**1. Explanation:**

Have a look at the code given below:

ConsoleJava

import java.util.\*;

import java.io.\*;

public class Main {

public static void main(String[] args)

{

int [] arr = {10, 2, 17, 3, 18, 9, 22};

PriorityQueue pq = new PriorityQueue<>();

for (int val : arr)

{

pq.add(val);

}

while (pq.size() != 0)

{

int val = pq.peek();

System.out.println(val);

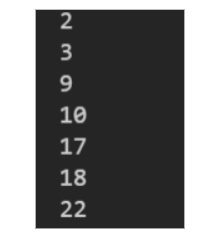
pq.remove();

}

}

}

In the code above, we have created a priority queue and we have inserted the elements into it from an array. When we remove the elements from this priority queue, we will get all the elements in an ascending order i.e. the elements get removed in the order of their priority (smallest element having the highest priority). (Have a look at the output given below)



The same behavior can be shown by the priority queue that we have implemented and created ourselves using heap. You may refer to the PRIORITY QUEUE USING HEAP video to create your own priority queue if you haven't till now. Now the question is,

The same behavior can be shown by the priority queue that we have implemented and created ourselves using heap. You may refer to the PRIORITY QUEUE USING HEAP video to create your own priority queue if you haven't till now. Now the question is

For instance, we have created a student class below and we try to add them to the priority queue. They will get added (no doubts about that). But, when we try to remove them, the priority is not getting decided and we get an error:

**2. Approach :**

what if the priority of the elements that we insert into the priority queue cannot be decided by the priority queue?

Let's code this:

ConsoleJava

import java.util.\*;

import java.io.\*;

public class Main {

static class Student {

int rno;

int ht;

int weight;

Student(int rno, int ht, int weight)

{

this.rno = rno;

this.ht = ht;

this.weight = weight;

}

}

public static void main(String[] args)

{

int [] arr = {10, 2, 17, 3, 18, 9, 22};

PriorityQueue < Student> pq = new PriorityQueue < >();

pq.add(new Student(1, 180, 82));

pq.add(new Student(2, 170, 81));

pq.add(new Student(3, 200, 85));

pq.add(new Student(4, 190, 87));

pq.add(new Student(5, 185, 70));

while (pq.size() != 0)

{

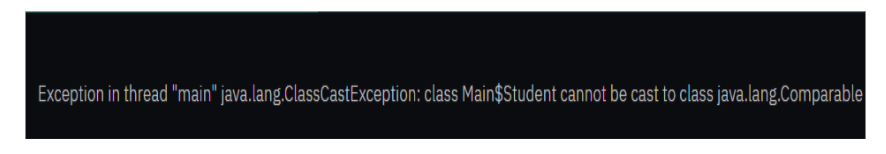
System.out.println(pq.peek());

pq.remove();

}

}

}



This is because earlier we were comparing integers and the priority queue was able to compare two integers. Now, we have given our own class and the priority queue is not able to decide the priority.

If you remember (from the create priority queue using heap video) that we used to compare the parent with its child and the value of higher priority i.e. lower in value used to move upwards i.e. we use upheapify for adding the value into the priority queue. Now, the problem is that the parent student object and the child student object cannot be compared as they are not simple integer values. So, how can we get rid of this problem and compare two student values?

**The priority queue in Java wants that whatever we are passing into the priority queue, that must implement an interface called Comparable.:**

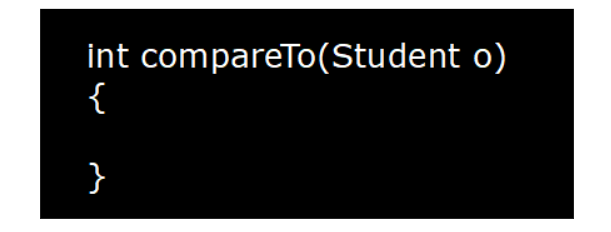
**Now the question is what is an interface?**

**3. Interface:**

**Interface is just like a class. The difference between an interface and a class is that an interface will not have the body of the functions inside it. It will have only the signature of the functions. (signature means the function name, parameter and return type).**

An interface can also be called as the contract for a class. This is because when a class has to implement an interface, it has to provide all the functions (bodies) with the same signature as that in the interface to it. It is just like to use an interface, class honors its contract.

So, the interface Comparable has only one function called compareTo(parameter). This function has an integer return type that is:



But, if this is a function to compare two entities, why does it have only one parameter? Actually, there are two parameters. One is the caller object for this

function which we call as the "this" object and the other object is passed as a parameter to the function.

