**PriorityQueues**

**Min Priority Queue**

public class Element<T> {

T value;

int priority;

public Element(T value, int priority) {

this.value = value;

this.priority = priority;

}

}

public class PriorityQueueMinHeap<T> {

private final ArrayList<Element<T>> heap;

public PriorityQueueMinHeap() {

heap = new ArrayList<>();

}

// 1st method

public void insertMin(T value, int priority) {

Element<T> e = new Element<>(value, priority);

heap.add(e);

int childIndex = heap.size() - 1;

int parentIndex = (childIndex - 1) / 2;

while (childIndex > 0) {

if (heap.get(childIndex).priority < heap.get(parentIndex).priority) {

Element<T> temp = heap.get(childIndex);

heap.set(childIndex, heap.get(parentIndex));

heap.set(parentIndex, temp);

childIndex = parentIndex;

parentIndex = (childIndex - 1) / 2;

}

else

return;

}

}

// 2nd method

public T getMin() throws PriorityQueueException {

if (isEmpty())

throw new PriorityQueueException();

return heap.getFirst().value;

}

// 3rd method

public int size() {

return heap.size();

}

// 4th method

public boolean isEmpty() {

return size() == 0;

}

// 5th method

public T removeMin() throws PriorityQueueException {

if (isEmpty())

throw new PriorityQueueException();

Element<T> remove = heap.getFirst();

T ans = remove.value;

heap.set(0, heap.getLast());

heap.removeLast();

int parentIndex = 0;

int leftChildIndex = 1;

int rightChildIndex = 2;

int minIndex = parentIndex;

while (leftChildIndex < heap.size()) {

if (heap.get(leftChildIndex).priority < heap.get(minIndex).priority) {

minIndex = leftChildIndex;

}

if (rightChildIndex < heap.size() &&

heap.get(rightChildIndex).priority < heap.get(minIndex).priority) {

minIndex = rightChildIndex;

}

if (minIndex == parentIndex)

break;

Element<T> temp = heap.get(minIndex);

heap.set(minIndex, heap.get(parentIndex));

heap.set(parentIndex, temp);

parentIndex = minIndex;

leftChildIndex = 2 \* parentIndex + 1;

rightChildIndex = 2 \* parentIndex + 2;

}

return ans;

}

}

public class PriorityQueueMaxHeap<T> {

public ArrayList<Element<T>> heap;

public PriorityQueueMaxHeap() {

heap = new ArrayList<>();

}

// 1st method

public void insertMax(T value, int priority) {

Element<T> element = new Element<>(value, priority);

heap.add(element);

int childIndex = heap.size() - 1;

int parentIndex = (childIndex - 1) / 2;

while (childIndex >= 0) {

if (heap.get(childIndex).priority > heap.get(parentIndex).priority) {

Element<T> temp = heap.get(childIndex);

heap.set(childIndex, heap.get(parentIndex));

heap.set(parentIndex, temp);

childIndex = parentIndex;

parentIndex = (childIndex - 1) / 2;

}

else

return;

}

}

// 2nd method

public int size() {

return heap.size();

}

// 3rd method

public boolean isEmpty() {

return size() == 0;

}

// 4th method

public T getMax() {

if (isEmpty())

return null;

return heap.getFirst().value;

}

// 5th method

public T removeMax() {

if (isEmpty())

return null;

Element<T> removeMax = heap.getFirst();

T elementValue = removeMax.value;

// placing last element at start

Element<T> lastElement = heap.getLast();

heap.set(0, lastElement);

heap.removeLast();

int parentIndex = 0;

int leftChildIndex = 1;

int rightChildIndex = 2;

int indexOfMaxElement; //let's assume

// finding the element which has max priority

while (leftChildIndex < heap.size()) {

indexOfMaxElement = parentIndex;

if (heap.get(leftChildIndex).priority > heap.get(indexOfMaxElement).priority) {

indexOfMaxElement = leftChildIndex;

