**GIT Department of Computer Engineering**

**CSE 222/505 - Spring 2021**

**Homework 4 Report**

**Sena Erdoğan**

**1901042680**

1. **SYSTEM REQUIREMENTS**

**Functional Requirements**

* The system shall let the user create Heaps.
* The user shall be able to add elements to the heap.
* The user shall be able to remove elements from the heap.
* The user shall be able to search for an element in the heap.
* The user shall be able to get an iterator for the heap.
* The user shall be able to set an element to the last value returned by the iterator.
* The system shall let the user create BSTHeapTrees.
* The user shall be able to add elements to the BSTHeapTree.
* The user shall be able to remove elements from the BSTHeapTree.
* The user shall be able to get the number of occurences of an element in the BSTHeapTree.
* The user shall be able to find the mode of the BSTHeapTree.

**Non-functional Requirements**

**Utility**

• The system shall be user friendly.

**Reliability**

• The system should give error messages and provides error handling.

**Performance**

• The system uses the driver code’s data and if that data gets larger, the system should work faster.

**Space requirements**

• The system should be scalable only to store runtime data.

**Supportability**

• The system supports the OS’s below amd is portable.

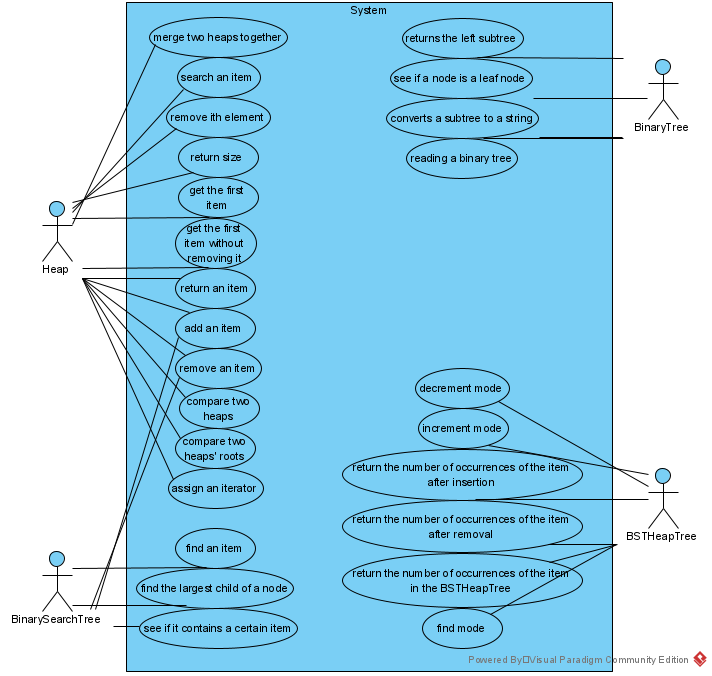
**User Interface**

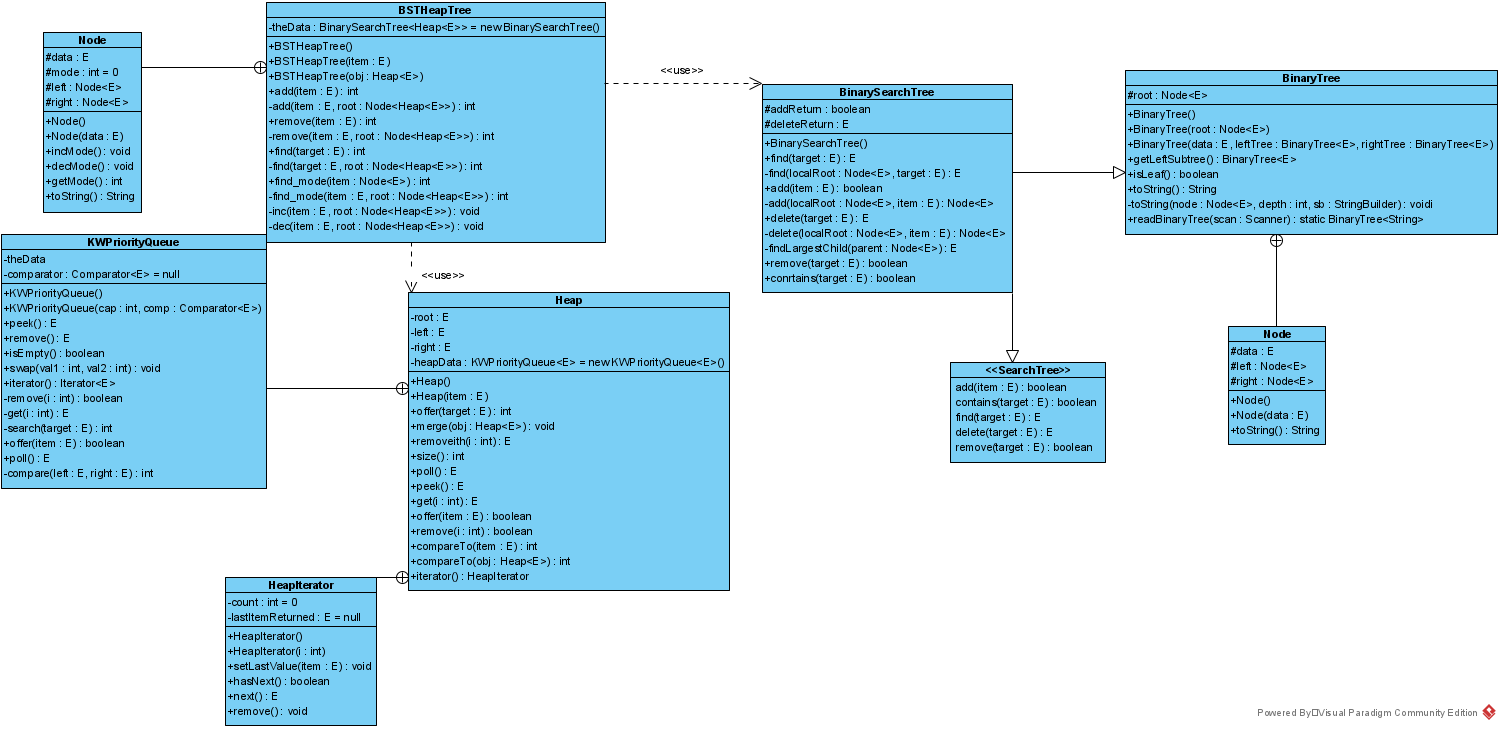
• All operations will be done via a standard PC.

**Domain Requirements**

• The system shall work on Windows, Linux and Mac Os.

1. **USE CASE AND CLASS DIAGRAMS**





1. **PROBLEM SOLUTION APPROACH**

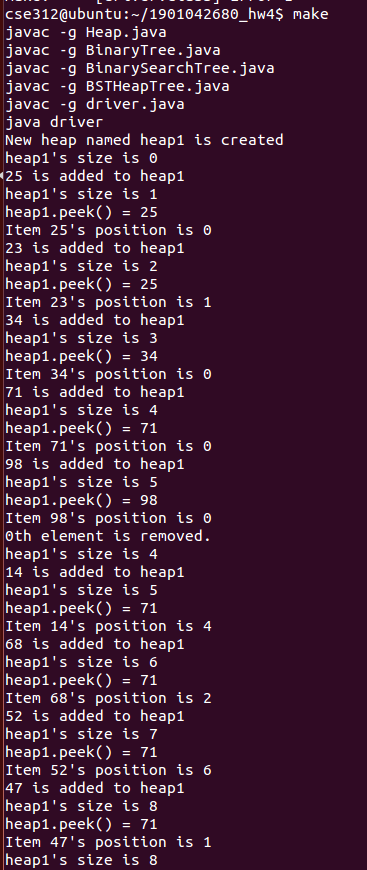
Heap data structure is formed using priority queue as a data field. Priority queue initially has the minimum items at the front. Method compareTo is modified to return a negative number if left is greater than right and a positive number if left is less than right. BSTHeapTree’s data is stored in a BinarySearchTree that has Heap data structures as elements. Node data structure stores both the data of that node has and the mode of the node.

