# 5 Pilas, Colas y Colas de Prioridad

En este capítulo veremos tres *tipos de datos abstractos* (*TDAs*) que son muy utilizados.

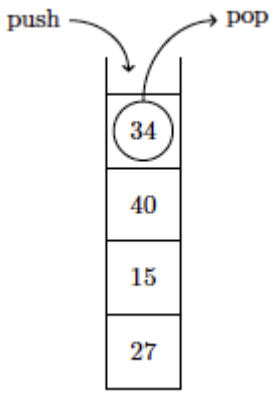
Un tipo de datos abstracto es un conjunto de datos, más operaciones asociadas, para el cual se aplica una política de "ocultamiento de información": los usuarios del TDA saben **qué** funcionalidad éste provee, pero no saben **cómo** se implementa esta funcionalidad.

Esta separación de responsabilidad es fundamental para mantener la complejidad bajo control. Sólo los implementadores del TDA necesitan preocuparse de su implementación, y además son libres para modificarla en la medida que la interfaz de uso se mantenga intacta.

## Pilas ("*Stacks*")

Una **pila**, también llamada *stack* o *pushdown* en inglés, es una lista de elementos en la cual todas las operaciones se realizan solo en un extremo de la lista.

Es usual visualizar la pila creciendo verticalmente hacia arriba, y llamamos "tope" a su extremo superior:




Las dos operaciones básicas son **push** (apilar), que agrega un elemento encima de todos, y **pop** (desapilar), que extrae el elemento del tope de la pila. Más precisamente, si `s` es un objeto de tipo Pila, están disponibles las siguientes operaciones:

- `s.push(x)` : apila x en el tope de la pila `s`
- `x=s.pop()` : extrae y retorna el elemento del tope de la pila `s`
- `b=s.is_empty()` : retorna verdadero si la pila `s` está vacía, falso si no

Dado que los elementos salen de la pila en el orden inverso en que ingresaron, esta estructura también se conoce como "lista LIFO", por "Last-In-First-Out".

## Implementación usando listas de Python

Es posible implementar una pila muy fácilmente usando las listas que provee el lenguaje Python:

In [1]:
```
class Pila:
    def __init__(self):
        self.s=[]
    def push(self,x):
        self.s.append(x)
    def pop(self):
        assert len(self.s)>0
        return self.s.pop() # pop de lista, no de Pila
    def is_empty(self):
        return len(self.s)==0
```

In [2]:
```
a=Pila()
a.push(10)
a.push(20)
print(a.pop())
a.push(30)
print(a.pop())
print(a.pop())
```

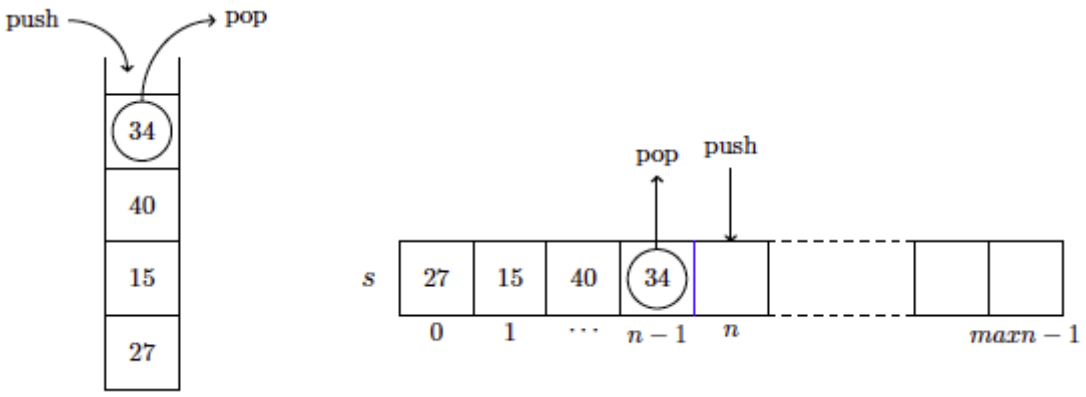
```
20
30
10
```

Esta implementación simple posiblemente sirve en la mayoría de los casos, pero si necesitamos poder garantizar su eficiencia, tenemos el problema que la implementación de las listas de Python está fuera de nuestro control, y no podemos garantizar, por ejemplo, que cada una de las operaciones tome tiempo constante.

Por ese motivo, es útil contar con implementaciones en que sí podamos dar ese tipo de garantía.

