GIT Department of Computer Engineering

CSE 222/505 - Spring 2021

Homework 4 - Report

Canberk Arıcı - 171044062

System Requirements

Part I

The first part of the assignment was divided into four groups.

- Operation of search,
- Merging Heaps,
- eliminating the heap's ith biggest number
- setting the heap's last value with new data passed as parameter

Data must be organised into heap by adding and after these operations have been applied to the heap, they can be completed successfully.

1. The searching operation will scan the entire heap for the specified item and finds the given data from the heap printing depth of given node and boolean value
2. The merging process would combine two heaps and reheapify the new heap.
3. Removing i^{th} maximum number, will remove given i^{th} biggest number in the heap and re-heapify the heap.
4. Setting the last value of heap with new data will use an iterator known as HeapIterator which is extended and set the last data of heap with given value and reheapify current heap.

Part II

.

Part 2 needed to be updated with some new features added to it to make it more compatible to be used as a node data in BST. of heap nodes can have maximum 7 elements. These figures are valid for maximum heap data structures. The maximum number must always be at the root or top of the heap.

- When a new element is added to BST, the program will add it at the end and heapify it upwards in heap until it reaches its proper location. The parent of a number must be larger than the infant, and the child must be smaller than the parent. If the added element is already present, occurrences are increased by 1 for that.

- Finding the element will find the element in the current node if not found then will go to left/right child depending upon BST conditions. Return the number of occurrences if found or -1.
- When using the BST's mode finding process, the program will return the highest number of occurrences in the BST.

- Removing Operation :
  This will remove the element in the heap node of BST conserving the BST properties and conditions of child nodes.

FUNCTIONAL REQUIREMENTS

PART1:

- **Swaps elements**
- **Gets max size**
- **Gets current size**
- **Gets ith element**
- **Resizes heap**
- **Heapifies**
- **Searchs element**
- **Inserts to heap**
- **Prints heap array**
- **Peeks max**
- **Removes max element**
- **Removes ith largest**
- **Merges heaps**
- **Returns heap**
- **Iterates through heap**
    - **Next**
    - **Has next**
    - **Set value**

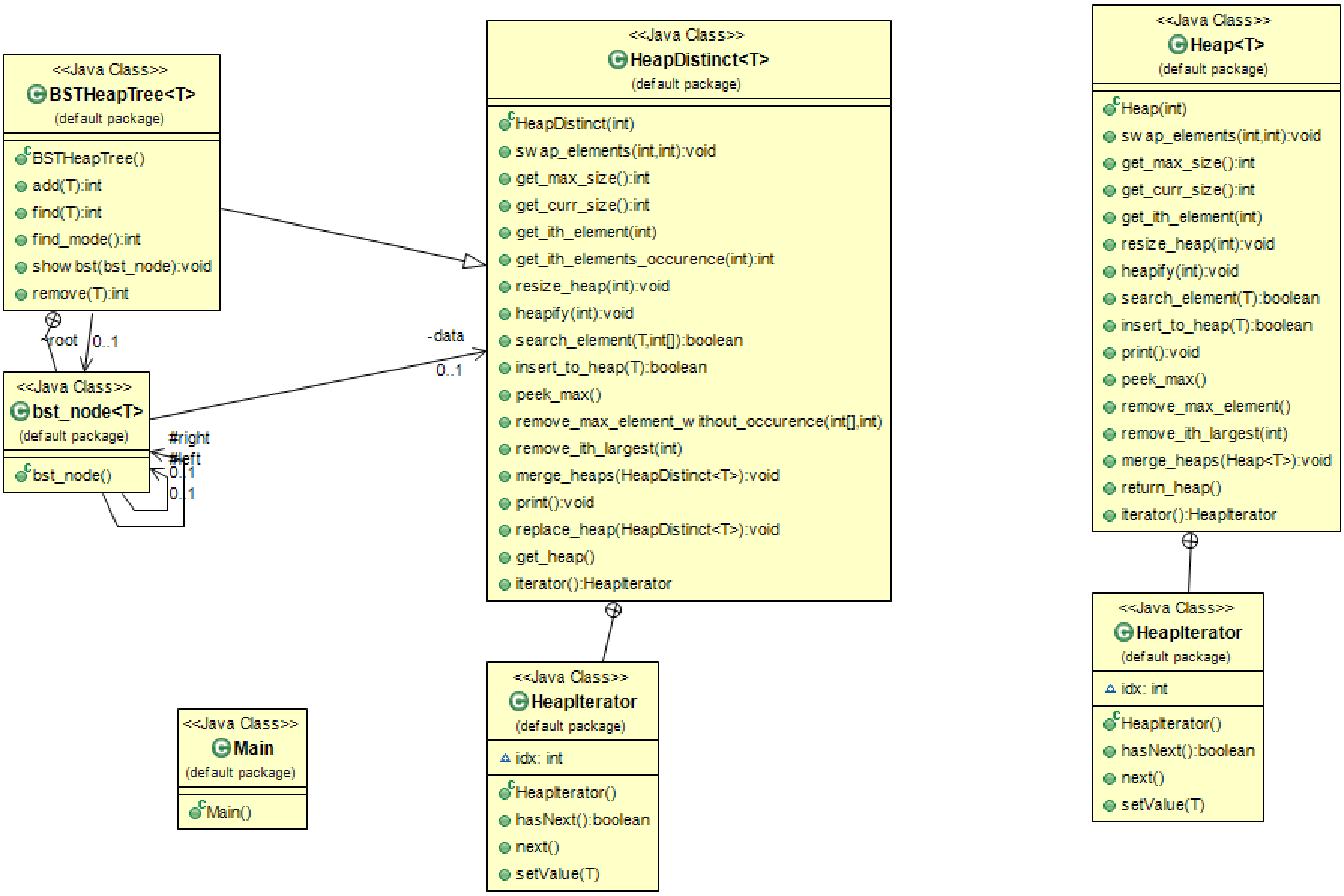
PART2:

- **Swaps elements**
- **Gets max size**
- **Gets current size**
- **Gets ith element**
- **Resizes heap**
- **Heapifies**
- **Searchs element**
- **Inserts to heap**
- **Prints heap array**
- **Peeks max**
- **Removes max element without occurence**
- **Removes ith largest**
- **Prints heap elements with frequencies**
- **Replaces heaps**
- **Gets heap**
- **Iterates through heap**
    - **Next**
    - **Has next**
    - **Set value**
- **Adds element to heap**
- **Finds element in heap**
- **Finds mode in heap**

- **Shows bst**
- **Inserts heap**
- **Deletes node**
- **Removes element**

- **javac 11.0.11 version is used**

CLASS DIAGRAM



Solution Approach

Heap_Distinct class has been made for Part II which is a modified version of Heap class. It has all functionalities of Heap Class for Part I which are modified as nodes will only be distinct which is not the case in Heap class.

Heap and Heap_Distinct Class

- Heap Distinct class is a class in which only distinct elements will be heap nodes whereas heap class can have similar elements as nodes. I used the MaxHeap implementation for both classes. All numbers can be inserted maximum to minimum in heap.Max number has to be at the top.. Heapify down carries small values to down and heapify up takes them to the top.

- Merging Heap implementation is very similar to insert_to_heap method. In which we just resize the current heap and insert all elements of the second heap. Distinction is maintained here otherwise occurrences array has been modified in Heap Distinct Class.

- The implementation of the Removes method is the inverse of that of the Adding Elements to Heap method. In Heap class, the element if found is removed simply. With the last element in the heap, data must shift. Only one size reduction is needed. After that, if the children of the element are larger than our last element, the last element must be modified. This is implemented by the heapify down method.
- Heap Distinct Removal is done on the basis of occurrences if more than 1 occurrences then occurrences-- else the element node is deleted with size - 1.

- Iterator method was written for HeapIterator class, which implements Iterator class, using Iterable class. Instead of the hasNext and next methods, the HeapIterator class now has a setValue method that can be used to set the heap's last value and carry it to top to its correct position in the heap.

BSTHeapTree Class

- BST Heap Tree class is a class representing Binary Search tree with Heap Data structures as Nodes. Basically one node will be having a heap with 7 elements in it which are distinct and their occurrences in it.
- BSTHeapTree Class extends HeapDistinct Class and is generic to data types. Inheritance is used to call parent methods into child class for successful operation of BST.
- BST starts with a root. Left child of the root keeps smaller numbers than root, the right child of the root keeps bigger numbers than root.

- Inserting elements begins with root; if root has already been filled and the item is not in the root heap, the maximum data of the left and right children is compared to the element to be inserted, and the item proceeds in the correct direction satisfying BST conditions. If an element has already been present, its occurrence must be increased by 1.

- Find_Element and Find_mode are closely related to each other as after insertion if we want to find a mode frequency or find an element in BST then I do a recursive postorder traversal of BST and search each heap collecting the maximum frequency or searching for an element in the BSTHeapTree.

- Remove method is the most significant function of this BST. It has many corner cases which have to be handled while removing an element such that the BST conditions are satisfied.

Cases Handled -

i.) When data is to be deleted has the occurrences > 1 them simply return occurrences - 1.
[BST conditions met]

ii.) When data is to be deleted is not a maximum in the heap which it is present then simply delete it and modify the current heap with size by decreasing size by 1 and heapifying the current heap.
[BST conditions met]

iii.) When data is to be deleted is the maximum in the heap which it is present then extracts it and modifies the current heap by size - 1. If the new root of the heap satisfies the BST conditions i.e. (> left && < right) then simply continue.
[BST conditions met]

iv.) When data is to be deleted is the maximum in the heap which it is present then extract it and modify the current heap by size - 1 and now size becomes 0 then basically remove the BSTnode and set the BST according to left subtrees max Heap root and Insert this root again if it has heap as not null in it.
[BST conditions met]

Test Cases Running Command and Results

Part I

Insertion

```java
int arr[] = new int[21];
int itr = 0;

System.out.println("Random Array : ");
for (int i = 0; i < 20; i++) {
    Random r = new Random();
    int num = r.nextInt((5000) + 1);
    arr[itr] = num;
    System.err.print(arr[itr++] + " ");
}

System.out.println();

Heap<Integer> h1 = new Heap<Integer>(itr);
Heap<Integer> h2 = new Heap<Integer>(itr);

for (int i = 0; i < itr; i++) {
    h1.insert_to_heap(arr[i]);
}

System.out.println();
System.out.println("Heap of that array : ");

h1.print();
System.out.println();
```

```
Random Array :
757 4847 3511 2375 866 1934 3028 2091 4588 737 246 360 2857 2622 2061 4864 1163 3075 2292 1881

Heap of that array :
4864 4847 3511 4588 1881 2857 3028 2375 3075 866 246 360 1934 2622 2061 757 1163 2091 2292 737
```

Search

Successful

```
Heap of that array :
4725 4351 3568 4246 3279 1923 3042 3045 1571 3042 1030 1412 1154 348 853 928 2976 137 114
0 852

Please Enter an Element to search :
852

Element Found at depth : 5
Successful Search. 852 is found.
```

Unsuccessful

```
Heap of that array :
4660 4498 3932 4205 3479 2073 2403 1800 2591 3166 2182 171 1398 662 1070 310 1201 991 251
1 2970

Please Enter an Element to search :
4782

Unsuccessful Search. 4782 not found!!
```