**Now, what does this function do?**

**This function compares the value of "this" and the other object (passed as a parameter). If the value of "this" is greater than the other object, this function returns a positive value. If they are equal, it returns zero, else it returns a negative value. But, it is able to compare only integer values. So, we have to modify this function inside the class in which we implement the Comparable interface. This can be seen in the code shown below:**

ConsoleJava

import java.util.\*;

import java.io.\*;

public class Main {

static class Student implements Comparable< Student> {

int rno;

int ht;

int weight;

Student(int rno, int ht, int weight)

{

this.rno = rno;

this.ht = ht;

this.weight = weight;

}

public int compareTo(Student o)

{

return this.rno - o.rno;

}

public String toString()

{

return "rno= " + rno + "height= " + ht + "weight= " + weight;

}

}

public static void main(String[] args)

{

PriorityQueue< Student> pq = new PriorityQueue<>();

pq.add(new Student(1, 180, 82));

pq.add(new Student(2, 170, 81));

pq.add(new Student(3, 200, 85));

pq.add(new Student(4, 190, 87));

pq.add(new Student(5, 185, 70));

while (pq.size() != 0)

{

System.out.println(pq.peek());

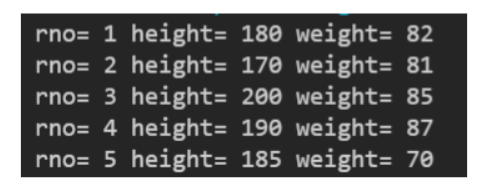
pq.remove();

}

}

}

**We have used the toString() function to print the object for the class student as we get the address of the objects if we try to print the objects directly. So, the**



What if we want to reverse the priority? Well, it is simple. We compare this with the other object and if this value minus the other object value is negative, we send it upwards in the heap. If we return the opposite of what we are returning i.e. instead of returning this roll number minus other object roll number, if we return other object roll number minus this roll number, the priority will be reversed. Why? Think!!!!

We request you to watch the solution video to understand how we can reverse the priority by interchanging the position of operands in the compareTo function.

ConsoleJava

import java.util.\*;

import java.io.\*;

public class Main {

static class Student implements Comparable< Student> {

int rno;

int ht;

int weight;

Student(int rno, int ht, int weight)

{

this.rno = rno;

this.ht = ht;

this.weight = weight;

}

public int compareTo(Student o)

{

return o.rno - this.rno;

}

public String toString()

{

return "rno= " + rno + " height= " + ht + " weight= " + weight;

}

}

public static void main(String[] args)

{

PriorityQueue< Student> pq = new PriorityQueue<>();

pq.add(new Student(1, 180, 82));

pq.add(new Student(2, 170, 81));

pq.add(new Student(3, 200, 85));

pq.add(new Student(4, 190, 87));

pq.add(new Student(5, 185, 70));

while (pq.size() != 0)

{

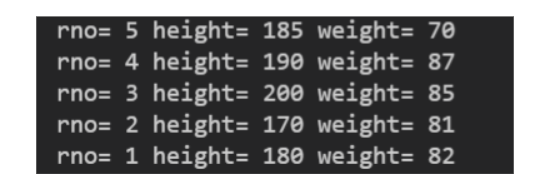
System.out.println(pq.peek());

pq.remove();

}

}

}



Now we want to implement this feature to our custom created priority queue. We will make our priority queue generic i.e. it will be able to add any kind of data that we like to and not only integers.

Also, we will now not compare by the values, rather we will type cast the objects into comparable and then compare them using the toCompare function.

Dear reader, we request you to watch the solution video to understand the above mentioned concept and code completely as it is difficult to explain the changes made to the code here. Also, we again request you that if you have not watched the CREATE PRIORITY QUEUE USING HEAP VIDEO, you should watch it first as the code for the same is used and modifications are made to it. For that you must have that code by-hearted till now i.e. you must have practiced it at least once without looking at the code from the video.

