**Time Complexities**

**1-) MaxHeap Class Methods**

-private int compare(E left, E right)

{

if(comparator!=null)

return comparator.compare(left, right);

return ((Comparable<E>)left).compareTo(right);

} **T(n)= Θ(1)**

-private void swap(int i, int j)

{

Node<E> temp=theData.get(i);

theData.set(i, theData.get(j));

theData.set(j, temp);

} **T(n)= Θ(1)** Because theData is an ArrayList and get method of array list takes constant time. Set method of an ArrayList also takes constant time.

-private boolean swapUp(int childNo)

{

boolean returnVal=false;

int parent=(childNo-1)/2;

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

{

swap(parent, childNo);

childNo=parent;

parent=(childNo-1)/2;

returnVal=true;

}

return returnVal;

} **T(n)= O(logn)** compare, get and swap methods are all constant time methods. There is a while loop. This while loop will has Tb(n)=Θ(1) because the loop will run only once if only one thing needs to be swapped. Tw(n)= O(logn)because the loop will run maximum height of the heap times. Height of the heap has a logn relationship with the element amount in the heap.

-private boolean swapDown(int parentNo)

{

boolean returnVal=false;

while(true)

{

int leftC=(parentNo\*2)+1;

int rightC=(parentNo\*2)+2;

if(leftC>=theData.size())

break;

int maxC=0;

if(rightC<theData.size() && compare(theData.get(rightC).data, theData.get(leftC).data)>0)

maxC=rightC;

else

maxC=leftC;

if(compare(theData.get(parentNo).data, theData.get(maxC).data)<0)

{

swap(parentNo, maxC);

parentNo=maxC;

returnVal=true;

}

else

break;

}

return returnVal;

} **T(n)= O(logn)** compare, get, swap and size methods are all constant time methods. There is a while loop. This while loop will has Tb(n)=Θ(1) because the loop will run only once if only one thing needs to be swapped. Tw(n)= O(logn)because the loop will run maximum height of the heap times. Height of the heap has a logn relationship with the element amount in the heap.

-private int findNthGreatest(int greatestNo)

{

MaxHeap<E> temp=new MaxHeap<E>();

temp.merge(this);

for(int i=1; i<greatestNo; ++i)

temp.poll();

return search(temp.peek());

} **T(n)= O(nlogn)** mergemethod is O(nlogn), search method is O(n), poll method is O(logn). The for loop will run greatestNo-1 times therefore it is O(logn)xO(n).

-public int size()

{

return theData.size();

} **T(n)= Θ(1)** size method of ArrayList is constant time.

-public int getCount(int index)

{

if(index>=theData.size())

throw new IllegalArgumentException();

return theData.get(index).count;

} **T(n)= Θ(1)** get method of ArrayList is constant time.

-public int getMode()

{

int tempMode=theData.get(0).count;

for(int i=1; i<theData.size(); ++i)

if(theData.get(i).count>tempMode)

tempMode=theData.get(i).count;

return tempMode;

} **T(n)= Θ(n)** get method of ArrayList is constant time. The for loop will run n-1 times.

-public E getModeElement()

{

Node<E> tempMode=theData.get(0);

for(int i=1; i<theData.size(); ++i)

if(theData.get(i).count>tempMode.count)

tempMode=theData.get(i);

return tempMode.data;

} **T(n)= Θ(n)** get method of ArrayList is constant time. The for loop will run n-1 times.

-public boolean isEmpty()

{

if(theData==null || theData.size()==0)

return true;

return false;

} **T(n)= Θ(1)** size method of ArrayList is constant time.

-public boolean isFull()

{

if(theData.size()==7)

return true;

return false;

} **T(n)= Θ(1)** size method of ArrayList is constant time.

-public boolean contains(E element)

{

if(search(element)!=-1)

return true;

return false;

} **T(n)= O(n)** because search method is O(n).

-public void printArray()

{

for(int i=0; i<theData.size(); ++i)

System.out.println(theData.get(i).data+","+theData.get(i).count);

} **T(n)= Θ(n)** get method of ArrayList is constant time. The for loop will run n-1 times.

