```
package cayci_hw5;

/**
 * A class which implements Binary Max Heap.
 * @author cagri cayci
 * @param <E> To make class generics.
 */
public class BinaryHeap<E extends Comparable<E>>{

    /**
     * Keeps the root nodes of the BinaryHeap structure.
     */
    private Node<E> root;

    /**
     * Keeps the parent of an element which is added last.
     */
    private Node<E> lastElementAdded;

    /**
     * Keeps the size of the BinaryHeap.
     */
    private int size = 0;

    /**
     * No parameter constructor for BinaryHeap.
     */
    public BinaryHeap(){ Theta(1)
        root = null;
    }

    /**
     * One parameter constructor for BinaryHeap.
     * @param _root Takes _root node as paramter.
     */
    protected BinaryHeap(Node<E> _root){ Theta(1)
        root = _root;
    }

    /**
     * Removes the element which has biggest priorities on the BinaryHeap.
     * @return Returns false if there is no element, otherwise true.
     */
    public boolean removeBiggest(){ O(height)
        if(root == null) /* If BinaryHeap is empty, there is no element to remove. */
            return false;
```

```
        if(size == 1) /* If there is only one element, deletes it. */
            root = null;
        else{ /* Copies the priority and data of the last element to root node to _data and _keyValue parameters and assign it to root node. And
make last element null. */
            E _data;
            int _keyValue;
            if(lastElementAdded.rightChild != null){ /* If the last element is right child of lastElementAdded node, continue. */ Theta(1)
                _data = lastElementAdded.rightChild.data;
                _keyValue = lastElementAdded.rightChild.keyValue;
                lastElementAdded.rightChild = null;
            }
            else{ /* If the last element is left child of lastElementAdded node, continue. */Theta(1)
                _data = lastElementAdded.leftChild.data;
                _keyValue = lastElementAdded.leftChild.keyValue;
                lastElementAdded.leftChild = null;
            }
            root.data = _data;
            root.keyValue = _keyValue;
            upToDown(root); /* Calls upToDown method to keep heap order property. */ O(height)
        }
        size = size - 1; /* Decreases size by one. */
        return true;
    }

    /**
     * Adds an element to BinaryHeap.
     * @param _data Gets _data as type of E.
     * @param _keyValue Gets _keyValue(priority) as integer.
     * @return Returns true always.
     */
    public boolean add(E _data, int _keyValue){ O(height)
        if(root == null){ /* If the BinaryHeap is empty, add the new element to root and update lastElementAdded field. */Theta(1)
            root = new Node<E>(_keyValue, _data);
            lastElementAdded = root;
        }
        else /* If the BinaryHeap is not empty, calls overloaded add method. */
            add(_keyValue, _data, 0);O(height)
        if(lastElementAdded.rightChild != null) /* Calls downToUp to keep heap order property for last element. */
            downToUp(lastElementAdded.rightChild, lastElementAdded);O(height)
        else /* Calls downToUp to keep heap order property for last element. */
            downToUp(lastElementAdded.leftChild, lastElementAdded); O(height)
        size = size + 1; /* Increases size by one. */
        return true;
    }

    /**
     * Merges a BinaryHeap with current BinaryHeap.
```

```
 * @param h1 Takes a BinaryHeap as parameter.
 * @return Returns A BinaryHeap which is union of two BinaryHeap.
 */
@SuppressWarnings("unchecked")
public BinaryHeap<E> mergeHeapWith(BinaryHeap<E> h1){ O(n * h)
    BinaryHeap temp = h1; /* Creates a temp BinayHeap and assign h1 to it. */
    return mergeHeapWith(temp, root); /* Calls overloaded mergeHeapWith method. */ O(n * h)
}

/**
 * Changes priority of an element.
 * @param _data Gets _data as type of E to find element in the BinaryHeap.
 * @param _keyValue Gets _keyValue which will changed with old one as type of integer.
 * @return Returns false if BinaryHeap is empty, otherwise true.
 */
public boolean setPriority(E _data, int _keyValue){ O(n)
    Node<E> temp = null;
    temp = searchFor(root, _data, temp); /* Searches the node with given data. */ O(n)
    if(temp == null) /* If the node is not found, return false. */
        return false;
    temp.keyValue = _keyValue; /* Changes keyValue of the node. */
    upToDown(temp); /* Calls upToDown to keep heap order property for the node */ Theta(height)
    if(temp.parent != null)
        downToUp(temp, temp.parent); /* Calls downToUp to keep heap order property for the node. */ Theta(height)
    return true;
}

/**
 * Gets the data of the root node.
 * @return Returns data of the root node if there is a root, otherwise returns null.
 */
public E getData(){ Theta(1)
    return (root == null) ? null : root.data;
}

/**
 * Checks root node whether it is a leaf or not.
 * @return Return true if root node is a leaf, otherwise false.
 * @throws TreeHasNotCreatedYetException Throws an exception if root node does not exist.
 */
public boolean isLeaf() throws TreeHasNotCreatedYetException{ Theta(1)
    if(root == null) /* If the BinaryTree is empty, throws an exception. */
        throw new TreeHasNotCreatedYetException();
    else
        return (root.rightChild == null && root.leftChild == null); /* If both children of the root is null return true. */
}
```

```
/**
 * Converts the BinaryHeap to String.
 * @return Returns String representation of BinaryHeap.
 */
public String toString(){ Theta(1)
    StringBuilder string = new StringBuilder(); /* Creates a StringBuilder. */
    string = printer(root, string, 0); Theta(n)
    return (string == null) ? null : string.toString();
}

/**
 * Recursively adds every element of current BinaryHeap to new BinaryHeap.
 * @param h1 Takes h1 as type of BinaryHeap.
 * @param _root Takes the root of the current BinaryHeap.
 * @return Returns BinaryHeap which is union of current and h1 BinaryHeaps.
 */
private BinaryHeap<E> mergeHeapWith(BinaryHeap<E> h1, Node<E> _root){ O(n * h)
    if(root == null) /* If the current BinaryHeap is empty, union of current BinaryHeap and h1 is equal to h1. */
        return h1;
    /* Adds every element of the current BinaryHeap to h1 with inorder traversal. */
    if(_root.leftChild != null)
        mergeHeapWith(h1, _root.leftChild);
    h1.add(_root.data, _root.keyValue);
    if(_root.rightChild != null)
        mergeHeapWith(h1, _root.rightChild);
    return h1;
}

/**
 * Searches an node in the BinaryHeap by its data recursively.
 * @param root Takes the root as parent node.
 * @param _data Takes the _data to search it.
 * @return Returns the node if it is found, otherwise null.
 */
private Node<E> searchFor(Node<E> root, E _data, Node<E> temp){ O(n)
    if(root == null) /* If the BinaryHeap is empty, returns null. */
        return null;
    int comparision = root.data.compareTo(_data); /* Compares the data of the current node with given data. */ Theta(1)
    if(comparision == 0) /* If they are equal return current node. */
        return root;
    else{ /* If they are not equal, search the item from most left to right. */
        temp = searchFor(root.rightChild, _data, temp);
        if(temp == null)
            temp = searchFor(root.leftChild, _data, temp);
    }
    return temp;
}
```

