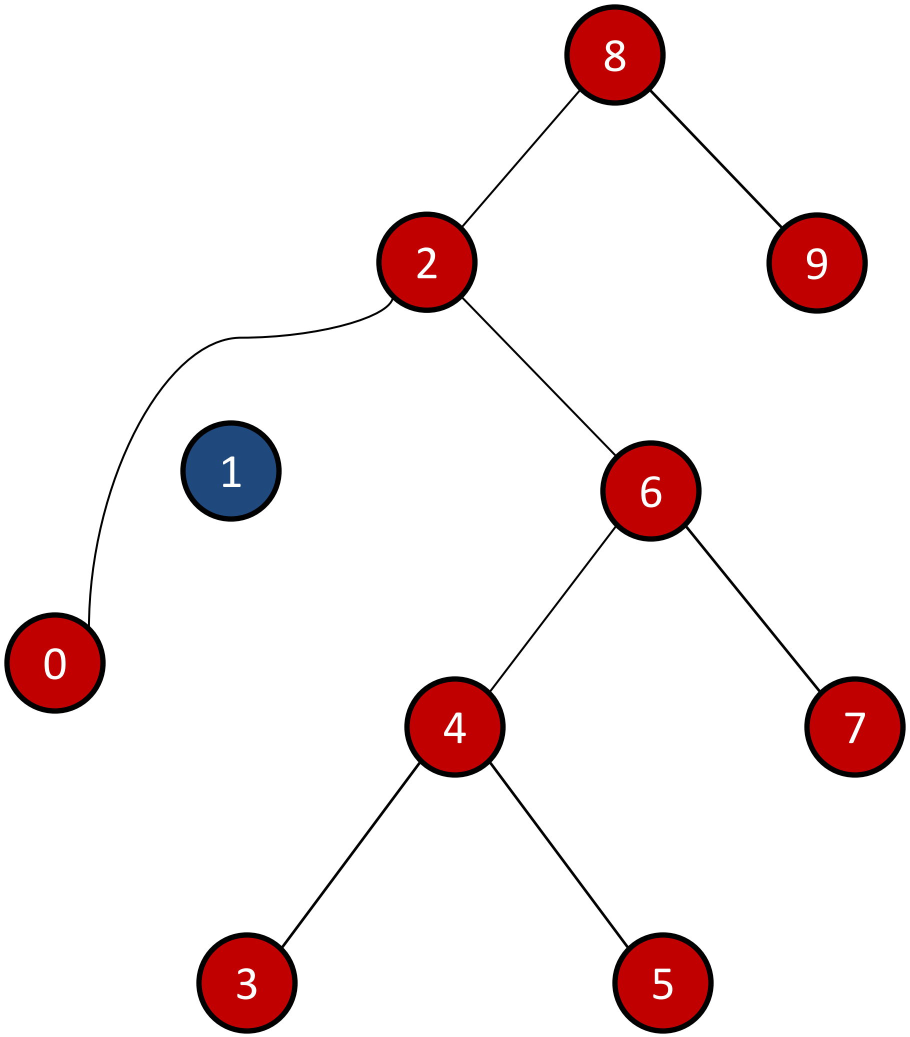


Case 2: Node has one child



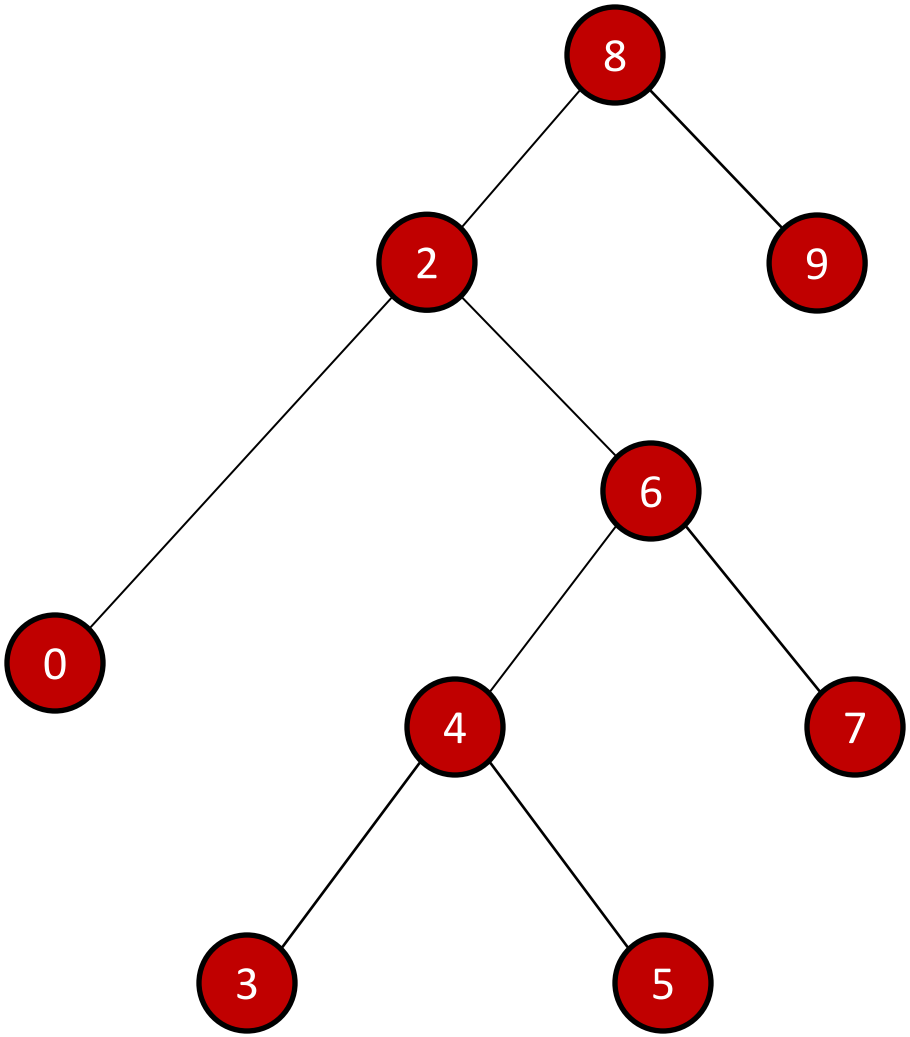
Case 2: Node has one child

- Set parent’s child pointer to point at orphaned grandchild
- Update grandchild’s parent pointer