-public E peek()

{

if(isEmpty())

return null;

return theData.get(0).data;

} **T(n)= Θ(1)** get method of ArrayList is constant time. isEmpty is also constant time.

-public void merge(MaxHeap<E> other)

{

for(int i=0; i<other.theData.size(); ++i)

offer(other.theData.get(i));

} **T(n)= O(nlogn)** because offer method is O(logn) and get method is constant time. The loop will run n times.

-public int compareTo(Object o)

{

return compare(peek(), ((MaxHeap<E>) o).peek());

} **T(n)= Θ(1)** peek method is constant time.

-public String toString()

{

if(isEmpty())

return String.format(null);

return peek().toString();

} **T(n)= Θ(1)** peek and isEmpty methods are constant time.

-public boolean offer(E element)

{

int contains=search(element);

if(contains!=-1)

{

theData.get(contains).count++;

return true;

}

if(theData.size()==7)

return false;

theData.add(new Node<E>(element));

int child=theData.size()-1;

swapUp(child);

return true;

} **T(n)= O(nlogn)** because swapUp method is O(logn) get, size, ArrayList’s add methods are all constant time.

-private boolean offer(Node<E> element)

{

int contains=search(element.data);

if(contains!=-1)

{

theData.get(contains).count++;

return true;

}

if(theData.size()==7)

return false;

theData.add(new Node<E>(element.data, element.count));

int child=theData.size()-1;

swapUp(child);

return true;

} **T(n)= O(n)** because swapUp method is O(logn) get, size, ArrayList’s add methods are all constant time. However search method is O(n)

-public E poll()

{

if(isEmpty())

return null;

if(theData.get(0).count!=1)

{

theData.get(0).count--;

return theData.get(0).data;

}

Node<E> temp=theData.get(theData.size()-1);

theData.remove(theData.size()-1);

if(theData.size()==0)

return temp.data;

E temp2=theData.get(0).data;

theData.set(0, temp);

int parent=0;

swapDown(parent);

return temp2;

} **T(n)= O(logn)** because swapDown method is O(logn) get, size, ArrayList’s remove methods (because removing from the end) are all constant time.

-public int search(E element)

{

if(isEmpty())

return -1;

if(compare(theData.get(0).data, element)<0)

return -1;

for(int i=0; i<theData.size(); ++i)

if(compare(theData.get(i).data, element)==0)

return i;

return -1;

} **T(n)= O(n)** because compare, get, size methods are all constant time. There is a for loop that will run maximum of n times. However if the heap is empty or the searched element is bigger than the root element then it is Θ(1).

-public E remove(int largestNo)

{

if(theData.size()<largestNo || largestNo<1)

return null;

int indexNo=findNthGreatest(largestNo);

MaxHeapIterator iter=iterator(indexNo);

E temp=(E)iter.next();

iter.remove();

return temp;

} **T(n)= O(nlogn)** Because findNthGreatest is O(nlogn) and iter.next, iter.remove take only O(logn) time.

-public MaxHeapIterator iterator()

{

return new MHIterator();

} **T(n)= Θ(1)**

-public MaxHeapIterator iterator(int index)

{

return new MHIterator(index);

} **T(n)= Θ(1)**

-public boolean hasNext()

{

if(nextItem!=-1)

return true;

return false;

} **T(n)= Θ(1)**

-public E next()

{

if(hasNext()==false)

throw new NoSuchElementException();

lastReturned=nextItem;

index++;

if(index==theData.size())

nextItem=-1;

else

nextItem=index;

return theData.get(lastReturned).data;

} **T(n)= Θ(1)** Because size and get methods are constant time.

