Árboles Binarios de Búsqueda

Introducción
- Generalidades
- Definición

Consultas a un ABB
- Introducción
- Búsqueda
- Mínimo y máximo
- Sucesor y predecesor

Modificaciones a un ABB
- Introducción
- Inserción
- Eliminación

ABBs balanceados
- Introducción
- Rotaciones

Árboles AVL
- Representación
- Inserción

Árboles rojo-negros
- Representación
- Inserción
- Eliminación

Introducción

GENERALIDADES. Los ABBs permiten ejecutar eficientemente mu-chas operaciones de conjuntos dinámicos:

- SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, DELETE
- pueden usarse como *diccionarios* y como *colas de prioridades*

Las operaciones básicas toman tiempo proporcional a la altura del árbol; esto es, para un ABB con n nodos:

- $\Theta(\log n)$ en el peor caso, si está completo
- $\Theta(n)$ en el peor caso, si, en cambio, es una cadena lineal
- la altura de un ABB construido aleatoriamente es $\boldsymbol{O}(\log n)$ y las operaciones toman tiempo $\Theta(\log n)$

No siempre podemos garantizar que los ABBs son construidos aleatoria-mente, pero hay "variedades" para las cuales sí puede garantizarse que el desempeño de las operaciones básicas es bueno.

DEFINICIÓN. Un ABB es un árbol binario que podemos representar mediante una estructura de datos ligada en la cual cada nodo es un objeto que contiene:

- un campo *key*, en que se almacena la clave del objeto
- los campos L, R y P que apuntan a los nodos correspondientes a su hijo izquierdo, su hijo derecho, y su padre

Las claves están almacenadas de manera de satisfacer la ***propiedad de ABB***:

Sea x un nodo en un ABB;

- si y es un nodo en el subárbol izquierdo de x, entonces $key[y] \leq key[x]$
- si y es un nodo en el subárbol derecho de x, entonces $key[x] \leq key[y]$

Usando esta propiedad, podemos listar las claves en un ABB en orden creciente usando el algoritmo recursivo ***recorrido inorden***:

la raíz de un subárbol se lista *después de listar* los valores en su subárbol izquierdo y *antes de listar* aquellos en su subárbol derecho

```
INORDEN(x) :
   if (x ≠ NIL)
      INORDEN(L[x])
      listar key[x]
      INORDEN(R[x])
```

Para listar todas las claves de un ABB T usando INORDEN, invocamos INORDEN($root[T]$).

Consultas a un ABB

INTRODUCCIÓN. La operación más común sobre un ABB es buscar una clave almacenada en el árbol (SEARCH).

Como veremos, además de SEARCH, las otras operaciones de consulta — MIN, MAX, SUCCESSOR, y PREDECESSOR — también corren en tiempo $O(h)$ en un ABB de altura h.

BÚSQUEDA. La búsqueda de un nodo con clave k a partir de un nodo x sigue una ruta hacia abajo en el árbol:

Para cada nodo x que encuentra, compara la clave k con $key[x]$:

- si $k = key[x]$, la búsqueda termina
- si $k < key[x]$, la búsqueda sigue en el subárbol izquierdo de x (la propiedad de ABB implica que k no puede estar en el subárbol derecho)
- si $k > key[x]$, la búsqueda sigue en el subárbol derecho de x (por la propiedad de ABB)

```
SEARCH(x, k) :
    if (x == NIL ∨ k == key[x])
        return x
    if (k < key[x])
        return SEARCH(L[x], k)
    else
        return SEARCH(R[x], k)
```

MÍNIMO Y MÁXIMO. El nodo de un ABB cuya clave es la mínima se encuentra siguiendo los punteros L (a los hijos izquierdos), desde la raíz hasta encontrar NIL.

La propiedad de ABB garantiza que MIN es correcto:

- Si x no tiene subárbol izquierdo, entonces, como todas las claves en su subárbol derecho son $\geq key[x]$, la clave mínima del árbol cuya raíz es x es $key[x]$.
- Si x tiene subárbol izquierdo, entonces, como todas las claves en su subárbol derecho son $\geq key[x]$ y todas las claves en el subárbol izquierdo son $\leq key[x]$, la clave mínima del árbol cuya raíz es x está en el subárbol cuya raíz es $L[x]$.

