ÁRBOLES DE EXPRESIONES

Estructuras de Datos para Representación de Expresiones Matemáticas

---

Asignatura: Estructuras de Datos y Algoritmos
Tema: Árboles de Expresiones
Nivel: Universitario - Pregrado

---

1. INTRODUCCIÓN

Los árboles de expresiones constituyen una estructura de datos fundamental en ciencias de la computación, ampliamente utilizada en el diseño de compiladores, interpretadores, calculadoras científicas y sistemas de álgebra computacional. Esta estructura permite representar expresiones matemáticas de manera jerárquica, facilitando su evaluación, manipulación y optimización.

1.1 Definición

Un **árbol de expresión** es un árbol binario donde:

- Los **nodos internos** representan operadores matemáticos (+, -, *, /)
- Los **nodos hoja** (hojas del árbol) representan operandos (números, variables o constantes)
- La **estructura jerárquica** del árbol refleja la precedencia de operadores y el orden de evaluación

1.2 Importancia y Aplicaciones

Los árboles de expresiones son esenciales en:

- **Compiladores**: Para el análisis sintáctico y generación de código intermedio
- **Interpretadores**: Para evaluar expresiones en tiempo de ejecución
- **Calculadoras**: Para procesar expresiones complejas respetando precedencia
- **Sistemas de álgebra computacional**: Para simplificación y transformación de expresiones
- **Bases de datos**: En la optimización de consultas SQL

---

2. CONCEPTOS FUNDAMENTALES

2.1 Estructura del Árbol

Un árbol de expresión se compone de nodos que pueden ser de dos tipos:

Nodo Operador (Nodo Interno)

- Contiene un operador matemático: +, -, *, /
- Tiene exactamente dos hijos (subárbol izquierdo y derecho)
- Representa una operación a realizar

Nodo Operando (Nodo Hoja)

- Contiene un valor numérico o variable
- No tiene hijos
- Representa un dato concreto

2.2 Precedencia de Operadores

La precedencia determina el orden de evaluación cuando no hay paréntesis:

1. **Alta precedencia**: Multiplicación (*) y División (/)
2. **Baja precedencia**: Suma (+) y Resta (-)

Regla importante: Los operadores de mayor precedencia se ubican más cerca de las hojas del árbol.

2.3 Asociatividad

Todos los operadores básicos (+, -, *, /) son **asociativos por la izquierda**, es decir, se evalúan de izquierda a derecha cuando tienen la misma precedencia.

Ejemplo: `8 - 3 - 2` se interpreta como `(8 - 3) - 2 = 3`

---

3. CONSTRUCCIÓN DE ÁRBOLES DE EXPRESIONES

3.1 Proceso General

Para construir un árbol de expresión a partir de una expresión infija (notación estándar):

1. Identificar el operador de menor precedencia que se evaluará último
2. Este operador se convierte en la raíz del árbol
3. La expresión a su izquierda forma el subárbol izquierdo
4. La expresión a su derecha forma el subárbol derecho
5. Aplicar recursivamente el proceso a cada subárbol

3.2 Ejemplos Básicos

Ejemplo 1: Expresión Simple

Expresión: `3 + 5`

Árbol resultante:

```
  +
 / \
3   5
```

Análisis:

- Operador `+` es la raíz
- Operando `3` es el hijo izquierdo
- Operando `5` es el hijo derecho
- Evaluación: 3 + 5 = 8

Ejemplo 2: Multiplicación Simple

Expresión: `4 * 7`

Árbol resultante:

Evaluación: 4 * 7 = 28

3.3 Expresiones con Precedencia

Ejemplo 3: Suma y Multiplicación

Expresión: `2 + 3 * 4`

Árbol resultante:

Análisis:

- El operador + tiene menor precedencia, por lo que es la raíz
- El operador * tiene mayor precedencia, se evalúa primero
- Orden de evaluación:
    1. Primero: 3 * 4 = 12
    2. Segundo: 2 + 12 = 14

Ejemplo 4: Multiplicación y Suma (orden inverso)

Expresión: `3 * 4 + 2`

Árbol resultante:

Análisis:

- Aunque `*` aparece primero, `+` es la raíz por su menor precedencia
- Evaluación:
    1. Primero: 3 * 4 = 12
    2. Segundo: 12 + 2 = 14

Ejemplo 5: División y Resta

Expresión: `10 - 6 / 2`

Árbol resultante:

Evaluación:

1. Primero: 6 / 2 = 3
2. Segundo: 10 - 3 = 7

3.4 Expresiones con Paréntesis

Los paréntesis modifican el orden de evaluación natural, forzando que ciertas operaciones se realicen primero.

