Software Engineering 2 (C++)

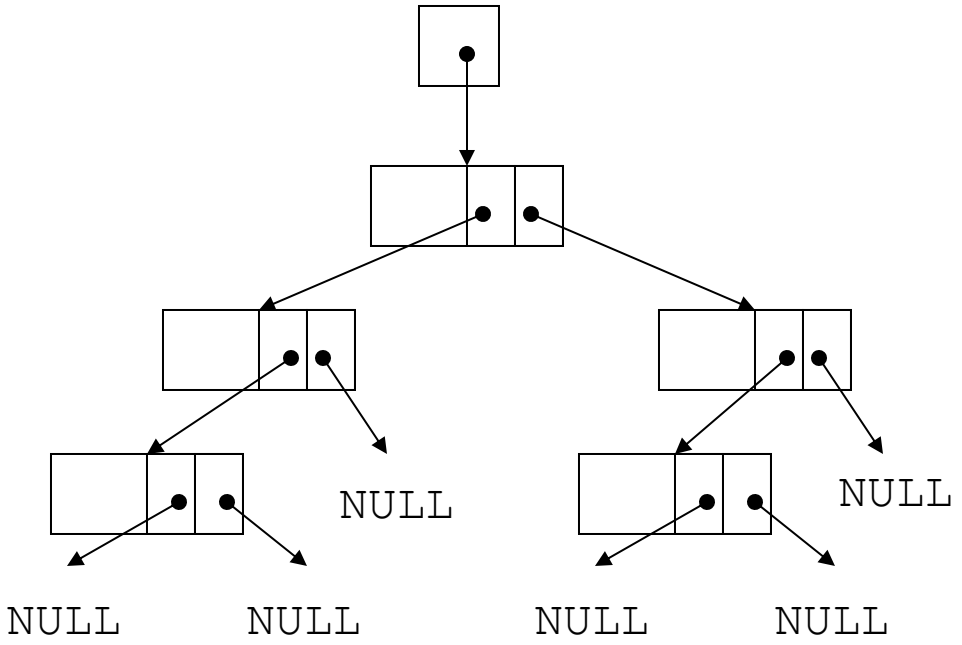
CSY2006 (Week 21)

Dr. Suraj Ajit

Definition and Application of Binary Trees

Definition and Application of Binary Trees

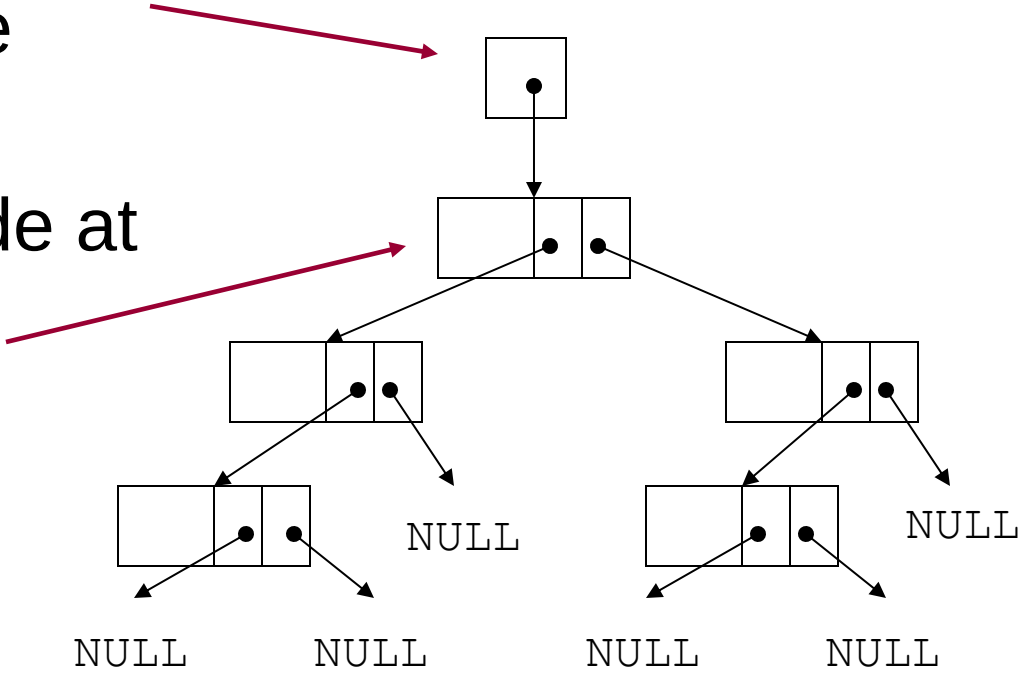
- Binary tree: a nonlinear linked list in which each node may point to 0, 1, or two other nodes
- Each node contains one or more data fields and two pointers



B-Trees make excellent data structures for searching large amounts of data efficient (fast). Arrays and Linked Lists are linear data structures which are slow. Data stored in a way that that makes binary search simple.

Binary Tree Terminology

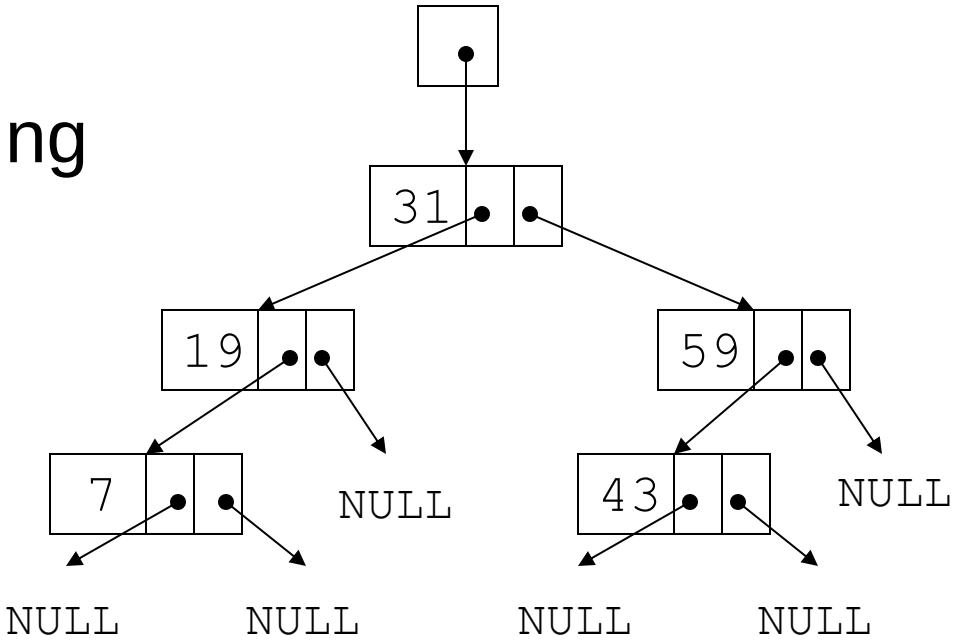
- Tree pointer: like a head pointer for a linked list, it points to the first node in the binary tree
- Root node: the node at the top of the tree



Binary Tree Terminology

- Leaf nodes: nodes that have no children

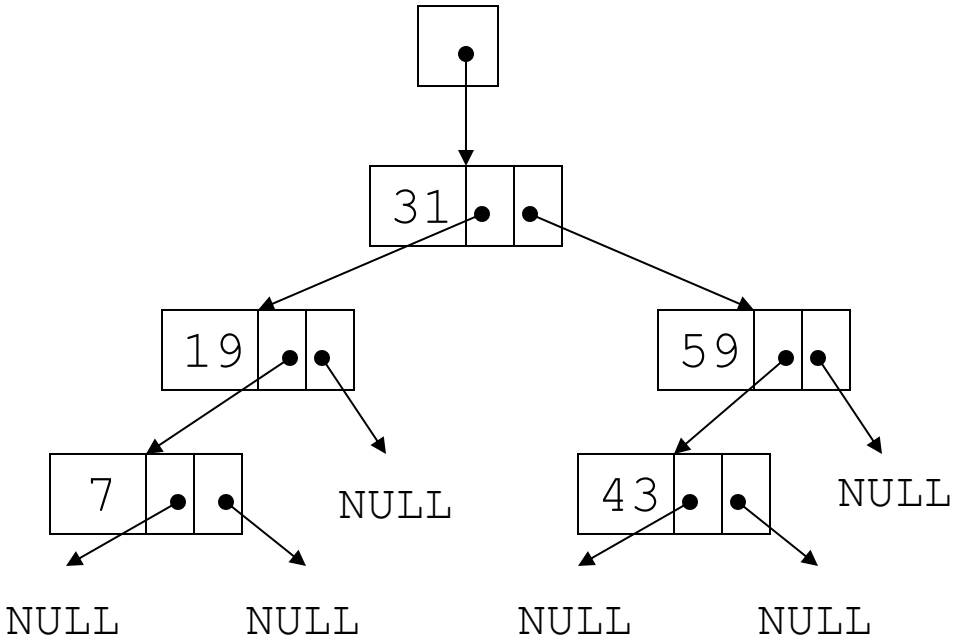
  The nodes containing `7` and `43` are leaf nodes



