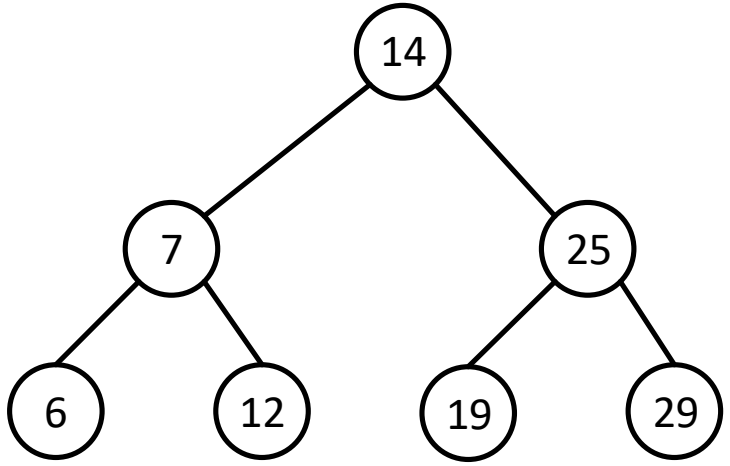
CS2400 - Data Structures and Advanced Programming
Module 10: Trees (III) – BSTs

Hao Ji
Computer Science Department
Cal Poly Pomona

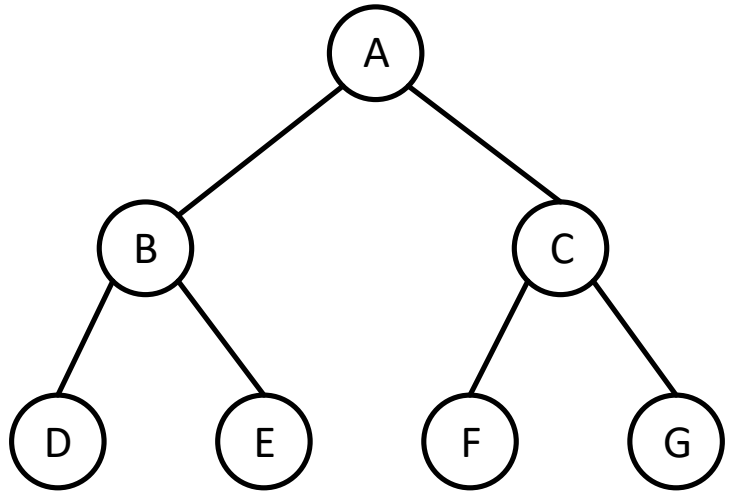
Today

- This Class
  - Binary Search Tree (BST)
    - Definition
    - Operations in Binary Search Tree
      - Search for an Entry
      - Adding an Entry (Iterative Version)
      - Removing an Entry
    - Efficiency of Operations



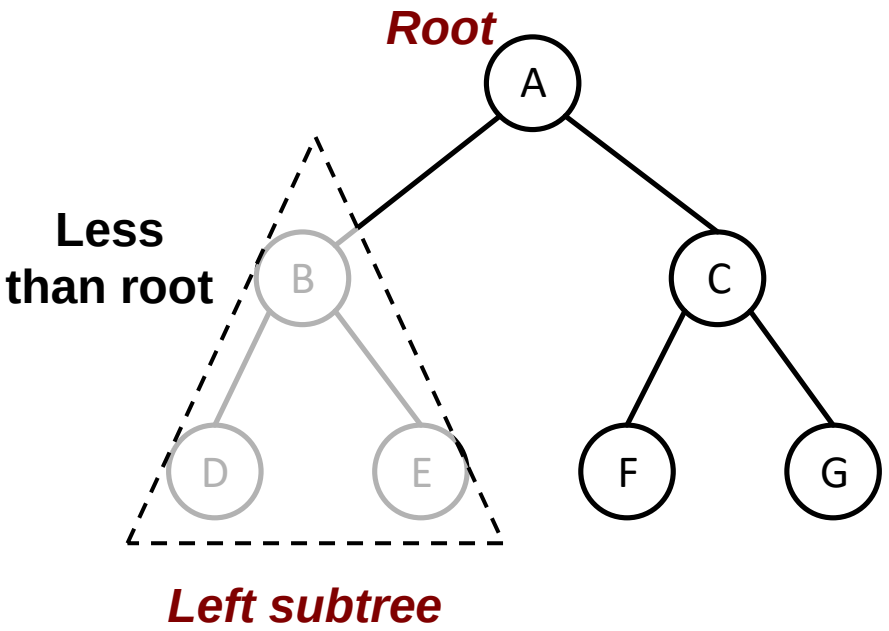
Definition of Binary Search Tree

- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - All values in the **left subtree** must be less than or equal to the root node.
    - All values in the **right subtree** must be greater than the root node.
    - Both the **left subtree** and **right subtree** are BST.



Definition of Binary Search Tree

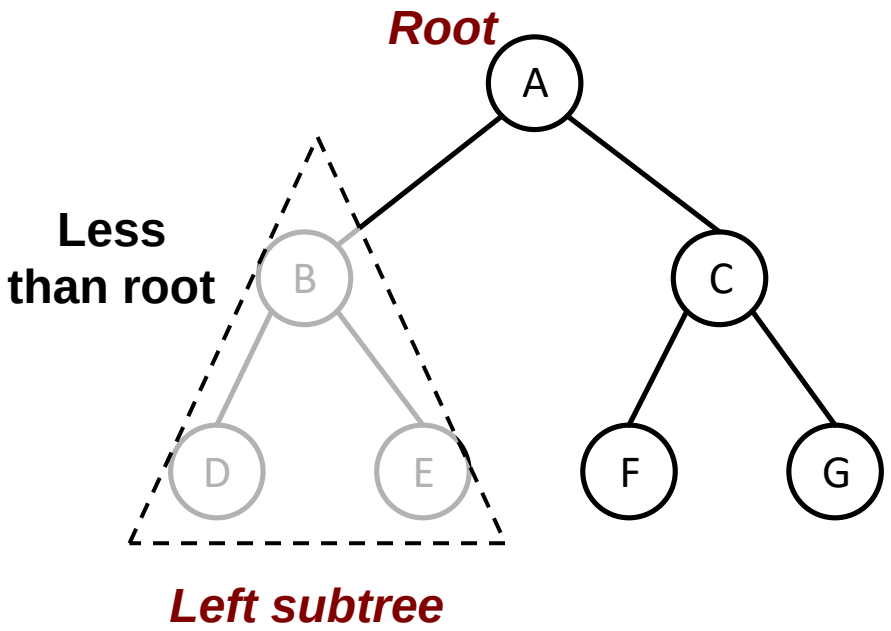
- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - All values in the **left subtree** must be less than or equal to the root node.
    - All values in the **right subtree** must be greater than the root node.
    - Both the **left subtree** and **right subtree** are BST.



Definition of Binary Search Tree

- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - All values in the **left subtree** must be less than or equal to the root node.

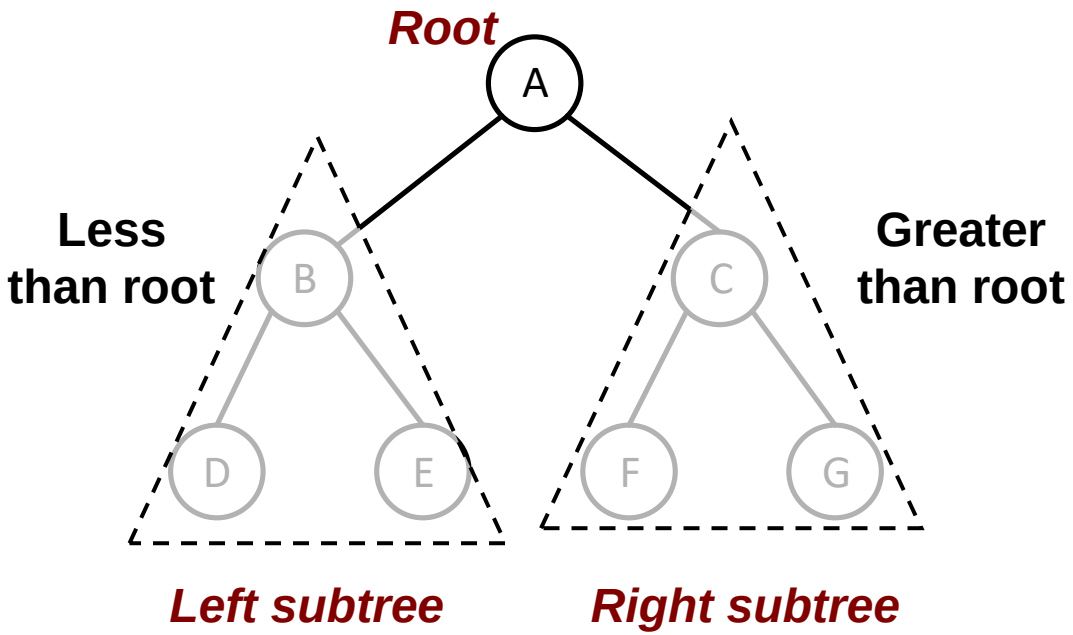
      (Left is less (or equal to) **)**
    - All values in the **right subtree** must be greater than the root node.
    - Both the **left subtree** and **right subtree** are BST.



Definition of Binary Search Tree

- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - All values in the **left subtree** must be less than or equal to the root node.

      (Left is less (or equal to) **)**
    - All values in the **right subtree** must be greater than the root node.
    - Both the **left subtree** and **right subtree** are BST.

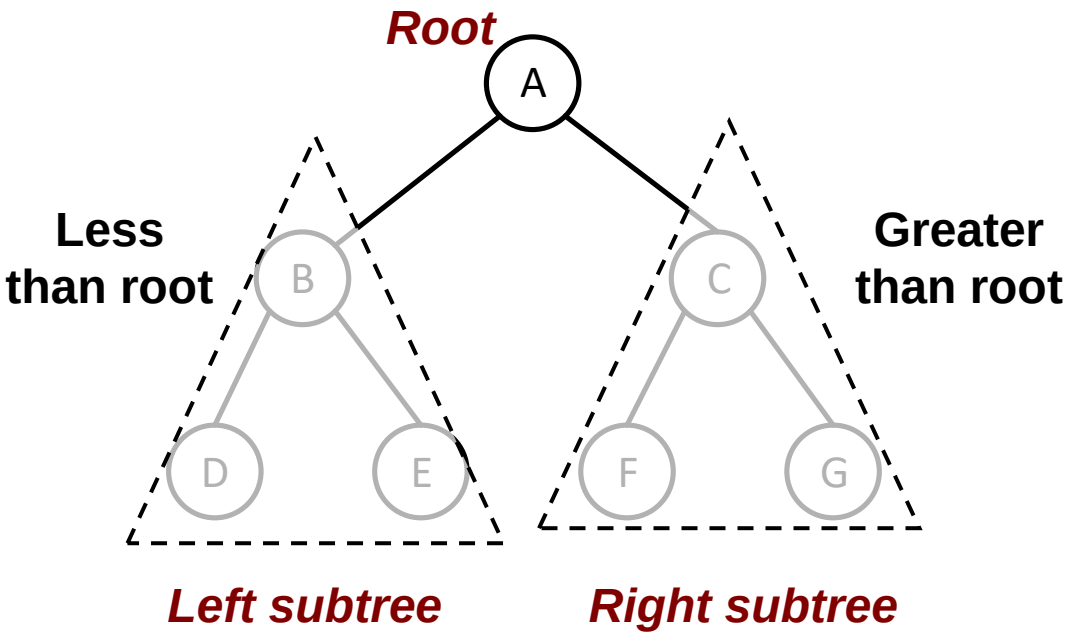


Definition of Binary Search Tree

- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - All values in the **left subtree** must be less than or equal to the root node.

      (Left is less (or equal to) **)**
    - All values in the **right subtree** must be greater than the root node.

      (Right is greater)
    - Both the **left subtree** and **right subtree** are BST.



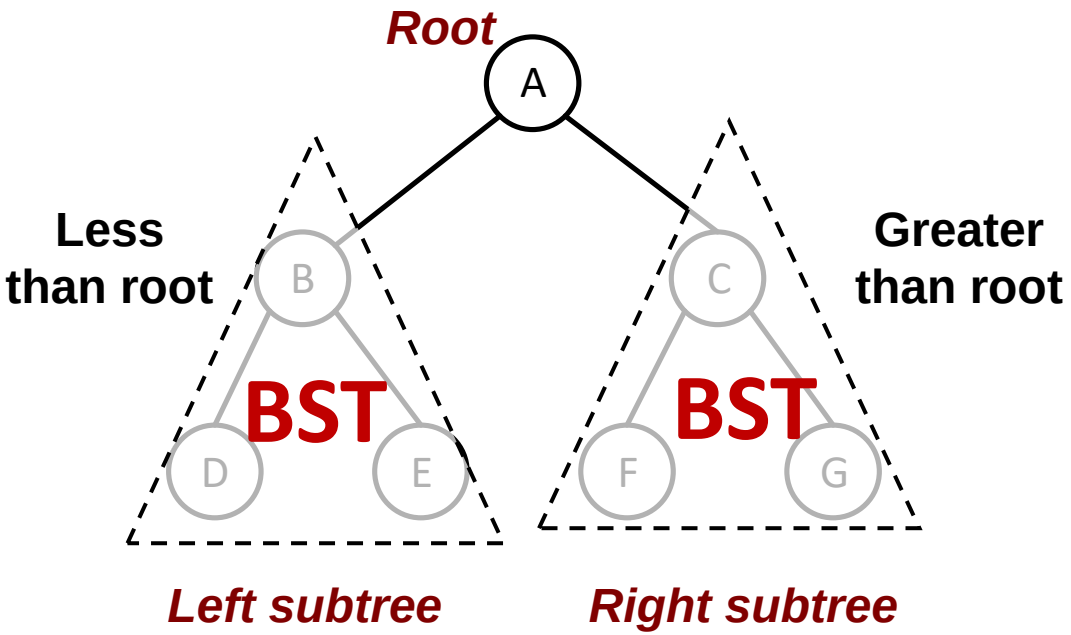
Definition of Binary Search Tree

- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - All values in the **left subtree** must be less than or equal to the root node.

      (Left is less (or equal to) **)**
    - All values in the **right subtree** must be greater than the root node.

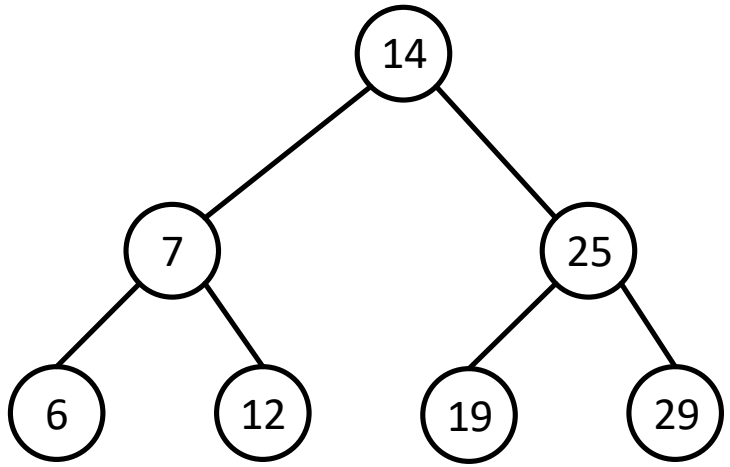
      (Right is greater)
    - Both the **left subtree** and **right subtree** are BST.

      (Follow the rules all the way down)



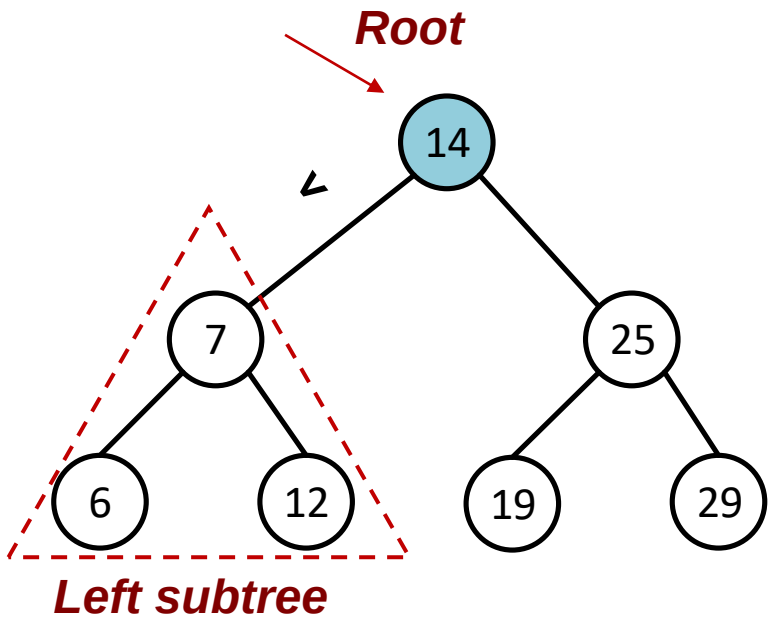
Definition of Binary Search Tree

- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - **Left is less (or equal to)**
    - **Right is greater**
    - **Follow the rules all the way down**



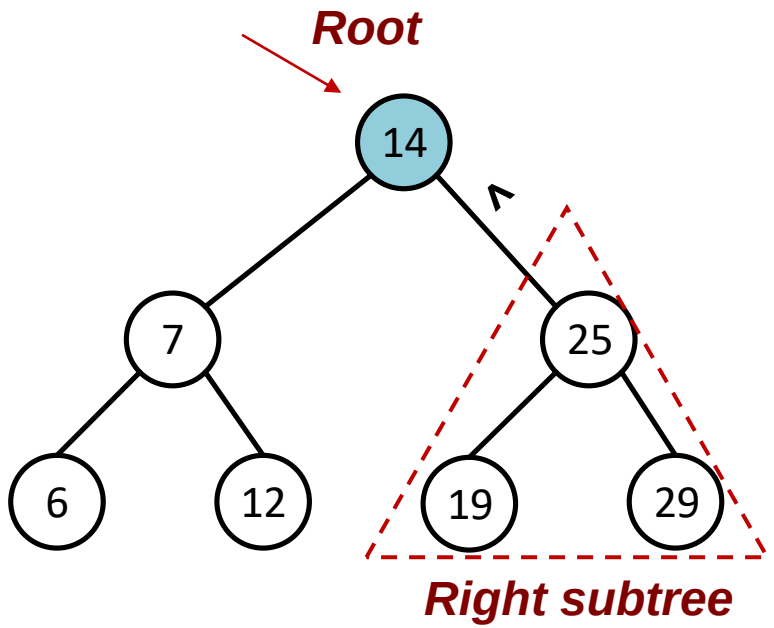
Definition of Binary Search Tree

- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - **Left is less (or equal to)**
    - **Right is greater**
    - **Follow the rules all the way down**



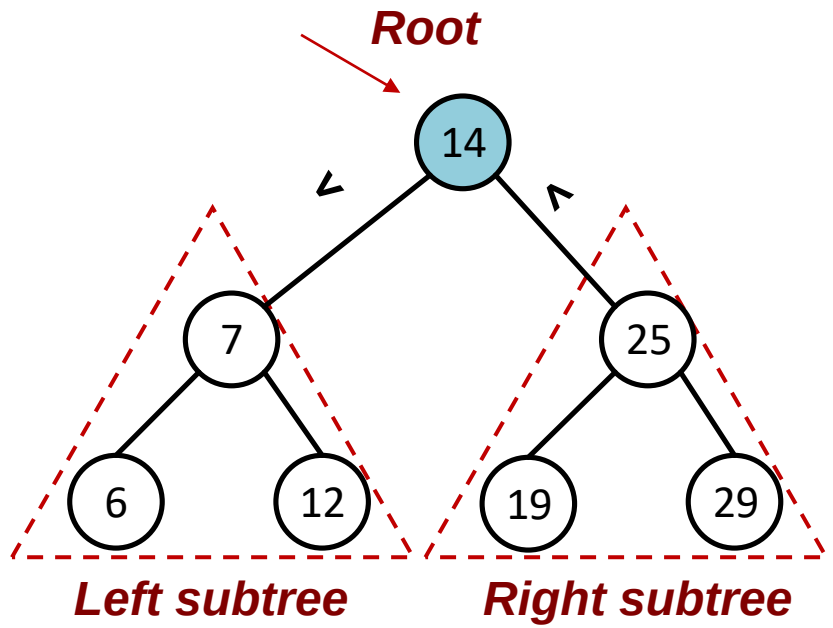
Definition of Binary Search Tree

- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - **Left is less (or equal to)**
    - **Right is greater**
    - **Follow the rules all the way down**



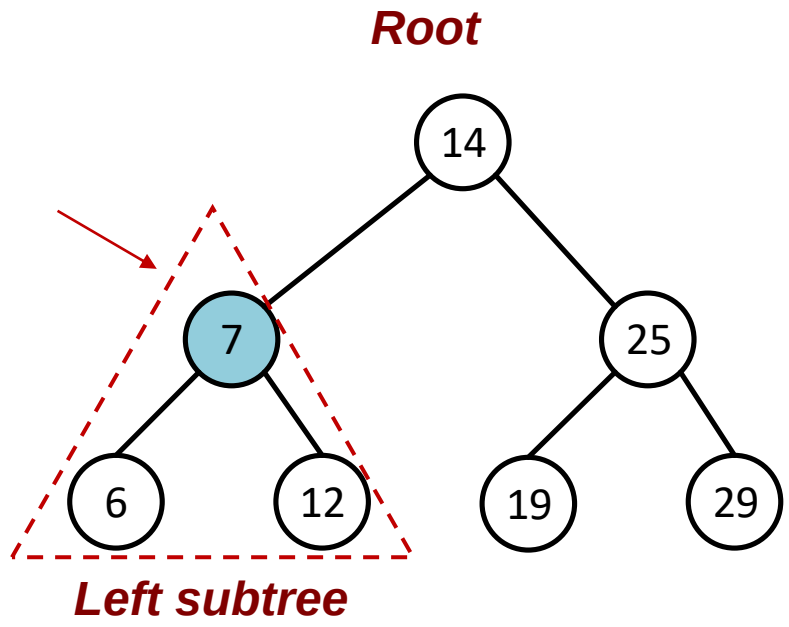
Definition of Binary Search Tree

- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - **Left is less (or equal to)**
    - **Right is greater**
    - **Follow the rules all the way down**



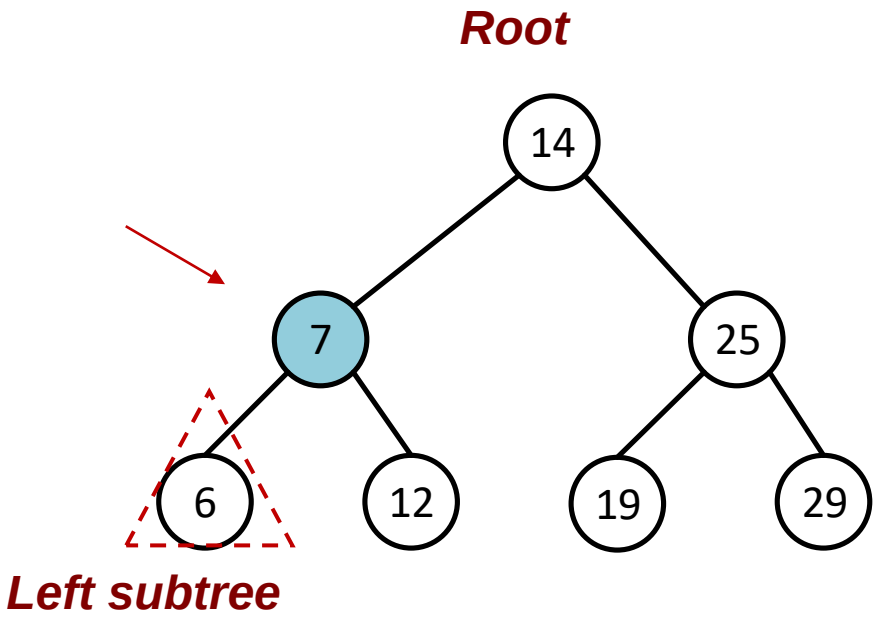
Definition of Binary Search Tree

- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - **Left is less (or equal to)**
    - **Right is greater**
    - **Follow the rules all the way down**



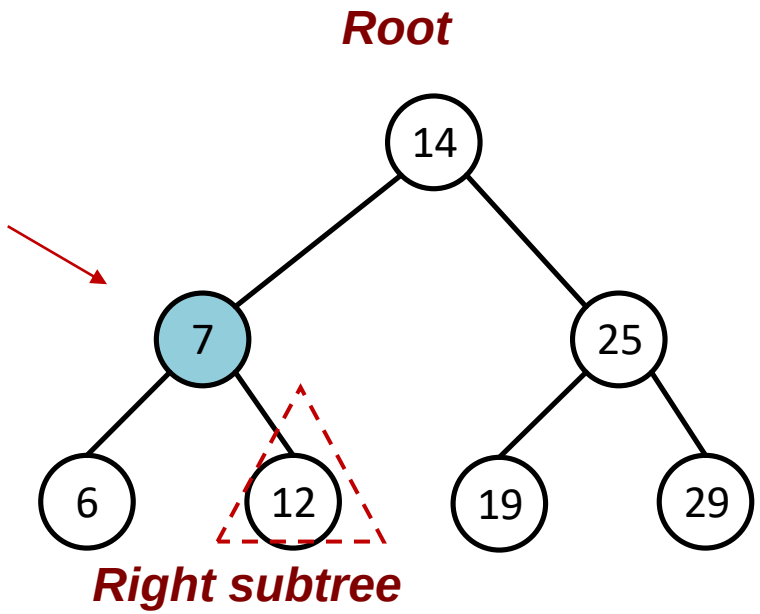
Definition of Binary Search Tree

- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - **Left is less (or equal to)**
    - **Right is greater**
    - **Follow the rules all the way down**