Merge Heap

```
System.out.println("Heap 1 : ");
h1.print();

int arr2[] = new int[5];
int itr2 = 0;
for (int i = 0; i < 5; i++) {
    Random r = new Random();
    int num = r.nextInt((5000) + 1);
    arr2[itr2++] = num;
}

System.out.println();
System.out.println("Heap 2 : ");

for (int i = 0; i < itr2; i++) {
    h2.insert_to_heap(arr[i]);
}

h2.print();
System.out.println();
System.out.println();
System.out.println("Merging Heaps :: ");
h1.merge_heaps(h2);
h1.print();
System.out.println();
```

```
Heap 1 :
4576 4238 3911 4041 1985 3289 3609 2649 3177 1148 1392 47 3129 486 3594 891 1663 1465 1400 674
Heap 2 :
4576 1465 3594 891 674

Merging Heaps ::
4576 4576 3911 4041 4238 3289 3609 2649 3177 1985 3594 891 3129 486 3594 891 1663 1465 1400 674 1148 1392 1465 47 674
```

Remove ith Largest

```java
System.out.println();

System.out.println("Heap 2 : ");
h2.print();
System.out.println("Remove Maximum Element " + (h2.remove_ith_largest(1)));
System.out.println("Remove Third largest Element " + (h2.remove_ith_largest(3)));

System.out.println();

System.out.println("Unsuccessful Removal : ");

System.out.println("Remove Element " + (h2.remove_ith_largest(-1)));
```

```
Heap 2 :
4701 4566 3172 119 4354
Remove Maximum Element 4701
Remove Third largest Element 3172

Unsuccessful Removal:
Removal is unsuccessful
```

Iterator Set Method :

```java
h2.insert_to_heap(20);
h2.insert_to_heap(35);
Heap<Integer>.HeapIterator iter = h2.iterator();
System.out.println("Next : " + iter.next());
System.out.println("Next : " + iter.next());
System.out.println("Set : " + iter.setValue(7000));

System.out.println();
System.out.println("After Setting and Heapifying");

h2.print();
```

```
Next : 3600
Next : 680
Set : 35

After Setting and Heapifying
7000 3600 3122 20 680
```

Part II

Inserting integers and finding number of occurences in BSTHeapTree.

```java
// /* Heap Functionaality Part I */
// /* Using class - Heap */

BSTHeapTree<Integer> tree = new BSTHeapTree<Integer>();

int random_array[] = new int[3001];
int itr3 = 0;
Random r2 = new Random();
for (int i = 0; i < 3000; i++) {
    int num = r2.nextInt((5000) + 1);
    random_array[itr3++] = num;
    tree.add(num);
}

Arrays.sort(random_array);
int last = random_array[0];

for (int i = 1; i < itr3; i++) {
    if(random_array[i] != last){
        last = random_array[i];
        System.out.println(last + " occurs : " + tree.find(random_arra
    }
}
}
```

```
68 occurs : 1
69 occurs : 1
73 occurs : 1
81 occurs : 1
83 occurs : 2
85 occurs : 2
87 occurs : 3
88 occurs : 1
91 occurs : 1
93 occurs : 2
95 occurs : 1
97 occurs : 3
99 occurs : 2
104 occurs : 2
105 occurs : 1
109 occurs : 3
110 occurs : 2
112 occurs : 1
114 occurs : 1
116 occurs : 2
117 occurs : 1
119 occurs : 1
120 occurs : 2
121 occurs : 1
122 occurs : 2
123 occurs : 1
124 occurs : 1
```

```
4661 occurs : 1
4662 occurs : 1
4663 occurs : 1
4666 occurs : 1
4668 occurs : 1
4669 occurs : 1
4673 occurs : 1
4674 occurs : 1
4677 occurs : 2
4679 occurs : 1
4680 occurs : 2
4681 occurs : 1
4683 occurs : 1
4693 occurs : 1
4694 occurs : 1
4697 occurs : 2
4698 occurs : 1
4700 occurs : 1
4702 occurs : 1
4703 occurs : 3
4704 occurs : 1
4705 occurs : 2
4706 occurs : 1
4707 occurs : 1
4709 occurs : 2
4714 occurs : 1
```

Searching in BSTHeapTree :

Searching for 100 numbers in array

```java
/* Searching for 100 numbers in Heap */
Arrays.sort(random_array);

int last = 0;
int number_times = 0;
for (int i = 0; i < itr3; i++) {
    // System.out.println(random_array[i]);
    if(number_times == 100){
        break;
    }
    if (random_array[i] != random_array[last]) {
        // System.out.println();
        number_times++;
        System.out.println(random_array[last] + " occurs : " + (i - last) + " in array and " + tree.find(random_array[last])
        last = i;
    }
}

System.out.println();
```

```
458 occurs : 1 in array and 1 times in BST.
462 occurs : 1 in array and 1 times in BST.
470 occurs : 1 in array and 1 times in BST.
483 occurs : 1 in array and 1 times in BST.
491 occurs : 1 in array and 1 times in BST.
527 occurs : 2 in array and 2 times in BST.
538 occurs : 1 in array and 1 times in BST.
540 occurs : 1 in array and 1 times in BST.
557 occurs : 1 in array and 1 times in BST.
573 occurs : 1 in array and 1 times in BST.
578 occurs : 1 in array and 1 times in BST.
585 occurs : 1 in array and 1 times in BST.
603 occurs : 1 in array and 1 times in BST.
610 occurs : 1 in array and 1 times in BST.
622 occurs : 1 in array and 1 times in BST.
628 occurs : 1 in array and 1 times in BST.
644 occurs : 1 in array and 1 times in BST.
647 occurs : 1 in array and 1 times in BST.
650 occurs : 1 in array and 1 times in BST.
651 occurs : 1 in array and 1 times in BST.
663 occurs : 1 in array and 1 times in BST.
666 occurs : 1 in array and 1 times in BST.
710 occurs : 1 in array and 1 times in BST.
734 occurs : 1 in array and 1 times in BST.
739 occurs : 1 in array and 1 times in BST.
741 occurs : 1 in array and 1 times in BST.
```

```
877 occurs : 1 in array and 1 times in BST.
879 occurs : 1 in array and 1 times in BST.
880 occurs : 1 in array and 1 times in BST.
895 occurs : 1 in array and 1 times in BST.
913 occurs : 1 in array and 1 times in BST.
917 occurs : 1 in array and 1 times in BST.
927 occurs : 1 in array and 1 times in BST.
936 occurs : 1 in array and 1 times in BST.
938 occurs : 1 in array and 1 times in BST.
939 occurs : 1 in array and 1 times in BST.
949 occurs : 1 in array and 1 times in BST.
956 occurs : 2 in array and 2 times in BST.
957 occurs : 1 in array and 1 times in BST.
961 occurs : 1 in array and 1 times in BST.
962 occurs : 1 in array and 1 times in BST.
966 occurs : 1 in array and 1 times in BST.
978 occurs : 1 in array and 1 times in BST.
984 occurs : 1 in array and 1 times in BST.
1007 occurs : 1 in array and 1 times in BST.
1040 occurs : 1 in array and 1 times in BST.
1061 occurs : 1 in array and 1 times in BST.
1069 occurs : 1 in array and 1 times in BST.
1078 occurs : 2 in array and 2 times in BST.
1108 occurs : 1 in array and 1 times in BST.
```