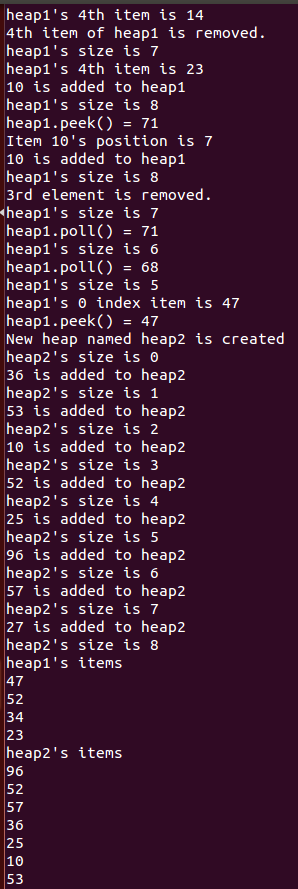
1. **TEST CASES**

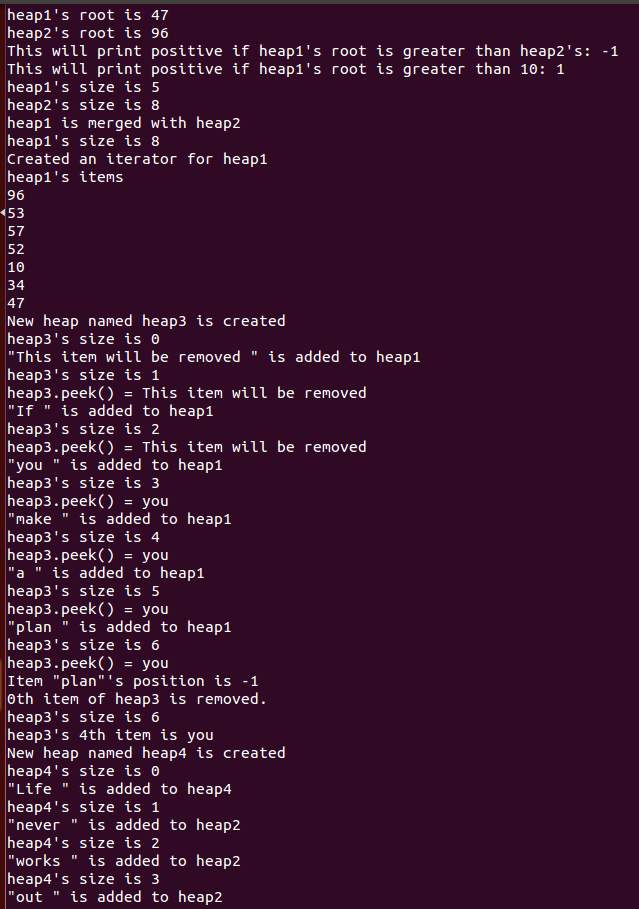
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **TEST CASE ID** | **TEST SCENARIO** | **TEST STEPS** | **TEST DATA** | **EXPECTED RESULTS** | **ACTUAL RESULTS** | **PASS/FAIL** |
| TU01 | When a new heap is created, its size should be 0 | Get heap1’s size | No data needed. | Size should be 0 | As expected | Pass |
| TU02 | Add 25 to heap1 | Offer heap1 25 | 25 | 25 should be added to heap1 | As expected | Pass |
| TU03 | After addition, is size incremented? | Get heap1’s size | No data needed. | Size should be 1 | As expected | Pass |
| TU04 | Add some more elements to heap1 | Offer integer values to heap1 | 98  14  68 | Peek should always return the maximum element | As expected | Pass |
| TU05 | Add some more elements to heap1 | Offer integer values to heap1 | 98  14  68 | Index of elements should change if there is an item that is less than any one of the items | As expected | Pass |
| TU06 | Removing an element | Remove an element | Remove 0th element | heap1’s size should decrement | As expected | Pass |
| TU07 | Removing an element | Remove an element | Remove 0th element | Peek should return the new maximum element | As expected | Pass |

