Programación 2
Estructuras Lineales

Fernando Orejas

1. Estructuras lineales
2. Pilas
3. Colas
4. Implementaciones con vectores
5. Listas

Objetivos

1. Estudiar algunas estructuras de datos
2. Ver cómo podemos construir programas que usan clases predefinidas

Estructuras lineales

Estructuras lineales

- Son estructuras de datos que contienen secuencias de valores
- Los accesos típicos que podemos tener son
  - Al primer elemento
  - Al último elemento
  - Al siguiente elemento
  - Al anterior elemento
- Las modificaciones típicas son inserciones o supresiones que pueden estar limitadas a los extremos

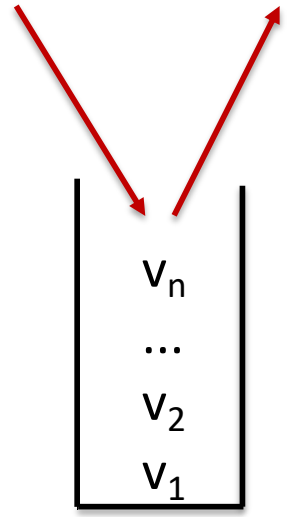
Estructuras lineales

- Ejemplos típicos son pilas, colas, deques, listas, listas con prioridades, etc.
- En la STL de C++: stack, queue, deque, list, priority list, etc.
- Son *templates* (tienen un tipo como parámetro)
- Son contenedores (*containers*)

Pilas

Pilas

- Son estructuras lineales a las que solo se puede *acceder* por un extremo.
- El último insertado es el primero suprimido: *Last In – First Out*
- También se les llama *pushdown stores*



Estructuras lineales

- Operaciones básicas:
  - push
  - pop
  - top
  - empty

Especificación de la clase stack

```
template <class T> class stack {
    public:
    // Constructoras

    // Pre: cert
    // Post: crea una pila vacía
     stack ();

    // Pre: cert
    // Post: crea una pila que es una copia de S
     stack (const stack& S);

    // Destructora
    ~stack();
```

```
// Modificadoras

/* Pre: La pila es [a1,...,an], n ≥ 0 */
/* Post: Se añade el elemento x como primero de la pila,
   es decir, la cola será [x,a1,...,an] */
void push(const T& x);

/* Pre: La pila es [a1,...,an] y no está vacía (n>0) */
/* Post: Se ha eliminado el primer elemento de la pila
   original, es decir, la pila será [a2,...,an] */
void pop ();
```

```
// Consultoras

/* Pre: la pila es [a1,...,an] y no está vacía (n>0) */
/* Post: Retorna a1 */
T top() const;

/* Pre: cert */
/* Post: Retorna true si y solo si la pila está vacía */
bool empty() const;

/* Pre: cert */
/* Post: Retorna el número de elementos de la pila*/
int size() const;

private:
...
};
```

Suma de los elementos de una pila

```
// Pre: cert
// Post: Si la pila está vacía retorna 0
//       si la pila es [a1,...,an], retorna a1+...+an
int suma (stack <int>& p);
```

Suma de los elementos de una pila

```
// Pre: cert
// Post: Si la pila está vacía retorna 0
//       si la pila es [a1,...,an], retorna a1+...+an
int suma (stack <int>& p) {
   int s = 0;
      while (not p.empty()) {
         s = s + p.top();
         p.pop()
      }
   return s;
}
```

Suma de los elementos de una pila

```
// Pre: cert
// Post: Si la pila está vacía retorna 0
//       si la pila es [a1,...,an], retorna a1+...+an
int suma (stack <int>& p) {
   if (p.empty()) return 0;
   else {
      int x = p.top();
      p.pop()
      return x + suma(p);
   }
}
```

