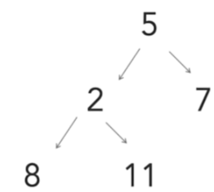
Chapter 07 Binary Trees

Tree Taversals

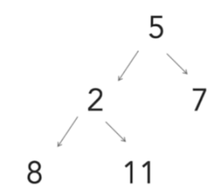
Pre-order: explores the roots before leaves

Post-order: explores leaves before roots

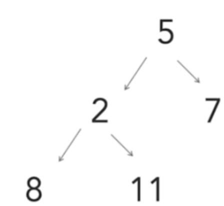
In-order: explores data sequentially



Pre-order: Visit root, recursively visit left subtree, recursively visit right subtree: 5, 2, 8, 11, 7



In-order: Recursively visit left subtree, visit root, recursively visit right subtree: 8, 2, 11, 5, 7



Post-order: Recurse left subtree, recurse right subtree, visit root: 8, 11, 2, 7, 5

Java Algorithms

- In a tree traversal algorithm, each node in a tree is accessed in a particular order
- With each traversal, we apply the same pattern throughout the tree
- Recursion can prove to be a useful tool for this type of algorithm because it continuously follows a pattern with slight modifications to the input

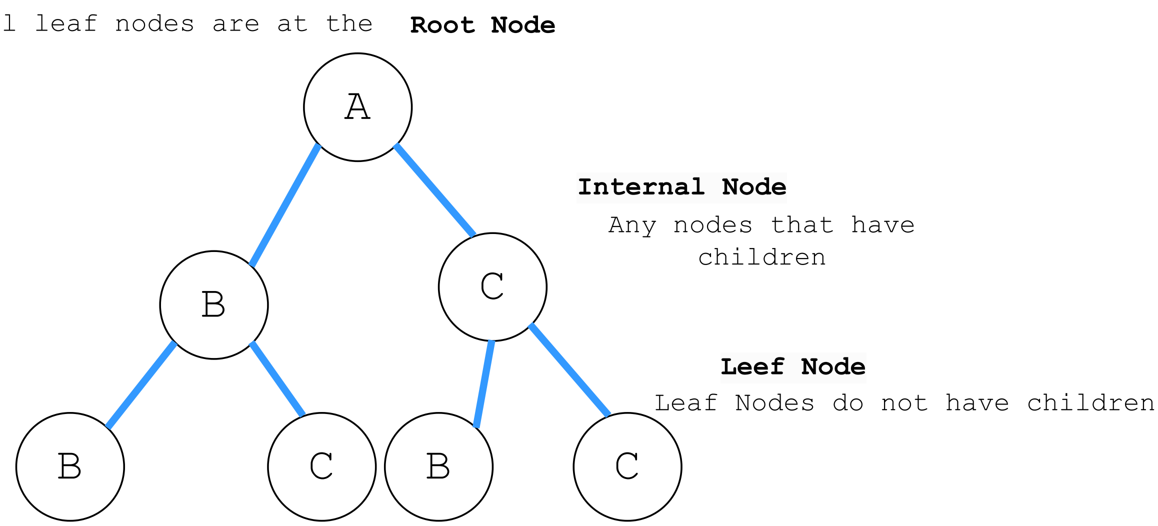
Chapter 07 Binary Trees

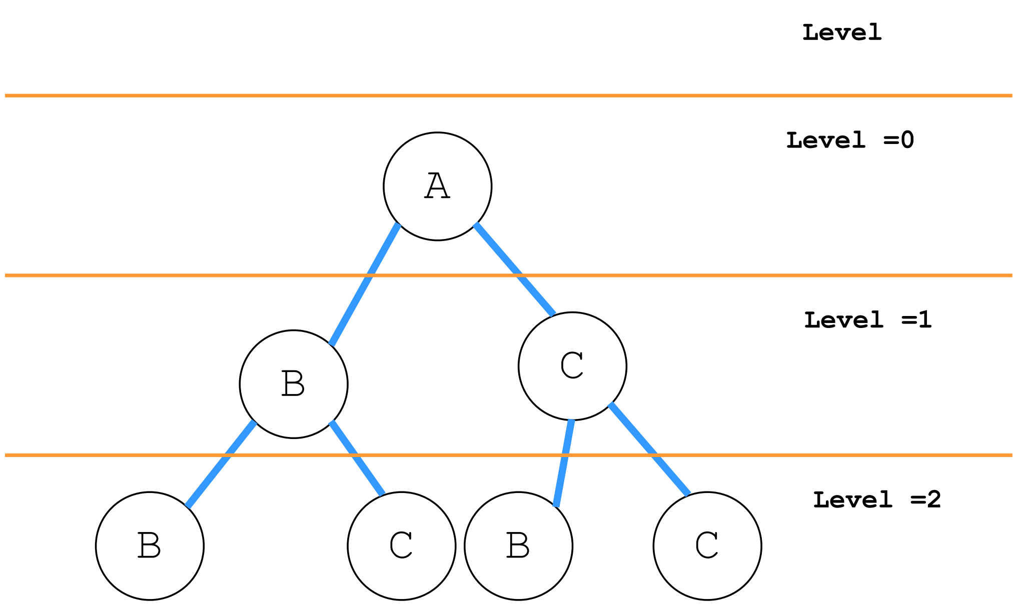
Types of Binary Trees

Full - if every node contains 0 or 2 children

Complete - if all levels except possibly the last level contain all possible nodes and all nodes in the last level are as far left as possible

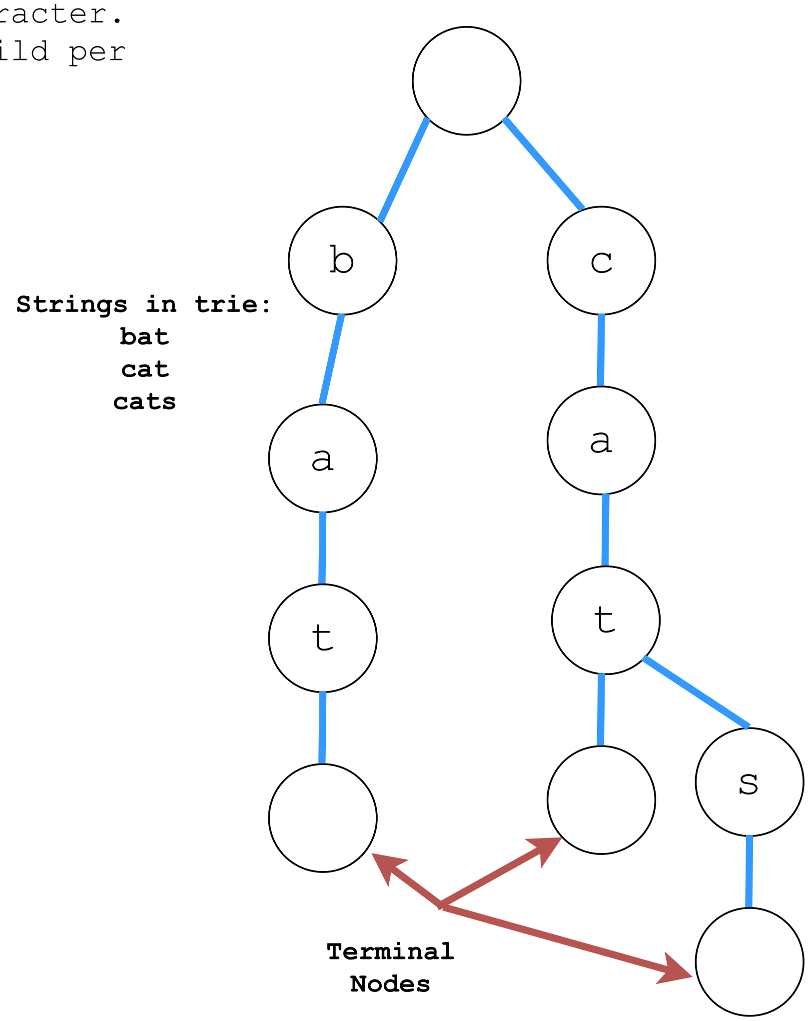
Perfect - if all internal nodes have 2 children and all leaf nodes are at the same level