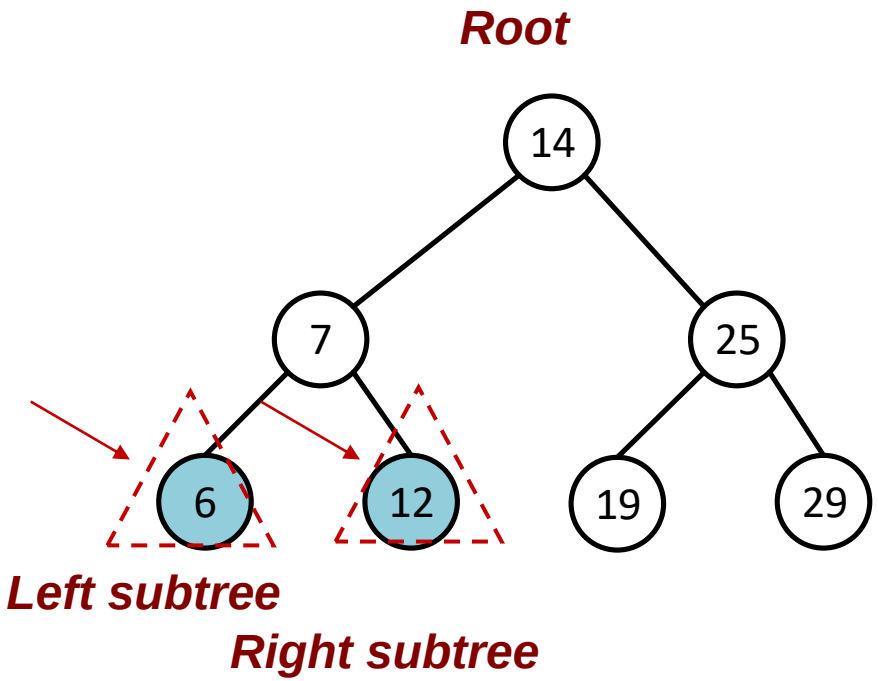
Definition of Binary Search Tree

- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - **Left is less (or equal to)**
    - **Right is greater**
    - **Follow the rules all the way down**



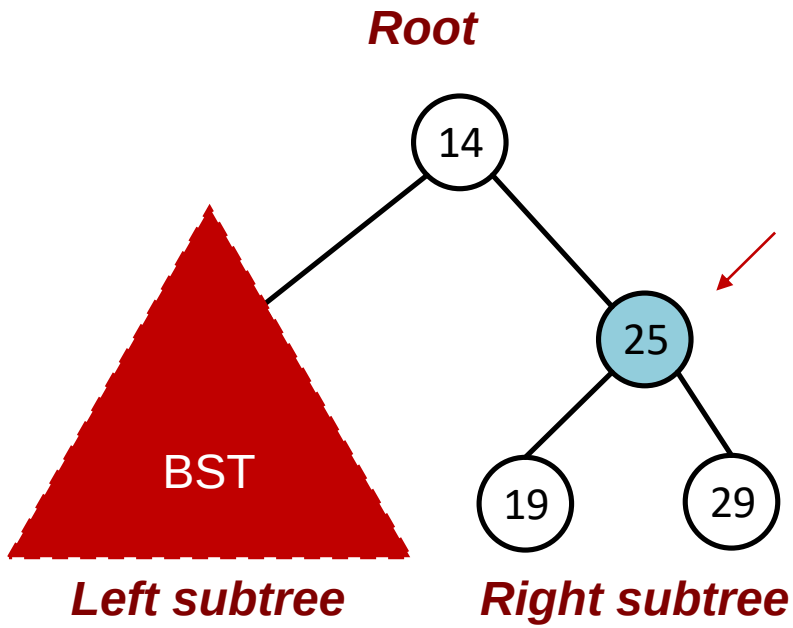
Definition of Binary Search Tree

- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - **Left is less (or equal to)**
    - **Right is greater**
    - **Follow the rules all the way down**



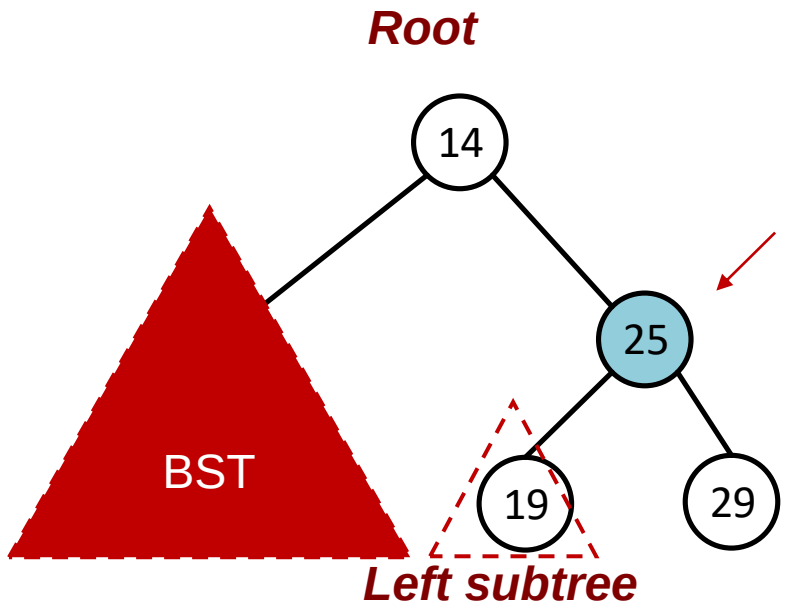
Definition of Binary Search Tree

- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - **Left is less (or equal to)**
    - **Right is greater**
    - **Follow the rules all the way down**



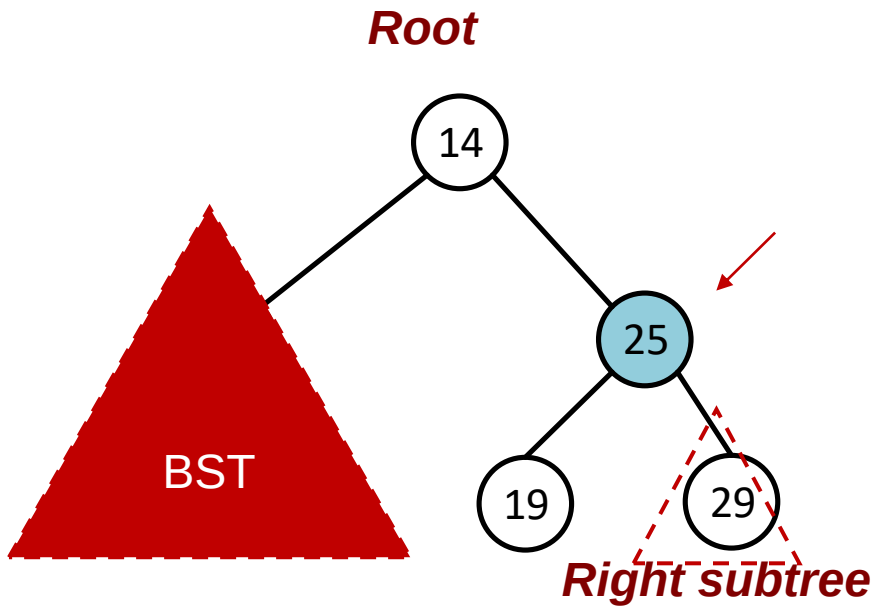
Definition of Binary Search Tree

- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - **Left is less (or equal to)**
    - **Right is greater**
    - **Follow the rules all the way down**



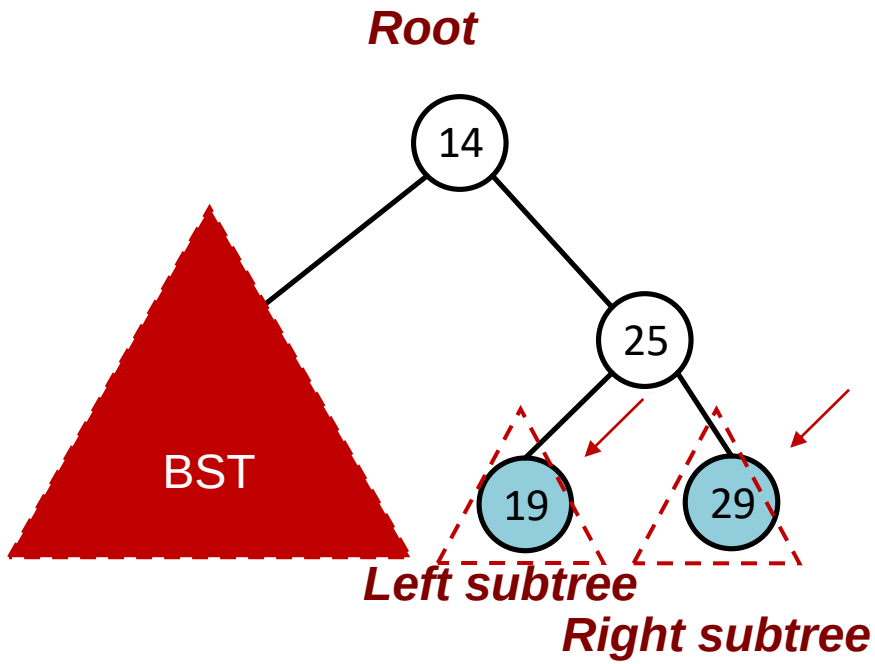
Definition of Binary Search Tree

- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - **Left is less**
    - **Right is greater**
    - **Follow the rules all the way down**



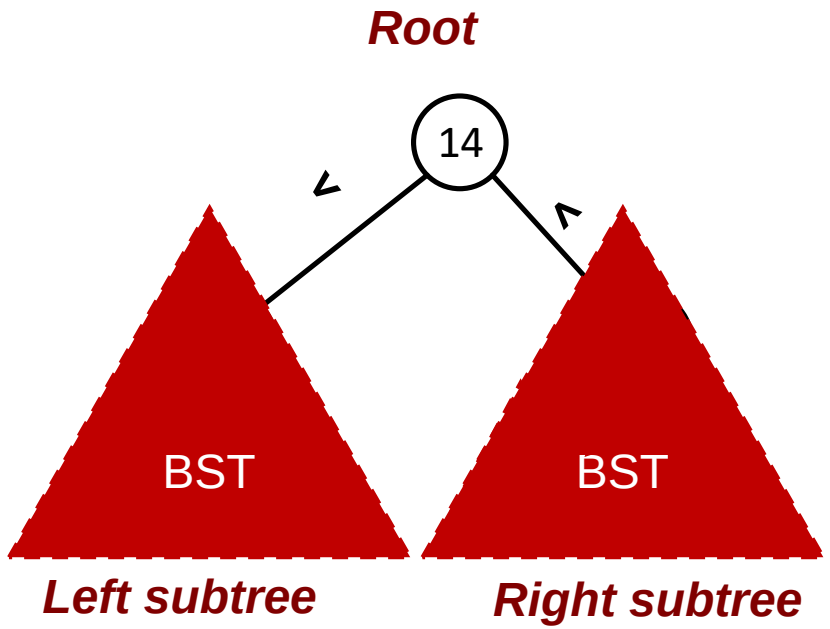
Definition of Binary Search Tree

- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - **Left is less (or equal to)**
    - **Right is greater**
    - **Follow the rules all the way down**



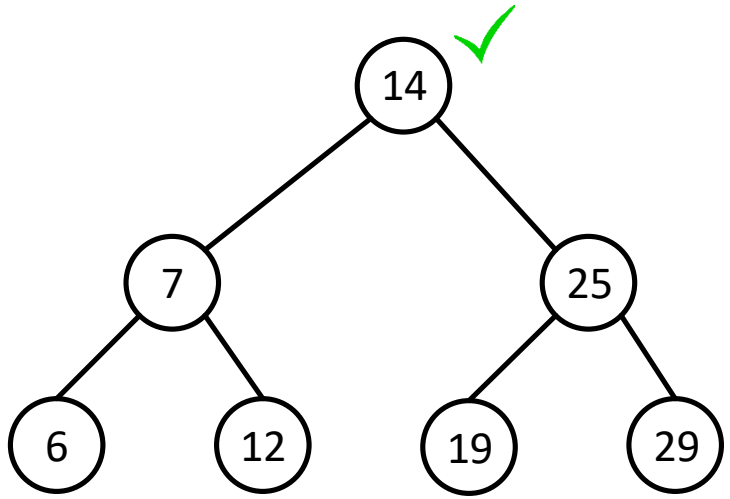
Definition of Binary Search Tree

- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - **Left is less (or equal to)**
    - **Right is greater**
    - **Follow the rules all the way down**



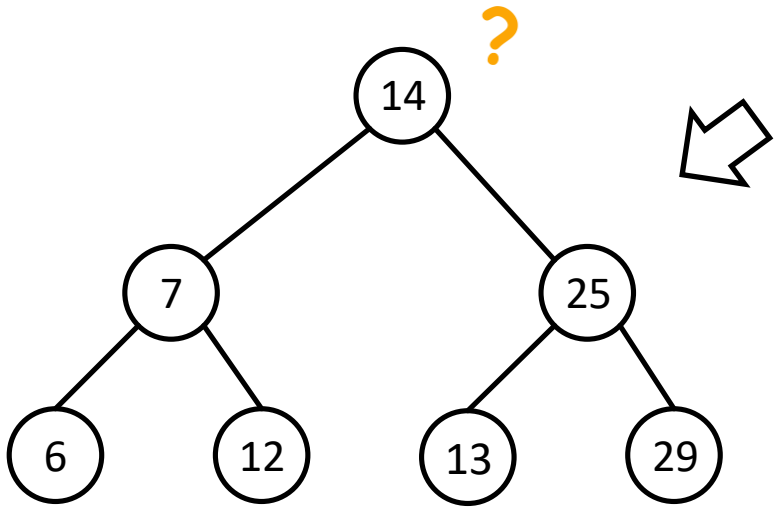
Definition of Binary Search Tree

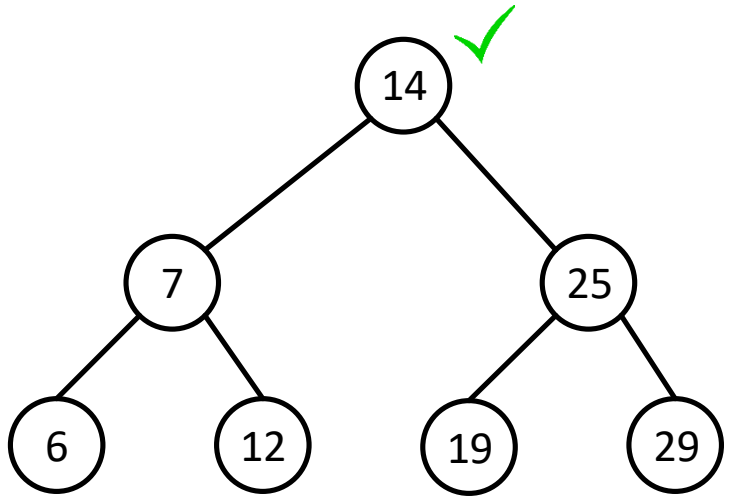
- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - **Left is less (or equal to)**
    - **Right is greater**
    - **Follow the rules all the way down**



Is This a Binary Search Tree?

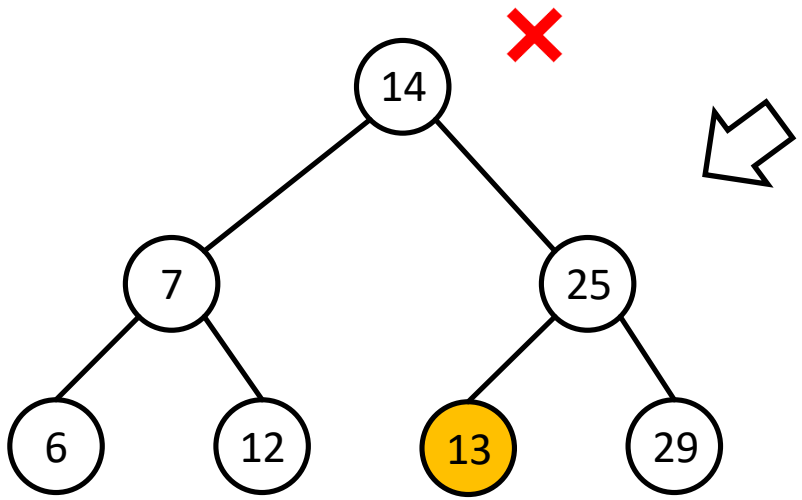
- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - **Left is less (or equal to)**
    - **Right is greater**
    - **Follow the rules all the way down**

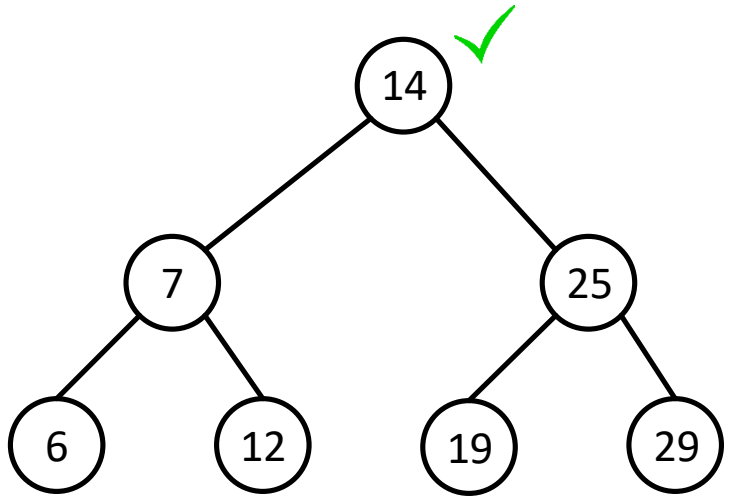




Is This a Binary Search Tree?

- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - **Left is less (or equal to)**
    - **Right is greater**
    - **Follow the rules all the way down**





Operations in Binary Search Trees

- An interface for a search tree

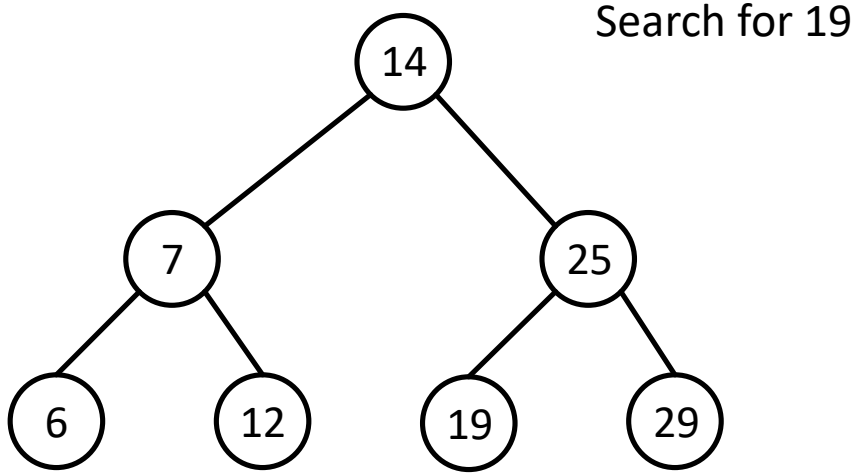
- Operations
  - Searching
  - Adding an entry
  - Removing an entry
  - …

```
1  package TreePackage;
2  import java.util.Iterator;
3  public interface SearchTreeInterface<T extends Comparable<? super T>>
4          extends TreeInterface<T>
5  {
6     /** Searches for a specific entry in this tree.
7         @param entry  An object to be found.
8         @return  True if the object was found in the tree. */
9     public boolean contains(T entry);
10
11    /** Retrieves a specific entry in this tree.
12        @param entry  An object to be found.
13        @return  Either the object that was found in the tree or
14                 null if no such object exists. */
15    public T getEntry(T entry);
16
17    /** Adds a new entry to this tree, if it does not match an existing
18        object in the tree. Otherwise, replaces the existing object with
19        the new entry.
20        @param newEntry  An object to be added to the tree.
21        @return  Either null if newEntry was not in the tree already, or
22                 the existing entry that matched the parameter newEntry
23                 and has been replaced in the tree. */
24    public T add(T newEntry);
25
26    /** Removes a specific entry from this tree.
27        @param entry  An object to be removed.
28        @return  Either the object that was removed from the tree or
29                 null if no such object exists. */
30    public T remove(T entry);
31
32    /** Creates an iterator that traverses all entries in this tree.
33        @return  An iterator that provides sequential and ordered access
34                 to the entries in the tree. */
35    public Iterator<T> getInorderIterator();
36 } // end SearchTreeInterface
```

Method getEntry in BinarySearchTree

```
public T getEntry(T entry)
{
   return findEntry(getRootNode(), entry);
} // end getEntry

private T findEntry(BinaryNode<T> rootNode, T entry)
{
   T result = null;

   if (rootNode != null)
   {
      T rootEntry = rootNode.getData();
      if (entry.equals(rootEntry))
         result = rootEntry;
      else if (entry.compareTo(rootEntry) < 0)
         result = findEntry(rootNode.getLeftChild(), entry);
      else
         result = findEntry(rootNode.getRightChild(), entry);
   } // end if

   return result;
} // end findEntry
```

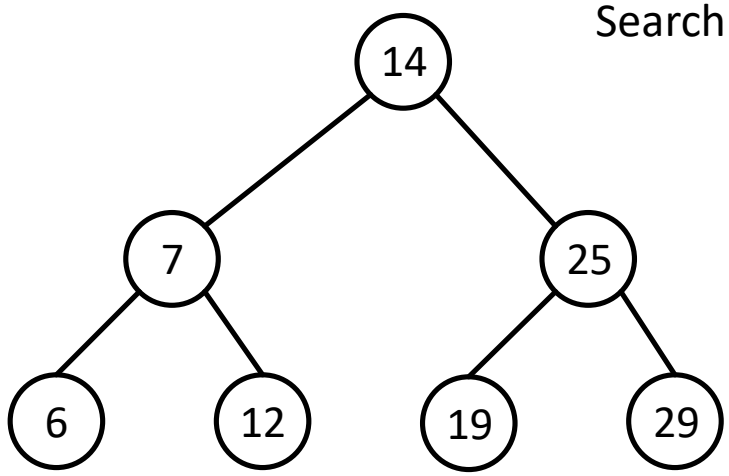
Recursive implementation



Method getEntry in BinarySearchTree

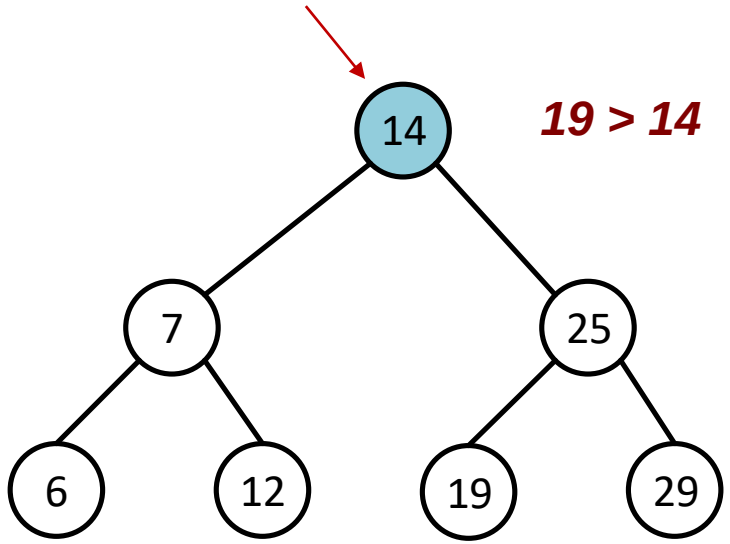
```
public T getEntry(T entry)
{
   return findEntry(getRootNode(), entry);
} // end getEntry

private T findEntry(BinaryNode<T> rootNode, T entry)
{
   T result = null;

   if (rootNode != null)
   {
      T rootEntry = rootNode.getData();
      if (entry.equals(rootEntry))
         result = rootEntry;
      else if (entry.compareTo(rootEntry) < 0)
         result = findEntry(rootNode.getLeftChild(), entry);
      else
         result = findEntry(rootNode.getRightChild(), entry);
   } // end if

   return result;
} // end findEntry
```

Search for 19



Searching a Binary Search Tree

```
public T getEntry(T entry)
{
   return findEntry(getRootNode(), entry);
} // end getEntry

private T findEntry(BinaryNode<T> rootNode, T entry)
{
   T result = null;

   if (rootNode != null)
   {
      T rootEntry = rootNode.getData();
      if (entry.equals(rootEntry))
         result = rootEntry;
      else if (entry.compareTo(rootEntry) < 0)
         result = findEntry(rootNode.getLeftChild(), entry);
      else
         result = findEntry(rootNode.getRightChild(), entry);
   } // end if

   return result;
} // end findEntry
```

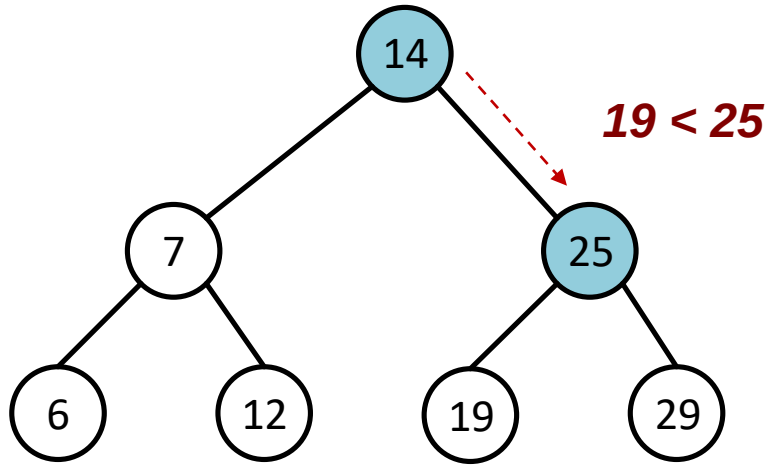


Left Less (or equal to)

