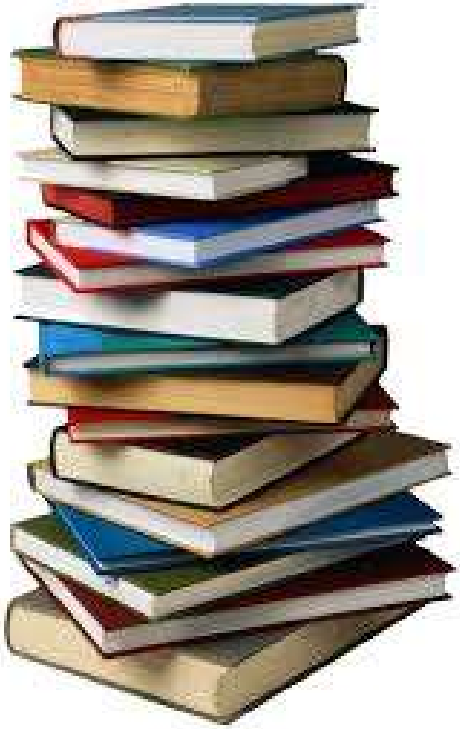
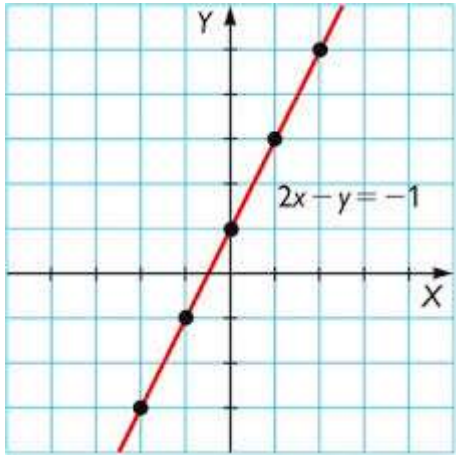


1.7 Estructuras de datos Lineales

1.7.1 Generalidades

- Las estructuras de datos lineales están compuestas por una secuencia de 0 o más elementos de algún tipo determinado en ocasiones ordenados de alguna forma.



1.7.1 Generalidades

- Se puede aumentar o disminuir la cantidad de datos y podrán insertarse o eliminarse elementos en determinadas posiciones sin alterar su estructura lógica.

1.7.1.1 El concepto de Lista

- Una lista es una colección de elementos del mismo tipo
- Para el acceso y manejo de esos elementos se utilizan índices o posiciones

1.7.1.1 El concepto de Lista

- Una lista tiene operaciones de inserción, eliminación y búsqueda
- Una lista es una estructura de datos flexible en las operaciones que se realizan

1.7.2 Pila (Stack)

1.7.2 Pila

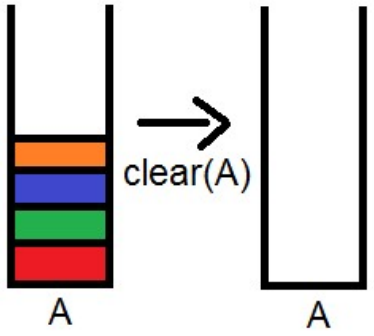
- Una pila (stack) es un tipo abstracto de datos tipo LIFO (*last in, first out*).
- Es un tipo de dato que soporta operaciones de inserción y eliminación de elementos de un conjunto.
- Normalmente se usa en aplicaciones en las que se necesita recuperar información en orden inverso a como ha entrado.

1.7.2 Pila

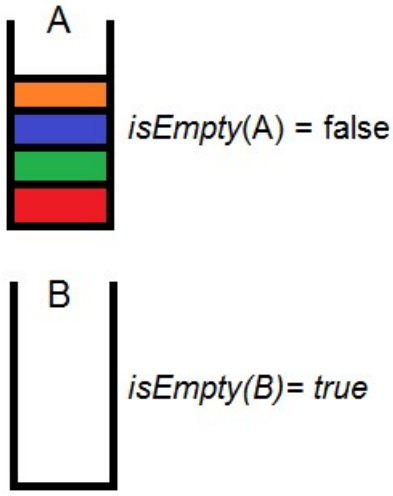
- Esta estructura funciona de forma similar a una torre de elementos del mismo tipo (documentos pendientes, platos, libros, etc).
- Cada vez que llega un elemento, se coloca arriba de los que ya están acumulados, y cuando se van atendiendo se toma el que se encuentra en la parte superior.

