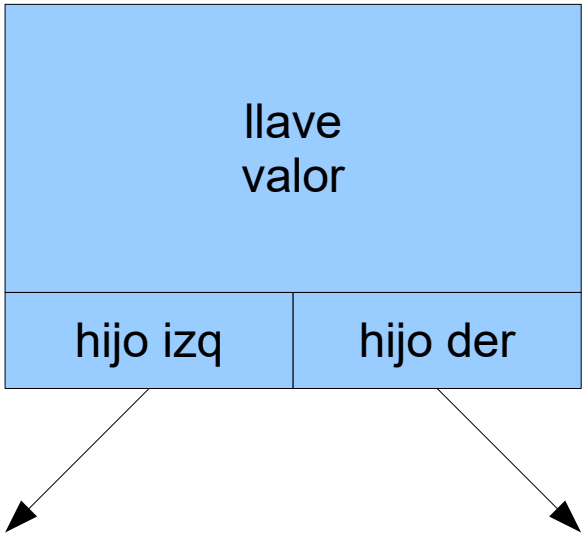
Árboles de Búsqueda Binarios

Binary Search Trees (BST)

Árboles Binarios

- Similar a como un nodo de una lista contiene una referencia a su sucesor, un nodo de un árbol se construye teniendo dos referencias, que llamaremos hijos izquierdo y derecho.




```
class Nodo {
    Key llave;
    Value valor;
    Nodo izq, der;
    ...
}
```

Estructura del árbol

- Un nodo contiene como máximo dos referencias: Árbol binario.
- Si ambas referencias son nulas, el nodo es una hoja.
- Todo nodo esta referenciado desde un nodo padre, salvo el nodo raíz.
- La estructura se puede describir recursivamente: Un nodo contiene referencias a dos subárboles binarios. Se entiende la referencia `null` como el árbol vacío.

Árbol de búsqueda

Definición:

Un árbol binario de búsqueda es un árbol binario que contiene llaves de tipo *Comparable* y que cumple la restricción que para todo nodo, su llave es mayor a las llaves del subárbol izquierdo y menor a las subllaves del subárbol derecho.

Implementación básica

- Se mantienen los nodos como clase interna y se mantiene también el tamaño del subárbol correspondiente a cada nodo.
- En el caso de referencias nulas (árboles vacíos) su tamaño es cero.
- Para todo nodo se garantiza:

```
size(x) = size(x.left) + size(x.right) + 1
```

- Una variable de instancia `root` mantiene la referencia a la raíz del árbol.

Implementación

```
public class BST<Key extends Comparable<Key>, Value> {
    private Node root;             // root of BST
    private class Node {
        private Key key;           // sorted by key
        private Value val;         // associated data
        private Node left, right;  // left and right subtrees
        private int N;             // number of nodes in subtree
        public Node(Key key, Value val, int N) {
            this.key = key;
            this.val = val;
            this.N = N;
        }
    }

    public BST() {
    }

    public boolean isEmpty() {
        return size() == 0;
    }

    public int size() {
        return size(root);
    }

    private int size(Node x) {
        if (x == null) return 0;
        else return x.N;
    }

...
```

Métodos `get` / `put`

- `get` : Se compara la llave buscada con la del nodo actual. Si es mayor, buscar en el subárbol derecho. Si es menor, en el izquierdo. Si son iguales ya se encontró el valor. Si el subárbol es vacío, la llave no está.
- `put` : Similar a la búsqueda. Si se encuentra la llave, se reemplaza el valor. Sino, si la llave es menor, se agrega al hijo izquierdo, de lo contrario al hijo derecho. Si el hijo es nulo, se agrega un nuevo nodo en esta posición.

Continuación

```
public Value get(Key key) {
    return get(root, key);
}

private Value get(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if      (cmp < 0) return get(x.left, key);
    else if (cmp > 0) return get(x.right, key);
    else              return x.val;
}

public void put(Key key, Value val) {
    if (key == null) throw new NullPointerException("first argument to put() is null");
    if (val == null) {
        delete(key);
        return;
    }
    root = put(root, key, val);
    assert check();
}

private Node put(Node x, Key key, Value val) {
    if (x == null) return new Node(key, val, 1);
    int cmp = key.compareTo(x.key);
    if      (cmp < 0) x.left  = put(x.left,  key, val);
    else if (cmp > 0) x.right = put(x.right, key, val);
    else              x.val   = val;
    x.N = 1 + size(x.left) + size(x.right);
    return x;
}
```

Código fuente

Análisis de eficiencia

- put / get realizan un recorrido desde la raíz hacia las hojas. El número de comparaciones es el mismo en ambas operaciones.
- El número de comparaciones depende de la estructura particular del árbol.
- Mejor caso: árbol perfectamente balanceado. # de comparaciones es la altura del árbol: ~lgN
- Peor caso: Todos los nodos dispuestos en una larga cadena. # comp ~ N

Análisis de caso medio (*hit*)

Proposición:

El número de comparaciones de caso medio es ~2lnN.

Dem:

Sea C_N la longitud media del camino de la raíz a las hojas para un BST de N nodos. En cada nodo se hace una comparación, más las comparaciones que se hacen en el subárbol hasta las hojas. Se tiene entonces que el número promedio de comparaciones es:

$$C_N = \frac{1}{N}\left((1+(C_0+C_{N-1})/2)+(1+(C_1+C_{N-2})/2)+\ldots+(1+(C_{N-1}+C_0)/2)\right)$$

Considerando C_0=1, C_1=1, se puede verificar que C_N ~ 2ln(N).