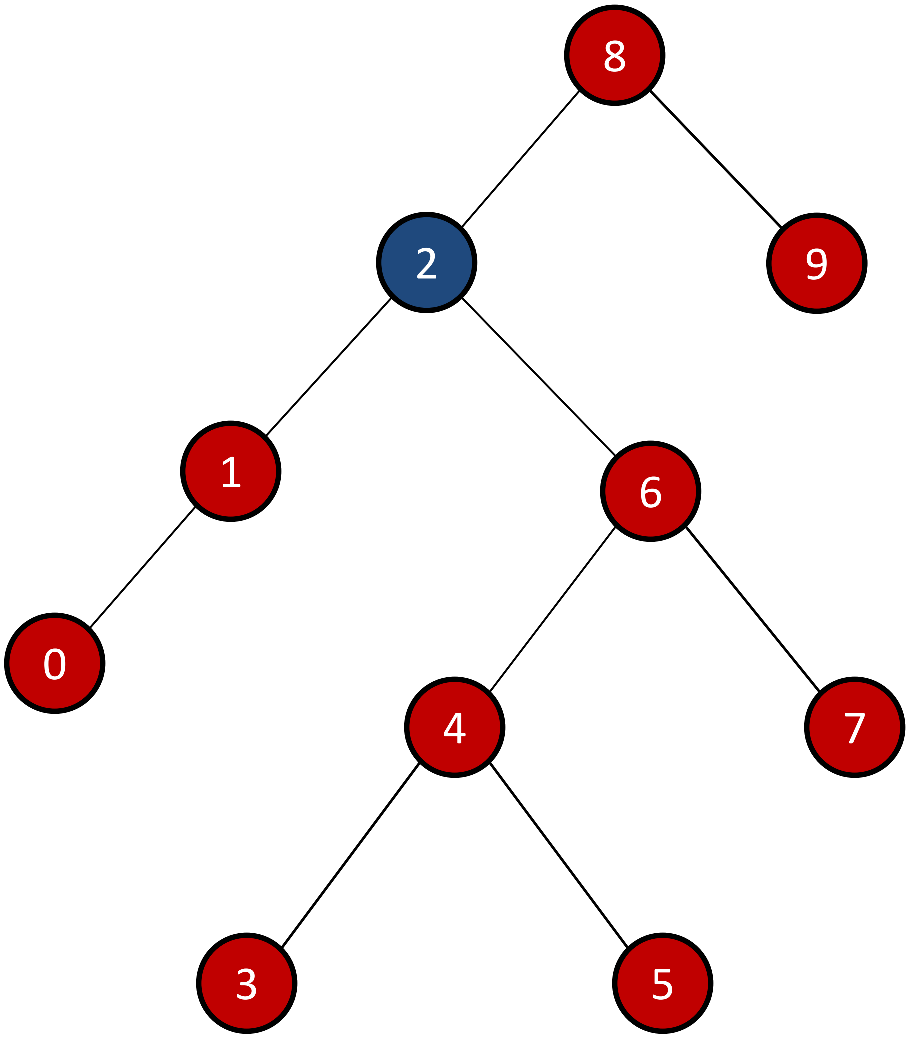


Case 2: Node has one child

- Set parent's child pointer to point at orphaned grandchild
- Update grandchild's parent pointer
- Call delete on old node if using C++