Operaciones (Pila)

- Las operaciones que se realizan sobre una pila son:
  - *clear()* Borra todos los elementos de una pila

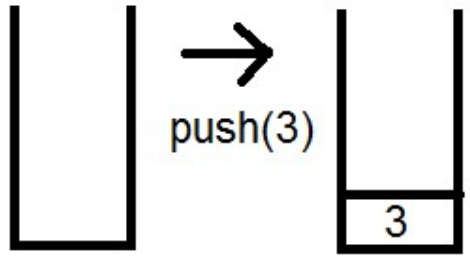


  - *isEmpty()* Verifica si una pila se encuentra vacía

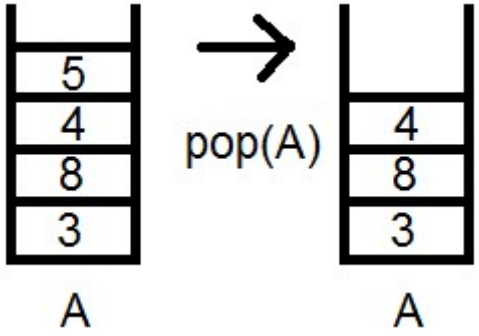


Operaciones (Pila)

- *push()* Ingresa un elemento *en* el tope de la pila

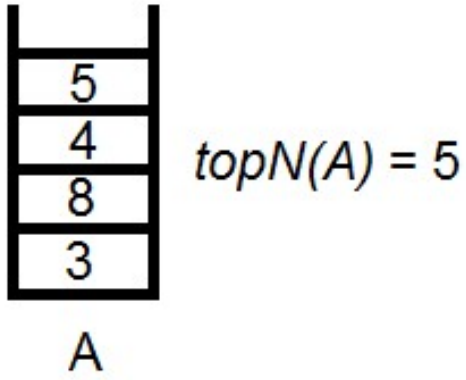


- *pop() Retira el elemento superior*



Operaciones (Pila)

- *top() Verifica* el elemento que se encuentra en la parte superior de la pila sin eliminarlo.



Operaciones (Pila)

- La complejidad temporal asintótica de las operaciones es:

Operación	Orden
Create	O(1)
clear()	O(n)
isEmpty()	O(1)
Push()	O(1)
Pop()	O(1)
top	O(1)

Implementación (Pila)

```
Pila:
    int tope;
    Lista elementos;

Pila crearPila()
    Pila p;
    Lista lista1;
    p.tope = -1;
    p.elementos = lista1;
    return p;
```

Implementación (Pila)

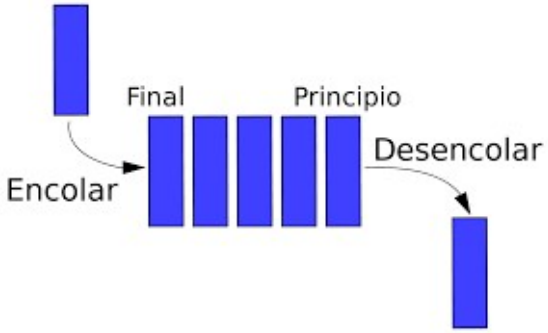
```
bool esVacia(Pila p)          //isEmpty(..)
    if(p.tope==-1)
        return true;
    return false;

void meter(Pila p,int x)   //void push(..)
    p.tope = p.tope+1
    p.elementos[p.tope] = x
```

