|  |
| --- |
| **TAD RB TREE** |
| Atributos o descripción. |
| {inv: } //Aquí van las propiedades |
| Operaciones básicas   * RBTree →RBTree * getMin → K * getMax → K * insert K, V → RBTree * insertFixup RBTree →RBTree * delete RBNode→ RBTree * deleteFixup RBNode→RBTree * search K → V * rightRotate RBNode→ RBTree * leftRotate RBNode→RBTree * getPredessor RBNode→RBNode * getSuccesor RBNode→RBNode * getNIL →RBNode * getRoot →RBNode * isInTree K → Boolean * searchBiggerThan K → ArrayList<V> * searchBiggerOrEqualThan K → ArrayList<V> * searchEqualTo K → ArrayList<V> * searchLowerTo K → ArrayList<V> * searchLowerOrEqualTo K → ArrayList<V> * setRoot RBNode→RBTree * transplant RBNode x RBNode→ RBTree |

Operaciones

|  |
| --- |
| **RBTree**  “ Este es el método constructor de la clase RBTree”  Pre:  Post: Se crea la clase RBTree |
| **getMin**  “Este método se encarga de buscar la llave que corresponde al valor más bajo dentro de todo el árbol Rojo y Negro”  Pre: El árbol Rojo y Negro no debe de ser nulo,  Post: |
| **getMax**  “Este método se encarga de encontrar la llave dentro del árbol con el valor más alto”  Pre: El árbol Rojo y Negro no debe ser Nulo  Post: |
| **Insert**  “Este método permite insertar un nodo dentro del árbol Rojo y Negro, donde este pertenece”  Pre: El nodo insertado de debe de ser rojo, El nodo no puede ser nulo,  Post: Se ha agregado un nuevo nodo y el árbol va a ser rebalanceado |
| **insertFixup**  “Este método se encarga de balancear el árbol Rojo y negro después de agregar un nodo”  Pre: UN árbol Rojo y negro desbalanceado  Post: El Arbol R y N esta blanceado |
| **Delete**  “Este método permite eliminar un nodo dentro del árbol Rojo y Negro con respecto a una llave que el usuario ingresa”  Pre: La llave debe pertenecer a los nodos del árbol, La llave no puede ser Nil  Post: Se elimina el nodo que se buscó |
| **deleteFixup**  “Este método balancea el árbol R y N después de eliminar algun nodo en el mismo”  Pre: Arbol R y N desbalanceado  Post: Arbol balanceado |
| **Modify**  “Este método permite modificar el nodo que se buscó con una llave que el usuario ingreso y esta modifica cualquier otro dato”  Pre: El árbol no es nulo, La llave no es nula, alguno de los valores nuevos no modifica a nulo alguno de los otros valores  Post: Se modifica el nodo |
| **Search**  “Este método tiene la función de buscar un nodo en base a una llave ingresada por el usuario”  Pre: La llave no es null, la llave existe dentro del árbol,  Post: |
| **rightRotate**  “ Este método tiene la funicion de rotar el nodo especificado hacia la derecha para reorganizar el árbol “  Pre: El nodo existe,  Post: los nodos son reorganizados en base al ingresado |
| **leftRotate**  “Este método se encarga de rotar el nodo hacia la izquierda en base al especificado por el usuario”  Pre: La llave existe, el nodo no es null  Post: El nodo es rotado hacia la izquierda |
| **getPredecessor**  “Este método busca el nodo que le precede al que el usuario busca”  Pre: Existe un numero menor a ese, Arbol no vaciio  Post: |
| **getSuccesor**  “Este método se encarga de buscar el nodo que le sigue al que se esta buscando”  Pre: El árbol no esta vacio, existe un valor mayor al buscado  Post: |
| **getNIL**  “Este método se encarga de retornar el nodo centinela del árbol Rojo y Negro”  Pre:  Post: |
| **getRoot**  “Este método se encarga de retornar la raíz del árbol Rojo y Negro”  Pre: El árbol Rojo y Negro no es Nulo  Post: |
| **isInTree**  “Este método verifica si la llave se encuentra o no dentro del árbol”  Pre:  Post: |
| **searchBiggerThan**  “Este método se encarga de buscar todos los nodos mayores a la llave y retorna un arreglo con los archivos”  Pre: Llave no es null  Post: |
| **searchBiggerOrEqualThan**  “Este método se encarga de buscar todos los nodos mayores o iguales a la llave y retorna un arreglocon los archivos”  Pre: Llave no es null  Post: |
| **searchEqualTo**  “Este método busca el nodo que es igual a la llave y retorna un arreglo con los archivos”  Pre: La llave no es null  Post: |
| **searchLowerOrEqualTo**  “Este método busca los nodos menores o iguales a la llave y retorna un arreglo con los archivos”  Pre: La llave no es null  Post: |
| **searchLowerTo**  “Este método busca los nodos menores a la llave y retorna un arreglo con los archivos”  Pre: La llave no es null  Post: |
| **Transplant**  “Este método se encarga de modificar el nodo hermano de el primer nodo “  Pre: el primer nodo debe existir  Post: Se ha modificado la ubicación del segundo nodo |