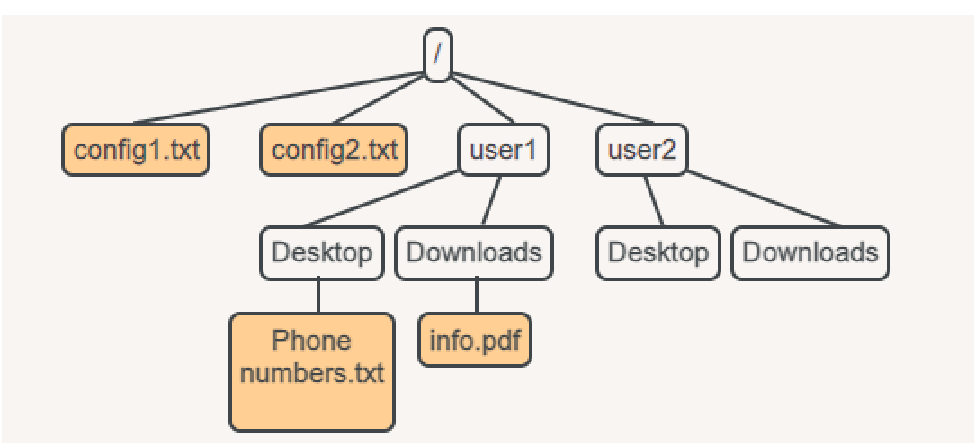
Tries

Trie (or prefix tree) is a tree representing a set of strings. Each non root represents a single character. Each node has at most one child per distinct alphabet character



7.2 Application of Trees

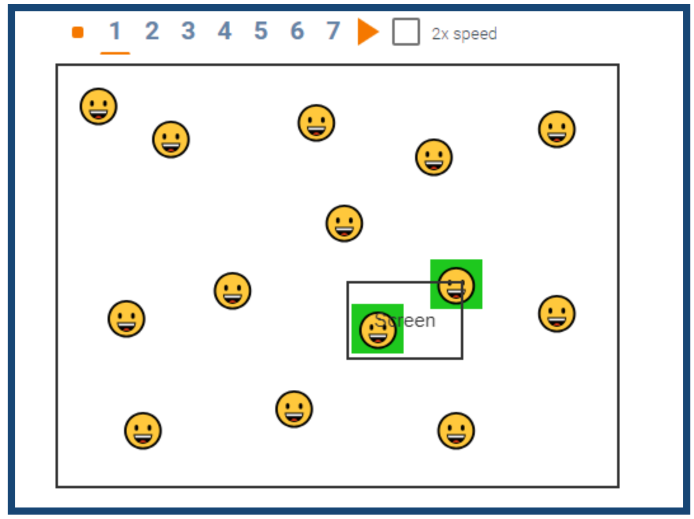
File Systems - Trees are commonly used to represent hierarchical data. A tree can represent files and directories in a file system



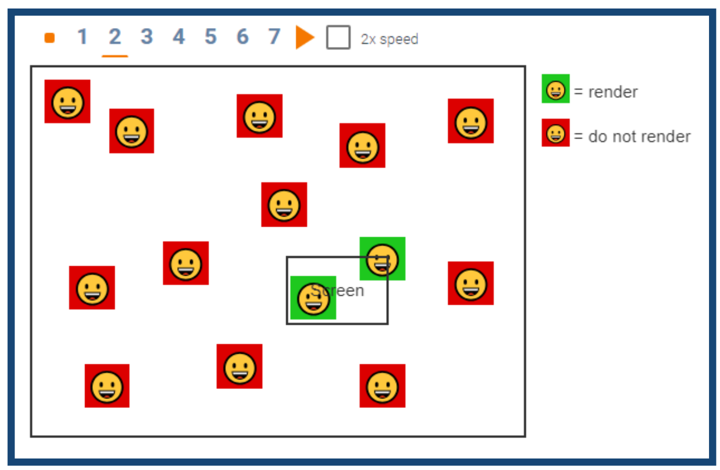
7.2 Binary Space Partition (BSP)

7.2 Binary Space Partition (BSP)
Binary Space Partitioning (BSP) is a technique of repeatedly separating a region of space into 2 parts and cataloging the objects contained within the regions.

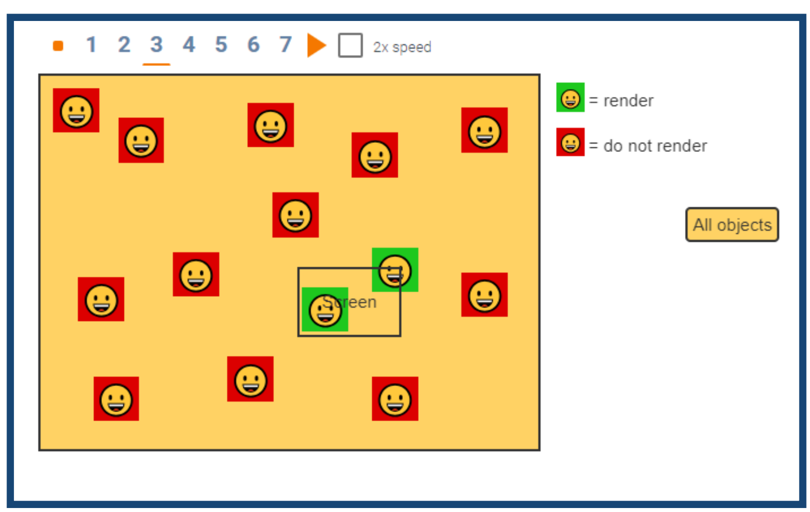
Data for a large, open 2-D world contains many objects. Only a few are visible on screen at any given time