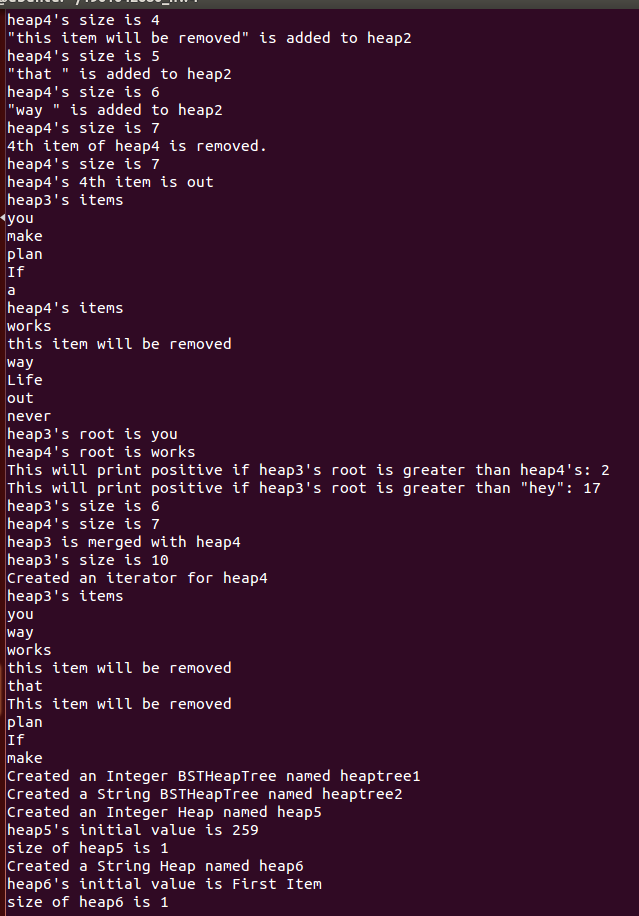
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| TU08 | Try adding an item that is already in the heap | Offer 10 to heap1 | 10 | Size should remain the same | As expected | Pass |
| TU09 | Get the first item | Poll from heap1 | No data needed. | Size should decrement | As expected | Pass |
| TU10 | Get the first item | Poll from heap1 | No data needed. | The maximum element should be returned | As expected | Pass |
| TU11 | Search an item in the heap | Search an item | Search 52 | 52’s position should be at index 6 | As expected | Pass |
| TU12 | Create another integer heap named heap2 and merge heap1 together with heap2 | Merge heap1 and heap2 | heap1 and heap2 | heap1 should have all items of heap2 with no duplicates | As expected | Pass |
| TU13 | Iterator for heap1 goes through heap1 and prints the items | For-each loop | No data needed | Every item of heap1 should be preinted | As expected | Pass |
| TU14 | Creating a String heap named heap3 | Heap3 is created | No data needed | Size should be 0 | As expected | Pass |
| TU15 | Search an item that is not in the heap | Search “plan” | “plan” | Position should be -1 | As expected | Pass |
| TU16 | Create an Integer BSTHeapTree | heaptree1 is created | No data needed | No errors | As expected | Pass |
| TU17 | Create a String BSTHeapTree | heaptree2 is created | No data needed | No errors | As expected | Pass |
| TU18 | Create an Integer heap named heap5 with an initial value | Use the parameterized constructor | 259 | Size of heap5 should be 1 | As expected | Pass |
| TU19 | Create a String heap named heap6 with an initial value | Use the parameterized constructor | “First Item” | Size of heap6 should be 1 | As expected | Pass |

1. **RUNNING AND RESULTS**









**TIME COMPLEXITIES**

public int size(){**ϴ(1)**

return theData.size(); **ϴ(1)**

}

public KWPriorityQueue() { **ϴ(1)**

theData = new ArrayList<>(); **ϴ(1)**

}

public KWPriorityQueue(int cap, Comparator<E> comp) { **ϴ(1)**

if (cap < 1) **ϴ(1)**

throw new IllegalArgumentException();

theData = new ArrayList<>(); **ϴ(1)**

comparator = comp;