## Implementación usando un arreglo

Utilizaremos un arreglo $s$, en donde los elementos de la pila se almacenarán en los casilleros $0, 1, \ldots$, con el elemento del tope en el casillero ocupado de más a la derecha. Mantendremos una variable $n$ para almacenar el número de elementos presentes en la pila, y el arreglo tendrá un tamaño máximo, el que se podrá especificar opcionalmente al momento de crear la pila.




In [1]:
```
import numpy as np
class Pila:
    def __init__(self,maxn=100):
        self.s=np.zeros(maxn)
        self.n=0
    def push(self,x):
        assert self.n<len(self.s)-1
        self.s[self.n]=x
        self.n+=1
    def pop(self):
        assert self.n>0
        self.n-=1
        return self.s[self.n]
    def is_empty(self):
        return self.n==0
```

In [2]:
```
a=Pila()
a.push(10)
a.push(20)
print(a.pop())
a.push(30)
print(a.pop())
print(a.pop())
```

```
20.0
30.0
10.0
```

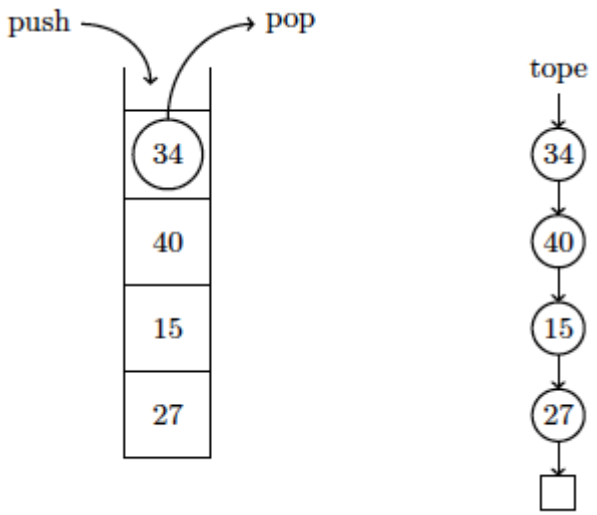
Esta implementación es muy eficiente: no solo es evidente que cada operación toma tiempo constante, sino además esa constante es muy pequeña. Sin embargo, tiene la limitación de que es necesario darle un tamaño máximo al arreglo, el cual a la larga puede resultar insuficiente.

Existe una manera de copiar todos los elementos a un arreglo más grande y seguir operando cuando el arreglo se llena. Si el nuevo arreglo es del doble del tamaño anterior, el costo de copiar todos los elementos se puede *amortizar* a lo largo de las operaciones, de modo que en *promedio* sea constante, pero se pierde la propiedad de que las operaciones tomen tiempo constante en el peor caso.

La siguiente es otra alternativa de implementación, que no sufre de ese problema.

## Implementación usando una lista enlazada

En esta implementación los elementos de la pila se almacenan en una lista de enlace simple (sin cabecera), en que el elemento del tope de la pila es el primero de la lista.



In [5]:
```
class NodoLista:
    def __init__(self,info,sgte=None):
        self.info=info
        self.sgte=sgte
class Pila:
    def __init__(self):
        self.tope=None
    def push(self,x):
        self.tope=NodoLista(x,self.tope)
    def pop(self):
        assert self.tope is not None
        x=self.tope.info
        self.tope=self.tope.sgte
        return x
    def is_empty(self):
        return self.tope is None
```

In [6]:
```
a=Pila()
a.push(10)
a.push(20)
print(a.pop())
a.push(30)
print(a.pop())
print(a.pop())
```

```
20
30
10
```

## Aplicaciones de pilas

### Evaluación de notación polaca

Si se tiene una fórmula en notación polaca, se puede calcular su valor usando una pila, inicialmente vacía. Los símbolos de la fórmula se van leyendo de izquierda a derecha, y:

- si el símbolo es un número, se le hace `push` en la pila
- si el símbolo es un operador, se hacen dos `pop` , se efectúa la operación indicada entre los dos datos obtenidos, y el resultado se agrega de vuelta a la pila con `push`

Al terminar, si la fórmula estaba bien formada, debe haber solo un elemento en la pila, que es el resultado de la evaluación de la fórmula.

