Universidad del Valle de Guatemala 2026
Algoritmos y Estructura de Datos – Proyecto 01_Fase1
Pavel Jezrael Huerta Ajquichi

Intérprete de Bitcoin Script

1. Fundamentos de Bitcoin Script

¿Qué es Bitcoin Script?

Bitcoin Script es un lenguaje de programación stack-based, basado en pila, diseñado específicamente para validar transacciones en la blockchain de Bitcoin. A diferencia de otros lenguajes de programación, Bitcoin Script es intencionalmente no Turing-completo, lo que significa que no puede ejecutar bucles infinitos ni realizar computaciones arbitrariamente complejas.

Características principales:

- **Stack-based:** Todas las operaciones utilizan una pila como estructura de datos principal
- **Forth-like:** Sintaxis similar al lenguaje Forth
- **No Turing-completo:** Sin bucles ni recursión, garantiza terminación
- **Orientado a validación:** Diseñado específicamente para verificar condiciones de gasto

Arquitectura de la Pila

La pila “stack” es una estructura de datos LIFO “(Last In, First Out)” que almacena arrays de bytes. Cada elemento en la pila puede representar:

- **Números:** Codificados en formato little-endian
- **Booleanos:** Array vacío = false, cualquier otro valor = true
- **Datos binarios:** Hashes, firmas, claves públicas

Tipos de Scripts

scriptPubKey (Script de Bloqueo): Define las condiciones que deben cumplirse para gastar los fondos. Se incluye en la salida de la transacción (output). Ejemplo: 'Solo puede gastar quien tenga la clave privada correspondiente a este hash de clave pública.'

scriptSig (Script de Desbloqueo): Proporciona los datos necesarios para satisfacer las condiciones del scriptPubKey. Se incluye en la entrada de la transacción (input). Ejemplo: 'Aquí está mi firma y mi clave pública.

Universidad del Valle de Guatemala 2026
Algoritmos y Estructura de Datos – Proyecto 01_Fase1
Pavel Jezrael Huerta Ajquichi

2. Flujo de Validación de Transacciones

Proceso de Validación

La validación de una transacción Bitcoin sigue un proceso específico que combina el scriptSig de la entrada con el scriptPubKey de la salida que se está gastando:

- **Paso 1:** Concatenación de Scripts, Se ejecuta primero el scriptSig, luego el scriptPubKey en la misma pila.
- **Paso 2:** Ejecución Secuencial, Los opcodes se procesan de izquierda a derecha, uno a la vez.
- **Paso 3:** Validación Final, La transacción es válida si y solo si:
    - Ninguna instrucción falló durante la ejecución
    - La pila no está vacía al finalizar
    - El elemento en la cima de la pila es verdadero (diferente de cero)

Ejemplo P2PKH (Pay-to-Public-Key-Hash)

P2PKH es el tipo de transacción más común en Bitcoin. Requiere que el receptor demuestre posesión de la clave privada correspondiente a un hash de clave pública específico.

ScriptSig: <firma> <clavePública>

ScriptPubKey: OP_DUP OP_HASH160 <hashClavePública> OP_EQUALVERIFY OP_CHECKSIG

Secuencia de ejecución paso a paso:

1. <firma>: Empuja la firma a la pila → Stack: [firma]

2. <clavePública>: Empuja la clave pública → Stack: [firma, pubKey]

3. OP_DUP: Duplica pubKey → Stack: [firma, pubKey, pubKey]

4. OP_HASH160: Hashea la pubKey → Stack: [firma, pubKey, hash(pubKey)]

5. <hashClavePública>: Empuja el hash esperado → Stack: [firma, pubKey, hash(pubKey), hashEsperado]

6. OP_EQUALVERIFY: Verifica igualdad y elimina ambos → Stack: [firma, pubKey] (falla si diferentes)

7. OP_CHECKSIG: Verifica firma con pubKey → Stack: [1] (si válida)

Universidad del Valle de Guatemala 2026
Algoritmos y Estructura de Datos – Proyecto 01_Fase1
Pavel Jezrael Huerta Ajquichi

3. Java Collections Framework (JCF)

Stack<byte[]> - Pila Principal

Decisión de diseño: java.util.Stack<byte[]> fue seleccionada como la estructura principal para implementar la pila de ejecución de Bitcoin Script. Justificación técnica:

1. Semántica clara: La clase Stack proporciona una API específica para operaciones LIFO con métodos como push(), pop() y peek() que reflejan directamente la naturaleza del lenguaje Bitcoin Script.

2. Complejidad temporal óptima:

- push(element): O(1) amortizado
- pop(): O(1)
- peek(): O(1)
- isEmpty(): O(1)

3. Complejidad espacial: O(n) donde n es el número de elementos en la pila, con overhead mínimo por ser implementada sobre Vector.

Alternativas consideradas:

Estructura	Ventajas	Desventajas	Decisión
ArrayList	Acceso aleatorio O(1), menor overhead	API no semántica para pila, requiere get(size-1)	✗
LinkedList	Implementa Deque, flexible	Mayor overhead de memoria por nodos	✗
ArrayDeque	Más eficiente que Stack, sin sincronización	API genérica (addFirst/removeLast)	△
Stack	API específica, semántica clara, thread-safe	Clase legacy, sincronización innecesaria	✓

ArrayList<String> - Tokenización

Para la tokenización y almacenamiento de instrucciones del script, se utilizó ArrayList<String> debido a:

- **Tamaño variable desconocido:** El número de tokens en un script varía
- **Acceso secuencial:** Los tokens se procesan en orden, O(1) por índice
- **No requiere inserción/eliminación en medio:** Solo lectura secuencial
- **Eficiencia espacial:** O(m) donde m = número de tokens

Universidad del Valle de Guatemala 2026
Algoritmos y Estructura de Datos – Proyecto 01_Fase1
Pavel Jezrael Huerta Ajquichi

4. Análisis de Complejidad Algorítmica

Complejidad Temporal por Opcode:

Opcode	Complejidad	Justificación
OP_DUP	O(n)	Arrays.copyOf copia n bytes
OP_DROP	O(1)	Stack.pop() es O(1)
OP_EQUAL	O(n)	Arrays.equals compara n bytes
OP_HASH160	O(n)	SHA-256 procesa n bytes de entrada
OP_CHECKSIG	O(1)*	*Mock simplificado, real sería O(n)

Complejidad del Sistema Completo

Tokenización: O(k) donde k = longitud del string del script

Ejecución: O(m × n) en el peor caso, donde:

- **m** = número de opcodes
- **n** = tamaño promedio de datos en pila

Espacio: O(p × n) donde:

- **p** = profundidad máxima de la pila
- **n** = tamaño promedio de elementos