}

public E peek(){**ϴ(1)**

return theData.get(0); **ϴ(1)**

}

public E remove(){**O(n)**

if(theData.size() != 0){ **ϴ(1)**

E item = theData.get(0); **ϴ(1)**

poll();**O(n)**

return item; **ϴ(1)**

}else throw new IndexOutOfBoundsException();

}

public boolean isEmpty(){**ϴ(1)**

return theData.size() == 0; **ϴ(1)**

}

public void swap(int val1, int val2){ **ϴ(1)**

E temp = theData.get(val1); **ϴ(1)**

theData.set(val1, theData.get(val2)); **ϴ(1)**

theData.set(val2, temp); **ϴ(1)**

}

public Iterator<E> iterator(){**ϴ(1)**

return theData.iterator();**ϴ(1)**

}

private boolean remove(int i){ **O(n)**

if(theData.size() != 0 || i<size()-1){ **ϴ(1)**

theData.remove(i); **O(n)**

return true; **ϴ(1)**

}else throw new IndexOutOfBoundsException();**ϴ(1)**

}

private E get(int i){ **ϴ(1)**

if(theData.size() == 0 || i>theData.size()-1) return null; **ϴ(1)**

return theData.get(i); **ϴ(1)**

}

private int search(E target){ **O(n)**

return theData.indexOf(target); **O(n)**

}

public boolean offer(E item) { **amortized ϴ(1)**

theData.add(item); **amortized ϴ(1)**

int child = theData.size() - 1; **ϴ(1)**

int parent = (child - 1) / 2; **ϴ(1)**

while (parent >= 0 && compare(theData.get(parent),

theData.get(child)) > 0) { **ϴ(1)**

swap(parent, child); **ϴ(1)**

child = parent; **ϴ(1)**

parent = (child - 1) / 2; **ϴ(1)**

}

return true; **ϴ(1)**

}

public E poll() { **O(n)**

if (isEmpty()) { **ϴ(1)**

return null; **ϴ(1)**

}

E result = theData.get(0); **ϴ(1)**

if (theData.size() == 1) { **ϴ(1)**

theData.remove(0); **O(n)**

return result; **ϴ(1)**

}

theData.set(0, theData.remove(theData.size() - 1)); **O(n)**

int parent = 0; **ϴ(1)**

while (true) { **ϴ(1)**

int leftChild = 2 \* parent + 1; **ϴ(1)**

if (leftChild >= theData.size()) { **ϴ(1)**

break; **ϴ(1)**

}

int rightChild = leftChild + 1; **ϴ(1)**

int minChild = leftChild; **ϴ(1)**

if (rightChild < theData.size()

&& compare(theData.get(leftChild),

theData.get(rightChild)) > 0) { **ϴ(1)**

minChild = rightChild; **ϴ(1)**

}

if (compare(theData.get(parent),

theData.get(minChild)) > 0) { **ϴ(1)**

swap(parent, minChild); **ϴ(1)**

parent = minChild; **ϴ(1)**

} else { break; **ϴ(1)**

}

}

return result; **ϴ(1)**

}

private int compare(E left, E right) { **ϴ(1)**

if (comparator != null) { **ϴ(1)**

return -comparator.compare(left, right); **ϴ(1)**

} else {

return -((Comparable<E>) left).compareTo(right); **ϴ(1)**

}

}

public Heap(E item){ **amortized ϴ(1)**

heapData.offer(item); **amortized ϴ(1)**

}

public int search(E target){ **O(n)**

return heapData.search(target); **O(n)**

}

public void merge(Heap<E> obj){ **ϴ(n2)**

for(int i=0; i<obj.size(); i++) offer(obj.poll());**O(n) \* ϴ(1) \* O(n)**

}

public E removeith(int i){ **O(n)**

if(heapData.size() == 0) return null; **ϴ(1)**

E item = get(i); **ϴ(1)**

remove(i); **O(n)**

return item; **ϴ(1)**

}

public int size(){**ϴ(1)**

return heapData.size();**ϴ(1)**

}

public E poll(){**O(n)**

return heapData.poll();**O(n)**

}

public E peek(){**ϴ(1)**

return heapData.peek();**ϴ(1)**

}

public E get(int i){ **ϴ(1)**

return heapData.get(i); **ϴ(1)**

}

public boolean offer(E item){ **amortized ϴ(1)**

if(search(item) == -1) return heapData.offer(item); **amortized ϴ(1)**

else return false; **ϴ(1)**

}

public boolean remove(int i){ **O(n)**

return heapData.remove(i); **O(n)**

}

public int compareTo(E item){ **ϴ(1)**

return heapData.get(0).compareTo(item); **ϴ(1)**

}

public int compareTo(Heap<E> obj){ **ϴ(1)**

return heapData.get(0).compareTo(obj.get(0)); **ϴ(1)**

}

public HeapIterator iterator(){**ϴ(1)**

return new HeapIterator();**ϴ(1)**

}

public HeapIterator(int i){ **ϴ(1)**

if(i > 0 && i < heapData.size()-1) count = i; **ϴ(1)**

else throw new NoSuchElementException();**ϴ(1)**

}

public void setLastValue(E item){ **amortized ϴ(1)**

heapData.offer(item); **amortized ϴ(1)**

}

public boolean hasNext(){**ϴ(1)**

if(count == heapData.size()-1) return false; **ϴ(1)**

else return true; **ϴ(1)**

}

public E next(){**ϴ(1)**

if(hasNext()){**ϴ(1)**

lastItemReturned = heapData.get(count); **ϴ(1)**

count++; **ϴ(1)**

return lastItemReturned; **ϴ(1)**

}else{

lastItemReturned = null; **ϴ(1)**

return null; **ϴ(1)**

}

}

public void remove(){**O(n)**

if(heapData.size() != 0){ **ϴ(1)**

heapData.remove(count); **O(n)**

}

}

public BSTHeapTree(){**ϴ(1)**

Heap tempHeap = new Heap();**ϴ(1)**

theData.add(tempHeap); **ϴ(1)**

}

public BSTHeapTree(E item){ **ϴ(1)**

Heap tempHeap = new Heap(item); **ϴ(1)**

theData.add(tempHeap); **ϴ(1)**

}

public BSTHeapTree(Heap<E> obj){ **ϴ(1)**

theData.add(obj); **ϴ(1)**

}

public int add(E item){ **O(n)**

//return add(item, root); **O(n)**

return 1; **ϴ(1)**

}

private int add(E item, Node<Heap<E>> root){ **O(n)**

if(theData.size() == 7){ **ϴ(1)**

Node<Heap<E>> tempNode = new Node(item); **ϴ(1)**

root.left = tempNode; **ϴ(1)**

inc(item, root); **O(n)**

return find(item); **O(n)**

}else if(root.left == null && root.right == null){ **ϴ(1)**

Heap<E> tempHeap = new Heap(item); **ϴ(1)**

theData.add(tempHeap); **O(n)**

inc(item, root); **O(n)**

return find(item); **ϴ(1)**

}else if(root.left != null) return add(item, root.left); **O(n)**

else return add(item, root.right); **O(n)**

}

private void inc(E item, Node<Heap<E>> root){ **O(n)**

if(root.data == item){ **ϴ(1)**