Binary Tree Terminology

- Child nodes, children: nodes below a given node

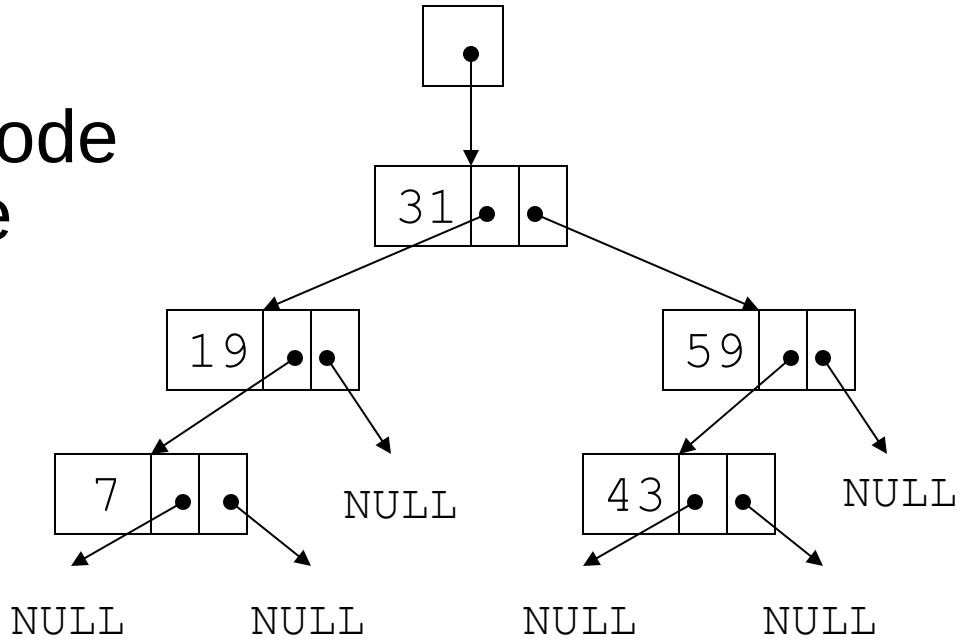
  The children of the node containing `31` are the nodes containing `19` and `59`



Binary Tree Terminology

- Parent node: node above a given node

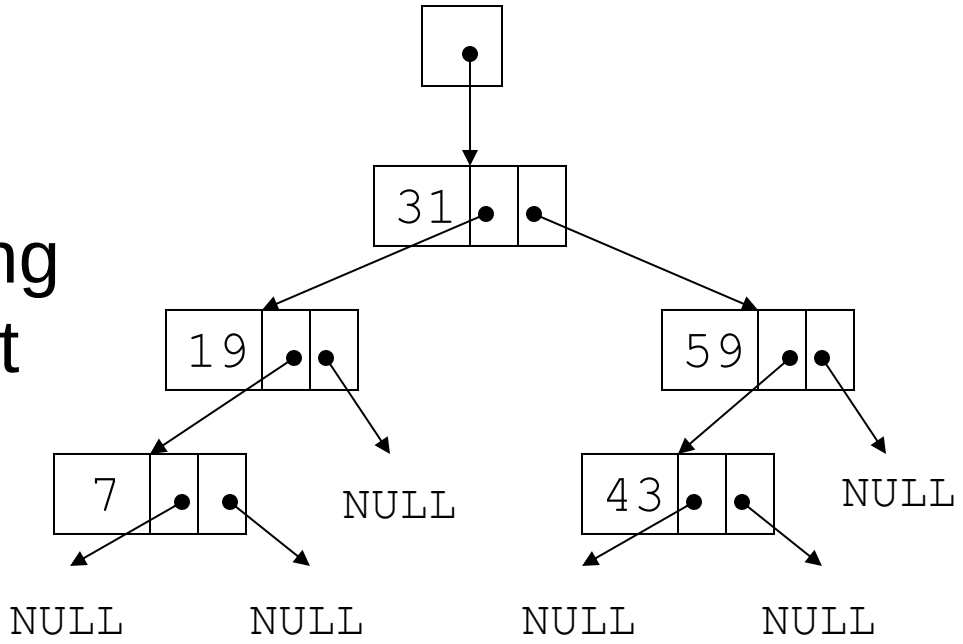
  The parent of the node containing `43` is the node containing `59`



Binary Tree Terminology

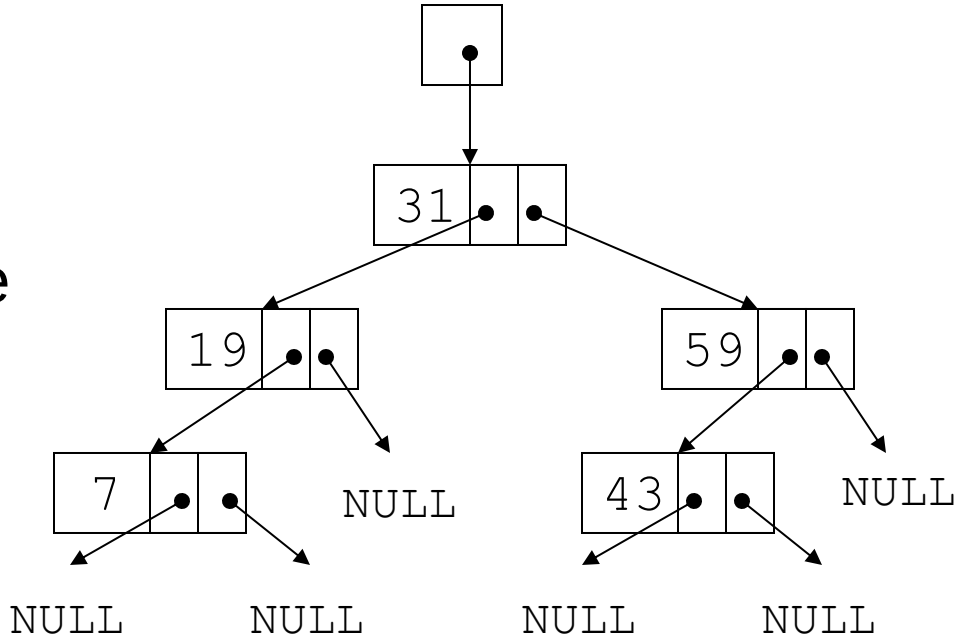
- Subtree: the portion of a tree from a node down to the leaves

  The nodes containing `19` and `7` are the left subtree of the node containing `31`



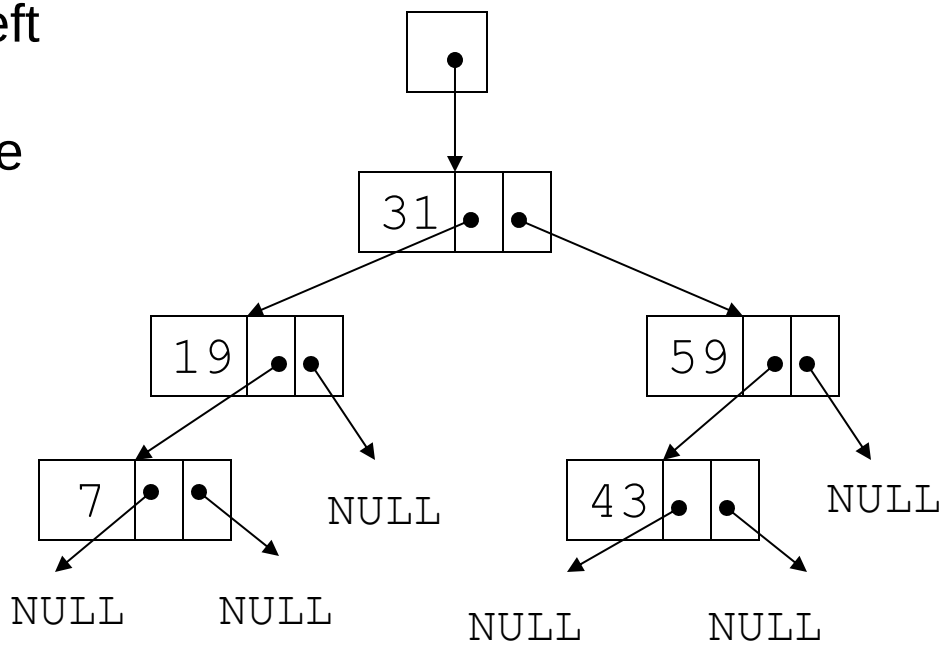
Uses of Binary Trees

- Binary search tree: data organized in a binary tree to simplify searches
- Left subtree of a node contains data values < the data in the node
- Right subtree of a node contains values > the data in the node