Right Greater

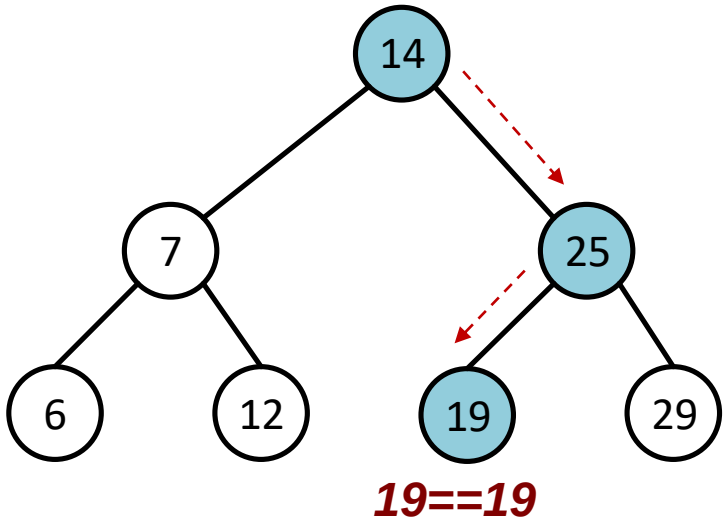
Searching a Binary Search Tree

```
public T getEntry(T entry)
{
   return findEntry(getRootNode(), entry);
} // end getEntry

private T findEntry(BinaryNode<T> rootNode, T entry)
{
   T result = null;

   if (rootNode != null)
   {
      T rootEntry = rootNode.getData();
      if (entry.equals(rootEntry))
         result = rootEntry;
      else if (entry.compareTo(rootEntry) < 0)
         result = findEntry(rootNode.getLeftChild(), entry);
      else
         result = findEntry(rootNode.getRightChild(), entry);
   } // end if

   return result;
} // end findEntry
```



Searching a Binary Search Tree

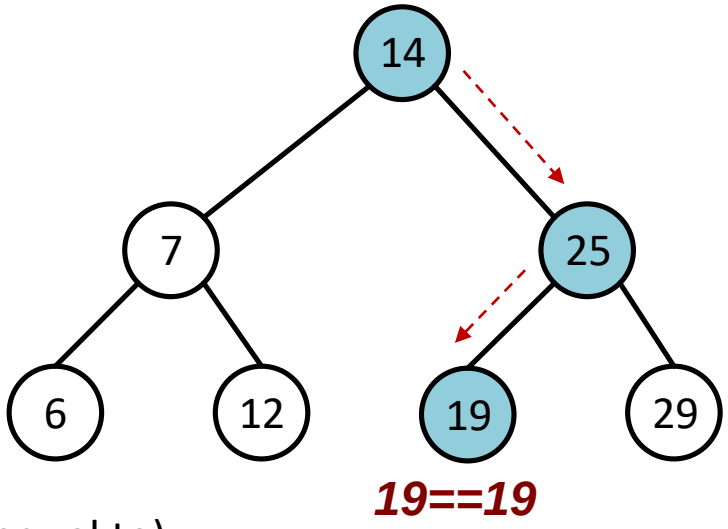
```
public T getEntry(T entry)
{
   return findEntry(getRootNode(), entry);
} // end getEntry

private T findEntry(BinaryNode<T> rootNode, T entry)
{
   T result = null;

   if (rootNode != null)
   {
      T rootEntry = rootNode.getData();
      if (entry.equals(rootEntry))
         result = rootEntry;
      else if (entry.compareTo(rootEntry) < 0)
         result = findEntry(rootNode.getLeftChild(), entry);
      else
         result = findEntry(rootNode.getRightChild(), entry);
   } // end if

   return result;
} // end findEntry
```



Right Greater

Searching a Binary Search Tree

- Search for 19



3 Steps

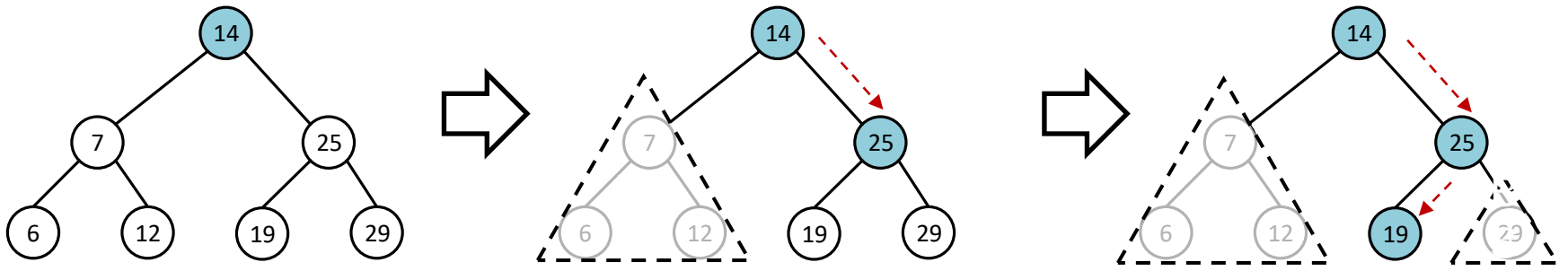
19==19

Left Less (or equal to)

Right Greater

Efficiency of a Search

- Search for 19



7 nodes

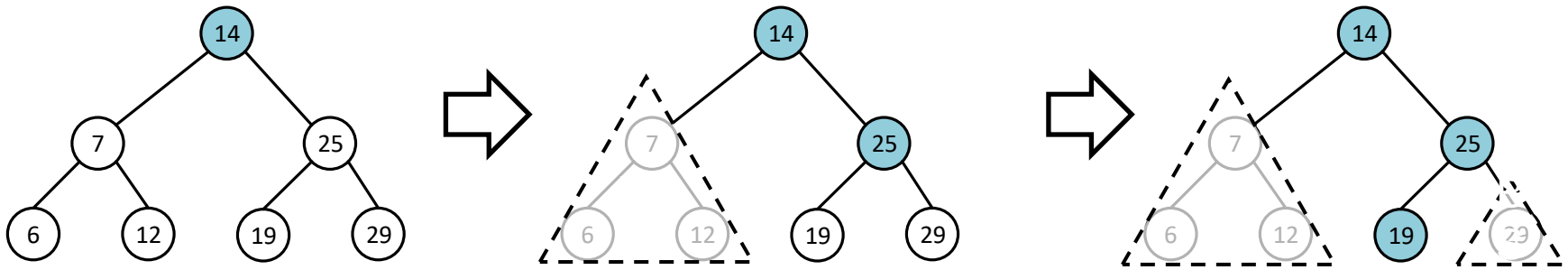
$\frac{7}{2}$ = 3 nodes

$\frac{3}{2}$ = 1 node

3 Steps

Efficiency of a Search

- Search for 19



7 nodes

$\frac{7}{2}$ = 3 nodes

$\frac{3}{2}$ = 1 node

3 Steps

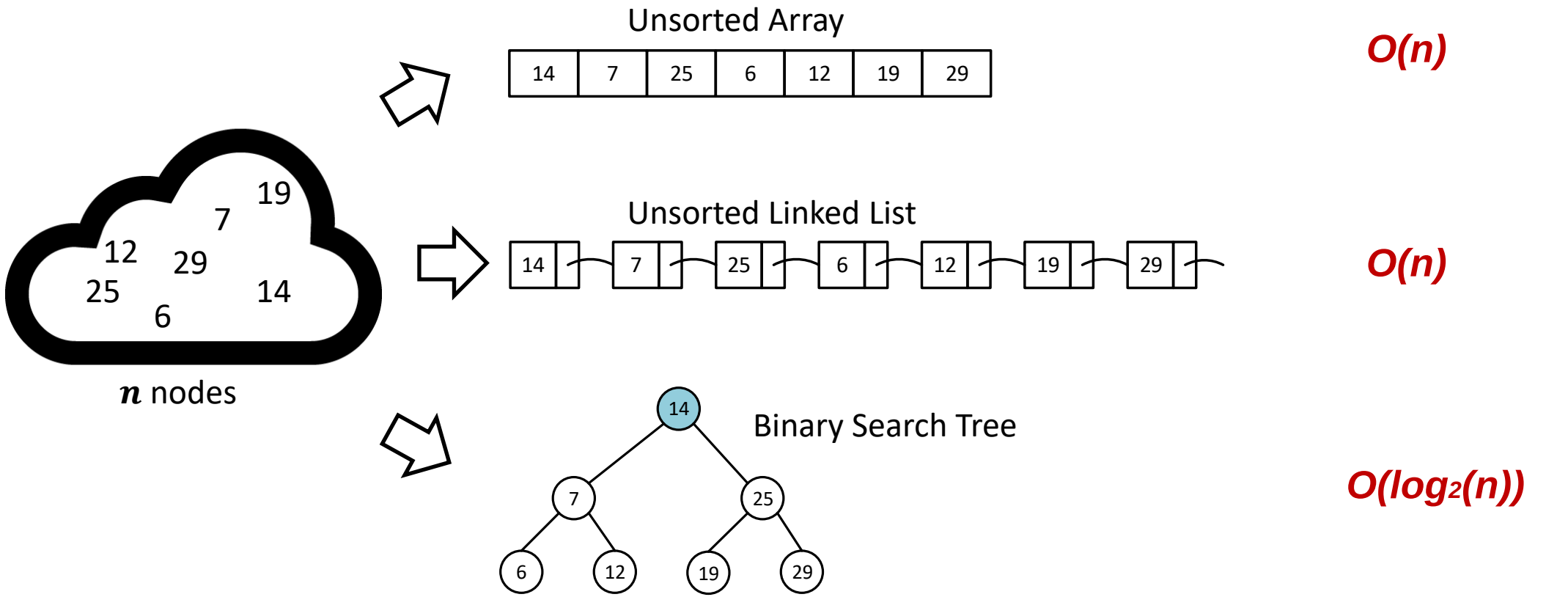
- If a BST has n nodes,

$log_2(n)$ Steps

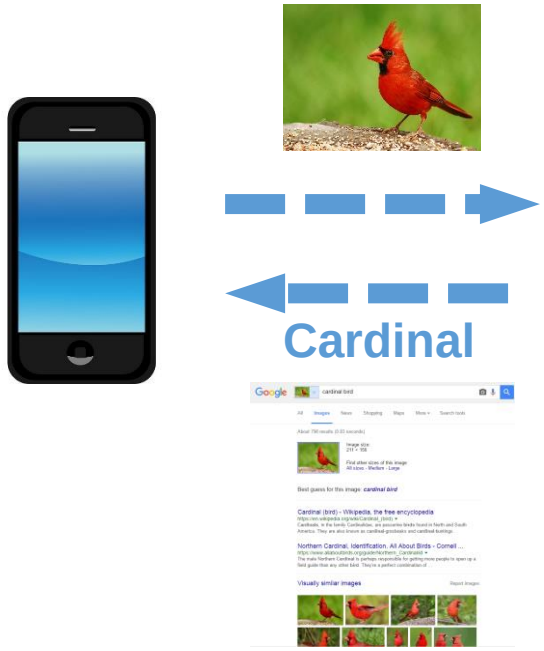
n nodes ⇨ $\frac{n}{2}$ **nodes** ⇨ $\frac{n}{4}$ **nodes** ⇨ … ⇨ **1 node**

Efficiency of a Search

- Performance Comparisons

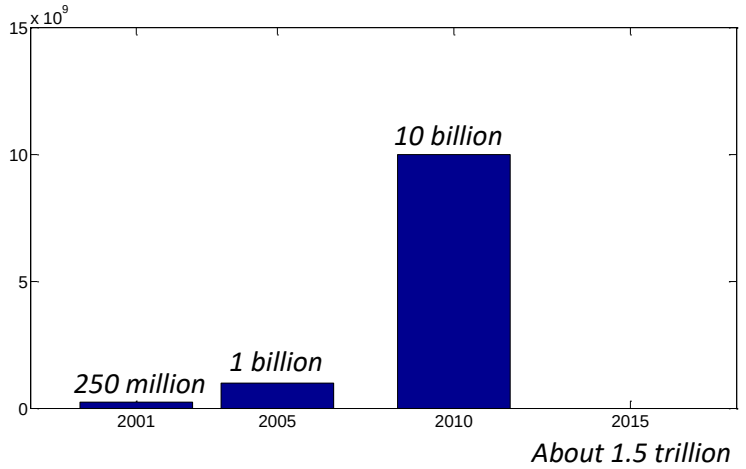


Example: Google image search



Google Image Search Service

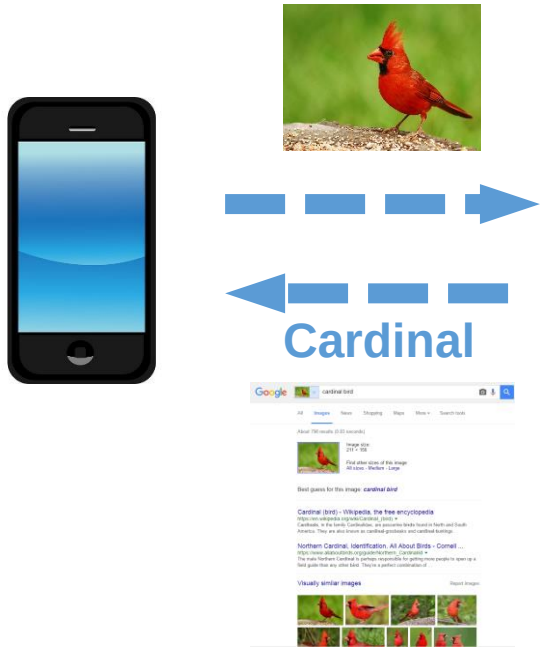
Number of images indexed by Google



About 1.5 trillion

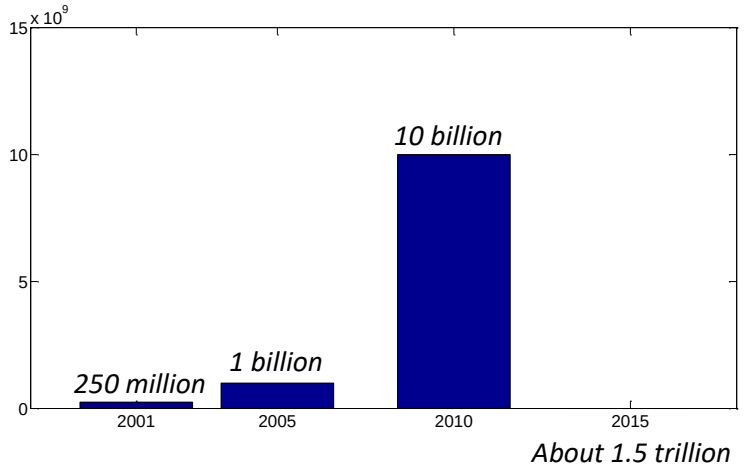
http://www.birdlist.org/images/virginia_state_bird_northern_cardinal.jpg

Example: Google image search



Google Image Search Service

Number of images indexed by Google



Search time (Binary Search Tree): *O(log₂(n))*

$log_2(1.5 \times 10^{12}) \times 10^{-9} \approx 40.44 \times 10^{-9} seconds$

Search time (Unsorted Array or Linked List) : *O(n)*

$1.5 \times 10^{12} \times 10^{-9} = 1500\ seconds$

http://www.birdlist.org/images/virginia_state_bird_northern_cardinal.jpg

Operations in Binary Search Trees

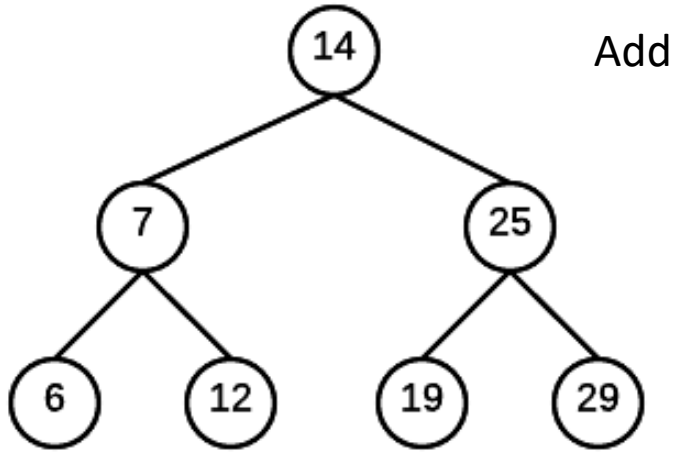
- An interface for a search tree

- Operations
  - Searching
  - **Adding an entry**
  - Removing an entry
  - …

```
1  package TreePackage;
2  import java.util.Iterator;
3  public interface SearchTreeInterface<T extends Comparable<? super T>>
4          extends TreeInterface<T>
5  {
6     /** Searches for a specific entry in this tree.
7         @param entry  An object to be found.
8         @return  True if the object was found in the tree. */
9     public boolean contains(T entry);
10
11    /** Retrieves a specific entry in this tree.
12        @param entry  An object to be found.
13        @return  Either the object that was found in the tree or
14                 null if no such object exists. */
15    public T getEntry(T entry);
16
17    /** Adds a new entry to this tree, if it does not match an existing
18        object in the tree. Otherwise, replaces the existing object with
19        the new entry.
20        @param newEntry  An object to be added to the tree.
21        @return  Either null if newEntry was not in the tree already, or
22                 the existing entry that matched the parameter newEntry
23                 and has been replaced in the tree. */
24    public T add(T newEntry);
25
26    /** Removes a specific entry from this tree.
27        @param entry  An object to be removed.
28        @return  Either the object that was removed from the tree or
29                 null if no such object exists. */
30    public T remove(T entry);
31
32    /** Creates an iterator that traverses all entries in this tree.
33        @return  An iterator that provides sequential and ordered access
34                 to the entries in the tree. */
35    public Iterator<T> getInorderIterator();
36 } // end SearchTreeInterface
```

Adding an Entry (Iterative Version)

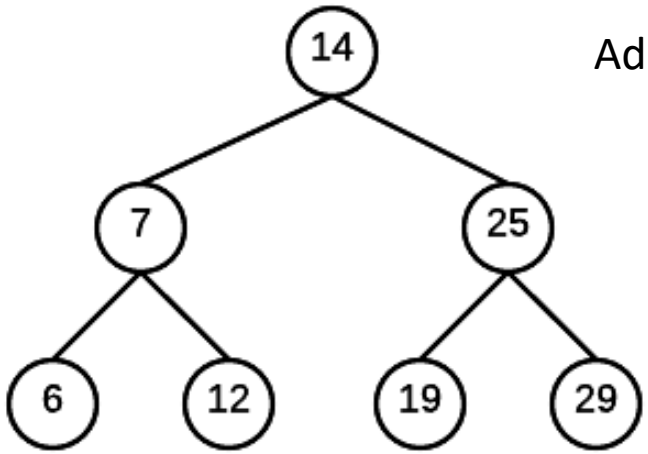
```
private T addEntry(T newEntry)
{
   BinaryNodeInterface<T> currentNode = getRootNode();
   assert currentNode != null;
   T result = null;
   boolean found = false;

   while (!found)
   {
      T currentEntry = currentNode.getData();
      int comparison = newEntry.compareTo(currentEntry);

      if (comparison == 0)
      {  // newEntry matches currentEntry;
         // return and replace currentEntry
         found = true;
         result = currentEntry;
         currentNode.setData(newEntry);
      }
      else if (comparison < 0)
      {
         if (currentNode.hasLeftChild())
            currentNode = currentNode.getLeftChild();
         else
         {
            found = true;
            currentNode.setLeftChild(new BinaryNode<T>(newEntry));
         } // end if
      }
      else
      {
         assert comparison > 0;

         if (currentNode.hasRightChild())
            currentNode = currentNode.getRightChild();
         else
         {
            found = true;
            currentNode.setRightChild(new BinaryNode<T>(newEntry));
         } // end if
      } // end if
   } // end while

   return result;
} // end addEntry
```



Add an entry 11

Adding an Entry (Iterative Version)

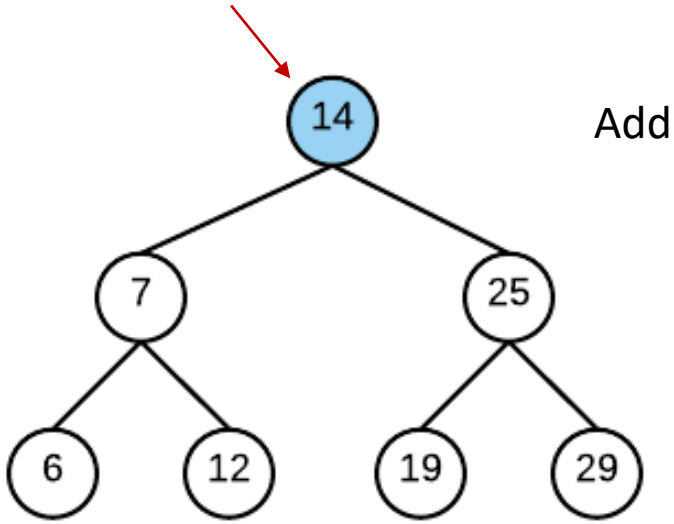
```
private T addEntry(T newEntry)
{
   BinaryNodeInterface<T> currentNode = getRootNode();
   assert currentNode != null;
   T result = null;
   boolean found = false;

   while (!found)
   {
      T currentEntry = currentNode.getData();
      int comparison = newEntry.compareTo(currentEntry);

      if (comparison == 0)
      {  // newEntry matches currentEntry;
         // return and replace currentEntry
         found = true;
         result = currentEntry;
         currentNode.setData(newEntry);
      }
      else if (comparison < 0)
      {
         if (currentNode.hasLeftChild())
            currentNode = currentNode.getLeftChild();
         else
         {
            found = true;
            currentNode.setLeftChild(new BinaryNode<T>(newEntry));
         } // end if
      }
      else
      {
         assert comparison > 0;

         if (currentNode.hasRightChild())
            currentNode = currentNode.getRightChild();
         else
         {
            found = true;
            currentNode.setRightChild(new BinaryNode<T>(newEntry));
         } // end if
      } // end if
   } // end while

   return result;
} // end addEntry
```



Add an entry 11

Adding an Entry (Iterative Version)

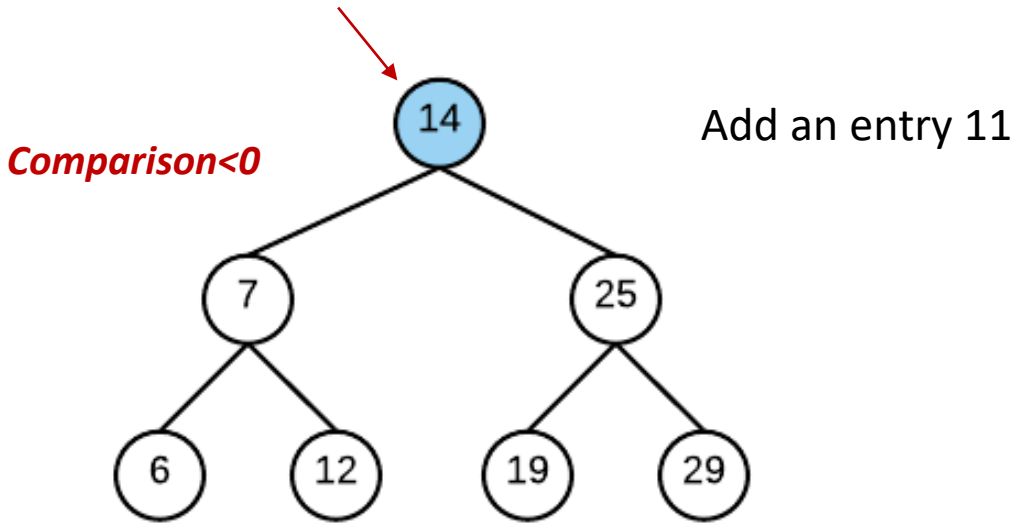
```
private T addEntry(T newEntry)
{
   BinaryNodeInterface<T> currentNode = getRootNode();
   assert currentNode != null;
   T result = null;
   boolean found = false;

   while (!found)
   {
      T currentEntry = currentNode.getData();
      int comparison = newEntry.compareTo(currentEntry);

      if (comparison == 0)
      {  // newEntry matches currentEntry;
         // return and replace currentEntry
         found = true;
         result = currentEntry;
         currentNode.setData(newEntry);
      }
      else if (comparison < 0)
      {
         if (currentNode.hasLeftChild())
            currentNode = currentNode.getLeftChild();
         else
         {
            found = true;
            currentNode.setLeftChild(new BinaryNode<T>(newEntry));
         } // end if
      }
      else
      {
         assert comparison > 0;

         if (currentNode.hasRightChild())
            currentNode = currentNode.getRightChild();
         else
         {
            found = true;
            currentNode.setRightChild(new BinaryNode<T>(newEntry));
         } // end if
      } // end if
   } // end while

   return result;
} // end addEntry
```



Add an entry 11

Adding an Entry (Iterative Version)