-public E setVal(E value, int amount)

{

if(lastReturned==-1)

return null;

E temp=theData.get(lastReturned).data;

theData.set(lastReturned, new Node<E>(value, amount));

if(swapUp(lastReturned))

return temp;

else

swapDown(lastReturned);

return temp;

} **T(n)= O(logn)** Because swapUp and swapDown methos are O(logn)

-public void remove()

{

if(lastReturned==-1)

return;

if(theData.get(lastReturned).count!=1)

{

theData.get(lastReturned).count--;

return;

}

if(index==theData.size())

{

theData.remove(theData.size()-1);

return;

}

Node<E> temp=theData.get(theData.size()-1);

theData.remove(theData.size()-1);

setVal(temp.data, temp.count);

} **T(n)= O(logn)** Because setVal method is O(logn)

**2-) BinarySearchTree Class Methods**

-public String toString()

{

return data.toString();

} **T(n)= Θ(1)**

-public boolean isEmpty()

{

if(root==null)

return true;

return false;

} **T(n)= Θ(1)**

-public E getRoot()

{

return root.data;

} **T(n)= Θ(1)**

-public BinarySearchTree<E> getLeftSubtree()

{

if (root != null && root.left != null)

return new BinarySearchTree<>(root.left);

else

return null;

} **T(n)= Θ(1)**

-public BinarySearchTree<E> getRightSubtree()

{

if (root != null && root.right != null)

return new BinarySearchTree<>(root.right);

else

return null;

} **T(n)= Θ(1)**

-public boolean isLeaf()

{

return (root.left == null && root.right == null);

} **T(n)= Θ(1)**

-public boolean contains(E target)

{

if(find(target)!=null)

return true;

return false;

} **T (n)= O(n)** Because find method takes O(n) time.

-public E find(E target)

{

return find(root, target);

} **T (n)= O(n)** Because find method takes O(n) time.

-private E find(Node<E> localRoot, E target)

{

if (localRoot == null)

return null;

// Compare the target with the data field at the root.

int compResult = ((Comparable<E>)target).compareTo(localRoot.data);

if (compResult == 0)

return localRoot.data;

else if (compResult < 0)

return find(localRoot.left, target);

else

return find(localRoot.right, target);

} **T (n)= O(n)** This find method will search through elements till it finds the target or reaches a leaf node. This search will take Tb(n)=Θ(1) at best if it finds the target directly. However in the worst case it will run n times depending on the height. T(n)=O(n). Also

logn<height<n therefore we can say it is **Tav(n)=O(logn).**

-public boolean add(E item)

{

root = add(root, item);

return addReturn;

} **T (n)= O(n)** Because add method takes O(n) time.

-private Node<E> add(Node<E> localRoot, E item)

{

if (localRoot == null)

{

// item is not in the tree ÃƒÂ¢Ã¢â€šÂ¬Ã¢â‚¬ï¿½ insert it.

addReturn = true;

return new Node<>(item);

}

else if (((Comparable<E>)item).compareTo(localRoot.data) == 0)

{

// item is equal to localRoot.data

addReturn = false;

return localRoot;

}

else if (((Comparable<E>)item).compareTo(localRoot.data) < 0)

{

// item is less than localRoot.data

localRoot.left = add(localRoot.left, item);

return localRoot;

}

else

{

// item is greater than localRoot.data

localRoot.right = add(localRoot.right, item);

return localRoot;

}

} **T (n)= O(n)** This find method will search through elements till it finds a proper place to add the element. This search will take Tb(n)=Θ(1) at best if it finds the target directly. However in the worst case it will run n times depending on the height. T(n)=O(n). Also

logn<height<n therefore we can say it is **Tav(n)=O(logn).**

-public E delete(E target)

{

root = delete(root, target);

return deleteReturn;

} **T (n)= O(n)** Because delete method takes O(n) time.

-private Node<E> delete(Node<E> localRoot, E item)

{

if (localRoot == null)

{

// item is not in the tree.

deleteReturn = null;

return localRoot;

}

// Search for item to delete.

int compResult = ((Comparable<E>)item).compareTo(localRoot.data);

if (compResult < 0)

{

// item is smaller than localRoot.data.

localRoot.left = delete(localRoot.left, item);

return localRoot;

}

else if (compResult > 0)

{

// item is larger than localRoot.data.

localRoot.right = delete(localRoot.right, item);

return localRoot;

}

else

{

// item is at local root.

deleteReturn = localRoot.data;

if (localRoot.left == null)

{

// If there is no left child, return right child

// which can also be null.

return localRoot.right;

}

else if (localRoot.right == null)

{

// If there is no right child, return left child.

return localRoot.left;

}

else

{

// Node being deleted has 2 children, replace the data

// with inorder predecessor.

if (localRoot.left.right == null)

{

// The left child has no right child.

