```java
package cayci_hw5;

/**
 * A class to implement BinarySearchTree with an array.
 * @author cagri cayci
 * @param <E> To make the class generics.
 */
public class BinarySearchTree<E extends Comparable<E>> implements SearchTree<E>{
    /**
     * An object array to keep datas.
     */
    private Object[] array;
    /**
     * An integer value to keep size of the tree.
     */
    private int size = 0;

    /**
     * An integer value to keep capacity of the array.
     */
    private int capacity = 1;

    /**
     * Creates a BinarySearchTree.
     */
    public BinarySearchTree(){
        array = new Object[capacity];
    }

    /**
     * Returns the size of the BinarySearchTree.
     * @return Returns the number of elements BinarySearchTree contains.
     */
    public int size(){
        return size;
    }

    /**
     * Inserts item where it belongs in the tree. Returns true if item is inserted, false if it is not inserted.
     * @param item Gets the item as type E.
     * @return Returns true if item is inserted, otherwise false.
     */
    @Override
    public boolean add(E item){ Amortized O(height)
        int index = whereToPlace(item); /* Search a correct index to place the item. */ O(height)
        if(index >= capacity) /* If index violates capacity, reallocate. */
```

```
            reallocate();Theta(n)
        if(array[index] != null) /* Checks the item is already there or not. */ Theta(1)
            return false; /* If the item is already inserted to tree, returns false. */
        array[index] = item; /* Place the item. */ Theta(1)
        size++; /* Increase size by one. */ Theta(1)
        return true; /* Returns true. */
    }

    /**
     * Search the target in the tree, returns true if the target is found, false if it is not found.
     * @param target Gets the target value as type E.
     * @return Returns true if the target is found, otherwise false.
     */
    @Override
    public boolean contains(E target) { O(height)
        return (find(target) != null); /* Calls find method, if it does not return null, the target is found. */ O(height)
    }

    /**
     * Removes target value if it is found from tree and returns it, otherwise returns null.
     * @param target Gets the target value as type E.
     * @return Returns the target value if it is found, otherwise null.
     */
    @Override
    public boolean remove(E target) { Amortized O(height)
        int index = indexOf(target); /* Finds the index of the target value in array. */ O(height)
        if(index == -1) /* If the index is -1 (There is no such an element), returns false. */ Theta(1)
            return false;
        remove(index); /* Calls overloaded remove function with index parameter. */ O(height)
        size--; /* Decreases size. */ Theta(1)
        if(capacity >= Math.pow(2, size) + 1) /* If the capacity is more than the number of nodes of an binary tree which its height equals to
number of nodes, decreases the capacity. */
            free(); /* Frees the empty space. */ Theta(n)
        return true;
    }

    /**
     * Returns a reference to the data on the tree node that is equal to target. If there is no such a node, returns null.
     * @param target Gets the target value as type E.
     * @return Returns a reference of a node which is equal to target, otherwise null.
     */
    @Override
    public E find(E target) { O(height)
        int index = indexOf(target); O(height)
        return (index != -1) ? (E) array[index] : null;
    }
```

```
    /**
     * Removes target value if it is found from tree and returns true, otherwise returns false.
     * @param target Gets the target value as type E.
     * @return Returns true if target value is found, otherwise false.
     */
    @Override
    public E delete(E target) {Amortized O(height)
        int index = indexOf(target); /* Finds the index of the target value in array. */ O(height)
        if(index == -1) /* If the index is -1 (There is no such an element), returns null. */Theta(1)
            return null;
        remove(index); /* Calls overloaded remove function with index parameter. */ O(height)
        size--; /* Decreases size. */ Theta(1)
        if(capacity >= Math.pow(2, size) + 1) /* If the capacity is more than the number of nodes of an binary tree which its height equals to
number of nodes, decreases the capacity. */
            free(); /* Frees the empty space. */ Theta(n)
        return target;
    }

    public String toString(){
        StringBuilder string = new StringBuilder();
        string.append("{");
        for(int i = 0; i < capacity; i++){
            string.append(array[i]);
            if(i != capacity - 1)
                string.append(", ");
        }
        string.append("}");
        return string.toString();
    }

    /**
     * Removes the element at the indexth element of array.
     * @param index Gets index as integer.
     */
    private void remove(int index){ O(height)
        int rightChild = 2 * index + 2;  /* Sets right child index. */ Theta(1)
        int leftChild = 2 * index + 1; /* Sets left child index. */ Theta(1)
        if(rightChild < capacity){ /* If the node has one or two children, continue. */ Theta(1)
            if(array[rightChild] != null && array[leftChild] == null){ /* If there is only right child, continue. */ Theta(1)
                array[index]  = array[rightChild]; /* Assign right child to node. */ Theta(1)
                slide(index, rightChild); /* Moves children of right child. */ O(height)
            }
            else if(array[rightChild] == null && array[leftChild] != null){ /* If there is only left child, continue. */ Theta(1)
                array[index]  = array[leftChild]; /* Assign left child to node. */ Theta(1)
                slide(index, leftChild); /* Moves children of left child. */ O(height)
            }
            else{ /* If there are both right and left child, continue. */
```

```
            int successor = successorOf(rightChild); /* Gets index of successor. */ O(height)
            array[index] = array[successor]; /* Assign node at successor to node at index. */
            remove(successor); /* Slide children of successor up. */ O(height)
        }
    }
    else /* If there is no child of the node, assign null to node. */ Theta(1)
        array[index] = null;
}

/**
 * Makes children of child to children of parent.