In [7]:
```
def eval_polaca(formula):
    a=Pila()
    for x in formula.split():
        if x.isnumeric():
            a.push(int(x))
        else: # tiene que ser un operador
            v=a.pop()
            u=a.pop()
            if x=="+":
                w=u+v
            elif x=="-":
                w=u-v
            elif x=="*":
                w=u*v
            elif x=="/":
                w=u/v
            else:
                print("Operador desconocido:",x)
                return 0
            a.push(w)
    return a.pop()
```

In [8]:
```
formula=input('Escriba la fórmula en notación polaca: ')
print("Resultado: ",eval_polaca(formula))
```

```
Escriba la fórmula en notación polaca: 2 3 + 9 4 2 / - *
Resultado:  35.0
```

### Recorrido no recursivo de un árbol binario

Supongamos que queremos recorrer un árbol binario en preorden. En lugar de utilizar un algoritmo recursivo, podemos imaginar que tenermos una "To DO list" en donde almacenamos la lista de nodos que debemos visitar en el futuro. Inicialmente, esta lista contiene solo la raíz. En cada iteración, extraemos un nodo de la lista, lo visitamos, y luego agregamos a la lista a sus dos hijos. Si la lista se mantiene como una pila, el orden del en que se visitan los nodos es exactamente preorden.

In [9]:
```
class Nodo:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der

class Arbol:
    def __init__(self,raiz=None):
        self.raiz=raiz

    def preorden(self):
        print("Preorden no recursivo:", end=" ")
        s=Pila()
        s.push(self.raiz)
        while not s.is_empty():
            p=s.pop()
            if p is not None:
                print(p.info, end=" ")
                s.push(p.der)
                s.push(p.izq)
        print()
```

Es importante que las operaciones `push` se hagan en el orden indicado (derecho-izquierdo), para que de acuerdo a la disciplina LIFO, salga primero el izquierdo y luego el derecho.

In [10]:
```
a=Arbol(
    Nodo(
        Nodo(
            Nodo(None,15,None),
            20,
            Nodo(
                Nodo(None,30,None),
                35,
                None
            )
        ),
        42,
        Nodo(
            Nodo(
                Nodo(
                    Nodo(None,65,None),
                    72,
                    Nodo(None,81,None)
                ),
                90,
                None
            ),
            95,
            None
        )
       )
)
```

In [11]:
```
a.preorden()
```

```
Preorden no recursivo: 42 20 15 35 30 95 90 72 65 81
```

Hay una pequeña optimización que se puede hacer al algoritmo de recorrido no recursivo. Cuando hacemos las dos operaciones `push` y volvemos a ejecutar el `while`, sabemos que la pila no está vacía, de modo que esa pregunta es superflua. Además, al hacer el `pop` sabemos que lo que va a salir de la pila es lo último que se agregó, o sea, `p.izq`. Por lo tanto, podemos saltarnos tanto la pregunta como el `pop` e ir directamente al `if`, el cual por lo tanto se transforma en un `while`.

In [12]:
```
class Arbol:
    def __init__(self,raiz=None):
        self.raiz=raiz

    def preorden(self):
        print("Preorden no recursivo optimizado:", end=" ")
        s=Pila()
        s.push(self.raiz)
        while not s.is_empty():
            p=s.pop()
            while p is not None:
                print(p.info, end=" ")
                s.push(p.der)
                p=p.izq
        print()
```

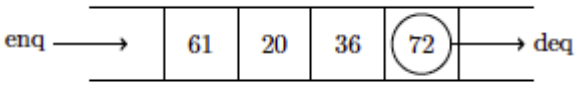
In [13]:
```
a=Arbol(
    Nodo(
        Nodo(
            Nodo(None,15,None),
            20,
            Nodo(
                Nodo(None,30,None),
                35,
                None
            )
        ),
        42,
        Nodo(
            Nodo(
                Nodo(
                    Nodo(None,65,None),
                    72,
                    Nodo(None,81,None)
                ),
                90,
                None
            ),
            95,
            None
        )
       )
)
```

In [14]:
```
a.preorden()
```

```
Preorden no recursivo optimizado: 42 20 15 35 30 95 90 72 65 81
```

# Colas ("*Queues*")

Una cola es una lista en que los elementos ingresan por un extremo y salen por el otro. Debido a que los elementos van saliendo en orden de llegada, una cola también se llama "lista FIFO", por "First-In-First-Out".




Las dos operaciones básicas son **enq** (encolar), que agrega un elemento al final de todos, y **deq** (desencolar), que extrae el elemento que encabeza la cola. Más precisamente, si `q` es un objeto de tipo Cola, están disponibles las siguientes operaciones:

- `q.enq(x)` : encola x al final de la cola `q`
- `x=q.deq()` : extrae y retorna el elemento a la cabeza de la cola `q`
- `b=q.is_empty()` : retorna verdadero si la cola `q` está vacía, falso si no

## Implementación usando listas de Python

Tal como hicimos en el caso de las pilas, es muy simple implementar colas usando las listas de Python, pero no tenemos mucho control sobre la eficiencia del resultado:

In [15]:
```
class Cola:
    def __init__(self):
        self.q=[]
    def enq(self,x):
        self.q.insert(0,x)
    def deq(self):
        assert len(self.q)>0
        return self.q.pop()
    def is_empty(self):
        return len(self.q)==0
```