La argumentación para MAX es simétrica.

```
MIN(x) :
    while (L[x] ≠ NIL)
        x = L[x]
    return x
```

```
MAX(x) :
    while (R[x] ≠ NIL)
        x = R[x]
    return x
```

SUCESOR Y PREDECESOR. En un ABB, el sucesor de un nodo x es el nodo con la clave más pequeña mayor que $key[x]$ (si todas las claves almacenadas en el ABB son distintas):

- Si x tiene un subárbol derecho ($\neq$ NIL), el sucesor de x es el nodo de más a la izquierda en este subárbol.
- Si el subárbol derecho de x es NIL, el sucesor de x es el ancestro más bajo de x cuyo hijo izquierdo es también un ancestro de x.

```
SUCCESSOR(x) :
    if (R[x] ≠ NIL)
        return MIN(R[x])
    y = P[x]
    while (y ≠ NIL ∧ x == R[y])
        x = y
        y = P[y]
    return y
```

La operación PREDECESSOR es simétrica.

Modificaciones a un ABB

INTRODUCCIÓN. Las operaciones de inserción y eliminación modifi-can el conjunto dinámico representado por el ABB, pero preservan la propiedad de ABB.

Ambas operaciones corren en tiempo $O(h)$ en un árbol de altura h.

INSERCIÓN. INSERT recibe un nodo z con una cierta clave $key[z]$ y con $L[z] = R[z] =$ NIL; y a partir de la raíz del árbol T sigue una ruta hacia abajo:

- el puntero x sigue la ruta
- el puntero y apunta siempre al padre de x
- el ciclo **while** hace bajar x y y hasta que x queda en NIL
- este NIL ocupa la posición en donde hay que insertar z

```
INSERT(T, z) :
    y = NIL
    x = root[T]
    while (x ≠ NIL)
        y = x
        if (key[z] < key[x])
            x = L[x]
        else x = R[x]
    P[z] = y
    if (y == NIL)
        root[T] = z
    else   if (key[z] < key[y])
                L[y] = z
           else R[y] = z
```

ELIMINACIÓN. Al eliminar un nodo z de un árbol T, DELETE conside-ra tres casos:

- z no tiene hijos — simplemente lo sacamos de T
- z tiene un solo hijo — lo reemplazamos por su hijo
- z tiene dos hijos — lo reemplzamos por su sucesor, eliminando a éste de su posición original

```
DELETE(T, z) :
    if (L[z] == NIL ∨ R[z] == NIL)
        y = z
    else y = SUCCESSOR(z)
    if (L[y] ≠ NIL)
        x = L[y]
    else x = R[y]
    if (x ≠ NIL) P[x] = P[y]
    if (P[y] == NIL)
        root[T] = x
    else   if (y == L[P[y]])
                L[P[y]] = x
           else R[P[y]] = x
    if (y ≠ z) key[z] = key[y]
    return y
```

ABBs Balanceados

Introducción. En un ABB de altura h, las operaciones de conjunto dinámico toman tiempo $\boldsymbol{O}(h)$:

- son rápidas si la altura del árbol es pequeña **…** pero
- pueden ser tan lentas como en el caso de una lista ligada.

¿Podemos insertar y eliminar claves (nodos) en un ABB con n nodos de modo que:

- la altura del árbol sea garantizadamente $\boldsymbol{O}(\log n)$; y
- el costo de tales operaciones no sea excesivo?

La respuesta es afirmativa y está basada en la idea de *árboles binarios de búsqueda balanceados* (ABBBs).

Un ABB es un ABBB si cumple una *propiedad de ABBB*; p.ej.:

- en los **árboles AVL** *para cada nodo la diferencia en altura de sus dos subárboles es a lo más* 1;
- en los **árboles rojo-negros** *ningún camino desde la raíz a una hoja es más del doble de largo que otro*.

ROTACIONES. Como INSERT y DELETE modifican el árbol, es posible que el árbol resultante no cumpla con la propiedad de árbol balanceado, en cuyo caso es necesario modificar la estructura del árbol para restaurar esta propiedad.

Modificamos la estructura de punteros del árbol mediante *rotaciones*— operaciones locales que preservan la ordenación *inorden* de las claves del árbol:

> Cuando hacemos *rotación a la izquierda* sobre un nodo x, suponiendo que su hijo derecho y no es NIL, pivotamos alrededor de la conexión de x a y, convirtiendo a y en la nueva raíz del subárbol, a x en el hijo izquierdo de y, y al hijo izquierdo de y en el hijo derecho de x.

```
ROTAR-IZQ(T, x) :
    y = R[x]
    R[x] = L[y]
    if (L[y] ≠ NIL) P[L[y]] = x
    P[y] = P[x]
    if (P[x] == NIL)
        root[T] = y
    else   if (x == L[P[x]])
                L[P[x]] = y
           else R[P[x]] = y
    L[y] = x
    P[x] = y
```

ROTAR-IZQ, y similarmente ROTAR-DER, corren en tiempo $\boldsymbol{O}(1)$ —sólo modifican una cantidad constante de punteros.