Reaching an element in current BinaryHeap takes O(number of nodes of current BinaryHeap), adding an element to h1 takes O(height of h1)

It searches all element until it found the correct node.

```
    /**
     * Recursively compares a node with children of it, replace one of the children with parent if the priority of the child is bigger than
priority of parent.
     * @param n1 Takes the parent as type of Node.
     */
    private void upToDown(Node<E> n1){ O(height)
        if(n1 == null) /* If the node is null, terminates the method. */
            return;
        if(n1.rightChild != null && n1.leftChild != null){ /* If the node has both right and left child, continue. */
            if(n1.rightChild.keyValue > n1.leftChild.keyValue && n1.rightChild.keyValue > n1.keyValue){ /* If priority of rightChild is bigger
than both priority */
                swap(n1.rightChild, n1);Theta(1)                                                    /*  of left child and priority of
node, swap rightChild */
                upToDown(n1.rightChild);                                                             /*  with node and calls the method
for rightChild. */
            }
            else if(n1.leftChild.keyValue > n1.rightChild.keyValue && n1.leftChild.keyValue > n1.keyValue){ /* If priority of leftChild is
bigger than both priority */
                swap(n1.leftChild, n1);Theta(1)                                                       /*  of right child and priority of
node, swap leftChild */
                upToDown(n1.leftChild);                                                               /*  with node and calls the
method for leftChild. */
            }
        }
        /* If the node has one child, continue. */
        else if(n1.leftChild != null && n1.leftChild.keyValue > n1.keyValue){ /* If priority of leftChild is bigger than both priority */
            swap(n1.leftChild, n1);                                                 /*  of right child and priority of node, swap leftChild */
            upToDown(n1.leftChild);                                                 /*  with node and calls the method for leftChild. */
        }
        else if(n1.rightChild != null && n1.rightChild.keyValue > n1.keyValue){ /* If priority of rightChild is bigger than both priority */
            swap(n1.rightChild, n1);                                                /*  of left child and priority of node, swap rightChild */
            upToDown(n1.rightChild);                                                /*  with node and calls the method for rightChild. */
        }
    }

    /**
     * Swaps to datas and priorities of two nodes.
     * @param n1 Takes first node as type of Node.
     * @param n2 Takes second node as type of Node.
     */
    private void swap(Node<E> n1, Node<E> n2){ Theta(1)
        if(n2 == null || n1 == null) /* If one of the node is null, terminates the method. */
            return;
        int keyValue = n1.keyValue; /* Takes key value of first node. */
        E data = n1.data; /* Takes data of first node. */
        n1.keyValue = n2.keyValue; /* Assign key value of second node to first node. */
```

