The Technical Interview

Coding Challenges

Andrew Ferlitsch

The Technical Interview

Coding Challenges

Andrew Ferlitsch

The coding challenges in this section are basic algorithms. A coding portion of a technical interview will likely start with one of these types of algorithms. Typically, after asking you to code the basic algorithm, you will be ask to make improvements or other variations. If you are unable to pass this part of the coding portion, you will probably not continue to the next round of an interview.

1. Prime Numbers

The most basic coding example you might get asked is to write an algorithm to output a series of prime numbers. Prime numbers are numbers that are only divisible by one and itself. The algorithm is a straight forward iterative algorithm.

The number one is a prime number, since it can only be divisible by itself. The number two is also a prime number, since there are no numbers between one and two to divide by. Therefore, I generally like to start the iterative process at three, though you may choose to start it at two.

Algorithm

1. For each number, we attempt to divide it by every number less than it, except for one.

2. The integer modulo operator is used to test if there is a remainder from the integer division.
3. If there is no remainder, it is divisible by the number and therefore not a prime.
4. If each of the numbers it is divided by has a remainder, it is a prime.

Python

```
print( "Prime Numbers between 1 and 100")

# Prime Numbers are numbers only divisible by 1 and itself.
print( "1" )
print( "2" )

# Primes for numbers above 2
for number in range( 3, 101 ):
        # Attempt to divide this number by every number between 2 and one less than itself
        for div in range(2,number):
                # use the mod function to see if their is a remainder in the division
                if ( number % div ) == 0:
                        break
        else:
                print(number)
```

Java

```
public class prime {
        public static void main( String args[] ) {
                System.out.println( "Prime Numbers between 1 and 100");

                // Prime Numbers are numbers only divisible by 1 and itself.
                System.out.println( "1" );
                System.out.println( "2" );

                // Primes for numbers above 2
                for ( int number = 3; number <=100; number++ ) {
                        // Attempt to divide this number by every number between 2 and one less than itself
                        int div = 2;
                        for ( /**/; div < number; div++ ) {
                                // use the mod function to see if their is a remainder in the division
                                if ( ( number % div ) == 0 )
                                        break;
                        }
                        if ( div == number )
```

```
                                System.out.println(number);
                        }
                }
        }
```

Can we make an improvement? Yes. Since we know all even numbers are divisible by two, we can skip checking even numbers and only check odd numbers. Likewise, since we know odd numbers are not divisible by even numbers, we can skip dividing by even numbers and only divide by odd numbers.

Algorithm - Skip Even Numbers

1. For each odd number, we attempt to divide it by every odd number less than it, except for one.
2. The integer modulo operator is used to test if there is a remainder from the integer division.
3. If there is no remainder, it is divisible by the number and therefore not a prime.
4. If each of the numbers it is divided by has a remainder, it is a prime.

Python

```
print( "Prime Numbers between 1 and 100")

# Prime Numbers are numbers only divisible by 1 and itself.
print( "1" )
print( "2" )
print( "3" )

# Primes for numbers above 5
for number in range( 5, 101, 2 ):
        # Attempt to divide this number by every odd number between 2 and one less than itself
        for div in range(3,number,2):
                # use the mod function to see if their is a remainder in the division
                if ( number % div ) == 0:
                        break
        else:
                print(number)
```

Java

```
public class prime {
        public static void main( String args[] ) {
                System.out.println( "Prime Numbers between 1 and 100");

                // Prime Numbers are numbers only divisible by 1 and itself.
                System.out.println( "1" );
```

```
                System.out.println( "2" );
                System.out.println( "3" );

                // Primes for numbers above 5
                for ( int number = 5; number <=100; number += 2 ) {
                        int div = 3;
                        // Attempt to divide this number by every odd number between 3 and one less than itself
                        for ( /**/; div < number; div += 2 ) {
                                // use the mod function to see if their is a remainder in the division
                                if ( ( number % div ) == 0 )
                                        break;
                        }
                        if ( div == number )
                                System.out.println(number);
                }
        }
}
```

Can we make another improvement? Yes. Since we are skipping even numbers, we know that each number to be prime must be divisible by at least the number three. Therefore, any divisible number must be one-third or less the value of the number. We can cut down on the number of iterations by only dividing the first 1/3 of values less than the number.

Algorithm - Skip Even Numbers, and divide only by numbers one-third or less than the number.

1. For each odd number, we attempt to divide it by every odd number that is less than one-third of the number, except for one and two.
2. The integer modulo operator is used to test if there is a remainder from the integer division.
3. If there is no remainder, it is divisible by the number and therefore not a prime.
4. If each of the numbers it is divided by has a remainder, it is a prime.

Python

```
print( "1" )
print( "2" )
print( "3" )
for number in range( 5, 101, 2 ):
        # Attempt to divide this number by every number between 3 and one third less than itself
        third = int( (number / 3 ) ) + 1
        for div in range(3,third,2):
                # use the mod function to see if their is a remainder in the division
                if ( number % div ) == 0:
                        break
        else:
```

```
                print(number)
```

Java

```
public class prime {
        public static void main( String args[] ) {
                System.out.println( "Prime Numbers between 1 and 100");

                // Prime Numbers are numbers only divisible by 1 and itself.
                System.out.println( "1" );
                System.out.println( "2" );
                System.out.println( "3" );

                // Primes for numbers above 5
                for ( int number = 5; number <=100; number += 2 ) {
                        int div = 3;
                        int third = ( number / 3 );     // calculate one-third of the number
                        // Attempt to divide this number by every number between 3 and one third of the number
                        for ( /**/; div <= third; div += 2) {
                                // use the mod function to see if their is a remainder in the division
                                if ( ( number % div ) == 0 )
                                        break;
                        }
                        if ( div >= third )
                                System.out.println(number);
                }
        }
}
```

2. Fibonacci Sequence

One of the most common and basic coding examples is to code a solution for the Fibonacci sequence. A Fibonacci sequence is a function where F(N) equals F(n-1) + F(n-2); that is the summation of the previous two values in the sequence. Originally, the seed sequence was F(0) = 1 and F(1) = 1, but modern solutions use F(0) = 0 and F(1) = 1. This sequence can be written in a very short recursive algorithm.

Algorithm - Recursive

1. If n = 0, then return 0.
2. If n = 1, then return 1.
3. If n > 1, return the sum of F(n-1) + F(n-2).

Python

```
# Fibonacci Sequence
# Is a sum (addition) of the previous two numbers in the sequence, as
#       F(n) = F(n-1) + F(n-2)
# Where
#       F(0) = 0 and F(1) = 1
# Hence
#       F(2) = 0 + 1 [F(1) + F(0)] = 1
#       F(3) = 1 + 1 [F(2) + F(1)] = 2

# Recursive Solution
def Fibonacci(n):
        # F(0) and F(1) case
        if n==0:
                return 0
        if n==1:
                return 1
        return Fibonacci(n-1) + Fibonacci(n-2)

# Fibonacci Sequence to F(10)
for number in range( 0, 10 ):
        print( Fibonacci(number) )
```

Java

```
public class fibonacci {
        public static void main( String args[] ) {
                System.out.println( "Fibonacci Sequence to F(10)");

                for ( int number = 0; number < 10; number++ )
                        System.out.println( Fibonacci( number ) );

        }

        // Recursive Solution
        public static int Fibonacci( int n) {
                if ( n == 0 ) return 0;
                if ( n == 1 ) return 1;
                return Fibonacci( n - 1 ) + Fibonacci( n - 2 );
        }
}
```

It is common to be asked if you can write the algorithm for an iterative solution (looping - no recursion).

Algorithm - Iterative

1. If n = 0, then return 0.
2. If n = 1, then return 1.
3. If n > 1, then set last F(n-2) = 0, and last F(n-1) = 0, and F(n) initialized to zero.
4. Loop from 2 to n.
    - Set current F(n) to current F(n-1) + F(n-2).
    - Set F(n-2) to current F(n-1).
    - Set F(n-1) to current F(n).

Python

```
def FibonacciIterative(n):
        # F(0) and F(1) case
        if n==0:
                return 0
        if n==1:
                return 1
        fn        = 0   # current value for F(n)
        f_minus_1 = 1   # current value for F(n-1)
        f_minus_2 = 0   # current value for F(n-2)
        for i in range( 2, n+1 ):
                fn        = f_minus_1 + f_minus_2   # F(n) = F(n-1) + F(n-2)
                f_minus_2 = f_minus_1               # next F(n-2)
                f_minus_1 = fn                      # next F(n-1)
        return fn

# Fibonacci Sequence to F(10)
for number in range( 0, 10 ):
        print( FibonacciIterative(number) )
```

Java

```
public class fibonacci {
        public static void main( String args[] ) {
                System.out.println( "Fibonacci Sequence to F(10)");

                for ( int number = 0; number < 10; number++ )
                        System.out.println( Fibonacci( number ) );
        }
```

```
        // Iterative Solution
        public static int Fibonacci( int n ) {
                if ( n == 0 ) return 0;
                if ( n == 1 ) return 1;

                int fn        = 0;      // current value for F(n)
                int f_minus_2 = 0;      // current value of F(n-2)
                int f_minus_1 = 1;      // current value of F(n-1)
                for ( int i = 2; i <= n; i++ ) {
                        fn        = f_minus_1 + f_minus_2;  // F(n) = F(n-1) + F(n-2)
                        f_minus_2 = f_minus_1;              // next F(n-2)
                        f_minus_1 = fn;                     // next F(n-1)
                }
                return fn;
        }
}
```

3. Dynamic Arrays

A dynamic array is an array that can be resized, can be randomly accessed via an cardinal ordering, where there is a one-to-one relationship between an integer index and a value. For example, if a dynamic array has the ordered elements A, B and D, they would be accessed (one-to-one relationship) by indices 0=>A, 1=>B and 2=>D. If we insert a C between B and D, the ordering would be rearranged but there still would be a one-to-one relationship, accessed as indices 0=>A, 1=>B, 2=>C and 3=>D. Note how index 2 now relates to C and index 3 to D.

You maybe asked to implement a dynamic array using a class. You would be expected to implement at least an Add() and Get() methods.

Algorithm

1. Define a class for an element.
    - Has a member to hold the value.
    - Has a next pointer for the next element in the array.
2. Define a class for an array.
    - Has a member to hold the head and a member to hold the tail of the array.
    - A member to hold the size of the array.
    - Define an Add method.
        - Instantiate a new element.
        - Add the element to the end of tail (the tail's next pointer) and update the tail to the new element.
        - If it is the first element in the array, set the head to the new element.
        - Increment the size of the array by one.
    - Define a Get method.

- Check for array out of bounds condition.
- Starting with the head, advance consecutively index number of times following the next pointer.
- Return the value of the current element.

Python

```
# Definition for an element in the dynamic array
class Element:
        next = None     # next element in array

        # Constructor: set the node data
        def __init__(self, value):
                self.value   = value    # node data

# Definition for a Dynamic Array
class DynamicArray:
        head = None # head of the array
        tail = None # tail of the array
        size = 0    # size of the array

        # The size (number of elements) of the array
        def Size( self ):
                return self.size;

        # Add an element to the end of the dynamic array
        def Add( self, value ):
                e = Element(value)

                # Set the next element of the previous tail to the new element
                if self.tail != None:
                        self.tail.next = e

                # set the current tail to the new element
                self.tail = e

                # If there is no head (first element), set the head to the new element.
                if self.head == None:
                        self.head = e

                # Increment the size of the array
                self.size += 1
```

```
        # Get the value at the corresponding index
        def Get( self, index ):
                # Check for out of bounds condition
                if index > self.size - 1 or index < 0:
                        return None

                # Step (linear) to the index position
                curr = self.head;
                for i in range( 0, index ):
                        curr = curr.next

                return curr.value
```

Java

```
// Definition for an element in the dynamic array
class Element {
        Object  value = null;   // Element value
        Element next  = null;   // Next element in the array

        // Constructor
        public Element( Object value ) {
                this.value = value;
        }
}

// Definition for a Dynamic Array
class DynamicArray {
        private Element head = null;    // Head of the Array
        private Element tail = null;    // Tail of the Array
        private int     size = 0;

        // The size (number of elements) of the array
        public int Size() {
                return size;
        }

        // Add an element to the end of the dynamic array
        public void Add( Object value ) {
                Element e = new Element( value );
```

```
                // Set the next element of the previous tail to the new element
                if ( tail != null )
                        tail.next = e;

                // set the current tail to the new element
                tail = e;

                // If there is no head (first element), set the head to the new element.
                if ( head == null )
                        head = e;

                // Increment the size of the array
                size++;
        }

        // Get the value at the corresponding index
        public Object Get( int index ) {
                // Check for out of bounds condition
                if ( index > size - 1 || index < 0 )
                        return null;

                // Step (linear) to the index position
                Element curr = head;
                for ( int i = 0; i < index; i++ )
                        curr = curr.next;

                return curr.value;
        }
}
```

Once you've solved the Add() and Get() methods, you maybe asked to code the methods for Delete() and Insert(). In the below coding example, I solve for an insert after. You maybe additionally asked to code an insert before, or both.

Algorithm

1. Define a Delete() method.
    - Check for array out of bounds condition.
    - If deleting the head (index = 0), then set the new head to what the head currently points to.
    - Starting with the head, advance consecutively index number of times following the next pointer, and remember the element before it.
    - Set the next pointer of the element before this element to this element's next pointer (thus dropping this element which is in between).

- Decrement the size of the array.
- If the element deleted was the tail (index equals the new array size), set the tail to the previous element.

2. Define an Insert() after method.
   - Check for array out of bounds condition.
   - If inserting after the tail (index == size-1), then insert the element using the Add() method.
   - Advance to the element at the specified index, and remember the element before it.
   - Instantiate a new element.
   - Set the next pointer of the new element to the element following the current element.
   - Update the next pointer of the current element to the new element (thus inserting in between).
   - Increment the size of the array by one.

Python

```
class DynamicArray:
        ...
        def Delete( self, index ):
                # Check for out of bounds condition
                if index > self.size - 1 or index < 0:
                        return False;

                # Remove the head of the dynamic array
                if index == 0:
                        # Set the head to the next head
                        self.head = self.head.next;

                        # If the array is now empty, set the tail to none
                        self.size -= 1
                        if self.size == 0:
                                self.tail = None

                        return True

                # Find the element at the specified index, and remember the element before it.
                curr = self.head
                prev = None
                for i in range( 0, index ):
                        prev = curr
                        curr = curr.next

                # Set the next pointer of the previous element to the next pointer of the current element,
                # thus dropping this element which is between them.
```

```
                prev.next = curr.next

                # We deleted the tail. Set the current tail to the previous element.
                self.size -= 1
                if self.size == index:
                        self.tail = prev

                return True

        def Insert( self, index, value ):
                # Check for out of bounds condition
                if index > self.size - 1 or index < 0:
                        return False

                # Adding to the tail
                if index == self.size-1:
                        self.Add( value )
                        return True

                # Find the element at the index
                curr = self.head
                for i in range( 0, index ):
                        curr = curr.next

                # Insert (after) between the current element and the next
                e = Element( value )
                e.next = curr.next
                curr.next = e

                self.size += 1
                return True
```

Java

```
// Definition for a Dynamic Array
class DynamicArray {
        ...

        public boolean Delete( int index ) {
                // Check for out of bounds condition
                if ( index > size - 1 || index < 0 )
```

```
                return false;

        // Remove the head of the dynamic array
        if ( index == 0 ) {
                // Set the head to the next head
                head = head.next;

                // If the array is now empty, set the tail to null
                if ( --size == 0 )
                        tail = null;

                return true;
        }

        // Find the element at the specified index, and remember the element before it.
        Element curr = head, prev = null;
        for ( int i = 0; i < index; i++ ) {
                prev = curr;
                curr = curr.next;
        }

        // Set the next pointer of the previous element to the next pointer of the current element,
        // thus dropping this element which is between them.
        prev.next = curr.next;

        // We deleted the tail. Set the current tail to the previous element.
        if ( --size == index )
                tail = prev;

        return true;
}

public boolean Insert( int index, Object value ) {
        // Check for out of bounds condition
        if ( index > size - 1 || index < 0 )
                return false;

        // Adding to the tail
        if ( index == size-1 ) {
                Add( value );
```

```
                        return true;
                }

                // Find the element at the index
                Element curr = head;
                for ( int i = 0; i < index; i++ )
                        curr = curr.next;

                // Insert (after) between the current element and the next
                Element e = new Element( value );
                e.next = curr.next;
                curr.next = e;

                size++;
                return true;
        }
        ....
}
```

4. Linked Lists - Queue

A linked list is a data structure where each element is linked to another element forming a chain, such as A links to B which links to C, and so forth. Depending on the type of chaining, one can use link lists to code queues and stacks. A queue is also known as a FIFO (first-in-first-out) structure.