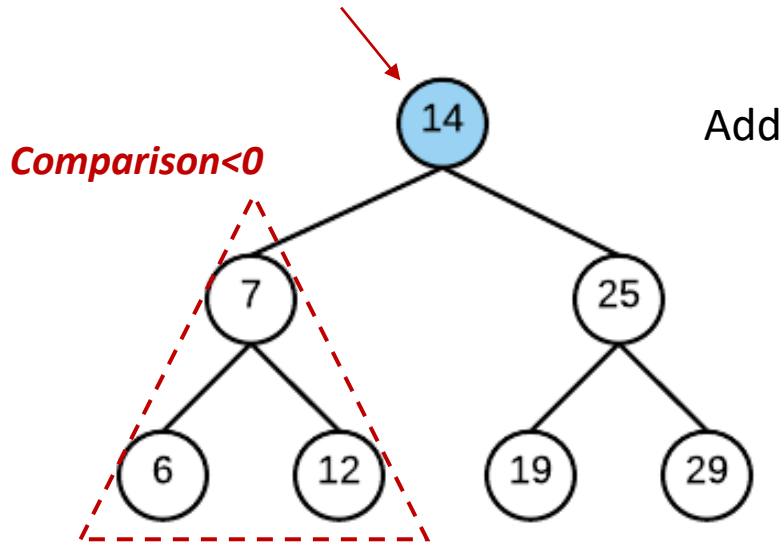
```
private T addEntry(T newEntry)
{
   BinaryNodeInterface<T> currentNode = getRootNode();
   assert currentNode != null;
   T result = null;
   boolean found = false;

   while (!found)
   {
      T currentEntry = currentNode.getData();
      int comparison = newEntry.compareTo(currentEntry);

      if (comparison == 0)
      {  // newEntry matches currentEntry;
         // return and replace currentEntry
         found = true;
         result = currentEntry;
         currentNode.setData(newEntry);
      }
      else if (comparison < 0)
      {
         if (currentNode.hasLeftChild())
            currentNode = currentNode.getLeftChild();
         else
         {
            found = true;
            currentNode.setLeftChild(new BinaryNode<T>(newEntry));
         } // end if
      }
      else
      {
         assert comparison > 0;

         if (currentNode.hasRightChild())
            currentNode = currentNode.getRightChild();
         else
         {
            found = true;
            currentNode.setRightChild(new BinaryNode<T>(newEntry));
         } // end if
      } // end if
   } // end while

   return result;
} // end addEntry
```



Add an entry 11

Adding an Entry (Iterative Version)

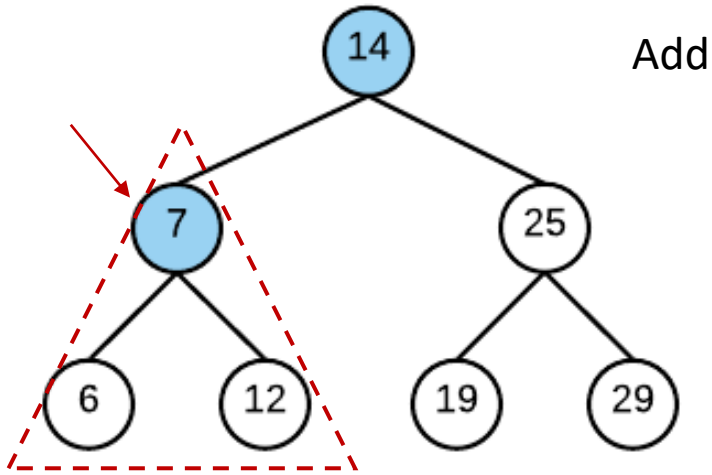
```
private T addEntry(T newEntry)
{
   BinaryNodeInterface<T> currentNode = getRootNode();
   assert currentNode != null;
   T result = null;
   boolean found = false;

   while (!found)
   {
      T currentEntry = currentNode.getData();
      int comparison = newEntry.compareTo(currentEntry);

      if (comparison == 0)
      {  // newEntry matches currentEntry;
         // return and replace currentEntry
         found = true;
         result = currentEntry;
         currentNode.setData(newEntry);
      }
      else if (comparison < 0)
      {
         if (currentNode.hasLeftChild())
            currentNode = currentNode.getLeftChild();
         else
         {
            found = true;
            currentNode.setLeftChild(new BinaryNode<T>(newEntry));
         } // end if
      }
      else
      {
         assert comparison > 0;

         if (currentNode.hasRightChild())
            currentNode = currentNode.getRightChild();
         else
         {
            found = true;
            currentNode.setRightChild(new BinaryNode<T>(newEntry));
         } // end if
      } // end if
   } // end while

   return result;
} // end addEntry
```



Add an entry 11

Adding an Entry (Iterative Version)

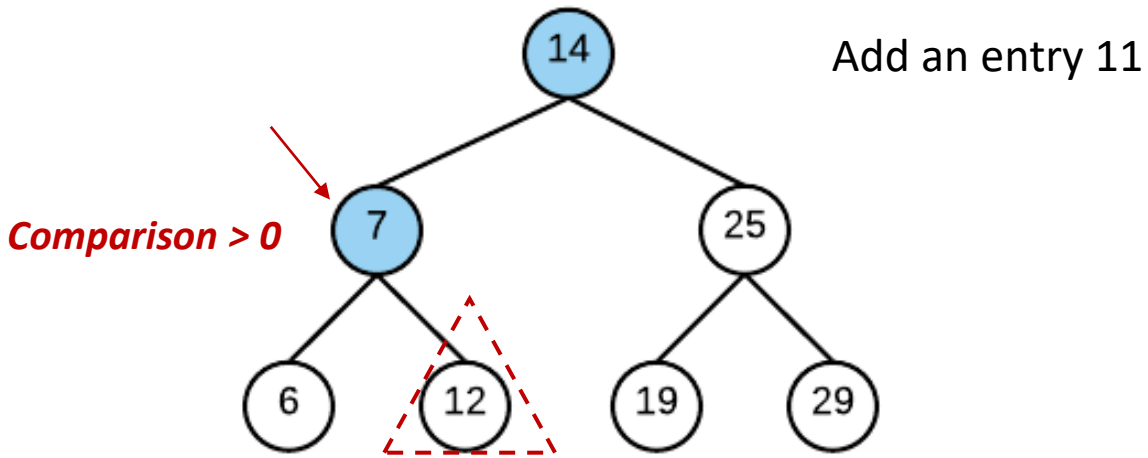
```
private T addEntry(T newEntry)
{
   BinaryNodeInterface<T> currentNode = getRootNode();
   assert currentNode != null;
   T result = null;
   boolean found = false;

   while (!found)
   {
      T currentEntry = currentNode.getData();
      int comparison = newEntry.compareTo(currentEntry);

      if (comparison == 0)
      {  // newEntry matches currentEntry;
         // return and replace currentEntry
         found = true;
         result = currentEntry;
         currentNode.setData(newEntry);
      }
      else if (comparison < 0)
      {
         if (currentNode.hasLeftChild())
            currentNode = currentNode.getLeftChild();
         else
         {
            found = true;
            currentNode.setLeftChild(new BinaryNode<T>(newEntry));
         } // end if
      }
      else
      {
         assert comparison > 0;

         if (currentNode.hasRightChild())
            currentNode = currentNode.getRightChild();
         else
         {
            found = true;
            currentNode.setRightChild(new BinaryNode<T>(newEntry));
         } // end if
      } // end if
   } // end while

   return result;
} // end addEntry
```



Add an entry 11

Adding an Entry (Iterative Version)

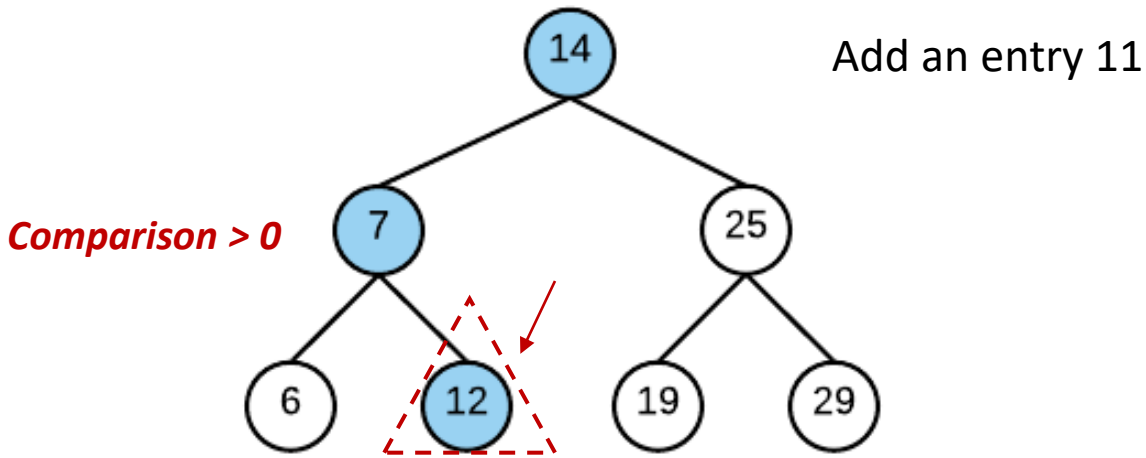
```
private T addEntry(T newEntry)
{
   BinaryNodeInterface<T> currentNode = getRootNode();
   assert currentNode != null;
   T result = null;
   boolean found = false;
   while (!found)
   {
      T currentEntry = currentNode.getData();
      int comparison = newEntry.compareTo(currentEntry);

      if (comparison == 0)
      {  // newEntry matches currentEntry;
         // return and replace currentEntry
         found = true;
         result = currentEntry;
         currentNode.setData(newEntry);
      }
      else if (comparison < 0)
      {
         if (currentNode.hasLeftChild())
            currentNode = currentNode.getLeftChild();
         else
         {
            found = true;
            currentNode.setLeftChild(new BinaryNode<T>(newEntry));
         } // end if
      }
      else
      {
         assert comparison > 0;

         if (currentNode.hasRightChild())
            currentNode = currentNode.getRightChild();
         else
         {
            found = true;
            currentNode.setRightChild(new BinaryNode<T>(newEntry));
         } // end if
      } // end if
   } // end while

   return result;
} // end addEntry
```



Add an entry 11

Adding an Entry (Iterative Version)

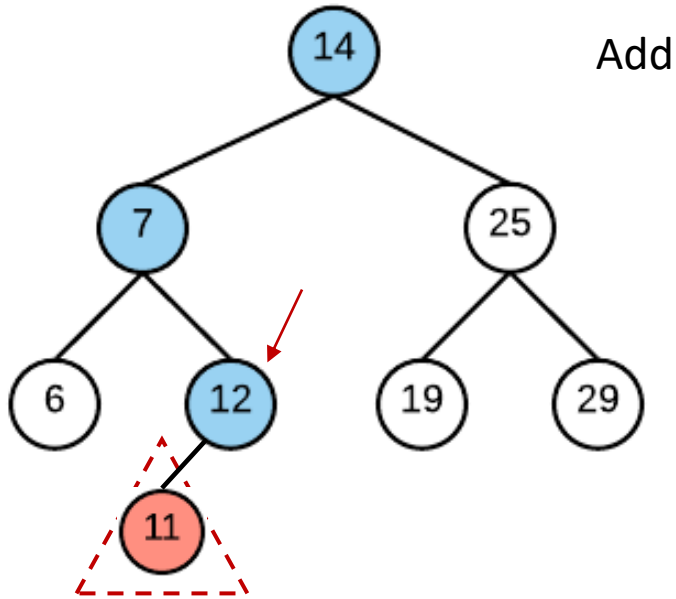
```
private T addEntry(T newEntry)
{
   BinaryNodeInterface<T> currentNode = getRootNode();
   assert currentNode != null;
   T result = null;
   boolean found = false;
   while (!found)
   {
      T currentEntry = currentNode.getData();
      int comparison = newEntry.compareTo(currentEntry);

      if (comparison == 0)
      {  // newEntry matches currentEntry;
         // return and replace currentEntry
         found = true;
         result = currentEntry;
         currentNode.setData(newEntry);
      }
      else if (comparison < 0)
      {
         if (currentNode.hasLeftChild())
            currentNode = currentNode.getLeftChild();
         else
         {
            found = true;
            currentNode.setLeftChild(new BinaryNode<T>(newEntry));
         } // end if
      }
      else
      {
         assert comparison > 0;

         if (currentNode.hasRightChild())
            currentNode = currentNode.getRightChild();
         else
         {
            found = true;
            currentNode.setRightChild(new BinaryNode<T>(newEntry));
         } // end if
      } // end if
   } // end while

   return result;
} // end addEntry
```



Adding an Entry (Iterative Version)

```
private T addEntry(T newEntry)
{
   BinaryNodeInterface<T> currentNode = getRootNode();
   assert currentNode != null;
   T result = null;
   boolean found = false;

   while (!found)
   {
      T currentEntry = currentNode.getData();
      int comparison = newEntry.compareTo(currentEntry);

      if (comparison == 0)
      {  // newEntry matches currentEntry;
         // return and replace currentEntry
         found = true;
         result = currentEntry;
         currentNode.setData(newEntry);
      }
      else if (comparison < 0)
      {
         if (currentNode.hasLeftChild())
            currentNode = currentNode.getLeftChild();
         else
         {
            found = true;
            currentNode.setLeftChild(new BinaryNode<T>(newEntry));
         } // end if
      }
      else
      {
         assert comparison > 0;

         if (currentNode.hasRightChild())
            currentNode = currentNode.getRightChild();
         else
         {
            found = true;
            currentNode.setRightChild(new BinaryNode<T>(newEntry));
         } // end if
      } // end if
   } // end while

   return result;
} // end addEntry
```

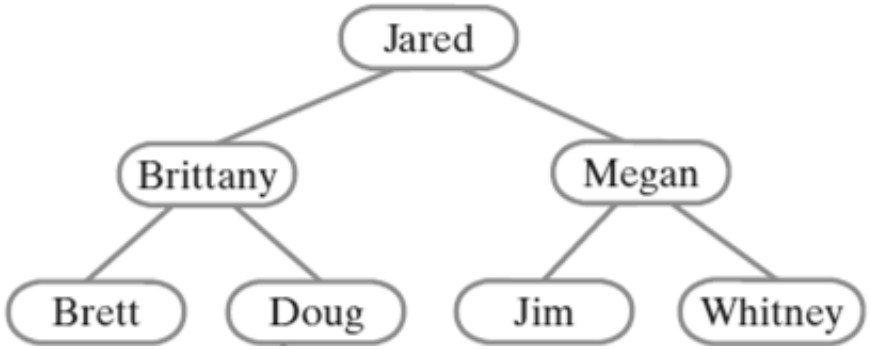
Comparison < 0



Add an entry 11

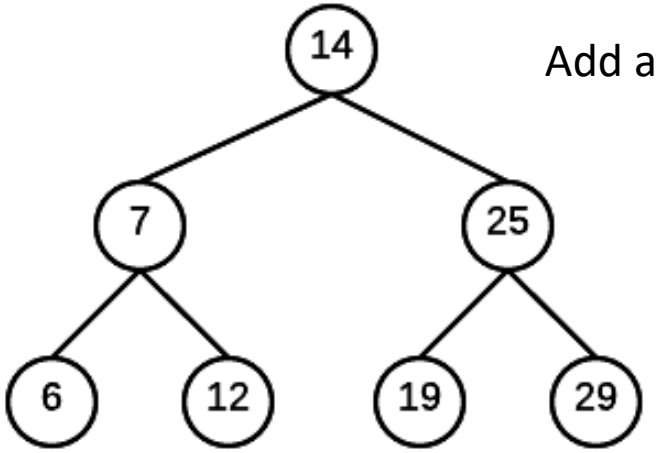
In-Class Exercise

- Add the names Chad, Chris, Jason, and Kelley to the binary search tree below.



In-Class Exercise

- Adding an Entry **(Recursive Version)**
- Complete the implementation to the right.

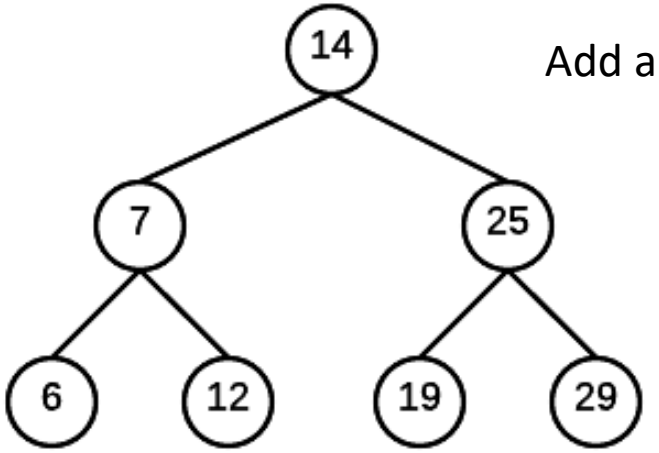


Add an entry 11

```
public T add(T newEntry)
{
   T result = null;

   if (isEmpty())
      setRootNode(new BinaryNode<T>(newEntry));
   else
      result = addEntry(getRootNode(), newEntry);

   return result;
} // end add
// Adds newEntry to the nonempty subtree rooted at rootNode.
private T addEntry(BinaryNodeInterface<T> rootNode, T newEntry)
{
   assert rootNode != null;
   T result = null;
   int comparison = newEntry.compareTo(rootNode.getData());

   if (comparison == 0)
   {
      result = rootNode.getData();
      rootNode.setData(newEntry);
   }
   else if (comparison < 0)
   {
      if (rootNode.hasLeftChild())

      else

   }
   else
   {
      assert comparison > 0;

      if (rootNode.hasRightChild())

      else

   } // end if

   return result;
} // end addEntry
```

In-Class Exercise

- Adding an Entry **(Recursive Version)**
- Complete the implementation to the right.

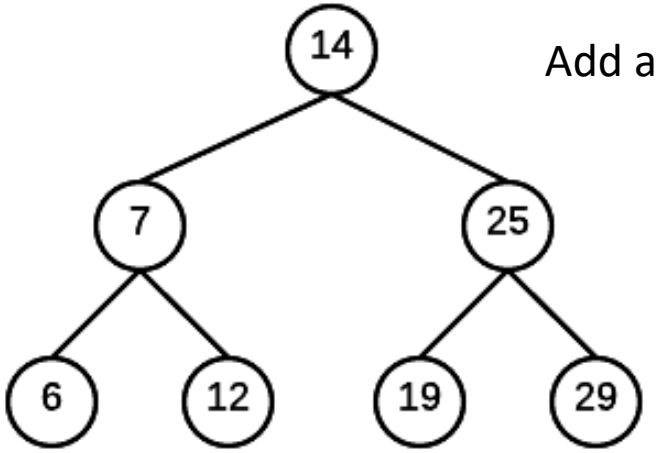


Add an entry 11

```
public T add(T newEntry)
{
   T result = null;

   if (isEmpty())
      setRootNode(new BinaryNode<T>(newEntry));
   else
      result = addEntry(getRootNode(), newEntry);

   return result;
} // end add
// Adds newEntry to the nonempty subtree rooted at rootNode.
private T addEntry(BinaryNodeInterface<T> rootNode, T newEntry)
{
   assert rootNode != null;
   T result = null;
   int comparison = newEntry.compareTo(rootNode.getData());

   if (comparison == 0)
   {
      result = rootNode.getData();
      rootNode.setData(newEntry);
   }
   else if (comparison < 0)
   {
      if (rootNode.hasLeftChild())
         result = addEntry(rootNode.getLeftChild(), newEntry);
      else

   }
   else
   {
      assert comparison > 0;

      if (rootNode.hasRightChild())

      else

   } // end if

   return result;
} // end addEntry
```

In-Class Exercise

- Adding an Entry **(Recursive Version)**
- Complete the implementation to the right.

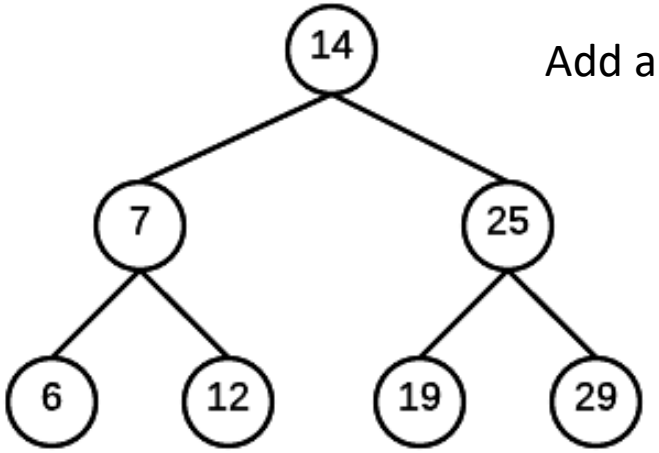


Add an entry 11

```
public T add(T newEntry)
{
   T result = null;

   if (isEmpty())
      setRootNode(new BinaryNode<T>(newEntry));
   else
      result = addEntry(getRootNode(), newEntry);

   return result;
} // end add
// Adds newEntry to the nonempty subtree rooted at rootNode.
private T addEntry(BinaryNodeInterface<T> rootNode, T newEntry)
{
   assert rootNode != null;
   T result = null;
   int comparison = newEntry.compareTo(rootNode.getData());

   if (comparison == 0)
   {
      result = rootNode.getData();
      rootNode.setData(newEntry);
   }
   else if (comparison < 0)
   {
      if (rootNode.hasLeftChild())
         result = addEntry(rootNode.getLeftChild(), newEntry);
      else
         rootNode.setLeftChild(new BinaryNode<T>(newEntry));
   }
   else
   {
      assert comparison > 0;

      if (rootNode.hasRightChild())

      else

   } // end if

   return result;
} // end addEntry
```

In-Class Exercise

- Adding an Entry **(Recursive Version)**
- Complete the implementation to the right.

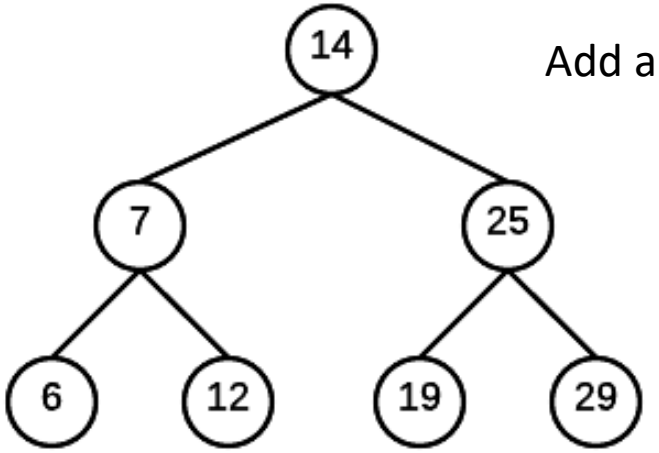


Add an entry 11

```
public T add(T newEntry)
{
   T result = null;

   if (isEmpty())
      setRootNode(new BinaryNode<T>(newEntry));
   else
      result = addEntry(getRootNode(), newEntry);

   return result;
} // end add
// Adds newEntry to the nonempty subtree rooted at rootNode.
private T addEntry(BinaryNodeInterface<T> rootNode, T newEntry)
{
   assert rootNode != null;
   T result = null;
   int comparison = newEntry.compareTo(rootNode.getData());

   if (comparison == 0)
   {
      result = rootNode.getData();
      rootNode.setData(newEntry);
   }
   else if (comparison < 0)
   {
      if (rootNode.hasLeftChild())
         result = addEntry(rootNode.getLeftChild(), newEntry);
      else
         rootNode.setLeftChild(new BinaryNode<T>(newEntry));
   }
   else
   {
      assert comparison > 0;

      if (rootNode.hasRightChild())
         result = addEntry(rootNode.getRightChild(), newEntry);
      else

   } // end if

   return result;
} // end addEntry
```

In-Class Exercise

- Adding an Entry **(Recursive Version)**
- Complete the implementation to the right.