Searching in a Binary Tree

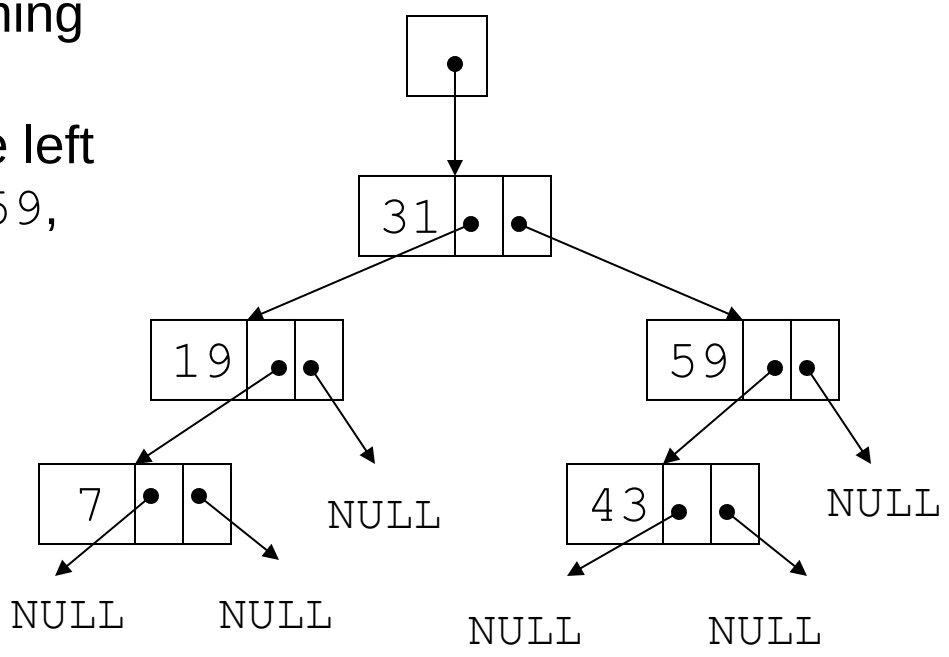
1) Start at root node
2) Examine node data:
   a) Is it desired value? Done
   b) Else, is desired data < node data? Repeat step 2 with left subtree
   c) Else, is desired data > node data? Repeat step 2 with right subtree
3) Continue until desired value found or `NULL` pointer reached



Searching in a Binary Tree

To locate the node containing `43`,

- Examine the root node (`31`) first
- Since `43 > 31`, examine the right child of the node containing `31`, (`59`)
- Since `43 < 59`, examine the left child of the node containing `59`, (`43`)
- The node containing 43 has been found



Binary Search Tree Operations

Binary Search Tree Operations

- Create a binary search tree – organize data into a binary search tree
- Insert a node into a binary tree – put node into tree in its correct position to maintain order
- Find a node in a binary tree – locate a node with particular data value
- Delete a node from a binary tree – remove a node and adjust links to maintain binary tree

Binary Search Tree Node

- A node in a binary tree is like a node in a linked list, with two node pointer fields:

```
struct TreeNode
{
   int value;
   TreeNode *left;
   TreeNode *right;
}
```

Creating a New Node

- Allocate memory for new node:

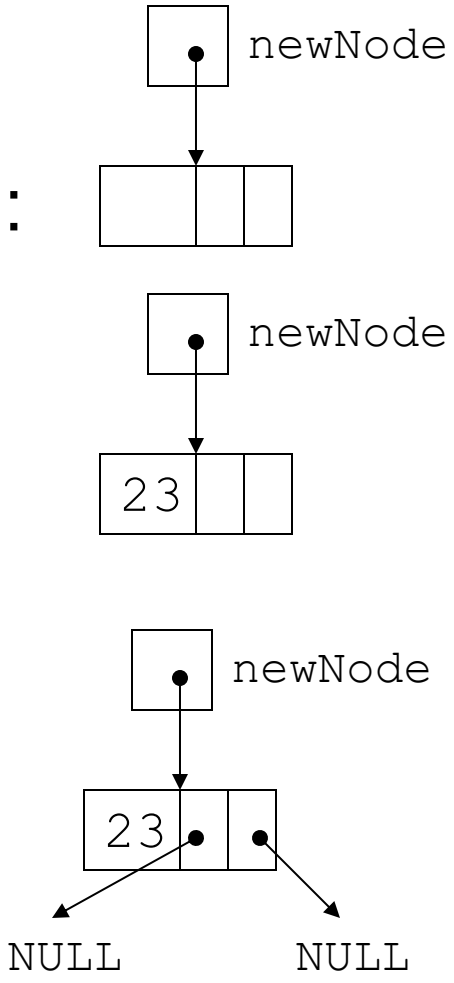
```
newNode = new TreeNode;
```

- Initialize the contents of the node:

```
newNode->value = num;
```

- Set the pointers to `NULL`:

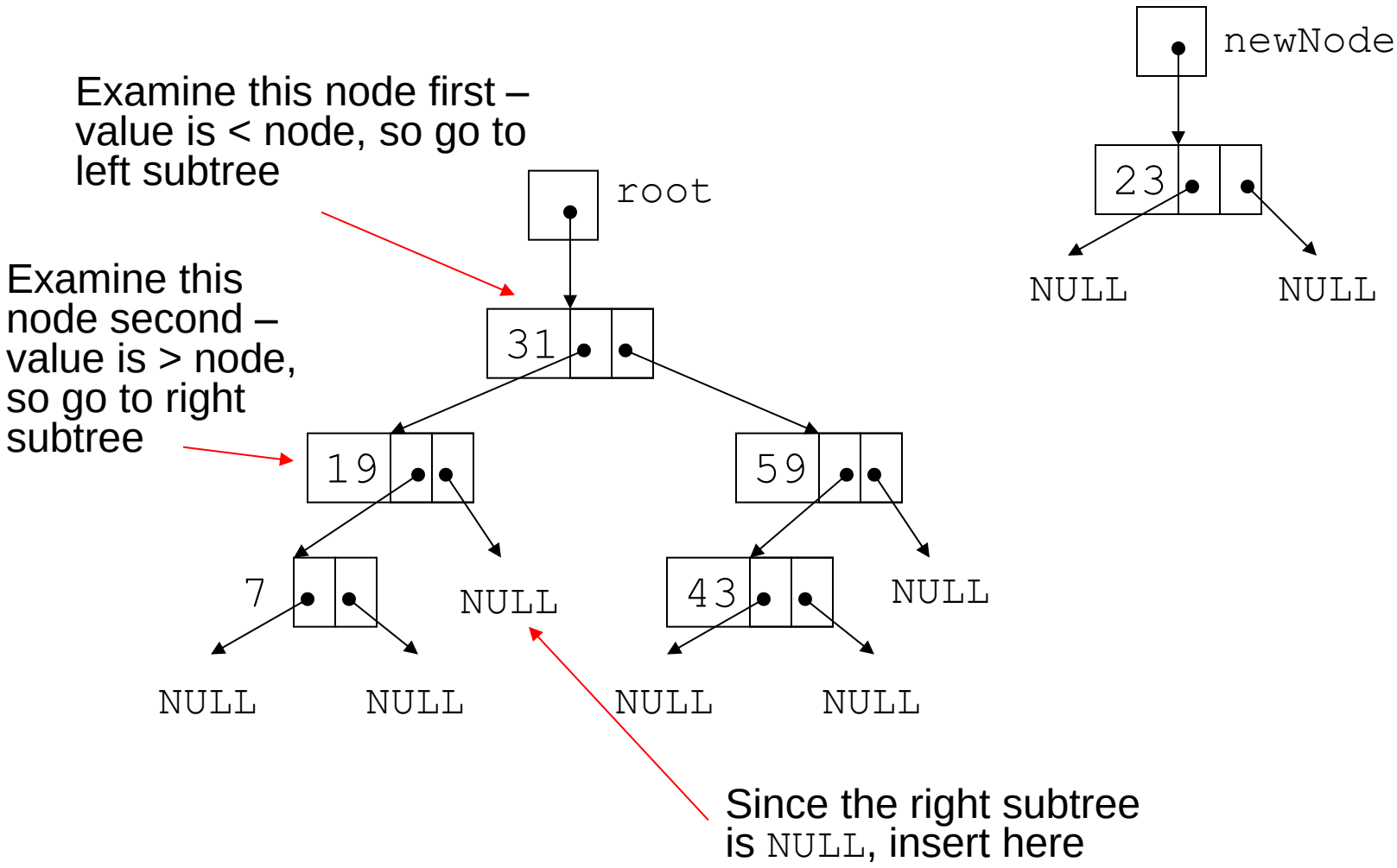
```
newNode->Left
= newNode->Right
= NULL;
```