A queue consists of a head and a tail, where the head points to the first element in the queue, and the tail points to the end of the queue. Each new element is added to the end of the tail, becoming the new tail. Throughout the chain, each element is linked to the next element starting from the head and proceeding to the tail (forward chaining). A queue is typically used to process elements in the same sequential order they were added (FIFO). To process an element, one removes the head (current element to process) of the queue and sets the new head to the next element following the previous head.

Algorithm - Queue

1. Define a Task object.
    - Define members for task to perform and next pointer for next element in the forward chain.
    - Define a constructor to instantiate a Task.
    - Define Next() methods to get and set the next element in the forward chain.
    - Define an Action() method to process the element's task.
2. Define a Queue object.
    - Define members for the head and tail of the queue.
    - Define an Empty() method to test if the queue is empty.
    - Define an Add() method to add a task to the tail of the queue.
    - Define a Pop() method to process the task at the head of the queue and then set the head of the queue to the next element.

Python

```
# Definition for a Task element in a Queue
class Task:
        next = None

        # constructor, task represents the task to be performed
        def __init__( self, task ):
                self.task = task

        # Get the next task this element is chained to
        def GetNext(self):
                return self.next

        # Set the next task this element is chained to
        def Next( self, task ):
                self.next = task

        # Action to take when task is processed
        def Action( self ):
                print( self.task )

# Definition for a Queue
class Queue:
        head = None
        tail = None

        # Check if Queue is empty
        def Empty(self):
                if self.head == None:
                        return True
                return False

        # Add a task to the queue
        def Add( self, task ):
                # the queue is empty, set head and tail to the task
                if self.Empty():
                        self.head = self.tail = task
                # otherwise, add it as the new tail
                else:
                        self.tail.Next( task )
```

```
                        self.tail = task

        # Remove the top of the queue and process the task
        def Pop( self):
                # Queue is empty
                if self.Empty():
                        return

                # Process the task here
                self.head.Action()

                # Move the head to the next element.
                self.head = self.head.GetNext()
                if self.Empty():
                        self.tail = None

print("Process in sequential order tasks: A, B and C" )
queue = Queue()
queue.Add( Task( "A" ) )
queue.Add( Task( "B" ) )
queue.Add( Task( "C" ) )
queue.Pop()
queue.Pop()
queue.Pop()
```

Java

```
// Definition for a Task element in a Queue
class Task {
        private Object task;        // the task to perform
        private Task   next = null; // the next task in the queue

        // constructor, task represents the task to be performed
        public Task( Object task ) {
                this.task = task;
        }

        // Get the next task this element is chained to
        public Task Next() {
                return next;
        }
```

```
        // Set the next task this element is chained to
        public void Next( Task task ) {
                next = task;
        }

        // Action to take when task is processed
        public void Action() {
                System.out.println( task );
        }
}

// Definition for a Queue
public class Queue {
        private Task head = null;       // the head of the queue
        private Task tail = null;       // the tail of the queue

        // Check if Queue is empty
        private boolean Empty() {
                if ( head == null )
                        return true;
                return false;
        }

        // Add a task to the queue
        public void Add( Task task ) {
                // the queue is empty, set head and tail to the task
                if ( Empty() ) {
                        head = tail = task;
                }
                // otherwise, add it as the new tail
                else {
                        tail.Next( task );
                        tail = task;
                }
        }

        // Remove the top of the queue and process the task
        public void Pop() {
                // Queue is empty
```

```
                if ( Empty() )
                        return;

                // Process the task here
                head.Action();

                // Move the head to the next element.
                head = head.Next();
                if ( Empty() )
                        tail = null;
        }

        public static void main(String[] args) {
                System.out.println("Process in sequential order tasks: A, B and C" );
                Queue queue = new Queue();
                queue.Add( new Task( "A" ) );
                queue.Add( new Task( "B" ) );
                queue.Add( new Task( "C" ) );
                queue.Pop();
                queue.Pop();
                queue.Pop();
        }
}
```

You maybe asked to improve the coding example by being ask to add a task priority and sort the queue according to the task's priority.

Algorithm - Queue with Task Priority

1. Define a Task object.
    - Define members for task to perform, the task's priority and next pointer for next element in the forward chain.
    - Define a constructor to instantiate a Task.
    - Define Priority() methods to get and set the task's priority.
    - Define Next() methods to get and set the next element in the forward chain.
    - Define an Action() method to process the element's task.
2. Define a Queue object.
    - Define members for the head of the queue (a tail is not needed anymore!).
    - Define an Empty() method to test if the queue is empty.
    - Define an Add() method to add a task to the tail of the queue.
    - Define an Insert() method to add a task into the queue based on the task's priority.
    - Define a Pop() method to process the task at the head of the queue and then set the head of the queue to the next element.

Python

```
# Definition for a Task element in a Queue
class Task:
        task = None   # the task to perform
        next = None   # the next element in the queue
        priority = 0  # the task's priority

        # constructor, task represents the task to be performed
        def __init__( self, task, priority ):
                self.task = task
                self.priority = priority

        # Get the task's priority
        def GetPriority( self ):
                return self.priority

        # Set the task's priority
        def Priority( self, priority ):
                self.priority = priority

        # Get the next task this element is chained to
        def GetNext( self ):
                return self.next

        # Set the next task this element is chained to
        def Next( self, task ):
                self.next = task

        # Action to take when task is processed
        def Action( self ):
                print( self.task )

# Definition for a Queue
class Queue:
        head = None
        tail = None

        # Check if Queue is empty
        def Empty( self ):
                if self.head == None:
```

```
                return True
        return False

# Add a task to the queue
def Add( self, task ):
        # the queue is empty, set head to the task
        if self.Empty():
                self.head = task
        # otherwise, insert the task into the queue according to the task's priority
        else:
                self.Insert( task )

# Insert a task into the queue based on priority
def Insert( self, task ):
        # Start at the head of the chain to search where to insert the task
        curr = self.head
        prev = self.head
        while curr != None:
                # Find a task whose's priority is less than the new task
                if task.GetPriority() > curr.GetPriority():
                        # Insert the task in front of the current task.
                        task.Next( curr )

                        # If inserting in front of the head of the queue,
                        # make the new task the head of the queue
                        if curr == self.head:
                                self.head = task
                        # Otherwise, set the previous element's next to the new task
                        else:
                                prev.Next( task )
                        break

                prev = curr;
                curr = curr.GetNext();

        # Add to the tail (end) of the queue
        if curr == None:
                prev.Next( task )

# Remove the top of the queue and process the task
```

```
        def Pop( self ):
                # Queue is empty
                if self.Empty():
                        return

                # Process the task here
                self.head.Action()

                # Move the head to the next element.
                self.head = self.head.GetNext()
                if self.Empty():
                        self.tail = None

print("Process in sequential order tasks: B, C and A" )
queue = Queue()
queue.Add( Task( "A", 1 ) )
queue.Add( Task( "B", 3 ) )
queue.Add( Task( "C", 2 ) )
queue.Pop()
queue.Pop()
queue.Pop()
```

Java

```
// Definition for a Task element in a Queue
class Task {
        private Object task;        // the task to perform
        private Task   next = null; // the next task in the queue
        private int    priority;    // the task's priority

        // constructor, task represents the task to be performed
        public Task( Object task, int priority ) {
                this.task = task;
                this.priority = priority;
        }

        // Get the task's priority
        public int Priority() {
                return priority;
        }
```

```
        // Set the task's priority
        public void Priority( int priority ) {
                this.priority = priority;
        }

        // Get the next task this element is chained to
        public Task Next() {
                return next;
        }

        // Set the next task this element is chained to
        public void Next( Task task ) {
                next = task;
        }

        // Action to take when task is processed
        public void Action() {
                System.out.println( task );
        }
}

// Definition for a Queue
public class Queue {
        private Task head = null;       // the head of the queue

        // Check if Queue is empty
        private boolean Empty() {
                if ( head == null )
                        return true;
                return false;
        }

        // Add a task to the queue
        public void Add( Task task ) {
                // the queue is empty, set head to the task
                if ( Empty() ) {
                        head = task;
                }
                // otherwise, insert the task into the queue according to the task's priority
                else {
```

```
                Insert( task );
        }
}

// Insert a task into the queue based on priority
private void Insert( Task task ) {
        // Start at the head of the chain to search where to insert the task
        Task curr = head, prev = head;
        while ( curr != null ) {
                // Find a task whose's priority is less than the new task
                if ( task.Priority() > curr.Priority() ) {
                        // Insert the task in front of the current task.
                        task.Next( curr );

                        // If inserting in front of the head of the queue,
                        // make the new task the head of the queue
                        if ( curr == head )
                                head = task;
                        // Otherwise, set the previous element's next to the new task
                        else {
                                prev.Next( task );
                        }
                        break;
                }

                prev = curr;
                curr = curr.Next();
        }

        // Add to the tail (end) of the queue
        if ( curr == null ) {
                prev.Next( task );
        }
}

// Remove the top of the queue and process the task
public void Pop() {
        // Queue is empty
        if ( Empty() )
                return;
```

```
                // Proces the task here
                head.Action();

                // Move the head to the next element.
                head = head.Next();
        }

        public static void main(String[] args) {
                System.out.println("Process in priority order tasks: B, C and A" );
                Queue queue = new Queue();
                queue.Add( new Task( "A", 1 ) );
                queue.Add( new Task( "B", 3 ) );
                queue.Add( new Task( "C", 2 ) );
                queue.Pop();
                queue.Pop();
                queue.Pop();
        }
}
```

You maybe further asked to make another improvement to dynamically change a priority of a task and resort the queue accordingly.

Algorithm - Queue with Dynamic Priority Update

1. Update the priority of the task.
2. Locate task in the queue and remove the task.
3. Add the task back in based on its updated priority.

Python

```
# Definition for a Queue
class Queue:
        ...

        # Update the priority of the task and re-sort the queue
        def Update( self, task, priority ):
                # Update the task's priority
                task.Priority( priority );

                # If the task is the head of the queue, remove it and update head to the next element
                if self.head == task:
                        self.head.Next( self.head.GetNext() )
```

```
                # Find the task in the queue
                else:
                        curr = self.head.GetNext()
                        prev = self.head;
                        while curr != None:
                                # Remove the task from the queue and update previous element's next to this task's next element
                                if task == curr:
                                        prev.Next( curr.GetNext() );
                                        break;

                # Add the task back to the queue
                self.Add( task )

print("Process in sequential order tasks: C, B and A" )
queue = Queue()
queue.Add( Task( "A", 1 ) )
queue.Add( Task( "B", 3 ) )
task =  Task( "C", 2 );
queue.Add( task );
queue.Update( task, 4 );
queue.Pop()
queue.Pop()
queue.Pop()
        ...
```

Java

```
// Definition for a Queue
public class Queue {
        ...

        // Update the priority of the task and re-sort the queue
        public void Update( Task task, int priority ) {
                // Update the task's priority
                task.Priority( priority );

                // If the task is the head of the queue, remove it and update head to the next element
                if ( head == task ) {
                        head.Next( head.Next() );
                }
                // Find the task in the queue
```

```
                else {
                        Task curr = head.Next(), prev = head;
                        while ( curr != null ) {
                                // Remove the task from the queue and update previous element's next to this task's next element
                                if ( task == curr ) {
                                        prev.Next( curr.Next() );
                                        break;
                                }
                        }
                }

                // Add the task back to the queue
                Add( task );
        }

        public static void main(String[] args) {
                System.out.println("Process in priority order tasks: C, B and A" );
                Queue queue = new Queue();
                queue.Add( new Task( "A", 1 ) );
                queue.Add( new Task( "B", 3 ) );
                Task task =  new Task( "C", 2 );
                queue.Add( task );
                queue.Update( task, 4 );
                queue.Pop();
                queue.Pop();
                queue.Pop();
        }
        ...
}
```

5. Linked Lists - Stacks

A linked list is a data structure where each element is linked to another element forming a chain, such as A links to B which links to C, and so forth. Depending on the type of chaining, one can use link lists to code queues and stacks. A stack is also known as a LIFO (last-in-first-out) structure.

A stack consists of a top, which points to the top of the stack. Each new element is added to the top of the stack, becoming the new top. Throughout the chain, each element is linked to the next element starting from the top and proceeding to the bottom (forward chaining). A stack is typically used to process elements in the reverse sequential order they were added (LIFO). To process an element, one removes the top (current element to process) of the stack and sets the new top to the next element following the previous top.

Algorithm - Stack

1. Define a Task object.
    - Define members for task to perform, and next pointer for next element in the forward chain.
    - Define a constructor to instantiate a Task.
    - Define Next() methods to get and set the next element in the forward chain.
    - Define an Action() method to process the element's task.
2. Define a Stack object.
    - Define member for the top of the stack.
    - Define an Empty() method to test if the stack is empty.
    - Define a Push() method to add a task to the top of the stack.
    - Define a Pop() method to process the task at the top of the stack and then set the top of the stack to the next element.

Python

```
# Definition for a Task element in a Stack
class Task:
        task = None  # the task to perform
        next = None  # the next task in the stack

        # constructor, task represents the task to be performed
        def __init__( self, task ):
                self.task = task

        # Get the next task this element is chained to
        def GetNext(self):
                return self.next

        # Set the next task this element is chained to
        def Next( self, task ):
                self.next = task;

        # Action to take when task is processed
        def Action( self ):
                print( self.task )

# Definition for a Stack
class Stack:
        top = None   # top of the stack

        # Check if the Stack is empty
        def Empty(self):
                if self.top == None:
```

```
                        return True
                return False

        # Push the task to the top of the stack
        def Push( self, task ):
                # Set the task next pointer to the current top
                task.Next( self.top )
                # Set the top to this task
                self.top = task

        # Pop the task from the top of the stack
        def Pop( self ):
                # Stack is Empty
                if self.Empty():
                        return

                # Perform the action for the task
                self.top.Action()

                # Move the top to the current top's next pointer.
                self.top = self.top.GetNext()

print("Process tasks in order: A, B and C" )
stack = Stack()
stack.Push( Task( "C" ) )
stack.Push( Task( "B" ) )
stack.Push( Task( "A" ) )
stack.Pop()
stack.Pop()
stack.Pop()
```

Java

```
// Definition for a Task element in a Stack
class Task {
        private Object task;        // the task to perform
        private Task   next = null; // the next task in the stack

        // constructor, task represents the task to be performed
        public Task( Object task ) {
                this.task = task;
```

```
        }

        // Get the next task this element is chained to
        public Task Next() {
                return next;
        }

        // Set the next task this element is chained to
        public void Next( Task task ) {
                next = task;
        }

        // Action to take when task is processed
        public void Action() {
                System.out.println( task );
        }
}

// Definition for a Stack
public class Stack {
        private Task top;       // top of the stack

        // Check if the Stack is empty
        private boolean Empty() {
                if ( top == null )
                        return true;
                return false;
        }

        // Push the task to the top of the stack
        public void Push( Task task ) {
                // Set the task next pointer to the current top
                task.Next( top );
                // set the top to this task
                top = task;
        }

        // Pop the task from the top of the stack
        public void Pop() {
                // Stack is Empty
```

```
                if ( Empty() )
                        return;

                // Perform the action for the task
                top.Action();

                // Move the top to the current top's next pointer.
                top = top.Next();
        }

        public static void main(String[] args) {
                System.out.println("Process tasks in order: A, B and C" );
                Stack stack = new Stack();
                stack.Push( new Task( "C" ) );
                stack.Push( new Task( "B" ) );
                stack.Push( new Task( "A" ) );
                stack.Pop();
                stack.Pop();
                stack.Pop();
        }
}
```

A common variation you maybe asked is to redesign your stack into a chain of stacks of fixed size. When an element is pushed to a stack that has reached its maximum (fixed) size, the bottom of the stack is removed and pushed to the top of the next stack on the chain and so forth. Likewise, when an element is popped from the stack, the top of the next stack that is chained to it is popped and added to the bottom of the stack, and so forth.