Implementación (Pila)

```
int sacar(Pila p) // int pop(..)
    if(esVacia(p))
        print("Error")
        return -1
    else
        int aux= p.elementos[p.tope]
        p.elementos[p.tope]=NULL;
        p.tope=p.tope-1
        return aux
```

Implementación (Pila)

```
int top(Pila p)
    if(esVacia(p))
        print("Error");
        return -1;
    else
        return p.elementos[p.tope]
```



1.7.3 Cola (Queue)

1.7.3 Cola

- Una cola (queue) es un tipo abstracto de datos de tipo FIFO (first in first out).
- Es un tipo de dato que soporta operaciones de inserción y eliminación de elementos de un conjunto de datos

1.7.3 Cola

- Modela situaciones en las que una tarea se realiza en el mismo orden en el que se reciben.
- Ejemplos de ello son tareas que esperan ser atendidas por alguna aplicación en la computadora o en un servidor.
- Los cambios que se realizan, se deben monitorear en ambos extremos de la estructura

Operaciones (Queue)

- Las operaciones que se realizan sobre una cola son:
  - *create()* Crea una cola vacía.
  - *isEmpty()* Devuelve verdadero si una cola se encuentra vacía.
  - *enqueue()* Ingresa un elemento a la cola

Operaciones (Queue)

- *dequeue()* Extraer el elemento al inicio de la cola.
- *clear()* Borra todos los elementos de la cola.
- *first()* Regresa el primer elemento de la cola sin eliminarlo.

Operaciones (Queue)

- La complejidad temporal asintótica de las operaciones es:

Operación	Orden
Create	O(1)
clear()	O(n)
isEmpty()	O(1)
enqueue()	O(1)
dequeue()	O(1)
first(n)	O(1)

Implementación (Queue)

```
Cola:
    int primero;
    int ultimo;
    Lista elementos;

Queue CrearCola()
    Queue c;
    Lista miLista;
    c.primero = 0
    c.ultimo = -1
    c.elementos = miLista;
    return c;
```

Implementación (Queue)

```
bool esVacia(Queue c)
     if(c.primero==c.ultimo+1)
          return true;
     return false;

void encolar(Queue c,int x)
     c.ultimo = c.ultimo+1
     c.elementos[c.ultimo] = x
```

Implementación (Queue)

```
int desencolar(Queue c)    //(dequeue)
    if(esVacia(c))
        "error"
        return -1
    else
        int aux = c.elementos[c.primero]
        c.elementos[c.primero]=null;
        c.primero = c.primero + 1
        if(primero==ultimo+1)
            c=crearCola()
        return aux
```

1.7.3.4 Cola Circular

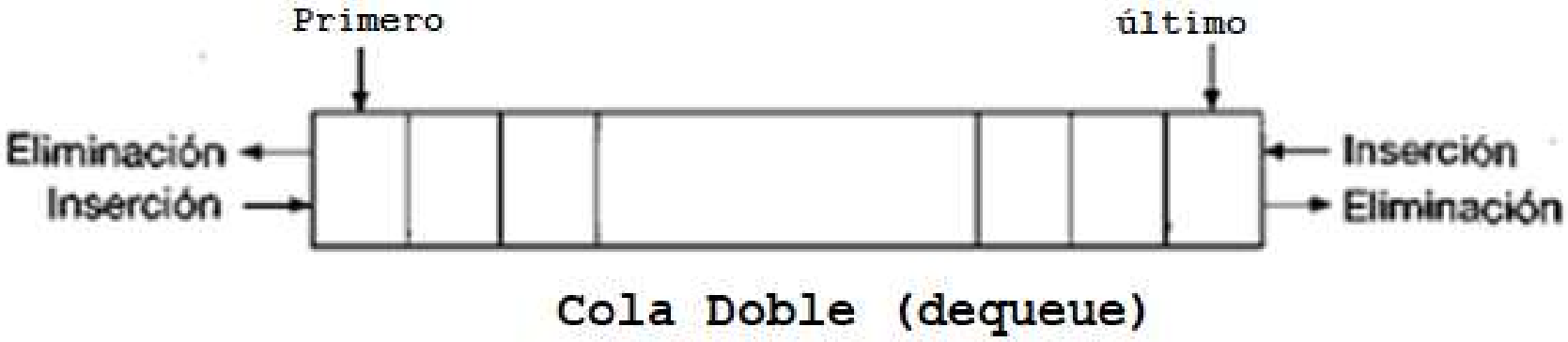
- Los elementos están de forma “circular”, lo que implica que todos los elementos tienen sucesor y predecesor.
- El sucesor del último elemento se trata del primer elemento mientras que el antecesor del primero es el último elemento.
- En una cola circular es indispensable definir el tamaño de la estructura
- La ventaja de una Cola circular es que el manejo de memoria es más eficiente