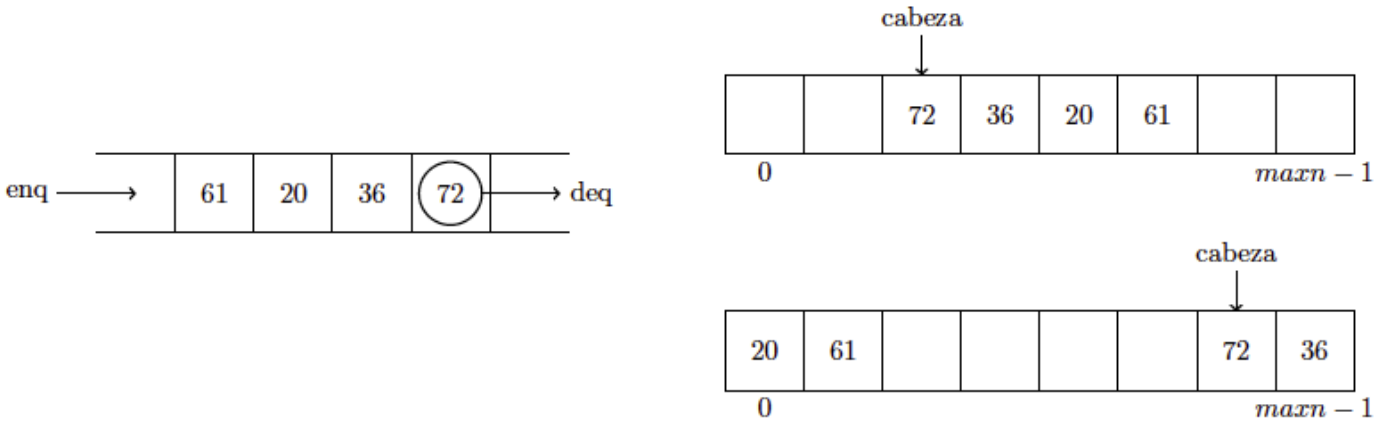
In [16]:
```
a=Cola()
a.enq(72)
a.enq(36)
print(a.deq())
a.enq(20)
print(a.deq())
print(a.deq())
a.enq(61)
print(a.deq())
```

```
72
36
20
61
```

## Implementación usando un arreglo

De manera análoga a lo que hicimos en el caso de la pila, podemos almacenar los $n$ elementos de la cola usando posiciones contiguas en un arreglo, por ejemplo, las $n$ primeras posiciones. Pero hay un problema: como la cola crece por un extremo y se achica por el otro, ese grupo de posiciones contiguas se va desplazando dentro del arreglo, y después de un rato choca contra el otro extremo. La solución es ver al arreglo como *circular*, esto es, que si el arreglo tiene tamaño $maxn$, a continuación de la posición $maxn-1$ viene la posición $0$. Esto se puede hacer fácilmente usando aritmética módulo $maxn$.

Para la implementación, utilizaremos un subíndice $cabeza$ que apunta al primer elemento de la cola, y una variable $n$ que indica cuántos elementos hay en la cola. La siguiente figura muestra dos situaciones en que podría encontrarse el arreglo:



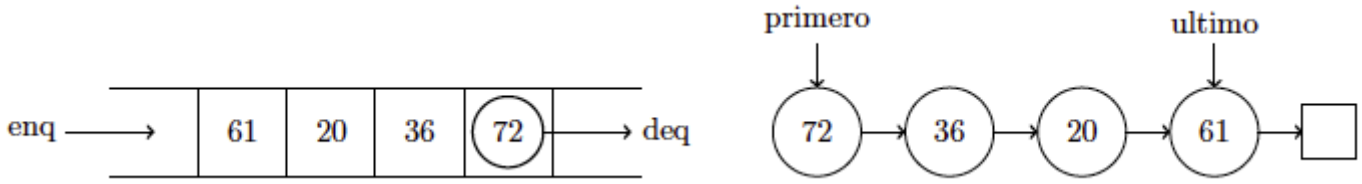

In [17]:
```
import numpy as np
class Cola:
    def __init__(self,maxn=100):
        self.q=np.zeros(maxn)
        self.n=0
        self.cabeza=0
    def enq(self,x):
        assert self.n<len(self.q)-1
        self.q[(self.cabeza+self.n)%len(self.q)]=x
        self.n+=1
    def deq(self):
        assert self.n>0
        x=self.q[self.cabeza]
        self.cabeza=(self.cabeza+1)%len(self.q)
        self.n-=1
        return x
    def is_empty(self):
        return self.n==0
```