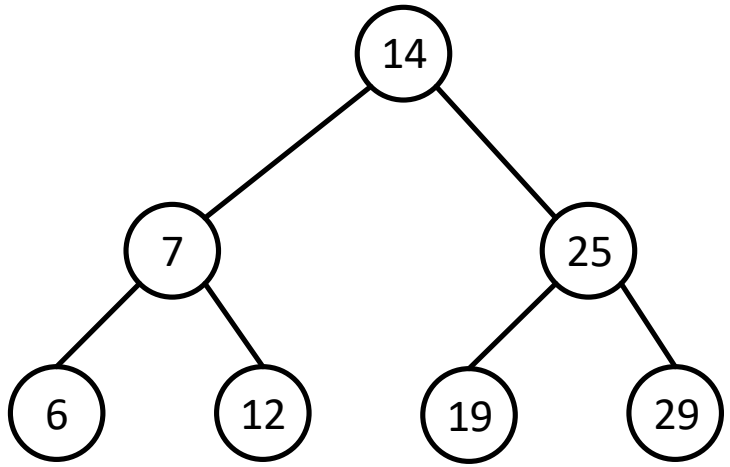


Add an entry 11

```
public T add(T newEntry)
{
   T result = null;

   if (isEmpty())
      setRootNode(new BinaryNode<T>(newEntry));
   else
      result = addEntry(getRootNode(), newEntry);

   return result;
} // end add
// Adds newEntry to the nonempty subtree rooted at rootNode.
private T addEntry(BinaryNodeInterface<T> rootNode, T newEntry)
{
   assert rootNode != null;
   T result = null;
   int comparison = newEntry.compareTo(rootNode.getData());

   if (comparison == 0)
   {
      result = rootNode.getData();
      rootNode.setData(newEntry);
   }
   else if (comparison < 0)
   {
      if (rootNode.hasLeftChild())
         result = addEntry(rootNode.getLeftChild(), newEntry);
      else
         rootNode.setLeftChild(new BinaryNode<T>(newEntry));
   }
   else
   {
      assert comparison > 0;

      if (rootNode.hasRightChild())
         result = addEntry(rootNode.getRightChild(), newEntry);
      else
         rootNode.setRightChild(new BinaryNode<T>(newEntry));
   } // end if

   return result;
} // end addEntry
```

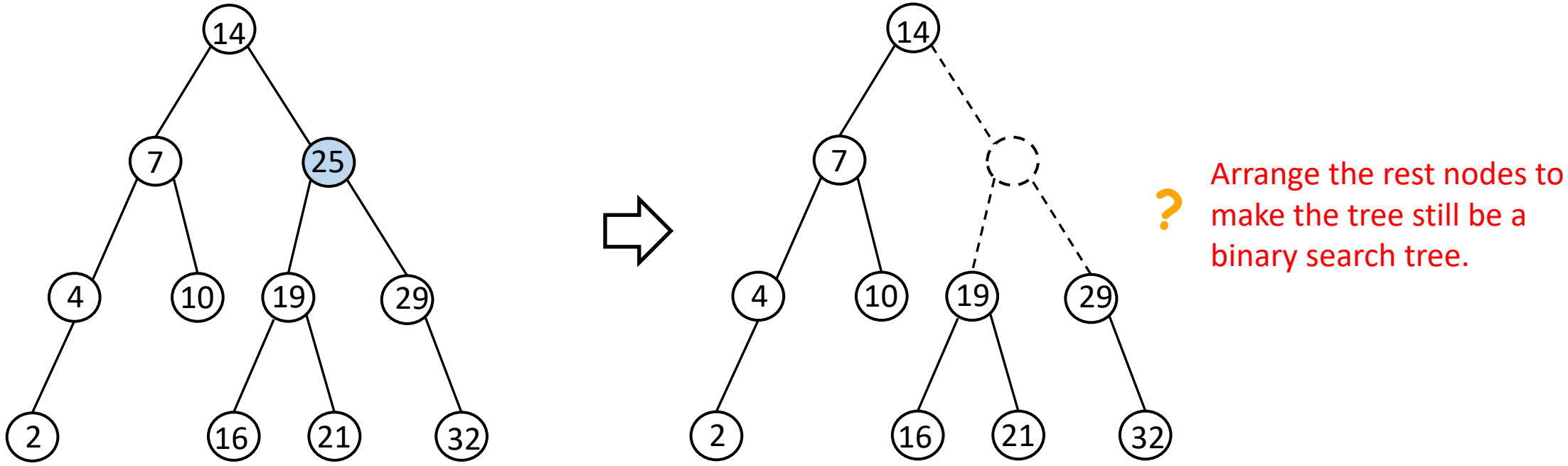
Definition of Binary Search Tree

- A Binary Search Tree (BST)
  - A Binary Tree
  - The nodes in BST stay sorted that
    - **Left is less (or equal to)**
    - **Right is greater**
    - **Follow the rules all the way down**



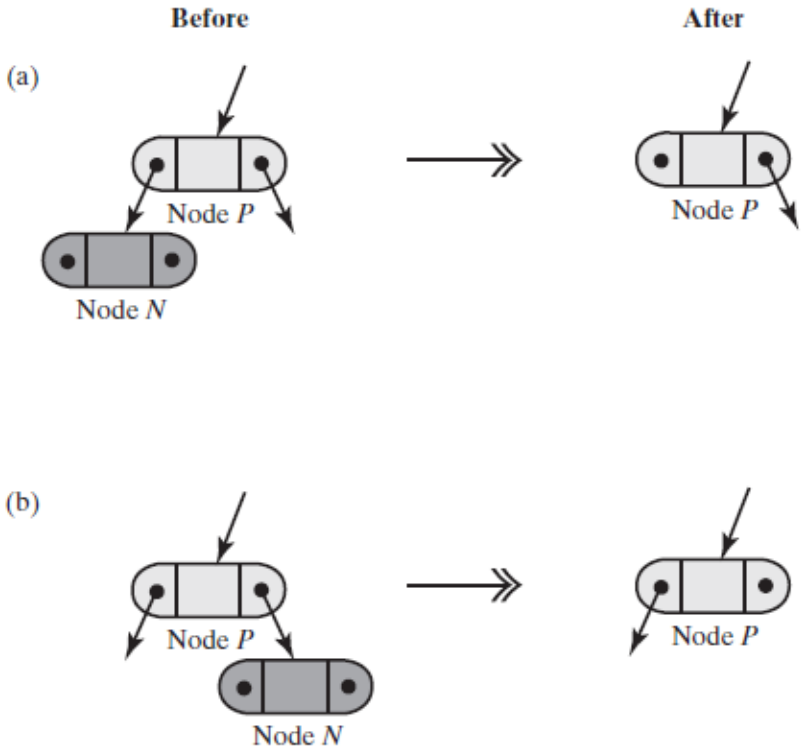
Removing an Entry

- Remove an entry from a binary search tree
  - The entry matched is removed from the tree and returned to the client.
  - If no such entry exists, returns null and the tree remain unchanged.



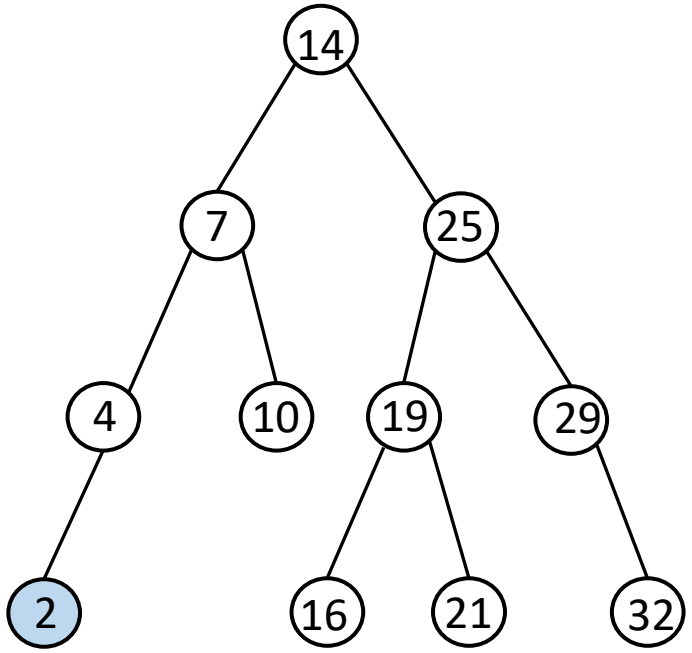
Removing an Entry

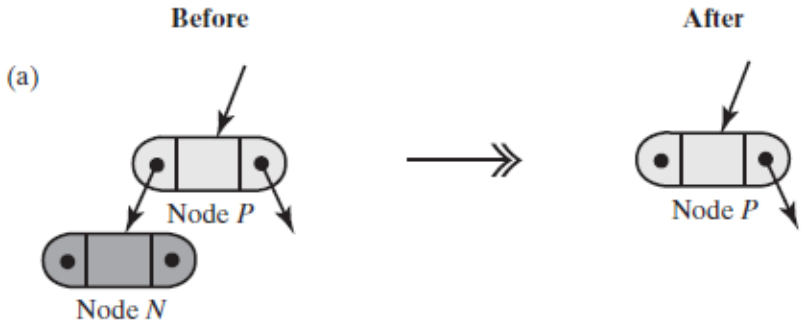
- **Case 1: Removing an entry whose node is a leaf**
  - Since N is a leaf, we can delete it by setting the appropriate child reference in node P to null

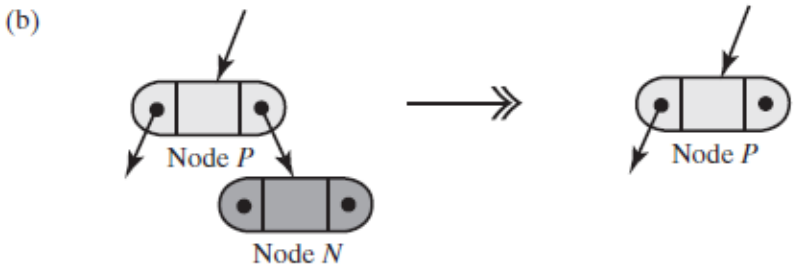


Removing an Entry

- **Case 1: Removing an entry whose node is a leaf**
  - Since N is a leaf, we can delete it by setting the appropriate child reference in node P to null

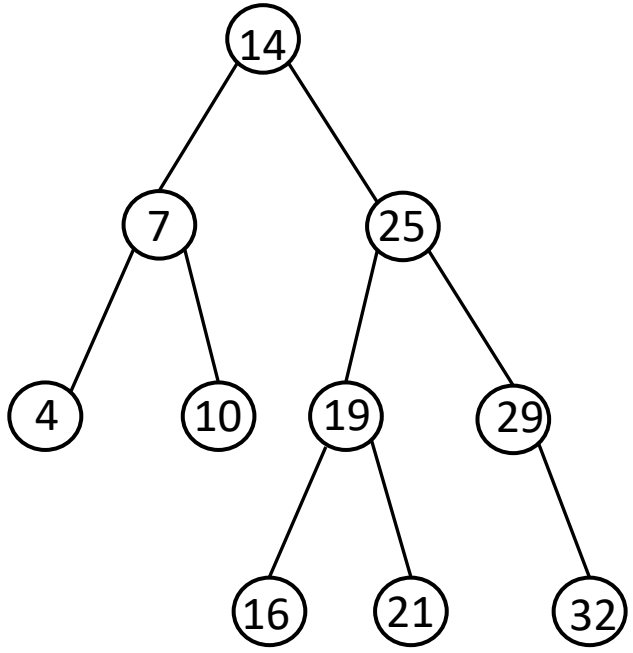


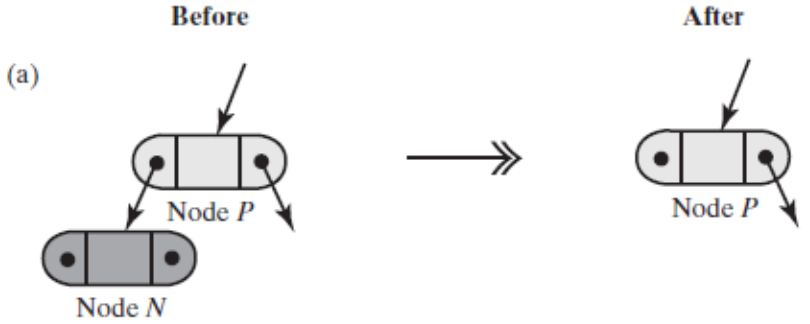


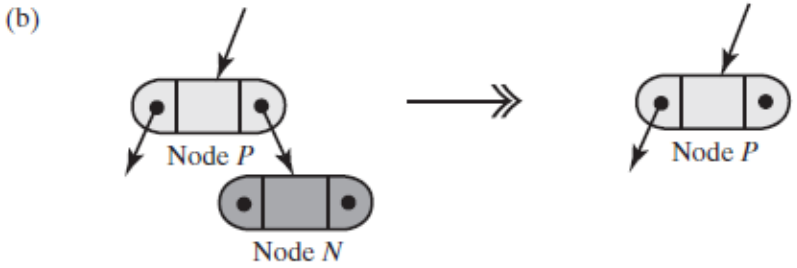


Removing an Entry

- **Case 1: Removing an entry whose node is a leaf**
  - Since N is a leaf, we can delete it by setting the appropriate child reference in node P to null

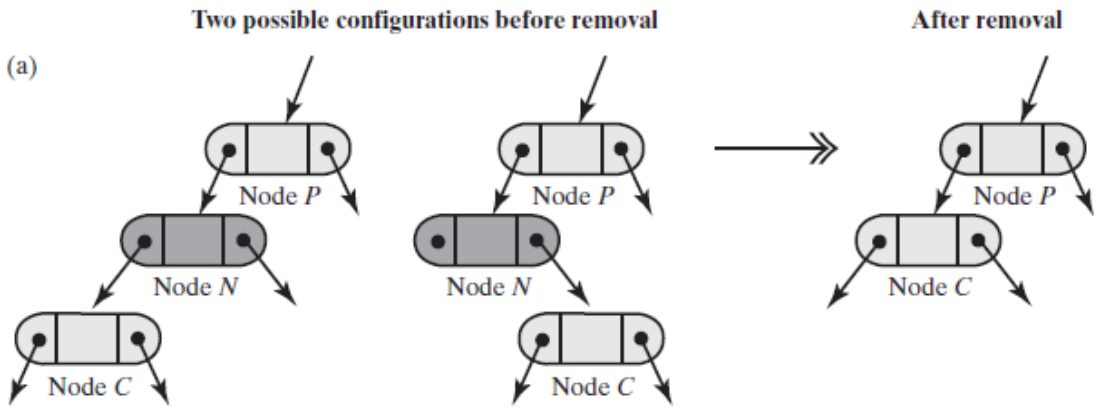






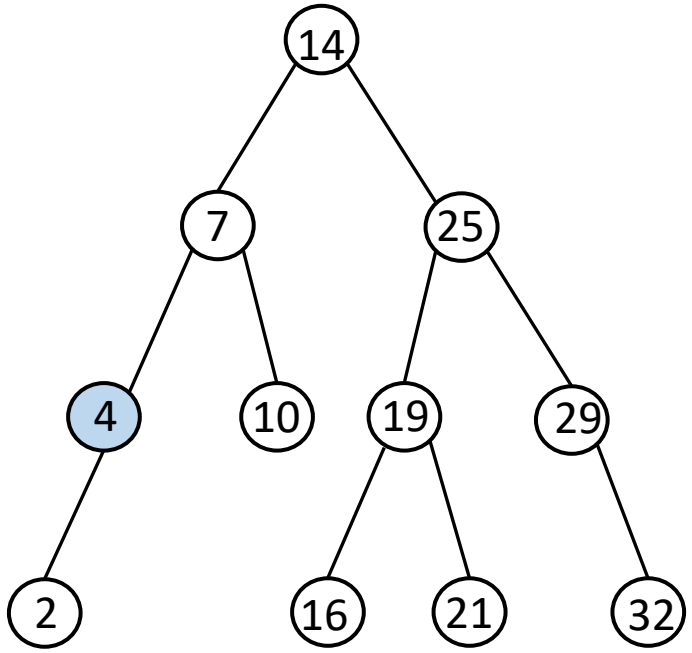
Removing an Entry

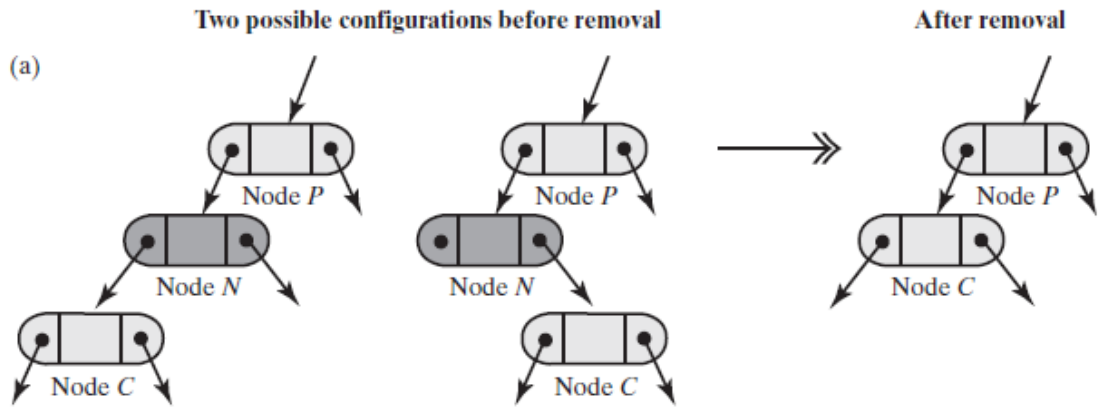
- **Case 2: Removing an entry whose node has one child**
  - To remove the entry in *N*, we remove *N* from the tree. We do this by making *C* a child of *P* instead of *N*



Removing an Entry

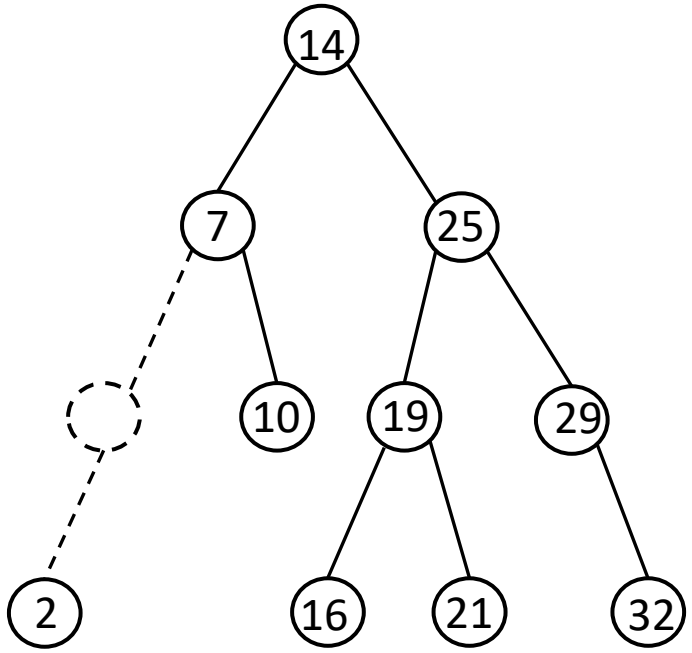
- **Case 2: Removing an entry whose node has one child**
  - To remove the entry in *N*, we remove *N* from the tree. We do this by making *C* a child of *P* instead of *N*

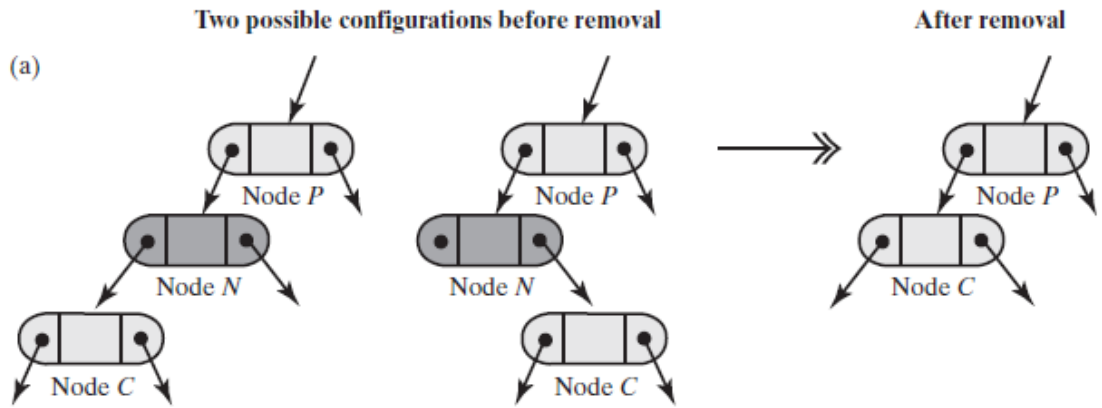




Removing an Entry

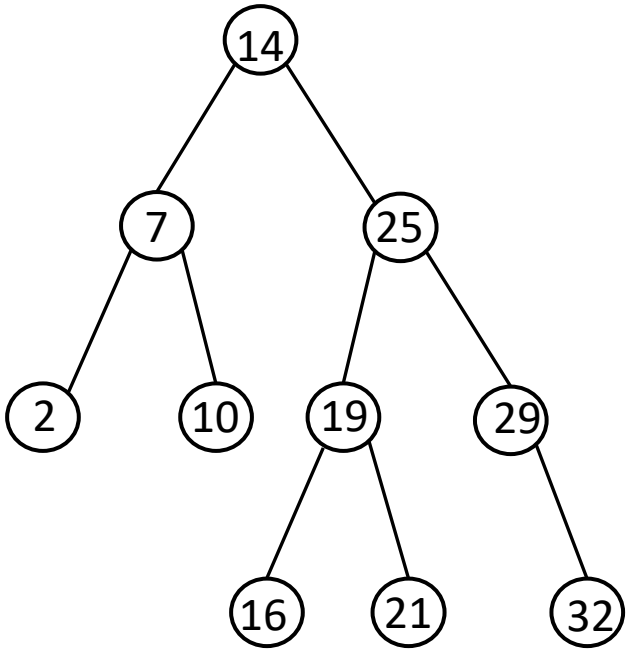
- **Case 2: Removing an entry whose node has one child**
  - To remove the entry in *N*, we remove *N* from the tree. We do this by making *C* a child of *P* instead of *N*

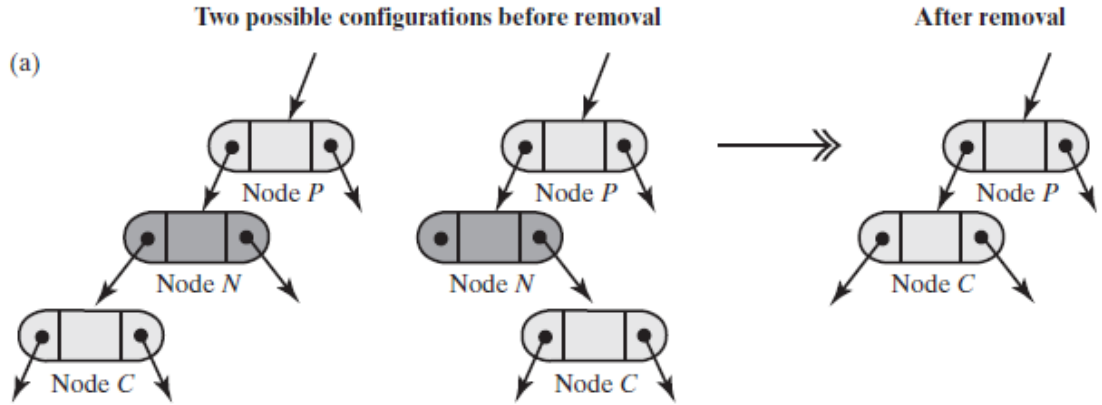




Removing an Entry

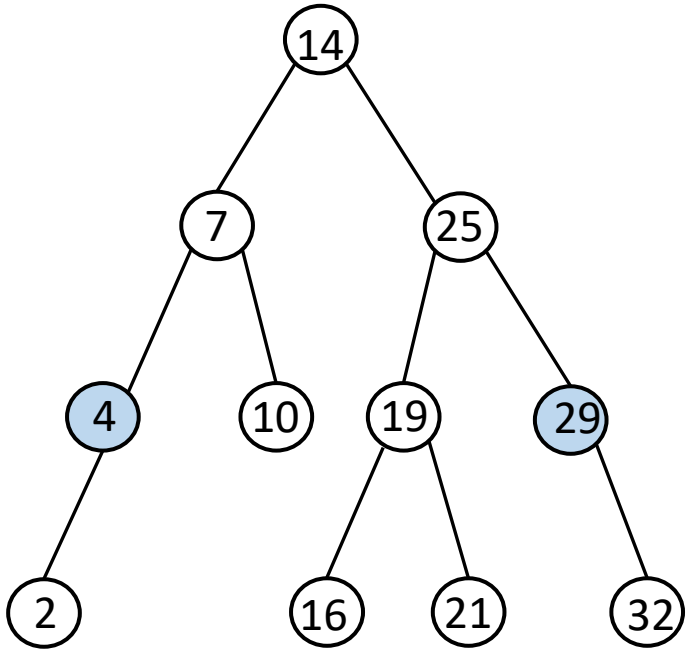
- **Case 2: Removing an entry whose node has one child**
  - To remove the entry in *N*, we remove *N* from the tree. We do this by making *C* a child of *P* instead of *N*

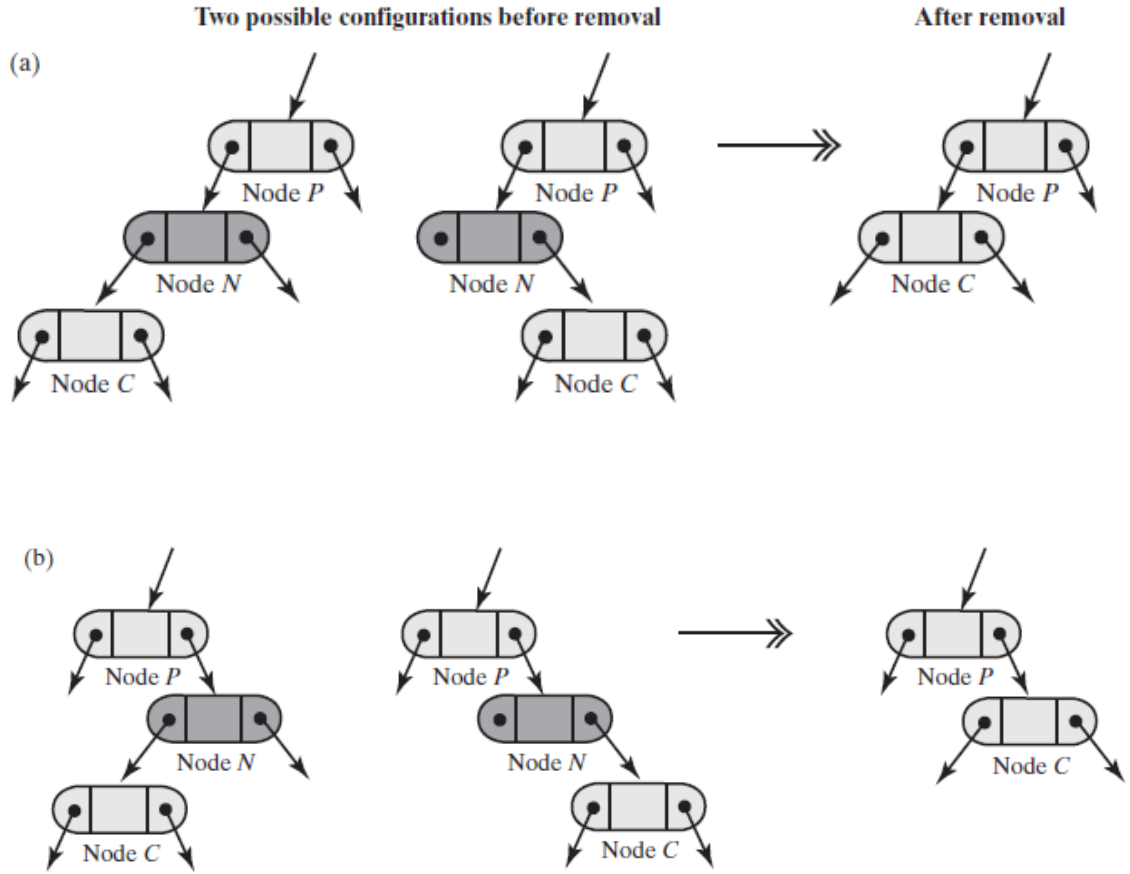




Removing an Entry

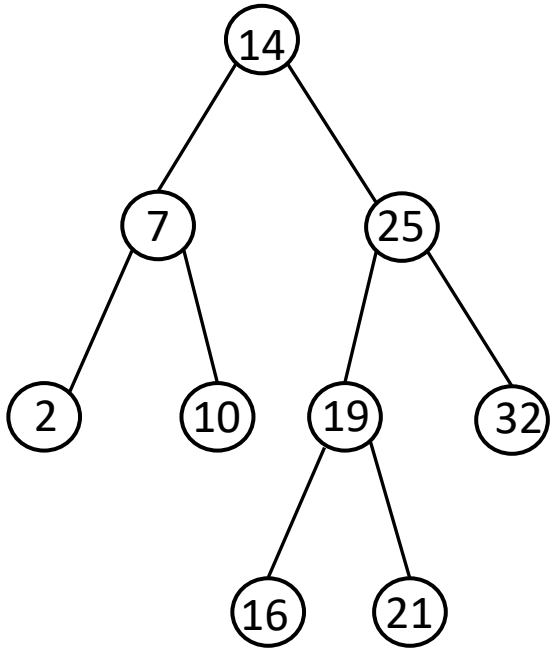
- **Case 2: Removing an entry whose node has one child**
  - To remove the entry in *N*, we remove *N* from the tree. We do this by making *C* a child of *P* instead of *N*