Searching for 10 numbers not in array

```
/* Searching for 10 unsuccessful numbers */

for (int i = -10; i < 0; i++) {
    System.out.println(tree.find(i));
}
```

```
-1
-1
-1
-1
-1
-1
-1
-1
-1
-1
```

Removal in BSTHeapTree

Unsuccessful Removal

```
for (int i = 7000; i < 7010 ; i++) {
    System.out.println(i + " before Removal : " + tree.find(i) + " Aft
}
```

```
180 is removed
before Removal : 2 After Removal : 1
7000 before Removal : 0 After Removal : 0
7001 before Removal : 0 After Removal : 0
7002 before Removal : 0 After Removal : 0
7003 before Removal : 0 After Removal : 0
7004 before Removal : 0 After Removal : 0
7005 before Removal : 0 After Removal : 0
7006 before Removal : 0 After Removal : 0
7007 before Removal : 0 After Removal : 0
7008 before Removal : 0 After Removal : 0
7009 before Removal : 0 After Removal : 0
```

Successful Removal

```java
// Remove BST

for (int i = 0; i < 100; i++) {
    System.out.println(random_array[i] + " is removed");
    System.out.println("before Removal : " + tree.find(random_array[i]
            + tree.remove(random_array[i])));
}
```

```
1 is removed
before Removal : 2 After Removal : 1
1 is removed
before Removal : 1 After Removal : 0
3 is removed
before Removal : 3 After Removal : 2
3 is removed
before Removal : 2 After Removal : 1
3 is removed
before Removal : 1 After Removal : 0
4 is removed
before Removal : 2 After Removal : 1
4 is removed
before Removal : 1 After Removal : 0
5 is removed
before Removal : 2 After Removal : 1
5 is removed
before Removal : 1 After Removal : 0
7 is removed
before Removal : 2 After Removal : 1
7 is removed
before Removal : 1 After Removal : 0
9 is removed
before Removal : 1 After Removal : 0
10 is removed
```

FINDING MODE IN BST

```java

        System.out.println("MODE VALUE : " + tree.find_mode());

        last = random_array[0];
        if (tree.find(last) == tree.find_mode()) {
            System.out.println(random_array[0] + " Mode in the BST occurs maxi

        }

        for (int i = 1; i < itr3; i++) {
            if (random_array[i] != last) {
                last = random_array[i];
                if (tree.find(last) == tree.find_mode()) {
                    System.out.println(random_array[i] + " Mode in the BST occ
                }
            }
        }
    }
}
```

```
MODE VALUE : 5
797 Mode in the BST occurs maximum times..
1315 Mode in the BST occurs maximum times..
4249 Mode in the BST occurs maximum times..
4854 Mode in the BST occurs maximum times..
```

Time Complexity Analysis for Heap Class

Constructor

```java
public Heap(int maximum_size) {

    this.heap_size = 0;
    this.MAX_CAPACITY = maximum_size;
    heap = (T[]) new Comparable[this.MAX_CAPACITY + 1];
}
```

O(n)

Swap Elements , Get_maximum_size , Get_Current_size

```java
/* To swap the two elements of heap array */
public void swap_elements(int idx1, int idx2) {
    T elem;

    elem = heap[idx1];

    heap[idx1] = heap[idx2];

    heap[idx2] = elem;
}

/* Get the Maximum Size of the Heap */
public int get_max_size() {
    return MAX_CAPACITY;
}

/* Get the Current Size of the Heap */
public int get_curr_size() {
    return heap_size;
}
```

O(1)

O(1)

O(1)

```java
/* Get Element at Ith Index */
public T get_ith_element(int idx) {

    if (idx < 1 || idx > heap_size) {
        throw new NoSuchElementException("Index Out of Bounds");
    }
    return heap[idx];
}

/* Resize the heap with new size provided */
public void resize_heap(int new_max_size) {
    this.MAX_CAPACITY = new_max_size;
    heap = Arrays.copyOf(heap, this.MAX_CAPACITY);
    return;
}
```

O(1)

O(n)

Heapify (Bottom) takes overall O(n) time.

```java
/* Heapify the heap to bottom carries the smaller element to bottom */

public void heapify(int curr_idx) {
    if ((curr_idx <= heap_size) && (curr_idx > heap_size / 2))
        return;

    if ((heap[curr_idx].compareTo(heap[2 * curr_idx]) < 0)
            || (heap[curr_idx].compareTo(heap[2 * curr_idx + 1]) < 0)) {

        if (heap[2 * curr_idx].compareTo(heap[2 * curr_idx + 1]) <= 0) {
            swap_elements(curr_idx, (2 * curr_idx + 1));
            heapify(2 * curr_idx + 1);
        } else {
            swap_elements(curr_idx, 2 * curr_idx);
            heapify(2 * curr_idx);
        }
    }
}
```

O(1)

O(1)