In [18]:
```
a=Cola(3) # para forzar circularidad
a.enq(72)
a.enq(36)
print(a.deq())
a.enq(20)
print(a.deq())
print(a.deq())
a.enq(61)
print(a.deq())
```

```
72.0
36.0
20.0
61.0
```

## Implementación usando una lista enlazada

El operar en los dos extremos de la cola sugiere de inmediato el uso de una lista de doble enlace, y esa es una opción posible. Pero, como veremos, se puede implementar una cola con una lista de enlace simple:




Una cosa que complica un poco la programación es que el invariante que se ve a la derecha se cumple solo si la cola es no vacía. Para una cola vacía, los dos punteros (primero y último) son nulos. Por lo tanto, un `enq` sobre una cola vacía, y un `deq` que deja una cola vacía serán casos especiales.

In [19]:
```
class NodoLista:
    def __init__(self,info,sgte=None):
        self.info=info
        self.sgte=sgte
class Cola:
    def __init__(self):
        self.primero=None
        self.ultimo=None
    def enq(self,x):
        p=NodoLista(x)
        if self.ultimo is not None: # cola no vacía, agregamos al final
            self.ultimo.sgte=p
            self.ultimo=p
        else: # la cola estaba vacía
            self.primero=p
            self.ultimo=p
    def deq(self):
        assert self.primero is not None
        x=self.primero.info
        if self.primero is not self.ultimo: # hay más de 1 elemento
            self.primero=self.primero.sgte
        else: # hay solo 1 elemento, el deq deja la cola vacía
            self.primero=None
            self.ultimo=None
        return x
    def is_empty(self):
        return self.primero is None
```

In [20]:
```
a=Cola()
a.enq(72)
a.enq(36)
print(a.deq())
a.enq(20)
print(a.deq())
print(a.deq())
a.enq(61)
print(a.deq())
```

```
72
36
20
61
```

## Aplicaciones de colas

Las colas se utilizan en los sistemas operativos siempre que hay algún recurso que no puede ser compartido. Uno de los procesos que lo requieren tiene acceso al recurso, mientras los demás deben esperar en una cola. Un ejemplo de esto son los sistemas de "spooling" para las impresoras.

También se usan mucho en sistemas de simulación, cuando se deben modelar situaciones del mundo real en que hay colas. Por ejemplo, la caja en un supermercado.

A continuación veremos una aplicación análoga a la que vimos en el caso de pilas para el recorrido de un árbol binario.

### Recorrido de un árbol binario por niveles

Supongamos que se desea recorrer un árbol binario, visitando sus nodos en orden de su distancia a la raíz. No hay manera de escribir esto de manera recursiva, pero el problema se puede resolver usando el mismo enfoque que utilizamos al recorrer un árbol binario en preorden de manera no recursiva, pero usando una cola en lugar de una pila.

In [21]:
```
class Nodo:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der

class Arbol:
    def __init__(self,raiz=None):
        self.raiz=raiz

    def niveles(self):
        print("Recorrido por niveles:", end=" ")
        c=Cola()
        c.enq(self.raiz)
        while not c.is_empty():
            p=c.deq()
            if p is not None:
                print(p.info, end=" ")
                c.enq(p.izq)
                c.enq(p.der)
        print()
```

In [22]:
```
a=Arbol(
    Nodo(
        Nodo(
            Nodo(None,15,None),
            20,
            Nodo(
                Nodo(None,30,None),
                35,
                None
            )
        ),
        42,
        Nodo(
            Nodo(
                Nodo(
                    Nodo(None,65,None),
                    72,
                    Nodo(None,81,None)
                ),
                90,
                None
            ),
            95,
            None
        )
       )
)
```

In [23]:
```
a.niveles()
```

```
Recorrido por niveles: 42 20 95 15 35 90 30 72 65 81
```

# Colas de Prioridad

Hay muchas situaciones en que los elementos que esperan en una cola deben ir siendo atendidos no por orden de llegada, sino de acuerdo a algún criterio de *prioridad*. En esos casos, no nos sirve la cola como la hemos visto, sino que se necesita un nuevo tipo de estructura.

Una cola de prioridad es un TDA que consiste de un conjunto de datos que poseen un atributo (llamado su *prioridad*) perteneciente a algún conjunto ordenado, y en el cual se pueden ejecutar dos operaciones básicas: **insertar** un nuevo elemento con una prioridad cualquiera y **extraer** el elemento de máxima prioridad.

