6 Diccionarios

El TDA Diccionario es uno de los más usados en la práctica, y se conocen mucha formas distintas de implementarlo.

Un Diccionario es un conjunto de n elementos, cada uno de los cuales tiene un campo que permite identificarlo de manera única (ese campo se llama su *llave primaria*), sobre el cual están definidas las operaciones de buscar, insertar, eliminar, y ocasionalmente otras que definiremos má adelante. Más precisamente, si d es un diccionario, existirán las operaciones:

- `r=d.search(x)` : buscar el elemento de llave `x` , retornar un resultado que permita ubicarlo, o `None` si no está
- `d.insert(x)` : insertar un elemento de llave `x` , evitando crear una llave duplicada
- `d.delete(x)` : eliminar el elemento de llave `x` , el cual debe estar en el diccionario

Diccionarios de Python

El lenguaje Python posee un tipo `dict` que implementa la funcionalidad de diccionarios que hemos descrito (más operaciones adicionales). En un diccionario se busca por una llave y se obtiene un valor asociado.

In [11]:
```
distancia = {'Valparaíso':102, 'Concepción': 433, 'Arica': 1664, 'Puerto Montt': 912, 'Rancagua': 80}
```

La forma de buscar es simplemente usando la llave como subíndice:

In [12]:
```
print(distancia['Arica'])
```

```
1664
```

Y la forma de agregar una nueva llave es asignándole un valor:

In [13]:
```
distancia['Talca']=237
```

Al buscar una llave inexistente se produce una excepción:

```
In [14]: print(distancia['La Serena'])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-14-0078e5a2585d> in <module>
----> 1 print(distancia['La Serena'])

KeyError: 'La Serena'
```

Pero hay una forma de buscar sin que dé un error, sino que retorne `None` :

```
In [15]: print(distancia.get('Rancagua'), distancia.get('La Serena'))
```

```
80 None
```

Para eliminar un dato, se usa `pop` (lo elimina y retrna su valor):

```
In [16]: distancia.pop('Rancagua')
```

```
Out[16]: 80
```

Aparte de esto, hay muchas otras operaciones que permiten obtener la lista de todas las llaves, etc.

Dado que en Python ya existe una implementación de diccionarios, ¿por qué querríamos estudiar nosotros cómo implementarlos?

La respuesta está en que, si nosotros controlamos todos los detalles de una implementación, sabremos exactamente cuan eficiente es, y para qué tipo de aplicaciones es más apropiada. Lo último es particularmente importante, porque no hay ninguna implementación de diccionarios que sea uniformemente mejor que las otras para todas las aplicaciones.

Estudiaremos entonces cómo se puede implementar un diccionario, comenzando por las estrategias más sencillas, y avanzando hacia enfoques más sofisticados.

En nuestros ejemplos supondremos que solo almacenamos la llave, pero en la práctica siempre habrá información adicional asociada a cada llave. También por simplicidad a menudo usaremos llaves numéricas, aunque en la práctica es más frecuente que las llaves sean strings.

Búsqueda secuencial

La manera más simple de implementar un diccionario es con una lista desordenada de llaves, en la cual se hace búsqueda secuencial. La inserción es especialmente eficiente si obviamos chequear por duplicados, y la eliminación es eficiente una vez que sabemos dónde está la llave.

```
In [17]: import numpy as np
```

In [20]:
```
class Lista_secuencial:
    def __init__(self, size=100):
        self.a=np.zeros(size,dtype=int)
        self.n=0
    def insert(self,x):
        assert self.n<len(self.a)
        self.a[self.n]=x
        self.n+=1
    def search(self,x):
        for k in range(0,self.n):
            if self.a[k]==x:
                return k
        return None
    def delete(self,x):
        k=self.search(x)
        self.a[k]=self.a[self.n-1] # modemos el último al lugar vacante
        self.n-=1
```

In [22]:
```
d=Lista_secuencial()
d.insert(30)
d.insert(10)
d.insert(25)
print(d.search(10))
print(d.search(80))
d.delete(30)
print(d.search(30))
```

```
1
None
None
```

La búsqueda secuencial también se puede implementar con una lista enlazada, en cuyo caso será más simple insertar al inicio.

En cualquier caso, la búsqueda demora tiempo $\Theta(n)$. Para estimar el costo promedio, suponemos que todos los elementos son igualmente probables de ser accesados y que el costo de buscar a un elemento que es el k-ésimo de la lista es k. Por lo tanto, el costo promedio es

$$\frac{1}{n}\sum_{1\leq k\leq n} k = \frac{n+1}{2} = \Theta(n)$$

Por lo tanto, este tipo de implementación solo será adecuada para conjuntos muy pequeños.

Búsqueda secuencial con probabilidades de acceso no uniformes

En la práctica, es muy raro que las probabilidades de acceso a los elementos sean uniformes. Con frecuencia hay algunos elementos que son mucho más populares que otros, y empíricamente a menudo se observan distribuciones de tipo "ley de potencias", con probabilidades de tipo

$$p_k \propto \frac{1}{k^\alpha}$$

para algún α. Para el caso $\alpha = 1$ esto se llama Ley de Zipf.

Si un conjunto de datos tiene elementos con probabilidades de acceso diferentes, entonces para la búsqueda secuencial el orden en que estén los elementos en la lista hace una diferencia.

Caso 1. Probabilidades conocidas

Si las probabilidades de acceso son conocidas, es fácil ver que el orden óptimo es en orden decreciente de probabilidad.