O(logn) as it traverses to its children which are 2i and 2i + 1 distance apart

Search Element

```java
/* Search an element in the heap */
public boolean search_element(T element) {
    boolean found = false;
    for (int i = 1; i <= heap_size; i++) {
        if (heap[i].compareTo(element) == 0) {
            return true;
        }
    }
    return found;
}
```

O(1)

Overall takes O(n)

Insert to Heap

```java
/* Insert an element to the heap */
public boolean insert_to_heap(T element) {

    /* Not present */

    if (heap_size == MAX_CAPACITY) {
        throw new NoSuchElementException("Heap is Full. Resize it to continue Further.");
    }

    if (heap_size == 0) {
        heap_size++;
        heap[heap_size] = element;
        return true;
    }

    heap_size++;
    heap[heap_size] = element;
    // adjust the heap;

    /* Bring the maximum element to top bottom up approach */
    int newly_inserted_idx = heap_size;
    while ((newly_inserted_idx > 1) && (heap[newly_inserted_idx].compareTo(heap[(newly_inserted_idx / 2)])) > 0) {
        swap_elements(newly_inserted_idx, (newly_inserted_idx / 2));
        newly_inserted_idx = (newly_inserted_idx / 2);
    }
    return true;

}
```

O(1)

Overall takes O(logn)

```java
/*
 * Remove the largest element and decrease size and again heapify to bring
 * second max to array
 */
public T remove_max_element() {

    if (heap_size == 1) {
        heap_size = 0;
        return heap[1];
    }

    T max_elem = heap[1];
    heap[1] = heap[heap_size];
    heap_size -= 1;

    /* Generate New Max */
    heapify(1);
    return max_elem;
}
```

O(1)

Overall takes O(logn)

Remove ith largest element

```java
/* Remove Ith maximum element from the heap */
public T remove_ith_largest(int kth) {

    if (heap_size == 0) {
        throw new NoSuchElementException("Heap is Empty");
    }

    if (kth > heap_size || kth < 1) {
        throw new NoSuchElementException("Index Out of Bounds");
    }
    /* Remove the first i - 1 elements and then re-insert */
    T[] rest_elements = (T[]) new Comparable[this.MAX_CAPACITY + 1];
    int itr = 0;

    T kth_max = null;

    for (int i = 0; i < kth; i++) {
        T removed_element = this.remove_max_element();
        if (i != kth - 1) {
            rest_elements[itr++] = removed_element;
        } else {
            kth_max = removed_element;
        }
    }

    for (int i = 0; i < itr; i++) {
        this.insert_to_heap(rest_elements[i]);
    }

    return kth_max;
}
```

O(1)

O(n)

O(logn)

Best case = O(1), Worst case = O(nlogn), T(n) = O(nlogn)

Merge Heaps

```java
/* Merge the two heaps */
public void merge_heaps(Heap<T> secondHeap) {
    this.resize_heap(this.MAX_CAPACITY + secondHeap.get_max_size() + 5);
    for (int i = 1; i <= secondHeap.get_curr_size(); i++) {
        this.insert_to_heap(secondHeap.get_ith_element(i));
    }
}

/* To provide user with heap array */
public T[] return_heap() {
    return heap;
}
```

O(n)

O(logn)

Overall time complexity becomes O(n*logn) after deletion and insertion

Heap Iterator class

```java
public class HeapIterator implements Iterator<T> {
    int idx = 1;

    public boolean hasNext() {
        return idx < heap_size;
    }

    public T next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        return heap[idx++];
    }


    public T setValue(T item) {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        T result = heap[heap_size];
        heap[heap_size] = item;
        int newly_inserted_idx = heap_size;
        while ((newly_inserted_idx > 1) && heap[newly_inserted_idx].compareTo(heap[(newly_inserted_idx / 2)]) > 0) {
            swap_elements(newly_inserted_idx, (newly_inserted_idx / 2));
            newly_inserted_idx = (newly_inserted_idx / 2);
        }
        return result;
    }
}
```

O(1)

O(1)

O(1)

Overall - O(logn)

Heapify up takes logn time

Time Complexity Analysis for HeapDistinct Class

Insert Function - O(n)

O(n)

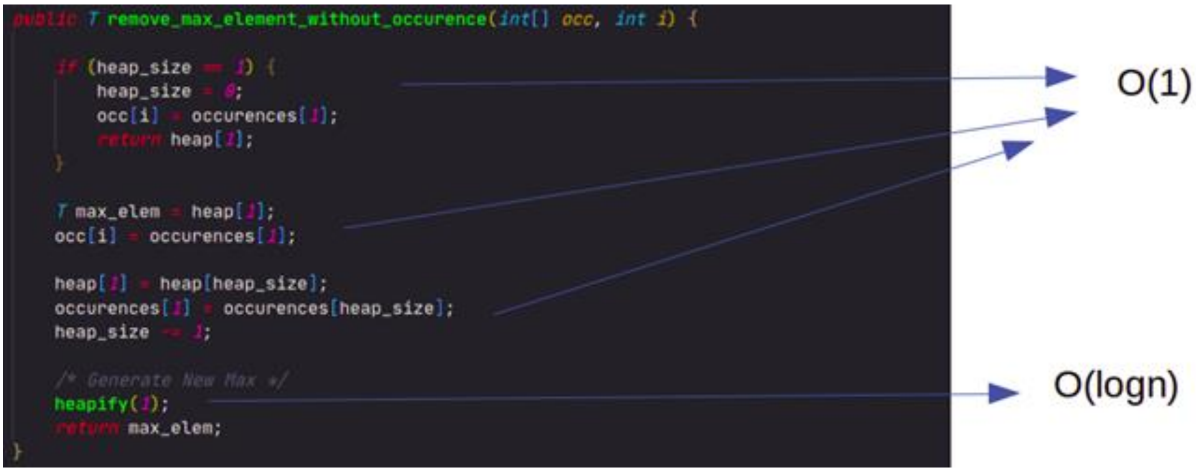
```
public boolean insert_to_heap(T element) {

    for (int i = 1; i <= heap_size; i++) {
        if (heap[i].compareTo(element) == 0) {
            occurences[i]++;
            return true;
        }
    }
    /* Not present */
    if (heap_size == max_heap_size) {
        throw new NoSuchElementException("Heap is Full. Resize it to continue Further.");
    }

    if (heap_size == 0) {
        heap_size++;
        heap[heap_size] = element;
        occurences[heap_size] = 1;
        return true;
    }

    heap_size++;
    heap[heap_size] = element;
    occurences[heap_size] = 1;
    // adjust the heap;

    int newly_inserted_idx = heap_size;
    while ((newly_inserted_idx > 1) && (heap[newly_inserted_idx].compareTo(heap[(newly_inserted_idx / 2)])) > 0) {
        swap_elements(newly_inserted_idx, (newly_inserted_idx / 2));
        newly_inserted_idx = (newly_inserted_idx / 2);
    }
    return true;
```