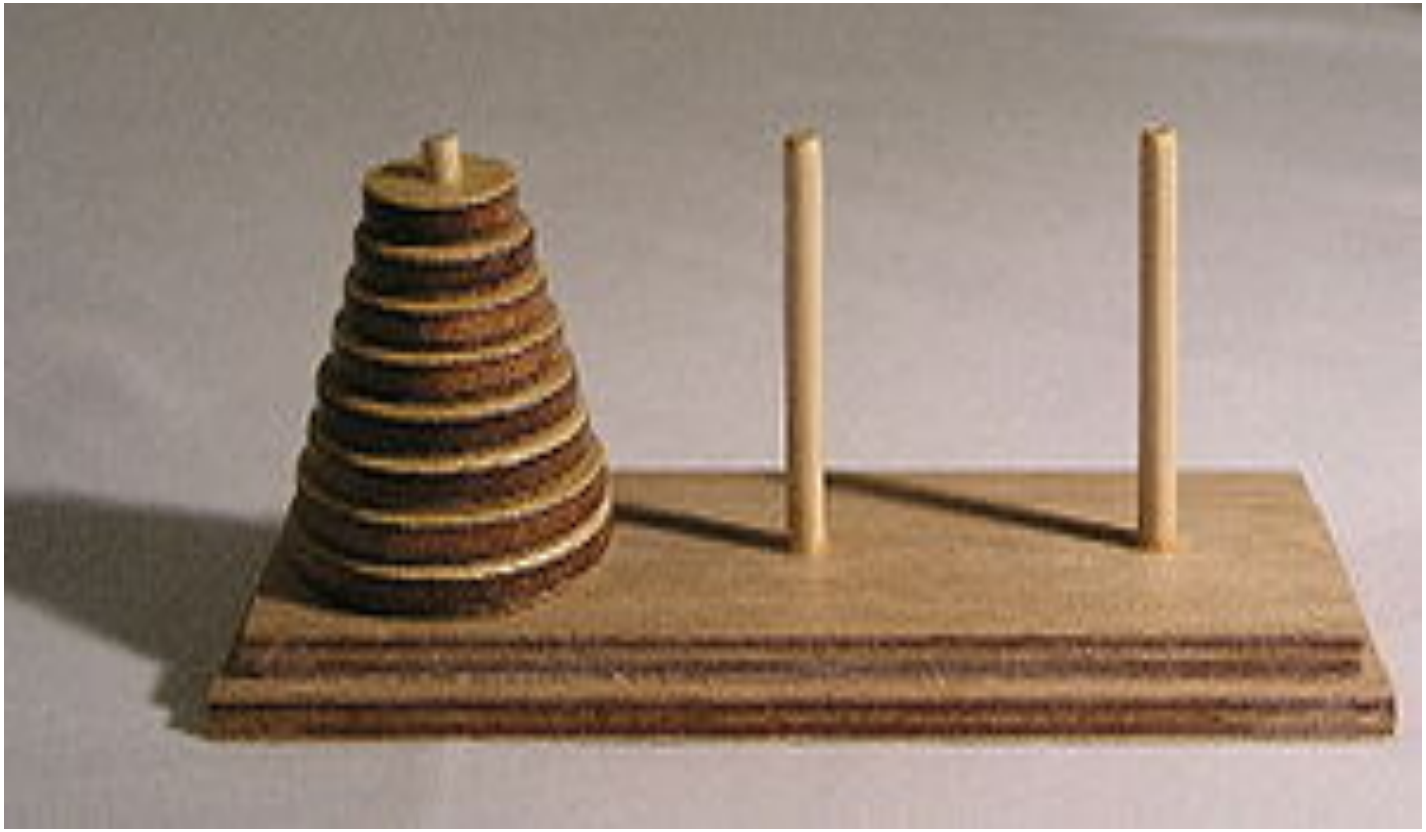
Suma de los elementos de una pila

```
// Pre: cert
// Post: Si la pila está vacía retorna 0
//       si la pila es [a1,...,an], retorna a1+...+an
int suma (stack <int>& p) {
   if (p.empty()) return 0;
   else {
      int x = p.top();
      p.pop()
      int s = x + suma(p);
      p.push(x);
      return s;
   }
}
```

Utilización de las pilas

Las pilas tienen muchos usos en programación, pero uno de los más importantes es la transformación recursiva-iterativa.