Análisis de caso medio (*miss*)

- En caso de no encontrarse la llave, el número de comparaciones es el número anterior más una.
- Se concluye en general que las búsquedas y adiciones requieren un número de comparaciones

$$\sim 2 \ln(N) = 1.39 \lg(N)$$

- Es decir un 39% superior a la búsqueda binaria. Pero comparativamente, la inserción en un arreglo ordenado es ~N.

Operaciones basadas en el orden de las llaves

- El BST mantiene las llaves ordenadas. Resulta sencillo implementar las operaciones
  - min, max
  - floor, ceil
  - select, rank

Implementación `min`, `floor`

```
public Key min() {
    if (isEmpty()) throw new NoSuchElementException("called min() with empty symbol table");
    return min(root).key;
}

private Node min(Node x) {
    if (x.left == null) return x;
    else                return min(x.left);
}

public Key floor(Key key) {
    if (key == null) throw new NullPointerException("argument to floor() is null");
    if (isEmpty()) throw new NoSuchElementException("called floor() with empty symbol table");
    Node x = floor(root, key);
    if (x == null) return null;
    else return x.key;
}

private Node floor(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp == 0) return x;
    if (cmp <  0) return floor(x.left, key);
    Node t = floor(x.right, key);
    if (t != null) return t;
    else return x;
}
```

Implementación `select`, `rank`

```
public Key select(int k) {
    if (k < 0 || k >= size()) throw new IllegalArgumentException();
    Node x = select(root, k);
    return x.key;
}

private Node select(Node x, int k) {
    if (x == null) return null;
    int t = size(x.left);
    if      (t > k) return select(x.left,  k);
    else if (t < k) return select(x.right, k-t-1);
    else            return x;
}

public int rank(Key key) {
    if (key == null) throw new NullPointerException("...");
    return rank(key, root);
}

private int rank(Key key, Node x) {
    if (x == null) return 0;
    int cmp = key.compareTo(x.key);
    if      (cmp < 0) return rank(key, x.left);
    else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);
    else              return size(x.left);
}
```

Operaciones de borrado

- `deleteMin` : Para el mínimo hacer un recorrido siempre buscando el hijo izquierdo, hasta encontrar un caso cuyo hijo izquierdo sea nulo. En este momento este es el mínimo. Se reemplaza por su hijo derecho (incluso si es nulo).
- `deleteMax` es similar siguiendo los enlaces a la derecha.

Implementación deleteMin

```
public void deleteMin() {
    if (isEmpty()) throw new NoSuchElementException("...");
    root = deleteMin(root);
    assert check();
}

private Node deleteMin(Node x) {
    if (x.left == null) return x.right;
    x.left = deleteMin(x.left);
    x.N = size(x.left) + size(x.right) + 1;
    return x;
}
```

Borrar nodo arbitrario

- Un nodo interno puede tener dos hijos. Para borrarlo, se debe reemplazar el nodo para mantener la estructura del árbol binario de búsqueda.
- Estrategia general: Reemplazar el nodo por su sucesor.
- Cinco pasos:
  1) t es el nodo a borrar
  2) encontrar x sucesor de t: x=min(t.right)
  3) x.right = deleteMin(t.right)
  4) x.left = t.left
  5) se actualiza tamaño y se devuelve x

Implementación `delete`

```
public void delete(Key key) {
    if (key == null) throw new NullPointerException("...");
    root = delete(root, key);
    assert check();
}

private Node delete(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if      (cmp < 0) x.left  = delete(x.left,  key);
    else if (cmp > 0) x.right = delete(x.right, key);
    else {
        if (x.right == null) return x.left;
        if (x.left  == null) return x.right;
        Node t = x;
        x = min(t.right);
        x.right = deleteMin(t.right);
        x.left = t.left;
    }
    x.N = size(x.left) + size(x.right) + 1;
    return x;
}
```

Consultas por rango

- Idea general: Un recorrido “en orden” imprime todas las llaves en el árbol de menor a mayor.

```
void recorridoEnOrden(Nodo x) {
   if (x==null) return;
   recorridoEnOrden(x.left);
   StdOut.println(x.key);
   recorridoEnOrden(x.right);
}
```

- Para obtener las llaves en un rango, se restringe los llamados recursivos mientras que `lo<x.key` y `x.key<hi`.

Implementación de la consulta por rango

```
public Iterable<Key> keys(Key lo, Key hi) {
    Queue<Key> queue = new Queue<Key>();
    keys(root, queue, lo, hi);
    return queue;
}

private void keys(Node x, Queue<Key> queue, Key lo, Key hi) {
    if (x == null) return;
    int cmplo = lo.compareTo(x.key);
    int cmphi = hi.compareTo(x.key);
    if (cmplo < 0) keys(x.left, queue, lo, hi);
    if (cmplo <= 0 && cmphi >= 0) queue.enqueue(x.key);
    if (cmphi > 0) keys(x.right, queue, lo, hi);
}
```

Análisis de las operaciones de orden

Proposición:

Todas las operaciones de orden (excepto la consulta por rango) requieren un número de comparaciones que es proporcional a la altura del árbol.

Justificación:

Todas las operaciones recorren uno o dos caminos partiendo de la raíz y llegando como máximo a una hoja. La longitud del mayor camino es la altura del árbol.