Best case Theta(1), worst case Theta(height)

Just changes to nodes datas.

```
    n1.data = n2.data; /* Assign data of second node to first node. */
    n2.keyValue = keyValue; /* Assign previous key value of first node to second node. */
    n2.data = data; /* Assign previous data of first node to second node. */
}

/**
 * Recursively compares a node with parent of it, if priority of child is bigger than priority of parent, replaces them each other.
 * @param n1 Takes first node as type of Node.
 * @param n2 Takes second node as type of Node.
 */
private void downToUp(Node<E> n1, Node<E> n2){ O(height)
    if(n2 == null || n1 == null) /* If one of the node is null, terminates the method. */
        return;
    if(n1.keyValue > n2.keyValue){ /* If key value of child is bigger than key value of parent, swap them. */
        swap(n1, n2);Theta(1)
        downToUp(n2, n2.parent); /* Calls the method for new version of parent with its parent. */
    }
}

/**
 * Helps printing BinaryHeap in pretty way.
 * @param _root Takes _root as type of Node.
 * @param string Takes string as type of StringBuilder.
 * @param emptySpace Takes number of empty space as type of integer.
 * @return
 */
private StringBuilder printer(Node<E> _root, StringBuilder string, int emptySpace){ Theta(n)
    for(int i = 0; i < emptySpace; i++){ /* For indentation. */
        string.append(" ");
    }
    string.append(_root); /* Adds String version of _root to string. */
    string.append("\n"); /* Adds new line sign. */
    if(_root == null) /* If root is null, terminates the method, without trying to reach its children. */
        return null;
    printer(_root.leftChild, string, emptySpace + 1); /* Calls the method for leftChild. */
    printer(_root.rightChild, string, emptySpace + 1); /* Calls the method for rightChild. */
    return string;
}

/**
 * Helps adding elements to BinaryHeap without changing type of the BinaryHeap(complete tree).
 * @param _data Gets _data as type of E.
 * @param _keyValue Gets _keyValue(priority) as integer.
 * @param mode A mode for the method to go most left child.
 */
private void add(int _keyValue, E _data, int mode){ O(height)
    if(lastElementAdded.leftChild == null){ /* If last element is left child of lastElementAdded field, continues. */ Theta(1)
```

Best case Theta(1), worst case Theta(height)

Prints all element in the BinaryHeap.

```
                lastElementAdded.leftChild = new Node<E>(_keyValue, _data); /* Adds the new node to left child of lastElementAdded field. */
                lastElementAdded.leftChild.parent = lastElementAdded; /* Set new node parent. */
            }
            else if(lastElementAdded.rightChild == null){ /* If last element is right child of lastElementAdded field, continues. */ Theta(1)
                lastElementAdded.rightChild = new Node<E>(_keyValue, _data); /* Adds the new node to right child of lastElementAdded field. */
                lastElementAdded.rightChild.parent = lastElementAdded; /* Set new node parent. */
            }
Theta(height)   else if(lastElementAdded.parent == null || mode == 1){ /* If the parent of lastElementAdded field is null or mode is 1, continue. */
                lastElementAdded = lastElementAdded.leftChild; /* To reach height + 1. */
                add(_keyValue, _data, 1);
            }
  Theta(1)  else if(lastElementAdded != lastElementAdded.parent.rightChild){ /* If right parent of grandparent of lastElementAdded field has not
   filled yet, continue. */
                lastElementAdded = lastElementAdded.parent.rightChild;
                add(_keyValue, _data, 0);
            }
            else{ Theta(height)
                lastElementAdded = lastElementAdded.parent; /* Going back from child to parent. */
                add(_keyValue, _data, 0);
            }
        }

        /**
         * A node class to keep priorities and datas.
         * @param <E> To make class generics.
         */
        protected class Node<E>{
            /**
             * To keep node of the right child of the node.
             */
            private Node<E> rightChild = null;

            /**
             * To keep node of the left child of the node.
             */
            private Node<E> leftChild = null;

            /**
             * To keep parent node of the node.
             */
            private Node<E> parent = null;

            /**
             * Keeps priority of the node.
             */
            private int keyValue;
```

```
        /**
         * Keeps data as type of E.
         */
        private E data;

        /**
         * Sets priority and data of a Node.
         * @param _keyValue Takes _keyValue of the node as type of integer.
         * @param _data  Takes _data of the node as type of E.
         */
        public Node(int _keyValue, E _data){ Theta(1)
            keyValue = _keyValue;
            data = _data;
        }

        /**
         * Converts Node to String.
         * @return Returns String representation of the Node class.
         */
        public String toString(){ Theta(1)
            return (data == null) ? "null" :"(" + keyValue + ", " + data + ")";
        }
    }
}
```