Más específicamente, las operaciones permitidas son:

- `q.insert(x)` : inserta un elemento de prioridad x en la cola de prioridad `q`
- `x=q.extract_max()` : extrae y retorna el elemento de máxima prioridad de la cola de prioridad `q`
- `b=q.is_empty()` : retorna verdadero si la cola de prioridad `q` está vacía, falso si no

## Implementación usando una lista desordenada

La implementación más simple consiste en mantener el conjunto como una lista desordenada. Agregar un elemento es trivial, pero encontrar el máximo requiere tiempo $\Theta(n)$.

La lista se puede mantener ya sea en un arreglo o en una lista enlazada. Para nuestro ejemplo, utilizaremos una lista enlazada con cabecera.

In [57]:
```
class NodoLista:
    def __init__(self,info,sgte=None):
        self.info=info
        self.sgte=sgte
class Cola_de_prioridad:
    def __init__(self):
        self.cabecera=NodoLista(0)
    def insert(self,x):
        self.cabecera.sgte=NodoLista(x,self.cabecera.sgte)
    def extract_max(self):
        assert self.cabecera.sgte is not None
        p=self.cabecera # apunta al previo del candidato a máximo
        r=self.cabecera.sgte
        while r.sgte is not None:
            if r.sgte.info>p.sgte.info:
                p=r
            r=r.sgte
        x=p.sgte.info # anotamos el valor máximo
        p.sgte=p.sgte.sgte # eliminamos el nodo con el máximo
        return x
    def is_empty(self):
        return self.cabecera.sgte is None
```

In [58]:
```
a=Cola_de_prioridad()
a.insert(45)
a.insert(12)
a.insert(30)
print("max=",a.extract_max())
a.insert(20)
print("max=",a.extract_max())
```

```
max= 45
max= 30
```

## Implementación usando una lista ordenada

Una implementación un poco más compleja consiste en mantener el conjunto como una lista ordenada. Agregar un elemento requiere recorrer la lista para encontrar el punto de inserción (tiempo $\Theta(n)$ en el peor caso), pero encontrar el máximo y extrarlo toma tiempo constante si la lista se ordena de mayor a menor.

In [59]:
```
class NodoLista:
    def __init__(self,info,sgte=None):
        self.info=info
        self.sgte=sgte
class Cola_de_prioridad:
    def __init__(self):
        self.cabecera=NodoLista(0)
    def insert(self,x):
        p=self.cabecera # al final apuntará al previo del punto de inser
ción
        while p.sgte is not None and x<p.sgte.info:
            p=p.sgte
        p.sgte=NodoLista(x,p.sgte)
    def extract_max(self):
        assert self.cabecera.sgte is not None
        x=self.cabecera.sgte.info # anotamos el valor máximo que está en
el primer nodo
        self.cabecera.sgte=self.cabecera.sgte.sgte # eliminamos el prime
r nodo
        return x
    def is_empty(self):
        return self.cabecera.sgte is None
```

In [60]:
```
a=Cola_de_prioridad()
a.insert(45)
a.insert(12)
a.insert(30)
print("max=",a.extract_max())
a.insert(20)
print("max=",a.extract_max())
```

```
max= 45
max= 30
```

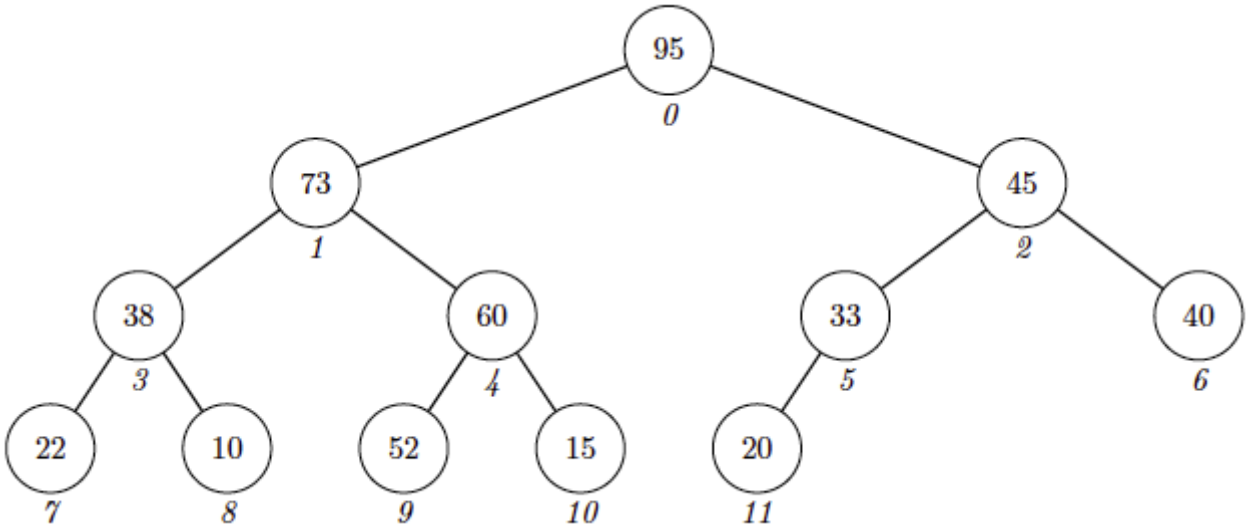
Las dos implementaciones que hemos visto están en extremos opuestos desde el punto de vista de su eficiencia. En ambas, una oparación toma tiempo constante ($\Theta(1)$) mientras la otra toma tiempo lineal ($\Theta(n)$).