Más precisamente, si los elementos son $X_1, X_2, \ldots, X_n$ con probabilidades de acceso $p_1, p_2, \ldots, p_n$ respectivamente, y si están ordenados de modo que $p_1 \geq p_2 \geq p_3 \geq \cdots$, entonces el costo esperado de búsqueda óptimo es

$$C_{OPT} = \sum_{1 \leq k \leq n} k p_k$$

Tomemos como ejemplo el capítulo 1 de "El Quijote" (en minúsculas y sin puntuación para simplificar su proceso), cuyo texto está en el archivo `cap1.txt` :

```
en un lugar de la mancha de cuyo nombre no quiero acordarme no ha mucho
tiempo que vivía un hidalgo de los de lanza en astillero adarga antigua
...
peregrino y significativo como todos los demás que a él y a sus cosas
había puesto
```

Acceso al archivo desde Colab

Si este notebook está usándose en Google Colab, el archivo se debe almacenar en la carpeta `Colab Notebooks` de Google Drive, y para que el código en Python pueda tener acceso a él se debe quitar los comentarios y ejecutar la siguiente celda:

In [150]:
```
#from google.colab import drive
#drive.mount("/content/gdrive")
#%cd "/content/gdrive/My Drive/Colab Notebooks/"
```

El costo óptimo para un archivo dado se puede obtener con el siguiente código en Python, en el cual hacemos uso de los diccionarios provistos por el lenguaje:

In [160]:
```
def calcula_costo_optimo(archivo): # lee el archivo, calcula frecuencias en orden descendente
    f=open(archivo,"r")
    texto=f.read()
    palabras=texto.split()
    frec={}
    for x in palabras:
        frec[x] = 1 if not x in frec else frec[x]+1
    lista=[]
    Copt=0
    costo=0
    for x in sorted(frec,key=frec.get,reverse=True):
        costo+=1
        Copt+=costo*frec[x]
        lista.append((frec[x],x))
    Copt/=len(palabras)
    f.close()
    return (Copt,lista)
```

Como resultado, mostramos el costo esperado de búsqueda en una lista ordenada de manera óptima (C_OPT) y las palabras más frecuentes:

In [170]:
```
(c,L)=calcula_costo_optimo("cap1.txt")
print('C_OPT={:6.2f}\n'.format(c))
for k in range(0,9):
    print(L[k])
```

```
C_OPT=157.81

(120, 'de')
(105, 'y')
(88, 'que')
(44, 'a')
(40, 'el')
(38, 'en')
(35, 'su')
(33, 'la')
(30, 'se')
```

Caso 2: Probabilidades desconocidas

Cuando las probabilidades son desconocidas, existen estrategias adaptativas, que van reordenando la lista dinámicamente a medida que los elementos son buscados, de modo de tratar de aproximar el orden óptimo. Hay dos técnicas que dan buenos resultados: "traspose" (TR) y "move to front" (MTF).

Transpose

Esta técnica consiste en que cada vez que un elemento es accesado, se le mueve un lugar más adelante en la lista (a menos que ya esté en el primer lugar). Esto se puede implementar ya sea en un arreglo o en una lista enlazada. En la siguiente implementación usaremos una lista enlazada con cabecera.

Si un elemento no se encuentra, simulamos como si hubiese estado al final de la lista. Para esto, mantendremos siempre disponible un nodo extra al final de la lista, en donde almacenaremos tentativamente la llave de búsqueda. Si finalmente se encuentra en ese nodo, se le incorpora a la lista y se crea un nuevo nodo extra.

Para contabilizar el costo, el método `search` retorna el número de comparaciones de llaves que se hizo en la búsqueda.

In [129]:
```
class NodoLista:
    def __init__(self,info,sgte=None):
        self.info=info
        self.sgte=sgte

class Lista_TR:
    def __init__(self):
        self.extra=NodoLista(0)
        self.cabecera=NodoLista(0,self.extra)

    def search(self,x): # busca x (si no está lo inserta al final) y lo
 adelanta un lugar
                        # retorna el costo de búsqueda
        self.extra.info=x # agregamos x al final, en caso que no estuvie
ra antes
        p=self.cabecera
        q=p.sgte

        # k cuenta el número de comparaciones de llaves
        if q.info==x: # x ya está primero en la lista, no hacemos nada
            k=1
        else:
            # buscamos del segundo en adelante
            r=q.sgte
            k=2
            while r.info!=x:
                (p,q,r)=(q,r,r.sgte)
                k+=1
            # r apunta al elemento buscado, lo movemos un lugar hacia ad
elante
            (p.sgte,q.sgte,r.sgte)=(r,r.sgte,q)
        if q.sgte is None: # se utilizó el nodo extra, agregamos uno nue
vo
            self.extra=NodoLista(0)
            q.sgte=self.extra
        return k

    def imprimir(self):
        p=self.cabecera.sgte
        print("[",end=" ")
        while p is not self.extra:
            print(p.info,end=" ")
            p=p.sgte
        print("]")
```

In [130]:
```
def test(Lista_adaptativa): # test interactivo
    a=Lista_adaptativa()
    while True:
        x=input("x=")
        if x=="fin":
            return
        print("costo=",a.search(x),end=" ")
        a.imprimir()
```

In [131]:
```
test(Lista_TR)
```

```
x=hola
costo= 1 [ hola ]
x=chao
costo= 2 [ chao hola ]
x=casa
costo= 3 [ chao casa hola ]
x=hola
costo= 3 [ chao hola casa ]
x=hola
costo= 2 [ hola chao casa ]
x=hola
costo= 1 [ hola chao casa ]
x=gato
costo= 4 [ hola chao gato casa ]
x=fin
```

In [171]:
```
def procesa(archivo,Lista_adaptativa): # lee el archivo y calcula costo
 promedio de búsqueda
    f=open(archivo,"r")
    texto=f.read()
    palabras=texto.split()
    npalabras=0
    costo_acum=0
    a=Lista_adaptativa()
    for x in palabras:
        costo_acum+=a.search(x)
        npalabras+=1
    print("Costo promedio de búsqueda= {:6.2f}".format(costo_acum/npalab
ras))
    f.close()
```

In [172]:
```
procesa("cap1.txt",Lista_TR)
```

```
Costo promedio de búsqueda= 208.74
```

Move-To-Front

Esta técnica consiste en que cada vez que un elemento es accesado, se le mueve al primer lugar de la lista (a menos que ya esté en el primer lugar). Si un elemento no se encuentra, simulamos como si hubiese estado al final de la lista.

In [138]:
```
class Lista_MTF:
    def __init__(self):
        self.extra=NodoLista(0)
        self.cabecera=NodoLista(0,self.extra)

    def search(self,x): # busca x (si no está lo inserta al final) y lue
go lo mueve al primer lugar
        # retorna el costo de búsqueda
        self.extra.info=x # agregamos x al final, en caso que no estuvie
ra antes
        p=self.cabecera
        q=p.sgte
        k=1 # cuenta el número de comparaciones de llaves
        while q.info!=x:
            (p,q)=(q,q.sgte)
            k+=1
        if q.sgte is None: # se utilizó el nodo extra, agregamos uno nue
vo
            self.extra=NodoLista(0)
            q.sgte=self.extra
        if k>1:  # x no está primero, move to front
            (self.cabecera.sgte,p.sgte,q.sgte)=(q,q.sgte,self.cabecera.s
gte)
        return k

    def imprimir(self):
        p=self.cabecera.sgte
        print("[",end=" ")
        while p is not self.extra:
            print(p.info,end=" ")
            p=p.sgte
        print("]")
```

In [139]:
```
test(Lista_MTF)
```

```
x=hola
costo= 1 [ hola ]
x=casa
costo= 2 [ casa hola ]
x=chao
costo= 3 [ chao casa hola ]
x=hola
costo= 3 [ hola chao casa ]
x=casa
costo= 3 [ casa hola chao ]
x=casa
costo= 1 [ casa hola chao ]
x=gato
costo= 4 [ gato casa hola chao ]
x=fin
```

In [173]:
```
procesa("cap1.txt",Lista_MTF)
```

```
Costo promedio de búsqueda= 188.82
```

En resumen, tenemos que para este texto en particular, el costo óptimo es 157.81, el costo promedio de TR es 208.74 y el de MTF es 188.82.

Si en lugar de considerar un caso se analiza matemáticamente el caso general, suponiendo que los accesos llegan independientemente siguiendo la distribución dada y que el algoritmo corre durante un tiempo que tiende a infinito, se puede demostrar que

$$C_{OPT} \leq C_{TR} \leq C_{MTF} \leq \frac{\pi}{2} C_{OPT}$$

Búsqueda en un arreglo ordenado: Búsqueda Binaria

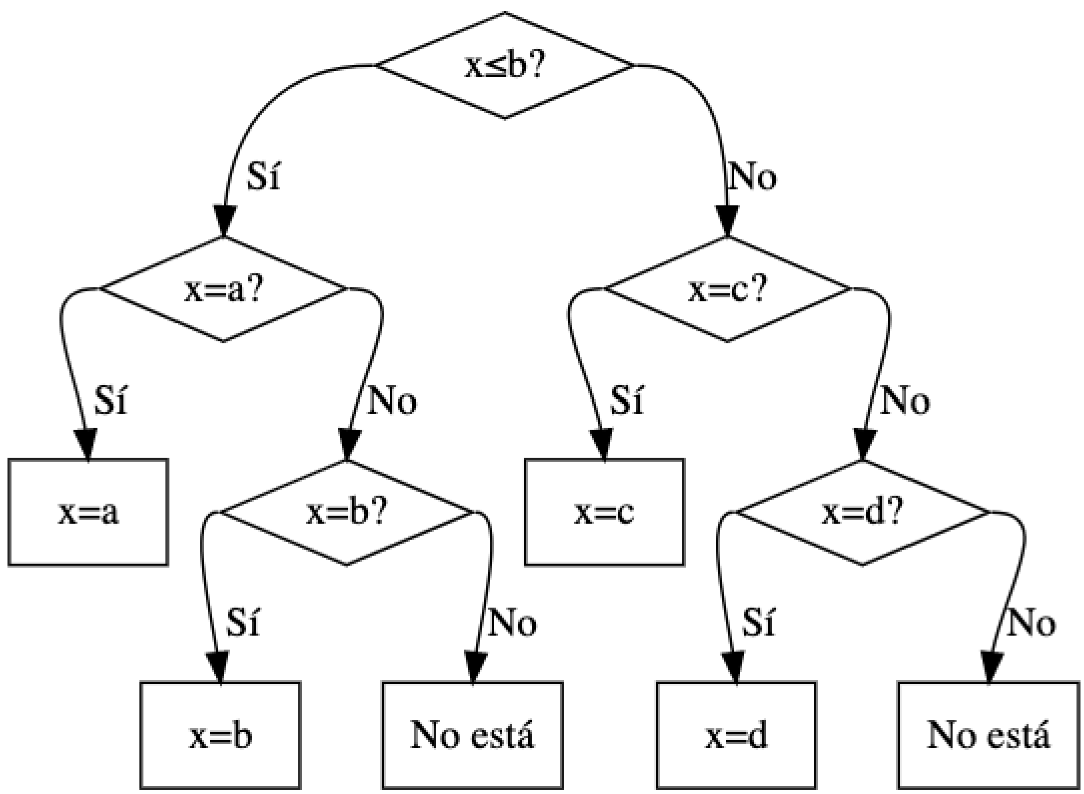
Ya hemos visto anteriormente que si los datos están en un arreglo ordenado, podemos hacer una búsqueda binaria, la que demora tiempo $\lceil \log_2 (n+1) \rceil = \Theta(\log n)$ en el peor caso.

Esto es bastante eficiente, pero tiene el problema que agregar o eliminar datos del arreglo toma tiempo $\Theta(n)$ en el peor caso, por la necesidad de mantener el conjunto ordenado y compacto. Un objetivo que perseguiremos en el resto de este capítulo es tratar de encontrar estructuras de datos que nos permitan buscar de manera tan eficiente como la búsqueda binaria, junto con inserciones y eliminaciones igualmente eficientes.

Pero antes de avanzar en esa dirección, consideremos la pregunta de si es posible buscar má rápido que la búsqueda binaria en el peor caso.

Cota inferior para la búsqueda por comparaciones

Consideremos el problema de buscar una llave x en un conjunto de tamaño 4, digamos $\{a, b, c, d\}$, con $a < b < c < d$. La siguiente figura ilustra una manera como podría hacerse esa búsqueda:



Este tipo de figura se llama un *árbol de decisión*, y en él los rombos representan preguntas y los rectángulos, las salidas (outputs) del algoritmo.

Este árbol de decisión es uno entre la infinidad de árboles que podrían resolver el problema de la búsqueda. Lo importante que hay que observar es que todo algoritmo que funcione mediante comparaciones binarias (comparaciones con salidas "Sí/No") se puede representar por un árbol de decisión.

En este tipo de árbol tenemos que:

- La altura representa el número de comparaciones que hace el algoritmo en el peor caso, y

- El número de hojas (cajas rectangulares) debe ser mayor o igual al número de respuestas posibles que debe ser capaz de emitir el algoritmo.

Recordemos que si N es el número de hojas y h la altura, siempre se tiene $N \leq 2^h$, de donde se deduce que $h \geq \lceil \log_2 N \rceil$ (porque la altura es un número entero), y en consecuencia, tenemos que

$$\text{Peor caso} \geq \lceil \log_2 (\text{número de respuestas distintas}) \rceil$$

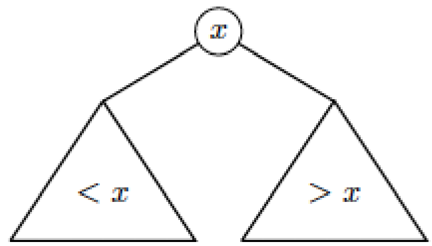
Para el caso de la búsqueda binaria, tenemos que $N = n + 1$, porque el algoritmo de búsqueda debe poder identificar a cada uno de los n elementos, más la respuesta negativa cuando el elemento buscado no está. En consecuencia:

Todo algoritmo que busque en un conjunto de tamaño n mediante comparaciones binarias debe hacer al menos $\lceil \log_2 (n+1) \rceil$ comparaciones en el peor caso.

Por lo tanto, la búsqueda binaria es óptima.

Árboles de Búsqueda Binaria (ABBs)

Un *árbol de búsqueda binaria* (*ABB*) es un árbol binario en que todos sus nodos internos cumplen la siguiente propiedad: Si la llave almacenada en el nodo es x, entonces todas las llaves en su subárbol izquierdo son menores que x, y las lleves en el subárbol derecho son mayores que x.



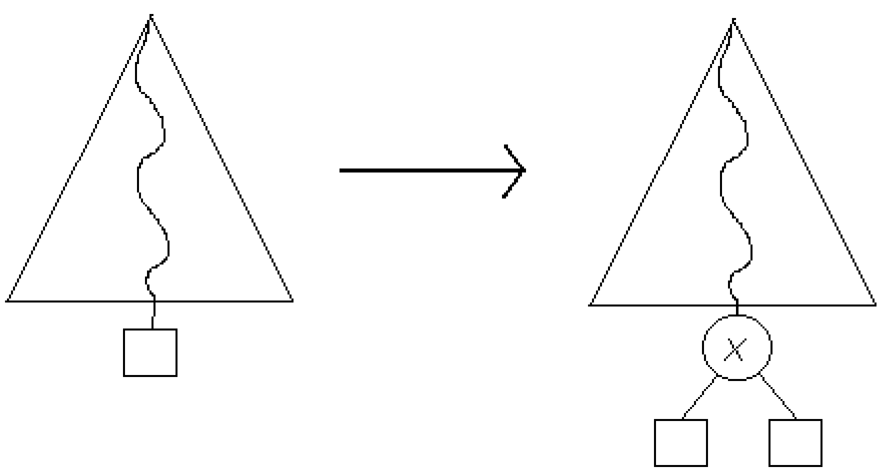
Los ABBs permiten realizar de manera eficiente (en promedio) las operaciones de inserción y búsqueda.

Búsqueda en un ABB

La búsqueda es similar a una búsqueda binaria (de ahí el nombre de estos árboles). Para buscar una llave x se comienza en la raíz. Si x se encuentra ahí, la búsqueda termina exitosamente. Si no está ahí, se continúa buscando en el subárbol izquierdo si x es menor que la raíz, o en el subárbol derecho si x es mayor que la raíz. Si se llega a una hoja (nodo externo), la búsqueda concluye infructuosamente.

Inserción en un ABB

Para insertar una llave x en un ABB, se realiza una búsqueda, que debe ser infructuosa, y la hoja en donde termina la búsqueda se reemplaza por un nodo interno conteniendo la llave x, con dos nuevas hojas como hijos.



Implementación recursiva

Los algoritmos de un ABB se prestan de manera natural a ser programados de manera recursiva, especialmente con la repersentación explícita de los nodos externos:

In [33]:
```python
class Nodoi:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der

    def search(self,x):
        if x==self.info:
            return self
        if x<self.info:
            return self.izq.search(x)
        else:
            return self.der.search(x)

    def insert(self,x):
        assert x!=self.info
        if x<self.info:
            return Nodoi(self.izq.insert(x),self.info,self.der)
        else:
            return Nodoi(self.izq,self.info,self.der.insert(x))

    def string(self):
        return "("+self.izq.string()+str(self.info)+self.der.string()+
")"

class Nodoe:
    def __init__(self):
        pass

    def search(self,x):
        return None

    def insert(self,x):
        return Nodoi(Nodoe(),x,Nodoe())

    def string(self):
        return"□"

class Arbol:
    def __init__(self,raiz=Nodoe()):
        self.raiz=raiz

    def insert(self,x):
        self.raiz=self.raiz.insert(x)

    def search(self,x):
        return self.raiz.search(x)

    def imprimir(self):
        print(self.raiz.string())
```

Hemos incluído una función `imprimir` para poder visualizar (de forma algo rudimentaria) el árbol construído.

In [40]:
```
a=Arbol()
a.insert(42)
a.insert(77)
a.insert(50)
a.insert(10)
a.imprimir()
```

```
((□10□)42((□50□)77□))
```

Para probar nuestra implementación, definiremos una función `test` :

In [41]:
```
def test(a,x):
    print(x, "está" if a.search(x) is not None else "no está")
```

In [42]:
```
test(a,50)
test(a,90)
```

```
50 está
90 no está
```

In [43]:
```
a.insert(90)
a.imprimir()
test(a,90)
```

```
((□10□)42((□50□)77(□90□)))
90 está
```

Implementación no recursiva

Las operaciones de búsqueda e inserción en el árbol no necesitan programarse recursivamente, porque se pueden realizar en una sola pasada de arriba a abajo, sin necesidad de volver hacia arriba. En este caso, toda esa funcionalidad se implementa dentro de al clase `Arbol` :

In [44]:
```
class Nodoi:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der

    def string(self):
        return "("+self.izq.string()+str(self.info)+self.der.string()+
")"

class Nodoe:
    def __init__(self):
        pass

    def string(self):
        return"□"

class Arbol:
    def __init__(self,raiz=Nodoe()):
        self.raiz=raiz

    def insert(self,x):
        if isinstance(self.raiz, Nodoe):
            self.raiz=Nodoi(Nodoe(),x,Nodoe())
            return
        p=self.raiz
        while True:
            assert x!=p.info
            if x<p.info:
                if isinstance(p.izq, Nodoe):
                    p.izq=Nodoi(Nodoe(),x,Nodoe())
                    return
                p=p.izq
            else: # x>p.info
                if isinstance(p.der, Nodoe):
                    p.der=Nodoi(Nodoe(),x,Nodoe())
                    return
                p=p.der

    def search(self,x):
        p=self.raiz
        while not isinstance(p, Nodoe):
            if x==p.info:
                return p
            p=p.izq if x<p.info else p.der
        return None

    def imprimir(self):
        print(self.raiz.string())
```

In [45]:
```
a=Arbol()
a.insert(42)
a.insert(77)
a.insert(50)
a.insert(10)
a.imprimir()
```

```
((□10□)42((□50□)77□))
```

In [46]:
```
test(a,50)
test(a,90)
```

```
50 está
90 no está
```

In [47]:
```
a.insert(90)
a.imprimir()
test(a,90)
```

```
((□10□)42((□50□)77(□90□)))
90 está
```

Costo de búsqueda en un ABB

Peor caso

El peor caso para un árbol dado es la altura del árbol, y el peor árbol posible tiene altura n. Por lo tanto, el costo de búsqueda en el peor caso es $\Theta(n)$.

Caso promedio

Para analizar el costo esperado de búsqueda en un ABB con n nodos, supondremos que en una inserción y en una búsqueda infructuosa, los $n+1$ nodos externos son igualmente probables como punto de destino de la búsqueda y de la inserción, y que en una búsqueda exitosa, los n nodos internos son igualmente probables como punto de destino de la búsqueda.

Recordemos que anteriormente definimos el *largo de caminos internos* (*LCI*) como $I_n = \sum_{x \in \text{Nodos internos}} \text{distancia}(\text{raiz}, x)$, y el *largo de caminos externos* (*LCE*) como $E_n = \sum_{y \in \text{Nodos externos}} \text{distancia}(\text{raiz}, y)$, para los cuales se cumple que $E_n = I_n + 2n$.

Utilizaremos las notaciones C_n y C'_n para el número esperado de comparaciones de llaves en una búsqueda exitosa e infructuosa, respectivamente. Entonces, tenemos que

$$C_n = 1 + \frac{I_n}{n}$$
$$C'_n = \frac{E_n}{n+1}$$

Reemplazando $E_n = I_n + 2n$ en la fórmula para C_n, tenemos que

$$C_n = 1 + \frac{E_n - 2n}{n} = \frac{E_n}{n} - 1 = \frac{n+1}{n} C'_n - 1$$

Por lo tanto, tenemos la siguiente relación entre los costos esperados de búsqueda exitoso e infructuoso:

$$C_n = \left(1 + \frac{1}{n}\right) C'_n - 1$$

de modo que, cuando $n \to \infty$

$$C_n \approx C'_n - 1$$

Para poder completar este análisis necesitamos una segunda ecuación que vincule a estas incógnitas. Para esto, observemos que *el costo de buscar un elemento que acaba de ser insertado es exactamente 1 más que el costo de buscarlo infructuosamente antes de su inserción*. Por lo tanto, si consideramos y promediamos los costos de búsqueda de los elementos en orden de inserción, tenemos que

$$C_n = \frac{(1 + C'_0) + (1 + C'_1) + \cdots + (1 + C'_{n-1})}{n} = 1 + \frac{1}{n} \sum_{0 \leq k \leq n-1} C'_k$$

Multiplicando ambos lados por n, tenemos

$$nC_n = n + \sum_{0 \leq k \leq n-1} C'_k$$

Sustituyendo n por $n+1$ en esta ecuación, tenemos

$$(n+1)C_{n+1} = n+1+\sum_{0\le k\le n} C'_k$$

Restando ambas ecuaciones, tenemos:

$$(n+1)C_{n+1} - nC_n = 1 + C'_n$$

La relación que habíamos obtenido antes entre C_c y C'_n se puede reescribir como

$$nC_n = (n+1)C'_n - n$$

Reemplazando en la ecuación anterior, obtenemos:

$$(n+2)C'_{n+1} - (n+1) - (n+1)C'_n + n = 1 + C'_n$$

Lo que se simplifica a

$$(n+2)(C'_{n+1} - C'_n) = 2$$

obteniéndose la ecuación de recurrencia

$$C'_{n+1} = C'_n + \frac{2}{n+2}$$
$$C'_0 = 0$$

Esta ecuación se puede resolver "desenrrollándola", para obtener

$$C'_n = 2\left(\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n+1}\right)$$

Si definimos los *números armónicos* $H_n = 1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}$, podemos expresar la solución como

$$C'_n = 2(H_{n+1}-1)$$

Los números armónicos son muy cercanos al logaritmo natural. Se puede demostrar que

$$H_n \le 1 + \int_1^n \frac{1}{x}dx = 1 + \ln n$$
$$H_n \ge \int_1^{n+1} \frac{1}{x}dx = \ln(n+1)$$

de donde obtenemos que

$$\ln(n+1) \le H_n \le 1 + \ln n$$

Más precisamente, se puede demostrar que

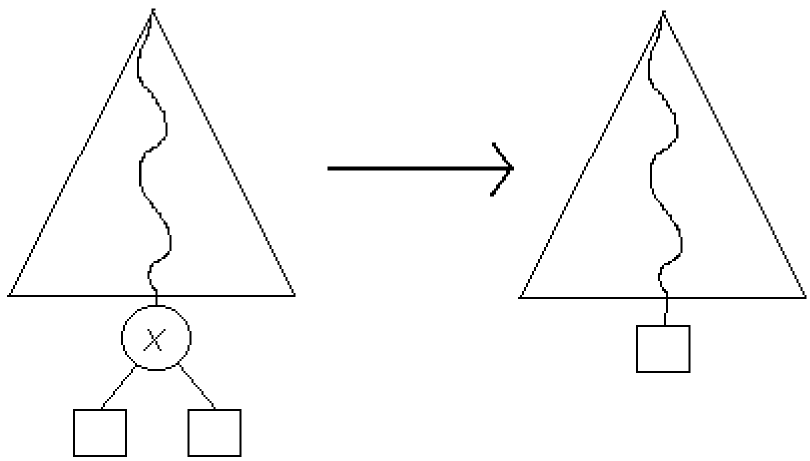
$$H_n = \ln n + \gamma + O\left(\frac{1}{n}\right)$$