Ejemplo 6: Paréntesis que Alteran la Precedencia

Expresión: `(2 + 3) * 4`

Árbol resultante:

Análisis:

- Los paréntesis fuerzan que + se evalúe primero
- Ahora + está más cerca de las hojas que *
- Evaluación:
    1. Primero: 2 + 3 = 5
    2. Segundo: 5 * 4 = 20

Comparación sin paréntesis: `2 + 3 * 4 = 2 + 12 = 14`

Ejemplo 7: Múltiples Operaciones con Paréntesis

Expresión: `(8 - 2) * (3 + 1)`

Árbol resultante:

Evaluación:

1. Subárbol izquierdo: 8 - 2 = 6
2. Subárbol derecho: 3 + 1 = 4
3. Raíz: 6 * 4 = 24

Ejemplo 8: Paréntesis Anidados

Expresión: `((4 + 2) * 3) - 5`

Árbol resultante:

Evaluación (desde las hojas hacia la raíz):

1. Nivel más profundo: 4 + 2 = 6
2. Siguiente nivel: 6 * 3 = 18
3. Raíz: 18 - 5 = 13

3.5 Expresiones Complejas

Ejemplo 9: Expresión con Cuatro Operaciones

Expresión: `5 + 3 * 2 - 8 / 4`

Árbol resultante:

Análisis:

- Operadores de alta precedencia (`*` y `/`) están más cerca de las hojas
- Operadores de baja precedencia (`+` y `-`) están más cerca de la raíz
- Por asociatividad, `-` es la raíz final

Evaluación paso a paso:

1. 3 * 2 = 6
2. 8 / 4 = 2
3. 5 + 6 = 11
4. 11 - 2 = 9

Ejemplo 10: Expresión Compleja con Paréntesis

Expresión: `(10 + 5) / (3 - 1) * 2`

Árbol resultante:

Evaluación:

1. 10 + 5 = 15
2. 3 - 1 = 2
3. 15 / 2 = 7.5
4. 7.5 * 2 = 15

---

4. RECORRIDOS DE ÁRBOLES DE EXPRESIONES

Los árboles de expresiones pueden recorrerse de diferentes maneras, generando distintas notaciones:

4.1 Recorrido Inorden (Infijo)

Visita: Izquierda → Raíz → Derecha

Genera la **notación infija** (notación estándar matemática).

Ejemplo para el árbol de `2 + 3 * 4`:

Recorrido: 2 + 3 * 4

Nota: Requiere agregar paréntesis para preservar la precedencia correcta.

4.2 Recorrido Preorden (Prefijo)

Visita: Raíz → Izquierda → Derecha

Genera la **notación prefija** (notación polaca).

Ejemplo para el árbol de `2 + 3 * 4`:

```
Recorrido: + 2 * 3 4
```

4.3 Recorrido Postorden (Sufijo)

Visita: Izquierda → Derecha → Raíz

Genera la **notación postfija** (notación polaca inversa).

Ejemplo para el árbol de `2 + 3 * 4`:

```
Recorrido: 2 3 4 * +
```

Esta notación es especialmente útil para evaluar expresiones sin necesidad de paréntesis ni conocer precedencia.

