Unidad 4 – Árboles y Estructuras Avanzadas

Aplicaciones – Expresiones Aritméticas y Planificación de Tareas

Objetivos

Al finalizar este tema, el estudiante será capaz de:

1. Comprender cómo los **árboles binarios** se utilizan para representar y evaluar **expresiones aritméticas**.
2. Aplicar los conceptos de **recorridos en árboles** para calcular el resultado de una expresión.
3. Implementar **planificación de tareas** usando **colas de prioridad** (basadas en heaps).
4. Identificar la relación entre la teoría de estructuras de datos y problemas reales de ejecución y optimización.

Introducción

Hasta este punto, ya conocen:

- Qué es un **árbol binario** y cómo recorrerlo.
- Qué son los **montículos (heaps)** y cómo implementar colas de prioridad.

En este tema, vamos a ver **cómo se aplican esas estructuras** para resolver problemas comunes en programación, como:

- Interpretar y evaluar expresiones matemáticas (calculadoras, compiladores).
- Organizar tareas según prioridad (sistemas operativos, impresoras, servidores).

Aplicación 1: Árboles de Expresiones Aritméticas

Concepto

Una **expresión aritmética** puede representarse como un **árbol binario**, donde:

- Las **hojas** son operandos (números o variables).
- Los **nodos internos** son operadores (+, -, *, /).

Esto se llama un **Árbol de Expresión**.

Ejemplo

Supongamos la expresión:

```
(3 + 5) * (2 - 4)
```

El árbol de expresión sería:

```
        *
      /   \
     +     -
    / \   / \
   3  5  2  4
```

Tipos de recorridos y su significado

Tipo de recorrido	Resultado	Nombre
Inorden (I-N-D)	3 + 5 * 2 - 4	**Notación infija** (la que usamos normalmente)
Preorden (N-I-D)	* + 3 5 - 2 4	**Notación prefija (Polaca)**
Postorden (I-D-N)	3 5 + 2 4 - *	**Notación postfija (Polaca inversa)**

> Las notaciones **prefija y postfija** se utilizan en compiladores y calculadoras para evaluar expresiones sin paréntesis.

Evaluación de una expresión postfija con pila

Ejemplo:

```
Expresión: 3 5 + 2 4 - *
```

Proceso paso a paso:

1. Lee el token `3` → apila `[3]`
2. Lee `5` → apila `[3, 5]`
3. Lee `+` → desapila 3 y 5 → calcula `3+5=8` → apila `[8]`
4. Lee `2` → apila `[8, 2]`
5. Lee `4` → apila `[8, 2, 4]`
6. Lee `-` → desapila 2 y 4 → calcula `2-4=-2` → apila `[8, -2]`
7. Lee `*` → desapila 8 y -2 → calcula `8 * -2 = -16`

Resultado final: `-16`

Implementación

```python
def evaluar_postfija(expresion):
    pila = []
    for token in expresion.split():
        if token.isdigit():
            pila.append(int(token))
        else:
            b = pila.pop()
            a = pila.pop()
            if token == '+': pila.append(a + b)
```

```
            elif token == '-': pila.append(a - b)
            elif token == '*': pila.append(a * b)
            elif token == '/': pila.append(a / b)
    return pila.pop()

expresion = "3 5 + 2 4 - *"
print("Resultado:", evaluar_postfija(expresion))
```

Salida:

```
Resultado: -16
```

Ventajas del uso de árboles de expresión

- Facilita la **evaluación automática** de operaciones complejas.
- Permite **optimizar expresiones** antes de su cálculo.
- Es base para el análisis sintáctico en **compiladores e intérpretes**.

Aplicación 2: Planificación de Tareas (Scheduling)

Concepto

En un sistema multitarea (por ejemplo, tu computadora o un servidor), hay muchos procesos esperando ser ejecutados. No todos tienen la misma prioridad o el mismo tiempo de ejecución.

Una **cola de prioridad** permite que siempre se atienda **primero la tarea más importante o urgente**.

Ejemplo práctico

Tenemos tareas con diferentes prioridades:

Tarea	Prioridad
Hacer respaldo	3
Enviar correo urgente	1
Actualizar sistema	2

El sistema debe ejecutarlas **en orden de prioridad** (menor número = mayor prioridad).

Implementación con `heapq`

```
import heapq
import time

# (prioridad, nombre_tarea)
tareas = [
    (3, "Hacer respaldo"),
```

```
    (1, "Enviar correo urgente"),
    (2, "Actualizar sistema")
]

heapq.heapify(tareas)

while tareas:
    prioridad, tarea = heapq.heappop(tareas)
    print(f"Ejecutando: {tarea} (Prioridad {prioridad})")
    time.sleep(1)  # Simula tiempo de ejecución
```

Salida:

```
Ejecutando: Enviar correo urgente (Prioridad 1)
Ejecutando: Actualizar sistema (Prioridad 2)
Ejecutando: Hacer respaldo (Prioridad 3)
```

Ejemplo avanzado: planificación con tiempos

Supón que además de prioridad, las tareas tienen **tiempo estimado** de ejecución.

```
import heapq

tareas = []
heapq.heappush(tareas, (2, "Backup", 5))
heapq.heappush(tareas, (1, "Urgente", 2))
heapq.heappush(tareas, (3, "Actualización", 3))

total_tiempo = 0

while tareas:
    prioridad, nombre, tiempo = heapq.heappop(tareas)
    total_tiempo += tiempo
    print(f"Tarea: {nombre} (Prioridad {prioridad}, Tiempo {tiempo})")
    print(f"Tiempo total transcurrido: {total_tiempo} segundos\n")
```

Salida:

```
Tarea: Urgente (Prioridad 1, Tiempo 2)
Tiempo total transcurrido: 2 segundos

Tarea: Backup (Prioridad 2, Tiempo 5)
Tiempo total transcurrido: 7 segundos

Tarea: Actualización (Prioridad 3, Tiempo 3)
Tiempo total transcurrido: 10 segundos
```

Relación entre ambos casos

Concepto	Árboles de Expresión	Colas de Prioridad
Estructura	Árbol binario	Heap (árbol binario completo)
Propósito	Evaluar operaciones	Ordenar por prioridad
Clave del nodo	Operador o valor	Nivel de prioridad
Operaciones comunes	Recorridos (inorden, postorden)	Insertar, extraer mínimo
Aplicaciones	Compiladores, calculadoras	Planificadores, sistemas, redes

Conclusiones

- Los **árboles de expresión** permiten representar y evaluar operaciones matemáticas complejas.
- Las **colas de prioridad**, implementadas con **heaps**, permiten gestionar tareas en orden eficiente.
- Ambos casos muestran cómo las estructuras de datos **dan solución directa a problemas reales**: desde interpretar código hasta gestionar procesos en un sistema operativo.