**4. Code:**

ConsoleJava

import java.io.\*;

import java.util.\*;

public class Main

{

public static class PriorityQueue < T>

{

ArrayList< T> data;

public PriorityQueue()

{

data = new ArrayList< T >();

}

private boolean isSmaller(int i, int j)

{

Comparable ith = (Comparable) data.get(i);

Comparable jth = (Comparable) data.get(j);

if (ith.compareTo(jth) < 0)

{

return true;

}

else {

return false;

}

}

public void add(T val)

{

data.add(val);

upheapify(data.size() - 1);

}

private void upheapify(int i)

{

if (i = = 0) {

return;

}

int pi = (i - 1) / 2;

if (isSmaller(i, pi) = = true)

{

swap(i, pi);

upheapify(pi);

}

}

private void swap(int i, int j)

{

T ith = data.get(i);

T jth = data.get(j);

data.set(i, jth);

data.set(j, ith);

}

public T remove()

{

if (this.size() = = 0)

{

System.out.println(" Underflow " ) ;

return null;

}

swap(0, data.size() - 1);

T val = data.remove(data.size() - 1);

downheapify(0);

return val;

}

private void downheapify(int i)

{

int mini = i;

int li = 2 \* i + 1;

if (li < data.size() && isSmaller(li, mni) = = true) {

mini = li;

}

int ri = 2 \* i + 2;

if (ri < data.size() && isSmaller(ri, mini) = = true)

{

mini = ri;

}

if (mini != i)

{

swap(i, mini);

downheapify(mini);

}

}

public T peek()

{

if (this.size() = = 0)

{

System.out.println( " Underflow" ) ;

return null;

}

return data.get(0);

}

public int size()

{

return data.size();

}

}

static class Student implements Comparable < Student>

{

int rno;

int ht;

int weight;

Student(int rno, int ht, int weight)

{

this.rno = rno;

this.ht = ht;

this.weight = weight;

}

public int compareTo(Student o)

{

return o.rno - this.rno;

}

public String toString()

{

return "rno= " + rno + " height= " + ht + " weight= " + weight;

}

}

public static void main(String[] args)

{

PriorityQueue< Student> pq = new PriorityQueue<>();

pq.add(new Student(1, 180, 82));

pq.add(new Student(2, 170, 81));

pq.add(new Student(3, 200, 85));

pq.add(new Student(4, 190, 87));

pq.add(new Student(5, 185, 70));

while (pq.size() != 0)

{

System.out.println(pq.peek());

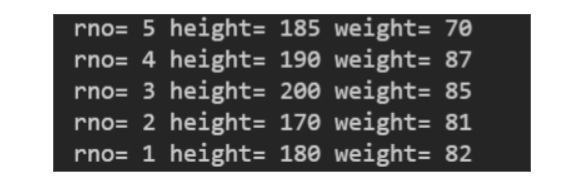
pq.remove();

}

}

}

The output for the above code will be:



There is another method by which we can implement the things we learnt above i.e. Comparator.

Comparator is also an interface. This is a little different from the Comparable. Have a look at the code shown below:

ConsoleJava

import java.util.\*;

import java.io.\*;

public class Main {

static class Student implements Comparable< Student> {

int rno;

int ht;

int weight;

Student(int rno, int ht, int weight)

{

this.rno = rno;

this.ht = ht;

this.weight = weight;

}

public int compareTo(Student o)

{

return o.rno - this.rno;

}

public String toString()

{

return "rno= " + rno + " height= " + ht + " weight= " + weight;

}

}

static class StudentHtComparator implements Comparator< Student> {

public int compare(Student s1, Student s2)

{

return s1.ht - s2.ht;

}

}

static class StudentWtComparator implements Comparator< Student>

{

public int compare(Student s1, Student s2)

{

return s1.weight - s2.weight;

}

}

public static void main(String[] args)

{

PriorityQueue< Student> pq = new PriorityQueue<>(new StudentHtComparator());

pq.add(new Student(1, 180, 82));

pq.add(new Student(2, 170, 81));

pq.add(new Student(3, 200, 85));

pq.add(new Student(4, 190, 87));

pq.add(new Student(5, 185, 70));

while (pq.size() != 0)

{

System.out.println(pq.peek());

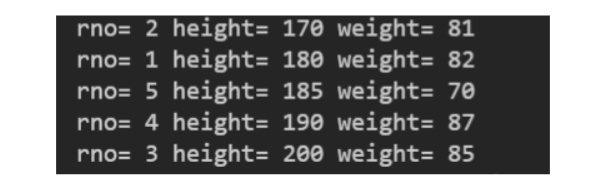
pq.remove();

}

}

}

As you can see, unlike Comparable interface, we are implementing the comparator interface in a different class and not in the Student class itself. Also, we can clearly see that the Comparable has only one parameter because the first object is the "this" object. Since the comparator is not in the Student class rather in a different class, it will have both the parameters as it cannot be called by an object of Student class. In the above example, we have passed the Comparator which sets the priority on the basis of height of the students. So, we should get the student of least height first when we remove from this priority queue. Let's see the output of the above code:



As you can see, the students are getting removed in the priority of their heights i.e. the student with the least height got removed first and so on. Try to pass the weight comparator and get the output on the basis of weights of the students yourself. Also try to reverse the priority in both the comparators.