Unlike the previous stack example, where we implicitly performed the task when it was popped, we will this time return the task on a pop and let the application explicitly invoke performing the task. This will simplify the push and pop operations in a chain.

Algorithm - Chain of Stacks

1. Define a Task object.
    - Define members for task to perform, and next and previous pointer for next and previous element in the forward and backward chain.
    - Define a constructor to instantiate a Task.
    - Define Next() and Prev() methods to get and set the next and previous elements in the forward and backward chain.
    - Define an Action() method to process the element's task.
2. Define a Stack Chain object.
    - Define members for the top, bottom, maximum size and current size of the stack.
    - Define a member for the next stack in the chain.
    - Define an Empty() method to test if the stack is empty.
    - Define a Push() method to add a task to the top of the stack.

        - If exceeds the maximum size of the stack, remove the bottom task of the stack.
        - Push the task removed from the bottom to the next stack in the chain.
    - Define a Pop() method to return the top of the stack and then set the top of the stack to the next element.
        - If the next stack in the chain is non-empty, then pop the top of the next stack in the chain.
        - Push the task removed from the top of the next stack in the chain onto the bottom of this stack.

Python

```
# Definition for a Task element in a Stack
class Task:
        ...
        prev = None  # the previous task in the stack

        # Get the previous task this element is chained to
        def GetPrev( self ):
                return self.prev;

        # Set the previous task this element is chained to
        def Prev( self, task ):
                self.prev = task;

# Definition for a Stack Chain
class StackChain:
        top    = None # top of the
        bottom = None # bottom of the stack
        max    = 0    # maximum size of the stack
        size   = 0    # size of the stack
        chain  = None # next stack in the chain

        # Constructor: instantiate a stack in a chain, specifying the maximum size of the stack.
        def __init__( self, max ):
                self.max = max

        # Check if the Stack is empty
        def Empty( self ):
                if self.top == None:
                        return True
                return False

        # Push the task to the top of the stack
        def Push( self, task ):
```

```
                # Set the task next pointer to the current top
                task.Next( self.top )

                # Chain the current top back to the task
                if self.top != None:
                        self.top.Prev( task )

                # Set the top to this task
                self.top = task

                # only element on the stack, set the bottom to the top
                self.size += 1
                if self.size == 1:
                        self.bottom = self.top

                # overflowed the maximum size of the stack
                if self.size > self.max:
                        # Create a new stack in the chain of the same maximum size
                        if self.chain == None:
                                self.chain = StackChain( self.max )

                        # Push the bottom of this stack onto the top of the next stack in the chain
                        self.chain.Push( self.bottom )

                        # Remove the bottom by setting the bottom to the previous element from the bottom
                        self.bottom = self.bottom.GetPrev()

        # Pop the task from the top of the stack
        def Pop( self ):
                # Stack is Empty
                if self.Empty():
                        return

                # The task that is being popped from the stack
                task = self.top

                # Stack is empty
                self.size -= 1
                if self.size == 0:
                        self.top = self.bottom = None
```

```
                else:
                        # Move the top to the current top's next pointer.
                        self.top = self.top.GetNext()

                        # There is a another stack chained to this stack
                        if self.chain != None:
                                # Pop the top of the stack chained to this stack
                                move = self.chain.Pop()

                                # Add this task to the bottom of the stack
                                self.bottom.Next( move )
                                # Set the task's next to null
                                task.Next( None )
                                # Set the task as the new bottom of the stack
                                self.bottom = move

                                # If the stack chained to this one is empty, set the chain pointer to null
                                if self.chain.Empty():
                                        self.chain = None

                # return the task that was popped
                return task

print("Process tasks in order: 6, 5, 4, 3, 2, 1, 0" )
stack = StackChain( 3 );
for i in range( 0, 7 ):
        stack.Push( Task( i ) )
for i in range( 0, 7 ):
        stack.Pop().Action()
```

Java

```
// Definition for a Task element in a Stack
class Task {
        ...
        private Task   prev = null; // the previous task in the stack

        // Get the previous task this element is chained to
        public Task Prev() {
                return prev;
        }
```

```
        // Set the previous task this element is chained to
        public void Prev( Task task ) {
                prev = task;
        }
        ...
}

// Definition for a Chain of Stacks
public class StackChain {
        private Task top    = null;      // top of the stack
        private Task bottom = null;      // bottom of the stack
        private final int max;           // maximum size of the stack
        private int size;                // current size of the stack
        private StackChain chain = null; // next stack in the chain

        // Constructor: instantiate a stack in a chain, specifying the maximum size of the stack.
        public StackChain( int max ) {
                this.max = max;
        }

        // Check if the Stack is empty
        private boolean Empty() {
                if ( top != null )
                        return false;
                return true;
        }

        // Push the task to the top of the stack
        public void Push( Task task ) {
                // Set the task next pointer to the current top
                task.Next( top );

                // Chain the current top back to the task
                if ( top != null )
                        top.Prev( task );

                // Set the top to this task
                top = task;
```

```
                // only element on the stack, set the bottom to the top
                if ( ++size == 1 )
                        bottom = top;

                // overflowed the maximum size of the stack
                if ( size > max ) {
                        // Create a new stack in the chain of the same maximum size
                        if ( chain == null )
                                chain = new StackChain( max );

                        // Push the bottom of this stack onto the top of the next stack in the chain
                        chain.Push( bottom );

                        // Remove the bottom by setting the bottom to the previous element from the bottom
                        bottom = bottom.Prev();
                }
        }

        // Pop the task from the top of the stack
        public Task Pop() {
                // Stack is Empty
                if ( Empty() )
                        return null;

                // The task that is being popped from the stack
                Task task = top;

                // Stack is empty
                if ( --size == 0 )
                        top = bottom = null;
                else {
                        // Move the top to the current top's next pointer.
                        top = top.Next();

                        // There is a another stack chained to this stack
                        if ( chain != null ) {
                                // Pop the top of the stack chained to this stack
                                Task move = chain.Pop();

                                // Add this task to the bottom of the stack
```

```
                                bottom.Next( move );
                                // Set the task's next to null
                                task.Next( null );
                                // Set the task as the new bottom of the stack
                                bottom = move;

                                // If the stack chained to this one is empty, set the chain pointer to null
                                if ( chain.Empty() )
                                        chain = null;
                        }
                }

                // return the task that was popped
                return task;
        }

        public static void main(String[] args) {
                System.out.println("Process tasks in order: 6, 5, 4, 3, 2, 1, 0" );
                StackChain stack = new StackChain( 3 );
                for ( int i = 0; i < 7; i++ )
                        stack.Push( new Task( i ) );
                for ( int i = 0; i < 7; i++ )
                        stack.Pop().Action();
        }
}
```

6. Binary Tree

A binary tree is a tree where each node contains at most two branches (subtrees), commonly known as the left tree and right tree. The recursive definition is a binary tree is either empty or a single node, where the left and right branches are binary subtrees.

```
binary tree:    empty node |
                node
                    left binary subtree
                    right binary subtree
```

Some other terminology that is useful to know:

- The topmost node is the root.
- Nodes at the same level are siblings.
- Nodes with no branches are leaf nodes (or leaves).
- Every node with branches is a parent. The nodes of the parent are children.

-
- A branch is a directed edge.
- A node depth is the number of edges between the root and the node.
- A node height is the number of edges between the node and the deepest leaf node.

Algorithm

You maybe asked to code a class definition for a binary tree node. The binary tree node will have the following members/methods:

1. Constructor for instantiating the node and initializing the node data. In my examples, I use the data type 'Object' (Java) to indicate that the node is not limited to the type of data it can hold, which is determined at run-time.
2. Private member fields (only accessible by the class object) for the data value, left and right subtrees.
3. Public Set and Get accessors for the data value, left and right subtrees, which can be used by a program accessing the class object.

Python

```
class BinaryTree:
        # Constructor: set the node data and left/right subtrees to null
        def __init__(self, key):
                self.__left  = None     # left binary subtree
                self.__right = None     # right binary subtree
                self.__key   = key      # node data

        # Set Left Binary Subtree
        def Left(self, left):
                self.__left = left

        # Get Left Binary Subtree
        def GetLeft(self ):
                return __left

        # Set Right Binary Subtree
        def Right(self, right):
                self.__right = right

        # Get Right Binary Subtree
        def GetRight(self):
                return __right

        # Set Node Data
        def Key(self, key):
```

```
                self.__key = key

        # Get Node Data
        def GetKey(self):
                return __key
```

Java

```
// Binary Tree Class
class BinaryTree {
    private Object key;         // node data
    private BinaryTree left;    // left binary subtree
    private BinaryTree right;   // right binary subtree

        // Constructor: set the node data and left/right subtrees to null
        public BinaryTree (Object key) {
                this.key = key;
                right = null;
                left = null;
        }

        // Set the left binary subtree
        public void Left( BinaryTree left ) {
                this.left = left;
        }

        // Get the left binary subtree
        public BinaryTree Left() {
                return left;
        }

        // Set the right binary subtree
        public void Right( BinaryTree right ) {
                this.right = right;
        }

        // Get the right binary subtree
        public BinaryTree Right() {
                return right;
        }
```

```
        // Set the node data
        public void Key( Object key ) {
                this.key = key;
        }

        // Get the node data
        public Object Key() {
                return key;
        }
}
```

You maybe asked for a more generalized representation, such as defining a hierarchical class structure where a binary tree is a class type of a k-ary tree. For this, we would define a base (super) class with methods common to k-ary trees, and then derive (extend) a binary tree class from the base (super) class with methods specific to a binary tree.

Algorithm

1. The base class defines the key member and corresponding access methods.
2. The base class is made abstract, so that one has to derive (extend) a class to use it.
3. An abstract method (must be defined in derived class) is added to defining an action to perform on each node in the tree that is specific to a derived class.

Python

```
# Base (Super) class definition for a k-ary tree
class Node:
        # Constructor: set the node data
        def __init__(self, key):
                self.__key   = key      # node data

        # Set Node Data
        def Key(self, key):
                self.__key = key

        # Get Node Data
        def GetKey(self):
                return __key

        # Action to perform on a node
        def Action(self, node):
                raise NotImplementedError("Please Implement this method")

# Derived definition for a Binary Tree
```

```
class BinaryTree(Node):
        # Constructor: set the node data and left/right subtrees to null
        def __init__(self, key):
                Node.__init__(self, key)
                self.__left  = None     # left binary subtree
                self.__right = None     # right binary subtree

        # Set Left Binary Subtree
        def Left(self, left):
                self.__left = left

        # Get Left Binary Subtree
        def GetLeft(self):
                return __left

        # Set Right Binary Subtree
        def Right(self, right):
                self.__right = right

        # Get Right Binary Subtree
        def GetRight(self):
                return __right

        # Example action to perform on a node
        def Action(self,node):
                print( node.key)
```

Java

```
// Base (Super) class definition for a k-ary tree
abstract class Node {
        // node data
        private Object key;

        // action to perform on a node
        public abstract void Action();

        // constructor: set the node data
        public Node( Object key ) {
                this.key = key;
        }
```

```
        // Set the node data
        public void Key( Object key ) {
                this.key = key;
        }

        // Get the node data
        public Object Key() {
                return key;
        }
}

// Derived definition for a Binary Tree
class BinaryTree extends Node {
    private BinaryTree left;    // left binary subtree
    private BinaryTree right;   // right binary subtree

        // Constructor: set the node data and left/right subtrees to null
        public BinaryTree (Object key) {
                super( key );
                right = null;
                left = null;
        }

        // Set the left binary subtree
        public void Left( BinaryTree left ) {
                this.left = left;
        }

        // Get the left binary subtree
        public BinaryTree Left() {
                return left;
        }

        // Set the right binary subtree
        public void Right( BinaryTree right ) {
                this.right = right;
        }

        // Get the right binary subtree
```

```
        public BinaryTree Right() {
                return right;
        }

        // Example action to perform on a node
        public void Action() {
                System.out.println( this.Key() );
        }
}
```

7. Binary Tree Traversals

Binary trees can be traversed either breadth first or depth first. In a breadth first traversal, the tree is traversed one level at a time. The root node (level 1) is first visited, then the left and right node (level 2) of the root, and then the left and right nodes of these subtrees (level 3), and so forth.

For depth first traversal or search (DFS), a tree can be traversed either inorder, preorder or postorder.

- Inorder: left (node), root, right
- Preorder: root, left, right
- Postorder: left, right, root

If you are ask questions about a binary search tree (BST), most likely the algorithm will use a postorder traversal.

Algorithm - Inorder

1. From the current node (root), traverse to left node. Recursively apply the algorithm to the left node.
2. Traverse back to the root node.
3. From the current node (root), traverse to right node. Recursively apply the algorithm to the right node.

Python

```
        #InOrder Traversal
        def InOrder(root):
                if root == None:
                        return
                InOrder( root.left )
                Action( root )
                InOrder( root.right )
```

Java

```
        // InOrder Traversal
        public void InOrder() {
```

```
                if ( Left() != null ) Left().InOrder();
                Action();
                if ( Right() != null ) Right().InOrder();
        }
```

Algorithm - Preorder

1. Traverse to the root node.
2. From the current node (root), traverse to left node. Recursively apply the algorithm to the left node.
3. From the current node (root), traverse to right node. Recursively apply the algorithm to the right node.

Python

```
        # PreOrder Traversal
        def PreOrder(root):
                if root == None:
                        return
                Action( root )
                PreOrder( root.left )
                PreOrder( root.right )
```

Java

```
        // PreOrder Traversal
        public void PreOrder() {
                Action();
                if ( Left() != null ) Left().PreOrder();
                if ( Right() != null ) Right().PreOrder();
        }
```

Algorithm - Postorder

1. From the current node (root), traverse to left node. Recursively apply the algorithm to the left node.
2. From the current node (root), traverse to right node. Recursively apply the algorithm to the right node.
3. Traverse back to the root node.

Python

```
        # PostOrder Traversal
        def PostOrder(root):
                if root == None:
                        return
                PostOrder( root.left )
```

```
                PostOrder( root.right )
                Action( root )
```

Java

```
        // PostOrder Traversal
        public void PostOrder() {
                if ( Left() != null ) Left().PostOrder();
                if ( Right() != null ) Right().PostOrder();
                Action();
        }
```

A breadth first traversal or search (BFS) traverses a tree one node level at a time, visiting each node at that level, generally from left to right. This traversal is referred to as a level order traversal.

Algorithm - Level Order

1. If the root node is non-null, add the root to the nodes to visit in sequential order.
2. While there are nodes to visit, do:
    - Visit the next node.
    - Add the left node and right node of the visited node, if they exist to the nodes to visit.