 * @param parent Takes parent node index as integer.
 * @param child Takes child node index as integer.
 */
private void slide(int parent, int child){ O(height)
    int leftGrandChild = 2 * child + 1; /* Sets left grandchild index.*/ Theta(1)
    int leftChild = 2 * parent + 1; /* Sets left child index. */ Theta(1)
    if(leftGrandChild >= capacity) /* If indexes of grandchildren violates capacity, terminate the method. */ Theta(1)
        return;
    array[leftChild + 1] = array[leftGrandChild + 1]; /* Assign right child of parent to right child of child. */ Theta(1)
    array[leftChild] = array[leftGrandChild]; /* Assign left child of parent to left child of child. */ Theta(1)
    array[leftGrandChild + 1]  = null; /* Makes right grandchild null. */ Theta(1)
    array[leftGrandChild] = null; /* Makes left grandchild null. */ Theta(1)
    slide(leftChild + 1, leftGrandChild + 1); /* Recursively call the function for right child of child. */
    slide(leftChild, leftGrandChild); /* Recursively call the function for left child of child. */
}

/**
 * Finds successor of an element.
 * @param index Gets index as integer.
 * @return Returns successor index as integer.
 */
private int successorOf(int index){ O(height)
    int successor = index; Theta(1)
    while(2 * successor + 1 < capacity && array[2 * successor + 1] != null) /* Moves until most left node. */ O(height)
        successor = 2 * successor + 1;
    return successor; /* Return index of the most left node of the tree. */ Theta(1)
}

/**
 * Gets a target and search for a node to place the target.
 * @param target Gets the target value as type E.
 * @return Returns index of the node.
 */
private int whereToPlace(E target){
    return whereToPlace(target, 0);
}
```

```
/**
 * Gets a target and search the target in tree.
 * @param target Gets a target value as type E.
 * @return Returns the index of the node which contains target value, otherwise -1.
 */
private int indexOf(E target){
    return indexOf(target, 0);
}

/**
 * A recursive function to find a suitable node to place the target value.
 * @param target Gets a target value as type E.
 * @param index Gets an index as type integer.
 * @return Returns the index of the node.
 */
private int whereToPlace(E target, int index){ O(height)
    if(index >= capacity) Theta(1)
        return index;
    if(array[index] == null) Theta(1)
        return index;
    int comparision = target.compareTo((E) array[index]); /* Compare item and indexth element of array. */ Theta(1)
    if(comparision == 0) /* If item is already in the tree, returns -1. */ Theta(1)
        return index;
    else if(comparision > 0) /* If item is bigger than indexth element of array, continue with right binary tree. */
        return whereToPlace(target, 2 * index + 2);
    else /* If item is less than indexth element of array, continue with left binary tree. */
        return whereToPlace(target, 2 * index + 1);
}
```

The time complexity of whereToPlace method is Theta(n) at worst case(height of the tree equals to number of nodes), Theta(1) at best case. But in avarage case, the time complexity of whereToPlace method is O(logn)

```
/**
 * Gets a target and search the target in tree.
 * @param target Gets a target value as type E.
 * @param index Gets a index as type integer.
 * @return Returns the index of the node which contains target value, otherwise -1.
 */
private int indexOf(E target, int index){ O(height)
    if(index >= capacity) Theta(1)
        return -1;
    if(array[index] == null) Theta(1)
        return -1;
    int comparision = target.compareTo((E) array[index]); /* Compare item and indexth element of array. */ Theta(1)
    if(comparision == 0) /* If item is already in the tree, returns index. */ Theta(1)
        return index;
    else if(comparision > 0) /* If item is bigger than indexth element of array, continue with right binary tree. */
        return indexOf(target, 2 * index + 2);
    else /* If item is less than indexth element of array, continue with left binary tree. */
```

The time complexity of indexOf method is Theta(n) at worst case(height of the tree equals to number of nodes), Theta(1) at best case. But in avarage case, the time complexity of indexOf method is O(logn)

```
            return indexOf(target, 2 * index + 1);
    }

    /**
     * Decreases the size of the array when it is needed.
     */
    private void free(){ Theta(n)
        Object[] temp = new Object[(capacity - 1) / 2]; /* Create an array which can keep a binary tree with height is 1 less than previous
height. */ Theta(1)
        for(int i = 0; i < (capacity - 1) / 2; i++) /* Copies the previous array to new array. */ Theta(n)
            temp[i] = array[i];
        array = temp; /* Assign the new array to previous array. */ Theta(1)
        capacity = (capacity - 1) / 2; /* Decrease capacity as if it were a perfect binary tree. */ Theta(1)
    }

    /**
     * Increases the size of the array when it is needed.
     */
    private void reallocate(){ Theta(n)
        Object[] temp = new Object[capacity * 2 + 1];Theta(1)
        for(int i = 0; i < capacity; i++) /* Copy the previous array to new array. */ Theta(n)
            temp[i] = array[i];
        array = temp; /* Assign the new array to previous array. */ Theta(1)
        capacity = 2 * capacity + 1; /* Increase capacity as if it were a perfect binary tree. */ Theta(1)
    }
}
```