---

5. EVALUACIÓN DE ÁRBOLES DE EXPRESIONES

5.1 Algoritmo de Evaluación

La evaluación de un árbol de expresión se realiza mediante un algoritmo recursivo:

```
FUNCIÓN Evaluar(nodo):
    SI nodo es hoja:
        RETORNAR valor del nodo
    SINO:
        izq = Evaluar(hijo_izquierdo)
        der = Evaluar(hijo_derecho)

        SEGÚN operador del nodo:
            CASO '+': RETORNAR izq + der
            CASO '-': RETORNAR izq - der
            CASO '*': RETORNAR izq * der
            CASO '/': RETORNAR izq / der
```

5.2 Ejemplo de Evaluación Paso a Paso

Para la expresión `(3 + 2) * (7 - 4)`:

Proceso de evaluación:

1. Evaluar subárbol izquierdo de *: 3 + 2 = 5
2. Evaluar subárbol derecho de *: 7 - 4 = 3
3. Evaluar raíz: 5 * 3 = 15

---

6. IMPLEMENTACIÓN EN C#

6.1 Definición de la Clase Nodo

csharp

```csharp
public class NodoArbol
{
    public string Valor { get; set; }
    public NodoArbol Izquierdo { get; set; }
    public NodoArbol Derecho { get; set; }

    // Constructor para nodos hoja (operandos)
    public NodoArbol(string valor)
    {
        Valor = valor;
        Izquierdo = null;
        Derecho = null;
    }

    // Constructor para nodos internos (operadores)
    public NodoArbol(string operador, NodoArbol izq, NodoArbol der)
    {
        Valor = operador;
        Izquierdo = izq;
        Derecho = der;
    }

    // Método para verificar si es un nodo hoja
    public bool EsHoja()
    {
        return Izquierdo == null && Derecho == null;
    }

    // Método para verificar si es un operador
    public bool EsOperador()
    {
        return Valor == "+" || Valor == "-" || Valor == "*" || Valor == "/";
    }
}
```

6.2 Clase para Evaluar Expresiones

csharp

```
public class ArbolExpresion
{
    private NodoArbol raiz;

    public ArbolExpresion(NodoArbol raiz)
    {
        this.raiz = raiz;
    }

    // Método recursivo para evaluar el árbol
    public double Evaluar()
    {
        return EvaluarNodo(raiz);
    }

    private double EvaluarNodo(NodoArbol nodo)
    {
        // Caso base: nodo hoja (operando)
        if (nodo.EsHoja())
        {
            return double.Parse(nodo.Valor);
        }

        // Caso recursivo: nodo operador
        double valorIzquierdo = EvaluarNodo(nodo.Izquierdo);
        double valorDerecho = EvaluarNodo(nodo.Derecho);

        // Aplicar la operación según el operador
        switch (nodo.Valor)
        {
            case "+":
                return valorIzquierdo + valorDerecho;
            case "-":
                return valorIzquierdo - valorDerecho;
            case "*":
                return valorIzquierdo * valorDerecho;
            case "/":
                if (valorDerecho == 0)
                {
                    throw new DivideByZeroException("División por cero");
                }
                return valorIzquierdo / valorDerecho;
            default:
                throw new InvalidOperationException($"Operador no válido: {nodo.Valor}");
```

```
    }
}

// Recorrido Inorden (Infijo)
public string RecorridoInorden()
{
    return InordenRecursivo(raiz);
}

private string InordenRecursivo(NodoArbol nodo)
{
    if (nodo == null) return "";

    if (nodo.EsHoja())
    {
        return nodo.Valor;
    }

    return "(" + InordenRecursivo(nodo.Izquierdo) + " " +
           nodo.Valor + " " +
           InordenRecursivo(nodo.Derecho) + ")";
}

// Recorrido Preorden (Prefijo)
public string RecorridoPreorden()
{
    return PreordenRecursivo(raiz);
}

private string PreordenRecursivo(NodoArbol nodo)
{
    if (nodo == null) return "";

    if (nodo.EsHoja())
    {
        return nodo.Valor;
    }

    return nodo.Valor + " " +
           PreordenRecursivo(nodo.Izquierdo) + " " +
           PreordenRecursivo(nodo.Derecho);
}

// Recorrido Postorden (Sufijo)
```

```
    public string RecorridoPostorden()
    {
        return PostordenRecursivo(raiz);
    }

    private string PostordenRecursivo(NodoArbol nodo)
    {
        if (nodo == null) return "";

        if (nodo.EsHoja())
        {
            return nodo.Valor;
        }

        return PostordenRecursivo(nodo.Izquierdo) + " " +
               PostordenRecursivo(nodo.Derecho) + " " +
               nodo.Valor;
    }
}
```