Veremos a continuación que es posible diseñar una estructura que equilibre de mejor forma el costo de las dos operaciones.

## Implementación usando un *Heap*

Un heap es un árbol binario de una forma especial, que permite su almacenamiento sin usar punteros.

Este árbol se caracteriza porque tiene todos sus niveles llenos, excepto posiblemente el último, y en ese último nivel, los nodos están lo más a la izquierda posible.




Un árbol que cumpla esta condición diremos que tiene "forma de heap".

Los números bajo cada nodo corresponde a una numeración por niveles, y esa numeración se utiliza para almacenar cada elemento en el casillero respectivo de un arreglo:

| 95 | 73 | 45 | 38 | 60 | 33 | 40 | 22 | 10 | 52 | 15 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* |

Este arreglo contiene toda la información necesaria para representar al árbol. En efecto, tenemos que la raíz está en el casillero $0$, y además

$$\text{hijos del nodo } j = \{2j+1, 2j+2\}$$
$$\text{padre del nodo } k = \left\lfloor \frac{k-1}{2} \right\rfloor$$

Si hay $n$ casilleros ocupados en el arreglo, cualquier subíndice que sea mayor o igual a $n$ corresponde a un nodo inexistente.

Un heap puede utilizarse para implementar una cola de prioridad almacenando los datos de modo que las llaves estén siempre ordenadas de arriba a abajo (a diferencia de un árbol de búsqueda binaria, que ordena sus llaves de izquierda a derecha). En otras palabras, el padre debe tener siempre mayor prioridad que sus hijos. Un árbol que cumple esta condición diremos que tiene "orden de heap".

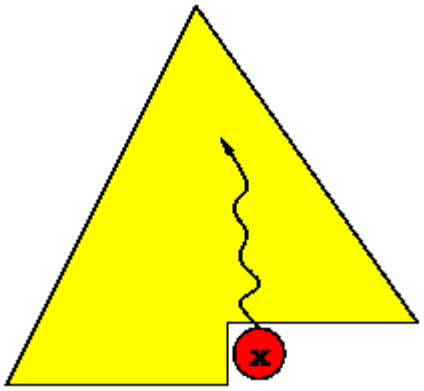
Por lo tanto, un heap debe satisfacer un invariante consistente en dos condiciones:

- Condición estructural: el árbol debe tener "forma de heap"
- Condición de orden: el árbol debe tener "orden de heap"

### Inserción

La inserción se realiza agregando el nuevo elemento en la primera posición libre del heap, esto es, el próximo nodo que debería aparecer en el recorrido por niveles o, equivalentemente, un casillero que se agrega al final del arreglo.

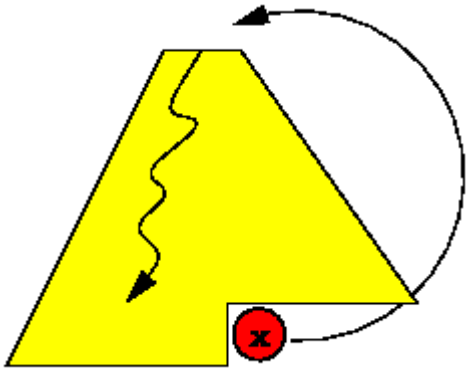
Después de agregar este elemento, la condición estructural se cumple, pero la condición de orden no tiene por qué cumplirse. Para resolver este problema, si el nuevo elemento es mayor que su padre, se intercambia con él, y ese proceso se repite mientras sea necesario. Una forma de describir esto es diciendo que el nuevo elemento "trepa" en el árbol hasta alcanzar el nivel correcto según su prioridad.




Como la altura del árbol es $\log_2 n$ y en cada nivel se hace un trabajo constante, el tiempo que demora esta operación en el peor caso es $\Theta(\log n)$.