O(1)

O(logn)

T(n) = O(n) + O(1)*3 + O(logn) = O(n)

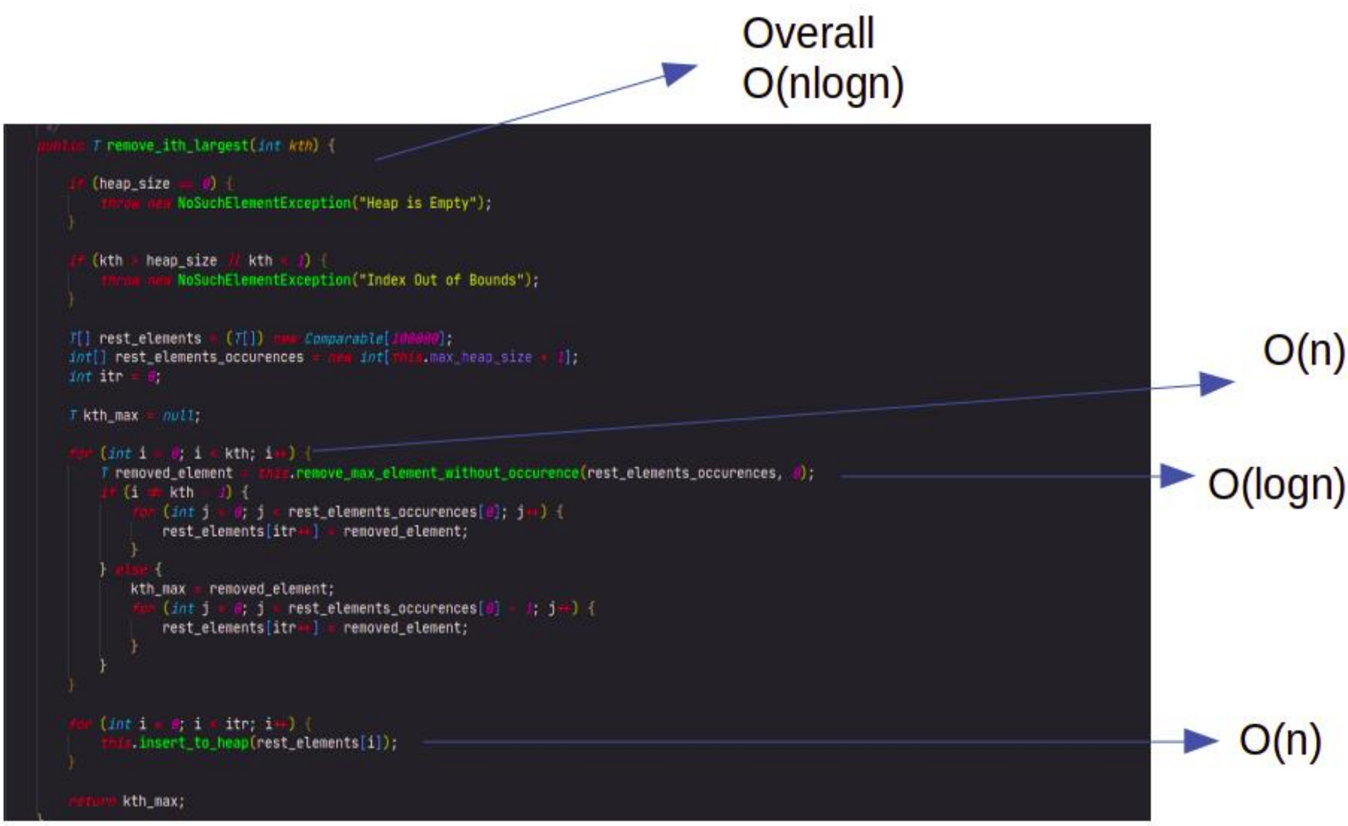
Remove max element without disturbing its occurrences - O(logn)



```java
public T remove_max_element_without_occurence(int[] occ, int i) {

    if (heap_size == 1) {
        heap_size = 0;
        occ[i] = occurences[1];
        return heap[1];
    }

    T max_elem = heap[1];
    occ[i] = occurences[1];

    heap[1] = heap[heap_size];
    occurences[1] = occurences[heap_size];
    heap_size -= 1;

    /* Generate New Max */
    heapify(1);
    return max_elem;
}
```


T(n) = O(logn)

Remove ith largest from Distinct Heap - O(n*logn)



```java
public T remove_ith_largest(int kth) {

    if (heap_size == 0) {
        throw new NoSuchElementException("Heap is Empty");
    }

    if (kth > heap_size || kth < 1) {
        throw new NoSuchElementException("Index Out of Bounds");
    }

    T[] rest_elements = (T[]) new Comparable[100000];
    int[] rest_elements_occurences = new int[this.max_heap_size + 1];
    int itr = 0;

    T kth_max = null;

    for (int i = 0; i < kth; i++) {
        T removed_element = this.remove_max_element_without_occurence(rest_elements_occurences, 0);
        if (i != kth - 1) {
            for (int j = 0; j < rest_elements_occurences[0]; j++) {
                rest_elements[itr++] = removed_element;
            }
        } else {
            kth_max = removed_element;
            for (int j = 0; j < rest_elements_occurences[0] - 1; j++) {
                rest_elements[itr++] = removed_element;
            }
        }
    }

    for (int i = 0; i < itr; i++) {
        this.insert_to_heap(rest_elements[i]);
    }

    return kth_max;
}
```