Inserting a Node in a Binary Search Tree

1) If tree is empty, insert the new node as the root node
2) Else, compare new node against left or right child, depending on whether data value of new node is $<$ or $>$ root node
3) Continue comparing and choosing left or right subtree until `NULL` pointer found
4) Set this `NULL` pointer to point to new node

Inserting a Node in a Binary Search Tree

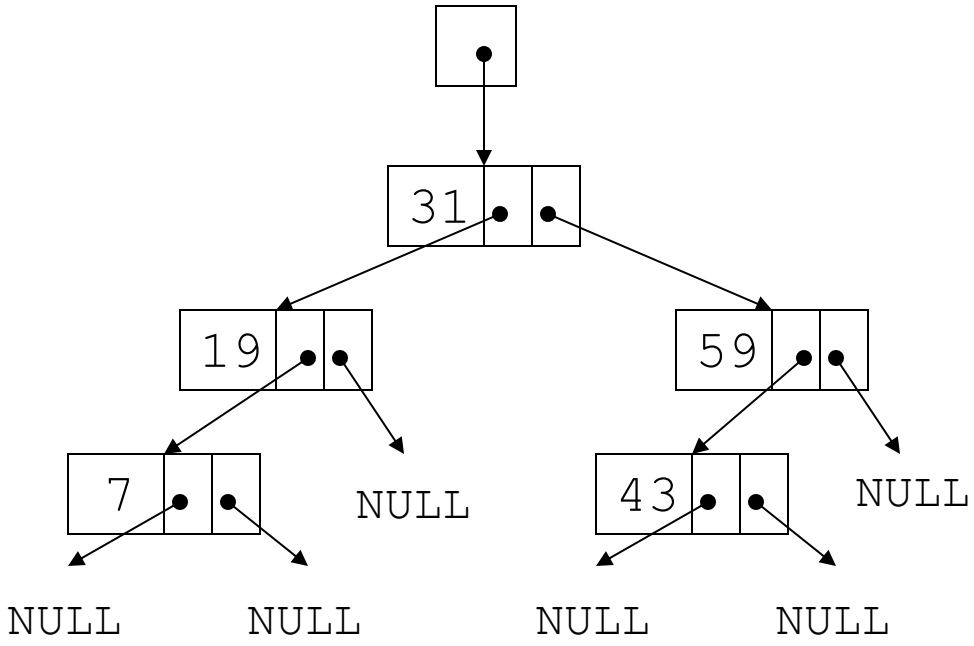


Traversing a Binary Tree

Three traversal methods:

1) Inorder:
    a) Traverse left subtree of node
    b) Process data in node
    c) Traverse right subtree of node
2) Preorder:
    a) Process data in node
    b) Traverse left subtree of node
    c) Traverse right subtree of node
3) Postorder:
    a) Traverse left subtree of node
    b) Traverse right subtree of node
    c) Process data in node

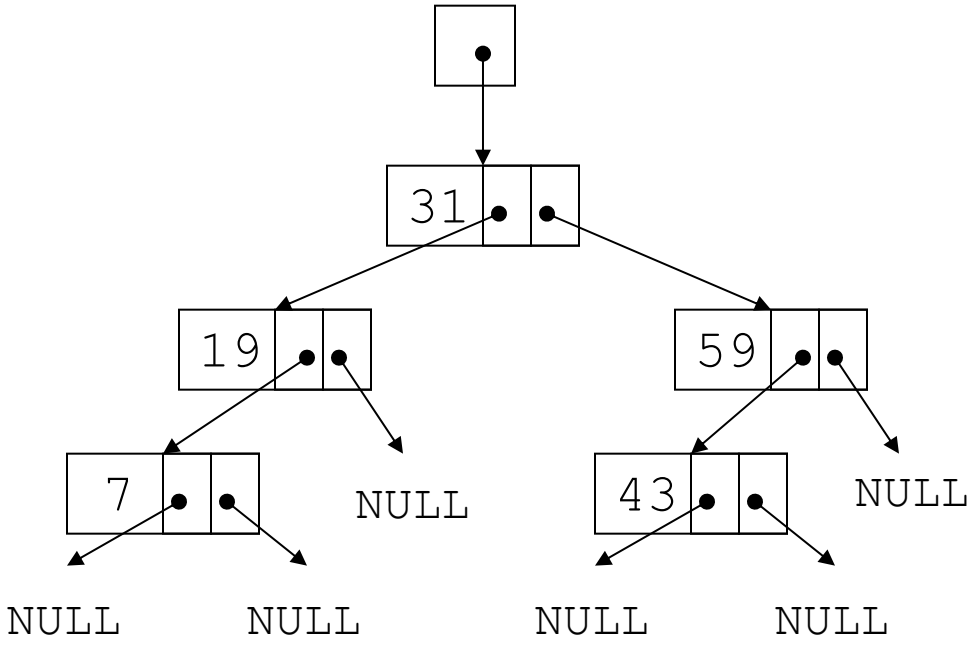
Traversing a Binary Tree



TRAVERSAL METHOD	NODES VISITED IN ORDER
Inorder	7, 19, 31, 43, 59
Preorder	31, 19, 7, 59, 43
Postorder	7, 19, 43, 59, 31

Searching in a Binary Tree

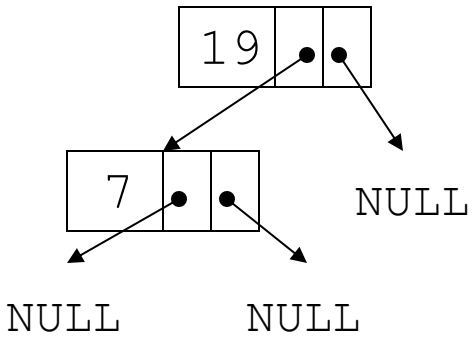
- Start at root node, traverse the tree looking for value
- Stop when value found or `NULL` pointer detected
- Can be implemented as a `bool` function



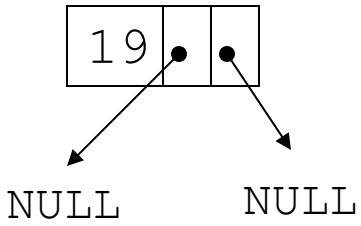
Search for `43`? return `true`
Search for `17`? return `false`

Deleting a Node from a Binary Tree – Leaf Node

- If node to be deleted is a leaf node, replace parent node's pointer to it with a `NULL` pointer, then delete the node