### Extracción del máximo

El máximo evidentemente está en la raíz del árbol (casillero 0 del arreglo). Al sacarlo de ahí, podemos imaginar que ese lugar queda vacante. Para llenarlo, tomamos al último elemento del heap y lo trasladamos al lugar vacante. En caso de que no esté bien ahí de acuerdo a su prioridad (¡que es lo más probable!), lo hacemos descender intercambiándolo siempre con el mayor de sus hijos. Decimos que este elemento "se hunde" hasta su nivel de prioridad.




Esta operación también demora un tiempo proporcional a la altura del árbolen el peor caso, esto es, $\Theta(\log n)$.

In [65]:
```
def trepar(a,j): # El elemento a[j] trepa hasta su nivel de prioridad
    while j>=1 and a[j]>a[(j-1)//2]:
        (a[j],a[(j-1)//2])=(a[(j-1)//2],a[j]) # intercambiamos con el pa
dre
        j=(j-1)//2 # subimos al lugar del padre

def hundir(a,j,n): # El elemento a[j] se hunde hasta su nivel de priorid
ad
    while 2*j+1<n: # mientras tenga al menos 1 hijo
        k=2*j+1 # el hijo izquierdo
        if k+1<n and a[k+1]>a[k]: # el hijo derecho existe y es mayor
            k+=1
        if a[j]>=a[k]: # tiene mejor prioridad que ambos hijos
            break
        (a[j],a[k])=(a[k],a[j]) # se intercambia con el mayor de los hij
os
        j=k # bajamos al lugar del mayor de los hijos

class Heap:
    def __init__(self,maxn=100):
        self.a=np.zeros(maxn)
        self.n=0
    def insert(self,x):
        assert self.n<len(self.a)
        self.a[self.n]=x
        trepar(self.a,self.n)
        self.n+=1
    def extract_max(self):
        x=self.a[0] # esta variable lleva el máximo, el casillero 0 qued
a vacante
        self.n-=1   # achicamos el heap
        self.a[0]=self.a[self.n] # movemos el elemento sobrante hacia el
casillero vacante
        hundir(self.a,0,self.n)
        return x
    def imprimir(self):
        print(self.a[0:self.n])
```

Para poder visualizar el efecto de cada operación, agregamos una operación `imprimir` que muestra el contenido del arreglo.

In [66]:
```
a=Heap(10)
a.insert(45)
a.imprimir()
a.insert(12)
a.imprimir()
a.insert(30)
a.imprimir()
print("max=",a.extract_max())
a.imprimir()
a.insert(20)
a.imprimir()
print("max=",a.extract_max())
a.imprimir()
```

```
[45.]
[45. 12.]
[45. 12. 30.]
max= 45.0
[30. 12.]
[30. 12. 20.]
max= 30.0
[20. 12.]
```

## Ordenando con una cola de prioridad

Las colas de prioridad tienen múltiples aplicaciones, algunas de las cuales veremos más adelante en este curso.

Una de las aplicaciones más importantes es para resolver el problema de la ordenación. Dada cualquier implementaciónde una cola de prioridad, se la puede utilizar para construir un algoritmo de ordenación, de la siguiente manera:

- Crear una cola de prioridad vacía e insertar en ella todos los elementos del conjunto a ordenar
- Luego ir extrayendo máximos sucesivamente. Los elementos irán saliendo de mayor a menor.

Para las dos primeras implementaciones de colas de prioridad que vimos (conjunto desordenado y conjunto ordenado) el algoritmo resultate demora tiempo $\Theta(n^2)$ en el peor caso (y correspone a lso algoritmos de ordenación por selección y ordenación por inserción, respectivamente.

En cambio, si se utiliza un heap, el algoritmo resultante demora tiempo $\Theta(n \log n)$ en el peor caso y se llama *Heapsort*. A continuación veremos una versión de Heapsort construída de acuerdo a estas ideas, y más adelante en el curso volveremos sobre el tema, porque es posible optimizar aspectos importantes del algoritmo.

In [67]:
```
def Heapsort(a): # Versión preliminar
    n=len(a)
    h=Heap(n)
    # Fase 1: insertamos los elementos en un heap
    for k in range(0,n):
        h.insert(a[k])
    # Fase 2: extraemos el máximo sucesivamente
    for k in range(n-1,-1,-1):
        a[k]=h.extract_max()
```

In [68]:
```
a = np.random.random(6)
print(a)
Heapsort(a)
print(a)
```

```
[0.84809569 0.60699999 0.8400833  0.76749145 0.18499666 0.09499075]
[0.09499075 0.18499666 0.60699999 0.76749145 0.8400833  0.84809569]
```

In [ ]: