Estructuras de datos elementales

Los sistemas o métodos de organización de datos que permiten un almacenamiento eficiente de la información en la memoria del computador son conocidos como estructuras de datos. Estos métodos de organización constituyen las piezas básicas para la construcción de algoritmos complejos, y permiten implementarlos de manera eficiente.

En el presente capítulo se presentan las estructuras de datos básicas como son arreglos, listas enlazadas y árboles, con las cuales se implementarán posteriormente los *tipos de datos abstractos*.

Arreglos

Un arreglo es una secuencia contigua en memoria, que almacena un número fijo de elementos homogéneos. En la siguiente figura se muestra un arreglo de enteros con 10 elementos:

80	45	2	21	92	17	5	65	14	34
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

Una ventaja que tienen los arreglos es que el costo de acceso a un elemento dado del arreglo es constante, es decir no hay diferencias de costo entre accesar el primer, el último o cualquier elemento del arreglo, lo cual es muy eficiente. La desventaja es que es necesario definir a priori el tamaño del arreglo, lo cual puede generar mucha pérdida de espacio en memoria si se definen arreglos muy grandes para contener conjuntos pequeños de elementos.

Esta característica de costo de acceso constante es esencial para la eficiencia de algunos algoritmos muy importantes, como por ejemplo el siguiente:

Ejemplo: Búsqueda Binaria

Supongamos que queremos buscar un elemento x en un arreglo a de tamaño n. Si no tenemos más información sobre el orden de los elementos dentro del arreglo, lo único que podemos hacer es una *búsqueda secuencial*, la cual tiene costo $\Theta(n)$ tanto en el peor caso como en el caso promedio.

Pero si sabemos que los elementos están en orden ascendente, existe una forma mucho más eficiente, llamada *búsqueda binaria*.

La idea es comparar primero x conta el elemento del centro del arreglo. Si tenemos suerte, lo encontramos ahí, pero incluso si no tenemos suerte, podemos de inmediato descartar la mitad del arreglo. En efecto, si x es mayor que el elemento del centro, entonces basta seguir buscando en la segunda mitad. De la misma manera, si x es menor, basta seguir buscando en la primera mitad.

In [1]:

```python
import numpy as np
a=np.array([12,25,29,34,45,53,59,67,86,92])
```

In [2]:

```python
# Búsqueda binaria, versión recursiva
# busca x en el arreglo a, retorna subíndice o -1 si no está
def bbin(x,a):
    # Definimos una función auxiliar para
    # buscar en el subarreglo a[i],...,a[j]
    def bbin_rec(x,a,i,j):
        if i>j:
            return -1
        k=(i+j)//2
        if x==a[k]:
            return k
        if x<a[k]:
            return bbin_rec(x,a,i,k-1)
        else:
            return bbin_rec(x,a,k+1,j)

    # puntapié inicial
    n=len(a)
    return bbin_rec(x,a,0,n-1)
```

In [3]:

```python
print(bbin(12,a), bbin(53,a), bbin(92,a), bbin(30,a))
```

```
0 5 9 -1
```

In [4]:

```
# Búsqueda binaria, versión iterativa
# busca x en el arreglo a, retorna subíndice o -1 si no está
def bbin(x,a):
    n=len(a)
    i=0
    j=n-1
    while i<=j:
        k=(i+j)//2
        if x==a[k]:
            return k
        if x<a[k]:
            j=k-1
        else:
            i=k+1
    return -1
```

In [5]:

```
print(bbin(12,a), bbin(53,a), bbin(92,a), bbin(30,a))
```

```
0 5 9 -1
```

Podemos estimar rápidamente la eficiencia de este algoritmo si vemos que para hacer una búsqueda en un conjunto de tamaño n, después de accesar el elemento del medio, en el peor caso continuamos buscando en un conjunto de tamaño aproximadamente igual a la mitad:

$$T(n) = 1 + T\left(\frac{n}{2}\right)$$

Aplicando el Teorema Maestro con $p = 1$, $q = 2$, $r = 1$, vemos que $T(n) = \Theta(\log n)$. Por lo tanto, gracias a que el arreglo está ordenado, una búsqueda binaria es mucho más eficiente que una búsqueda secuencial.

La ecuación anterior nos permite obtener una estimación rápida, pero no refleja de manera totalmente exacta lo que ocurre en el algoritmo. Si queremos modelar de manera precisa lo que ocurre en el peor caso, la ecuación correcta es

$$\begin{aligned} T(n) &= 1 + T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) \\ T(1) &= 1 \end{aligned}$$

donde la notación "techo" $\lceil x \rceil$ denota el menor entero mayor o igual a x (y, similarmente, la notación "piso" $\lfloor x \rfloor$ denota el mayor entero menor o igual a x).