}

/\* here we are checking corner case if the right child don’t exist but then also we are checking for that then ArrayOutOfBound exception will occur so we have check for that also.\*/

if (rightChildIndex < heap.size() &&

heap.get(rightChildIndex).priority > heap.get(indexOfMaxElement).priority) {

indexOfMaxElement = rightChildIndex;

}

// No need to swap further if the parent's priority is already maximum

if (indexOfMaxElement == parentIndex) {

break;

}

// swapping of element

Element<T> temp = heap.get(indexOfMaxElement);

heap.set(indexOfMaxElement, heap.get(parentIndex));

heap.set(parentIndex, temp);

// updating the index

parentIndex = indexOfMaxElement;

leftChildIndex = 2 \* parentIndex + 1;

rightChildIndex = 2 \* parentIndex + 2;

}

return elementValue;

}

}

// Inplace Heap sort algorithm

public class InplaceHeapSort {

public static void main(String[] args) {

int[] arr = {4, 7, 3, 2, 8, 9, 6, 1};

heapSort(arr);

for (int i : arr)

System.out.print(i + " ");

}

private static void heapSort(int[] arr) {

// build the heap

int n = arr.length;

for (int i = (n / 2) - 1; i >= 0; i--) {

// downHeapify(arr, i, n); // down Heap

upHeapify(arr, i, n);

}

// removing elements from start one by one and put them at respective last position

for (int i = n - 1; i >= 0; i--) {

int temp = arr[i];

arr[i] = arr[0];

arr[0] = temp;

// downHeapify(arr, 0, i); // down Heap

upHeapify(arr, 0, i);

}

}

private static void downHeapify(int[] arr, int i, int n) {

int parentIndex = i;

int leftChildIndex = 2 \* parentIndex + 1;

int rightChildIndex = 2 \* parentIndex + 2;

while (leftChildIndex < n) {

int minChildIndex = parentIndex;

if (arr[leftChildIndex] < arr[minChildIndex])

minChildIndex = leftChildIndex;

if (rightChildIndex < n && arr[rightChildIndex] < arr[minChildIndex])

minChildIndex = rightChildIndex;

if (minChildIndex == parentIndex)

return;

int temp = arr[parentIndex];

arr[parentIndex] = arr[minChildIndex];

arr[minChildIndex] = temp;

parentIndex = minChildIndex;

leftChildIndex = 2 \* parentIndex + 1;

rightChildIndex = 2 \* parentIndex + 2;

}

}

private static void upHeapify(int[] arr, int i, int n) {

int parentIndex = i;

int leftChildIndex = 2 \* parentIndex + 1;

int rightChildIndex = 2 \* parentIndex + 2;

while (leftChildIndex < n) {

int minChildIndex = parentIndex;

if (arr[leftChildIndex] > arr[minChildIndex])

minChildIndex = leftChildIndex;

if (rightChildIndex < n && arr[rightChildIndex] > arr[minChildIndex])

minChildIndex = rightChildIndex;

if (minChildIndex == parentIndex)

return;

int temp = arr[parentIndex];

arr[parentIndex] = arr[minChildIndex];

arr[minChildIndex] = temp;

parentIndex = minChildIndex;

leftChildIndex = 2 \* parentIndex + 1;

rightChildIndex = 2 \* parentIndex + 2;

}

}

}

public class KLargestAndSmallest {

public static void printKLargest(int[] arr, int k) {

PriorityQueue<Integer> queue = new PriorityQueue<>();

for (int i = 0; i < k; i++) {

queue.add(arr[i]);

}

for (int i = k; i < arr.length; i++) {

if (queue.peek() < arr[i]) {

queue.poll();

queue.add(arr[i]);

}

}

while (!queue.isEmpty()) {

System.out.print(queue.poll() + " ");

}

}

public static void printKSmallest(int[] arr, int k) {

PriorityQueue<Integer> queue = new PriorityQueue<>();

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

queue.add(arr[i]);

}

int i = 0;

while (i++ < k) {

System.out.print(queue.poll() + " ");

}

}

}