Árboles AVL

Representación. Es similar a la de ABB; pero agregamos para cada nodo x un atributo *balance* que registra la diferencia entre las alturas de los dos subárboles de x:

- $balance[x] = -1 \Rightarrow$ subárbol izquierdo tiene mayor altura;
- $balance[x] = 0 \Rightarrow$ ambos subárboles tienen la misma altura;
- $balance[x] = +1 \Rightarrow$ subárbol derecho tiene mayor altura.

La implementación es idéntica a la de ABBs excepto para Insert y Delete — son más complejas porque deben devolver un árbol AVL.

Inserción. Insertamos el nuevo nodo como si el árbol fuera un ABB, pero

- la ruta desde la raíz del árbol hasta este nuevo nodo es nuestra *ruta de búsqueda*, y
- el nodo de menor altura en esta ruta cuyo balance sea –1 o +1 es nuestro nodo *pivote*.

Tenemos tres casos:

Caso 1. No hay pivote; todos los nodos en la ruta de búsqueda están balanceados:

> El árbol se ajusta actualizando los valores del *balance* de todos los nodos en la ruta de búsqueda.

Caso 2. Hay un pivote y el subárbol al cual se agrega el nuevo nodo es el de menor altura:

> El árbol se ajusta actualizando los valores del *balance* de todos los nodos en el camino de búsqueda a partir del pivote.

Caso 3. Hay un pivote y el subárbol al cual se agrega el nuevo nodo es el de mayor altura.

El árbol resultante no es AVL y es necesario ajustar su estructura.

Suponiendo que el *balance* del pivote es –1 (el subárbol izquierdo es un nivel más alto que el subárbol derecho), tenemos dos subcasos:

- **Subcaso 3.1**. Si el nodo fue agregado al subárbol izquierdo del hijo izquierdo del pivote, entonces hacemos una rotación a la derecha sobre el pivote.
- **Subcaso 3.2**. Si el nodo fue agregado al subárbol derecho del hijo izquierdo del pivote, entonces hacemos dos rotaciones:
  - primero, una rotación a la izquierda sobre el hijo izquierdo del pivote (sin involucrar al pivote ni a su subárbol derecho)
  - luego, una rotación a la derecha sobre el pivote.
- En ambos subcasos, debemos actualizar los valores del *balance* de todos los nodos en la ruta desde el nodo que ahora ocupa la posición del pivote hasta el nuevo nodo.
- Si el balance del pivote es originalmente +1 tenemos otros dos subcasos simétricos.

Árboles Rojo-Negros

REPRESENTACIÓN. Es similar a la de ABBs, pero agregamos para cada nodo un atributo *color* que puede ser ROJO o NEGRO.

Además, consideramos todos los nodos del árbol como nodos internos, y consideramos los punteros con valor NIL como punteros a hojas (nodos externos).

Si restringimos la forma en que se pinta los nodos en cualquier ruta desde la raíz hasta una hoja, podemos asegurar que ninguna de estas ruta sea más del doble de larga que cualquiera otra.

Un ABB es un árbol rojo-negro si satisface las siguientes 4 propiedades:

1) Todo nodo es rojo o negro.
2) Toda hoja (NIL) es negra.
3) Si un nodo es rojo, entonces sus dos hijos son negros.
4) Todas las rutas simples desde un mismo nodo a cualquiera de sus hojas descendientes tienen el mismo número de nodos negros.

Puede demostrarse, a partir de estas propiedades, que la altura de un árbol rojo-negro con n nodos internos es a lo más $2 \log(n + 1)$.

INSERCIÓN. La inserción de un nodo en un árbol rojo-negro de n nodos puede hacerse en tiempo $\boldsymbol{O}(\log n)$.

INSERT-RN usa INSERT para insertar un nodo x en un árbol T, luego pinta x rojo, y finalmente restaura la calidad de árbol rojo-negro; esto puede ser necesario ya que al pintar x de rojo:

- las propiedades 1 y 2, obviamente, siguen vigentes;
- la propiedad 4 sigue vigente—x, que es rojo, reemplaza a una hoja (NIL) negra, pero ambos hijos de x son hojas negras;
- *la propiedad 3 deja de estar vigente si el padre de x es rojo*.

Para restaurar la calidad de árbol rojo-negro de T, INSERT-RN distingue 3 casos:

Caso 1. El padre y el tío de x son rojos: ambos se pintan negros y el abuelo de x se pinta rojo; éste pasa a ser el nuevo x.

Caso 2. El padre de x es rojo, el tío de x es negro, y x es el hijo derecho de su padre: se realiza una rotación a la izquierda sobre el padre de x, que pasa a ser el nuevo x, y se obtiene el caso 3.