Si tabulamos el valor de la función $T(n)$ para los primeros valores de n, tenemos:

n	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
$T(n)$	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	5	5

Observando esta tabla, no es difícil adivinar la solución:

$$T(n) = \log_2{(n+1)}$$

Ejercicio:

Demostrarlo.

Una manera más eficiente de programar la búsqueda binaria

En el análisis anterior, hemos considerado que en cada iteración, el costo de accesar el elemento $a[k]$ es igual a 1, representando así el costo total de comparar primero con == y luego con < . Si quisiéramos hacer una contabilidad más precisa, deberíamos decir que ese costo es en realidad de 2 comparaciones por cada iteración. A continuación veremos que es posible reducir eso a 1 comparación por ciclo, si utilizamos comparaciones de tipo <= :

In [6]:

```
# Búsqueda binaria, versión iterativa y con <=
# busca x en el arreglo a, retorna subíndice o -1 si no está
def bbin(x,a):
    n=len(a)
    i=0
    j=n-1
    while i<j: # conjunto tiene al menos 2 elementos
        k=(i+j)//2
        if x<=a[k]:
            j=k    # x estaría en a[i],...,a[k]
        else:
            i=k+1  # x estaría en a[k+1],...,a[j]
    # al terminar, el conjunto factible se ha reducido a 0 o 1 elemento
    if i==j and x==a[i]:
        return i
    else:
        return -1
```

In [7]:

```
print(bbin(12,a), bbin(53,a), bbin(92,a), bbin(30,a))
```

```
0 5 9 -1
```

En esta versión logramos ahorrar una comparación de elementos por iteración, al precio de que toda las búsquedas ahora hacen el máximo de iteraciones, a diferencia del algoritmo original, en donde si teníamos suerte el algoritmo buscado se podría encontrar en las primeras iteraciones.

Este es un precio que vale la pena pagar, porque en el algoritmo original son muy pocos los casos en que la búsqueda termina tempranamente, y en la gran mayoría de los casos igual se hace un número de iteraciones muy cercano al máximo.

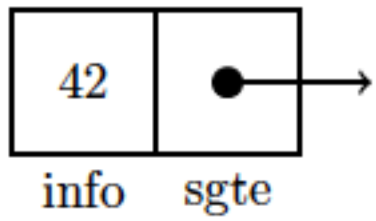
Estructuras enlazadas

Como hemos visto, los arreglos permiten que algunos algoritmos se puedan programar de manera muy eficiente, pero las estructuras basadas en arreglos suelen ser muy rígidas. Por ejemplo, si quisiéramos agregar un nuevo elemento al arreglo ordenado en que se hace búsqueda binaria (suponiendo que hubiera holgura suficiente), la inserción tomaría tiempo $\Theta(n)$ tanto en el peor caso como en promedio, por la necesidad de preservar el orden de los elementos.

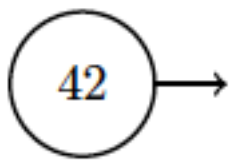
Veremos a continuación que podemos diseñar estructuras mucho más flexibles sin hacemos uso de la capacidad de definir clases de objetos que contienen dentro de sus campos referencias a otros objetos.

Listas de enlace simple

Comenzaremos viendo la estructura más sencilla de este tipo: una secuencia de nodos, en que cada uno contiene una referencia al siguiente de la lista. Consideremos nodos compuestos de dos *campos* (o *atributos*): `info` y `sgte`. El primero almacena el elemento de la secuencia, y el segundo apunta al siguiente nodo. Por ejemplo, un nodo que almacena el valor 42 y que apunta al siguiente nodo se puede representar gráficamente así:



O, más simplemente:



Para definir el formato de estos nodos utilizaremos la siguiente definición de clase, la que incluye un constructor para inicializar sus campos al crear un objeto:

In [42]:

```
class Nodo:
    def __init__(self, info, sgte=None):
        self.info=info
        self.sgte=sgte
```

In [43]:

```
p=Nodo(42)
print(p.info, p.sgte)
```

```
42 None
```

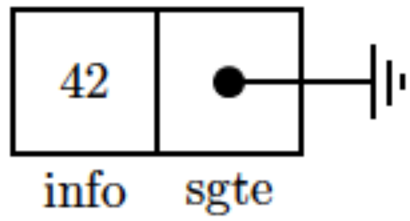
El siguiente trozo de programa muestra la construcción de una lista con 4 elementos: $42, 65, 13$ y 44, y un ejemplo simple de uso:

In [44]:

```python
primero=Nodo(42,Nodo(65,Nodo(13,Nodo(44))))
p=primero
while p is not None:
    print(p.info, end=" ")
    p=p.sgte
print()
```

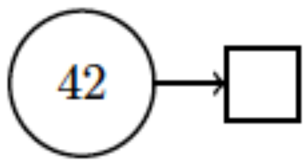
```
42 65 13 44
```

Algo adicional respecto de la representación gráfica. Cuando una referencia es nula (`None`), es tradicional representarla como "conectada a tierra":

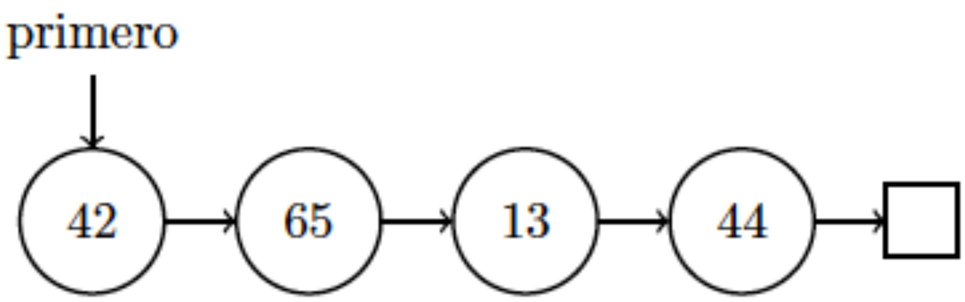


Al usar la representación con nodos circulares, la ausencia de un nodo siguiente la podemos representar simplemente por la ausencia de la flecha saliente:

O, si queremos hacer explícita la ausencia de un nodo siguiente, podemos representarlo por un nodo cuadrado, que es una convención que nos resultará muy conveniente más adelante, al ver *árboles*:



Con esta última convención, la lista que construimos en el ejemplo anterior, se visualizaría así:



A continuación definiremos una clase `Lista` , que contendrá el puntero al primer nodo de la lista, así como la funcionalidad que necesitamos para operar sobre la lista:

In [45]:

```python
class Lista:
    def __init__(self):
        self.primero=None

    def insertar_al_inicio(self,info):
        self.primero=Nodo(info,self.primero)

    def insertar_despues_de(self,p,info): # inserta después de nodo p
        p.sgte=Nodo(info,p.sgte)

    def eliminar_al_inicio(self):
        assert self.primero is not None
        self.primero=self.primero.sgte

    def eliminar_sgte_de(self,p): # elimina el nodo siguiente de p
        assert p.sgte is not None
        p.sgte=p.sgte.sgte

    def k_esimo(self,k): # retorna k-esimo nodo, o None si fuera de rango
        p=self.primero
        j=1
        while p is not None:
            if j==k:
                return p
            p=p.sgte
            j+=1
        return None

    def imprimir(self):
        p=self.primero
        while p is not None:
            print(p.info, end=" ")
            p=p.sgte
        print()
```

In [46]:

```python
L=Lista()
L.insertar_al_inicio(44)
L.insertar_al_inicio(13)
L.insertar_al_inicio(65)
L.insertar_al_inicio(42)
L.imprimir()
```

```
42 65 13 44
```

In [47]:

```
L.insertar_despues_de(L.k_esimo(2),88)
L.imprimir()
```

```
42 65 88 13 44
```

In [48]:

```
L.eliminar_al_inicio()
L.imprimir()
```

```
65 88 13 44
```

In [49]:

```
L.eliminar_sgte_de(L.k_esimo(1))
L.imprimir()
```

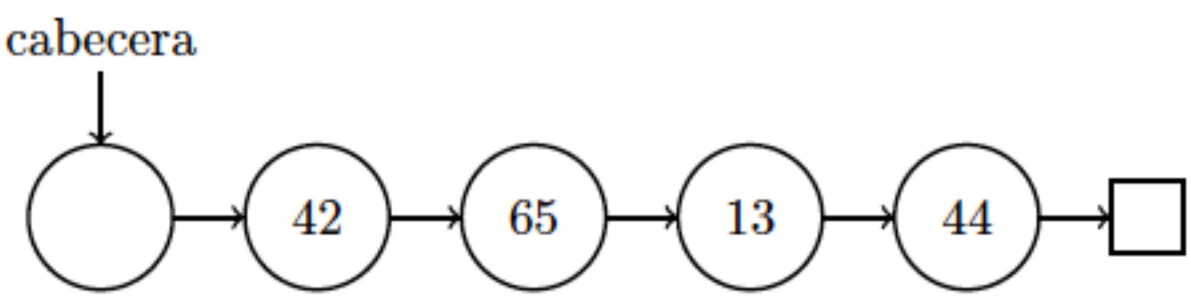
```
65 13 44
```

Hay algunas cosas que no resultan muy elegantes en este diseño.