Deleting node with 7 – before deletion

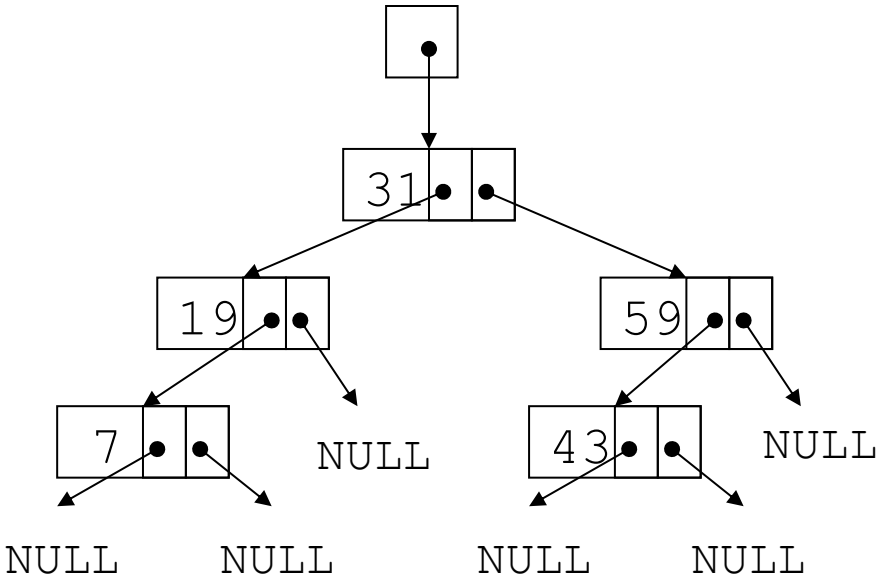


Deleting node with 7 – after deletion

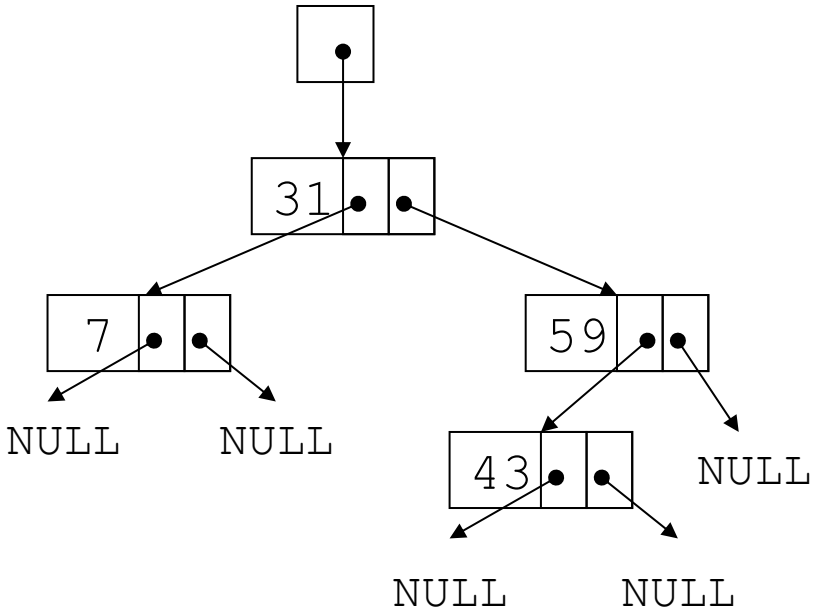
Deleting a Node from a Binary Tree – One Child

- If node to be deleted has one child node, adjust pointers so that parent of node to be deleted points to child of node to be deleted, then delete the node

Deleting a Node from a Binary Tree – One Child



Deleting node with 19
– before deletion

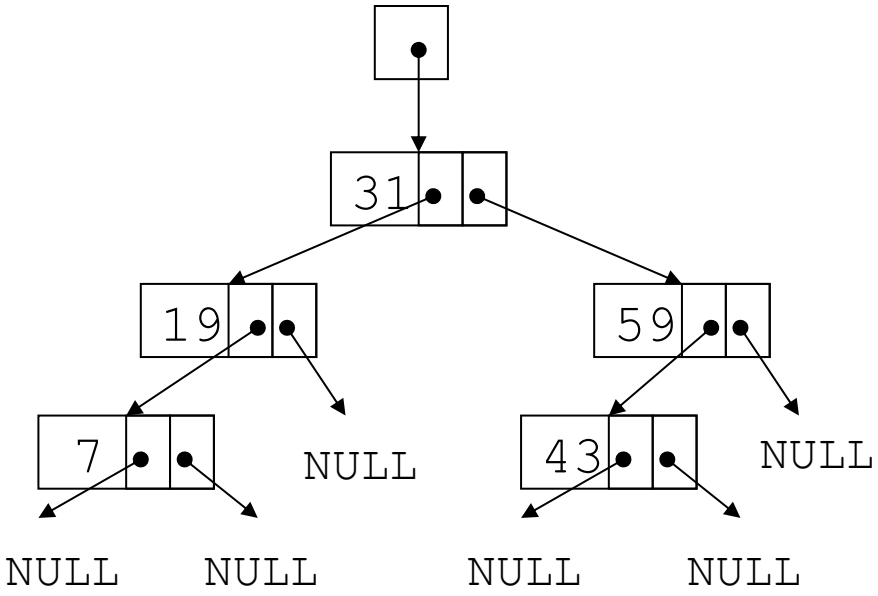


Deleting node with 19
– after deletion

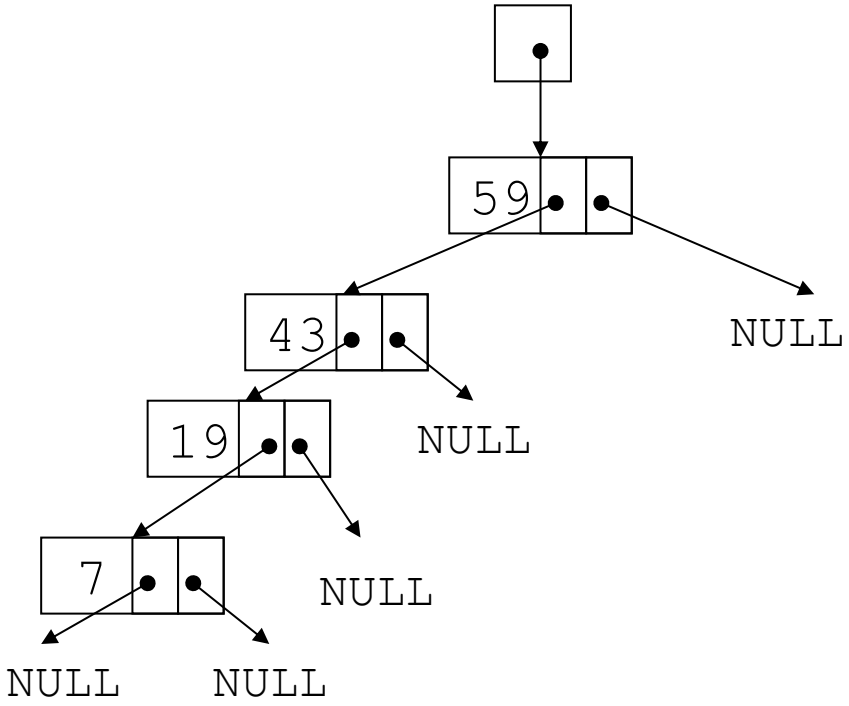
Deleting a Node from a Binary Tree – Two Children

- If node to be deleted has left and right children,
  - ‘Promote’ one child to take the place of the deleted node
  - Locate correct position for other child in subtree of promoted child
- Convention in text: promote the right child, position left subtree underneath

Deleting a Node from a Binary Tree – Two Children



Deleting node with 31
– before deletion



Deleting node with 31
– after deletion

Template Considerations for Binary Search Trees

Template Considerations for Binary Search Trees

- Binary tree can be implemented as a template, allowing flexibility in determining type of data stored
- Implementation must support relational operators $>$, $<$, and $==$ to allow comparison of nodes

Template Considerations for Binary Search Trees

```
1  // Specification file for the IntBinaryTree class
2  #ifndef INTBINARYTREE_H
3  #define INTBINARYTREE_H
4
5  class IntBinaryTree
6  {
7  private:
8     struct TreeNode
9     {
10       int value;          // The value in the node
11       TreeNode *left;     // Pointer to left child node
12       TreeNode *right;    // Pointer to right child node
13    };
14
15    TreeNode *root;        // Pointer to the root node
16
17    // Private member functions
18    void insert(TreeNode *&, TreeNode *&);
19    void destroySubTree(TreeNode *);
20    void deleteNode(int, TreeNode *&);
21    void makeDeletion(TreeNode *&);
22    void displayInOrder(TreeNode *) const;
23    void displayPreOrder(TreeNode *) const;
24    void displayPostOrder(TreeNode *) const;
25
```

Template Considerations for Binary Search Trees

```
1  #ifndef BINARYTREE_H
2  #define BINARYTREE_H
3  #include <iostream>
4  using namespace std;
5  // Stack template
6  template <class T>
7  class BinaryTree{
8  private:
9     struct TreeNode{
10       T value;           // The value in the node
11       TreeNode *left;    // Pointer to left child node
12       TreeNode *right;   // Pointer to right child node
13    };
14
15    TreeNode *root;       // Pointer to the root node
16
17    // Private member functions
18    void insert(TreeNode *&, TreeNode *&);
19    void destroySubTree(TreeNode *);
20    void deleteNode(T, TreeNode *&);
21    void makeDeletion(TreeNode *&);
22    void displayInOrder(TreeNode *) const;
23    void displayPreOrder(TreeNode *) const;
24    void displayPostOrder(TreeNode *) const;
```

Background Information (Reference Only)

Introduction to Recursion

Introduction to Recursion

- A recursive function contains a call to itself:

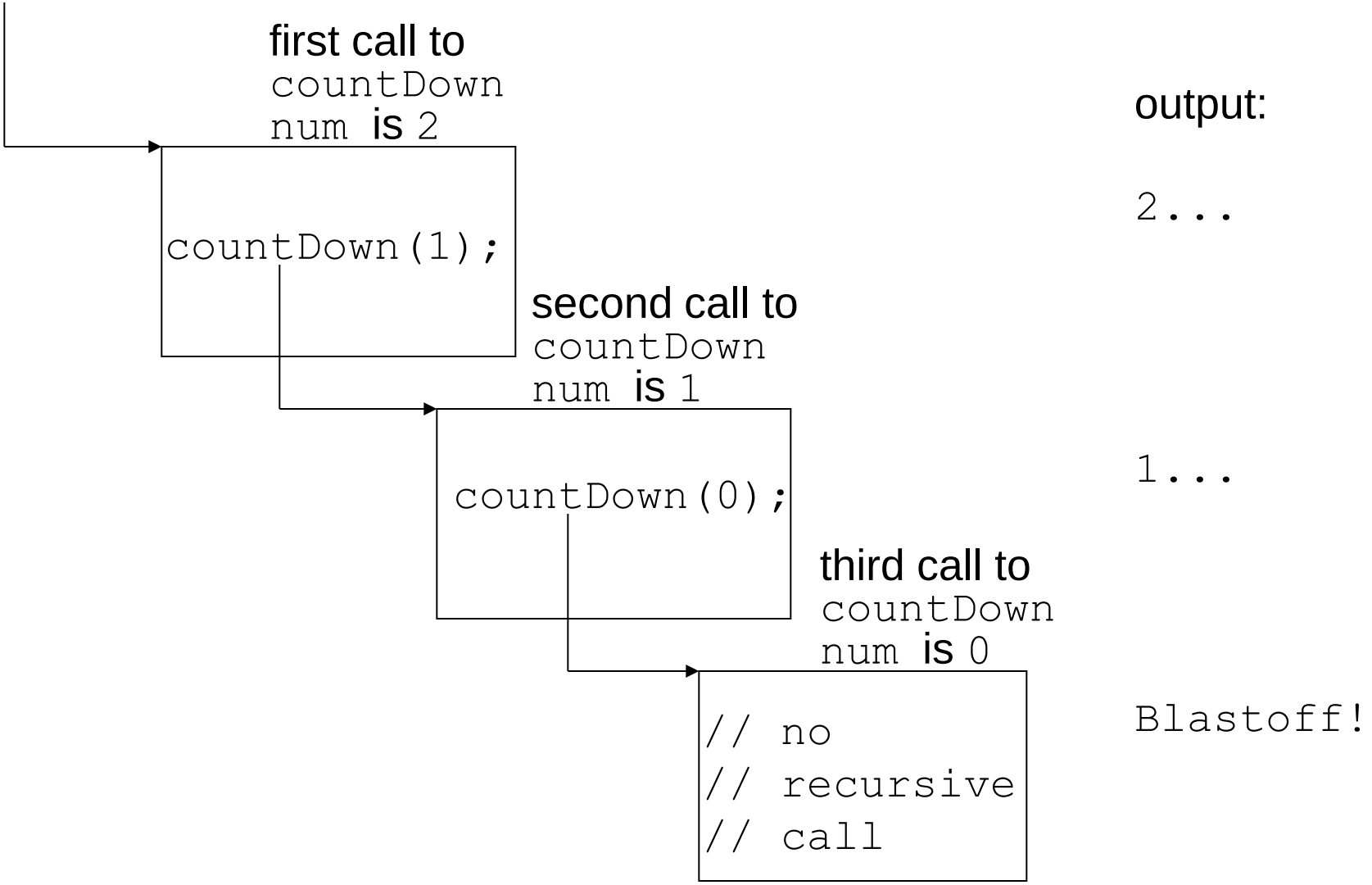
```
void countDown(int num)
{
   if (num == 0)
      cout << "Blastoff!";
   else
   {
      cout << num << "...\n";
      countDown(num-1); // recursive
   }                    // call
}
```

What Happens When Called?

If a program contains a line like `countDown(2);`

1. `countDown(2)` generates the output `2...`, then it calls `countDown(1)`
2. `countDown(1)` generates the output `1...`, then it calls `countDown(0)`
3. `countDown(0)` generates the output `Blastoff!`, then returns to `countDown(1)`
4. `countDown(1)` returns to `countDown(2)`
5. `countDown(2)` returns to the calling function

What Happens When Called?



Solving Problems with Recursion

Recursive Functions - Purpose

- Recursive functions are used to reduce a complex problem to a simpler-to-solve problem.
- The simpler-to-solve problem is known as the base case
- Recursive calls stop when the base case is reached

Stopping the Recursion

- A recursive function must always include a test to determine if another recursive call should be made, or if the recursion should stop with this call
- In the sample program, the test is:

```
if (num == 0)
```

Stopping the Recursion

```
void countDown(int num)
{
   if (num == 0) // test
      cout << "Blastoff!";
   else
   {
      cout << num << "...\n";
      countDown(num-1); // recursive
   }                    // call
}
```

Stopping the Recursion

- Recursion uses a process of breaking a problem down into smaller problems until the problem can be solved
- In the `countDown` function, a different value is passed to the function each time it is called
- Eventually, the parameter reaches the value in the test, and the recursion stops

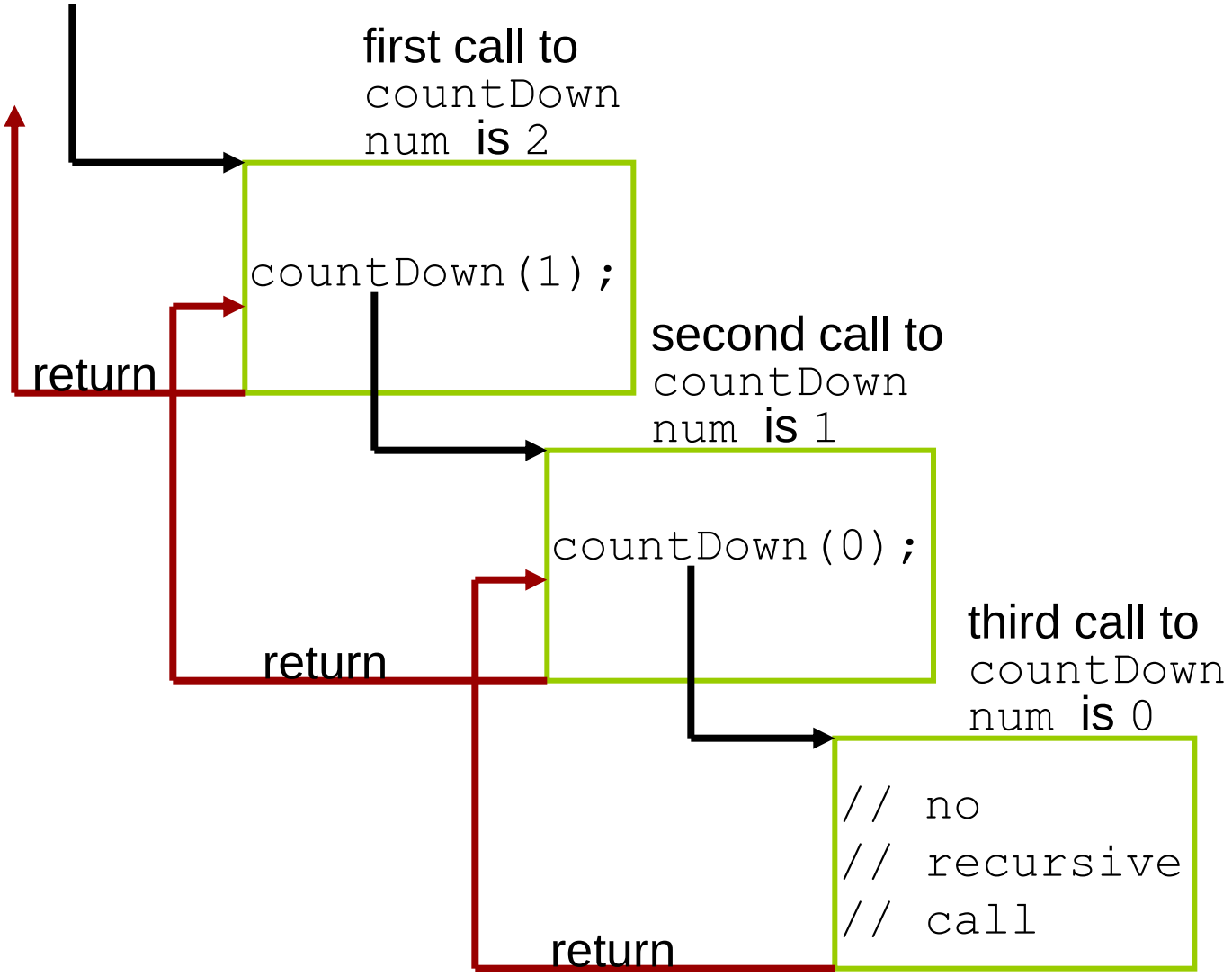
Stopping the Recursion

```
void countDown(int num)
{
   if (num == 0)
      cout << "Blastoff!";
   else
   {
      cout << num << "...\n";
      countDown(num-1);// note that the value
   }                   // passed to recursive
}                      // calls decreases by
                       // one for each call
```

What Happens When Called?