T(n) = O(n)*O(logn) + O(n

O((n+m)*log(n + m))

```java
/*
 * Merges the second heap passed as a parameter to the first one by resizing the
 * first one.
 */
public void merge_heaps(HeapDistinct<T> secondHeap) {

    this.resize_heap(this.max_heap_size + secondHeap.get_max_size() + 5);
    for (int i = 1; i <= secondHeap.get_curr_size(); i++) {
```

O(m)

```java
        int y = secondHeap.get_ith_elements_occurence(i);
        for (int j = 0; j < y; j++) {
            this.insert_to_heap(secondHeap.get_ith_element(i));
```

O(log(m+n))

```java
        }
    }
}

/*
 * Prints the heap elements with there frequency:
 */
public void print() {
    System.out.println("Element:Frequency");
    for (int i = 1; i <= heap_size; i++) {
```

O(n)

```java
        System.out.print(heap[i] + ":" + occurences[i] + " ");
    }
    System.out.println();
}

/*
 * Replaces the current heap with the new heap parameterized to it by adjusting
 * sizes and elements
 */
public void replace_heap(HeapDistinct<T> secondHeap) {
    this.max_heap_size = secondHeap.max_heap_size;
    this.heap_size = secondHeap.heap_size;
    heap = Arrays.copyOf(secondHeap.heap, this.max_heap_size + 1);
```

O((m+n))

```java
    occurences = Arrays.copyOf(secondHeap.occurences, this.max_heap_size + 1);
    return;
}
```

Time Complexity Analysis for BSTHeapTree Class

N = number of nodes in BST
n = number of nodes in heap [0,7]

```java
public static class bst_node<T extends Comparable<T>> {
    protected bst_node left;
    protected bst_node right;

    private HeapDistinct<T> data;

    public bst_node() {
        left = right = null;
        data = new HeapDistinct<T>(7);
    }
}
```

O(n)

Node Class representing the BST tree node with left and right child

Add method :

```java
public BSTHeapTree() {
    super(7);
    root = null;
}

/*
 * Add element to the heap . If already there then increases the occurences
 */
public int add(T element) {

    if (root == null) {
        root = new bst_node();
    }

    bst_node temp = root;

    while (true) {
        int idx[] = new int[2];
        boolean in_this_node = temp.data.search_element(element, idx);

        if (in_this_node) {
            temp.data.insert_to_heap(element);
            return temp.data.get_ith_elements_occurence(idx[0]);
        }
```

Constructor O(1)

O(1)

O(n)

O(n) as we check already present too

```java
        /* If not Found in Current Node */

        int Hsize = temp.data.get_curr_size();
        int Msize = temp.data.get_max_size();

        if (Hsize >= 0 && Hsize < 7) {
            temp.data.insert_to_heap(element);
            return 1;
        }

        if (Hsize == Msize) {

            if ((temp.data.get_ith_element(1).compareTo(element)) > 0) {

                if (temp.left != null) {
                    temp = temp.left;
                } else {
                    temp.left = new bst_node();
                    temp.left.data.insert_to_heap(element);
                    return 1;
                }
            } else if ((temp.data.get_ith_element(1).compareTo(element)) < 0) {
                if (temp.right != null) {
                    temp = temp.right;
                } else {
                    temp.right = new bst_node();
                    temp.right.data.insert_to_heap(element);
                    return 1;
                }

            }
        }
```

O(n) as we check already present too

Overall -
O(logN*O(n))

Here n = 7
So O(logN)

Find in BST :

```java
public int find(T element) {

    bst_node temp = root;

    if (temp == null) {
        throw new NoSuchElementException("Tree Is Empty. Root is Null.");
    }

    while (temp != null) {
        int idx[] = new int[2];
        boolean in_this_node = temp.data.search_element(element, idx);

        if (in_this_node) {
            return temp.data.get_ith_elements_occurence(idx[0]);
        }

        /* Not Present in current */

        if (temp.data.get_ith_element(1).compareTo(element) > 0) {
            temp = temp.left;
        } else {
            temp = temp.right;
        }
    }
```

O(n)

O(1)

Overall -
O(logN*O(n))

Here n = 7
So O(logN)

Find Mode

```java
/*
 * Helper function to mode in the BST which element has maximum frequency by
 * post order traversal of BST and checking heap at everynode.
 */
private void helper_find_mode(bst_node node, Comparable[] element, int[] mode) {

    if (node == null) {
        return;
    }

    helper_find_mode(node.left, element, mode);
    helper_find_mode(node.right, element, mode);

    /* find mode for each node in BST */

    int Hsize = node.data.get_curr_size();

    for (int i = 1; i <= Hsize; i++) {

        if (mode[1] < node.data.get_ith_elements_occurence(i)) {

            element[0] = (T) (node.data.get_ith_element(i));
            mode[1] = node.data.get_ith_elements_occurence(i);

        }
    }

}
```

O(N) time for postorder traveral

Overall - O(N*n)

O(1)

```java
/*
 * Function to find the mode in BST.
 */
public int find_mode() {
    bst_node temp = root;
    int mode[] = new int[2];
    Comparable[] Tmode = new Comparable[2];
    helper_find_mode(temp, Tmode, mode);

    return mode[1];
}
```

O(N*n)

O(1)

T(n) of find_mode() = O(N*n)

Remove from BST

Helper Functions ::