// Replace the data with the data in the

// left child.

localRoot.data = localRoot.left.data;

// Replace the left child with its left child.

localRoot.left = localRoot.left.left;

return localRoot;

}

else

{

// Search for the inorder predecessor (ip) and

// replace deleted node's data with ip.

localRoot.data = findLargestChild(localRoot.left);

return localRoot;

}

}

}

} **T (n)= O(n)** This find method will search through elements till it manages to find and delete the element then fix the tree structure. This search will take Tb(n)=Θ(1) at best if it does directly. However in the worst case it will run n times depending on the height. T(n)=O(n). Also

logn<height<n therefore we can say it is **Tav(n)=O(logn).**

-private E findLargestChild(Node<E> parent)

{

// If the right child has no right child, it is

// the inorder predecessor.

if (parent.right.right == null)

{

E returnValue = parent.right.data;

parent.right = parent.right.left;

return returnValue;

}

else

{

return findLargestChild(parent.right);

}

} **T(n)= Θ(1)**

-private void toString(Node<E> node, int depth, StringBuilder sb)

{

for (int i = 1; i < depth; i++)

sb.append(" ");

if (node == null)

sb.append("null\n");

else

{

sb.append(node.toString());

sb.append("\n");

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

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

}

}

-public String toString()

{

StringBuilder sb = new StringBuilder();

toString(root, 1, sb);

return sb.toString();

}**T(n,h)=O(n\*h)** Because it will do the operations for each element in the tree also it depends on the height of the tree too. There will be many “ “ appended to the string depending on the height. In the best case height equals to logn. Tb(n)=O(nlogn) Tw(n)=O(n^2).

**3-) BSTHeapTree Class Methods**

-public boolean isEmpty()

{

if(theData==null || theData.isEmpty())

return true;

return false;

} **T(n)= Θ(1)**

-public String toString()

{

return theData.toString();

} **T(n)= Θ(1)**

-private void printArray(BinarySearchTree<MaxHeap<E>> newRoot)

{

if(newRoot==null)

return;

newRoot.getRoot().printArray();

System.out.println();

printArray(newRoot.getLeftSubtree());

printArray(newRoot.getRightSubtree());

}**T(n,k)=O(n\*k)** printArray method of MaxHeap is O(n). This method will be done for each node of the tree. If three has k nodes it will be done k times. O(n)\*O(k)=O(n\*k). In the worst case if n>k it is O(n^2)

-public void printArray()

{

if(isEmpty())

return;

printArray(theData);

} **T(n,k)=O(n\*k)** Because printArray method is O(n\*k).

-public boolean add(MaxHeap<E> element)

{

return theData.add(element);

} **T (n)= O(n)** because BinarySearchTree add method is O(n).

-private int add(BinarySearchTree<MaxHeap<E>> newRoot, E element, int count)

{

if(newRoot==null)

{

MaxHeap<E> newHeap=new MaxHeap<E>(element);

add(newHeap);

return 1;

}

else if(!newRoot.getRoot().isFull() || newRoot.getRoot().contains(element))

{

newRoot.getRoot().offer(element);

int amount=newRoot.getRoot().getCount(newRoot.getRoot().search(element));

return amount;

}

else if(((Comparable<E>)element).compareTo(newRoot.getRoot().peek())<0)

count=add(newRoot.getLeftSubtree(), element, count);

else

count=add(newRoot.getRightSubtree(), element, count);

return count;

}**T(n)=O(nlogn)** The best case happens when newRoot is null and only the first if statement is run. Tb(n)=O(n). The worst case happens when the proper place to add the element is searched till the node that has the greatest height. Each searching process will happen in Θ(1) time however it will happen h-1 times therefore O(h). When the proper place is found at h’th height first else if statement will run and the offer method there has O(nlogn) time complexity. In the end Tw(n)=O(h)+O(nlogn)=O(nlogn)

-public int add(E element)

{

if(isEmpty())

{

MaxHeap<E> newHeap=new MaxHeap<E>(element);

add(newHeap);

return 1;

}

return add(theData, element, 0);

} **T(n)=O(nlogn)** Because add method is O(nlogn). However in the best case, if the tree is empty, it is O(n) because of the other add method.