Eliminación en un ABB

La eliminación de una llave x es sencilla de efectuar en algunos casos, pero el caso complicado es cuando la llave tiene dos hijos:

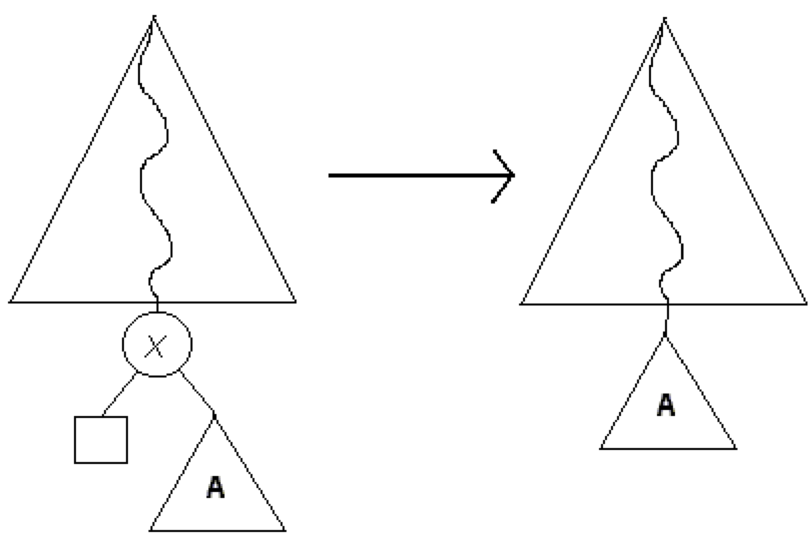
Eliminación de una llave sin hijos

En este caso, el nodo interno que contiene a x desaparece y en su lugar queda un nodo externo:



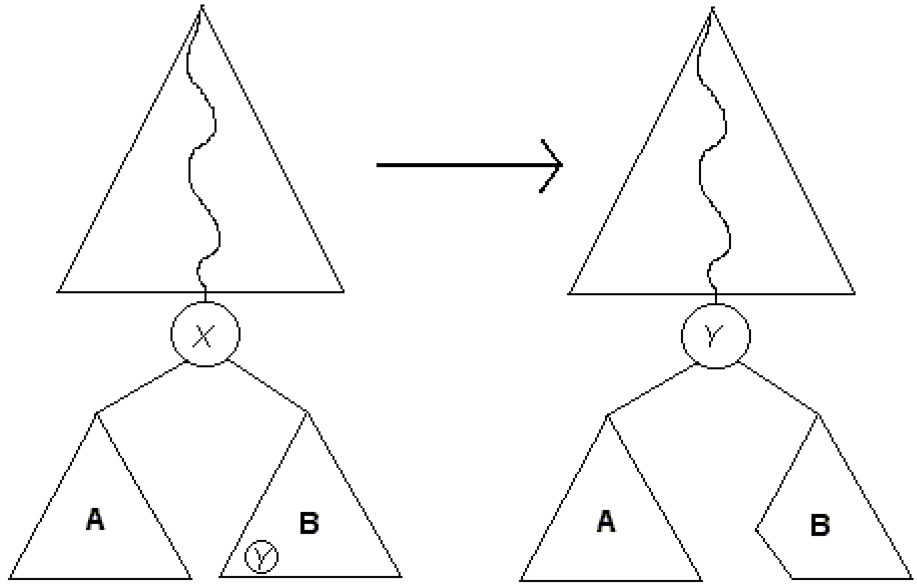
Eliminación de una llave con 1 hijo

En este caso, el padre de la llave x pasa a apuntar al único hijo de x:



Eliminación de una llave con 2 hijos

Si x tiene 2 hijos, no podemos eliminarla directamente, pero sí podemos eliminar a la que la sigue en orden ascendente, digamos y. Se puede demostrar que y necesariamente es unos de los dos casos anteriores, de modo que es fácil de eliminar. Luego de concluído ese proceso, escribimos y en lugar de x en el campo `info` del nodo respectivo.



Por simetría, esto mismo podría haberse hecho con la llave que sigue a x en orden descendente.