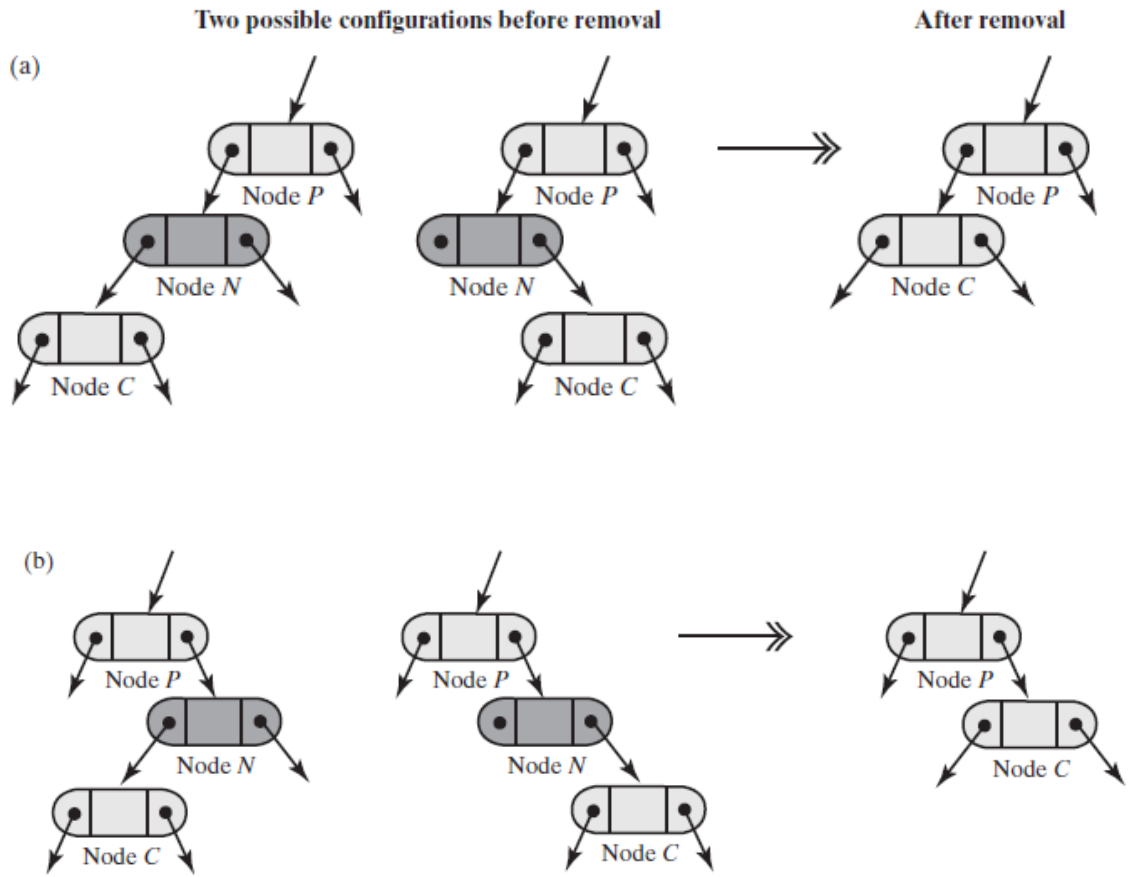




Removing an Entry

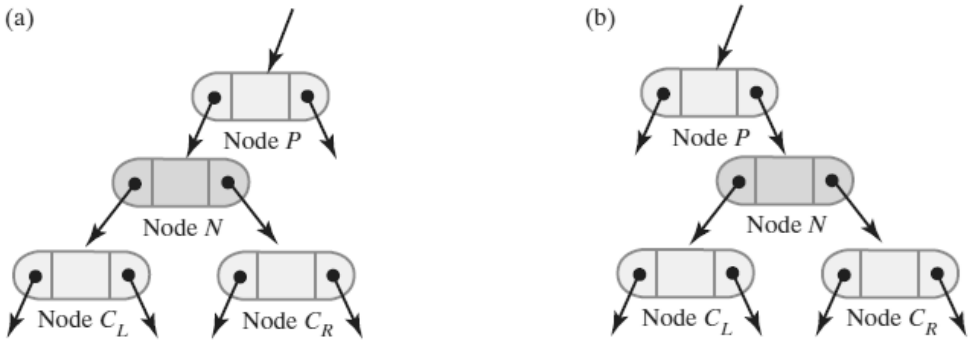
- **Case 2: Removing an entry whose node has one child**
  - To remove the entry in *N*, we remove *N* from the tree. We do this by making *C* a child of *P* instead of *N*





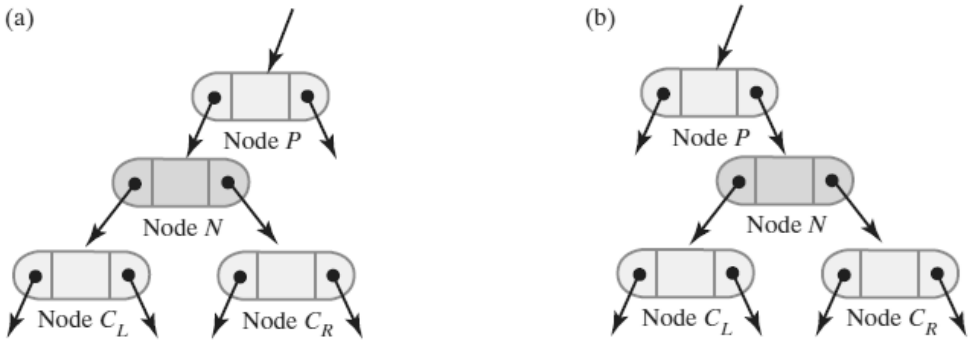
Removing an Entry

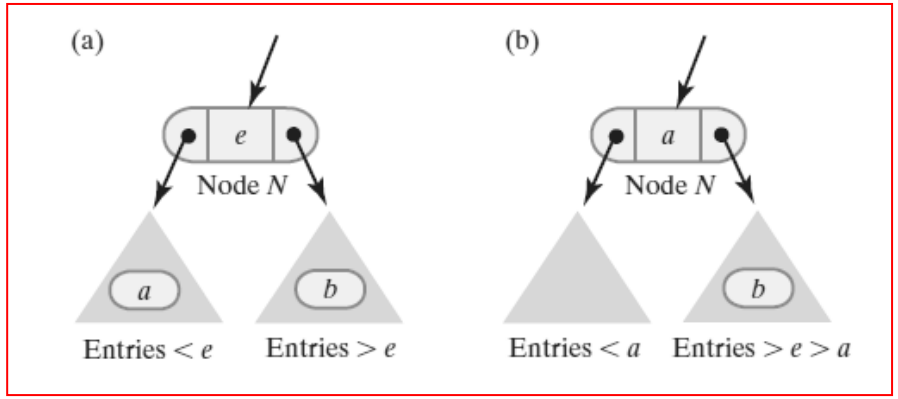
- **Case 3: Removing an entry whose node has two children**
  - Find the rightmost node R in N's left subtree
  - Replace the entry in node N with the entry that is in node R
  - Delete node R



Removing an Entry

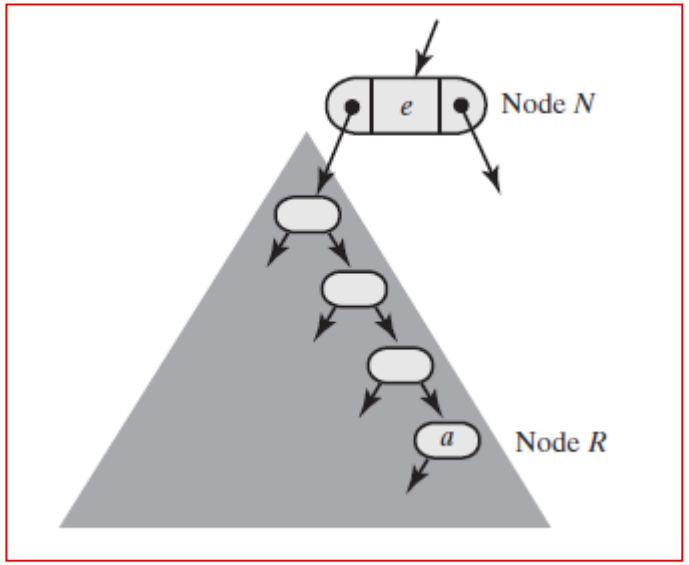
- **Case 3: Removing an entry whose node has two children**
  - Find the rightmost node R in N's left subtree
  - Replace the entry in node N with the entry that is in node R
  - Delete node R

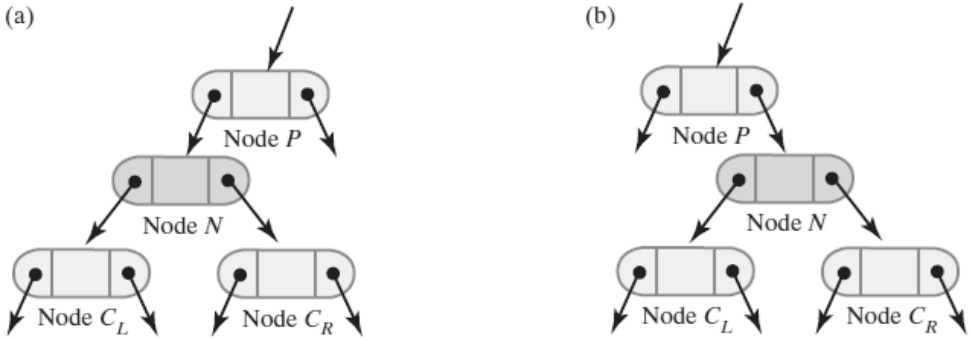


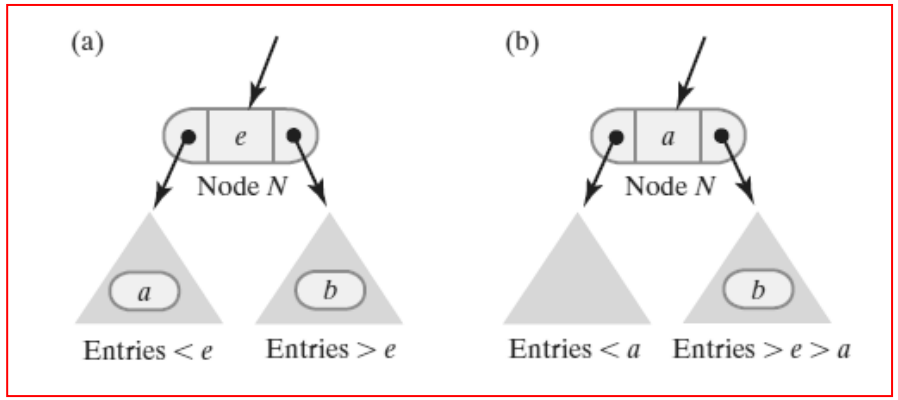


Removing an Entry

- **Case 3: Removing an entry whose node has two children**
  - Find the rightmost node R in N's left subtree
  - Replace the entry in node N with the entry that is in node R
  - Delete node R

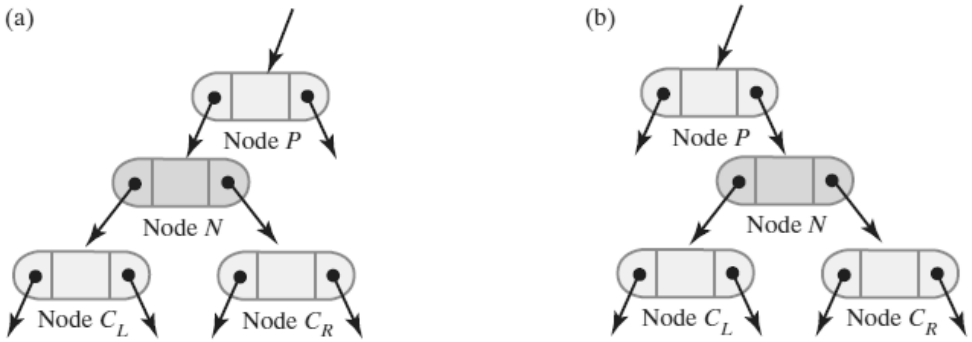


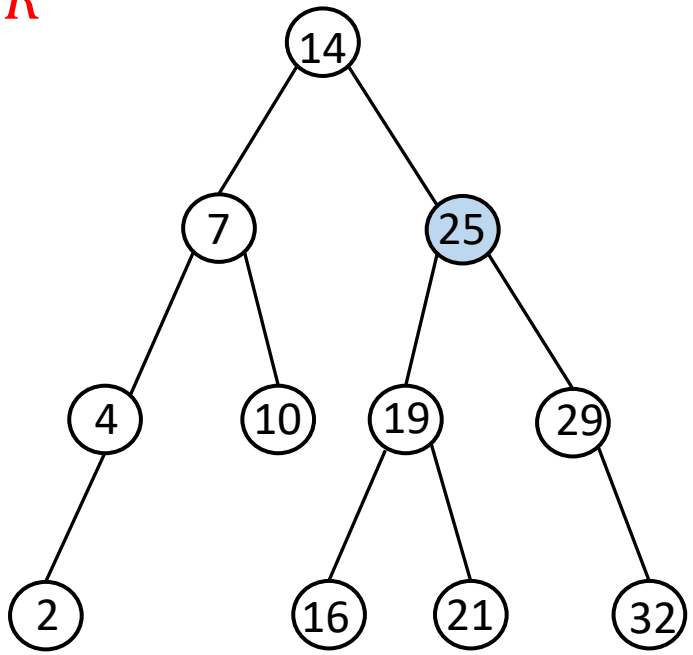


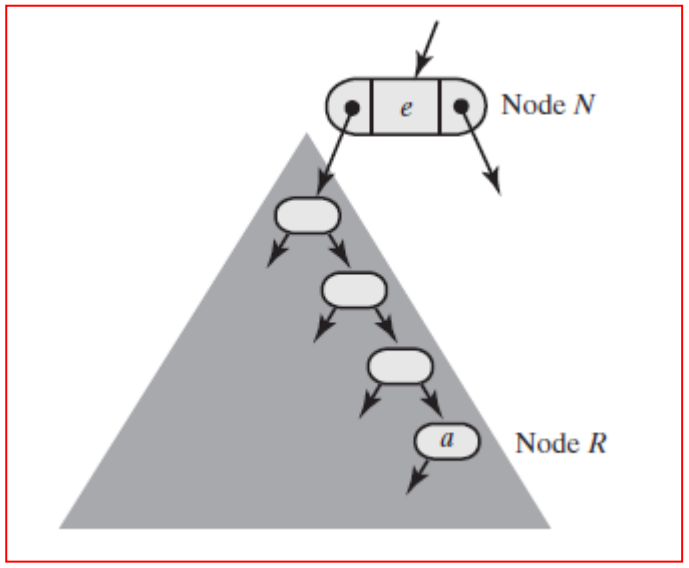


Removing an Entry

- **Case 3: Removing an entry whose node has two children**
  - Find the rightmost node R in N's left subtree
  - Replace the entry in node N with the entry that is in node R
  - Delete node R

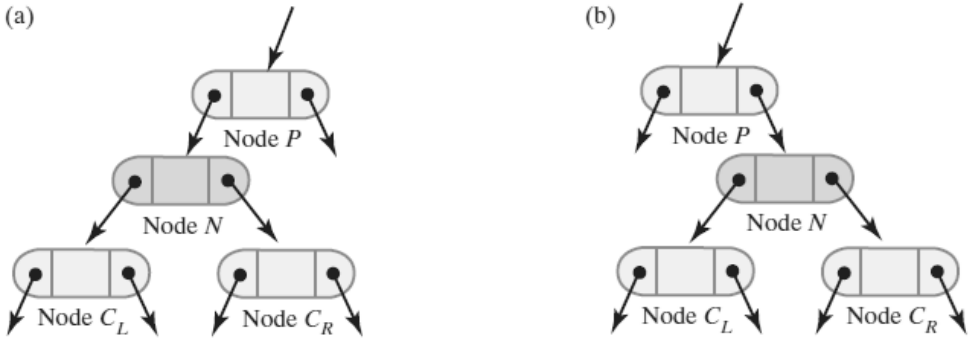


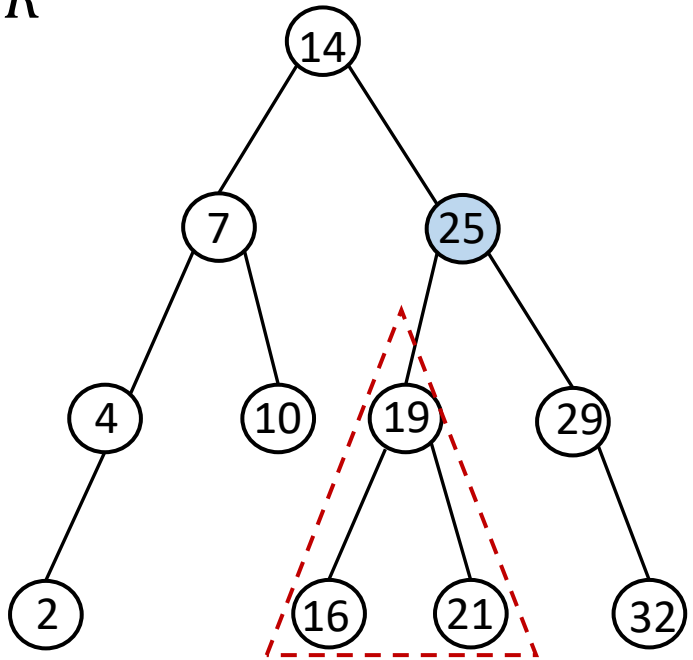


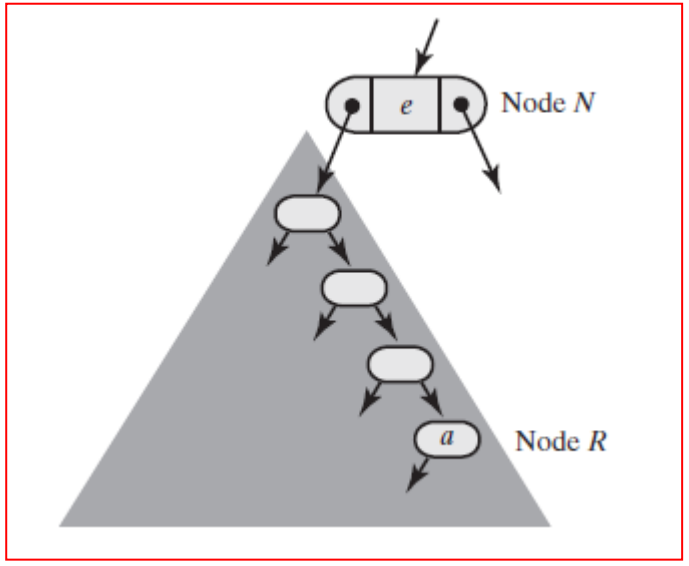


Removing an Entry

- **Case 3: Removing an entry whose node has two children**
  - Find the rightmost node R in N's left subtree
  - Replace the entry in node N with the entry that is in node R
  - Delete node R

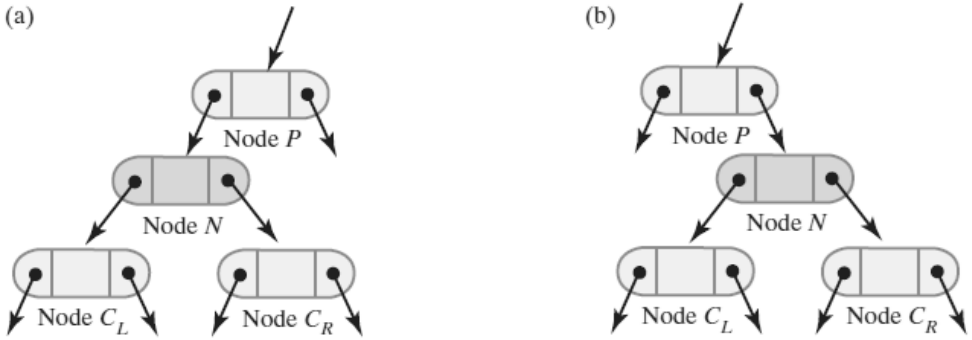


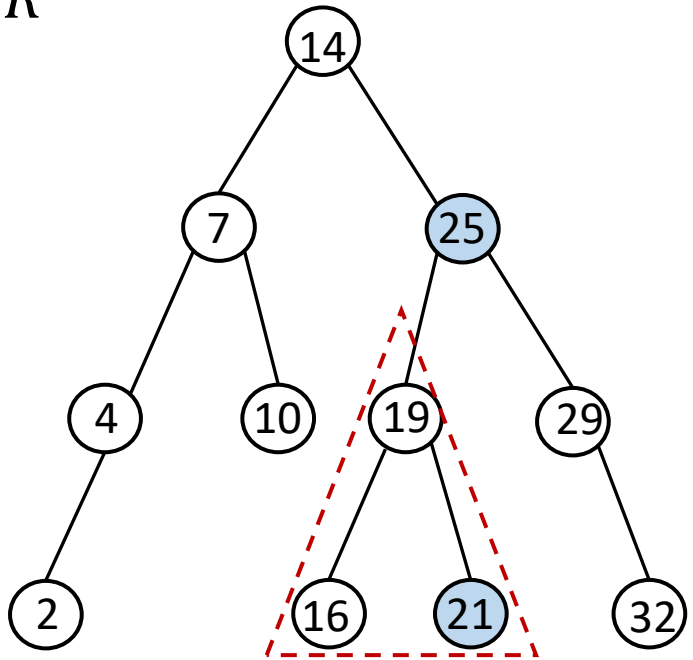


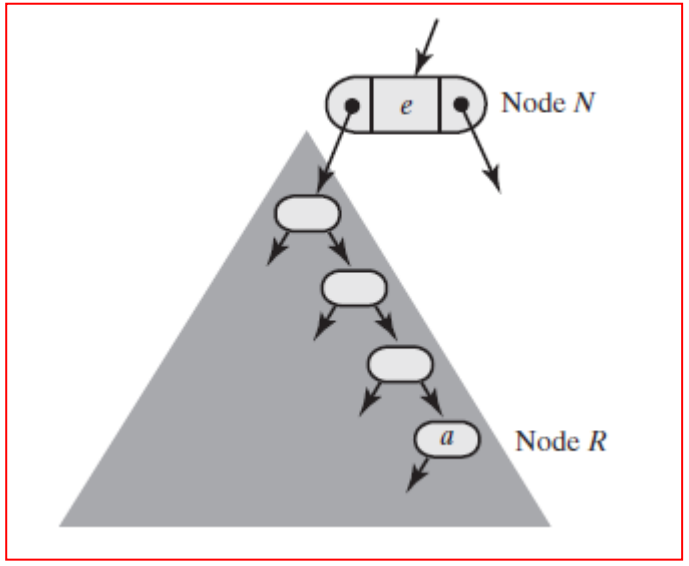


Removing an Entry

- **Case 3: Removing an entry whose node has two children**
  - Find the rightmost node R in N's left subtree
  - Replace the entry in node N with the entry that is in node R
  - Delete node R