1.7.3.4 Cola Circular (operaciones)

- Las operaciones con la cola circular, son las mismas que en una “queue” simple (*enqueue, dequeue, first, etc*).
- El usuario no ve diferencia cuando utiliza una u otra
- **En la implementación** se requiere añadir elementos a las funciones encolar y desencolar, para el correcto manejo de índices (si se implementa con arreglos) o para el correcto manejo de memoria (si se implementa con listas ligadas)

1.7.3.5 Cola Doble

- Una cola doble (dqueue) es un tipo abstracto de datos en donde cada elemento puede ser ingresado y recuperado por ambos lados de la estructura.
- Con esto se da una mayor flexibilidad al tipo de dato ya que el elemento que ingresa puede ser el primero o el último en salir.



Cola Doble (dequeue)

Operaciones (Cola doble)

- *create()* Crea una cola vacía.
- *isEmpty()* Devuelve verdadero si una cola se encuentra vacía.
- *enqueueEnd():* Ingresa un elemento al final de la cola.
- *dequeueBegin()*: Elimina y devuelve el elemento que se encuentra al principio de la cola.
- ***enqueueBegin():*** Ingresa un elemento al principio de la cola.
- ***dequeueEnd()*:** Elimina y devuelve el elemento que este a final de la cola.

Cola de Prioridad

Se caracteriza por admitir inserciones, consulta y eliminación de elementos con valores de prioridad ponderados.

Cuando ingresa un elemento se coloca en la última posición entre aquellos con su misma prioridad

Aplicaciones (Pila)

- Evaluación de expresiones aritméticas (1)

Shunting-yard Algorithm

Propuesto por Edsger W. Dijkstra en la década de los 60's.
Utiliza dos pilas, una para operandos y una para operadores.

Para este tipo de expresiones se requiere hacer una lectura de la expresión revisando cada uno de los caracteres que la conforman

Aplicaciones (Pila)

Dada una expresión aritmética (leyendo de izquierda a derecha)

- Ignorar los paréntesis de apertura
- Operando: **Push(opn)** en stack de operandos
- Operador: **Push(opr)** en stack de operadores
- Al encontrar un paréntesis de cierre
  - **Pop** operando
  - **Pop** operador
  - **Pop** operando
  - Realizar operación
  - **Push** resultado en stack de operandos
- Al finalizar la expresión
  - Pop resultado final.

Aplicaciones (Pila)

- Evaluación de expresiones aritméticas (2).

Reverse Polish Notation

La ventaja de esta forma de escribir expresiones aritméticas, es que no es necesario utilizar paréntesis ni reglas de precedencia.

En este algoritmo se utiliza una sola pila (stack).

Aplicaciones (Pila)

Sea una expresión aritmética (leyendo de izquierda a derecha)

- Si se encuentra un operando:
  - **Push** operando.
- Si se encuentra un operador:
  - **Pop** operando.
  - **Pop** operando.
  - **Push** resultado.

Aplicaciones (Queue)

- Algunos ejemplos de uso de “queues”:
  - Procesos atendidos por un procesador
  - Documentos pendientes de imprimir
  - En redes de computadoras, los paquetes suelen transportarse a través de colas.