**Diseños de casos de pruebas unitarias**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prueba 1: Verifica que el método añade inserta correctamente un nodo al árbol manteniendo las propiedades. | | | | |
| Clase | Método | Escenario | Entrada | Resultado |
| RBTree | +insert (K, V): void | Hay un árbol rojinegro vacío | K=10  V=1 | La raíz es negra |
| RBTree | +insert (K, V): void | Hay un árbol rojinegro con un nodo: Key=10, value=1 | K=5  V=2 | La raíz es negra  El hijo izquierdo de la raíz es rojo. |
| RBTree | +insert (K, V): void | Hay un árbol rojinegro con los siguientes nodos:  Key=10 value=1  Key=5 value=2 | K=15  V=3 | La raíz es negra y sus dos hijos son rojos. |
| RBTree | +insert (K, V): void | El mismo que el anterior más un nodo con key=15 y value=3 | K=4  V=4 | La raíz es negra y sus dos hijos son negros. El hijo izquierdo el hijo izquierdo de la raíz es rojo. |
| RBTree | +insert (K, V): void | El mismo que el anterior más un nodo con key=4 y value=4 | K=4  V=5 | La raíz es negra. Sus dos hijos son negros y tiene key 4 y 15 respectivamente. Los dos hijos del hijo izquierdo de la raíz son rojos y tienen key 4 y 5 respectivamente. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prueba 2: Verifica que el método delete elimina correctamente un nodo al árbol manteniendo las propiedades. | | | | |
| Clase | Método | Escenario | Entrada | Resultado |
| RBTree | +delete(K): V | Hay un árbol rojinegro vacío | K=10 | Retorna null.  El árbol sigue estando vacío |
| RBTree | +delete(K): V | Hay un árbol rojinegro con un nodo: Key=10, value=1 | K=10 | Retorna 1  El árbol está vacío. |
| RBTree | +delete(K): V | Hay un el siguiente árbol rojinegro:    Con los siguientes Values:  V(11)=1  V(2)=2  V(1)=3  V(7)=4  V(5)=5  V(8)=6  V(14)=7  V(15)=8 | K=5 | Retorna 5  No ocurre ninguna rotación, los colores permanecen estables. El hijo izquierdo de 7 es NIL. |
| RBTree | +delete(K): V | El mismo que el anterior | K=7 | Retorna 4  8 pasa a ser el hijo derecho de 2 y es negro, además su hijo derecho es 5 y sigue siendo rojo. |
| RBTree | +delete(K): V | El mismo del anterior | K=20 | Retorna null. |
| RBTree | +delete(K): V | Se tiene el siguiente árbol rojinegro    Con los siguiente values:  V(8)=1  V(7)=2 y su key ahora va a ser 6  V(12)=3  V(6)=4 y su key ahora va a ser 5  V(7.5)=5 y su key ahora va a ser 7  V(10)=6  V(14)=7  V(15)=8  V(13)=9 | K=10 | Retorna 6.  La raíz es negra y es 8  Su hijo derecho es 14 y es negro.  El hijo izquierdo de 14 es 12 y es negro y su hijo derecho es 13 y es rojo.  El hijo derecho de 14 es 15 y es negro. |
| RBTree | +delete(K): V | Existe el siguiente árbol rojinegro:    Con los siguiente Values:  V(32)=1  V(21)=2  V(64)=3  V(15)=4  V(75)=5 | K=21 | Retorna 2  La raíz es 32 y sus dos hijos son negros y son 15 y 64 respectivamente.  El hijo derecho de 64 es 75 y es rojo. |
| RBTree | +delete(K): V | El mismo que el anterior | K=64 | Retorna 3  La raíz es 32 y sus dos hijos son negros y son 21 y 75 respectivamente.  El hijo izquierdo de 21 es 15 y es rojo. |