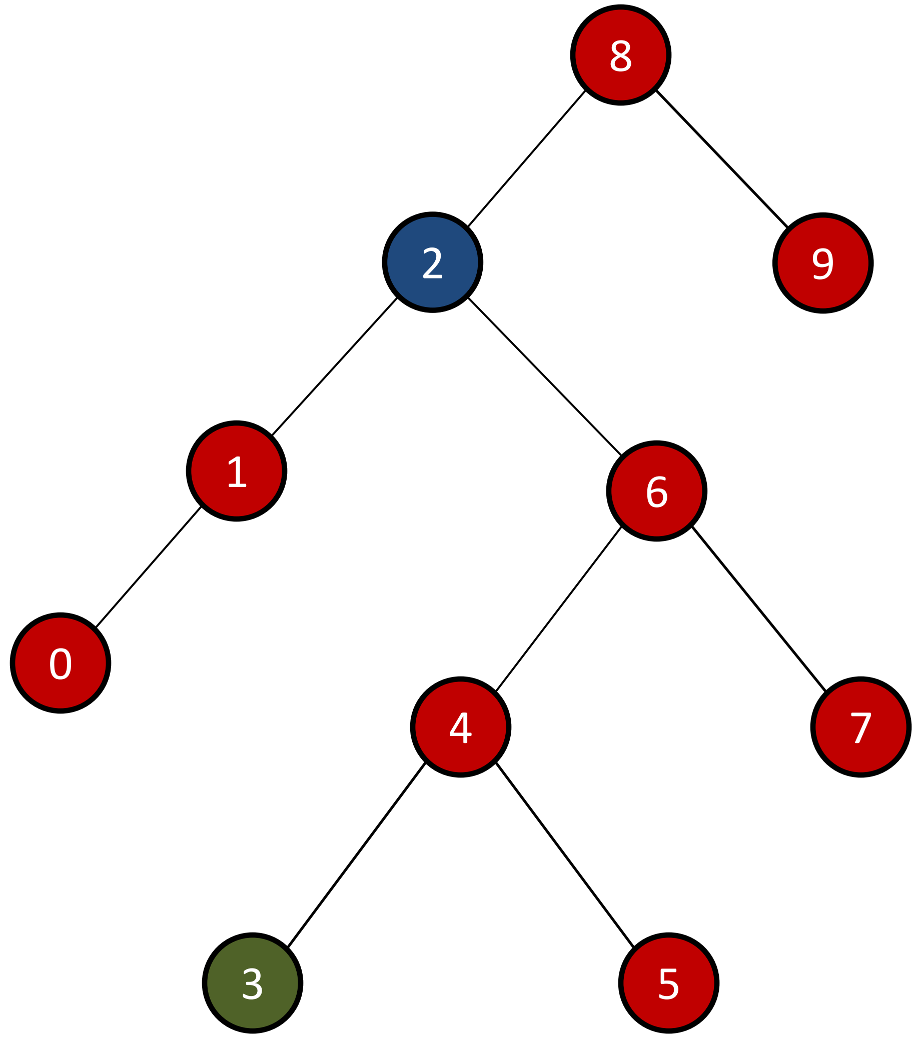


Case 3: Node has two children



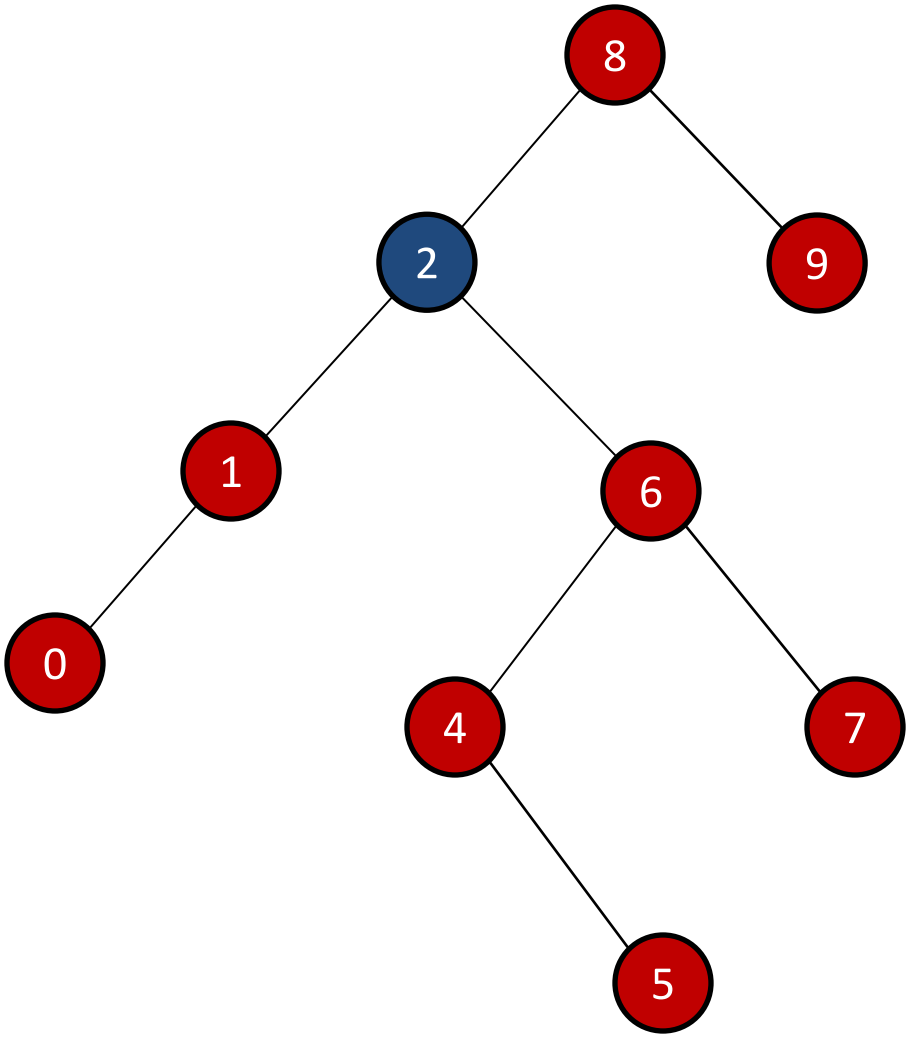
Case 3: Node has two children

- Find successor node
  - Claim: **successor can have a most one child**
  - Proof:
    - Successor is the minimum of the right subtree (by definition)
    - The minimum of a tree can't have a left child
      - Or the child would be less than it
      - So it wouldn't be minimal
    - So at most a right child