- Each time a recursive function is called, a new copy of the function runs, with new instances of parameters and local variables created
- As each copy finishes executing, it returns to the copy of the function that called it
- When the initial copy finishes executing, it returns to the part of the program that made the initial call to the function

What Happens When Called?



output:

```
2...

1...

Blastoff!
```

Types of Recursion

- Direct
  - a function calls itself
- Indirect
  - function A calls function B, and function B calls function A
  - function A calls function B, which calls …, which calls function A

The Recursive Factorial Function

- The factorial function:

  `n! = n*(n-1)*(n-2)*...*3*2*1` if `n > 0`

  `n! = 1` if `n = 0`

- Can compute factorial of `n` if the factorial of `(n-1)` is known:

  `n! = n * (n-1)!`

- `n = 0` is the base case

The Recursive Factorial Function

```
int factorial (int num)
{
  if (num > 0)
     return num * factorial(num - 1);
  else
     return 1;
}
```

Program 19-3

```
1  // This program demonstrates a recursive function to
2  // calculate the factorial of a number.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototype
7  int factorial(int);
8
9  int main()
10 {
11    int number;
12
13    // Get a number from the user.
14    cout << "Enter an integer value and I will display\n";
15    cout << "its factorial: ";
16    cin >> number;
17
18    // Display the factorial of the number.
19    cout << "The factorial of " << number << " is ";
20    cout << factorial(number) << endl;
21    return 0;
22 }
23
```

Program 19-3 (Continued)

```
24  //*********************************************************
25  // Definition of factorial. A recursive function to calculate *
26  // the factorial of the parameter n.                       *
27  //*********************************************************
28
29  int factorial(int n)
30  {
31     if (n == 0)
32        return 1;                    // Base case
33     else
34        return n * factorial(n - 1);  // Recursive case
35  }
```

Program Output with Example Input Shown in Bold

```
Enter an integer value and I will display
its factorial: 4 [Enter]
The factorial of 4 is 24
```

The Recursive gcd Function

The Recursive gcd Function

- Greatest common divisor (gcd) is the largest factor that two integers have in common
- Computed using Euclid's algorithm:

  `gcd(x, y) = y` if `y` divides `x` evenly

  `gcd(x, y) = gcd(y, x % y)` otherwise
- `gcd(x, y) = y` is the base case

The Recursive gcd Function

```
int gcd(int x, int y)
{
    if (x % y == 0)
        return y;
    else
        return gcd(y, x % y);
}
```

Solving Recursively Defined Problems

Solving Recursively Defined Problems

- The natural definition of some problems leads to a recursive solution
- Example: Fibonacci numbers:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```

- After the starting `0, 1`, each number is the sum of the two preceding numbers
- Recursive solution:

```
fib(n) = fib(n – 1) + fib(n – 2);
```

- Base cases: `n <= 0, n == 1`

Solving Recursively Defined Problems

```
int fib(int n)
{
   if (n <= 0)
       return 0;
   else if (n == 1)
       return 1;
   else
        return fib(n – 1) + fib(n – 2);
}
```

Recursive Linked List Operations

Recursive Linked List Operations

- Recursive functions can be members of a linked list class
- Some applications:
  - Compute the size of (number of nodes in) a list
  - Traverse the list in reverse order

Counting the Nodes in a Linked List

- Uses a pointer to visit each node
- Algorithm:
  - pointer starts at head of list
  - If pointer is NULL, return 0 (base case)

    else, return 1 + number of nodes in the list pointed to by current node

- See the `NumberList` class

The `countNodes` function, a private member function

```
173  int NumberList::countNodes(ListNode *nodePtr) const
174  {
175     if (nodePtr != NULL)
176        return 1 + countNodes(nodePtr->next);
177     else
178        return 0;
179  }
```

The `countNodes` function is executed by the public `numNodes` function:

```
int numNodes() const
   { return countNodes(head); }
```

Contents of a List in Reverse Order

- Algorithm:
  - pointer starts at head of list
  - If the pointer is NULL, return (base case)
  - If the pointer is not NULL, advance to next node
  - Upon returning from recursive call, display contents of current node

The `showReverse` function, a private member function

```
187  void NumberList::showReverse(ListNode *nodePtr) const
188  {
189     if (nodePtr != NULL)
190     {
191        showReverse(nodePtr->next);
192        cout << nodePtr->value << " ";
193     }
194  }
```

The `showReverse` function is executed by the public `displayBackwards` function:

```
void displayBackwards() const
   { showReverse(head); }
```

A Recursive Binary Search Function

A Recursive Binary Search Function

- Binary search algorithm can easily be written to use recursion
- Base cases: desired value is found, or no more array elements to search
- Algorithm (array in ascending order):
  - If middle element of array segment is desired value, then done
  - Else, if the middle element is too large, repeat binary search in first half of array segment
  - Else, if the middle element is too small, repeat binary search on the second half of array segment

A Recursive Binary Search Function (Continued)

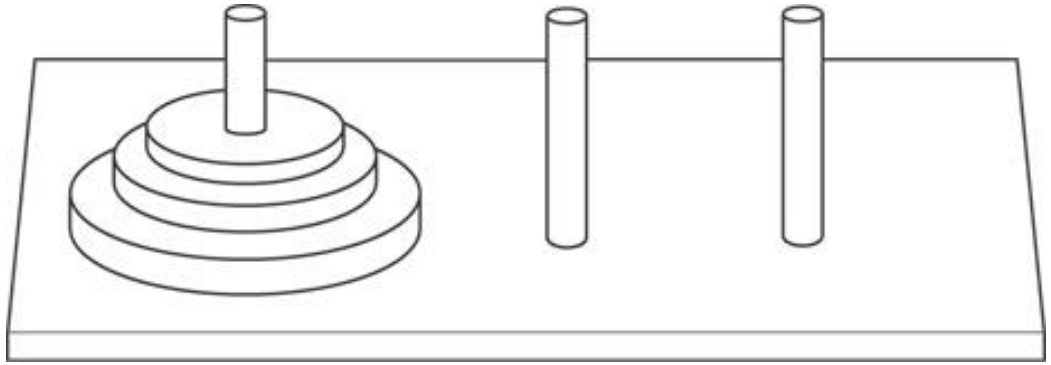
```
 int binarySearch(int array[], int first, int last, int value)
{
   int middle;   // Mid point of search

   if (first > last)
      return -1;
   middle = (first + last) / 2;
   if (array[middle] == value)
      return middle;
   if (array[middle] < value)
      return binarySearch(array, middle+1,last,value);
   else
      return binarySearch(array, first,middle-1,value);
}
```

The Towers of Hanoi

The Towers of Hanoi

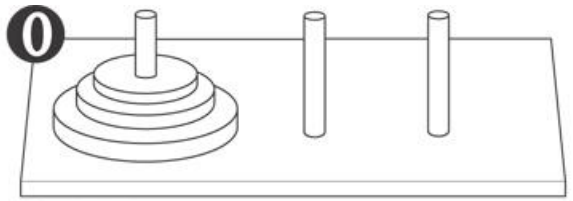
- The Towers of Hanoi is a mathematical game that is often used to demonstrate the power of recursion.
- The game uses three pegs and a set of discs, stacked on one of the pegs.



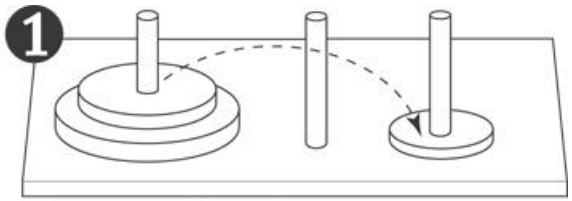
The Towers of Hanoi

- The object of the game is to move the discs from the first peg to the third peg. Here are the rules:
  - Only one disc may be moved at a time.
  - A disc cannot be placed on top of a smaller disc.
  - All discs must be stored on a peg except while being moved.