Una de ellas es que para eliminar un elemento, no se pueda indicar al elemento que se desea eliminar, sino que haya que indicar al previo. Esto es inevitable, dado el caracter unidireccional de los enlaces, y más adelante, cuando veamos *listas de doble enlace* veremos que eso puede mejorarse.

Otro punto molesto en la interfaz de uso es la necesidad de distinguir entre si se opera al comienzo de la lista, o en un punto interior. Esto es porque las operaciones afectan al elemento previo, y el primero de la lista, por definición, no tiene un elemento previo.

Esto puede subsanarse, sin embargo, introduciendo un nodo "cabecera" ("*header*") al comienzo de la lista. Este nodo no contiene información útil y para todos los efectos es como si no existiera, excepto que sirve como el previo del primer nodo real de la lista. Para poder ubicarlo, lo identificaremos con el nodo "0-ésimo" de la lista.



A continuación reescribimos nuestra definición de la clase `Lista` y sus ejemplos de uso, bajo el supuesto de que existe un nodo cabecera.

In [50]:

```
class Lista_con_cabecera:
    def __init__(self):
        self.cabecera=Nodo(0,None)

    def insertar_despues_de(self,p,info): # inserta después de nodo p
        p.sgte=Nodo(info,p.sgte)

    def eliminar_sgte_de(self,p): # elimina el nodo siguiente de p
        assert p.sgte is not None
        p.sgte=p.sgte.sgte

    def k_esimo(self,k): # retorna k-esimo nodo, o None si fuera de rango
        p=self.cabecera
        j=0
        while True:
            if j==k:
                return p
            p=p.sgte
            if p is self.cabecera:
                return None
            j+=1

    def imprimir(self):
        p=self.cabecera.sgte
        while p is not None:
            print(p.info,end=" ")
            p=p.sgte
        print()
```

In [51]:

```
L=Lista_con_cabecera()
L.insertar_despues_de(L.k_esimo(0),42)
L.insertar_despues_de(L.k_esimo(1),65)
L.insertar_despues_de(L.k_esimo(2),13)
L.insertar_despues_de(L.k_esimo(3),44)
L.imprimir()
```

```
42 65 13 44
```

In [52]:

```
L.eliminar_sgte_de(L.k_esimo(0)) # eliminar el primero
L.imprimir()
```

```
65 13 44
```

In [53]:

```
L.eliminar_sgte_de(L.k_esimo(1)) # eliminar un elemento en el medio
L.imprimir()
```

```
65 44
```

Recorriendo la lista con un iterador

A continuación veremos cómo, en lugar de imprimir la lista, podemos implementar un iterador que vaya entregando los elementos de la lista cada vez que es llamado:

In [54]:

```
class Lista_con_cabecera:
    def __init__(self):
        self.cabecera=Nodo(0,None)

    def insertar_despues_de(self,p,info): # inserta después de nodo p
        p.sgte=Nodo(info,p.sgte)

    def eliminar_sgte_de(self,p): # elimina el nodo siguiente de p
        assert p.sgte is not None
        p.sgte=p.sgte.sgte

    def k_esimo(self,k): # retorna k-esimo nodo, o None si fuera de rango
        p=self.cabecera
        j=0
        while p is not None:
            if j==k:
                return p
            p=p.sgte
            j+=1
        return None

    def valores(self):
        p=self.cabecera.sgte
        while p is not None:
            yield p.info
            p=p.sgte
```

In [56]:

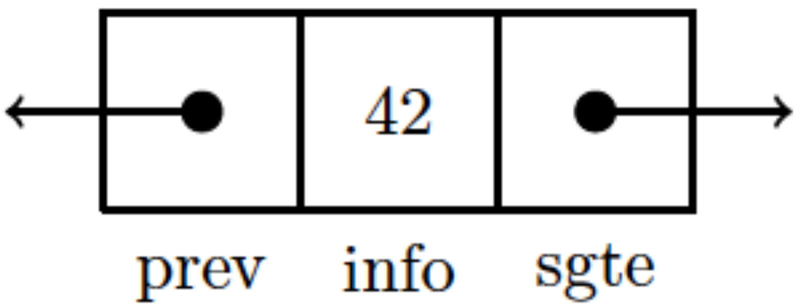
```
L=Lista_con_cabecera()
L.insertar_despues_de(L.k_esimo(0),42)
L.insertar_despues_de(L.k_esimo(1),65)
L.insertar_despues_de(L.k_esimo(2),13)
L.insertar_despues_de(L.k_esimo(3),44)
for x in L.valores():
    print(x, end=" ")
print()
```

```
42 65 13 44
```

Listas de doble enlace

Las listas de enlace simple permiten solo procesos unidireccionales, por lo que no son muy apropiadas cuando los procesos necesitan poder recorrerlas en ambas dicecciones.