-private int find(BinarySearchTree<MaxHeap<E>> newRoot, E element, int count)

{

if(newRoot==null)

return count+0;

int indexNo=newRoot.getRoot().search(element);

if(indexNo!=-1)

return count+newRoot.getRoot().getCount(indexNo);

else if(indexNo==-1 && ((Comparable<E>)element).compareTo(newRoot.getRoot().peek())<0)

count=find(newRoot.getLeftSubtree(), element, count);

else

count=find(newRoot.getRightSubtree(), element, count);

return count;

}**T(n)=O(n^2)** Because the method will be repeated till it finds the element. In the worst case the element cannot be found meaning the method should run n times. In each run, there is search method. This method is O(n). Therefore O(n)\*O(n)=O(n^2).

-public int find(E element)

{

if(isEmpty())

return 0;

return find(theData, element, 0);

} **T(n)=O(n^2)** Because find method is O(n^2). However in the best case, if the tree is empty, it is Θ(1).

-private E find\_mode(BinarySearchTree<MaxHeap<E>> newRoot, E tempModeE, Integer tempMode)

{

if(newRoot==null)

return tempModeE;

if(newRoot.getRoot().getMode()>tempMode)

{

tempMode=newRoot.getRoot().getMode();

tempModeE=newRoot.getRoot().getModeElement();

}

tempModeE=find\_mode(newRoot.getLeftSubtree(), tempModeE, tempMode);

tempMode=find(tempModeE);

tempModeE=find\_mode(newRoot.getRightSubtree(), tempModeE, tempMode);

tempMode=find(tempModeE);

return tempModeE;

}**T(n)=O(n^2)** This method will run n times for each node in the tree. In each run there is getMode and getModeElements method which both take O(n) time. O(n)\*(n)=O(n^2).

-public E find\_mode()

{

if(isEmpty())

return null;

else if(theData.getLeftSubtree()==null && theData.getRightSubtree()==null)

return theData.getRoot().getModeElement();

E modeElement=theData.getRoot().getModeElement();

Integer mode=new Integer(theData.getRoot().getMode());

modeElement=find\_mode(theData, modeElement, mode);

return modeElement;

} **T(n)=O(n^2)** Because find\_mode method is O(n^2). However in the best case, if the tree is empty, it is Θ(1).

-public MaxHeap<E> remove(MaxHeap<E> element)

{

return theData.delete(element);

} **T (n)= O(n)** Because delete method takes O(n) time.

-private int remove(BinarySearchTree<MaxHeap<E>> newRoot, E element, int count)

{

if(newRoot==null)

return 0;

int indexNo=newRoot.getRoot().search(element);

if(indexNo!=-1)

{

int amount=newRoot.getRoot().getCount(indexNo);

if(amount==1 && newRoot.getRoot().size()==1)

{

remove(newRoot.getRoot());

return 0;

}

MaxHeapIterator iter=newRoot.getRoot().iterator(indexNo);

E temp=(E)iter.next();

iter.remove();

return amount-1;

}

else if(indexNo==-1 && ((Comparable<E>)element).compareTo(newRoot.getRoot().peek())<0)

count=remove(newRoot.getLeftSubtree(), element, count);

else

count=remove(newRoot.getRightSubtree(), element, count);

return count;

} **Tw(n,h)=(h\*n)**The best case happens if newRoot is null at the start. Tb(n)= Θ(1). The worst case happens if the element cannot be found or if the element is found and removed but also it’s node should be deleted too. When that happens firstly the method runs h times (h=height of the tree) and in each run search method is run too which has O(n) time complexity. In the end remove method might be run too which also has O(n) time complexity. Tw(n,h)=O(n)\*O(h)=(h\*n)