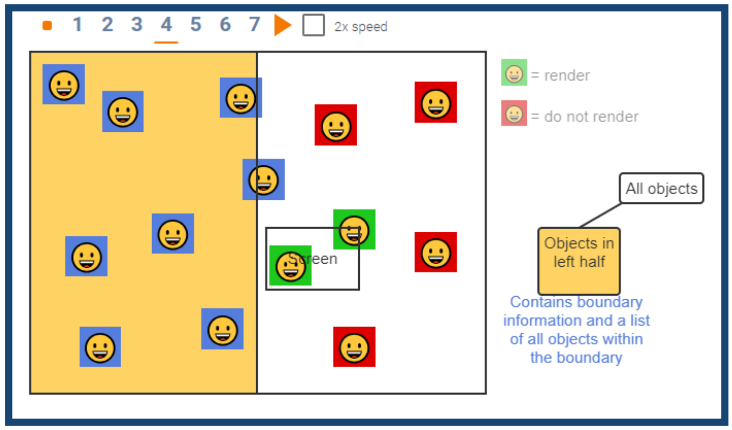
Avoiding rendering off-screen objects is crucial for realtime graphics. But checking the intersection of all objects with the screens rectable is too time consuming.



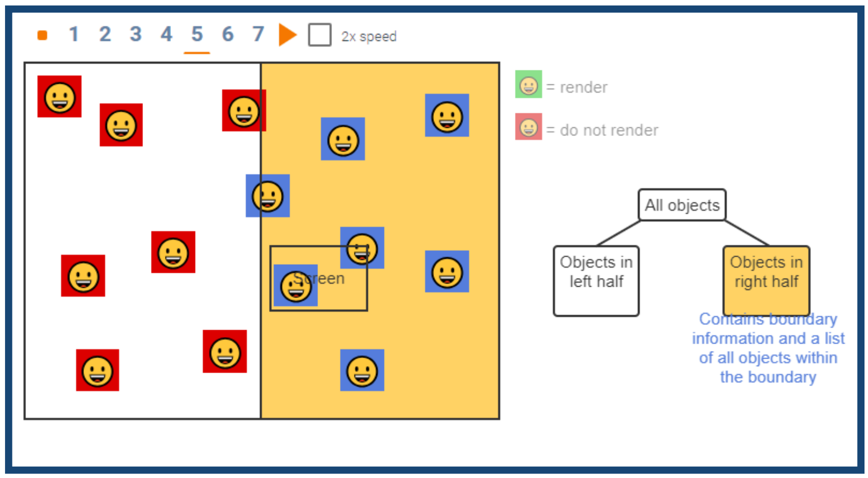
A BSP tree represents partitioned space. The root represents the entire world and stores a list of all objects in the world, as well as the world's geometric boundary.



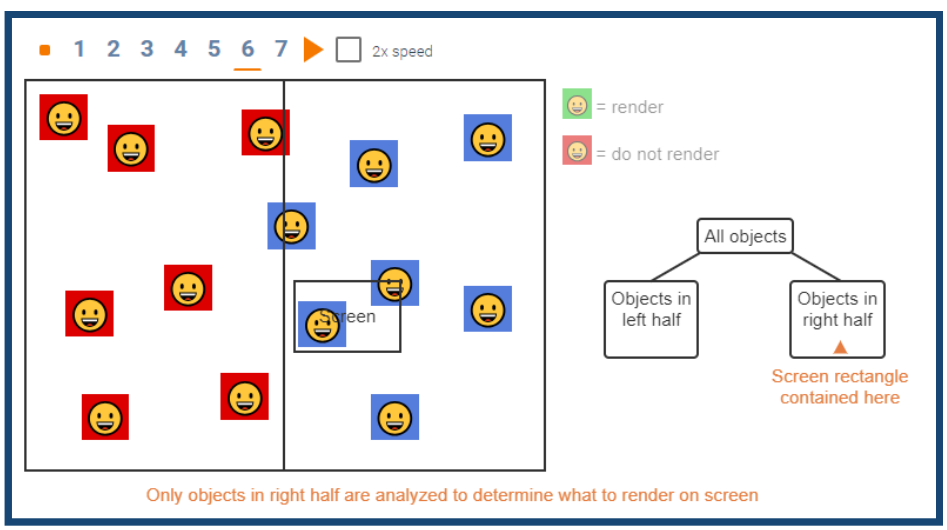
The root's left child represents the world's left half. The node stores information about the left half's geometric boundary, and a list of all objects contained within.



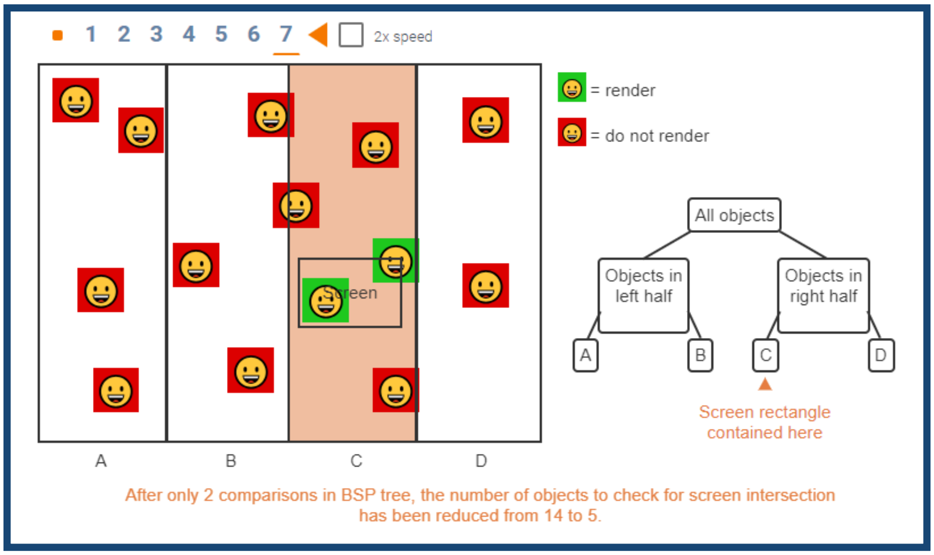
The root's right child contains similar information for the right half



Using the screen's position within the world as a lookup into the BSP tree quickly yields the right node's list of objects. A large number of objects are quickly eliminated from the list of potential objects on screen.