Caso 3. El padre de x es rojo, el tío de x es negro, y x es el hijo izquierdo de su padre: se realiza una rotación a la derecha sobre el abuelo de x, resolviéndose el problema.

```
INSERT-RN(T, x) :
    INSERT(T, x)
    color[x] = ROJO
    while (x ≠ root[T] ∧ color[P[x]] == ROJO)
        if (P[x] == L[P[P[x]]])
                y = R[P[P[x]]]
                if (color[y] == ROJO)
                        color[P[x]] = NEGRO                    {caso 1}
                        color[y] = NEGRO                       {caso 1}
                        color[P[P[x]]] = ROJO                  {caso 1}
                        x = P[P[x]]                            {caso 1}
                else   if (x == R[P[x]])
                                x = P[x]                       {caso 2}
                                ROTAR-IZQ(T, x)                {caso 2}
                    color[P[x]] = NEGRO                        {caso 3}
                    color[P[P[x]]] = ROJO                      {caso 3}
                    ROTAR-DER(T, P[P[x]])                      {caso 3}
        else {igual al caso "if" pero con R y L intercambiados}
    color[root[T]] = NEGRO
```

ELIMINACIÓN. La eliminación de un nodo de un árbol rojo-negro de n nodos también toma tiempo $\boldsymbol{O}(\log n)$.

Hay algunas diferencias entre DELETE-RN y DELETE:

- Para simplificar las condiciones de borde, usamos un centinela $nil[T]$ para representar NIL:
  - reemplazamos las referencias a NIL por referencias a $nil[T]$
  - al manipular un hijo NIL de un nodo w, primero asignamos w a $P[nil[T]]$
  - eliminamos la prueba de si x—el único hijo de y si y tenía uno, o $nil[T]$—es NIL y asignamos $P[x] = P[y]$ incondicionalmente
- Llamamos a RESTAURAR-RN si y — el nodo efectivamente eliminado — es negro:
  - el número de nodos negros en al menos un camino desde un no-do a una hoja cambió
  - posiblemente un nodo rojo es hijo de otro nodo rojo.

```
DELETE-RN(T, z) :
    if (L[z] == nil[T] ∨ R[z] == nil[T])
        y = z
    else y = SUCCESOR(z)
    if (L[y] ≠ nil[T])
        x = L[y]
    else x = R[y]
    P[x] = P[y]
    if (P[y] == nil[T])
        root[T] = x
    else   if (y == L[P[y]])
                L[P[y]] = x
           else R[P[y]] = x
    if (y ≠ z) key[z] = key[y]
    if (color[y] == NEGRO) RESTAURAR-RN(T, x)
    return y
```

```
RESTAURAR-RN(T, x) :
    while (x ≠ root[T] ∧ color[x] == NEGRO)
        if (x == L[P[x]])
                w = R[P[x]]
                if (color[w] == ROJO)
                        color[w] = NEGRO                        {caso 1}
                        color[P[x]] = ROJO                      {caso 1}
                        ROTAR-IZQ(T, P[x])                      {caso 1}
                        w = R[P[x]]                             {caso 1}
                if (color[L[w]] == NEGRO ∧ color[R[w]] == NEGRO)
                        color[w] = ROJO                         {caso 2}
                        x = P[x]                                {caso 2}
                else  if (color[R[w]] == NEGRO)
                                color[L[w]] = NEGRO             {caso 3}
                                color[w] = ROJO                 {caso 3}
                                ROTAR-DER(T, w)                 {caso 3}
                                w = R[P[x]]                     {caso 3}
                        color[w] = color[P[x]]                  {caso 4}
                        color[P[x]] = NEGRO                     {caso 4}
                        color[R[w]] = NEGRO                     {caso 4}
                        ROTAR-IZQ(T, P[x])                      {caso 4}
                        x = root[T]                             {caso 4}
        else {igual al caso "if" pero con L y R intercambiados}
    color[x] = NEGRO
```

El ciclo **while** sólo se ejecuta si x es negro; si x es rojo, entonces simple-mente se pinta negro al final de RESTAURAR-RN.

Si x es negro y su hermano w es rojo, se tiene el **caso 1**:

una rotación a la izquierda en torno al padre de x nos lleva al **caso 2**, **3** ó **4** — x y su nuevo hermano w son ambos negros.

Los **casos 2**, **3** y **4** se distinguen según el color de los hijos de w:

- en **2**, ambos son negros
- en **3**, el hijo izquierdo es rojo y el hijo derecho es negro
- en **4**, el hijo derecho es rojo (el color del hijo izquierdo no importa)