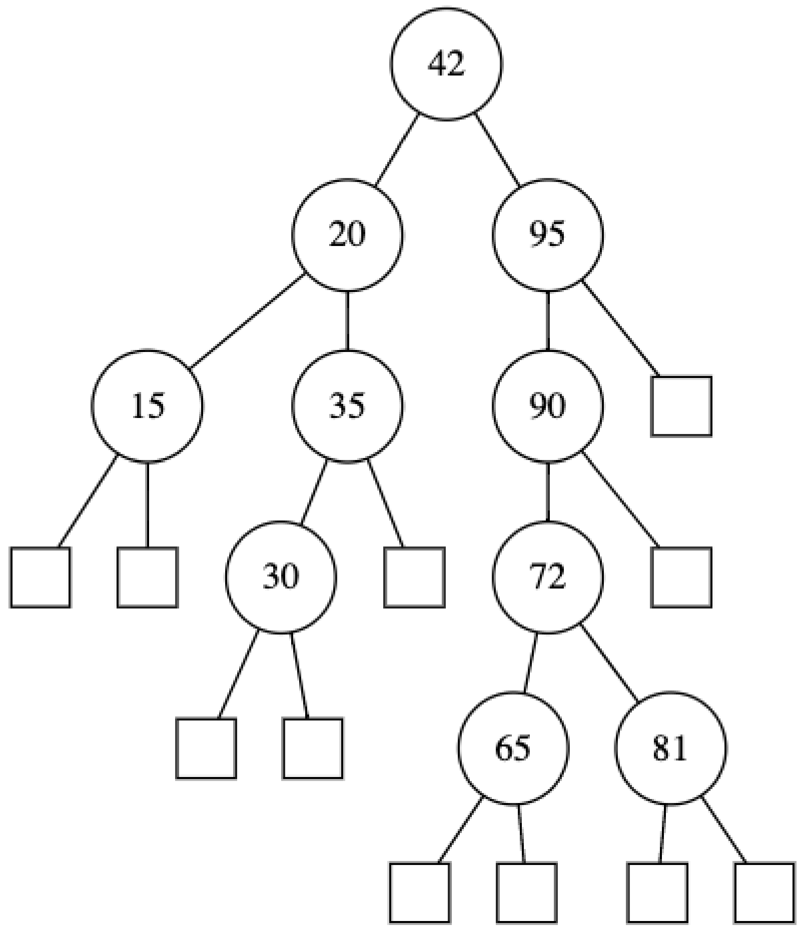
El análisis del costo esperado de búqueda que hicimos anteriormente es válido si solo hay inserciones. El análisis en el caso en que se incluyen eliminaciones es un un problema matemáticamente muy complicado, y sigue siendo un problema abierto. La evidencia experimental indica que se obtienen mejores resultados si se alterna o si se aleatoriza al elegir entre sucesor o el predecesor de x en caso que haya que elegir.

Implementación recursiva de la eliminación

Por simplicidad, omitimos el código para la inserción y búsqueda, y en el caso de un nodo con 2 hijos, eliminamos siempre el sucesor. También ignoramos las eliminaciones de llaves que no están en el árbol.

In [77]:
```
class Nodoi:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der

    def deletemin(self): # Elimina llave mínima del árbol, retorna (llav
e_min,raiz_arbol_restante)
        if isinstance(self.izq,Nodoe): # No hay hijo izquierdo
            return (self.info,self.der)
        # hay hijo izquierdo
        (llave_min,izq_sin_min)=self.izq.deletemin()
        return (llave_min,Nodoi(izq_sin_min,self.info,self.der))

    def delete(self,x):
        if x<self.info:
            return Nodoi(self.izq.delete(x),self.info,self.der)
        if x>self.info:
            return Nodoi(self.izq,self.info,self.der.delete(x))
        # x==self.info
        if isinstance(self.izq,Nodoe): # No hay hijo izquierdo
            return self.der
        if isinstance(self.der,Nodoe): # No hay hijo derecho
            return self. izq
        # Hay hijo izquierdo y derecho
        (y,der_sin_min)=self.der.deletemin()
        return(Nodoi(self.izq,y,der_sin_min))

    def string(self):
        return "("+self.izq.string()+str(self.info)+self.der.string()+
")"

class Nodoe:
    def __init__(self):
        pass

    def delete(self,x):
        return self

    def string(self):
        return"□"

class Arbol:
    def __init__(self,raiz=Nodoe()):
        self.raiz=raiz

    def delete(self,x):
        self.raiz=self.raiz.delete(x)

    def imprimir(self):
        print(self.raiz.string())
```

Para probar este algoritmo utilizaremos el árbol que vimos en el capítulo 4:



In [78]:
```
a=Arbol(
    Nodoi(
        Nodoi(
            Nodoi(Nodoe(),15,Nodoe()),
            20,
            Nodoi(
                Nodoi(Nodoe(),30,Nodoe()),
                35,
                Nodoe()
            )
        ),
        42,
        Nodoi(
            Nodoi(
                Nodoi(
                    Nodoi(Nodoe(),65,Nodoe()),
                    72,
                    Nodoi(Nodoe(),81,Nodoe())
                ),
                90,
                Nodoe()
            ),
            95,
            Nodoe()
        )
       )
)
```

In [79]:
```
a.imprimir()
```

```
(((□15□)20((□30□)35□))42((((□65□)72(□81□))90□)95□))
```

In [80]:
```
a.delete(30)
a.imprimir()
```

```
(((□15□)20(□35□))42((((□65□)72(□81□))90□)95□))
```

In [81]:
```
a.delete(95)
a.imprimir()
```

```
(((□15□)20(□35□))42(((□65□)72(□81□))90□))
```

In [82]:
```
a.delete(42)
a.imprimir()
```

```
(((□15□)20(□35□))65((□72(□81□))90□))
```

In [83]:
```
a.delete(44) # 44 no está en el árbol
a.imprimir()
```

```
(((□15□)20(□35□))65((□72(□81□))90□))
```

In [84]:
```
a.delete(20)
a.imprimir()
```

```
(((□15□)35□)65((□72(□81□))90□))
```

Implementación no recursiva de la eliminación

Al programar la eliminación de esta manera, necesitamos modificar el nodo padre de x. Para simplificar, asegurando que todo nodo tenga un padre, incluso la raíz, durante el proceso de eliminación simularemos que la raíz es hija derecha de un nodo con llave "$-\infty$".

In [108]:
```
class Nodoi:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der

    def string(self):
        return "("+self.izq.string()+str(self.info)+self.der.string()+
")"

class Nodoe:
    def __init__(self):
        pass

    def string(self):
        return"□"

import math
class Arbol:
    def __init__(self,raiz=Nodoe()):
        self.raiz=raiz

    def delete(self,x):
        cabecera=Nodoi(None,-math.inf,self.raiz)
        p=cabecera # padre del candidato a ser eliminado
        q=cabecera.der # el candidato
        while not isinstance(q,Nodoe):
            if x<q.info:
                (p,q)=(q,q.izq)
            elif x>q.info:
                (p,q)=(q,q.der)
            else: # encontramos x
                if isinstance(q.izq,Nodoe): # no hay hijo izquierdo
                    r=q.der
                elif isinstance(q.der,Nodoe): # no hay hijo derecho
                    r=q.izq
                else: # hay 2 hijos, eliminamos el mínimo del árbol dere
cho y lo movemos a q
                    s=q.der
                    if isinstance(s.izq,Nodoe): # encontramos el mín de
 inmediato
                        q.info=s.info
                        q.der=s.der
                    else: # el mín está más abajo
                        t=s.izq # s es el padre del candidato a min, t e
s el candidato
                        while not isinstance(t.izq,Nodoe): # mientras no
encontremos el final de la rama izquierda
                            (s,t)=(t,t.izq)
                        q.info=t.info
                        s.izq=t.der
                    r=q
                if x<p.info:
                    p.izq=r
                else:
                    p.der=r
```

```
                self.raiz=cabecera.der
                return
        # si llegamos acá, x no estaba, no hacemos nada

    def imprimir(self):
        print(self.raiz.string())
```