1. ***Insert_node inserts a node already containing the heap to BST.***

Overall - O(logN * n)

```java
/*
 * Helper function to insert into the bst the Heap itself which is used in
 * remove method to maintain the bst satisying the conditions. It positions that
 * heap to proper place in BST.
 */

private bst_node helper_insert(bst_node root, HeapDistinct<T> node) {
    if (root == null) {
        root = new bst_node();
        root.data.replace_heap(node);
        return root;
    }

    if (root.data.get_ith_element(1).compareTo(node.get_ith_element(1)) > 0) {
        root.left = helper_insert(root.left, node);
    } else if (root.data.get_ith_element(1).compareTo(node.get_ith_element(1)) < 0)
        root.right = helper_insert(root.right, node);

    return root;
}

/*
 * Insert function to insert a heap in the BST.
 */
private void insert_tree_node(bst_node root, HeapDistinct<T> curr_node) {
    if (curr_node.get_curr_size() == 0) {
        return;
    }
    root = helper_insert(root, curr_node);
}
```

O(n)

Recurive calls

O(logN*n)

2. Delete node of BST

```
private HeapDistinct<T> minnodeRST(bst_node root) {
    HeapDistinct<T> min_heap = root.data;

    while (root.left != null) {
        min_heap = root.data;
        root = root.left;
    }
    return min_heap;
}
```

Finds min node in Right Subtree..

O(logN) worse

```
/*
 * Function to delete a whole node in the BST with possibly heap in it used when
 * the maximum element is removed from a BST node and the new maximum doesn't
 * satisfies the BST conditions
 */
private bst_node deleteNode(bst_node root, HeapDistinct<T> heap, bst_node[] to_remove) {
    if (root == null)
        return root;

    if (root.data.get_ith_element(1).compareTo(heap.get_ith_element(1)) > 0)
        root.left = deleteNode(root.left, heap, to_remove);
    else if (root.data.get_ith_element(1).compareTo(heap.get_ith_element(1)) < 0)
        root.right = deleteNode(root.right, heap, to_remove);

    else {

        if (root.left == null)
            return root.right;
        else if (root.right == null)
            return root.left;

        root.data.replace_heap(minnodeRST(root.right));

        root.right = deleteNode(root.right, root.data, to_remove);
    }

    return root;

}
```

Average – O(logN)

Worse - O(N)

3. custom_remove *removes occurrences and if the BST conditions are met after removing else hands over to deleteNode function..*

```
*
    * Function to remove the Element if its occurrences are greater
than one or if
    * it is not the maximum in current heap or after removal the new
maximum
    * satisfies BST criteria.
    */
   private bst_node custom_remove(bst_node root, T element, int occ[],
bst_node[] to_remove, HeapDistinct<T> h) {

       if (root == null) {
           return root;
       }

       int Hsize = root.data.get_curr_size();
       Comparable[] ele = new Comparable[Hsize];
       int idx[] = new int[2];
```

```
        int itr = 0;
        boolean yes_here = root.data.search_element(element, idx);

        for (int i = 1; i <= Hsize; i++) {
            ele[itr++] = root.data.get_ith_element(i);
        }

        Arrays.sort(ele);    /********** O(nlogn)  ************/
        /* Element is here but not a Max Element - Simply Delete */
        if (yes_here) {
            if (idx[0] != 1) {

                int pos = -1;
                for (int i = itr - 1; i >= 0; i--) {
                    if (ele[i].compareTo(element) == 0) {
                        pos = itr - i;
                        break;
                    }
                }
                root.data.remove_ith_largest(pos); O(n*logn)

                boolean is_there = root.data.search_element(element,
idx);
                if (is_there) {
                    occ[0] =
root.data.get_ith_elements_occurence(idx[0]);
                } else {
                    occ[0] = 0;
                }

                return root;

            } else {
                if (root.data.get_ith_elements_occurence(1) > 1) {
                    root.data.remove_ith_largest(1);

                    System.out.println();
                    occ[0] = root.data.get_ith_elements_occurence(1);
                    return root;
                } else {

                    /* Remove this whole node and insert the heap after
*/
```

```
                    if (((root.left != null) && (itr > 1)
                            &&
(root.left.data.get_ith_element(1).compareTo(ele[itr - 2]) < 0))
                            && ((root.right != null) && (itr > 1)
                                    &&
(root.right.data.get_ith_element(1).compareTo(ele[itr - 2]) > 0))) {
                        root.data.remove_ith_largest(1);
                        occ[0] = 0;
                        return root;
                    } else {
                        /* Delete this whole node and insert again */
                        occ[0] = -2;
                        h.replace_heap(root.data);
                        return root;
                        /* using delete node */
                    }

                }
            }
        } else {
            if (root.data.get_ith_element(1).compareTo(element) > 0) {
                root.left = custom_remove(root.left, element, occ,
to_remove, h);
            } else {
                root.right = custom_remove(root.right, element, occ,
to_remove, h);
            }
        }

        return root;
    }
```

This can take LogN to find the BSTNode and to remove the heapnode nlogn

O(logN*n*logn)

*Remove method   O(logN * n * logn)*

```java
public int remove(T element) {

    if (root == null) {
        System.out.println("Binary Search Tree is Empty.");
        return -1;
    }

    int[] occ_after_removal = new int[2];
    bst_node[] removed_node = new bst_node[2];
    HeapDistinct<T> h = new HeapDistinct<T>(7);
    removed_node[0] = removed_node[1] = null;

    root = custom_remove(root, element, occ_after_removal, removed_node, h);

    if (occ_after_removal[0] != -2) {
        return occ_after_removal[0];
    }

    if (occ_after_removal[0] == -1) {
        System.out.println(element + " Not Found in BST");
        return -1;
    }

    root = deleteNode(root, h, removed_node);

    h.remove_ith_largest(1);

    insert_tree_node(root, h);

    return 0;
}
```

O(logN * nlogn)

O(N) worse

O(N*n) worse

O(nlogn) worse