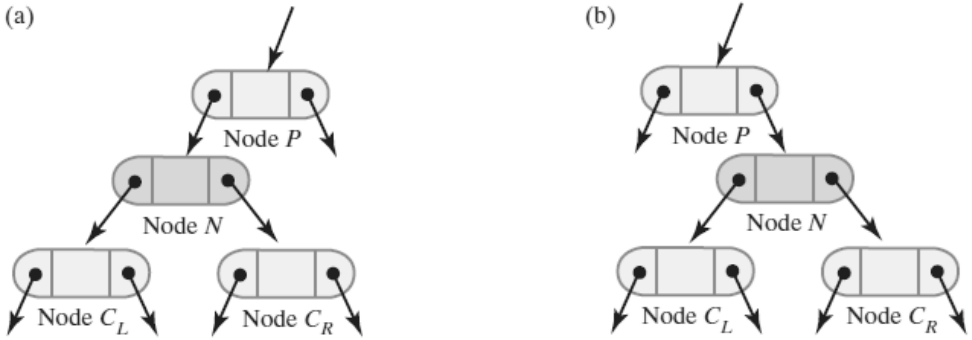


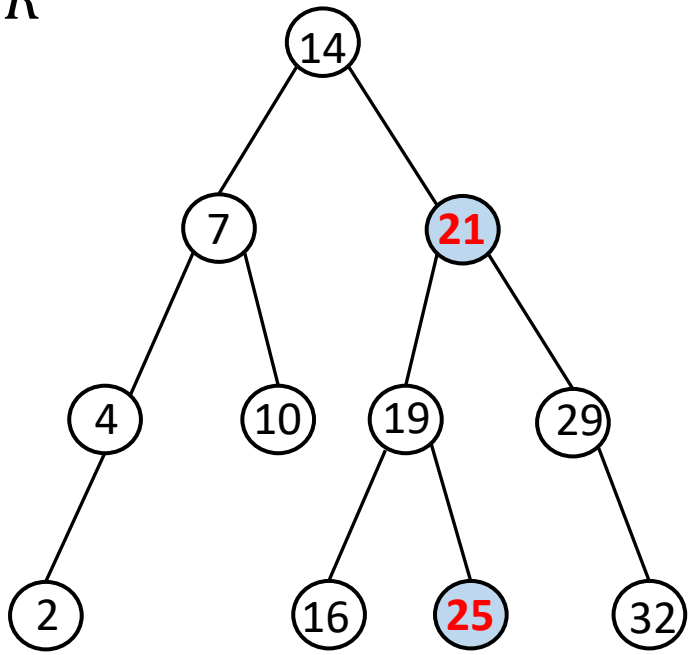


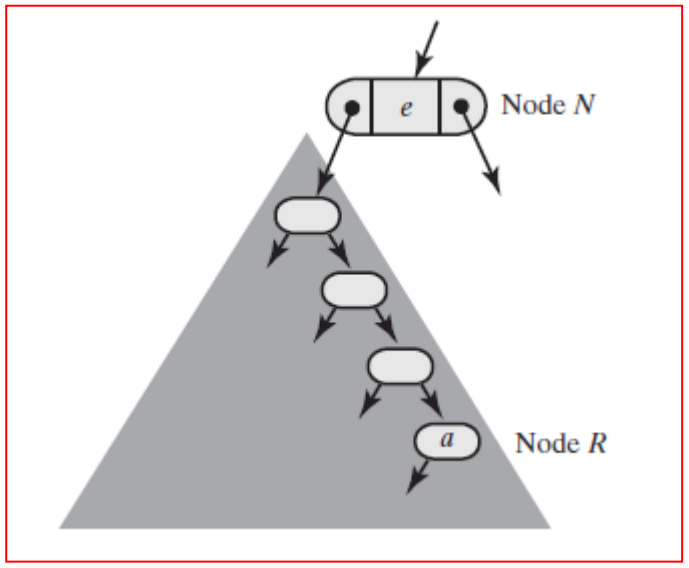


Removing an Entry

- **Case 3: Removing an entry whose node has two children**
  - Find the rightmost node R in N's left subtree
  - Replace the entry in node N with the entry that is in node R
  - Delete node R

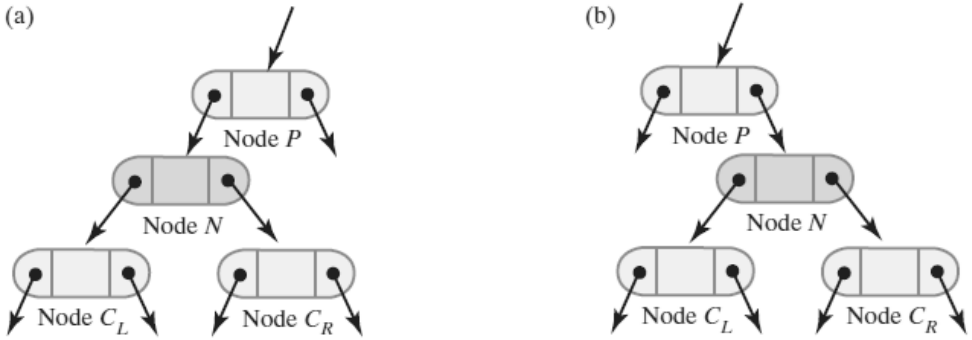


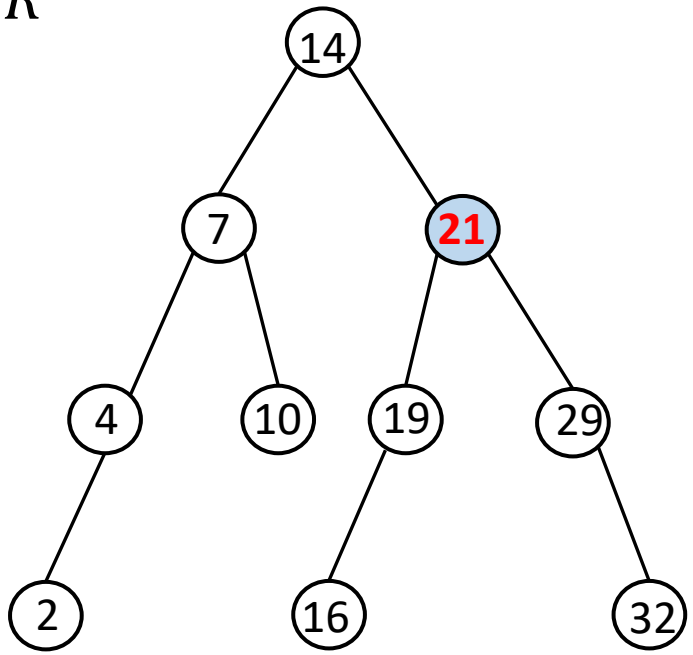


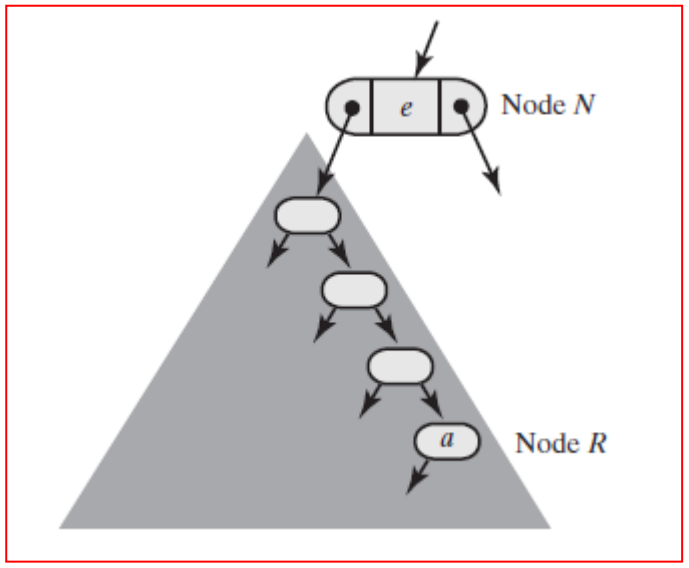


Removing an Entry

- **Case 3: Removing an entry whose node has two children**
  - Find the rightmost node R in N's left subtree
  - Replace the entry in node N with the entry that is in node R
  - Delete node R

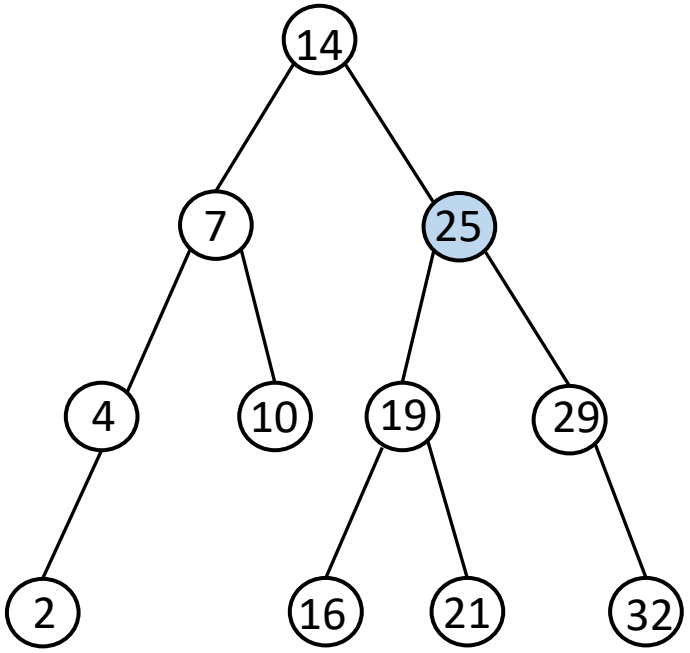






Removing an Entry

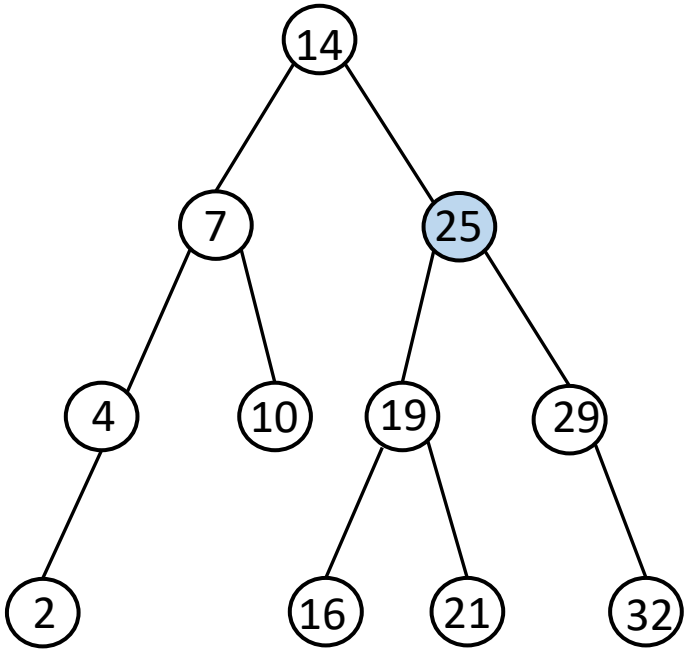
- When a node is removed, we normally replace it with rightmost (the largest) in its left subtree. **Do we have other possible option?**



Removing an Entry

- When a node is removed, we normally replace it with rightmost (the largest) in its left subtree. **Do we have other possible option?**

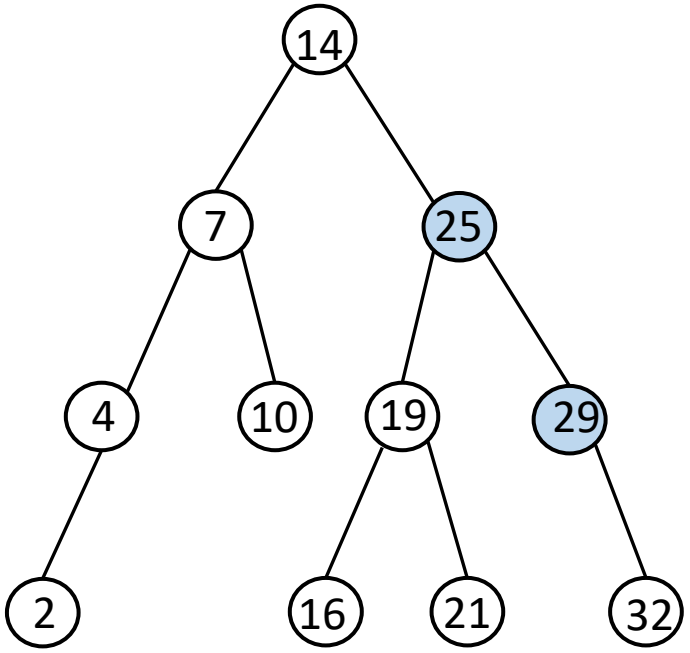
- **An alternate approach**
  - Find the leftmost node L in N's right subtree
  - Replace the entry in node N with the entry that is in node L
  - Delete node L



Removing an Entry

- When a node is removed, we normally replace it with rightmost (the largest) in its left subtree. **Do we have other possible option?**

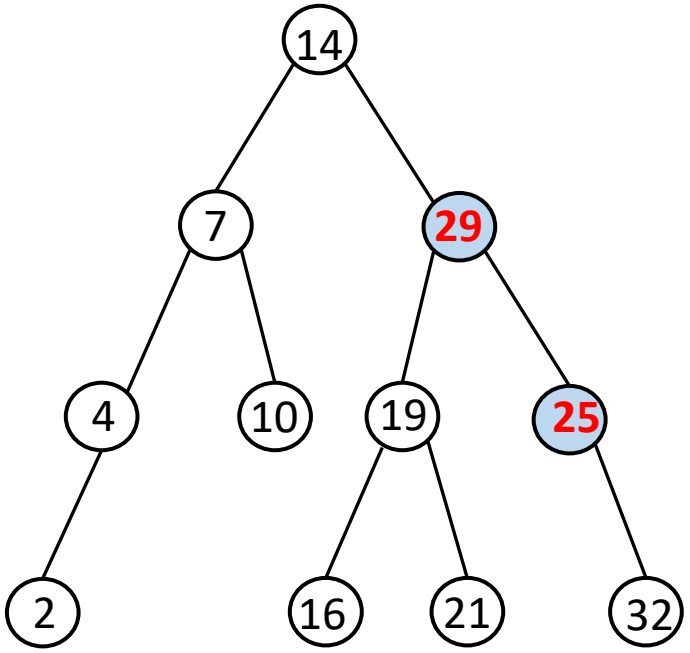
- **An alternate approach**
  - Find the leftmost node L in N's right subtree
  - Replace the entry in node N with the entry that is in node L
  - Delete node L



Removing an Entry

- When a node is removed, we normally replace it with rightmost (the largest) in its left subtree. **Do we have other possible option?**

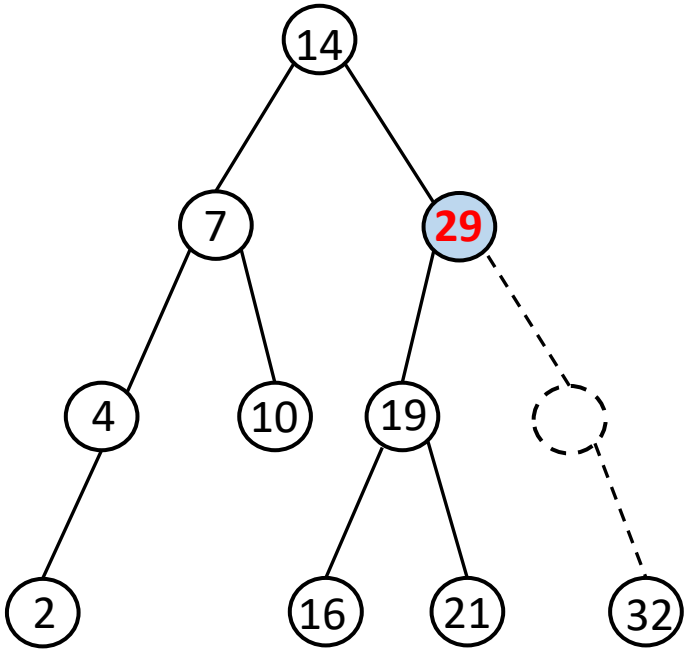
- **An alternate approach**
  - Find the leftmost node L in N's right subtree
  - Replace the entry in node N with the entry that is in node L
  - Delete node L



Removing an Entry

- When a node is removed, we normally replace it with rightmost (the largest) in its left subtree. **Do we have other possible option?**

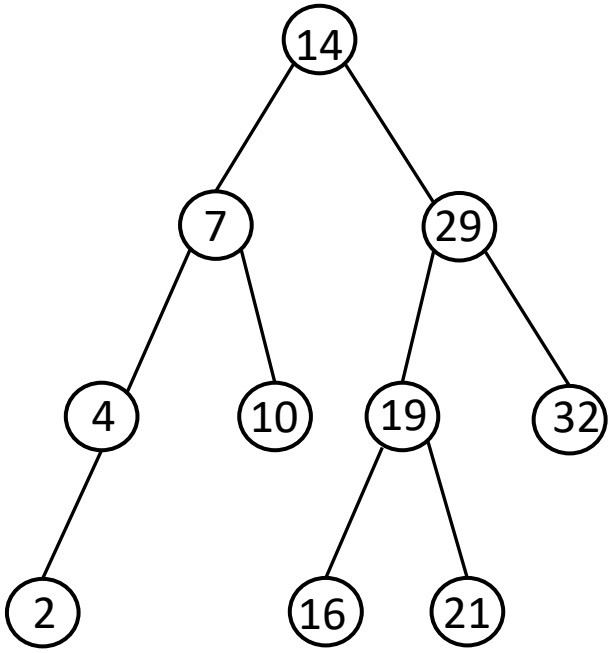
- **An alternate approach**
  - Find the leftmost node L in N's right subtree
  - Replace the entry in node N with the entry that is in node L
  - Delete node L



Removing an Entry

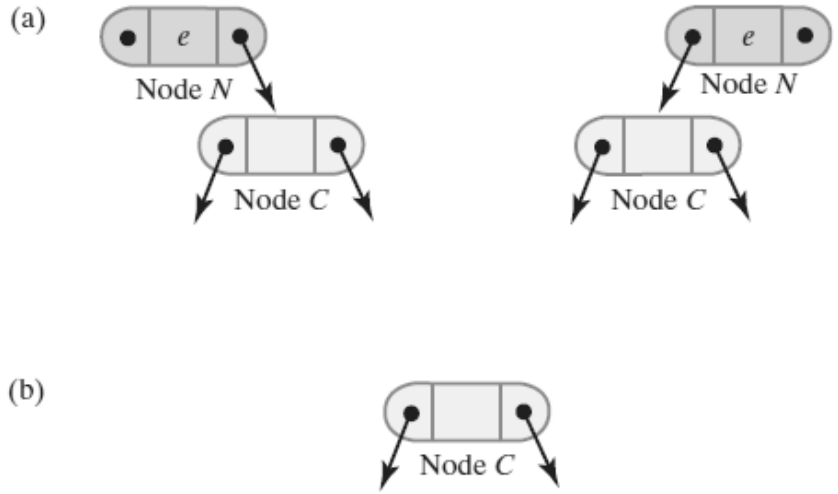
- When a node is removed, we normally replace it with rightmost (the largest) in its left subtree. **Do we have other possible option?**

- **An alternate approach**
  - Find the leftmost node L in N's right subtree
  - Replace the entry in node N with the entry that is in node L
  - Delete node L



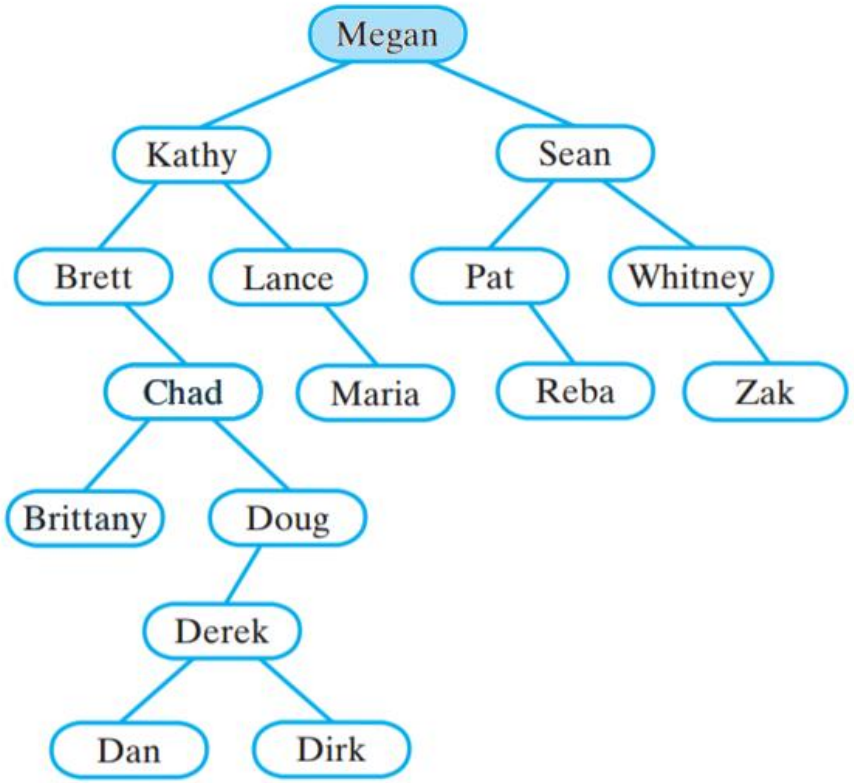
Removing an Entry

- **Removing an Entry in the Root**
  - If the root has two children, we replace the root's entry and delete a different node
  - If the root has one child, we delete the root node by making the child node C the root of the tree. (need to determine if a right child or a left child exists!)
  - If the root is a leaf, we delete it and get an empty tree



In-Class Exercise

- Remove Megan from the tree below in two different ways.

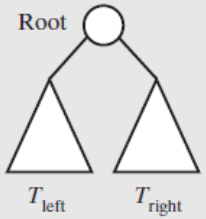


In-Class Exercise

- Show the resulting binary search tree after performing the following operations
  - Create a binary search tree with the values: 51, 29, 68, 90, 36, 40, 22, 59, 44, 99, 77, 60, 83, 15, 75, 3.
  - Add 33, 88, 1 to the binary search tree .
  - Remove 44, 90, 68, 3 from the binary search tree .

Removing an Entry (Recursive Version)

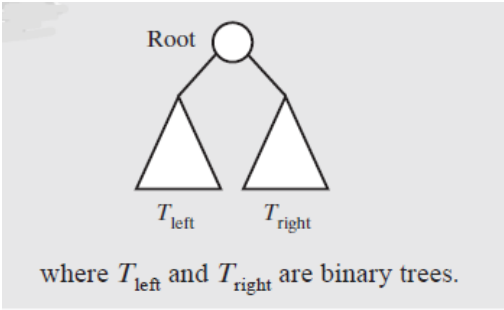
```
Algorithm remove(binarySearchTree, entry)
oldEntry = null
if (binarySearchTree is not empty)
{
   if (entry matches the entry in the root of binarySearchTree)
   {
      oldEntry = entry in root
      removeFromRoot(root of binarySearchTree)
   }
   else if (entry < entry in root)
      oldEntry = remove(left subtree of binarySearchTree, entry)
   else  // entry > entry in root
      oldEntry = remove(right subtree of binarySearchTree, entry)
}
return oldEntry
```



where T_{left} and T_{right} are binary trees.

Removing an Entry (Recursive Version)

```
Algorithm remove(binarySearchTree, entry)
oldEntry = null
if (binarySearchTree is not empty)
{
   if (entry matches the entry in the root of binarySearchTree)
   {
      oldEntry = entry in root
      removeFromRoot(root of binarySearchTree)
   }
   else if (entry < entry in root)
      oldEntry = remove(left subtree of binarySearchTree, entry)
   else  // entry > entry in root
      oldEntry = remove(right subtree of binarySearchTree, entry)
}
return oldEntry
```

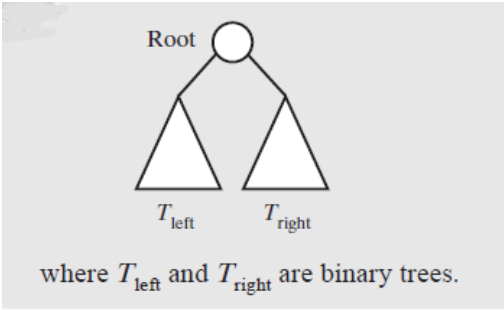


where T_{left} and T_{right} are binary trees.

```
Algorithm removeFromRoot(rootNode)
// Removes the entry in a given root node of a subtree.

if (rootNode has two children)
{
   largestNode = node with the largest entry in the left subtree of rootNode
   Replace the entry in rootNode with the entry in largestNode
   Remove largestNode from the tree
}
else if (rootNode has a right child)
   rootNode = rootNode's right child
else
   rootNode = rootNode's left child  // possibly null
// Assertion: if rootNode was a leaf, it is now null

return rootNode
```

Removing an Entry (Recursive Version)

```
Algorithm remove(binarySearchTree, entry)
oldEntry = null
if (binarySearchTree is not empty)
{
   if (entry matches the entry in the root of binarySearchTree)
   {
      oldEntry = entry in root
      removeFromRoot(root of binarySearchTree)
   }
   else if (entry < entry in root)
      oldEntry = remove(left subtree of binarySearchTree, entry)
   else  // entry > entry in root
      oldEntry = remove(right subtree of binarySearchTree, entry)
}
return oldEntry
```



where T_{left} and T_{right} are binary trees.