root.incMode();**ϴ(1)**

return; **ϴ(1)**

}else if(root.left == null && root.right == null) return; **ϴ(1)**

else if(root.left == null) inc(item, root.right); **O(n)**

else inc(item, root.left); **O(n)**

}

private void dec(E item, Node<Heap<E>> root){ **O(n)**

if(root.data == item){ **ϴ(1)**

root.decMode();**ϴ(1)**

return; **ϴ(1)**

}else if(root.left == null && root.right == null) return; **ϴ(1)**

else if(root.left == null) dec(item, root.right); **O(n)**

else dec(item, root.left); **O(n)**

}

public int remove(E item){ **O(n)**

//return remove(item, root); **O(n)**

return 1; **ϴ(1)**

}

private int remove(E item, Node<Heap<E>> root){ **O(n)**

if(root.data == item){ **ϴ(1)**

remove(item); **O(n)**

dec(item, root); **O(n)**

return find(item); **O(n)**

}else if(root.left == null && root.right == null) return 0; **ϴ(1)**

else if(root.left == null) return remove(item, root.right); **O(n)**

else return remove(item, root.left); **O(n)**

}

public int find(E target){ **O(n)**

return find(target); **O(n)**

}

public int find(E target, Node<Heap<E>> root){ **O(n)**

if(root.data == target) return root.mode; **ϴ(1)**

else if(root.left == null && root.right == null) return 0; **ϴ(1)**

else if(root.left != null) return find(target, root.left); **O(n)**

else return find(target, root.right); **O(n)**

}

public int find\_mode(E item){ **O(n)**

//return find\_mode(item, theData.root); **O(n)**

return 1; **ϴ(1)**

}

private int find\_mode(E item, Node<Heap<E>> root){ **O(n)**

if(root.data == item) return root.mode; **ϴ(1)**

else if(root.left == null && root.right == null) return 0; **ϴ(1)**

else if(root.left != null) return find\_mode(item, root.left); **O(n)**

else return find\_mode(item, root.right); **O(n)**

}

public Node(){**ϴ(1)**

left = null; **ϴ(1)**

right = null; **ϴ(1)**

}

public Node(E data) { **ϴ(1)**

this.data = data; **ϴ(1)**

left = null; **ϴ(1)**

right = null; **ϴ(1)**

}

public String toString () { **O(n)**

return data.toString();**O(n)**

}

public BinaryTree() { **ϴ(1)**

root = null; **ϴ(1)**

}

protected BinaryTree(Node<E> root) { **ϴ(1)**

this.root = root; **ϴ(1)**

}

public int size(){**O(n)**

return size(root); **O(n)**

}

private int size(Node<E> root){ **O(n)**

if(root.left == null && root.right == null) return 0; **ϴ(1)**

else if(root.left != null) return 1 + size(root.left); **O(n)**

else return 1 + size(root.right); **O(n)**

}

public BinaryTree(E data, BinaryTree<E> leftTree,

BinaryTree<E> rightTree) { **ϴ(1)**

root = new Node<>(data); **ϴ(1)**

if (leftTree != null) { **ϴ(1)**

root.left = leftTree.root; **ϴ(1)**

} else {

root.left = null; **ϴ(1)**

}

if (rightTree != null) { **ϴ(1)**

root.right = rightTree.root; **ϴ(1)**

} else {

root.right = null; **ϴ(1)**

}

}

public BinaryTree<E> getLeftSubtree() { **ϴ(1)**

if (root != null && root.left != null) { **ϴ(1)**

return new BinaryTree<>(root.left); **ϴ(1)**

} else {

return null; **ϴ(1)**

}

}

public BinaryTree<E> getRightSubtree() { **ϴ(1)**

if (root != null && root.right != null) { **ϴ(1)**

return new BinaryTree<>(root.right); **ϴ(1)**

} else {

return null; **ϴ(1)**

}

}

public boolean isLeaf() { **ϴ(1)**

return (root.left == null && root.right == null); **ϴ(1)**

}

public String toString() { **O(n)**

StringBuilder sb = new StringBuilder();**ϴ(1)**

toString(root, 1, sb); **O(n)**

return sb.toString();**O(n)**

}