-private MaxHeap<E> findLeaf(BinarySearchTree<MaxHeap<E>> newRoot)

{

if(newRoot.isLeaf())

return newRoot.getRoot();

MaxHeap<E> leafHeap=null;

if(newRoot.getLeftSubtree()!=null)

leafHeap=findLeaf(newRoot.getLeftSubtree());

else

leafHeap=findLeaf(newRoot.getRightSubtree());

return leafHeap;

}**T(n)=O(n)** The best case happens if newRoot is a leaf at the start. Tb(n)= Θ(1). The worst case happens if the leaf is far from the first newRoot. Tw(n)=O(n) (n=height)

-private void fixTree(BinarySearchTree<MaxHeap<E>> newRoot)

{

if(newRoot==null || newRoot.isLeaf())

return;

while(!newRoot.getRoot().isFull())

{

MaxHeap<E> chosenLeaf=findLeaf(newRoot);

E temp;

int count;

if(chosenLeaf.size()==1 && chosenLeaf.getCount(0)==1)

{

temp=chosenLeaf.peek();

remove(chosenLeaf);

count=1;

}

else

{

count=chosenLeaf.getCount(0);

temp=chosenLeaf.poll();

for(int i=0; i<count-1; ++i)

chosenLeaf.poll();

}

for(int i=0; i<count; ++i)

newRoot.getRoot().offer(temp);

}

fixTree(newRoot.getLeftSubtree());

fixTree(newRoot.getRightSubtree());

}**T(n)=O(nlogn)** The best case happens if the newRoot is a leaf node or null at the start. Tb(n)= Θ(1). The worst case happens if the newRoot has only 1 element inside and needs to be filled with 6 more elements. Fort his the while loop needs to run 6 times at max. Inside the while loop there is findLeaf which will run in each iteration. It is O(n). There is poll method which might run in each iteration and it is O(logn). Therefore O(n)\*O(logn)

-public int remove(E element)

{

if(isEmpty())

return 0;

int result=remove(theData, element, 0);

fixTree(theData);

return result;

}**T(n)=O(n\*h)** remove method is O(n\*h), fixTree method is O(nlogn) O(n\*h)>O(nlogn)

**4-) HelperMethods Class Methods**

-public int findOccurences(E[] array, E number)

{

int count=0;

int start=findFirstIndex(array, number);

if(start==-1)

return 0;

for(int i=start; i<array.length; ++i)

{

if(i==array.length-1 && array[i].equals(array[i-1]))

count++;

else if(i==array.length-1 && !array[i].equals(array[i-1]))

break;

else if(array[i].equals(array[i+1]))

count++;

else

{

count++;

break;

}

}

return count;

}**T(n)=O(n)** findFirstIndex is O(n) and the for loop inside the method also runs max n times.

-public int findFirstIndex(E[] array, E element)

{

int start=-1;

for(int i=0; i<array.length; ++i)

if(array[i].equals(element))

{

start=i;

break;

}

return start;

}**T(n)=O(n)** Tb(n)= Θ(1) if the element is at the first index. Tw(n)= Θ (n) if the element is at the last index.

-public E findMode(E[] array)

{

int occurenceCount=findOccurences(array, array[0]);

int modeTemp=occurenceCount;

E modeETemp=array[0];

int i=0;

while(i<array.length)

{

i+=occurenceCount;

if(i>=array.length-1)

break;

occurenceCount=findOccurences(array, array[i]);

if(occurenceCount>modeTemp)

{

modeTemp=occurenceCount;

modeETemp=array[i];

}

}

return modeETemp;

}**T(n)=O(n^2)** findOccurrences is O(n) and this method is used in the while loop. The while loop will run as long as i is smaller than n. i is not incremented by 1 in each run it is incremented by amounts that change all the time however in the worst case, it is incremented by 1 each run and the while loop runs n times. Tw(n)=O(n^2). We can say that in average the while loop will run a lot less than n times. However, we cannot define how many times it runs for sure.