Further partitioning makes the tree even more useful




```
//Trie Insert Algorithm
TrieInsert(root, string ){
    node = root;
    for (character in string) {
        if ( character is not in node --> children ) {
            node-->children [ character ] = new TrieNode();
        }
        node = node-->children[character];
    }

    if (0 is not in node --> children){
    node --> children[0] = new TriNode();
    }
    return node --> children[0];
}
```

```
//Trie Search Algorithm
TrieSearch(root, string){
    node = root;

    for (character in string) {
        if (character is not in node-->children) {
            return null ;
        }
        node = node --> children[character];
        }

    if (0 is in node --> children) {
        return node --> children [0];
    }
    return null ;
}
```

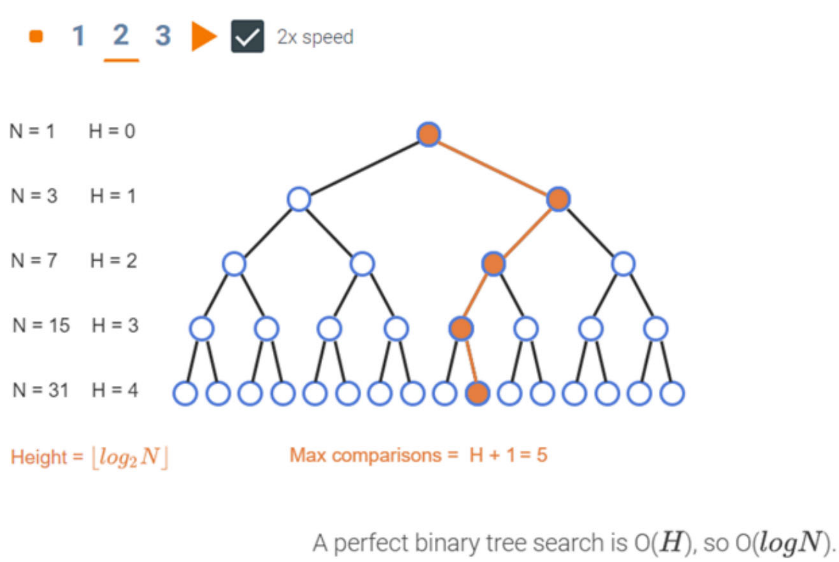
```
TrieRemoveRecursive(node, string, charIndex) {
   if (charIndex == string --> length) {
      if (0 in node --> children) {
         Remove 0 from node-->children[0]
         return true;
      }
      return false // string not found
   }
   character = string[charIndex];

   if (character is not in node-->children) {
      return false
   }
   child = node-->children[character];
   TrieRemoveRecursive(child, string, charIndex + 1)
   if (child-->children-->length == 0) {
      Remove character from node-->children;
   }
   return true;
}
```

Successor and Predecessors

- Successor – the node that comes after in BST ordering
- Predecessors – the node that comes before in BST ordering

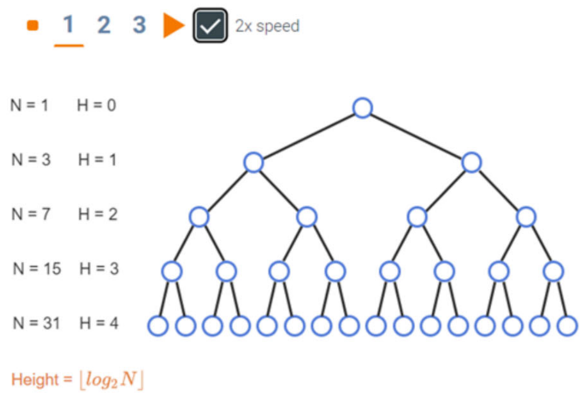
7.3 Binary Trees Search Runtime

PARTICIPATION ACTIVITY | 7.3.5: Searching a perfect BST with N nodes requires only O($logN$) comparisons.



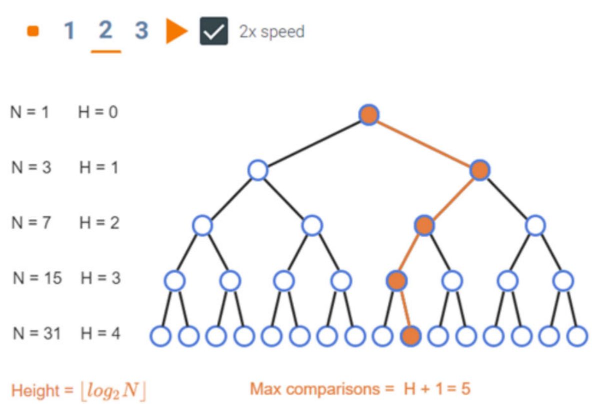
7.3 Binary Trees Search Runtime

PARTICIPATION ACTIVITY | 7.3.5: Searching a perfect BST with N nodes requires only O($logN$) comparisons.



A perfect binary tree has height $\lfloor log_2 N \rfloor$.

PARTICIPATION ACTIVITY | 7.3.5: Searching a perfect BST with N nodes requires only O($logN$) comparisons.

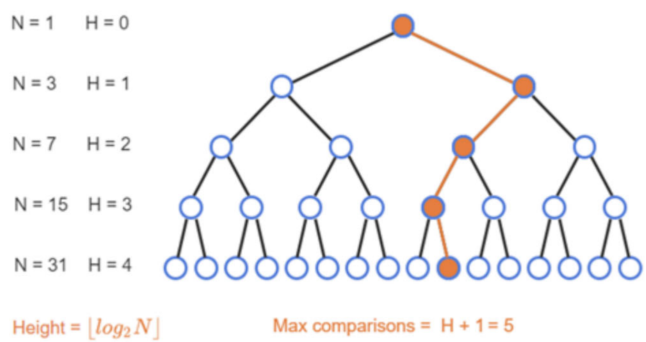


A perfect binary tree search is O(H), so O($logN$).

7.3 Binary Trees Search Runtime

PARTICIPATION ACTIVITY | 7.3.5: Searching a perfect BST with N nodes requires only O($log N$) comparisons.

1 2 3 2x speed



Searching a full list vs perfect BST

Nodes (N)	List	BST
10	10	4
100	100	7
1,000	1,000	10
10,000	10,000	14
100,000	100,000	17

Searching a BST may be faster than searching a list.

BST Search Algorithm

```
if (currentNode→key == desiredKey) {
   return currentNode; // The desired node was found
}
else if (desiredKey < currentNode→key) {
   // Visit left child, repeat
}
else if (desiredKey > currentNode→key) {
   // Visit right child, repeat
}
```

- A BST may yield faster searches than a list.
- Searching a BST starts at the root node

BST Search Runtime

- When searching a BST, the worst case requires H+1 comparisons " O(H) ", where H is the tree height
- Benefit of BST is that N-node binary tree's height may be as small as O(logN)

Table 7.3.1: Minimum binary tree heights for N nodes are equivalent to $\lfloor log_2 N \rfloor$.