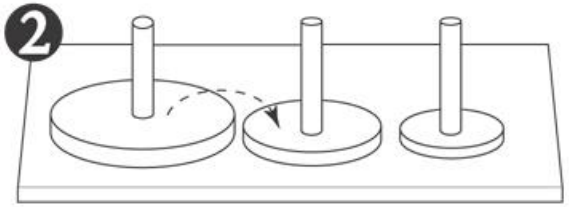
Moving Three Discs



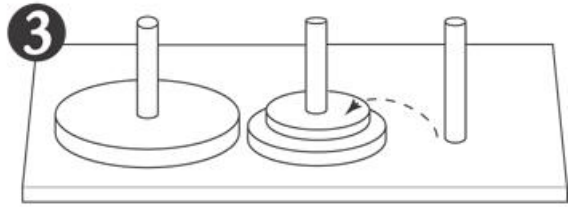
Original setup.



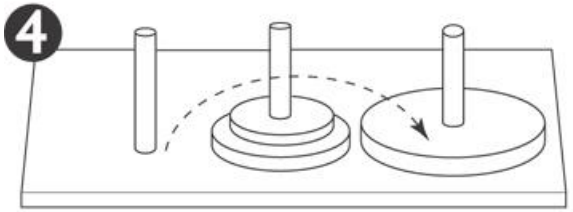
First move: Move disc 1 to peg 3.



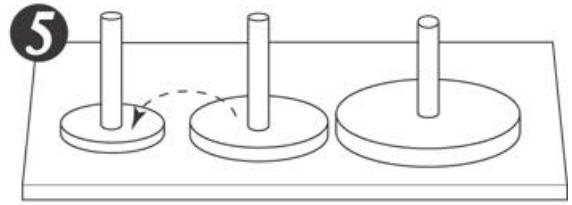
Second move: Move disc 2 to peg 2.



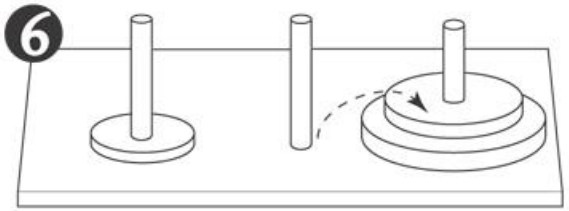
Third move: Move disc 1 to peg 2.



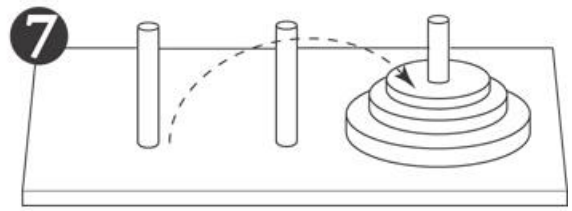
Fourth move: Move disc 3 to peg 3.



Fifth move: Move disc 1 to peg 1.



Sixth move: Move disc 2 to peg 3.



Seventh move: Move disc 1 to peg 3.

The Towers of Hanoi

- The following statement describes the overall solution to the problem:
  - *Move n discs from peg 1 to peg 3 using peg 2 as a temporary peg.*

The Towers of Hanoi

- Algorithm
  - *To move n discs from peg A to peg C, using peg B as a temporary peg:*
    If n > 0 Then
    Move n – 1 discs from peg A to peg B, using peg C as a temporary peg.

    Move the remaining disc from the peg A to peg C.

    Move n – 1 discs from peg B to peg C, using peg A as a temporary peg.

    End If

Program 19-10

```
1  // This program displays a solution to the Towers of
2  // Hanoi game.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototype
7  void moveDiscs(int, int, int, int);
8
9  int main()
10 {
11    const int NUM_DISCS = 3;    // Number of discs to move
12    const int FROM_PEG = 1;     // Initial "from" peg
13    const int TO_PEG = 3;       // Initial "to" peg
14    const int TEMP_PEG = 2;     // Initial "temp" peg
15
```

Program 19-10 *(continued)*

```
16      // Play the game.
17      moveDiscs(NUM_DISCS, FROM_PEG, TO_PEG, TEMP_PEG);
18      cout << "All the pegs are moved!\n";
19      return 0;
20   }
21
22   //*************************************************
23   // The moveDiscs function displays a disc move in   *
24   // the Towers of Hanoi game.                        *
25   // The parameters are:                              *
26   //    num:     The number of discs to move.         *
27   //    fromPeg: The peg to move from.                *
28   //    toPeg:   The peg to move to.                  *
29   //    tempPeg: The temporary peg.                   *
30   //*************************************************
31
32   void moveDiscs(int num, int fromPeg, int toPeg, int tempPeg)
33   {
34      if (num > 0)
35      {
36         moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
37         cout << "Move a disc from peg " << fromPeg
38              << " to peg " << toPeg << endl;
39         moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
40      }
41   }
```

Program 19-10 (Continued)

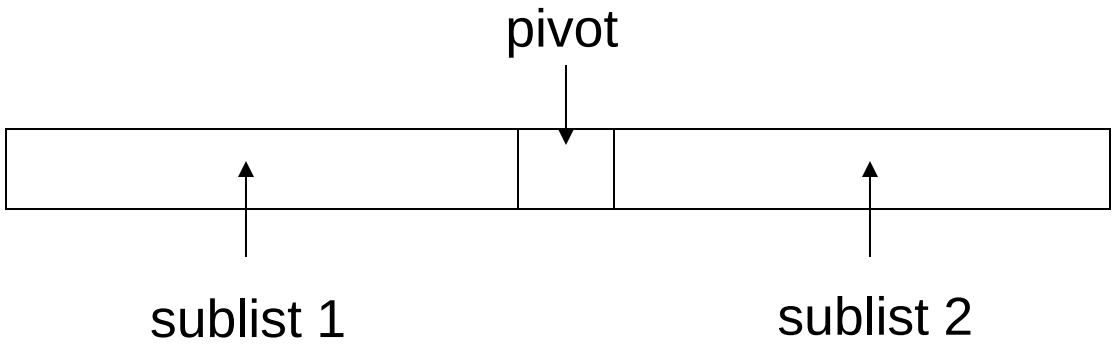
Program Output

```
Move a disc from peg 1 to peg 3
Move a disc from peg 1 to peg 2
Move a disc from peg 3 to peg 2
Move a disc from peg 1 to peg 3
Move a disc from peg 2 to peg 1
Move a disc from peg 2 to peg 3
Move a disc from peg 1 to peg 3
All the pegs are moved!
```

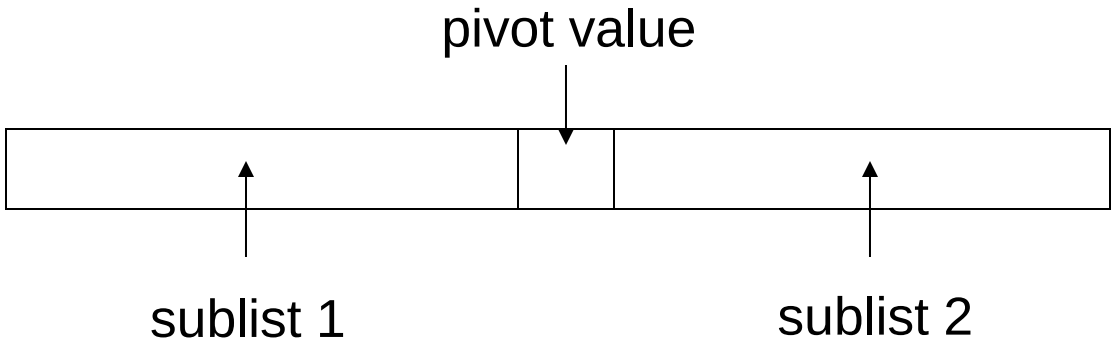
The QuickSort Algorithm

The QuickSort Algorithm

- Recursive algorithm that can sort an array or a linear linked list
- Determines an element/node to use as pivot value:



The QuickSort Algorithm



- Once pivot value is determined, values are shifted so that elements in sublist1 are < pivot and elements in sublist2 are > pivot
- Algorithm then sorts sublist1 and sublist2
- Base case: sublist has size 1

Exhaustive and Enumeration Algorithms

Exhaustive and Enumeration Algorithms

- Exhaustive algorithm: search a set of combinations to find an optimal one

  Example: change for a certain amount of money that uses the fewest coins

- Uses the generation of all possible combinations when determining the optimal one.

Recursion vs. Iteration

Recursion vs. Iteration

- Benefits (+), disadvantages(-) for recursion:
  - + Models certain algorithms most accurately
  - + Results in shorter, simpler functions
  - – May not execute very efficiently
- Benefits (+), disadvantages(-) for iteration:
  - + Executes more efficiently than recursion
  - – Often is harder to code or understand