AA2021a-05-Estructuras de datos fundamentales 1

Abdiel E. Cáceres González

14 de febrero de 2021

Índice

Código de Honor

La UJAT espera que sus estudiantes muestren respeto por el orden, la moral y el honor de sus compañeros, sus maestros y su persona. Por ello, se establece este Código de Honor, con el propósito de guiar la vida escolar de los alumnos. Estos lineamientos no permiten todo acto que deshonre los ámbitos académicos. Las siguientes acciones son consideradas como violatorias al Código de Honor de:

1. Usar, proporcionar o recibir apoyo o ayuda no autorizada al presentar exámenes y al elaborar o presentar reportes, tareas y en general en cualquier otra forma por la que el maestro evalúe el desempeño académico del estudiante.

2. Presentar trabajos o exámenes por otra persona, o permitir a otro que lo haga por uno mismo.

Las sanciones podrán ser desde la reprobación con calificación 0 (cero) en la tarea por evaluar, hasta una calificación reprobatoria de 0 (cero) en la materia.

PARTE TRES

Estructuras de datos

1. Pilas y colas

Las pilas y las colas son conjuntos dinámicos en donde un elemento preestablecido es quitado del conjunto por medio de una operación `Delete`. En una pila, el elemento removido del conjunto es el que se ha insertado más recientemente o sea el último en haber sido insertado: la pila implementa la política **LIFO** [del inglés *last-in, first-out*]. De manera similar en una cola,el elemento que se debe eliminar es el primero que fue insertado: la cola implementa una política **FIFO** [del inglés `first-in, first-out`]. En esta lección usaremos un arreglo para ilustrar una manera eficiente de implementar estas estructuras.

1.1. Pilas

La operación `Insert` en una pila frecuentemente se llama `Push`, y a la operación `Delete` se le conoce como `Pop`. Esos nombres vienen de la imagen de un apilamiento real de objetos, como los platos en un servicio de buffete. Tanto la operación `Push` como `Pop` no requieren argumentos ya que el elemento que debe ser removido siempre es el último que fue insertado.

Digamos que S es una pila. La pila S tiene un atributo que se llama $S.pop$ que apunta al elemento mas recientemente insertado. Los elementos de la pila son almacenados en las posiciones $S\langle 1 \ldots S.top\rangle$, donde $S\langle 1\rangle$ es el elemento que está en el fondo de la pila [el primero que se agregó] y $S\langle S.top\rangle$ es el elemento al tope [el último que fue agregado].

Cuando $S.top = 0$, la pila está **vacía**. Podemos probar si la pila está vacía haciendo una operación `Stack_empty`. Si tratamos de sacar un elemento de una pila vacía, decimos que hay un error por **carencia** en la pila. Si $S.top$ sobrepasa n, la pila se **desborda**. En el pseudocódigo presentado aquí, no nos preocupamos por una pila desbordada.

Podemos implementar las operaciones con una pila, solamente con unas pocas líneas:

```
1  Stack_empty(S)
2    if S.top = 0
3      return True
4    else
5      return False
```

```
1  Push(S,x)
2    S.top ⟵ S.top + 1
3    S[S.top] ⟵ x
```

```
1  Pop(S,x)
2    if Stack_Empty(S)
3      return "ERROR: Pila vacia"
4    else
5      S.top ⟵ S.top - 1
6      return S[S.top + 1]
```

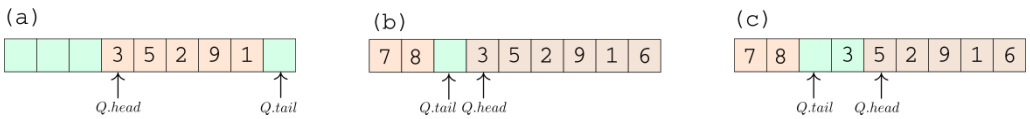
Observa que todas estas operaciones tienen complejidad $\boldsymbol{O}(1)$.

1.2. Colas

Una Cola es una estructura de datos que almacena elementos de forma lineal. La operación de insertar un elemento la llamamos `Enqueue` [que se pronuncua como *inkiu*] y la operación de remover un elemento de una cola se llama `Dequeue` [que se pronuncua como *di-kiu*]. La propiedad $FIFO$ *first in first out* de la cola,ocasioa que el comportamiento de la cola funcione como una línea de clientes que esperan por ser atendidos por un cajero.

La cola tiene los atributos `head` y `tail`. Cuando un elemento se inserta, se coloca al final de la cola, justo como se hace en la fila de clientes. El elemento que es removido, siempre sale por el frente de la cola, justo como lo hace la persona recién atendida en una fila de clientes.

La cola tiene un atributo $Q.head$ que apunta al elemento que deberá ser removido, representa el frente de la cola. El atributo $Q.tail$ apunta a la siguiente posición en la que será insertado un nuevo elemento. Los elementos de la cola se ubican en las posiciones $Q.head, Q.head + 1, \ldots, Q.tail - 1$.



En la figura se muestra cómo opera la cola cuando es modalada con un arreglo de tamaño fijo. En (a) se muestra la cola después de haber insertado los elementos 3,5,2,9 y 1 [en ese orden], en ese momento los apuntadores $Q.head = 4$ y $Q.tail = 9$, considerando que el arreglo tieno posiciones que inician en 1, es decir $Q[Q.head] = Q[4] \rightarrowtail 3$ luego en (b) se han agregado los nuevos elementos 6, 7 y 8, observa que la cola empieza a llenar los espacios vacíos que están al inicio del arreglo; en (c) se ha removido un elemento de la cola, por lo que el apuntador $Q.tail$ se actualiza al siguiente elemento en la cola. Observa que el elemento que estaba antes en el frente de la cola, sigue estando en el arreglo, pero eso no afecta el funcionamiento de la cola.

Cuando el apuntado al final de la cola alcanza el límite del arreglo, el apuntador debe «dar la vuelta» e iniciar en la localidad 1. Cuando $Q.head = Q.tail$, significa que la cola esta vacía. Inicialmente tenemos $Q.head = Q.tail = 1$. Si intentamos remover un elemento de una cola vacía, la cola debe emitir un error. Cuando $Q.head = Q.tail + 1$, la cola está llena y si queremos insertar un elemento en la cola, entonces la cola debe emitir un error de «Cola llena».

En nuestros procedimientos `Enqueue` y `Dequeue`, hemos omitido el chequeo de errores [que debes arreglar en un ejercicio]. El pseudo código asume que $sz = Q.length$.

```
1  Enqueue(Q,x)
2    Q[Q.tail] ⟵ x
3    if Q.tail ⟵ Q.length
4      Q.tail ⟵ 1
5    else
6      Q.tail ⟵ Q.tail + 1
```

```
1  Dequeue(Q)
2    x ⟵ Q[Q.head]
3    if Q.head = Q.lenght
4      Q.head ⟵ 1
5    else
6      Q.head ⟵ Q.head + 1
7  return x
```

2. Listas ligadas

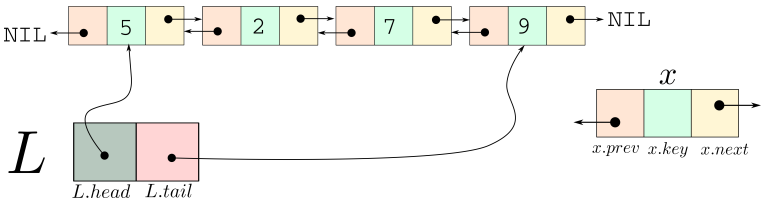
Una de las decisiones más importantes a la hora de diseñar un algoritmo es la estructura de datos que debemos ocupar. La estructura de datos empleada influye significativamente en el diseño y eficiencia del algoritmo. La información presentada en este curso dista mucho de ser completa ya que no es un curso de estructuras de datos, sino que está más enfocada al tema de la eficiencia, de modo que en ocasiones se omitirán algunos detalles por resaltar otros.

Una **lista ligada** es una estructura de datos en la que los objetos se disponen en un orden lineal. A deferencia de un arreglo, en donde el orden lineal lo establecen los índices del arreglo, el orden en una lista ordenada lo establece un apuntador que tiene cada nodo, en donde se direcciona al siguiente nodo en la lista, en donde quiera que esté almacenado. Las listas ligadas proporcionan una representación flexible y simple de un conjunto dinámico, en donde si se quiere, se pueden modelar las operaciones de las pilas y las colas.

Cada elemento en una **lista doblemente ligada** L es un objeto que tiene un atributo key y otros dos atributos que son los apuntadores $next$ y $prev$. El objeto pudiera contener más información. Dado un elemento x en la lista, $x.next$ apunta a su sucesor en la lista

ligada y $x.prev$ apunta a su predecesor. Si $s.prev = \texttt{NIL}$, el elemento no tiene predecesor por lo que es el primer elemento o la cabeza de la lista, llamada **head**. Si $x.next = \texttt{NIL}$, el elemento x no tiene sucesor por lo que es el último elemento de la lista, llamado la **cola** de la listao **tail**. Un atributo $L.head$ apunta al primer elemento de la lista. Si $L.head = \texttt{NIL}$, la lista está vacía.

Una lista puede tener una o más formas. Puede ser simplemente ligada, doblemente ligada, puede estar ordenada o no y puede ser circular o no. Si una lista es **simplemente ligada** omitimos el apuntador *prev* en cada elemento. Si la lista es **ordenada**, el orden lineal de la lista corresponde al orden lineal de la información almacenada en los elemento de la lista; el elemento menor está en la cabeza de la lista y el mayor se encuentra en la cola.



Si la lista es **desordenada**, los elementos pueden aparecer en cualquier orden. En una **lista circular**. el apuntador *prev* de la cabeza de la lista apunta a la cola de la lista y el apuntador *next* de la cola de la lista apunta a la cabeza de la lista. En esta parte del curso trabajaremos con listas desordenadas y doblemente ligadas.

2.1. Búsqueda en una lista ligada

El procedimiento `List_Search` (L, k) encuentra [si existe] el primer elemento con la clave k en la lista L mediante una simple búsqueda lineal, devolviendo un apuntador a ese elemento. Si no existe tal objeto, entonces el procedimiento devuelve `NIL`.

Ejemplo 1

En la lista L mostrada en la imagen, al hacer `List_Search` $(L, 7)$, se obtiene el índice 3, que indica que el número buscado se encuentra en la tercera posición.

Al hacer `List_Search` $(L, 8)$ se obtiene `NIL`.

```
1  List_Search(L,k)
2    x ⟵ L.head
3    while x≠ NIL ∧ x.key≠ k
4      x⟵ x.next
5    return x
```

Para buscar un elemento en una lista de n objetos, el procedimiento `List_Search` toma un tiempo del orden $\boldsymbol{\Theta}(n)$ en el peor caso, ya que tiene que buscar en toda la lista.

2.2. Insertar un elemento en la lista ligada

Dado un elemento x cuyo atributo *key* ya ha sido establecido, el procedimiento `List_Insert` (L, x) coloca el nuevo elemento x en la cabeza de la lista L, aprovechando que la lista no es ordenada.

```
1  List_Insert(L,x)
2    x.next ⟵ L.head
3    if L.head ≠ NIL
4      L.head.prev ⟵ x
5    L.head ⟵ x
6    x.prev ⟵ NIL
```

Observa que la notación `A.atr1.atr2` significa que `A` es un objeto que tiene un atributo `atr1`, el valor de ese atributo es un objeto que tiene un atributo `atr2`.

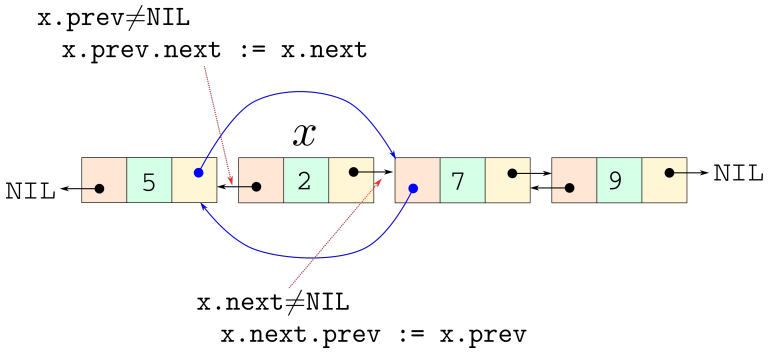
El tiempo de ejecución de este algoritmo es $\boldsymbol{O}(1)$ para una lista de n elementos.

2.3. Eliminar un elemento de la lista ligada

El procedimiento `List_Delete` (L, x) quita un elemento x de la lista ligada L. Para borrar el elemento x de la lista se debe considerar un par de casos, si el elemento tiene un predecesor y un sucesor, es decir está entre dos elementos, o bien se encuentra en algún extremo de la lista.

El proceso de eliminar un elemento de la lista se puede dividir en dos partes: buscar el elemento deseado y borrar el elemento. El proceso de buscar el elemento deseado ya fue descrito.

```
1  List_Delete(L,x)
2    if x.prev ≠ NIL
3      x.prev.next ⟵ x.next
4    else
5      L.head ⟵ x.next
6    if x.next ≠ NIL
7      x.next.prev ⟵ x.prev
```



Observa que el procedimiento `List_Delete` permite que el elemento x eliminado, mantenga sus apuntadores a elementos actuales de la lista, sin embargo ningún apuntador de elementos de la lista hace referencia a x, por lo que queda «oculto».

El procedimiento `List_Delete` corre en tiempo $\boldsymbol{O}(1)$, pero si queremos eliminar algún elemento donde primero se tiene que buscar, el procedimiento toma $\boldsymbol{\Theta}(n)$ en el peor caso, porque primero hay que buscar el elemento y luego eliminarlo.

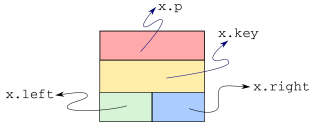
3. Árboles

Los métodos que se han proporcionado para las listas, se pueden utilizar en cualquier estructura de datos homogénea. En esta parte de la lección veremos cómo utilizar la estructura de una lista ligada para representar árboles con raíz.

Vamos a representar cada nodo de un árbol mediante un objeto. Del mismo modo que en las listas ligadas, supondremos que cada cono contiene un atributo *key* que contiene la información. Los demás atributos de interés son apuntadores a otros nodos, y pueden variar de acuerdo al tipo de árbol.

3.1. Árboles binarios

En la siguiente figura se muestra en primer lugar una representación gráfica de una estructura de nodo. Los nodos en un árbol binario tienen al menos tres atributos, un campo para almacenar la clave *key*, otro para almacenar un apuntador al lado izquierdo *left* y otro para almacenar un apuntador al lado derecho *right*. Aunque también se necesitará un apuntador al padre p.

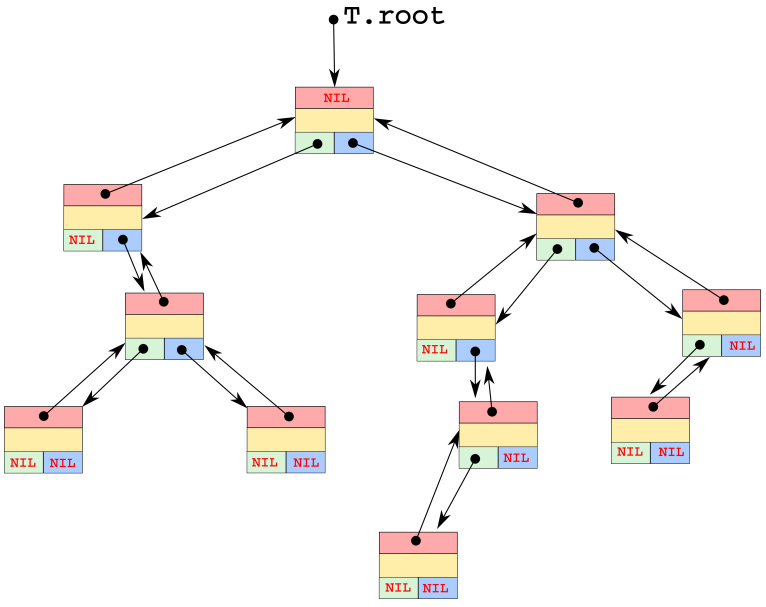


El siguiente segmento de código muestra la definición de una clase para los nodos de un árbol binario. Observa que el constructor de la clase [si se proporciona] es la información que es almacenada en el nodo. La clase tiene cuatro atributos: `k` que almacena la información del nodo; `p` que deberá almacenar una referencia al padre; `left` que almacena una referencia al subárbol izquierdo y `right` que hace lo propio para el lado derecho.

```
class nodo():
  def __init__(self, k=None):
    self.key = k
    self.p = None
    self.left = None
    self.right = None
  def __str__(self):
    if self.left == None:
      sl = 'None'
    else:
      sl = str(self.left.key)
    if self.right == None:
      sr = 'None'
    else:
      sr = str(self.right.key)
    sc = "%s ---> [%s<| %s |>%s]" %(self.p,sl,str(self.key),sr)
    return sc
  def set_parent(self, n):
    self.p = n
  def set_left(self, n):
    self.left = n
  def set_right(self, n):
    self.right = n
  def p(self):
    return self.p
  def left(self):
    return self.left
  def right(self):
    return self.right
  def key(self):
    return self.key
```

Si el árbol no es binario, sino que pudiera almacenar a lo más 3 hijos [subárboles], entonces se pueden agregar más atributos, o bien crear una lista dinámica para ir agregando tantos hijos como sea necesario.

En la siguiente figura te muestro cómo utilizamos los atributos p, $left$ y $right$ para almacenar los apuntadores al padre, al hijo izquierdo y al derecho de cada nodo en el árbol binario T respectivamente. Si $x.p = \texttt{NIL}$, entonces x es la raíz del árbol. Si un nodo x no tiene hijo izquierdo, entonces $x.left = \texttt{NIL}$ y de manera similar parael hijo derecho. La raíz de todo el árbol T es direccionada mediante el atributo [del árbol] $T.root$ si $T.root = \texttt{NIL}$, entonces el árbol es vacío.



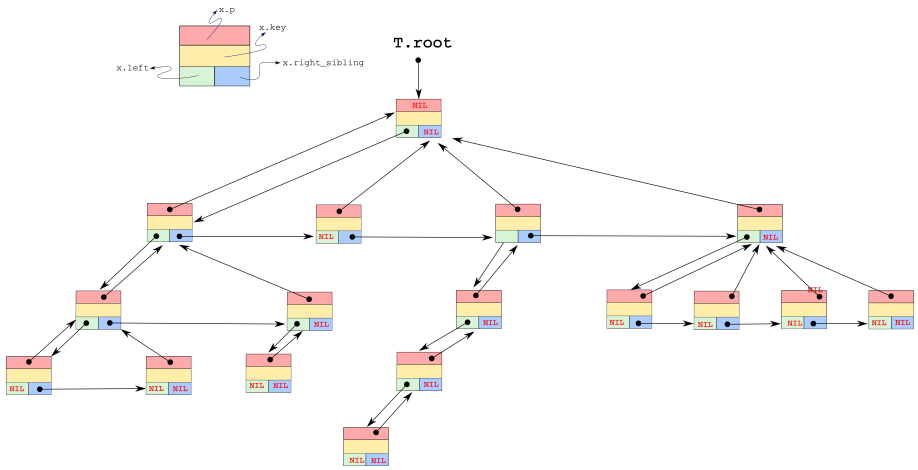
Podemos extender el esquema de representar un árbol binario para representar árboles que tengan hasta k hijos en cada nodo, con k una constante. Cambiaremos los atributos $left$ y $right$ por $child_1$, $child_2$, ..., $child_k$. Sin embargo este esquema ya no funciona cuando cada nodo puede tener un número variable e ilimitado de hijos, porque no sabemos cuántos atributos colocar; es mas, si supieramos que el árbol tiene nodos con a lo más

k hijos, pero con k mucho mayor que el número de hijos que efectivamente presenta, podemos tener un desperdicio de memoria.

Afortunadamente, hay un esquema adecuado para representar árboles con un número arbitrario de hijos. Se aprovecha el uso de memoria del orden $\boldsymbol{O}(n)$ para cualquier nodo de n hijos del árbol. La representación **hijo-izquierdo, hermano derecho**.

Como antes, cada nodo tiene un apuntador al padre p y $T.root$ apunta a la raíz del árbol T. Cada nodo, en lugar de tener un apuntador para cada uno de los hijos, solamente tendrá dos apuntadores:

1. $x.left$, el hijo izquierdo.
2. $x.right\_sibling$ el hermano derecho.



Si el nodo x no tiene hijos, entonces $x.right\_sibling =$ `NIL`, y si el nodo x es el hijo de más a la derecha de su padre, entonces $x.right\_sibling =$ `NIL`.

3.2. Árboles binarios de búsqueda

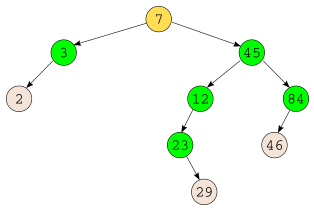
El árbol de búsqueda soporta varias operaciones sobre conjuntos dinámicos, incluyendo `Search`, `Minimun`, `Maximun`, `Predecessor`, `Successor`, `Insert` y `Delete`.

Las operaciones básicas sobre un árbol binario de búsqueda toman un tiempo proporcional a la altura del árbol. Para un árbol binario completo de n nodods, estas operaciones corren en tiempo $\boldsymbol{\Theta}(\log n)$ en el peor caso. Si el árbol es una lista de n nodos, la misma operación toma un tiempo $\boldsymbol{\Theta}(n)$ en el peor caso.

Un árbol binario de búsqueda está organizado [como se sugiere] en un árbol binario en donde la principal característica es la forma en que están organizados los nodos en el árbol.

Además de la clave key y cualquier otra información satelital, cada nodo tiene los atributos $left$, $right$ y p que son apuntadores a los nodos que corresponden al hijo izquierdo, al hijo derecho y al padre respectivamente. Si no existe un hijo o el padre, entonces ese atributo tiene el valor `NIL`. El nodo raíz es el único nodo que tiene `NIL` en el apuntador al nodo padre.

Las claves en un árbol binario de búsqueda siempre se almacenan de tal forma que se satisface la siguiente propiedad:



Definición 1 (Propiedad del árbol binario de búsqueda)**.** Sea x un nodo en el árbol binario de búsqueda (`ABB`). Si y es un nodo en el subárbol izquierdo de x, entonces $y.key \leq x.key$. Si y es un nodo en el subárbol derecho de x, entonces $y.key \geq x.key$.

3.2.1. Recorrido inorden

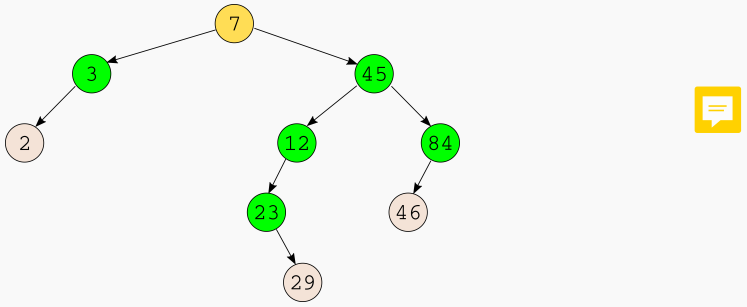
La proiedad de los `ABB` nos permite imprimir todas las claves en orden, utilizando un algoritmo recursvo simple, que se llama **recorrido inorden del árbol**. Este algoritmo se

llama de ese modo porque imprime la clave del nodo raíz de un subárbol, luego imprime los valores de los nodos en el subárbol izquierdo y finalmente los valores de las claves en los nodos del subárbol derecho [también hay **recorrido en preorden** y **recorrido en postorden**]. Para utilizar el siguiente procedimiento para imprimir todos los elementos de un árbol binario de búsqueda t, invocaremos el método `Inorder_Tree_Walk` $(T.root)$.

```
1  Inorder_Tree_Walk(x)
2    if x ≠ NIL
3      Inorder_Tree_Walk(x.left)
4      print x.key
5      Inorder_Tree_Walk(x.right)
```

Ejemplo 2: Recorrrido inorden

Dado el `ABB` que se muestra, el recorrido inorden es $2, 3, 7, 23, 29, 12, 45, 46, 84$



Recorrer el árbol de n nodos toma un tiempo $\boldsymbol{\Theta}(n)$, porque después de la llamada inicial, el procedimiento se invoca a si mismo recursivamente exactamente dos veces por cada nodo en el árbol, una vez por su hijo izquierdo y una vez por el hijo derecho. El siguiente teorema proporciona una demostración formal del tiempo lineal empleado para realizar el recorrido inorden.

Teorema 1. *Si x es la raíz de un subárbol de n nodos, entonces* `Inorder_Tree_Walk` (x) *toma un tiempo del orden* $\boldsymbol{\Theta}(n)$.

Demostración. Digamos que $\mathtt{T}(n)$ es el tiempo que toma `Inorder_Tree_Walk` (x) cuando se invoca sobre la raíz de un subárbol de n nodos. Como `Inorder_Tree_Walk` visita todos los n nodos del subárbol, tenemos $\mathtt{T}(n) = \boldsymbol{\Omega}(n)$. Lo que queda para demostrar es que $\mathtt{T}(n) = \boldsymbol{O}(n)$. Como `Inorder_Tree_Walk` toma una pequeña y constante cantidad de tiempo para un subárbol vacío [haciendo la prueba $x \neq$ `NIL`], tenemos que $\mathtt{T}(0) = c$ para alguna constante $c > 0$. Para $n > 0$, supongamos que `Inorder_Tree_Walk` se invoca sobre un nodo x cuyo subárbol izquierdo tiene k nodos y cuyo subárbol derecho tiene $n-k-1$ nodos. El tiempo para realizar `Inorder_Tree_Walk` (x) está acotado por $\mathtt{T}(n) \leq \mathtt{T}(k) + \mathtt{T}(n-k-1) + d$ para alguna constante $d > 0$ que refleja una cota superior sobre el tiempo para ejecutar `Inorder_Tree_Walk` (x), solamente para el tiempo dedidado en las llamadas recursivas. Usamos el método de la substitución para mostrar que $\mathtt{T}(n) = \boldsymbol{O}(n)$ demostrando que $\mathtt{T}(n) \leq (c+d)n + c$. Para $n = 0$, tenemos $(c+d) \cdot 0 + c = c = \mathtt{T}(0)$. Para $n > 0$ tenemos

$$\begin{aligned} \mathtt{T}(n) &\leq \mathtt{T}(k) + \mathtt{T}(n-k-1) + d \\ &= ((c+d)k + c) + ((c+d)(n-k-1) + c) + d \\ &= ((c+d)n + c) - (c+d) + c + d \\ &= (c+d)n + c) \end{aligned}$$

□

3.2.2. Consultas a un árbol binario de bśuqueda

Al utilizar un árbol binario de búsqueda, lo primero que necesitamos es buscar un elemento en el árbol, además de la operación `Search`, podemos definir las operaciones `Minimum`, `Maximum`, `Successor` y `Predecessor`. Aquí examinaremos estas operaciones y

mostraremos cómo hacerlas en tiempo $\boldsymbol{O}(h)$, donde h es la altura de un árbol binario de búsqueda.

Buscar un nodo en el árbol

Usaremos el procedimiento `Tree_Search` para buscar un nodo con una clave dada. Dado un apuntador a la raíz del árbol y una clave k, `Tree_Search` devuelve un apuntador al nodo con la clave k si existe; de otro modo devolverá `NIL`.

```
1  Tree_Search(x,k)
2    if x = NIL∨ k=x.key
3      return x
4    if k<x.key
5      return Tree_Search(x.left,k)
6    else
7      return Tree_Search(x.right.k)
```

El procedimiento empieza su búsqueda en la raíz y traza una ruta simple desendiendo en el árbol. Para cada nodo x encuentra, compara la clave k con $x.key$. Si lasdos claves soniguales, la búsqueda termina. Si k es menor que $x.key$, la búsqueda continua en el subárbol izquierdo de x, como la propiedad del árbol binario de búsqueda implica que k no puede haber sido almacenada en el lado derecho. Simetricamente, si k es man grnde que $x.key$, la búsqueda continua en el subárbol derecho. Los nodos encontrados durante la recursión forman una ruta simple que desciende desde la raíz del árbol. así el tiempo de ejecución de `Tree_Search` es $\boldsymbol{O}(h)$, donde h es la altura del árbol.

Mínimo y máximo

Si queremos buscar el elemento **mínimo**, quien tiene la clave menor, podemos hacerlo siguiente los apuntadores al hijo izquierdo desde la raíz hasta que encontremos `NIL`. El siguiente procedimiento devuelve el apuntador al elemento mínimo en el subárbol con raíz en un nodo x, que podemos asumir que no es nulo:

```
1  Tree_Minimum(x)
2    while x.left≠ NIL
3      x ←↩ x.left
4    return x
```

La propiedad del árbol binario de búsqueda garantiza que `Tree_Minimum` es correcto. Si un nodo x no tiene subárbol izquierdo, entonces como cualquier clave en le subárbol derecho de x es al menos tan grande como $x.key$, la mínima clave en el subárbol con raíz en x es precisamente $x.key$. Si el nodo x tiene subárbol izquierdo, entonces ninguna clave en el subárbol derecho es más pequeña que $x.key$ y cualquier claver en el subárbol izquierdo no es mayor que $x.key$, la clave mínima en el subarbol con raíz en x reside en el subárbol con raíz en $x.left$.

El psecudocódigo para `Tree_Maximum` es simétrico:

```
1  Tree_Maximum(x)
2    while x.right≠ NIL
3      x ←↩ x.right
4    return x
```

Ambos procedimientos corren en tiempo $\boldsymbol{O}(h)$ para un árbol de altura h, como en `Tree_Search`, la secuencia de nodos encontrados forma una ruta simple que desciende desde la raíz del árbol.

Sucesor y predecesor

Dado un nodo en un árbol binario de búsqeda, algunas veces necesitamos encontrar susucesor en el orden determinado por el recorrido inorden. Si todas las claves son diferentes, el sucesor de un nodo x es el nodo con la clave más pequeña que es mayor que $x.key$. La estrcutura de un árbol binario de búsqueda nos permite determinar el sucesor de un nodo sin comparar claves. El siguiente procedimiento devuelve el sucesor de un nodo x en un árbol binario si existe y `NIL` si x es la clave más grande en el árbol:

```
Tree_Successor(x)
  if x.right≠ NIL
    return Tree_Minimun(x.right)
  y ←∼ x.p
  while y≠ NIL∧ x= y.right
    x ←∼ y
    y ←∼ y.p
  return y
```

Dividimos el código `Tree_Successor` en dos casos. Si el subárbol derecho del nodo x no es vacío, entonces el sucesor de x es exactamente el nodo de más a la izquierda en el subárbol derecho de x.

Por otro lado, su el subárbol del nodo x está vacío y x tiene un sucesor y, entonces y es el antecesor más bajo de x cuyo hijo izquierdo también es antecesor de x. Para encontrar y, simplemente tenemos que subir el árbol desde x hasta que encontremos un nodo que tenga un hijo izquierdo de su padre.

El tiempo de ejecución de `Tree_Successor` en un árbol de altura h es $\boldsymbol{O}(h)$, ya que tenemos que seguir un camino simple por toda una rama del árbol. El procedimiento `Tree_Predecessor` es simétrico y también corre en tiempo $\boldsymbol{O}(h)$.

3.2.3. Insertar y borrar

Las operaciones de insertar y borrar ocasionan que el árbol binario cambie su conformación. La estructura de datos debe ser modificada para reflejar este cambio, pero de tal forma que las propiedades del árbol binario continuen siendo ciertas. Como veremos, modificar el árbol para insertar un nuevo elemento es relativamente un procedimiento directo, pero borrar un elemento es algo un poco más laborioso.

Insertar

Para insertar un nuevo valor ν en el árbol binario T, utilizamos el procedimiento `Tree_Insert`. El procedimiento toma un nodo z donde $z.key = \nu$, $z.left =$ `NIL` y $z.right =$ `NIL`. Se modifica T y algunos atributos de z de tal forma que z queda en el lugar adecuado.

```
Tree_Insert(T,z)
  y ←∼ NIL
  x ←∼ T.root
  while x≠ NIL
    y ←∼ x
    if z.key < x.key
      x ←∼ x.left
    else
      x ←∼ x.right
  z.p ←∼ y
  if y = NIL
    T.root ←∼ z
  elif z.key < y.key
    y.left ←∼ z
  else
    y.right ←∼ z
```

Al igual que en el procedimiento `Tree_Search`, este algoritmo `Tree_Insert` empieza en la raíz del árbol y el apuntador x traza una simple ruta que desciende el árbol hasta encontrar el valor `NIL` para reemplazarlo con el valor de entrada z. El procedimiento mantiene un apuntador y como el padre de x. Después de la inicialización, el ciclo `while` causa que esos dos apuntadores desciendan el árbol, llendo a la izquierda o a la derecha dependiendo de la comparación de $z.key$ con $x.key$, hasta que x sea `NIL`. Este `NIL` ocupa la posición donde deseamos colocar el valor de entrada z. Necesitamos el apuntador y, porque cuando encontrmos `NIL` donde z pertenece, la búsqueda ha avanzado un paso más allá del nodo que secesita ser cambiado.

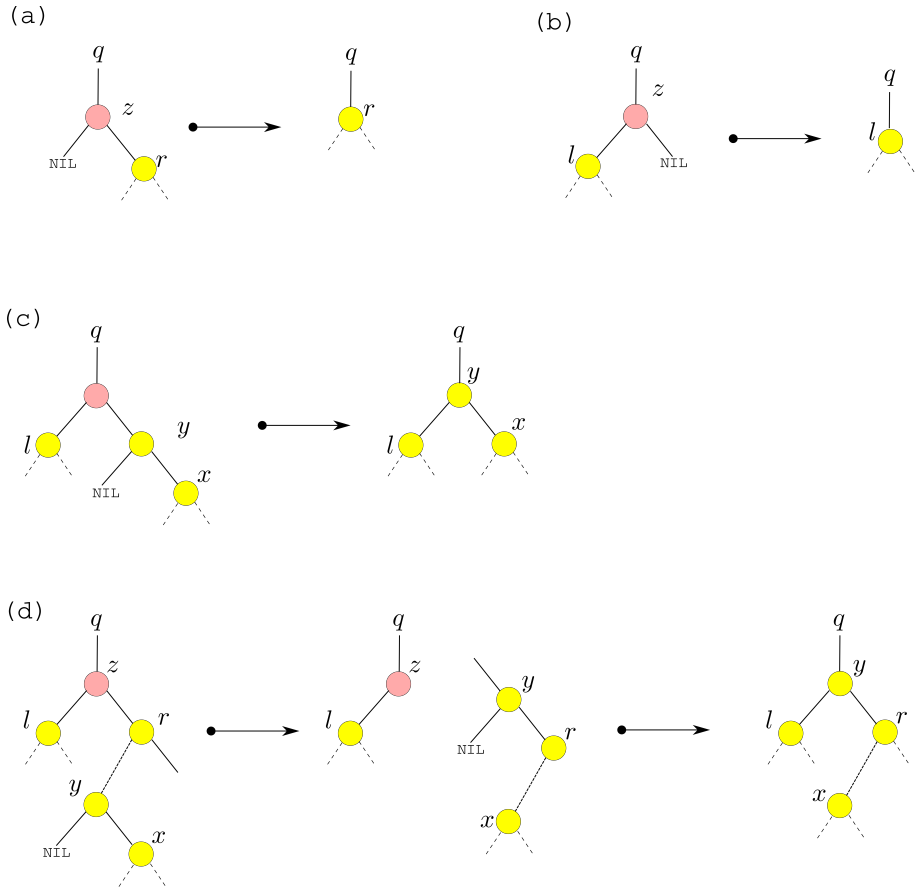
El procedimiento `Tree_Insert` corre en tiempo $\boldsymbol{O}(h)$ en un árbol de altura h.

Borrar

La estrategia general para borrar un nodo z de un árbol binario de búsqueda T tiene tres casos básicos, pero como veremos, uno de esos casos es un poco truculento.

- Si z no tiene hijos, entonces simplemente lo quitamos modificando su padre para reemplazar z con `NIL` como su hijo.
- Si z tiene solamente un hijo, entonces elevamos ese hijo para tomar la posición de z en el árbol modificando el padre de z para que apunte al hijo de z.
- Si z tiene los dos hijos, entonces encontramos el sucesor de z que es y –quien debe estar en el subárbol derecho de z– y que y tome la posición de z en el árbol. El resto del original subárbol derecho de z se convierte en el nuevo subárbol derecho de y, y el subárbol izquierdo de z se convierte en el nuevo subárbol izquierdo de y. Este caso es el truculento porque, como veremos tiene que ver que y sea el hijo derecho de z.

El procedimiento para borrar un nodo dado z del árbol binario de búsqueda T, toma como argumentos el apuntador a T y z. Se organizan sus casos un poco diferente de los tres casos delineados antes considerando los cuatro casos que se muestran en la siguiente figura.



- Si z no tiene hijo izquierdo [parte (a) en la figura], entonces reemplazamos z por su hijo derecho, que puede o no ser `NIL`. Cuando el hijo derecho de z es `NIL`, este caso se trata de acuerdo a la situación en que z no tiene hijos. Cuando el hijo derecho de z no es `NIL`, es precisamente la situación en que z tiene un solo hijo, que es su hijo derecho.
- Si z tiene solo un hijo, que es el hijo izquierdo [parte (b) en la figura], entonces reemplazamos z por su hijo izquierdo.
- De otro modo, z tiene ambos hijos. Encontramos el sucesor de z, que es y, que lo encontramos en el subarbol derecho de z y no tiene hijo izquierdo. Ahora tenemos que separar a y de su posición actual y reemplazar z en el árbol.

- Si y es el hijo derecho de z [parte (c)], entonces reemplazamos z por y, dejando quieto el hijo derecho de y.
- De lo contrario, y se encuentra dentro del subárbol derecho de z, pero no es el hijo derecho de z. En este caso, primero reemplazamos a y por su propio hijo derecho, y luego reemplazamos a z por y.

Para mover subárboles dentro del `ABB`, definimos el algoritmo `Transplant`, que reemplaza un subárbol como uno de los hijos de su padre con otro subárbol. Cuando `Transplant` reemplaza el subárbol con raíz en el nodo u con el subárbol con raíz en el nodo v, el padre de u se convierte en el padre de v el padre de u termina teniendo a v como su propio hijo.

```
1  Transplant(T,u,v)
2    if u.p= NIL
3      T.root ←≺ v
4    elif u = u.p.left
5      u.p.left ←≺ v
6    else
7      u.p.right ←≺ v
8    if v ≠ NIL
9      v.p ←≺ u.p
```

Ya teniendo el procedimiento para transplantar árboles, aquí esta el procedimiento que borra un nodo z del `ABB` T:

```
1  Tree_Delete(T,z)
2    if z.left = NIL
3      Transplant(T,z,z.right)
4    elif z.right = NIL
5      Transplant(T,z,z.left)
6    else
7      y ←≺ Tree_Minimum(z.right)
8      if y.p ≠ z
9        Transplant(T,y,y.right)
10       y.right ←≺ z.right
11       y.right.p ←≺ y
12     Transplant(T,z,y)
13     y.left ←≺ z.left
14     y.left.p ←≺ y
```

El procedimiento `Tree_Delete` se ejecuta en cuatro casos: las líneas 2–3 son para el caso en que z no tiene hijo izquierdo y las líneas 4–5 para cuando z tiene un hijo izquierdo pero no tiene hijo derecho. Las líneas 7–14 consideran los otros dos casos, cuando z tienen ambos hijos. La línea 6 encuentra el nodo y, quien es el sucesor de z. Como z tiene un subárbol derecho no vacío, su sucesor debe ser el nodo en el aubárbol con la clave más pequeña; por eso se invoca `Tree_Minimum(z.right)`. Como notamos antes, y no tiene hijo izquierdo. Vamos ahora a cambiar a y de su lugar actual al lugar en donde estaba z. Si y es el hijo derecho de z, entonces las líneas 12 – 14 reemplazan z como un hijo de su padre por y y reemplaza el hijo derecho de y por el hojo izquierdo de z. Si y no es el hijo izquierdo de z, las líneas 9 – 11 reemplazan y como un hijo de su padre por el hijo derecho de y y cambian el hijo derecho de z en el hijo derecho de y, y las líneas 12 – 14 reemplazan z como un hijo de su padre por y y reemplazan el hijo izquierdo de y por el hijo izquierdo de z.

Cada línea de `Tree_Delete`, incluyendo las invocaciones a `Transplant`, toman un tiempo constante, excepto por la llamada a `Tree_Minimum` en la línea 7, que corre en tiempo $\boldsymbol{O}(h)$ en un árbol de altura h.

En resumen, se ha probado el siguiente teorema.

Teorema 2. *Podemos implementar las operaciones `Insert` y `Delete` de tal forma que cada una corra en tiempo $\boldsymbol{O}(h)$ para un `ABB` de altura h.*

Ejercicios

Criterio de evaluación

Para que seas conciente de cómo se evalúan y califican estos ejercicios, considera el siguiente criterio que otorga cierto número de puntos por cada ejercicio:

3: Un ejercicio entregado a tiempo y sin alguna falla. Un ejercicio de programación corre a la perfección; sigue fielmente las instrucciones y su notación, y tiene su contrato correctamente escrito.

2: Un ejercicio mayormente correcto entregado a tiempo. Un ejercicio de programación o bien no tiene su contrato; o bien ha cambiado identificadores o es parcialmente correcto.

1: Un ejercicio entregado a destiempo o mayormente equivocado. Un ejercicio de programación no tiene su contrato y ha cambiado identificadores y es parcialmente correcto.

0: Un ejercicio completamente equivocado o no se entrega.

La calificación de la lista de ejercicios se obtiene al ponderar el número de puntos alzanzado en una escala de 0 a 10.

1. Escribe un programa en `Python` que se llame `pila.py`. El programa debe tener la definición de una pila de números enteros, modelada como una clase llamada `Stack`, con un constructor llamado `sz` que determinará el tamaño del arreglo `S` de tamaño `sz`, que es propio de la pila. Además del atributo `S`, la pila debe tener el atributo `top` que es un valor entero para marcar el elemento que debe ser removido. La pila debe tener los métodos mostrados en el texto [página 2], pero además un método `Show` que debe imprimir en un arreglo, los elementos que hay actualmente en la pila, sin mostrar aquellas localidades vacías [o que han sido desocupadas al hacer `pop` a la pila].

```
S = Stack(20)
In[]: S
Out[]: <__main__.Stack at 0x7f543822c490>
In[]: S.Push(3)
In[]: S.Push(4)
In[]: S.Push(5)
In[]: S.Push(6)
In[]: S.Show()
Out[]: [3, 4, 5, 6]
In[]: S.Pop()
Out[]: 6
In[]: S.Show()
Out[]: [3, 4, 5]
```

2. Explica mediante un algoritmo cómo puedes implementar dos pilas en un mismo arreglo, sin que las pilas se sobrepongan. Las pilas deben considerarse llenas cuando la cantidad de elementos almacenados en ambas pilas sea igual al tamaño del arreglo. Cada pila puede almacenar elementos mientras haya algún espacio en el arreglo.

3. Escribe una clase llamada `Queue` que debe tener un constructor llamado `sz` que servirá para establecer el tamaño del arreglo con el que se modelará la cola. La clase `Queue` debe tener los atributos `sz` que es un entero; `Q` que es un arreglo de tamaño `sz`; `head` y `tail` también enteros en el rango del arreglo. Los métodos de la clase deben ser `Enqueue` que recibe un número entero, que deberá ser agregado a la cola; `Dequeue` que no recibe parámetro alguno, pero debe devolver el elemento que se encuentra al frente de la cola. Además de estos dos métodos, debes agregar otros tres: un método `Empty` que sirva para verificar si la cola esta vacía; un método `Full` que determina si la cola esta llena y un método `Show` que muestra exclusivamente

los elementos de la cola, sin contar aquellos elementos del arreglo que no esten considerados en la cola.

```
In []:print("Inicio")
  ...:Q = Queue(5)
  ...:Q.Enqueue(7)
  ...:Q.Enqueue(3)
  ...:Q.Show()
  ...:Q.Enqueue(8)
  ...:Q.Enqueue(1)
  ...:print("La cola esta vacia?", Q.Empty())
  ...:print("La cola esta llena?", Q.Full())
  ...:Q.Show()
  ...:x = Q.Dequeue()
  ...:print("El elemento", x, "fue removido:")
  ...:Q.Show()
  ...:x = Q.Dequeue()
  ...:print("El elemento", x, "fue removido:")
  ...:Q.Show()
Inicio
[7, 3]
La cola esta vacia? False
La cola esta llena? False
[7, 3, 8, 1]
El elemento 7 fue removido:
[3, 8, 1]
El elemento 3 fue removido:
[8, 1]

In []:
```

4. Explica mediante un algoritmo cómo puedes implementar una pila utilizando dos colas. Analiza el tiempo de ejecución de las operaciones `Push` y `Pop` de esta nueva implementación.

5. Explica mediante algoritmos, cómo se puede implementar una pila utilizando una lista simplemente ligada. Debes escribir los algoritmos para las operaciones `Pop` y `Push` de la pila en esta nueva implementación.

6. Dibuja el árbol binario que tiene raíz en el nodo con índice 6 que está representado por los siguientes atributos:

índice	*key*	*left*	*right*
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

7. Escribe un programma llamado `ABB.py` que sirva para demostrar todas las operaciones dinámicas de los árboles binarios de búsqueda. El programa debe tener al menos la clase `ABB` y la clase `nodo` [la clase `nodo` es como aparece en el texto]. La clase `ABB` es para definir objetos de tipo árbol binario de búsqueda, y debe tener al menos el atributo constructor:

   `root`: cuyo valor por defecto debe ser `None`, pero en general toma un objeto `nodo`.

   El `ABB` debe tener los métodos:

   +`Insert` (`val: Entero`) : `None`; Método para insertar un valor en el árbol.

   +`Inorder_Walk` (`x:nodo`) : `None`; Método para mostrar el recorrido inorden del árbol a partir del nodo `x`.

+`Search` (`x:node,k:entero`) : `nodo`|`None`; Método para buscar un nodo en el árbol. Devuelve el nodo encontrado o bien devuelve `None` si no se encuentra.

+`Minimum` (`x:node`) : `nodo`|`None`; Encuentra el nodo que tiene la clave mínima de todos los nodos del `ABB`. En caso de que el árbol sea vacío se devuelve `None`.

+`Maximum` (`x:node`) : `nodo`|`None`; Encuentra el nodo que tiene la clave mínima de todos los nodos del `ABB`. En caso de que el árbol sea vacío se devuelve `None`.

+`Successor` (`x:node`) : `nodo`; Encuentra el nodo siguiente al nodo dado, de acuerdo al orden inducido por las claves. En caso de que el árbol sea vacío se devuelve `None`.

+`Predecessor` (`x:node`) : `nodo`; Encuentra el nodo anterior al nodo dado, de acuerdo al orden inducido por las claves. En caso de que el árbol sea vacío se devuelve `None`.

+`Delete` (`val: Entero`) : `None`; Método para borrar un valor en el árbol.

+`Transplant` (`u: nodo, v: nodo`) : `None`; Método para transplantar un subárbol con raíz en `u` como hijo de `v`.

```
In[]: T = ABB()
 ...: T.Insert(7)
 ...: T.Insert(3)
 ...: T.Insert(45)
 ...: T.Insert(2)
 ...: T.Insert(12)
 ...: T.Insert(84)
 ...: T.Insert(23)
 ...: T.Insert(46)
 ...: T.Insert(29)
 ...: T.Inorder_Walk(T.root)
 ...: b = T.Search(T.root, 12)
 ...: print(b)
 ...: c = T.Search(T.root, 78)
 ...: print(T.Minimum())
 ...: print(T.Maximum())
 ...: d = T.Search(T.root, 45)
 ...: print(T.Successor(d))
 ...: n23 = T.Search(T.root, 23)
 ...: T.Delete(23)
 ...: T.Inorder_Walk(T.root)
out[]:
2
3
7
12
23
29
45
46
84
[None<| 12 |>23]
None
[None<| 2 |>None]
[46<| 84 |>None]
[None<| 46 |>None]
2
3
7
12
29
45
46
84
```