Torres de Hanoi



Especificación

```
// Pre:  N es el número de discos (N≥0).
// Post: Se escribe una solución de cómo mover n discos
//       del poste org al poste dst usando el poste aux como
//       auxiliar

typedef char poste;
void hanoi(int N, poste org, poste aux, poste dst);
```

```
> Hanoi
5
Mueve un disco desde L a R
Mueve un disco desde L a M
Mueve un disco desde R a M
Mueve un disco desde L a R
Mueve un disco desde M a L
Mueve un disco desde M a R
Mueve un disco desde L a R
Mueve un disco desde L a M
Mueve un disco desde R a M
Mueve un disco desde R a L
Mueve un disco desde M a L
Mueve un disco desde R a M
Mueve un disco desde L a R
Mueve un disco desde L a M
Mueve un disco desde R a M
```

```
Mueve un disco desde L a R
Mueve un disco desde M a L
Mueve un disco desde M a R
Mueve un disco desde L a R
Mueve un disco desde M a L
Mueve un disco desde R a M
Mueve un disco desde R a L
Mueve un disco desde M a L
Mueve un disco desde M a R
Mueve un disco desde L a R
Mueve un disco desde L a M
Mueve un disco desde R a M
Mueve un disco desde L a R
Mueve un disco desde M a L
Mueve un disco desde M a R
Mueve un disco desde L a R
```

Caso general:

- Movemos n-1 discos de la izquierda al centro
- Movemos el mayor disco a la derecha
- Movemos n-1 discos del centro a la derecha

Algoritmo recursivo

```
void Hanoi(int N, poste org, poste aux, poste dst) {
    if (N == 1) {
       cout << “Mueve un disco desde “ << org
            << “ a “ << dst << endl;
    else
       Hanoi(N-1, org, dst, aux);
       cout << “Mueve un disco desde “ << org
             << “ a “ << dst << endl;
       Hanoi(N-1, aux, org, dst);
    }
}
```

Algoritmo iterativo

```
class HanoiTask {
private:
    int n_;
    poste org_, aux_, dst_;
public:
    HanoiTask(int n, poste org, poste aux, poste dst) {
        n_ = n; org_ = org; aux_ = aux; dst_ = dst;
    }
```

Algoritmo iterativo

```
class HanoiTask {
    ...
    // consultoras
    int ndiscos() const { return n_; }
    poste origen() const { return org_; }
    poste auxiliar() const { return aux_; }
    poste destino() const { return dst_; }
    // Pre: ndiscos() == 1
    // Post: escribe en el cout el movimiento necesario
    // para moure un disco)
    void escribir_mov() {
        cout << "Mueve disco de " << org_ << " a " << dst_
             << endl;
    }
};
```

Algoritmo iterativo

```
void hanoi(int N, poste org, poste aux, poste dst) {
    HanoiTask initial_task(N, org, aux, dst);
    stack <HanoiTask > S;
    S.push(initial_task);
    while (not S.empty()) {
        HanoiTask curr = S.top(); S.pop();
        if (curr.ndiscos() == 1) {
            curr.escribir_mov();
         } else {
        // curr.ndiscos() > 1
        ...
        }
    }
}
```

Algoritmo iterativo

```
        } else {
        // curr.ndiscos() > 1
        ...
    HanoiTask task1(curr.ndiscos()-1,
        curr.origen(), curr.destino(), curr.auxiliar());
    HanoiTask task2(1, curr.origen(), '*',
        curr.destino());
    HanoiTask task3(curr.ndiscos()-1, curr.auxiliar(),
        curr.origen(), curr.destino());
    S.push(task3);
    S.push(task2);
    S.push(task1);
        }
    }
}
```

Evaluación de expresiones

En una expresión en notación postfija (o polaca inversa) se escriben, primero, los operandos y, a continuación el operador. Por ejemplo:

34 25 – 12 4 + *

es la expresión (34 – 25) * (12 + 4)

Especificación

```
class Token { public:
...
};

// Pre:  Expr es un vector que contiene una secuencia de
//       tokens que representa una expresión en notacion
//       postfija
// Post: devuelve el valor de la expresion
//       del poste org al poste dst usando el poste aux como
//       auxiliar

int evalua(const vector <Token >& expr);
```

```
class Token {
    public:
    ...
    bool es_operand() const;
    bool es_operador() const;
    int valor() const;
    char operador() const;
    ...
};

int opera(char c, int x, int y) {
    if (c == '+') return x+y;
    if (c == '*') return x*y;
    ...
}
```

```
int evalua(const vector <Token >& expr){
    stack<int> S;
    for (int i = 0; i < expr.size(); ++i) {
        if (expr[i].es_operand()) {
            S.push(expr[i].valor());
        } else { // expr[i] és un operador
            int op2 = S.top(); S.pop();
            int op1 = S.top(); S.pop();
            char c = expr[i].operador();
            S.push(opera(c, op1, op2));
        }
    }
    // S.size() == 1
    return S.top();
}
```