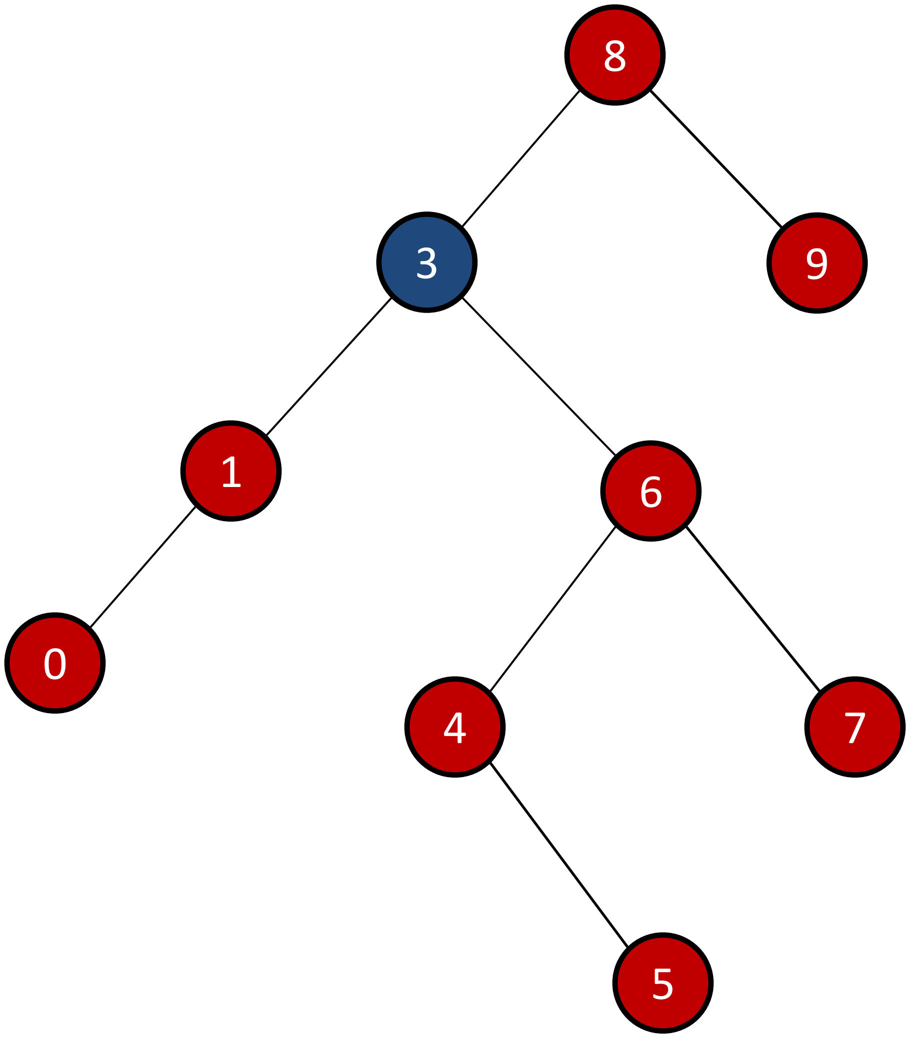
Case 3: Node has two children

- Find successor node
- Delete it
  - Easy, because it falls under case 1 or case 2
  - So at most one level of recursion



Case 3: Node has two children

- Find successor node
- Delete it
- Relabel node with label of successor



Pseudocode

```
Delete(v) {  // v for "victim"
  if (v.left != null && v.right != null)
    Delete2children(v);
  else
    Delete1or0children(v);
}

Delete2children(v) {
  s = Successor(v);
  Delete0or1Children(s);
  v.key = s.key;
}
```

```
Delete1or0children(v) {
  child = v.left, if non-null,
            else v.right

  if (child != null)
    child.parent = v.parent;

  if (v.parent != null) {
    if (v.parent.left == v)
      v.parent.left = child;
    else
      v.parent.right = child;
  }
}
```

Note: this code finesses the case where we delete the root of the tree, however it's easier to read. See the CLR book for (ugly) code that handles root deletion

Analysis: how long do these take?

```
Delete(v) {  // v for "victim"
  if (v.left != null && v.right != null)
    Delete2children(v);
  else
    Delete1or0children(v);
}

Delete2children(v) {
  s = Successor(v);
  Delete0or1Children(s);
  v.key = s.key;
}
```

```
Delete1or0children(v) {
  child = v.left, if non-null,
          else v.right

  if (child != null)
    child.parent = v.parent;

  if (v.parent != null) {
    if (v.parent.left == v)
      v.parent.left = child;
    else
      v.parent.right = child;
  }
}
```

Note: this code finesses the case where we delete the root of the tree, however it's easier to read. See the CLR book for (ugly) code that handles root deletion

Analysis

```
Delete(v) {  // v for "victim"
  if (v.left != null && v.right != null)
    Delete2children(v);
  else
    Delete1or0children(v);
}
```

$O(h)$

```
Delete2children(v) {
  s = Successor(v);
  Delete0or1Children(s);
  v.key = s.key;
}
```

$O(h)$

```
Delete1or0children(v) {
  child = v.left, if non-null,
          else v.right

  if (child != null)
    child.parent = v.parent;

  if (v.parent != null) {
    if (v.parent.left == v)
      v.parent.left = child;
    else
      v.parent.right = child;
  }
}
```

$O(1)$

Note: this code finesses the case where we delete the root of the tree, however it's easier to read. See the CLR book for (ugly) code that handles root deletion

Reading

Introduction to Algorithms (“CLR book”)

- Third edition: chapter 12
- Second edition: chapter 13
- Don’t need to read section on randomly built binary search trees