**Note :***If we do not pass any comparator to the priority queue then it will work as per the Comparable interface.*

Now we want to add this functionality to our custom created priority queue. So, we will make another constructor with a Comparator as a parameter and we will also make a Comparator data member. Now, if the comparator data member is null i.e. comparator was not passed in the priority queue while creating it then, we will make the priority queue work on the basis of the Comparable else it will work as per the Comparator passed.

ConsoleJava

import java.io.\*;

import java.util.\*;

public class Main {

public static class PriorityQueue {

ArrayList data;

Comparator comp;

public PriorityQueue() {

data = new ArrayList < >();

comp = null;

}

public PriorityQueue(Comparator comp) {

data = new ArrayList < >();

this.comp = comp;

}

private boolean isSmaller(int i, int j)

{

if (comp == null) {

Comparable ith = (Comparable) data.get(i);

Comparable jth = (Comparable) data.get(j);

if (ith.compareTo(jth) < 0)

{

return true;

}

else {

return false;

}

}

else

{

T ith = data.get(i);

T jth = data.get(j);

if (comp.compare(ith, jth) < 0)

{

return true;

}

else

{

return false;

}

}

}

public void add(T val) {

data.add(val);

upheapify(data.size() - 1);

}

private void upheapify(int i) {

if (i = = 0) {

return;

}

int pi = (i - 1) / 2;

if (isSmaller(i, pi) = = true) {

swap(i, pi);

upheapify(pi);

}

}

private void swap(int i, int j) {

T ith = data.get(i);

T jth = data.get(j);

data.set(i, jth);

data.set(j, ith);

}

public T remove() {

if (this.size() = = 0) {

System.out.println(" Underflow " );

return null;

}

swap(0, data.size() - 1);

T val = data.remove(data.size() - 1);

downheapify(0);

return val;

}

private void downheapify(int i) {

int mini = i;

int li = 2 \* i + 1;

if (li < data.size() && isSmaller(li, mini) = = true) {

mini = li;

}

int ri = 2 \* i + 2;

if (ri < data.size() && isSmaller(ri, mini) = = true) {

mini = ri;

}

if (mini != i ) {

swap(i, mini);

downheapify(mini);

}

}

public T peek() {

if (this.size() = = 0 )

{

System.out.println( " Underflow " );

return null;

}

return data.get(0);

}

public int size() {

return data.size();

}

}

static class Student implements Comparable< Student> {

int rno;

int ht;

int weight;

Student(int rno, int ht, int weight)

{

this.rno = rno;

this.ht = ht;

this.weight = weight;

}

public int compareTo(Student o)

{

return o.rno - this.rno;

}

public String toString()

{

return "rno= " + rno + " height= " + ht + " weight= " + weight;

}

}

static class StudentHtComparator implements Comparator< Student> {

public int compare(Student s1, Student s2)

{

return s1.ht - s2.ht;

}

}

static class StudentWtComparator implements Comparator< Student>

{

public int compare(Student s1, Student s2)

{

return s1.weight - s2.weight;

}

}

public static void main(String[] args)

{

PriorityQueue< Student> pq = new PriorityQueue<>(new StudentWtComparator());

pq.add(new Student(1, 180, 82));

pq.add(new Student(2, 170, 81));

pq.add(new Student(3, 200, 85));

pq.add(new Student(4, 190, 87));

pq.add(new Student(5, 185, 70));

while (pq.size() != 0)

{

System.out.println(pq.peek());

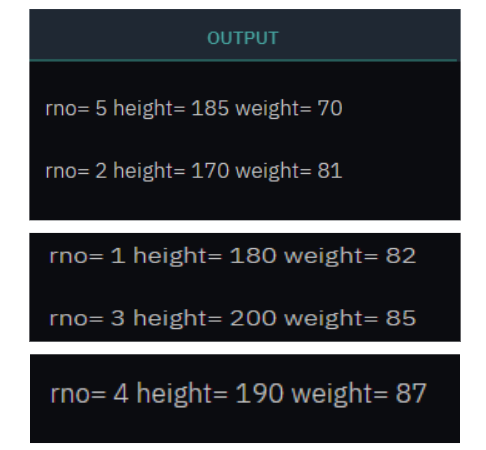
pq.remove();

}

}

}

The output for the above code is:



So, dear reader, we hope that you have understood the comparable and comparator and the difference between them too. With this, we have come to an end of this article as well as the hashmaps and heaps section. If you have any doubts regarding any concept, refer to the complete solution video.