In [94]:
```
a=Arbol(
    Nodoi(
        Nodoi(
            Nodoi(Nodoe(),15,Nodoe()),
            20,
            Nodoi(
                Nodoi(Nodoe(),30,Nodoe()),
                35,
                Nodoe()
            )
        ),
        42,
        Nodoi(
            Nodoi(
                Nodoi(
                    Nodoi(Nodoe(),65,Nodoe()),
                    72,
                    Nodoi(Nodoe(),81,Nodoe())
                ),
                90,
                Nodoe()
            ),
            95,
            Nodoe()
        )
       )
)
```

In [95]:
```
a.imprimir()
```

```
(((□15□)20((□30□)35□))42((((□65□)72(□81□))90□)95□))
```

In [96]:
```
a.delete(30)
a.imprimir()
```

```
(((□15□)20(□35□))42((((□65□)72(□81□))90□)95□))
```

In [97]:
```
a.delete(95)
a.imprimir()
```

```
(((□15□)20(□35□))42(((□65□)72(□81□))90□))
```

In [98]:
```
a.delete(42)
a.imprimir()
```

```
(((□15□)20(□35□))65((□72(□81□))90□))
```

In [99]:
```
a.delete(44) # 44 no está en el árbol
a.imprimir()
```

```
(((□15□)20(□35□))65((□72(□81□))90□))
```

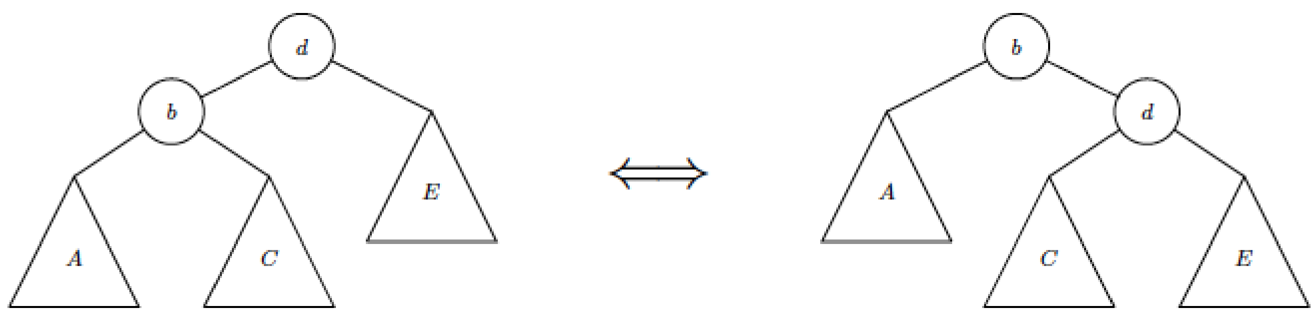
In [100]:
```
a.delete(20)
a.imprimir()
```

```
(((□15□)35□)65((□72(□81□))90□))
```

Rotaciones en un ABB

Una operación que es la base de muchos algoritmos es la *rotación* (o rotación simple, para diferenciarla de la rotación doble, que veremos más adelante).

Una rotación entre los nodos b y d es la siguiente transformación:



Como lo sugiere la figura, por simetría esta operación también se puede hacer en la dirección inversa.

Esta operación tiene costo constante, porque solo requiere modificar tres punteros, y preserva el orden de izquierda a derecha que caracteriza a un ABB. Su efecto es que A y b suben un nivel, mientras que d y E bajan un nivel.

Aplicación: Inserción en la raíz

El método estándar de inserción en un ABB es inserción en las hojas. Veremos a continuación que existe un método alternativo, que deja al nuevo elemento como raíz del árbol.

El método consiste en insertar el nuevo elemento de la manera usual, y luego hacer una secuencia de rotaciones que vayan haciéndolo ascender, hasta que llegue a estar en la raíz.

In [147]:
```
class Nodoi:
    def __init__(self, izq, info, der):
        self.izq=izq
        self.info=info
        self.der=der
    def right_rotation(self):
        p=self.izq
        (p.der,self.izq)=(self,p.der)
        return p
    def left_rotation(self):
        p=self.der
        (p.izq,self.der)=(self,p.izq)
        return p
    def root_insert(self,x):
        assert x!=self.info
        if x<self.info:
            self.izq=self.izq.root_insert(x) # x queda como raíz del hij
o izquierdo
            return self.right_rotation() # la rotación deja a x como raí
z
        else:
            self.der=self.der.root_insert(x) # x queda como raíz del hij
o derecho
            return self.left_rotation() # la rotación deja a x como raíz

    def string(self):
        return "("+self.izq.string()+str(self.info)+self.der.string()+
")"

class Nodoe:
    def __init__(self):
        pass

    def root_insert(self,x):
        return Nodoi(Nodoe(),x,Nodoe())

    def string(self):
        return"□"

class Arbol:
    def __init__(self,raiz=Nodoe()):
        self.raiz=raiz

    def root_insert(self,x):
        self.raiz=self.raiz.root_insert(x)

    def imprimir(self):
        print(self.raiz.string())
```

In [148]:
```
a=Arbol()
a.root_insert(10)
a.imprimir()
a.root_insert(20)
a.imprimir()
a.root_insert(30)
a.imprimir()
a.root_insert(25)
a.imprimir()
a.root_insert(15)
a.imprimir()
```

```
(□10□)
((□10□)20□)
(((□10□)20□)30□)
(((□10□)20□)25(□30□))
((□10□)15((□20□)25(□30□)))
```