Question: which order of traversal is used in Algorithm remove()?

```
Algorithm removeFromRoot(rootNode)
// Removes the entry in a given root node of a subtree.

if (rootNode has two children)
{
   largestNode = node with the largest entry in the left subtree of rootNode
   Replace the entry in rootNode with the entry in largestNode
   Remove largestNode from the tree
}
else if (rootNode has a right child)
   rootNode = rootNode's right child
else
   rootNode = rootNode's left child  // possibly null
// Assertion: if rootNode was a leaf, it is now null

return rootNode
```

Removing an Entry (Recursive Version)

```
// Removes an entry from the tree rooted at a given node.
// rootNode is a reference to the root of a tree.
// entry is the object to be removed.
// oldEntry is an object whose data field is null.
// Returns the root node of the resulting tree; if entry matches
//         an entry in the tree, oldEntry's data field is the entry
//         that was removed from the tree; otherwise it is null.
private BinaryNodeInterface<T> removeEntry(BinaryNodeInterface<T> rootNode,
                                           T entry, ReturnObject oldEntry)
{
   if (rootNode != null)
   {
      T rootData = rootNode.getData();
      int comparison = entry.compareTo(rootData);

      if (comparison == 0)       // entry == root entry
      {
         oldEntry.set(rootData);
         rootNode = removeFromRoot(rootNode);
      }
      else if (comparison < 0)   // entry < root entry
      {
         BinaryNodeInterface<T> leftChild = rootNode.getLeftChild();
         BinaryNodeInterface<T> subtreeRoot = removeEntry(leftChild,
                                                          entry, oldEntry);
         rootNode.setLeftChild(subtreeRoot);
      }
      else                       // entry > root entry
      {
         BinaryNodeInterface<T> rightChild = rootNode.getRightChild();
         rootNode.setRightChild(removeEntry(rightChild, entry, oldEntry));
      } // end if
   } // end if

   return rootNode;
} // end removeEntry
```

```
Algorithm remove(binarySearchTree, entry)
oldEntry = null
if (binarySearchTree is not empty)
{
   if (entry matches the entry in the root of binarySearchTree)
   {
      oldEntry = entry in root
      removeFromRoot(root of binarySearchTree)
   }
   else if (entry < entry in root)
      oldEntry = remove(left subtree of binarySearchTree, entry)
   else  // entry > entry in root
      oldEntry = remove(right subtree of binarySearchTree, entry)
}
return oldEntry
```

Removing an Entry (Recursive Version)

```
// Removes the entry in a given root node of a subtree.
// rootNode is the root node of the subtree.
// Returns the root node of the revised subtree.
private BinaryNodeInterface<T> removeFromRoot(BinaryNodeInterface<T> rootNode)
{
   // Case 1: rootNode has two children
   if (rootNode.hasLeftChild() && rootNode.hasRightChild())
   {
      // find node with largest entry in left subtree
      BinaryNodeInterface<T> leftSubtreeRoot = rootNode.getLeftChild();
      BinaryNodeInterface<T> largestNode = findLargest(leftSubtreeRoot);

      // replace entry in root
      rootNode.setData(largestNode.getData());

      // remove node with largest entry in left subtree
      rootNode.setLeftChild(removeLargest(leftSubtreeRoot));
   } // end if

   // Case 2: rootNode has at most one child
   else if (rootNode.hasRightChild())
      rootNode = rootNode.getRightChild();
   else
      rootNode = rootNode.getLeftChild();

   // Assertion: if rootNode was a leaf, it is now null

   return rootNode;
} // end removeEntry
```

```
Algorithm removeFromRoot(rootNode)
// Removes the entry in a given root node of a subtree.

if (rootNode has two children)
{
   largestNode = node with the largest entry in the left subtree of rootNode
   Replace the entry in rootNode with the entry in largestNode
   Remove largestNode from the tree
}
else if (rootNode has a right child)
   rootNode = rootNode's right child
else
   rootNode = rootNode's left child  // possibly null
// Assertion: if rootNode was a leaf, it is now null

return rootNode
```

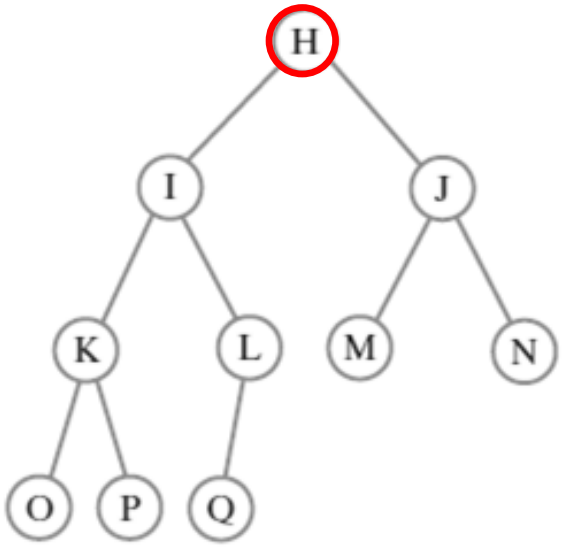
Removing an Entry (Recursive Version)

```
private BinaryNodeInterface<T> findLargest(BinaryNodeInterface<T> rootNode)
{
   if (rootNode.hasRightChild())
      rootNode = findLargest(rootNode.getRightChild());

   return rootNode;
} // end findLargest

// Removes the node containing the largest entry in a given tree.
// rootNode is the root node of the tree.
// Returns the root node of the revised tree.
private BinaryNodeInterface<T> removeLargest(BinaryNodeInterface<T> rootNode)
{
   if (rootNode.hasRightChild())
   {
      BinaryNodeInterface<T> rightChild = rootNode.getRightChild();
      BinaryNodeInterface<T> root = removeLargest(rightChild);
      rootNode.setRightChild(root);
   }
   else
      rootNode = rootNode.getLeftChild();

   return rootNode;
} // end removeLargest
```

Removing an Entry (Recursive Version)

```
private BinaryNodeInterface<T> findLargest(BinaryNodeInterface<T> rootNode)
{
   if (rootNode.hasRightChild())
      rootNode = findLargest(rootNode.getRightChild());

   return rootNode;
} // end findLargest

// Removes the node containing the largest entry in a given tree.
// rootNode is the root node of the tree.
// Returns the root node of the revised tree.
private BinaryNodeInterface<T> removeLargest(BinaryNodeInterface<T> rootNode)
{
   if (rootNode.hasRightChild())
   {
      BinaryNodeInterface<T> rightChild = rootNode.getRightChild();
      BinaryNodeInterface<T> root = removeLargest(rightChild);
      rootNode.setRightChild(root);
   }
   else
      rootNode = rootNode.getLeftChild();

   return rootNode;
} // end removeLargest
```

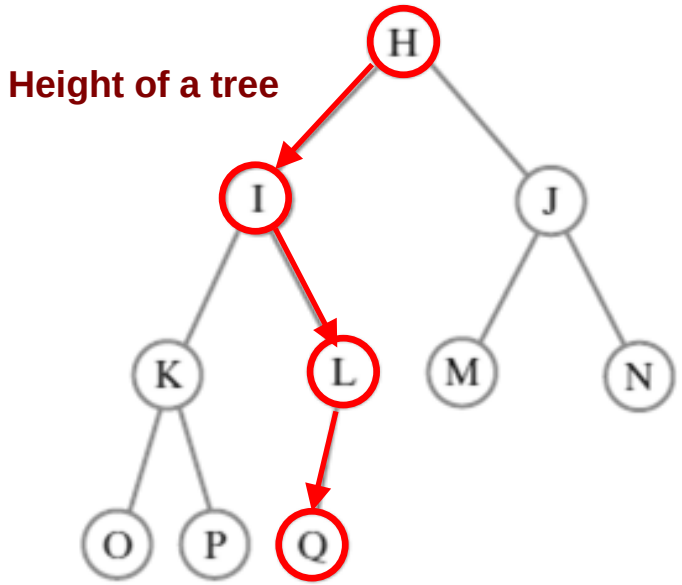
//Why we use getLeftChild here?



Efficiency of Operations

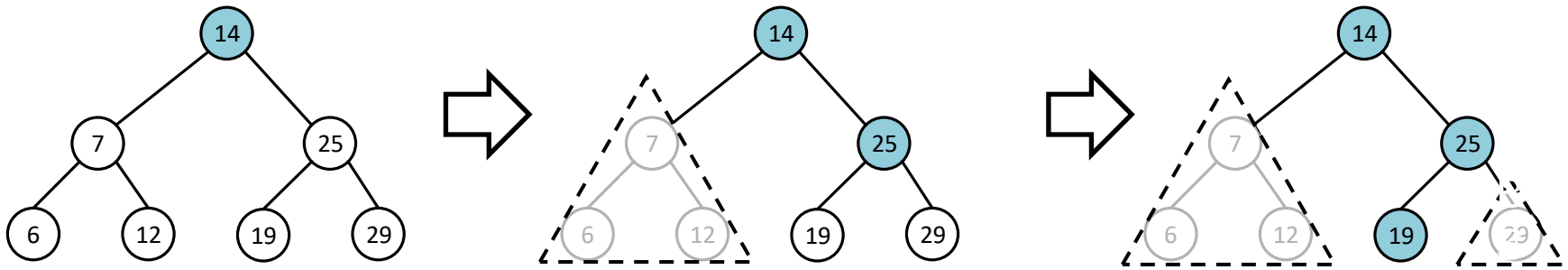
- Operations: add, remove, and getEntry
  - Require a search that begins at the root of the tree

- The maximum number of comparisons that each operation requires is directly proportional to the **height h** of the tree.
  - The time complexity of these operations are $O(h)$



Efficiency of Operations

- (Recall) Search for 19



7 nodes

$\frac{7}{2}$ = 3 nodes

$\frac{3}{2}$ = 1 node

3 Steps

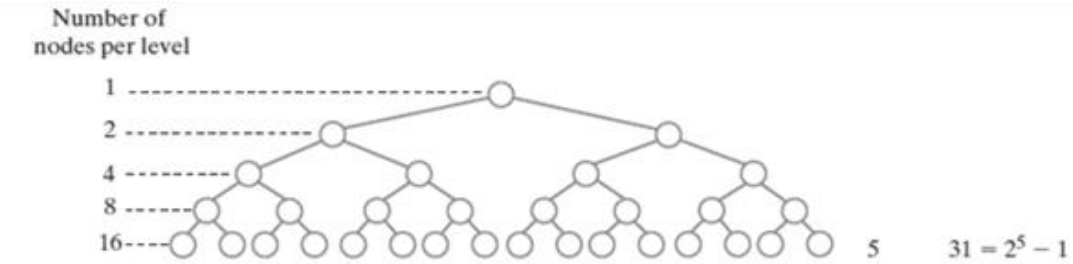
- If a BST has $\boldsymbol{n}$ nodes,

$\log_2(n)$ Steps

n nodes ⇨ $\frac{n}{2}$ **nodes** ⇨ $\frac{n}{4}$ **nodes** ⇨ ⋯ ⇨ **1 node**

Efficiency of Operations

- A shortest tree is full or compete
  - The height of a full or complete tree that has n nodes is $log_2(n+1)$ rounded up
  - Results in these operations being $O(\log_2 n)$

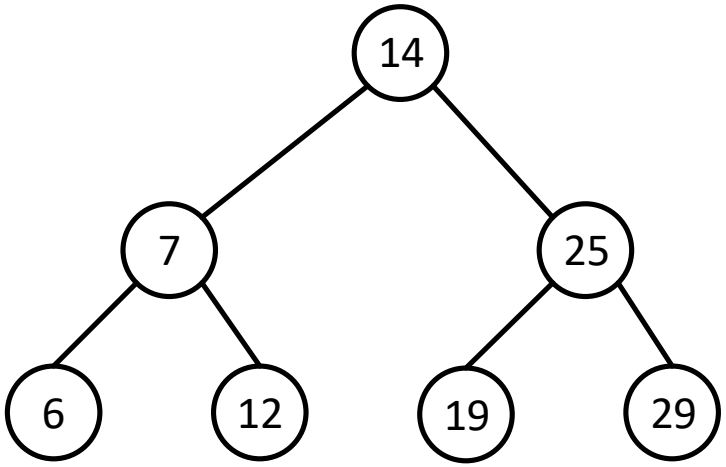


- Balance
  - The substrees of each node in the tree differ in height by no more than 1.
  - Unbalanced trees affect the performance of the operations

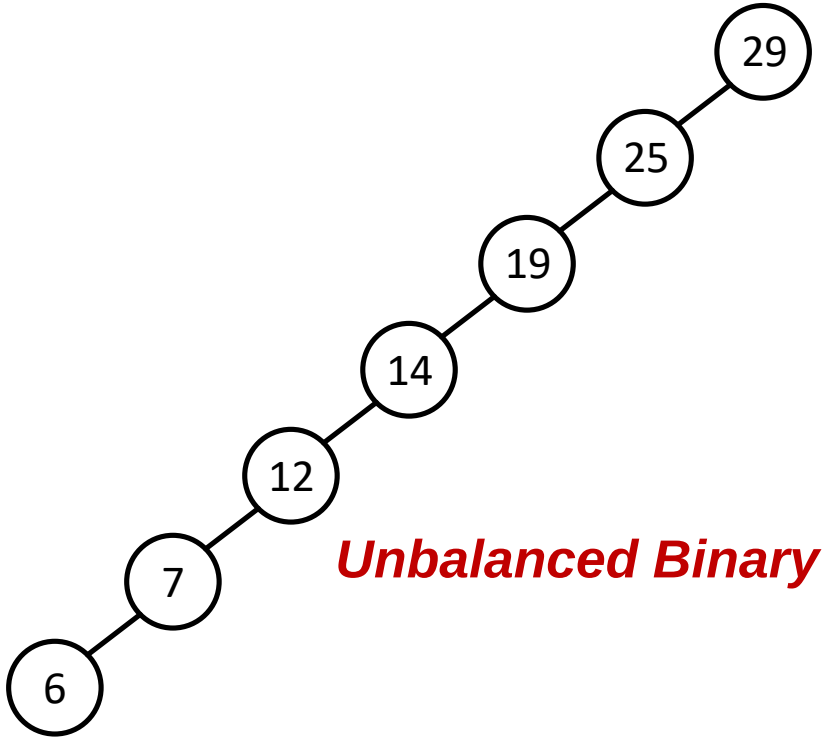
Reference: https://www.cs.odu.edu/~zeil/cs361/sum15/Public/bst/index.html

Importance of Being Balanced

- If we search for 6



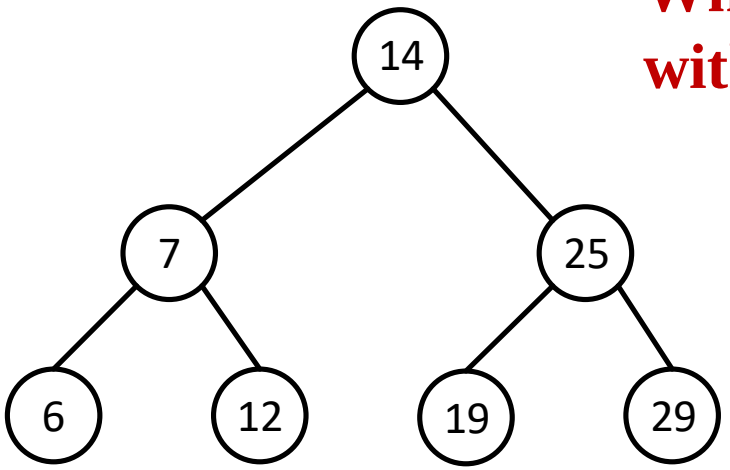
Balanced Binary Search Tree



Unbalanced Binary Search Tree

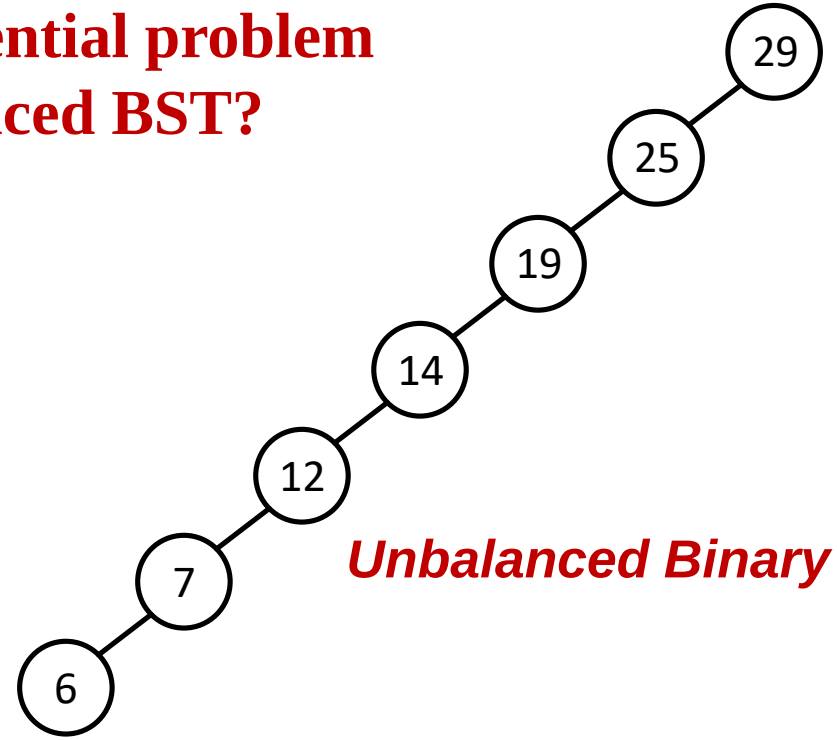
Importance of Being Balanced

- If we search for 6



Balanced Binary Search Tree

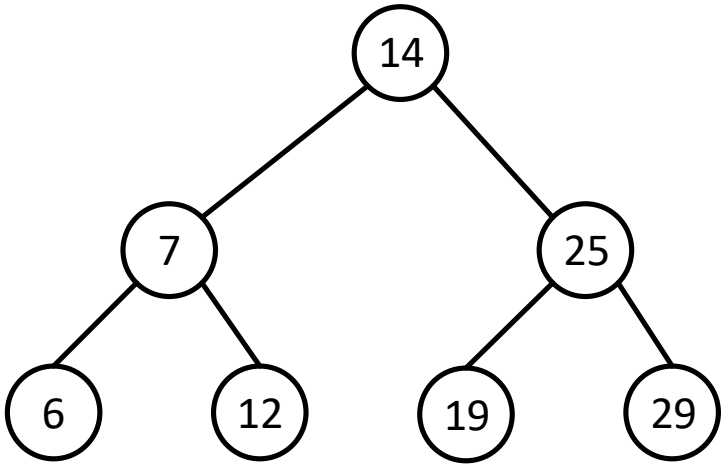
What is the potential problem with an unbalanced BST?



Unbalanced Binary Search Tree

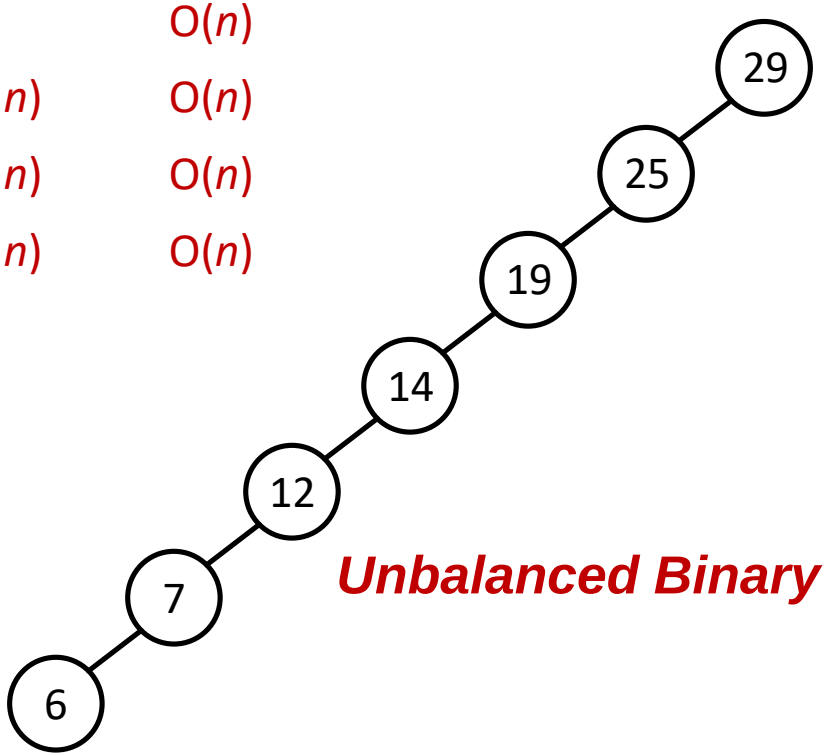
Importance of Being Balanced

- If we search for 6



Balanced Binary Search Tree

Algorithm	Average	Worst Case
Space	Θ(*n*)	O(*n*)
Search	Θ(log *n*)	O(*n*)
Insert	Θ(log *n*)	O(*n*)
Delete	Θ(log *n*)	O(*n*)



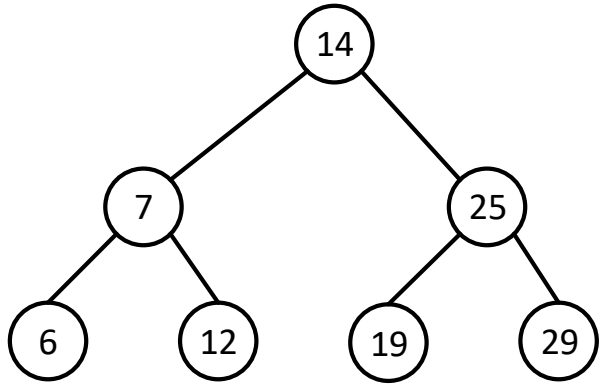
Unbalanced Binary Search Tree

Order in Which Nodes Are Added

- Order in which you add entries to a binary search tree affects the shape of the tree

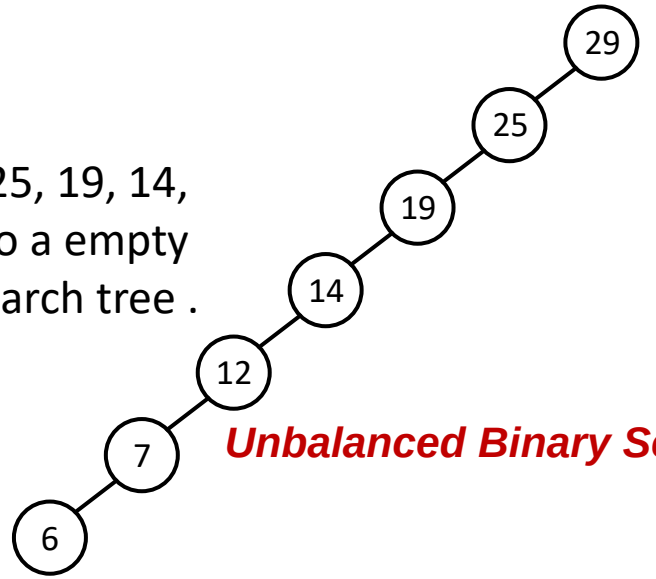
- If you add entries into an initially empty binary search tree, do not add them in sorted order.

Add 14, 7, 25, 6, 12, 19, 29 to a empty binary search tree .



Balanced Binary Search Tree

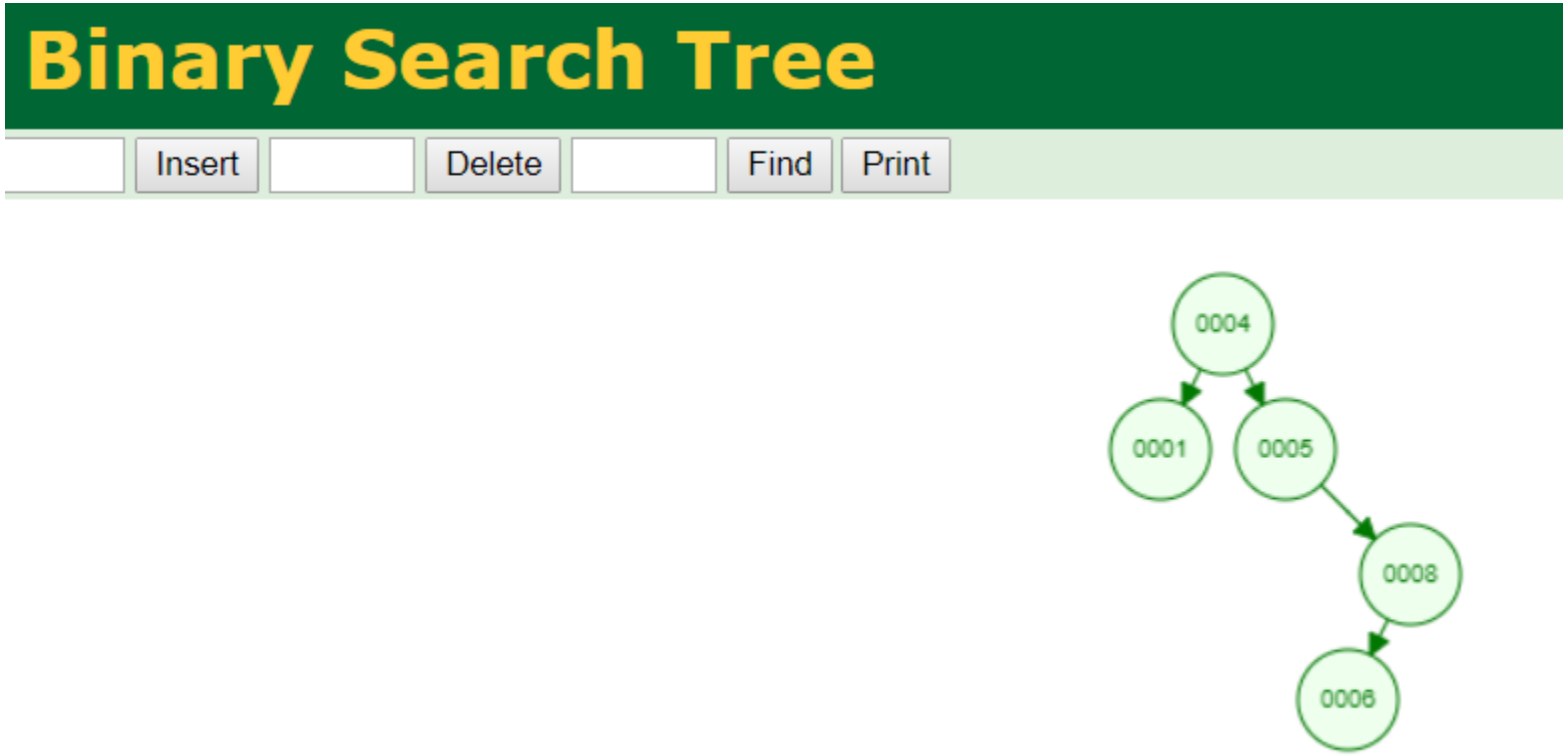
Add 29, 25, 19, 14, 12, 7, 6 to a empty binary search tree .



Unbalanced Binary Search Tree

Interactive and Visualization Demo

- https://www.cs.usfca.edu/~galles/visualization/BST.html



Summary

- Binary Search Tree
  - Definition
  - Operations in Binary Search Tree
    - Search for an Entry
    - Adding an Entry (Iterative Version)

What I Want You to Do

- Review Class Slides
- Review Chapters 24 and 25