private void toString(Node<E> node, int depth,

StringBuilder sb) { **O(n)**

for (int i = 1; i < depth; i++) { **ϴ(1)**

sb.append(" "); **ϴ(1)**

}

if (node == null) { **ϴ(1)**

sb.append("null\n"); **ϴ(1)**

} else {

sb.append(node.toString());**O(n)**

sb.append("\n"); **ϴ(1)**

toString(node.left, depth + 1, sb); **O(n)**

toString(node.right, depth + 1, sb); **O(n)**

}

}

public static BinaryTree<String> readBinaryTree(Scanner scan) { **ϴ(1)**

String data = scan.nextLine().trim();**ϴ(1)**

if (data.equals("null")) { **ϴ(1)**

return null; **ϴ(1)**

} else {

BinaryTree<String> leftTree = readBinaryTree(scan); **ϴ(1)**

BinaryTree<String> rightTree = readBinaryTree(scan); **ϴ(1)**

return new BinaryTree<>(data, leftTree, rightTree); **ϴ(1)**

}

}

public E find(E target) { **O(n)**

return find(root, target); **O(n)**

}

private E find(Node<E> localRoot, E target) { **O(n)**

if (localRoot == null) **ϴ(1)**

return null; **ϴ(1)**

int compResult = target.compareTo(localRoot.data); **ϴ(1)**

if (compResult == 0) **ϴ(1)**

return localRoot.data; **ϴ(1)**

else if (compResult < 0) **ϴ(1)**

return find(localRoot.left, target); **O(n)**

else

return find(localRoot.right, target); **O(n)**

}

public boolean add(E item) { **O(n)**

root = add(root, item); **O(n)**

return addReturn; **ϴ(1)**

}

private Node<E> add(Node<E> localRoot, E item) { **O(n)**

if (localRoot == null) { **ϴ(1)**

addReturn = true; **ϴ(1)**

return new Node<>(item); **ϴ(1)**

} else if (item.compareTo(localRoot.data) == 0) { **ϴ(1)**

addReturn = false; **ϴ(1)**

return localRoot; **ϴ(1)**

} else if (item.compareTo(localRoot.data) < 0) { **ϴ(1)**

localRoot.left = add(localRoot.left, item); **O(n)**

return localRoot; **ϴ(1)**

} else {

localRoot.right = add(localRoot.right, item); **O(n)**

return localRoot; **ϴ(1)**

}

}

public E delete(E target) { **O(n)**

root = delete(root, target); **O(n)**

return deleteReturn; **ϴ(1)**

}

private Node<E> delete(Node<E> localRoot, E item) { **ϴ(1)**

if (localRoot == null) { **ϴ(1)**

deleteReturn = null; **ϴ(1)**

return localRoot; **ϴ(1)**

}

int compResult = item.compareTo(localRoot.data); **ϴ(1)**

if (compResult < 0) { **ϴ(1)**

localRoot.left = delete(localRoot.left, item); **O(n)**

return localRoot; **ϴ(1)**

} else if (compResult > 0) { **ϴ(1)**

localRoot.right = delete(localRoot.right, item); **ϴ(1)**

return localRoot; **ϴ(1)**

} else {

deleteReturn = localRoot.data; **ϴ(1)**

if (localRoot.left == null) { **ϴ(1)**

return localRoot.right; **ϴ(1)**

} else if (localRoot.right == null) { **ϴ(1)**

return localRoot.left; **ϴ(1)**

} else {

if (localRoot.left.right == null) { **ϴ(1)**

localRoot.data = localRoot.left.data; **ϴ(1)**

localRoot.left = localRoot.left.left; **ϴ(1)**

return localRoot; **ϴ(1)**

} else {

localRoot.data = findLargestChild(localRoot.left); **O(n)**

return localRoot; **ϴ(1)**

}

}

}

}

private E findLargestChild(Node<E> parent) { **O(n)**

if (parent.right.right == null) { **ϴ(1)**

E returnValue = parent.right.data; **ϴ(1)**

parent.right = parent.right.left; **ϴ(1)**

return returnValue; **ϴ(1)**

} else {

return findLargestChild(parent.right); **O(n)**

}

}

public boolean remove(E target){ **O(n)**

if(delete(target) == null) return false; **ϴ(1)**

else delete(target); **O(n)**

return true; **ϴ(1)**

}

public boolean contains(E target){ **O(n)**

if(find(target) == null) return false; **O(n)**

else return true; **ϴ(1)**

}