6.3 Ejemplo de Uso

csharp

```
class Program
{
    static void Main()
    {
        // Construir el árbol para la expresión: (3 + 2) * 4
        // Estructura:
        //      *
        //     / \
        //    +   4
        //   / \
        //  3   2

        NodoArbol nodo3 = new NodoArbol("3");
        NodoArbol nodo2 = new NodoArbol("2");
        NodoArbol nodo4 = new NodoArbol("4");

        NodoArbol suma = new NodoArbol("+", nodo3, nodo2);
        NodoArbol multiplicacion = new NodoArbol("*", suma, nodo4);

        ArbolExpresion arbol = new ArbolExpresion(multiplicacion);

        // Evaluar la expresión
        double resultado = arbol.Evaluar();
        Console.WriteLine($"Resultado: {resultado}"); // Salida: 20

        // Mostrar recorridos
        Console.WriteLine($"Inorden: {arbol.RecorridoInorden()}");     // ((3 + 2) * 4)
        Console.WriteLine($"Preorden: {arbol.RecorridoPreorden()}");   // * + 3 2 4
        Console.WriteLine($"Postorden: {arbol.RecorridoPostorden()}"); // 3 2 + 4 *
    }
}
```

---

7. EJERCICIOS PROPUESTOS

Ejercicio 1

Construir el árbol de expresión para: `7 - 3 + 2` Evaluarlo paso a paso.

Ejercicio 2

Dada la expresión `4 * (5 + 3) / 2`, construir su árbol y determinar el resultado.

Ejercicio 3

Convertir el siguiente árbol a notación infija, prefija y postfija:

Ejercicio 4

Implementar un método en C# que calcule la altura de un árbol de expresión.

Ejercicio 5

Implementar un método que cuente el número de operadores en un árbol de expresión.

---

8. COMPLEJIDAD COMPUTACIONAL

8.1 Complejidad Temporal

- **Construcción del árbol**: O(n), donde n es el número de tokens en la expresión
- **Evaluación del árbol**: O(n), donde n es el número de nodos
- **Recorridos**: O(n) para inorden, preorden y postorden

8.2 Complejidad Espacial

- **Almacenamiento del árbol**: O(n)
- **Espacio de pila para recursión**: O(h), donde h es la altura del árbol

---

9. VENTAJAS Y DESVENTAJAS

Ventajas

1. Representación clara de la precedencia de operadores
2. Facilita la evaluación eficiente de expresiones
3. Permite la manipulación algebraica (simplificación, derivación)
4. Base fundamental para compiladores e interpretadores
5. Independiente de la notación (infija, prefija, postfija)

Desventajas

1. Requiere más memoria que otras representaciones
2. La construcción puede ser compleja para expresiones muy largas
3. Necesita manejo cuidadoso de paréntesis y precedencia

---

10. REFERENCIAS BIBLIOGRÁFICAS

1. **Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C.** (2022). *Introduction to Algorithms* (4th ed.). MIT Press. ISBN: 978-0262046305
2. **Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D.** (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Pearson. ISBN: 978-0321486813
3. **Weiss, M. A.** (2013). *Data Structures and Algorithm Analysis in C++* (4th ed.). Pearson. ISBN: 978-0132847377
4. **Sedgewick, R., & Wayne, K.** (2011). *Algorithms* (4th ed.). Addison-Wesley Professional. ISBN: 978-0321573513
5. **Knuth, D. E.** (1997). *The Art of Computer Programming, Volume 1: Fundamental Algorithms* (3rd ed.). Addison-Wesley. ISBN: 978-0201896831

CONCLUSIONES

Los árboles de expresiones constituyen una herramienta fundamental en ciencias de la computación, proporcionando una representación estructurada y eficiente de expresiones matemáticas. Su comprensión es esencial para:

- El desarrollo de compiladores y sistemas de procesamiento de lenguajes
- La implementación de calculadoras y evaluadores de expresiones
- El análisis y optimización de código
- La manipulación simbólica en sistemas de álgebra computacional

El dominio de esta estructura de datos permite a los estudiantes comprender mejor cómo las computadoras procesan y evalúan expresiones matemáticas complejas, sentando las bases para conceptos más avanzados en teoría de compiladores y diseño de lenguajes de programación.

Documento preparado para fines académicos
Material de apoyo para cursos de Estructuras de Datos y Algoritmos