Python

```
# Breadth First Search
def BFS( root ):
        # Check if tree is empty
        if root == None:
                return

        # list of nodes to visit in node level order
        visit = DynamicArray()
        visit.Add( root );

        # sequentially visit in node in level order as it is dynamically added to the list
        i = 0
        while i < visit.Size():
                # Perform the node action
                visit.Get( i ).Action()

                # Add to the list the child siblings of this node
                if visit.Get( i ).GetLeft() != None:
```

```
                        visit.Add( visit.Get( i ).GetLeft() )
                if visit.Get( i ).GetRight() != None:
                        visit.Add( visit.Get( i ).GetRight() )
                i += 1

tree = BinaryTree( "1" )
tree.Left ( BinaryTree( "2" ) )
tree.Right( BinaryTree( "3" ) )
tree.GetLeft().Left( BinaryTree( "4" ) )
tree.GetLeft().Right( BinaryTree( "5" ) )
print( "BFS output 1 2 3 4 5")
BFS( tree )
```

Java

```
public class BFS {
        public static void BFS( BinaryTree root ) {

                // Check if tree is empty
                if ( root == null )
                        return;

                // list of nodes to visit in node level order
                LinkedList visit = new LinkedList();
                visit.add( root );

                // sequentially visit each node in level order as it is dynamically added to the list
                for ( int i = 0; i < visit.size(); i++ ) {
                        // Perform the node action
                        visit.get( i ).Action();

                        // Add to the list the child siblings of this node
                        if ( visit.get( i ).Left() != null )
                                visit.add( visit.get( i ).Left() );
                        if ( visit.get( i ).Right() != null )
                                visit.add( visit.get( i ).Right() );
                }
        }

        public static void main( String[] args ) {
                BinaryTree tree = new BinaryTree( "1" );
```

```
                tree.Left ( new BinaryTree( "2" ) );
                tree.Right( new BinaryTree( "3" ) );
                tree.Left().Left( new BinaryTree( "4" ) );
                tree.Left().Right( new BinaryTree( "5" ) );
                System.out.println( "BFS output 1 2 3 4 5");
                BFS( tree );
        }
}
```

If you are asked to count the size (number of nodes) of a binary tree, using any of the traversal methods, then replace the call to the method Action() with a counter that is increment each time a node is visited.

You maybe asked to add to your solution a method to calculate the maximum and minimum depth of the tree. The maximum depth is the number of nodes from the root to the leaf node that is the furthest from the root, while the minimum depth is the number of nodes from the root to the leaf node that is the closest the root.

Algorithm - Maximum and Minimum Depth of Binary Tree

1. Do a preorder traversal of the tree starting from the root.
2. When a node is visited, increment the depth count by 1.
3. Recursively apply the algorithm to the left and right nodes.
4. When returning from a child back to a parent node, pass back the node depth of the child.
5. Return the maximum or minimum value from the left and right child (or the parent if there are no children).

Python

```
# Derived definition for a Binary Tree
class BinaryTree(Node):
        ...

        # Calculate the maximum depth of the binary tree
        def MaxDepth( self, max ):
                max += 1    # increment by one level for the node

                lmax = max
                rmax = max # maximum depth on left and right nodes.

                # Calculate the maximum depth along the left binary subtree
                if self.GetLeft() != None:
                        lmax = self.GetLeft().MaxDepth( max )
                # Calculate the maximum depth along the right binary subtree
                if self.GetRight() != None:
                        rmax = self.GetRight().MaxDepth( max )
```

```
                if lmax > rmax:
                        return lmax
                else:
                        return rmax

        # Calculate the minimum depth of the binary tree
        def MinDepth( self, min ):
                min += 1    # increment by one level for the node

                lmin = min
                rmin = min  # minimum depth on left and right nodes.

                # Calculate the maximum depth along the left binary subtree
                if self.GetLeft() != None:
                        lmin = self.GetLeft().MaxDepth( min )
                # Calculate the maximum depth along the right binary subtree
                if self.GetRight() != None:
                        rmin = self.GetRight().MaxDepth( min )

                if lmin < rmin:
                        return lmin
                else:
                        return rmin

tree = BinaryTree( "1" )
tree.Left ( BinaryTree( "2" ) )
tree.Right( BinaryTree( "3" ) )
tree.GetLeft().Left( BinaryTree( "4" ) )
tree.GetLeft().Right( BinaryTree( "5" ) )
print("Max Depth(3) is  " + str( tree.MaxDepth( 0 ) ) )
print("Max Depth(2) is  " + str( tree.MinDepth( 0 ) ) )
```

Java

```
// Derived definition for a Binary Tree
class BinaryTree extends Node {
        ...
        // Calculate the maximum depth of the binary tree
        public int MaxDepth( int max )
        {
```

```
                max++;  // increment by one level for the node

                int lmax = max, rmax = max;     // maximum depth on left and right nodes.

                // Calculate the maximum depth along the left binary subtree
                if ( Left() != null )
                        lmax = Left().MaxDepth( max );
                // Calculate the maximum depth along the right binary subtree
                if ( Right() != null )
                        rmax = Right().MaxDepth( max );

                return Math.max( lmax, rmax );
        }

        // Calculate the minimum depth of the binary tree
        public int MinDepth( int min )
        {
                min++;  // increment by one level for the node

                int lmin = min, rmin = min;     // minimum depth on left and right nodes.

                // Calculate the maximum depth along the left binary subtree
                if ( Left() != null )
                        lmin = Left().MaxDepth( min );
                // Calculate the maximum depth along the right binary subtree
                if ( Right() != null )
                        rmin = Right().MaxDepth( min );

                return Math.min( lmin, rmin );
        }

        // Driver method
    public static void main(String[] args)
    {
        BinaryTree tree = new BinaryTree( 1 );
        tree.Left(  new BinaryTree( 2 ) );
        tree.Right( new BinaryTree( 3 ) );
        tree.Left().Left( new BinaryTree( 4 ) );
        tree.Left().Right( new BinaryTree( 5 ) );
        System.out.println("\nMax Depth(3) is  " + tree.MaxDepth( 0 ) );
```

```
        System.out.println("\nMax Depth(2) is  " + tree.MinDepth( 0 ) );
    }
}
```

Another addition you maybe asked is to add to your solution a method to calculate the maximum and minimum value of the tree. The maximum value is the node with the maximum value of all nodes, while the minimum value is the node with the minimum value of all nodes.

Algorithm - Maximum and Minimum Value

1. Do a level order (Breadth First) traversal of the tree.
2. Keep a counter of the maximum and minimum value encountered.

Python

```
# Derived definition for a Binary Tree
class BinaryTree(Node):
        ...
        # Calculate the minimum and maximum value in the binary tree
        def MinMax( self ):
                min = 2147483648  # counter for min value (start at maximum signed int)
                max = -2147483647 # counter for max value (start at minimum signed int)

                # list of nodes to visit in node level order
                visit = DynamicArray()
                visit.Add( self )

                # sequentially visit each node in level order as it is dynamically added to the list
                i = 0
                while i < visit.Size():
                        # Perform the node action
                        if int( visit.Get( i ).GetKey() ) > int( max ):
                                max = visit.Get( i ).GetKey()
                        if int( visit.Get( i ).GetKey() ) < int( min ):
                                min = visit.Get( i ).GetKey()

                        # Add to the list the child siblings of this node
                        if visit.Get( i ).GetLeft() != None:
                                visit.Add( visit.Get( i ).GetLeft() )
                        if visit.Get( i ).GetRight() != None:
                                visit.Add( visit.Get( i ).GetRight() )
                        i += 1
```

```
                return min, max

tree = BinaryTree( "1" )
tree.Left ( BinaryTree( "2" ) )
tree.Right( BinaryTree( "3" ) )
tree.GetLeft().Left( BinaryTree( "4" ) )
tree.GetLeft().Right( BinaryTree( "5" ) )
min,max = tree.MinMax()
print("Min Value(1) is  " + str( min ) )
print("Max Value(5) is  " + str( max ) )
```

Java

```
// Derived definition for a Binary Tree
class BinaryTree extends Node {
        ...

        // Calculate the minimum and maximum value in the binary tree
        public int[] MinMax() {
                int min = 0x7FFFFFFF;  // counter for min value (start at maximum signed int)
                int max = 0x80000000;  // counter for max value (start at minimum signed int)

                // list of nodes to visit in node level order
                LinkedList visit = new LinkedList();
                visit.add( this );

                // sequentially visit each node in level order as it is dynamically added to the list
                for ( int i = 0; i < visit.size(); i++ ) {
                        // Perform the node action
                        if ( (int)  visit.get( i ).Key() > max )
                                max = (int)  visit.get( i ).Key();
                        if ( (int)  visit.get( i ).Key() < min )
                                min = (int)  visit.get( i ).Key();

                        // Add to the list the child siblings of this node
                        if ( visit.get( i ).Left() != null )
                                visit.add( visit.get( i ).Left() );
                        if ( visit.get( i ).Right() != null )
                                visit.add( visit.get( i ).Right() );
                }
```

```
                int[] ret = new int[2];
                ret[0] = min; ret[1] = max;
                return ret;
        }

        // Driver method
    public static void main(String[] args)
    {
        BinaryTree tree = new BinaryTree( 1 );
        tree.Left(  new BinaryTree( 2 ) );
        tree.Right( new BinaryTree( 3 ) );
        tree.Left().Left( new BinaryTree( 4 ) );
        tree.Left().Right( new BinaryTree( 5 ) );

        int[] minmax = tree.MinMax();
            System.out.println("\nMin Value(1) is  " + minmax[ 0 ] );
            System.out.println("\nMax Value(5) is  " + minmax[ 1 ] );
    }
        ...
}
```

8. k-ary Trees

As a follow up to coding examples related to a binary tree, you maybe asked to generalize a solution for a k-ary tree. A k-ary tree is a tree that may at most k children. You may be challenged, by being asked to create a base (super) class from which any k-ary tree can derived (extended) from, with the condition that the children (branch) members must be defined in the base (super) class.

Since we do not know in the base (super) class how many children (branches) each node will have, we will use a dynamic array to represent the member for children. In the Java coding example, I will use the predefined dynamic array type ArrayList.

Algorithm

1. The base class defines the key member and corresponding access methods.
2. The base class defines a member for the maximum number of children and a dynamic array for pointers to the children.
3. The base class is made abstract, so that one has to derive (extend) a class to use it.
4. An abstract method (must be defined in derived class) is added to defining an action to perform on each node in the tree that is specific to a derived class.
5. The base class defines an Add() method for adding a child (branch) to the node.
6. The base class defines a Children() method for accessing all the children (branches) of the node.

Python

```
# Base (Super) class definition for a k-ary tree
```

```
class KNode:
        key = None # node data
        k = 0      # max number of children

        # Dynamic array for children
        children = DynamicArray()

        # constructor: set the node data
        def __init__ ( self, k, key ):
                self.key = key
                self.k   = k

        # Set the node data
        def Key( self, key ):
                self.key = key

        # Get the node data
        def GetKey( self ):
                return self.key

        # Get the number of children
        def Size( self ):
                return self.children.Size()

        # Add another child (branch)
        def Add( self, key ):
                # exceeds number of allowed children
                if Size() == self.k:
                        return None
                e = Tree( self.k, key )
                self.children.add( e )
                return e

        # Get all the children of this node
        def Children( self ):
                return this.children

# Definition for a k-ary tree
class Tree(KNode):
        def __init__( self, k, key ):
```

```
                KNode.__init__( k, key )

        def Action( self ):
                self # do something
```

Java

```
import java.util.ArrayList;

// Base (Super) class definition for a k-ary tree
abstract class KNode {
        // node data
        private Object key;
        // max number of children
        private final int k;

        // Dynamic array for children
        private ArrayList children = new ArrayList();

        // action to perform on a node
        public abstract void Action();

        // constructor: set the node data
        public KNode( int k, Object key ) {
                this.key = key;
                this.k   = k;
        }

        // Set the node data
        public void Key( Object key ) {
                this.key = key;
        }

        // Get the node data
        public Object Key() {
                return key;
        }

        // Get the number of children
        public int Size() {
                return children.size();
```

```
        }

        // Add another child (branch)
        public KNode Add( Object key ) {
                // exceeds number of allowed children
                if ( Size() == k )
                        return null;
                Tree e = new Tree( k, key );
                children.add( e );
                return e;
        }

        // Get all the children of this node
        public ArrayList Children() {
                return children;
        }
}

// Definition for a k-ary tree
class Tree extends KNode {
        public Tree( int k, Object key ) {
                super( k, key );
        }

        public void Action() {
                // do something
        }
}
```

9. Binary Search Tree

A binary search tree (BST) is a sorted tree where at each node, the data value of the left branch (child) is less than the data value of the node, and the right node is greater than, and where there are no duplicate values. For example, if the data value of the root node is 5, and we added a node with value 3 it would be inserted on the left branch, while a node of 7 would be inserted on the right branch.

You maybe asked to code the algorithm to insert, find and delete nodes in a binary search tree.

Algorithm - Insert

1. If the root node is null, make the root the node.
2. Otherwise, starting at the root, traverse the tree:

- If the node is equal to the current node, then return (duplicate).
- If the node is less than the current node:
  - If the left node is null, then make the node the left node.
  - Otherwise, traverse to the left node.
- If the node is greater than the current node:
  - If the right node is null, then make the node the right node.
  - Otherwise, traverse to the right node.

Python

```
# Definition for a binary search tree
class BST:
        root = None # root of the binary search tree

        # Insert a node into a binary tree
        def Insert( self, node ):
                # Empty Tree, make the node the root
                if self.root == None:
                        self.root = node
                        return

                # Follow a path to insert the node
                curr = self.root
                while True:
                        # node is a duplicate, do not insert
                        if int( node.GetKey() ) ==  int( curr.GetKey() ):
                                return

                        # The node is less than the current node, traverse left
                        if int( node.GetKey() )  < int( curr.GetKey() ):
                                # If there is no left node, make this node the left node
                                if curr.GetLeft() == None:
                                        curr.Left( node )
                                        return
                                curr = curr.GetLeft()
                        # The node is greater than to the current node, traverse left
                        else:
                                # If there is no right node, make this node the right node
                                if curr.GetRight() == None:
                                        curr.Right( node )
                                        return
```

```
                                curr = curr.GetRight()

tree = BST()
tree.Insert( BinaryTree( 3 ) )
tree.Insert( BinaryTree( 6 ) )
tree.Insert( BinaryTree( 2 ) )
tree.Insert( BinaryTree( 4 ) )
tree.Insert( BinaryTree( 1 ) )
print("BST level order: 3 2 6 1 4")
BFS( tree.root )
```

Java

```
// Definition for a binary search tree
class BST {
        public BinaryTree root = null;  // root of the binary search tree

        // Insert a node into a binary tree
        public void Insert( BinaryTree node ) {
                // Empty Tree, make the node the root
                if ( root == null ) {
                        root = node;
                        return;
                }

                // Follow a path to insert the node
                BinaryTree curr = root;
                while ( true ) {
                        // node is a duplicate, do not insert
                        if ( (int) node.Key() == (int) curr.Key() ) {
                                return;
                        }

                        // The node is less than the current node, traverse left
                        if ( (int) node.Key() < (int) curr.Key() ) {
                                // If there is no left node, make this node the left node
                                if ( curr.Left() == null ) {
                                        curr.Left( node );
                                        return;
                                }
                                curr = curr.Left();
```

```
                        }
                        // The node is greater than to the current node, traverse left
                        else {
                                // If there is no right node, make this node the right node
                                if ( curr.Right() == null ) {
                                        curr.Right( node );
                                        return;
                                }
                                curr = curr.Right();
                        }
                }
        }

        // Driver method
    public static void main(String[] args)
    {
        BST tree = new BST();
                System.out.println("BST level order: 3 2 6 1 4");
                tree.Insert( new BinaryTree( 3 ) );
                tree.Insert( new BinaryTree( 6 ) );
                tree.Insert( new BinaryTree( 2 ) );
                tree.Insert( new BinaryTree( 4 ) );
                tree.Insert( new BinaryTree( 1 ) );
                BFS( tree.root );
        }
}
```

Algorithm - Find

1. Starting at the root, traverse a path in the tree.
2. If the current node is null, return not found.
3. If the value is equal to the current node, return found.
4. If the value is less than the current node, traverse to the left node.
5. Otherwise (the value is greater than the current node), traverse to the right node.

Python

```
# Definition for a binary search tree
class BST:
        ...
        # Find a node in a binary tree
```

```
        def Find( self, key ):

                # Follow a path to find the node
                curr = self.root
                while True:
                        # current node is null, return null (not found)
                        if curr == None:
                                return None

                        # the value is equal to the current node, return the current node
                        if int( key ) == int( curr.GetKey() ):
                                return curr

                        # The value is less than the current node, traverse left
                        if int( key ) <  int( curr.GetKey() ):
                                curr = curr.GetLeft()
                        # The value is greater than to the current node, traverse left
                        else:
                                curr = curr.GetRight()

# Driver Method
tree = BST()
tree.Insert( BinaryTree( 3 ) )
tree.Insert( BinaryTree( 6 ) )
tree.Insert( BinaryTree( 2 ) )
tree.Insert( BinaryTree( 4 ) )
tree.Insert( BinaryTree( 1 ) )
print("BST level order: 3 2 6 1 4")
BFS( tree.root )
print( "FIND(4) = " + str( tree.Find( 4 ) ) + " FIND(0) = " + str( tree.Find(0) ) )
```

Java

```
// Definition for a binary search tree
class BST {
        ...

        // Find a node in a binary tree
        public BinaryTree Find( Object key ) {

                // Follow a path to find the node
```

```
                BinaryTree curr = root;
                while ( true ) {
                        // current node is null, return null (not found)
                        if ( curr == null )
                                return null;

                        // the value is equal to the current node, return the current node
                        if ( (int) key == (int) curr.Key() ) {
                                return curr;
                        }

                        // The value is less than the current node, traverse left
                        if ( (int) key < (int) curr.Key() ) {
                                curr = curr.Left();
                        }
                        // The value is greater than to the current node, traverse left
                        else {
                                curr = curr.Right();
                        }
                }
        }

        // Driver method
        public static void main(String[] args)
        {
        BST tree = new BST();
                tree.Insert( new BinaryTree( 3 ) );
                tree.Insert( new BinaryTree( 6 ) );
                tree.Insert( new BinaryTree( 2 ) );
                tree.Insert( new BinaryTree( 4 ) );
                tree.Insert( new BinaryTree( 1 ) );
                System.out.println( "FIND(4) = " + tree.Find( 4 ) + " FIND(0) = " + tree.Find(0) );
        }
}
```

Algorithm - Delete

1. Traverse the tree, starting with the root and remember the previous node visited.
2. If the current node is null, return (not found).
3. If the value is equal to the current node:
    - If the node is the left node of the previous node, set the previous node's left node to null.

- Otherwise (is the right node of the previous node), set the previous node's right node to null.
- Remember the root and temporarily set the root to the previous node.
- If the left node of the current (deleted) node is non-null, then Insert() the current node's left node back into the tree.
- If the right node of the current (deleted) node is non-null, then Insert() the current node's right node back into the tree.

4. If the value is less than the current node:
   - Set a flag that the current node is the left node of the current node.
   - Set the previous node to the current node.
   - Set the current node to the left node of the previous node.
5. Otherwise (value is greater than the current node):
   - Set a flag that the current node is the right node of the current node.
   - Set the previous node to the current node.
   - Set the current node to the right node of the previous node.

Python

```
# Definition for a binary search tree
class BST:
        ...

        # Find a node in a binary tree
        def Delete( self, key ):

                # Follow a path to find the node
                curr = self.root
                prev = None
                isLeft = False
                while True:
                        # current node is null, return (not found)
                        if curr == None:
                                return

                        # the value is equal to the current node, delete the node
                        if int( key ) == int( curr.GetKey() ):
                                if isLeft:
                                        prev.Left( None )
                                else:
                                        prev.Right( None )
                                saveRoot = self.root;
                                self.root = prev;
                                if curr.GetLeft()  != None:
                                        self.Insert( curr.GetLeft() )
```

```
                                if curr.GetRight() != None:
                                        self.Insert( curr.GetRight() )
                                self.root = saveRoot
                                return
                        elif int( key ) < int( curr.GetKey() ):
                                isLeft = True
                                prev = curr
                                curr = curr.GetLeft()
                        else:
                                isLeft = False
                                prev = curr
                                curr = curr.GetRight();

# Driver method
tree = BST()
tree.Insert( BinaryTree( 3 ) )
tree.Insert( BinaryTree( 6 ) )
tree.Insert( BinaryTree( 2 ) )
tree.Insert( BinaryTree( 4 ) )
tree.Insert( BinaryTree( 1 ) )
tree.Delete( 6 )
print("BST level order: 3 2 4 1")
BFS( tree.root )
```

Java

```
// Definition for a binary search tree
class BST {
        ...

        // Find a node in a binary tree
        public void Delete( Object key ) {

                // Follow a path to find the node
                BinaryTree curr = root, prev = null;
                boolean isLeft = false;
                while ( true ) {
                        // current node is null, return (not found)
                        if ( curr == null )
                                return;
```

```
                        // the value is equal to the current node, delete the node
                        if ( (int) key == (int) curr.Key() ) {
                                if ( isLeft )
                                        prev.Left( null );
                                else
                                        prev.Right( null );
                                BinaryTree saveRoot = root;
                                root = prev;
                                if ( curr.Left()  != null ) Insert( curr.Left() );
                                if ( curr.Right() != null ) Insert( curr.Right() );
                                root = saveRoot;
                                return;
                        }
                        else if ( (int) key < (int) curr.Key() ) {
                                isLeft = true;
                                prev = curr;
                                curr = curr.Left();
                        }
                        else {
                                isLeft = false;
                                prev = curr;
                                curr = curr.Right();
                        }
                }
        }

        // Driver method
    public static void main(String[] args)
    {
        BST tree = new BST();
                tree.Insert( new BinaryTree( 3 ) );
                tree.Insert( new BinaryTree( 6 ) );
                tree.Insert( new BinaryTree( 2 ) );
                tree.Insert( new BinaryTree( 4 ) );
                tree.Insert( new BinaryTree( 1 ) );
                tree.Delete( 6 );
                System.out.println("BST level order: 3 2 4 1");
                BFS( tree.root );
        }
}
```

10. Arithmetic Operations

A very basic arithmetic problem you maybe asked is to implement a multiply operation where you can only use the addition and equality operator.

Algorithm - Multiply

1. Initialize accumulators for result and repeat to 0.
2. Loop indefinitely for 'x + y'
    - If the repeat accumulator equals y, then break.
    - Increment the repeat accumulator by one.
    - Add x to the result accumulator.

Python

```
# Multiple using only + and = operator
def Mul( x, y ):
        result = 0  # accumlator for the result
        repeat = 0  # accumlator for the number of times to repeat the addition

        while ( True ):
                # starting at zero, exit the loop after y+1 times
                repeat += 1
                if repeat == y + 1:
                        break
                # accumulate the result by adding x to the accumulator
                result += x;

        return result

print( Mul( 5, 6 ) )
```

Java

```
public class Calc
{
        // Multiple using only + and = operator
        public static int Mul( int x, int y ) {
                int result = 0; // accumlator for the result
                int repeat = 0; // accumlator for the number of times to repeat the addition

                while ( true ) {
                        // starting at zero, exit the loop after y+1 times
```

```
                        if ( repeat++ == y )
                                break;
                        // accumulate the result by adding x to the accumulator
                        result += x;
                }
                return result;
        }

        public static void main(String[] args) {
                System.out.println( "5 * 6 = " + Mul( 5, 6 ) );
        }
}
```

You would then be likely asked to extend the solution to include an exponent operation where you can only use the addition and equality operator.

Algorithm - Exponent

1. If the exponent is zero, return 1.
2. Initialize accumulators for result to 1 and repeat to 0.
3. Loop indefinitely for 'x power e'
    - If the repeat accumulator equals e, then break.
    - Increment the repeat accumulator by one.
    - Update the result accumulator to Mul(x, result).

Python

```
...
# Exponent using only + and = operator
def Exp( x, e ):
        # x raised to power of 0 is always 1
        if e == 0:
                return 1

        result = 1      # accumlator for the result
        repeat = 0      # accumlator for the number of times to repeat the multiplication
        while ( True ):
                repeat += 1
                if repeat == e + 1:
                        break
                result = Mul( x, result );

        return result
```

```
print( Exp( 3, 4 ) )
```

Java

```
public class Calc
{
        ...
        // Exponent using only + and = operator
        public static int Exp( int x, int e ) {
                // x raised to power of 0 is always 1
                if ( e == 0 )
                        return 1;

                int result = 1; // accumlator for the result
                int repeat = 0; // accumlator for the number of times to repeat the multiplication
                while ( true ) {
                        if ( repeat++ == e )
                                break;
                        result = Mul( x, result );
                }
                return result;
        }

        public static void main(String[] args) {
                System.out.println( "3 exp 4 = " + Exp( 3, 4 ) );
        }
}
```

You maybe asked to further optimize the exponent algorithm by reducing the number of multiplications in half. This can be done by factoring the equation into fewer multiplication steps. There are two factors we can use that reduced the exponent in half, resulting in half the multiplications, one for even exponents and one for odd exponents.

Algorithm

1. Use the following factoring rules:
    1. x**e = (x**a)*b, if a*b = e
    2. x**e = x**a * x**b, if a+b = e
2. For even exponents, choose a = 2 and b = e/2, where x**e = (x**2)**(e/2).
3. For odd exponents, choose a = 1 and b = e - 1 to make the exponent even and then apply the even method, where x**e = x * ((x**2)**((e-1)/2).

Python

```
# Exponent with half the multiplications
def Exp2( x, e ):
        if e == 0:
                return 1
        # even
        if ( e % 2 ) == 0:
                return Exp( x * x, ( e / 2 ) )
        # odd
        else:
                return x * Exp( x * x, ( ( e - 1 ) / 2  ) )

print( Exp2( 3, 4 ) )
```

Java

```
public class Calc
{
        ...
        // Exponent with half the multiplications
        public static int Exp2( int x, int e ) {
                if ( e == 0 )
                        return 1;
                // even
                if ( ( e % 2 ) == 0 )
                        return Exp( x * x, ( e / 2 ) );
                // odd
                else
                        return x * Exp( x * x, ( ( e - 1 ) / 2  ) );
        }

        public static void main(String[] args) {
                System.out.println( "3 exp 4 = " + Exp2( 3, 4 ) );
        }
}
```

Another algorithm you may be asked to code is Euclid's algorithm for find the great common denominator between two numbers. The algorithm dates back to 300BC. It is a very efficient algorithm that involves a few small steps. It can be coded as iteratively or recursively.

Algorithm - GCD Iterative Solution

1. Calculate the remainder of dividing the 2nd number by the first number (y % x).

2. Swap the value of the first number (x) with the second number (y).
3. Set the second number (y) to the remainder.
4. Iteratively repeat the process until the 2nd number (y) is zero.
5. Return the value of first number (x) when the 2nd number is zero (y).

Python

```
# Euclid's algorithm as an iterative solution
def GCD( x, y ):
        # continue until the division of of the two numbers does not leave a remainder (evenly divisible)
        while y > 0:
                # calculate the remainder of the division by between x and y
                remainder = ( y % x )

                # swap the value of x with y
                x = y

                # set y to the remainder
                y = remainder

        return x

print( "GCD " + str( GCD( 12, 16 ) ) )
```

Java

```
public class GCD {
        // Euclid's algorithm as an iterative solution
        private static int GCD( int x, int y )
        {
                // continue until the division of of the two numbers does not leave a remainder (evenly divisible)
                while ( y > 0 )
                {
                        // calculate the remainder of the division by between x and y
                        int remainder = ( y % x );

                        // swap the value of x with y
                        x = y;

                        // set y to the remainder
                        y = remainder;
                }
```

```
                return x;
        }

        public static void main(String[] args ) {
                System.out.println( "GCD(4) " + GCD( 12, 16 ) );
        }
}
```

You maybe then be asked to code the solution using recursion.

Algorithm - GCD Recursive Solution

1. When the 2nd number (y) is reduced to zero, return the current value of the first number (x).
2. Otherwise, swap the first number (x) with the second number (y) and set the 2nd number (y) to the remainder of the division of the 2nd number by the first number (x % y).
3. Recursively call GCD() with the updated first and second numbers.

Python

```
# Euclid's algorithm as a recursive solution
def GCDR( x, y ):
        # return the current value of x when y is reduced to zero.
        if y == 0:
                return x

        return GCDR( y, ( y % x ) )

print( "GCD " + str( GCDR( 12, 16 ) ) )
```

Java

```
public class GCD {
        // Euclid's algorithm as a recursive solution
        private static int GCD( int x, int y )
        {
                // return the current value of x when y is reduced to zero.
                if ( y == 0 )
                        return x;

                return GCD( y, ( y % x ) );
        }
```

```
        public static void main(String[] args ) {
                System.out.println( "GCD(4) " + GCD( 12, 16 ) );
        }
}
```

Finally, you maybe asked to extend your solution to additionally find the least common multiple (denominator). The least common multiple (LCM) is the product of the two numbers divided by the GCD of the two numbers.

Algorithm - LCM

1. Multiple the first and second number.
2. Divide the result by the GCD of the two numbers.

Python

```
# Calculate the least common multiple
def LCM( x, y ):
        return ( x * y ) / GCD( x, y )

print( "LCM " + str( LCM( 6, 8 ) ) )
```

Java

```
public class GCD {
        ...
        // Calculate the least common mulitple
        private static int LCM( int x, int y )
        {
                return ( x * y ) / GCD( x, y );
        }

        public static void main(String[] args ) {
                System.out.println( "LCM(24) " + LCM( 6, 8 ) );
        }
}
```

11. Variable Length Byte Encoding

You maybe asked to code a variable length byte encoding, using the least amount of bits for the encoding across all sequences. For example, some encodings will be one byte, some two bytes, etc. In an encoding with between 1 and 8 bytes, if we restrict the first byte to must contain a zero bit, then we can use the position of the zero bit to determine the byte length, while retaining the use of the less significant bits for the encoding.

Algorithm

1. If the most significant bit is zero, then the encoding is one byte long and the remaining bits are the encoding.
2. If the most significant bit is one, the bit position (most to least significant) is the number of bytes in the encoding, and the remaining less significant bits in the first byte and all the bits in the remaining bytes are the encoding.

Python

```
def Size( encoding ):
        first = encoding[ 0 ]
        # checking leading bit
        if ( first & 0x80 ) == 0:
                return 1
        if ( first & 0x40 ) == 0:
                return 2
        if ( first & 0x20 ) == 0:
                return 3
        if ( first & 0x10 ) == 0:
                return 4
        if ( first & 0x08 ) == 0:
                return 5
        if ( first & 0x04 ) == 0:
                return 6
        if ( first & 0x02 ) == 0:
                return 7
        if ( first & 0x01 ) == 0:
                return 8
        return 0        # error

encoding = [ 0 ]
encoding[ 0 ] = 0x49
print( "Size(1) = " + str( Size( encoding ) ) )
encoding[ 0 ] = 0x89
print( "Size(2) = " + str( Size( encoding ) ) )
encoding[ 0 ] = 0xCB
print( "Size(3) = " + str( Size( encoding ) ) )
```

Java

```
public class Encoding {
        public static int Size( byte[] encoding ) {
                byte first = encoding[ 0 ];
```

```
                // checking leading bit
                if ( ( first & 0x80 ) == 0 )
                        return 1;
                if ( ( first & 0x40 ) == 0 )
                        return 2;
                if ( ( first & 0x20 ) == 0 )
                        return 3;
                if ( ( first & 0x10 ) == 0 )
                        return 4;
                if ( ( first & 0x08 ) == 0 )
                        return 5;
                if ( ( first & 0x04 ) == 0 )
                        return 6;
                if ( ( first & 0x02 ) == 0 )
                        return 7;
                if ( ( first & 0x01 ) == 0 )
                        return 8;
                return 0;       // error
        }

        public static void main(String[] args) {
                byte[] encoding = new byte[8];
                encoding[ 0 ] = (byte) 0x49;
                System.out.println( "Size(1) = " + Size( encoding ) );
                encoding[ 0 ] = (byte) 0x89;
                System.out.println( "Size(2) = " + Size( encoding ) );
                encoding[ 0 ] = (byte) 0xCB;
                System.out.println( "Size(3) = " + Size( encoding ) );
        }
}
```

12. Sorting

There are numerous basic algorithms for sorting. You may get asked to start with a simple sort algorithm and progressively code more complex algorithms. One simple algorithm is a bubble sort. It works by going through a list and comparing adjacent values, and swapping them if the first value is greater than the second value (ascending) or vice-versa (descending). Repeated passes are made through the list until no values are swapped. The term bubble sort comes from the concept that the smaller (or larger) values slowly bubble up. The performance is poor and has a time complexity of O(n*n).

Algorithm - Bubble Sort

1. Make a pass through the list of values.

- Compare adjacent values.
- If the first value is less than the second value, swap the values.

2. If one of more values were swapped, repeat the process of making a pass through the list.

Python

```
# Definition for a Bubble Sort
def BubbleSort( data ):
        swapped = True
        # continue to repeat until no more adjacent values are swapped
        while swapped:
                swapped = False
                # Make a scan through the list
                for i in range( 0, len( data ) - 1 ):
                        # Compare adjacent values. If the first value > second value, swap the values.
                        if data[ i ] > data[ i + 1 ]:
                                swap = data[ i ]
                                data[ i ] = data[ i + 1 ]
                                data[ i + 1 ] = swap
                                swapped = True
        return data

data = [ 5, 2, 4, 1, 6, 3 ]
data = BubbleSort( data )
for i in range( 0, len( data ) ):
        print( data[ i ] )
```

Java

```
class Sort {
        // Definition for a Bubble Sort
        public static int[] BubbleSort( int[] data ) {
                boolean swapped = true;
                // continue to repeat until no more adjacent values are swapped
                while ( swapped ) {
                        swapped = false;
                        // Make a scan through the list
                        for ( int i = 0; i < data.length - 1; i++ ) {
                                // Compare adjacent values. If the first value > second value, swap the values.
                                if ( data[ i ] > data[ i + 1 ] ) {
                                        int swap = data[ i ];
                                        data[ i ] = data[ i + 1 ];
```

```
                                        data[ i + 1 ] = swap;
                                        swapped = true;
                                }
                        }
                }
                return data;
        }

        public static void main( String[] args ) {
                int[] data = new int[ 6 ];
                data[ 0 ] = 5; data[ 1 ] = 2; data[ 2 ] = 4; data[ 3 ] = 1; data[ 4 ] = 6; data[ 5 ] = 3;
                data = BubbleSort( data );
                for ( int i = 0; i < data.length; i++ )
                        System.out.println( data[ i ] );
        }
}
```

Another simple sorting algorithm you maybe asked to code is an insertion sort. It makes N iterations on the list of elements. On each iteration it advances to the next element and inserts the element into the correct position in the list. It's performance is good on small sets, but poor on large sets with a time complexity of O(n*n).

Algorithm - Insertion Sort

1. Iterate through the list of elements.
    - Remove and Insert the element into the correct position in the list of elements proceeding it.

Python

```
# Definition for an Insertion Sort
def InsertionSort( data ):
        # iterate through the list for each element except the first element
        for i in range( 1, len( data ) ):
                # starting with the current element, remove/insert proceeding
                # elements so they are in sorted order
                for j in range( i, 0, -1):
                        # swap adjacent elements
                        if data[ j ] < data[ j - 1 ]:
                                temp = data[ j ]
                                data[ j ] = data[ j -1 ]
                                data[ j - 1 ] = temp

        return data
```

```
data = [ 5, 2, 4, 1, 6, 3 ]
data = InsertionSort( data )
for i in range( 0, len( data ) ):
        print( data[ i ] )
```

Java

```
class Sort {
        // Definition for an Insertion Sort
        public static int[] InsertionSort( int[] data ) {
                // iterate through the list for each element except the first element
                for ( int i = 1; i < data.length; i++ ) {
                        // starting with the current element, remove/insert proceeding
                        // elements so they are in sorted order
                        for ( int j = i; j > 0; j-- ) {
                                // swap adjacent elements
                                if ( data[ j ] < data[ j - 1 ] ) {
                                        int temp = data[ j ];
                                        data[ j ] = data[ j -1 ];
                                        data[ j - 1 ] = temp;
                                }
                        }
                }
                return data;
        }

        public static void main( String[] args ) {
                int[] data = new int[ 6 ];
                data[ 0 ] = 5; data[ 1 ] = 2; data[ 2 ] = 4; data[ 3 ] = 1; data[ 4 ] = 6; data[ 5 ] = 3;
                data = InsertionSort( data );
                for ( int i = 0; i < data.length; i++ )
                        System.out.println( data[ i ] );
        }
}
```

Another simple sorting algorithm you maybe asked to code is a quick sort. This is a divide and conquer algorithm. It works by continuously dividing the list in half. On each iteration, a midpoint in the list is selected. The value at the midpoint is referred to as the pivot. Iterations are made to swap values between the partitions where all values less than the pivot are moved to the first (left) partition and all values greater than or equal are on the second (right) partition. The process is then applied recursively to each partition. The average time complexity is O(n*logn), with a worst case complexity of O(n*n).

Algorithm - Quick Sort

1. Calculate a midpoint in the list. The value at the midpoint is the pivot value.
2. Move all values in the first partition that are not less than the pivot to the second partition.
3. Move all values in the second partition that are not greater than or equal to the pivot to the first partition.
4. Recursively apply the algorithm to the two partitions.

Python

```
# Definition for an Quick Sort
def QuickSort( data ):
        qSort( data, 0, len( data )  - 1 )
        return data

def qSort( data, low, high ):
        i     = low                       # starting lower index
        j     = high                      # starting higher index
        mid   = int( low + ( high - low ) / 2 )  # midway index
        pivot = data[ mid ]               # pivot, value at the midway index

        # Divide the array into two partitions
        while i <= j:
                # keep advancing (ascending) the lower index until we find a value that is not less than the pivot
                # we will move this value to the right half partition.
                while data[ i ] < pivot:
                        i += 1
                # keep advancing (descending) the higher index until we find a value that is not greater than the pivot
                # we will move this value to the left half partition.
                while data[ j ] > pivot:
                        j -= 1

                # if the lower index has past the higher index, there is no values to swap
                # otherwise, swap the values and continue
                if i <= j:
                        # swap the higher than pivot value on the left side with the lower than pivot value on the right side
                        temp  = data[ i ]
                        data[ i ] = data[ j ]
                        data[ j ] = temp

                        # advance the lower and higher indexes accordingly and continue
                        i += 1
                        j -= 1
```

```
        # recursively sort the two partitions if the index has not crossed over the pivot index
        if low < j:
                qSort( data, low, j )
        if i < high:
                qSort( data, i, high )

data = [ 5, 2, 4, 1, 6, 3 ]
data = QuickSort( data )
for i in range( 0, len( data ) ):
        print( data[ i ] )
```

Java

```
class Sort {
        // Definition for an Quick Sort
        public static int[] QuickSort( int[] data ) {
                qSort( data, 0, data.length - 1 );
                return data;
        }

        private static void qSort( int[] data, int low, int high ) {
                int i     = low;                                          // starting lower index
                int j     = high;                                         // starting higher index
                int mid   = low + ( high - low ) / 2;   // midway index
                int pivot = data[ mid ];                                  // pivot, value at the midway index

                // Divide the array into two partitions
                while ( i <= j ) {
                        // keep advancing (ascending) the lower index until we find a value that is not less than the pivot
                        // we will move this value to the right half partition.
                        while ( data[ i ] < pivot ) {
                i++;
            }
                        // keep advancing (descending) the higher index until we find a value that is not greater than the pivot
                        // we will move this value to the left half partition.
                        while ( data[ j ] > pivot ) {
                j--;
            }
                        // if the lower index has past the higher index, there is no values to swap
                        // otherwise, swap the values and continue
```

```
                        if ( i <= j ) {
                                // swap the higher than pivot value on the left side with the lower than pivot value on the right side
                                int temp  = data[ i ];
                                data[ i ] = data[ j ];
                                data[ j ] = temp;

                // advance the lower and higher indexes accordingly and continue
                i++;
                j--;
            }
                }

                // recursively sort the two partitions if the index has not crossed over the pivot index
        if ( low < j )
            qSort( data, low, j );
        if ( i < high )
            qSort( data, i, high );
        }

        public static void main( String[] args ) {
                int[] data = new int[ 6 ];
                data[ 0 ] = 5; data[ 1 ] = 2; data[ 2 ] = 4; data[ 3 ] = 1; data[ 4 ] = 6; data[ 5 ] = 3;
                data = QuickSort( data );
                for ( int i = 0; i < data.length; i++ )
                        System.out.println( data[ i ] );
        }
}
```

Another common divide and conquer sorting algorithm you maybe asked to code is a merge sort. This algorithm divides a list into partitions of single elements, and then progressively merges the partitions together and sorts them. This algorithm can be implemented either top-down (recursive) or bottom up (iterative). The average and worst time complexity is O(n*logn).

Algorithm - Merge Sort (top-down)

1. Allocate space for a temporary copy of the array.
2. Recursively split the data into two partitions (halves), until each partition is a single element.
3. Merge and sort each pair of partitions.

Python

```
# Definition for a Merge Sort
def MergeSort( data ):
```

```
        # allocate space for a temporary copy of the data
        tdata = [ 0 ] * len( data );

        # sort the data (pass in the temporary copy so routine is thread safe)
        mSort( data, 0, len( data ) - 1, tdata )
        return data

def mSort( data, low, high, tdata ):
        # if the partition has more than one element, then recursively divide the partition and merge the parts back in
        if low < high:
                mid = int( low + ( high - low ) / 2 )   # midway index

                # sort the lower (first) half partition
                mSort( data, low, mid, tdata )
                # sort the upper (second) half partition
                mSort( data, mid + 1, high, tdata )

                # merge the partitions together
                merge( data, low, mid, high, tdata )

def merge( data, low, mid, high, tdata ):
        # make a temporary copy of the two separately sorted partitions
        for i in range( low, high + 1 ):
           tdata[ i ] = data[ i ]

        # starting from the beginning of the first partition, iteratively search for the next lowest
        # number from the lower (first) and higher (second) and move into current position in the
        # lower (first) partition
        i = low
        k = low
        j = mid + 1
        while i <= mid and j <= high:
                if tdata[ i ] <= tdata[ j ]:
                        data[ k ] = tdata[ i ]
                        i += 1
                else:
                        data[ k ] = tdata[ j ]
                        j += 1
                k += 1
```

```
        # Copy any remaining elements back into the first partition
        while i <= mid:
                data[ k ] = tdata[ i ]
                k += 1
                i += 1

data = [ 5, 2, 4, 1, 6, 3 ]
data = MergeSort( data )
for i in range( 0, len( data ) ):
        print( data[ i ] )
```

Java

```
class Sort {
        // Definition for a Merge Sort
        public static int[] MergeSort( int[] data ) {
                // allocate space for a temporary copy of the data
                int[] tdata = new int[ data.length ];

                // sort the data (pass in the temporary copy so routine is thread safe)
                mSort( data, 0, data.length - 1, tdata );
                return data;
        }

        private static void mSort( int[] data, int low, int high, int[] tdata ) {
                // if the partition has more than one element, then recursively divide the partition and merge the parts back in
                if ( low < high ) {
                        int mid   = low + ( high - low ) / 2;   // midway index

                        // sort the lower (first) half partition
                        mSort( data, low, mid, tdata );
                        // sort the upper (second) half partition
                        mSort( data, mid + 1, high, tdata );

                        // merge the partitions together
                        merge( data, low, mid, high, tdata );
                }
        }

        private static void merge( int[] data, int low, int mid, int high, int[] tdata ) {
                // make a temporary copy of the two separately sorted partitions
```

```
                for ( int i = low; i <= high; i++ )
            tdata[ i ] = data[ i ];

                // starting from the beginning of the first partition, iteratively search for the next lowest
                // number from the lower (first) and higher (second) and move into current position in the
                // lower (first) partition
                int i = low, k = low;
        int j = mid + 1;
                while ( i <= mid && j <= high ) {
                        if ( tdata[ i ] <= tdata[ j ] ) {
                data[ k ] = tdata[ i ];
                i++;
            } else {
                data[ k ] = tdata[ j ];
                j++;
            }
            k++;
                }

                // Copy any remaining elements back into the first partition
                while ( i <= mid ) {
            data[ k ] = tdata[ i ];
            k++;
            i++;
        }
        }

        public static void main( String[] args ) {
                int[] data = new int[ 6 ];
                data[ 0 ] = 5; data[ 1 ] = 2; data[ 2 ] = 4; data[ 3 ] = 1; data[ 4 ] = 6; data[ 5 ] = 3;
                data = MergeSort( data );
                for ( int i = 0; i < data.length; i++ )
                        System.out.println( data[ i ] );
        }
}
```

13. Hashing

Hashing is another common area to be asked questions about. Hashing is used to solve indexing lookups without the overhead of sequential scanning and name comparisons. The most basic use of hashing is insertion and lookup in a key-value(s) dictionary. Please note that sometimes people mistake checksum methods as part of hashing, they are not.

In hashing, we create an index where the entries point to the location of the corresponding values. The index is constructed so we do not have to do a sequential scan to find the entry. The most basic index is an integer indexed array. The keys are mapped (hashed) to an integer value which is in the range of the indexed array. A direct array access is then done to retrieve the entry. All hash algorithms have common issues to resolve:

1. Handling of collisions when two keys mapped to the same index.
2. Efficiency of the hashing algorithm.
3. The frequency of collisions.
4. The sparseness of the memory allocated for the index.

The basic algorithm you maybe asked to code is to create an index for large integers that maps into a much smaller range, consuming less memory, handle collisions (two integers map to the same index), and updates. Typically one would use the module operator (integer remainder) to divide the number by the range size and use the remainder as the index.

Algorithm

1. Map each integer value into a smaller fixed size integer range using the modulo operator.
2. If there is no entry at the index, add the key/value to the index as the first entry in the chain.
3. If there are entries there, and the key is the same as one of the entries (duplicate), update the value.
4. If there are entries there, and the key is not the same (collision) as any entry, add the key to the chain of entries.

Python

```
# Definition for an Entry in the Hash Table (Dictionary)
class Entry:
        key = 0      # the key value
        value = None # the value for this key
        next  = None # the next entry at this range

        # Constructor
        def __init__( self, key,value ):
                self.key   = key
                self.value = value

        # Get the key
        def GetKey( self ):
                return self.key

        # Accessors for setting and getting the value of the entry
        def GetValue( self ):
                return self.value
```

```
        def Value( self, value ):
                self.value = value

        # Accessors for setting and getting the next entry on the chain
        def GetNext( self ):
                return self.next

        def Next( self, next ):
                self.next = next

        # Comparator for checking if key matches this entry.
        def Compare( self, key ):
                return ( self.key == key )

# Definition for a Modulo based Hashing Algorithm
class Hash:
        RANGE = 0    # the range of the index.
        index = []   # the index

        # constructor
        def __init__( self, range ):
                # set the index range and allocate the index
                self.RANGE = range
                self.index = [None] * self.RANGE

        # Map the key into an index within the set range
        def Index( self, key ):
                return key % self.RANGE

        # Add a key/value entry to the index
        def Add( self, key, value ):
                ix = self.Index( key )

                # there is no entry at this index, add the key/value
                if self.index[ ix ] == None:
                        self.index[ ix ] = Entry( key, value )
                else:
                        # See if the key already exists in the chain
                        next = self.index[ ix ]
                        while next != None:
```

```
                                # Entry found, update the value
                                if next.Compare( key ):
                                        next.Value( value )
                                        break
                                next = next.GetNext()

                        # no entry found, add one
                        if next == None:
                                # Add the entry to the front of the chain
                                add = Entry( key, value )
                                add.Next( self.index[ ix ] )
                                self.index[ ix ] = add

        # Get the value for the key
        def Get( self, key ):
                ix = self.Index( key )

                # See if the key exists in the chain at this entry in the index
                next = self.index[ ix ]
                while next != None:
                        # Entry found, update the value
                        if next.Compare( key ):
                                return next.GetValue()
                        next = next.GetNext()

                # not found
                return None

index = Hash( 100 )
index.Add( 17,   100 )
index.Add( 117,  600 ) # this will cause a collision
index.Add( 228,  300 )
index.Add( 675,  400 )
index.Add( 2298, 500 )
index.Add( 117,  200 ) # this will cause an update
print( index.Get( 17 ) )
print( index.Get( 117 ) )
print( index.Get( 228 ) )
print( index.Get( 675 ) )
print( index.Get( 2298 ) )
```

Java

```
// Definition for an Entry in the Hash Table (Dictionary)
class Entry {
        private int    key;                  // the key value
        private Object value = null; // the value for this key
        private Entry  next  = null; // the next entry at this range

        // Constructor
        public Entry( int key, Object value ) {
                this.key   = key;
                this.value = value;
        }

        // Get the key
        public int Key() {
                return key;
        }

        // Accessors for setting and getting the value of the entry
        public Object Value() {
                return value;
        }

        public void Value( Object value ) {
                this.value = value;
        }

        // Accessors for setting and getting the next entry on the chain
        public void Next( Entry next ) {
                this.next = next;
        }

        public Entry Next() {
                return next;
        }

        // Comparator for checking if key matches this entry.
        public boolean Compare( int key ) {
                return ( this.key == key );
        }
```

```
}

// Definition for a Modulo based Hashing Algorithm
public class Hash {
        private final int RANGE;       // the range of the index.
        private Entry[] index = null;  // the index

        // constructor
        public Hash( int range ) {
                // set the index range and allocate the index
                RANGE = range;
                index = new Entry[ RANGE ];
        }

        // Map the key into an index within the set range
        public int Index( int key ) {
                return key % RANGE;
        }

        // Add a key/value entry to the index
        public void Add( int key, Object value ) {
                int ix = Index( key );

                // there is no entry at this index, add the key/value
                if ( index[ ix ] == null ) {
                        index[ ix ] = new Entry( key, value );
                }
                else {
                        // See if the key already exists in the chain
                        Entry next = index[ ix ];
                        for ( /**/; next != null; next = next.Next() ) {
                                // Entry found, update the value
                                if ( next.Compare( key ) ) {
                                        next.Value( value );
                                        break;
                                }
                        }

                        // no entry found, add one
                        if ( next == null ) {
```

```
                                // Add the entry to the front of the chain
                                Entry add = new Entry( key, value );
                                add.Next( index[ ix ] );
                                index[ ix ] = add;
                        }
                }
        }

        // Get the value for the key
        public Object Get( int key ) {
                int ix = Index( key );

                // See if the key exists in the chain at this entry in the index
                Entry next = index[ ix ];
                for ( /**/; next != null; next = next.Next() ) {
                        // Entry found, update the value
                        if ( next.Compare( key ) )
                                return next.Value();
                }

                // not found
                return null;
        }

        public static void main( String[] args ) {
                Hash index = new Hash( 100 );
                index.Add( 17,   100 );
                index.Add( 117,  600 ); // this will cause a collision
                index.Add( 228,  300 );
                index.Add( 675,  400 );
                index.Add( 2298, 500 );
                index.Add( 117,  200 ); // this will cause an update
                System.out.println( index.Get( 17 ) );
                System.out.println( index.Get( 117 ) );
                System.out.println( index.Get( 228 ) );
                System.out.println( index.Get( 675 ) );
                System.out.println( index.Get( 2298 ) );
        }
}
```

You maybe then asked to modify the algorithm where the key can be any object type (not just an integer). This means that you will need another function to convert

the key to an integer value, which then can be divided by the range to get the remainder (modulo). There is a builtin method for all objects in Java for this called hashCode(). This builtin method will digest the data in the object instance and return a 32 bit value. In Python, the equivalent is the builtin function hash().

Python

```
        ix = hash( key ) % RANGE
```

Java

```
        int ix = hashCode( key ) % RANGE;
```

Another variation you maybe asked to make is to create a hash table without linked lists and instead handle collisions with linear probing. In this case, we assume that the size of the table will be greater than or equal to the number of keys. When we have a collision, we simply place the entry at the next empty index after the collision. When searching, we go to the calculated index, and if the keys do not match, we probe downward in a linear fashion until we find the matching key or an empty entry (key is not in the hash table).

Algorithm: Hash Table with Linear Probing

1. Map each value into a fixed size integer range using a hash function.
2. If there is no entry at the index, add the key/value to the index.
3. If there is an entry and the key matches the entry (duplicate), then update the value.
4. If there is an entry and the key does not match the entry, then probe downward in a linear fashion.
    - If the entry is empty, add the key/value pair to the entry.
    - If a key matches (duplicate), then update the entry.
    - If the key does not match, continue to the next entry.

Python

```
# Definition for an Entry in the Hash Table (Dictionary)
class Entry:
        key = 0      # the key value
        value = None # the value for this key

        # Constructor
        def __init__( self, key,value ):
                self.key   = key
                self.value = value

        # Get the key
        def GetKey( self ):
                return self.key

        # Accessors for setting and getting the value of the entry
```

```
        def GetValue( self ):
                return self.value

        def Value( self, value ):
                self.value = value

        # Comparator for checking if key matches this entry.
        def Compare( self, key ):
                return ( self.key == key )

# Definition for a Linear Probe Hashing Algorithm
class HashLP:
        RANGE = 0    # the range of the index.
        index = []   # the index

        # constructor
        def __init__( self, range ):
                # set the index range and allocate the index
                self.RANGE = range
                self.index = [None] * self.RANGE

        # Map the key into an index within the set range
        def Index( self, key ):
                return key % self.RANGE

        # Add a key/value entry to the index
        def Add( self, key, value ):
                # Linear probe the entries for an empty or matching slot.
                for ix in range( self.Index( key ), self.RANGE ):
                        # there is no entry at this index, add the key/value
                        if self.index[ ix ] == None:
                                self.index[ ix ] = Entry( key, value )
                                break

                        # Entry found, update the value
                        if self.index[ ix ].Compare( key ):
                                self.index[ ix ].Value( value )
                                break;

        # Get the value for the key
```

```
        def Get( self, key ):
                ix = self.Index( key )

                # Linear probe the entries for an empty or matching slot.
                for ix in range( self.Index( key ), self.RANGE ):
                        # there is no entry at this index, return not found
                        if self.index[ ix ] == None:
                                return None

                        # Entry found
                        if self.index[ ix ].Compare( key ):
                                return self.index[ ix ].GetValue();

                # not found
                return None

index = HashLP( 100 )
index.Add( 17,   100 )
index.Add( 117,  600 ) # this will cause a collision
index.Add( 228,  300 )
index.Add( 675,  400 )
index.Add( 2298, 500 )
index.Add( 117,  200 ) # this will cause an update
print( index.Get( 17 ) )
print( index.Get( 117 ) )
print( index.Get( 228 ) )
print( index.Get( 675 ) )
print( index.Get( 2298 ) )
```

Java

```
// Definition for an Entry in the Hash Table (Dictionary)
class Entry {
        private int    key;                  // the key value
        private Object value = null; // the value for this key

        // Constructor
        public Entry( int key, Object value ) {
                this.key   = key;
                this.value = value;
        }
```

```
        // Get the key
        public int Key() {
                return key;
        }

        // Accessors for setting and getting the value of the entry
        public Object Value() {
                return value;
        }

        public void Value( Object value ) {
                this.value = value;
        }

        // Comparator for checking if key matches this entry.
        public boolean Compare( int key ) {
                return ( this.key == key );
        }
}

// Definition for a Linear Probe based Hashing Algorithm
public class HashLP {
        private final int RANGE;       // the range of the index.
        private Entry[] index = null;  // the index

        // constructor
        public HashLP( int range ) {
                // set the index range and allocate the index
                RANGE = range;
                index = new Entry[ RANGE ];
        }

        // Map the key into an index within the set range
        public int Index( int key ) {
                return key % RANGE;
        }

        // Add a key/value entry to the index
        public void Add( int key, Object value ) {
```

```
                int ix = Index( key );

                // Linear probe the entries for an empty or matching slot.
                for ( /**/; ix < RANGE; ix++ ) {
                        // there is no entry at this index, add the key/value
                        if ( index[ ix ] == null ) {
                                index[ ix ] = new Entry( key, value );
                                break;
                        }

                        // Entry found, update the value
                        if ( index[ ix ].Compare( key ) ) {
                                index[ ix ].Value( value );
                                break;
                        }
                }
        }

        // Get the value for the key
        public Object Get( int key ) {
                int ix = Index( key );

                // Linear probe the entries for an empty or matching slot.
                for ( /**/; ix < RANGE; ix++ ) {
                        // there is no entry at this index, return not found
                        if ( index[ ix ] == null )
                                return null;

                        // Entry found
                        if ( index[ ix ].Compare( key ) )
                                return index[ ix ].Value();
                }

                // not found
                return null;
        }

        public static void main( String[] args ) {
                HashLP index = new HashLP( 100 );
                index.Add( 17,   100 );
```

```
                index.Add( 117,  600 ); // this will cause a collision
                index.Add( 228,  300 );
                index.Add( 675,  400 );
                index.Add( 2298, 500 );
                index.Add( 117,  200 ); // this will cause an update
                System.out.println( index.Get( 17 ) );
                System.out.println( index.Get( 117 ) );
                System.out.println( index.Get( 228 ) );
                System.out.println( index.Get( 675 ) );
                System.out.println( index.Get( 2298 ) );
        }
}
```

14. String Manipulation

The most common string manipulation question that you maybe asked is to reverse the order of a string (without using a builtin library). You maybe asked to do both an iterative and a recursive solution.

Algorithm - Reverse String (Iterative)

1. Create a StringBuffer (or character array) to hold the reversed string.
2. Starting at the end of the string and moving backwards, append each character to the reversed string.

Python

```
# Reverse the order of a string using Iterative Solution
def Reverse( original ):
        # Create a second string to hold/return the reversed string
        length = len( original )
        reversed = ""

        # copy over the string in reverse order
        for i in range( length-1, -1, -1 ):
                reversed += original[ i ]

        return reversed

print( Reverse( "abcdefg" ) )
```

Java

```
// Definition for String Manipulation Routines
```

```
public class Strings {
        // Reverse the order of a string using Iterative Solution
        public static String Reverse( String original ) {
                // Create a second string to hold/return the reversed string
                int length = original.length();
                StringBuffer reversed = new StringBuffer( length );

                // copy over the string in reverse order
                for ( int i = length-1; i >= 0; i-- ) {
                        reversed.append( original.charAt( i ) );
                }

                return reversed.toString();
        }

        public static void main( String[] args ) {
                System.out.println( Reverse( "abcdefg" ) );

        }
}
```

Algorithm - Revere String (Recursive)

1. If the original string is one character, if so then return the string.
2. Otherwise, break the string into two parts: the first character and the remainder of the string.
3. Make a recursive call with the remaining string and append the first character to the return from the call.

Python

```
# Reverse the order of a string (Recursion)
def ReverseR( original ):
        if len( original ) > 1:
                return ReverseR( original[1:] ) + original[ 0 ]
        return original

print( ReverseR( "abcdefg" ) )
```

Java

```
// Definition for String Manipulation Routines
public class Strings {
        ...
```

```
        // Reverse the order of a string using Recursion
        public static String ReverseR( String original ) {
                if ( original.length() > 1 )
                        return ReverseR( original.substring( 1 ) ) + original.charAt( 0 );
                return original;
        }

        public static void main( String[] args ) {
                System.out.println( ReverseR( "abcdefg" ) );

        }
}
```

Another algorithm you maybe asked that is similar to reversing a string is to determine if a string is an palindrome. A palindrome is the same word/phrase spelled forward or backwards, such as: madam, civic, racecar, ... A simple solution would be to reverse the string and compare the reversed string to the original. But, you are likely to be asked to solve the algorithm without making a copy of the string, for single word palindromes.

Algorithm - Palindrome

1. Iterate through the first half of the string. When the string is an odd number of characters, skip the middle character.
2. On each iteration, compare the current character at the front of the string to the same position but from the rear of the string.
3. If they do not much, it is not a palindrome.

Python

```
# Determine if a string is a Palindrome
def Palindrome( s ):
        length = len( s )
        for i in range( 0, int( length / 2 ) ):
                if s[ i ] != s[ length - i - 1 ]:
                        return False
        return True

print( Palindrome( "noon" ) )
print( Palindrome( "rotator" ) )
print( Palindrome( "notone" ) )
```

Java

```
// Definition for String Manipulation Routines
public class Strings {
        ...
```

```
        // Determine if a string is a Palindrome
        public static boolean Palindrome( String s ) {
                int length = s.length();
                for ( int i = 0; i < length / 2; i++ ) {
                        if ( s.charAt( i ) != s.charAt( length - i - 1 ) )
                                return false;
                }
                return true;
        }

        public static void main( String[] args ) {
                System.out.println( Palindrome( "noon" ) );
                System.out.println( Palindrome( "rotator" ) );
                System.out.println( Palindrome( "notone" ) );

        }
}
```

Once you've solved the algorithm for single word palindromes, you maybe asked to now solve the same for phrases, regardless of the positioning of spaces, punctuation and capitalizing, such as: My gym. Note, you cannot solve by making a first pass thru the string to remove spaces, punctuation and lowercase, since strings are immutable; hence, if you modify the original string, you end up making a copy.

Algorithm - Palindrome (Phrases)

1. Iterate simultaneously from the front (forward) and back (backward) of the string, until the two iterators meet.
2. If current character of either iterator is a punctuation or space character, then advance the corresponding iterator.
3. Otherwise, if the lowercase value of the current character from both iterators do not equal, then it is not a palindrome.
4. Advance both iterators and repeat.

Python

```
# Determine if a string is a Palindrome for Phrases
def PalindromeP( s ):
        length = len( s )
        i = 0
        j = length - 1
        while i < j:
                if isPunctOrSpace( s[ i ] ):
                        i += 1
                        continue

                if isPunctOrSpace( s[ j ] ):
```

```
                        j -= 1
                        continue

                if s[ i ].lower() != s[ j ].lower():
                        return False
                i += 1
                if j != i:
                        j -= 1

        return True

def isPunctOrSpace(c):
        val = ord(c)
        return  ( val >= 32 and val <= 47 ) or ( val >= 58 and val <= 64 ) or ( val >= 91 and val <= 96 ) or ( val >= 123 and val <= 126 )

print( PalindromeP( "my gym" ) )
print( PalindromeP( "Red rum, sir, is murder" ) )
```

Java

```
// Definition for String Manipulation Routines
public class Strings {
        ...
        // Determine if a string is a Palindrome for Phrases
        public static boolean PalindromeP( String s ) {
                int length = s.length();
                for ( int i = 0, j = length - 1; i < j; /**/ ) {
                        if ( isPunctOrSpace( s.charAt( i ) ) )  {
                                i++;
                                continue;
                        }
                        if ( isPunctOrSpace( s.charAt( j ) ) ) {
                                j--;
                                continue;
                        }

                        if ( Character.toLowerCase( s.charAt( i ) ) != Character.toLowerCase( s.charAt( j ) ) )
                                return false;
                        i++;
                        if ( j != i ) j--;
                }
```

```
                return true;
        }

        public static boolean isPunctOrSpace(final char c) {
                final int val = (int)c;
                return  ( val >= 32 && val <= 47 ) ||
                                ( val >= 58 && val <= 64 ) ||
                                ( val >= 91 && val <= 96 ) ||
                                ( val >= 123 && val <= 126 );
        }

        public static void main( String[] args ) {
                System.out.println( PalindromeP( "my gym" ) );
                System.out.println( PalindromeP( "Red rum, sir, is murder" ) );

        }
}
```

A simple algorithm you maybe ask is to count all occurrences of every character in a string.

Algorithm - Character Count

1. Create a counter for the 96 ASCII printable characters.
2. Iterate through the string, incrementing the character specific index in the counter for each character.

Python

```
# Count the number of occurrences of a character in a string
def CharOccur( s ):
        counter = [0] * 96      # codes 32 .. 127 are printable (so skip first 32)
        length = len(s)
        # use counter as an accumulator while we count each character in string
        for i in range( 0, length):
                counter[ ord( s[ i ] ) - 32 ] += 1      # offset ascii code by 32
        return counter

res = CharOccur( "jack and jill jumped over the hill to fetch a pale of water")
for i in range(32, 128):
        if res[ i - 32 ] > 0:
                print( chr( i ) + ": " + str( res[ i - 32 ] ) )
```

Java

```
// Definition for String Manipulation Routines
public class Strings {
        ...
        // Count the number of occurrences of a character in a string
        public static short[] CharOccur( String s ) {
                short[] counter = new short[96];        // codes 32 .. 127 are printable (so skip first 32)
                int length = s.length();
                // use counter as an accumulator while we count each character in string
                for ( int i = 0; i < length; i++ )
                        counter[ (int) s.charAt( i ) - 32 ]++;  // offset ascii code by 32
                return counter;
        }

        public static void main( String[] args ) {
                short[] res = CharOccur( "jack and jill jumped over the hill to fetch a pale of water");
                for ( int i = 32; i < 128; i++ )
                        if ( res[ i - 32 ] > 0 ) System.out.println( (char) i + ": " + res[ i - 32 ] );
        }
}
```

You maybe asked to make a variation of this by counting only duplicated characters (i.e., at least two occurrences).

Algorithm - Duplicate Chars

1. Make a pass over the string to generate a character count index.
2. Make a pass over the character count index using a bitwise mask to mask out single character occurrences (leaving only duplicates with non-zero values).

Python

```
# Count all duplicated characters
def DupChar( s ):
        # Get the character occurrences
        dup = CharOccur( s )
        # Mask out all single count occurrences
        length = len( dup )
        for i in range( 0, length):
                dup[ i ] &= ~0x001;
        return dup

res = DupChar( "jack and jill jumped over the hill to fetch a pale of water")
for i in range(32, 128):
```

```
        if res[ i - 32 ] > 0:
                print( chr( i ) + ": " + str( res[ i - 32 ] ) )
```

Java

```
// Definition for String Manipulation Routines
public class Strings {
        ...
        // Count all duplicated characters
        public static short[] DupChar( String s ) {
                // Get the character occurrences
                short[] dup = CharOccur( s );
                // Mask out all single count occurrences
                for ( int i = 0; i < dup.length; i++ )
                        dup[ i ] &= ~0x001;
                return dup;
        }

        public static void main( String[] args ) {
                short[] res = DupChar( "jack and jill jumped over the hill to fetch a pale of water");
                for ( int i = 32; i < 128; i++ )
                        if ( res[ i - 32 ] > 0 ) System.out.println( (char) i + ": " + res[ i - 32 ] );
        }
}
```

15. Graphs

You maybe asked to implement algorithms related to traversing graphs. The basic algorithms are those for traversing (visiting) all nodes in an undirected graph, where in an undirected graph an edge between nodes can be traversed in either direction.

Traversing an undirected tree can be done with either bread-first search (BFS) or depth-first search (DFS) methods. Generally, the algorithm will implement a tricolor node marking to indicate:

- White : node has never been visited.
- Gray : node is waiting to be visited (known as the frontier).
- Black: node has been visited.

Below are examples for a breadth first search, followed by a iterative and recursive solutions for depth first search.

Algorithm - BFS Graph Traversal

1. Create a queue for the frontier list (waiting to be visited) and add the root (starting) node to the queue.
2. While there are nodes on the frontier waiting to be visited, pop the next node from the queue, and marked it as visited.

3. For each of the node's neighbors that are unvisited (not marked as visited or on frontier), add the node to the frontier and mark as frontier.
4. Note that since the frontier is a queue (FIFO), the nodes will be popped in a breadth first manner.

Python

```
# Definition for a node in a graph
class Node:
        # Visit State
        UNVISITED = 0  # not visited or on frontier (while)
        FRONTIER  = 1  # on frontier, waiting to be visited (gray)
        VISITED   = 2  # visited (black)

        key = None                  # node data
        neighbors = DynamicArray()      # list of neighboring nodes (vertices)
        visited = UNVISITED         # node has been visited

        # constructor
        def __init__( self, key ):
                self.key = key

        # Get the Key
        def GetKey( self ):
                return self.key

        # Add a neighbor to this node
        def Neighbor( self, neighbor ):
                # list the node as a neighbor of this node
                self.neighbors.Add( neighbor )

                # ist this node as a neighbor of the node
                neighbor.neighbors.Add( self )

        # Get the nodes neighbors
        def GetNeighbors( self ):
                return self.neighbors

        # Set the node visit
        def Visited( self, visited ):
                self.visited = visited

        # Get whether the node has been visited
```

```
        def GetVisited( self ):
                return self.visited

# Breadth first search of a graph
def BFS( root ):
        # Create a queue for frontier nodes (nodes to be visited next)
        frontier = DynamicArray()

        # Add the root node to the frontier
        frontier.Add( root )
        root.Visited( root.FRONTIER )

        # Traverse each node in the frontier list
        while frontier.Size() != 0:
                # visit the next node
                visit = frontier.Get( 0 )
                frontier.Delete( 0 )
                visit.Visited( visit.VISITED )
                print( visit.GetKey() )

                # Add the node's neighbors (if not visited and not on frontier list)
                # to the frontier list
                nodes = visit.GetNeighbors()
                for i in range( 0, nodes.Size() ):
                        node = nodes.Get( i )
                        if node.GetVisited() == node.UNVISITED:
                                frontier.Add( node )
                                node.Visited( node.FRONTIER )

nodes = [None] * 6;
for i in range( 0, 6 ):
        nodes[ i ] = Node( i )
nodes[ 0 ].Neighbor( nodes[ 1 ] )
nodes[ 0 ].Neighbor( nodes[ 2 ] )
nodes[ 0 ].Neighbor( nodes[ 3 ] )
nodes[ 1 ].Neighbor( nodes[ 2 ] )
nodes[ 1 ].Neighbor( nodes[ 3 ] )
nodes[ 3 ].Neighbor( nodes[ 4 ] )
nodes[ 3 ].Neighbor( nodes[ 5 ] )
nodes[ 4 ].Neighbor( nodes[ 5 ] )
```

```
BFS( nodes[ 0 ] )
```

Java

```
import java.util.LinkedList;

// Definition for a node in a graph
class Node {
        private Object key;                         // node data
        // list of neighboring nodes (vertices)
        private LinkedList neighbors = new LinkedList();
        private Visit visited = Visit.UNVISITED;  // node has been visited

        // Visit State
        public enum Visit {
                UNVISITED,      // not visited or on frontier (while)
                FRONTIER,       // on frontier, waiting to be visited (gray)
                VISITED         // visited (black)
        }

        // constructor
        public Node( Object key ) {
                this.key = key;
        }

        // Get the Key
        public Object Key() {
                return this.key;
        }

        // Add a neighbor to this node
        public void Neighbor( Node neighbor ) {
                // list the node as a neighbor of this node
                neighbors.add( neighbor );

                // list this node as a neighbor of the node
                neighbor.neighbors.add( this );
        }

        // Get the nodes neighbors
        public LinkedList Neighbors() {
```

```
                return neighbors;
        }

        // Set the node visit
        public void Visited( Visit visited ) {
                this.visited = visited;
        }

        // Get whether the node has been visited
        public Visit Visited() {
                return visited;
        }
}

// Definition for a Graph
public class Graph {
        // Breadth first search of a graph
        public static void BFS( Node root ) {
                // Create a queue for frontier nodes (nodes to be visited next)
                LinkedList frontier = new LinkedList();

                // Add the root node to the frontier
                frontier.add( root );
                root.Visited( Node.Visit.FRONTIER );

                // Traverse each node in the frontier list
                while ( frontier.size() != 0 ) {
                        // visit the next node
                        Node visit = frontier.remove();
                        visit.Visited( Node.Visit.VISITED );
                        System.out.println( visit.Key() );

                        // Add the node's neighbors (if not visited and not on frontier list)
                        // to the frontier list
                        for ( Node node : visit.Neighbors() ) {
                                if ( node.Visited() == Node.Visit.UNVISITED ) {
                                        frontier.add( node );
                                        node.Visited( Node.Visit.FRONTIER );
                                }
                        }
```

```
                }
        }

        public static void main(String[] args ) {
                Node nodes[] = new Node[ 6 ];
                for ( int i = 0; i < 6; i++ )
                        nodes[ i ] = new Node( i );
                nodes[ 0 ].Neighbor( nodes[ 1 ] );
                nodes[ 0 ].Neighbor( nodes[ 2 ] );
                nodes[ 0 ].Neighbor( nodes[ 3 ] );
                nodes[ 1 ].Neighbor( nodes[ 2 ] );
                nodes[ 1 ].Neighbor( nodes[ 3 ] );
                nodes[ 3 ].Neighbor( nodes[ 4 ] );
                nodes[ 3 ].Neighbor( nodes[ 5 ] );
                nodes[ 4 ].Neighbor( nodes[ 5 ] );
                BFS( nodes[ 0 ] );
        }
}
```

Algorithm - DFS Graph Traversal (Iterative)

1. Create a stack for the frontier list (waiting to be visited) and add the root (starting) node to the stack.
2. While there are nodes on the frontier waiting to be visited, pop the next node from the stack, and marked it as visited.
3. For each of the node's neighbors that are unvisited (not marked as visited or on frontier), add the node to the frontier and mark as frontier.
4. Note that since the frontier is a stack (LIFO), the nodes will be popped in a depth first manner.

Python

```
# Depth first search of a graph (Iterative)
def DFS( root ):
        # Create a stack for frontier nodes (nodes to be visited next)
        frontier = Stack();

        # Add the root node to the frontier
        frontier.Push( root )
        root.Visited( root.FRONTIER )

        # Traverse each node in the frontier list
        while frontier.Empty() == False:
                # visit the next node
                visit = frontier.Pop()
```

```
                visit.Visited( visit.VISITED );
                print( visit.GetKey() )

                # Add the node's neighbors (if not visited and not on frontier list)
                # to the frontier list
                nodes = visit.GetNeighbors()
                for i in range( 0, nodes.Size() ):
                        node = nodes.Get( i )
                        if node.GetVisited() == node.UNVISITED:
                                frontier.Push( node )
                                node.Visited( node.FRONTIER )

nodes = [None] * 6;
for i in range( 0, 6 ):
        nodes[ i ] = Node( i )
nodes[ 0 ].Neighbor( nodes[ 1 ] )
nodes[ 0 ].Neighbor( nodes[ 2 ] )
nodes[ 0 ].Neighbor( nodes[ 3 ] )
nodes[ 1 ].Neighbor( nodes[ 2 ] )
nodes[ 1 ].Neighbor( nodes[ 3 ] )
nodes[ 3 ].Neighbor( nodes[ 4 ] )
nodes[ 3 ].Neighbor( nodes[ 5 ] )
nodes[ 4 ].Neighbor( nodes[ 5 ] )
DFS( nodes[ 0 ] )
```

Java

```
import java.util.Stack;

// Definition for a Graph
public class Graph {
        ...
        // Depth first search of a graph
        public static void DFS( Node root ) {
                // Create a stack for frontier nodes (nodes to be visited next)
                Stack frontier = new Stack();

                // Add the root node to the frontier
                frontier.push( root );
                root.Visited( Node.Visit.FRONTIER );
```

```
                // Traverse each node in the frontier list
                while ( !frontier.empty() ) {
                        // visit the next node
                        Node visit = (Node) frontier.pop();
                        visit.Visited( Node.Visit.VISITED );
                        System.out.println( visit.Key() );

                        // Add the node's neighbors (if not visited and not on frontier list)
                        // to the frontier list
                        for ( Node node : visit.Neighbors() ) {
                                if ( node.Visited() == Node.Visit.UNVISITED ) {
                                        frontier.push( node );
                                        node.Visited( Node.Visit.FRONTIER );
                                }
                        }
                }
        }

        public static void main(String[] args ) {
                Node nodes[] = new Node[ 6 ];
                for ( int i = 0; i < 6; i++ )
                        nodes[ i ] = new Node( i );
                nodes[ 0 ].Neighbor( nodes[ 1 ] );
                nodes[ 0 ].Neighbor( nodes[ 2 ] );
                nodes[ 0 ].Neighbor( nodes[ 3 ] );
                nodes[ 1 ].Neighbor( nodes[ 2 ] );
                nodes[ 1 ].Neighbor( nodes[ 3 ] );
                nodes[ 3 ].Neighbor( nodes[ 4 ] );
                nodes[ 3 ].Neighbor( nodes[ 5 ] );
                nodes[ 4 ].Neighbor( nodes[ 5 ] );
                DFS( nodes[ 0 ] );
        }
}
```

Algorithm - DFS Graph Traversal (Recursive)

1. If the node is null, then stop.
2. Visit the node.
3. For each of the node's neighbors that are unvisited (not marked as visited or on frontier), recursively call the method.

Python

```
# Depth first search of a graph (Recursive)
def DFSR( root ):
        if root == None:
                return

        # mark the node as visited
        root.Visited( root.VISITED )
        print( root.GetKey() )

        # Recursively visit each unvisited neighbor
        nodes = root.GetNeighbors()
        for i in range( 0, nodes.Size() ):
                node = nodes.Get( i )
                if node.GetVisited() == node.UNVISITED:
                        DFSR( node )

nodes = [None] * 6;
for i in range( 0, 6 ):
        nodes[ i ] = Node( i )
nodes[ 0 ].Neighbor( nodes[ 1 ] )
nodes[ 0 ].Neighbor( nodes[ 2 ] )
nodes[ 0 ].Neighbor( nodes[ 3 ] )
nodes[ 1 ].Neighbor( nodes[ 2 ] )
nodes[ 1 ].Neighbor( nodes[ 3 ] )
nodes[ 3 ].Neighbor( nodes[ 4 ] )
nodes[ 3 ].Neighbor( nodes[ 5 ] )
nodes[ 4 ].Neighbor( nodes[ 5 ] )
DFSR( nodes[ 0 ] )
```

Java

```
import java.util.Stack;

// Definition for a Graph
public class Graph {
        ...
        // Depth first search of a graph (Recursive)
        public static void DFSR( Node root ) {
                if ( root == null )
                        return;
```

```
                // mark the node as visited
                root.Visited( Node.Visit.VISITED );
                System.out.println( root.Key() );

                // Recursively visit each unvisited neighbor
                for ( Node node : root.Neighbors() ) {
                        if ( node.Visited() == Node.Visit.UNVISITED ) {
                                DFSR( node );
                        }
                }
        }

        public static void main(String[] args ) {
                Node nodes[] = new Node[ 6 ];
                for ( int i = 0; i < 6; i++ )
                        nodes[ i ] = new Node( i );
                nodes[ 0 ].Neighbor( nodes[ 1 ] );
                nodes[ 0 ].Neighbor( nodes[ 2 ] );
                nodes[ 0 ].Neighbor( nodes[ 3 ] );
                nodes[ 1 ].Neighbor( nodes[ 2 ] );
                nodes[ 1 ].Neighbor( nodes[ 3 ] );
                nodes[ 3 ].Neighbor( nodes[ 4 ] );
                nodes[ 3 ].Neighbor( nodes[ 5 ] );
                nodes[ 4 ].Neighbor( nodes[ 5 ] );
                DFSR( nodes[ 0 ] );
        }
}
```

Best Regards,
Andrew Ferlitsch
Portland Data Science Group, Co-Organizer