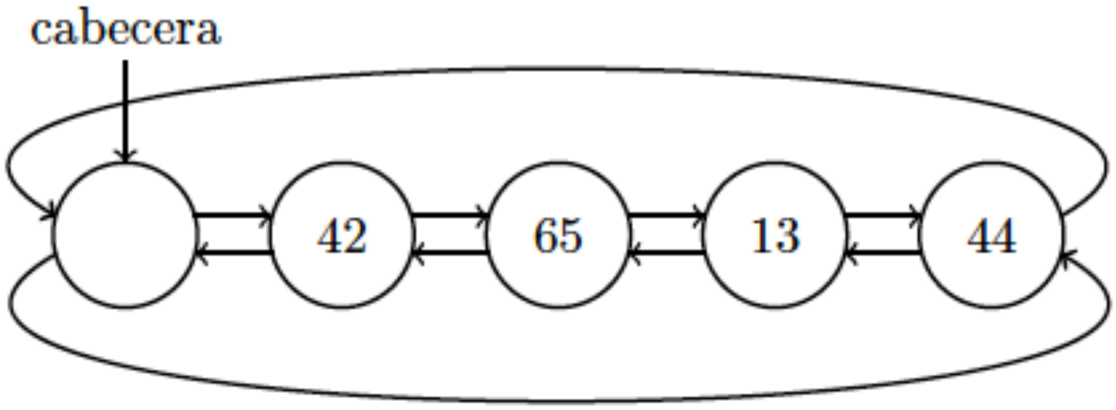
Podemos mejorar esto si agregamos a los nodos una referencia al nodo *previo*, además del nodo siguiente:



In [57]:

```
class Nodo:
    def __init__(self, prev, info, sgte):
        self.prev=prev
        self.info=info
        self.sgte=sgte
```

Con este tipo de nodos podemos formar una lista que puede ser recorrida en ambas direcciones. Por consideraciones similares a las anteriores, resulta conveniente agregar un nodo cabecera en cada extremo, pero en realidad un mismo nodo puede jugar ambos roles, con lo cual la lista adopta un aspecto físicamente circular, aunque desde un punto de vista conceptual no lo sea:



La siguiente es una definición de lista de doble enlace, cun alguna de la funcionalidad que ella permite:

In [58]:

```python
class Lista_doble_enlace:
    def __init__(self):
        self.cabecera=Nodo(None,0,None)
        self.cabecera.prev=self.cabecera
        self.cabecera.sgte=self.cabecera

    def insertar_despues_de(self,p,info): # inserta después de nodo p
        r=p.sgte
        p.sgte=r.prev=Nodo(p,info,r)

    def eliminar(self,p): # elimina el nodo p
        assert p is not self.cabecera
        (p.prev.sgte,p.sgte.prev)=(p.sgte,p.prev)

    def k_esimo(self,k): # retorna k-esimo nodo, o None si fuera de rango
        p=self.cabecera
        j=0
        while True:
            if j==k:
                return p
            p=p.sgte
            if p is self.cabecera:
                return None
            j+=1

    def ascendente(self):
        p=self.cabecera.sgte
        while p is not self.cabecera:
            yield p.info
            p=p.sgte

    def descendente(self):
        p=self.cabecera.prev
        while p is not self.cabecera:
            yield p.info
            p=p.prev
```

In [59]:

```
L=Lista_doble_enlace()
L.insertar_despues_de(L.k_esimo(0),42)
L.insertar_despues_de(L.k_esimo(1),65)
L.insertar_despues_de(L.k_esimo(2),13)
L.insertar_despues_de(L.k_esimo(3),44)
for x in L.ascendente():
    print(x,end=" ")
print()
for x in L.descendente():
    print(x,end=" ")
print()
```

```
42 65 13 44
44 13 65 42
```

In [60]:

```
L.eliminar(L.k_esimo(3))
for x in L.ascendente():
    print(x,end=" ")
print()
```

```
42 65 44
```

In [ ]:

```

```