Nodes N	Height H	$log_2 N$	$\lfloor log_2 N \rfloor$	Nodes per level
1	0	0	0	1
2	1	1	1	1/1
3	1	1.6	1	1/2
4	2	2	2	1/2/1
5	2	2.3	2	1/2/2
6	2	2.6	2	1/2/3
7	2	2.8	2	1/2/4
8	3	3	3	1/2/4/1
9	3	3.2	3	1/2/4/2
...				
15	3	3.9	3	1/2/4/8
16	4	4	4	1/2/4/8/1

7.5 BST Insert Algorithm

→ Insert as left child:

If the following is true

- If the new nodes key is less than the current node
- Current node's left child is null

-Algorithm assigns left child with new Node

→ Insert as right child:

If the following is true

- If the new node's key is greater than or equal to the current node
- Current nodes right child is null

- Algorithm assigns right child with new node

7.5 BST insert algorithm

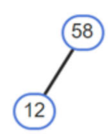
Given a new node, a BST ***insert*** operation inserts the new node in a proper location obeying the BST ordering property. A simple BST insert algorithm compares the new node with the current node (initially the root).

- *Insert as left child*: If the new node's key is less than the current node, and the current node's left child is null, the algorithm assigns that node's left child with the new node.
- *Insert as right child*: If the new node's key is greater than or equal to the current node, and the current node's right child is null, the algorithm assigns the node's right child with the new node.
- *Search for insert location*: If the left (or right) child is not null, the algorithm assigns the current node with that child and continues searching for a proper insert location.

PARTICIPATION ACTIVITY | 7.5.1: Binary search tree insertions.

1 2 2x speed

```
BSTInsert(tree, node) {
   if (tree→root is null) {
      tree→root = node
   }
   else {
      currentNode = tree→root
      while (currentNode is not null) {
         if (node→key < currentNode→key) {
            if (currentNode→left is null) {
               currentNode→left = node
               currentNode = null
            }
            else {
               currentNode = currentNode→left
            }
         }
         else {
            if (currentNode→right is null) {
               currentNode→right = node
               currentNode = null
            }
            else {
               currentNode = currentNode→right
            }
         }
      }
   }
}
```



Benefit of a BST is that inserts require only 0(logN) iterations to find the insert location in a nearly-full node tree

The BST is searched to find a suitable location to insert the new node as a leaf node.

BST Insert Algorithm Complexity

- → Complexity
- → Best case
  - → O(logN)
- → Worst case
  - → O(N)

BST Remove Algorithm Code

Figure 7.6.1: BST remove algorithm.

```
BSTRemove(tree, key) {
   par = null
   cur = tree→root
   while (cur is not null) { // Search for node
      if (cur→key == key) { // Node found
         if (cur→left is null && cur→right is null) { // Remove leaf
            if (par is null) // Node is root
               tree→root = null
            else if (par→left == cur)
               par→left = null
            else
               par→right = null
         }
         else if (cur→right is null) {                // Remove node with only left child
            if (par is null) // Node is root
               tree→root = cur→left
            else if (par→left == cur)
               par→left = cur→left
            else
               par→right = cur→left
         }
         else if (cur→left is null) {                 // Remove node with only right child
            if (par is null) // Node is root
               tree→root = cur→right
            else if (par→left == cur)
               par→left = cur→right
            else
               par→right = cur→right
         }
         else {                                       // Remove node with two children
            // Find successor (leftmost child of right subtree)
            suc = cur→right
            while (suc→left is not null)
               suc = suc→left
            successorData = Create copy of suc's data
            BSTRemove(tree, suc→key)     // Remove successor
            Assign cur's data with successorData
         }
         return // Node found and removed
      }
      else if (cur→key < key) { // Search right
         par = cur
         cur = cur→right
      }
      else {                     // Search left
         par = cur
         cur = cur→left
      }
   }
   return // Node not found
}
```

BST Remove Algorithm Complexity

BST Remove- operation removes the first-found matching node
Three possible actions

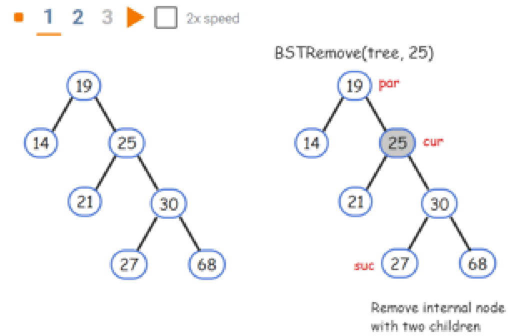
1. Remove a leaf node
2. Remove an internal node
3. Remove an internal node with two children

A BST with N nodes has at least log2N Levels and at most N levels.

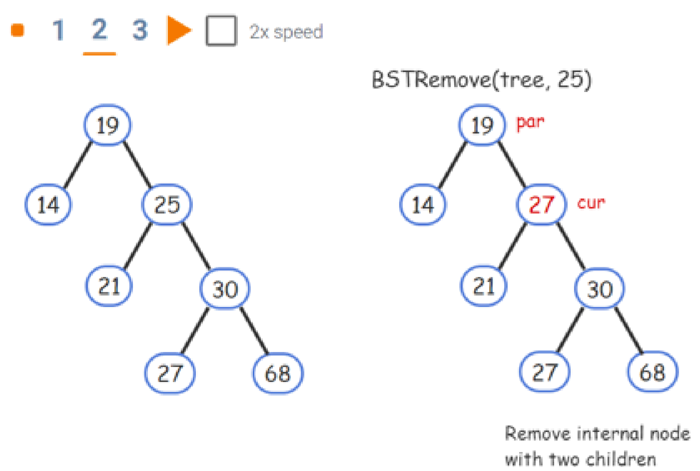
- → Complexity
- → Best Case = ??
- → Worst case for BST with log2(N) levels
  - → O(logN)
- → Worst case for BST with N Levels
  - → O(N)

BST Remove Internal node with two Children

PARTICIPATION ACTIVITY | 7.6.2: BST remove: Removing internal node with two children.

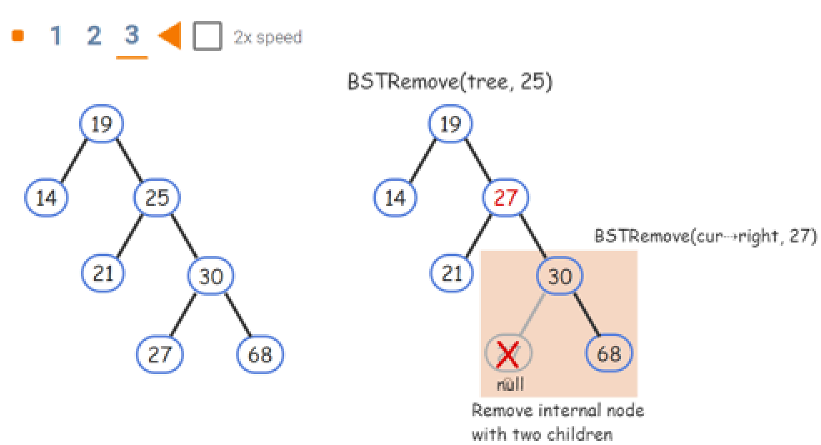


Find successor: Leftmost child in node 25's right subtree is node 27.



Copy successor to current node.

7.6.2: BST remove: Removing internal node with two children.



Remove successor from right subtree.

7.7 BST Inorder Traversal

```
BSTPrintInorder(node) {
  if (node is null)
      return

  BSTPrintInorder(node⇢left)
  Print node
  BSTPrintInorder(node⇢right)
}
```

BST Insert Algorithm

```
BSTInsert(tree, node) {
   if (tree→root is null) {
      tree→root = node
   }
   else {
      currentNode = tree→root
      while (currentNode is not null) {
         if (node→key < currentNode→key) {
            if (currentNode→left is null) {
               currentNode→left = node
               currentNode = null
            }
            else {
               currentNode = currentNode→left
            }
         }
         else {
            if (currentNode→right is null) {
               currentNode→right = node
               currentNode = null
            }
            else {
               currentNode = currentNode→right
            }
         }
      }
   }
}
```

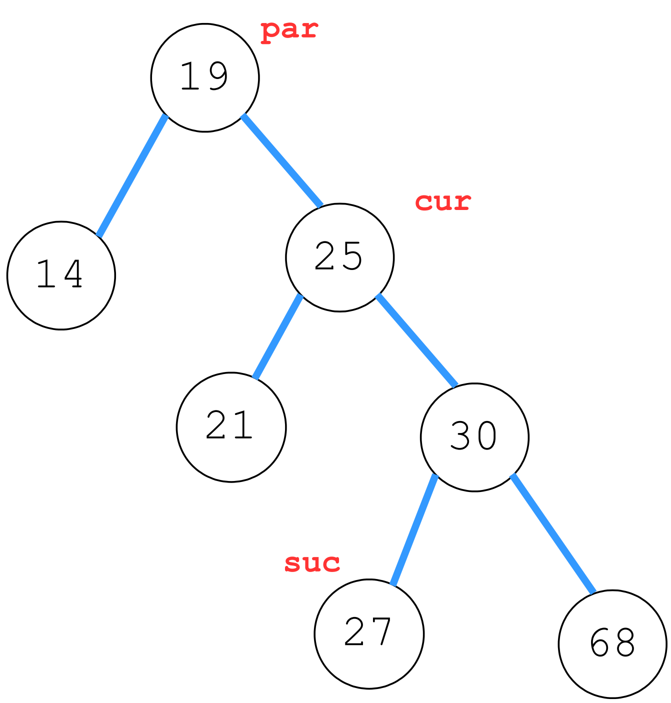
BST Insert Algorithm Complexity

The Space complexity of insertion is O(1) because only a single pointer is used to traverse the tree to find the insertion location

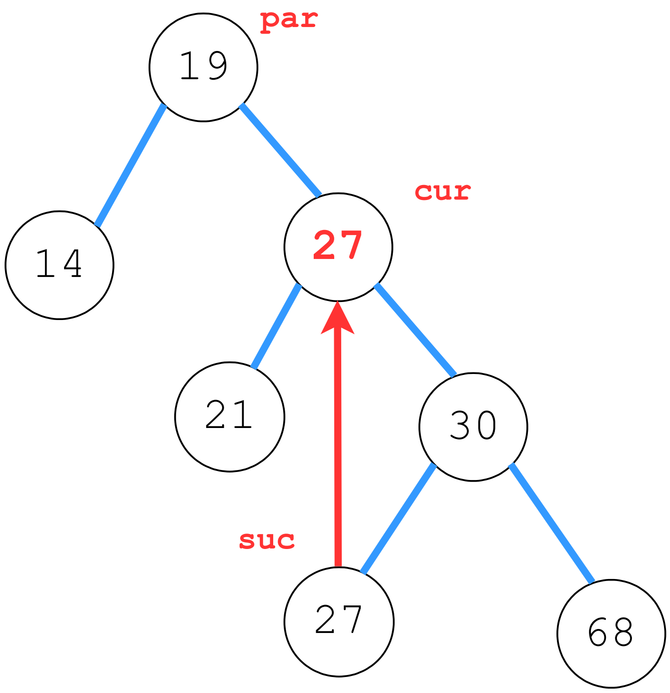
BST Remove Algorithm Complexity

- BST remove operation removes the first-found matching node, and then restructures the BST to preserve the ordering properties
- Steps
  1. Remove leaf node
  2. Remove internal node with single child
  3. Remove an internal node with two children

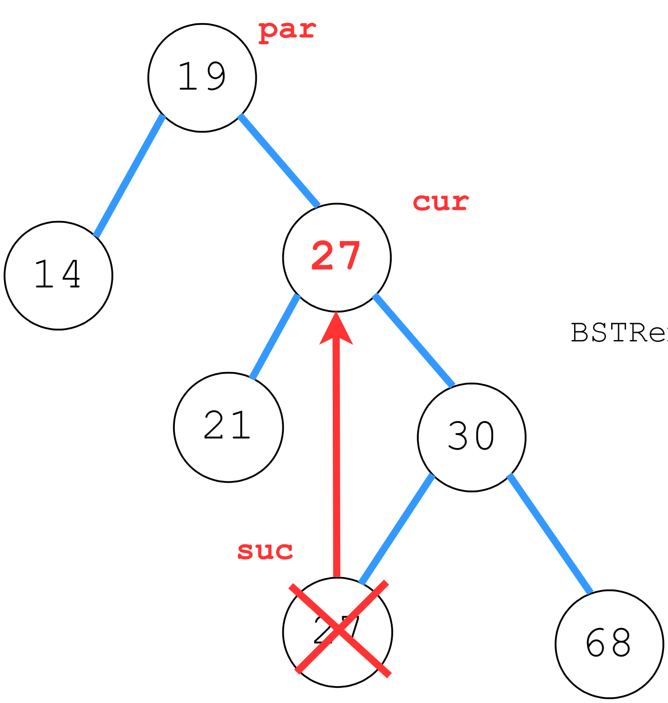
- BSTRemove(tree,25)
- Remove internal node with two children



- Remove internal node with two children



- Remove successor from the right subtree

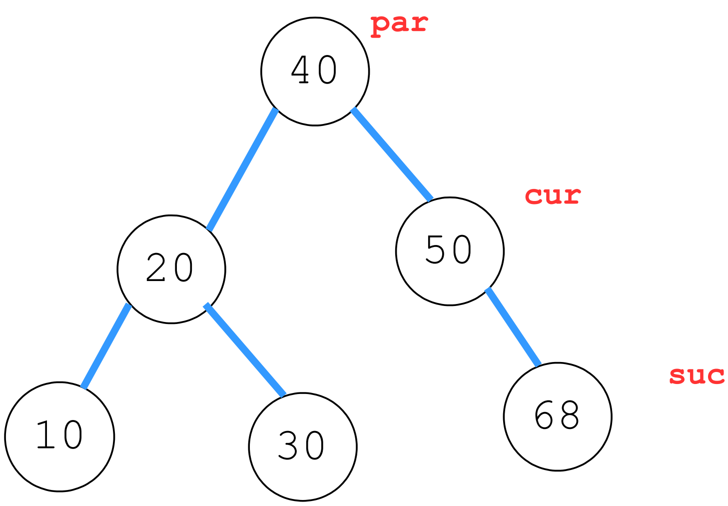


BSTRemove(cur --> right, 27)

```
BSTRemove(tree, key) {
   par = null
   cur = tree⇢root
   while (cur is not null) { // Search for node
      if (cur⇢key == key) { // Node found
         if (cur⇢left is null && cur⇢right is null) { // Remove leaf
            if (par is null) // Node is root
               tree⇢root = null
            else if (par⇢left == cur)
               par⇢left = null
            else
               par⇢right = null
         }
         else if (cur⇢right is null) {                // Remove node with only left child
            if (par is null) // Node is root
               tree⇢root = cur⇢left
            else if (par⇢left == cur)
               par⇢left = cur⇢left
            else
               par⇢right = cur⇢left
         }
         else if (cur⇢left is null) {                // Remove node with only right child
            if (par is null) // Node is root
               tree⇢root = cur⇢right
            else if (par⇢left == cur)
               par⇢left = cur⇢right
            else
               par⇢right = cur⇢right
         }
         else {                                      // Remove node with two children
            // Find successor (leftmost child of right subtree)
            suc = cur⇢right
            while (suc⇢left is not null)
               suc = suc⇢left
            successorData = Create copy of suc's data
            BSTRemove(tree, suc⇢key)     // Remove successor
            Assign cur's data with successorData
         }
         return // Node found and removed
      }
      else if (cur⇢key < key) { // Search right
         par = cur
         cur = cur⇢right
      }
      else {                       // Search left
         par = cur
         cur = cur⇢left
      }
   }
   return // Node not found
}
```

BST Remove Algorithm Complexity

- Removals worst case time complexity for BST with log2N levels is O(logN)
- Removals worst case time complexity for for a tree with N levels is O(N)



```java
/**
 *
 */
package com.practice01;

import java.util.Scanner;

/**
 *
 */
public class Main {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Scanner scanner = new Scanner(System.in);
        BinarySearchTree tree = new BinarySearchTree();

        tree.insert(new Node(5));
        tree.insert(new Node(2));
        tree.insert(new Node(3));
        tree.insert(new Node(6));
        tree.insert(new Node(9));
        tree.insert(new Node(7));
        tree.insert(new Node(8));

        tree.display();

    }

}
```

```java
/**
 *
 */
package com.practice01;

/**
 *
 */
public class Node {

    int data;
    Node left;
    Node right;

    public Node(int data) {
        this.data = data;
    }

}
```

```java
package com.practice01;

/**
 *
 */
public class BinarySearchTree {

    Node root;

    public void insert(Node node) {

        root = insertHelper(root, node);
        System.out.println("Inserted root.data # " + root.data);
        System.out.println();

    };

    // we are using the helper method because we are using recursion
    private Node insertHelper(Node root, Node node) {

        int data = node.data;

        if (root == null) {
            root = node;
            System.out.println("data= " + data);
            System.out.println("root.data = " + root.data);
            return root;

        } else if (data < root.data) {

            System.out.println("data= " + data);
            System.out.println("root.data= " + root.data);

            // System.out.println("root.left.data= " + root.left.data);
            root.left = insertHelper(root.left, node);

        } else {
            System.out.println("data= " + data);
            System.out.println("root.data= " + root.data);

            root.right = insertHelper(root.right, node);
        }
        return node;
    };

    // display method
    public void display() {

        displayHelper(root);
    };

    private void displayHelper(Node root) {

        if (root != null) {

            displayHelper(root.left);
            System.out.println(root.data);
            displayHelper(root.right);
        }

    };
```

```java
public static void main(String[] args) {
    // TODO Auto-generated method stub
    Scanner scanner = new Scanner(System.in);
    BinarySearchTree tree = new BinarySearchTree();

    tree.insert(new Node(5));
    tree.insert(new Node(2));
    tree.insert(new Node(3));
    tree.insert(new Node(6));
    tree.insert(new Node(9));
    tree.insert(new Node(7));
    tree.insert(new Node(8));

    tree.display();
```

```java
public static void main(String[] args) {
    // TODO Auto-generated method stub
    Scanner scanner = new Scanner(System.in);
    BinarySearchTree tree = new BinarySearchTree();

    tree.insert(new Node(5));
    tree.insert(new Node(2));
    tree.insert(new Node(3));
    tree.insert(new Node(6));
    tree.insert(new Node(9));
    tree.insert(new Node(7));
    tree.insert(new Node(8));

    tree.display();
```

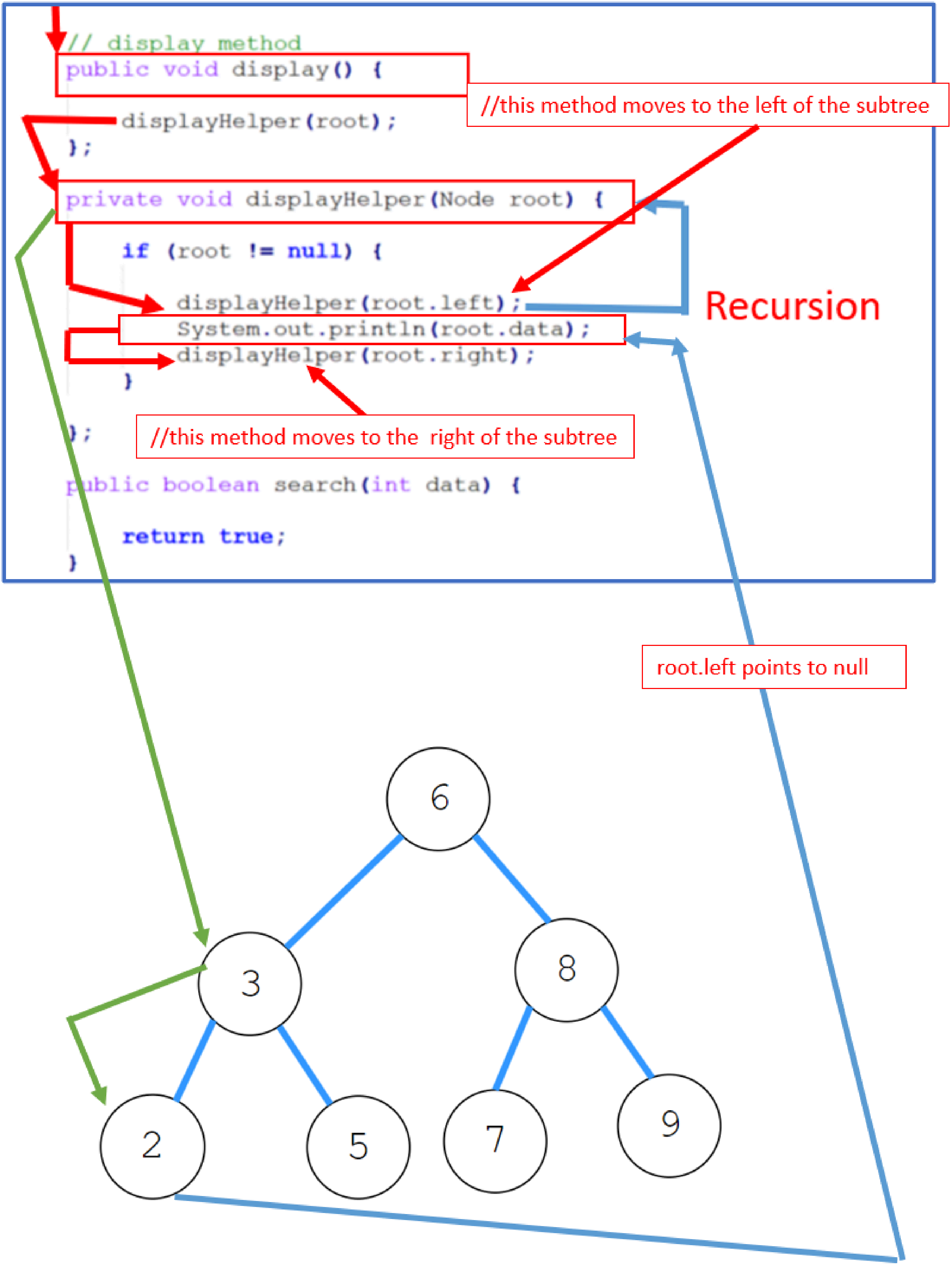
```java
package com.practice01;

/**
 *
 */
public class BinarySearchTree {

    Node root;

    public void insert(Node node) {

        root = insertHelper(root, node);
        System.out.println("Inserted root.data # " + root.data);
        System.out.println();

    };

    // we are using the helper method because we are using recursion
    private Node insertHelper(Node root, Node node) {

        int data = node.data;

        if (root == null) {
            root = node;
            System.out.println("data= " + data);
            System.out.println("root.data = " + root.data);
            return root;

        } else if (data < root.data) {

            System.out.println("data= " + data);
            System.out.println("root.data= " + root.data);

            // System.out.println("root.left.data= " + root.left.data);
            root.left = insertHelper(root.left, node);

        } else {
            System.out.println("data= " + data);
            System.out.println("root.data= " + root.data);

            root.right = insertHelper(root.right, node);
        }
        return      root;
    };

    // display method
    public void display() {

        displayHelper(root);
    };

    private void displayHelper(Node root) {

        if (root != null) {

            displayHelper(root.left);
            System.out.println(root.data);
            displayHelper(root.right);
        }

    };

    public boolean search(int data) {

        return true;
    }
```

RECURSION

```java
public static void main(String[] args) {
    // TODO Auto-generated method stub
    Scanner scanner = new Scanner(System.in);
    BinarySearchTree tree = new BinarySearchTree();

    tree.insert(new Node(5));
    tree.insert(new Node(2));
    tree.insert(new Node(3));
    tree.insert(new Node(6));
    tree.insert(new Node(9));
    tree.insert(new Node(7));
    tree.insert(new Node(8));

    tree.display();
```

```java
package com.practice01;

/**
 *
 */
public class BinarySearchTree {

    Node root;

    public void insert(Node node) {

        root = insertHelper(root, node);
        System.out.println("Inserted root.data # " + root.data);
        System.out.println();

    };

    // we are using the helper method because we are using recursion
    private Node insertHelper(Node root, Node node) {

        int data = node.data;

        if (root == null) {
            root = node;
            System.out.println("data= " + data);
            System.out.println("root.data = " + root.data);
            return root;

        } else if (data < root.data) {

            System.out.println("data= " + data);
            System.out.println("root.data= " + root.data);

            // System.out.println("root.left.data= " + root.left.data);
            root.left = insertHelper(root.left, node);

        } else {
            System.out.println("data= " + data);
            System.out.println("root.data= " + root.data);

            root.right = insertHelper(root.right, node);
        }
        return root;
    };

    // display method
    public void display() {

        displayHelper(root);
    };

    private void displayHelper(Node root) {

        if (root != null) {

            displayHelper(root.left);
            System.out.println(root.data);
            displayHelper(root.right);
        }

    };
```

Recursion


// display method
public void display() {
displayHelper(root);
};
private void displayHelper(Node root) {
if (root != null) {
displayHelper(root.left);
System.out.println(root.data);
displayHelper(root.right);
}
};
public boolean search(int data) {
return true;
}
//this method moves to the left of the subtree
Recursion
//this method moves to the right of the subtree
root.left points to null
6
3
8
2
5
7
9

BST Height and Insertion Order

- BST Trees height is the maximum edges from the root to any leaf (N- node binary tree has height = N-1 because the root is at height 0

Recursive BST insertion and removal

BST insertion and removal can also be implemented using recursion. The insertion algorithm uses recursion to traverse down the tree until the insertion location is found. The removal algorithm uses the recursive search functions to find the node and the node's parent, then removes the node from the tree. If the node to remove is an internal node with 2 children, the node's successor is recursively removed.

```
BSTInsert(tree, node) {
   if (tree→root is null)
      tree→root = node
   else
      BSTInsertRecursive(tree→root, node)
}

BSTInsertRecursive(parent, nodeToInsert) {
   if (nodeToInsert→key < parent→key) {
      if (parent→left is null)
         parent→left = nodeToInsert
      else
         BSTInsertRecursive(parent→left, nodeToInsert)
   }
   else {
      if (parent→right is null)
         parent→right = nodeToInsert
      else
         BSTInsertRecursive(parent→right, nodeToInsert)
   }
}

BSTRemove(tree, key) {
   node = BSTSearch(tree, key)
   parent = BSTGetParent(tree, node)
   BSTRemoveNode(tree, parent, node)
}

BSTRemoveNode(tree, parent, node) {
   if (node == null)
      return false

   // Case 1: Internal node with 2 children
   if (node→left != null && node→right != null) {
      // Find successor and successor's parent
      succNode = node→right
      successorParent = node
      while (succNode→left != null) {
         successorParent = succNode
         succNode = succNode→left
      }

      // Copy the value from the successor node
      node = Copy succNode

      // Recursively remove successor
      BSTRemoveNode(tree, successorParent, succNode)
   }

   // Case 2: Root node (with 1 or 0 children)
   else if (node == tree→root) {
      if (node→left != null)
         tree→root = node→left
      else
         tree→root = node→right
   }

   // Case 3: Internal with left child only
   else if (node→left != null) {
      // Replace node with node's left child
      if (parent→left == node)
         parent→left = node→left
      else
         parent→right = node→left
   }

   // Case 4: Internal with right child only OR leaf
   else {
      // Replace node with node's right child
      if (parent→left == node)
         parent→left = node→right
      else
         parent→right = node→right
   }

   return true
}
```