Nama : Naufal Rizq Ramadhan

NIM : 6706213150

1. Penjelasan Koding

**HeapDemo.java**

*package* com.company;  
  
*public class* HeapDemo {  
 *public static void* main(String[] args) {  
 MinHeap q = *new* MinHeap();  
 q.add(*new* WorkOrder(60, "-"));  
 q.add(*new* WorkOrder(5, "-"));  
 q.add(*new* WorkOrder(43, "-"));  
 q.add(*new* WorkOrder(11, "-"));  
 q.add(*new* WorkOrder(12, "-"));  
 q.add(*new* WorkOrder(50, "-"));  
 q.add(*new* WorkOrder(23, "-"));  
 q.add(*new* WorkOrder(8, "-"));  
 q.add(*new* WorkOrder(17, "-"));  
 q.add(*new* WorkOrder(3, "-"));  
 *while* (q.size() > 0)  
 System.out.println(q.remove());  
 }  
}

Fungsi dari kelas ini digunakan sebagai Main, yang dimana berfungsi untuk menerima masukan data ke dalam Array list yang nantinya akan diurutkan dari level prioritas paling kecil sampai paling besar. Dalam class ini juga proses pengurutannya menggunakan method method dari kelas lain yang sudah dibuat

**HeapSorter.java**

*package* com.company;  
  
*public class* HeapSorter {  
 *private int*[] a;  
 */\*\*  
 \* Constructs a heap sorter that sorts a given array.  
 \*  
 \* @param anArray an array of integers  
 \*/  
 public* HeapSorter(*int*[] anArray) {  
 a = anArray;  
 }  
 */\*\*  
 \* Sorts the array managed by this heap sorter.  
 \*/  
 public void* sort() {  
 *int* n = a.length - 1;  
 *for* (*int* i = (n - 1) / 2; i >= 0; i--)  
 fixHeap(i, n);  
 *while* (n > 0) {  
 swap(0, n);  
 n--;  
 fixHeap(0, n);  
 }  
 }  
 */\*\*  
 \* Ensures the heap property for a subtree, provided its  
 \* children already fulfill the heap property.  
 \*  
 \* @param rootIndex the index of the subtree to be fixed  
 \* @param lastIndex the last valid index of the tree that  
 \* contains the subtree to be fixed  
 \*/  
 private void* fixHeap(*int* rootIndex, *int* lastIndex) {  
 *// Remove root  
 int* rootValue = a[rootIndex];  
 *// Promote children while they are larger than the root  
 int* index = rootIndex;  
 *boolean* more = *true*;  
 *while* (more) {  
 *int* childIndex = getLeftChildIndex(index);  
 *if* (childIndex <= lastIndex) {  
 *// Use right child instead if it is larger  
 int* rightChildIndex = getRightChildIndex(index);  
 *if* (rightChildIndex <= lastIndex && a[rightChildIndex] >  
 a[childIndex]) {  
 childIndex = rightChildIndex;  
 }  
 *if* (a[childIndex] > rootValue) {  
 *// Promote child* a[index] = a[childIndex];  
 index = childIndex;  
 } *else* {  
 *// Root value is larger than both children* more = *false*;  
 }  
 } *else* {  
 *// No children* more = *false*;  
 }  
 }  
 *// Store root value in vacant slot* a[index] = rootValue;  
 }  
 */\*\*  
 \* Swaps two entries of the array.  
 \*  
 \* @param i the first position to swap  
 \* @param j the second position to swap  
 \*/  
 private void* swap(*int* i, *int* j) {  
 *int* temp = a[i];  
 a[i] = a[j];  
 a[j] = temp;  
 }  
 */\*\*  
 \* Returns the index of the left child.  
 \*  
 \* @param index the index of a node in this heap  
 \* @return the index of the left child of the given node  
 \*/  
 private static int* getLeftChildIndex(*int* index) {  
 *return* 2 \* index + 1;  
 }  
 */\*\*  
 \* Returns the index of the right child.  
 \*  
 \* @param index the index of a node in this heap  
 \* @return the index of the right child of the given node  
 \*/  
 private static int* getRightChildIndex(*int* index) {  
 *return* 2 \* index + 2;  
 }  
}

Didalam class ini berfungsi untuk proses sorting dari Array MinHeap, yang nanti akan ditentukan berdasarkan dari angka inputannya.

**WorkOrder.java**

*package* com.company;  
  
*public class* WorkOrder *implements* Comparable {  
 *private int* priority;  
 *private* String description;  
 */\*\*  
 \* Constructs a work order with a given priority and description.  
 \*  
 \* @param aPriority the priority of this work order  
 \* @param aDescription the description of this work order  
 \*/  
 public* WorkOrder(*int* aPriority, String aDescription) {  
 priority = aPriority;  
 description = aDescription;  
 }  
 *public* String toString() {  
 *return* "priority=" + priority + ", description=" + description;  
 }  
 *public int* compareTo(Object otherObject) {  
 WorkOrder other = (WorkOrder) otherObject;  
 *if* (priority < other.priority) *return* -1;  
 *if* (priority > other.priority) *return* 1;  
 *return* 0;  
 }  
}

Pada class ini, berfungsi untuk mengcompare string to string dimana yang dicompare adalah inputan level prioritasnya.

**MinHeap.java**

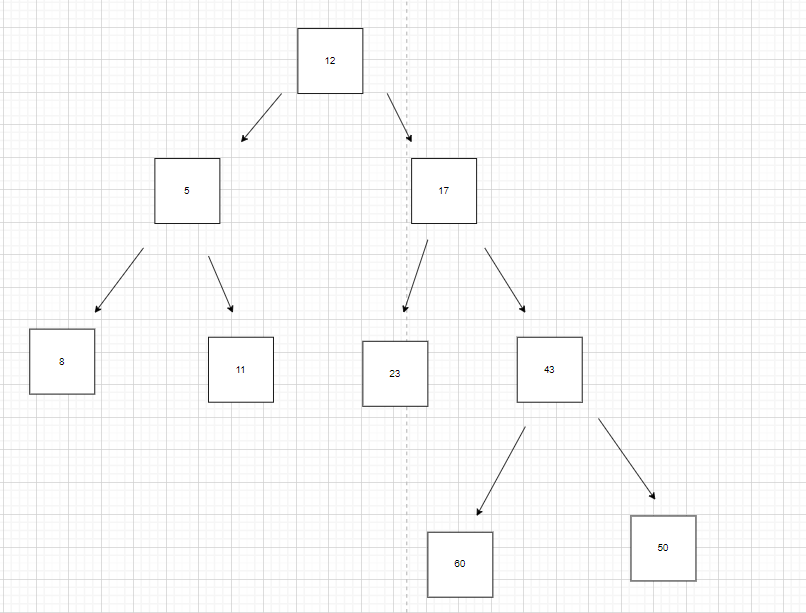
*package* com.company;  
  
*import* java.util.ArrayList;  
  
*public class* MinHeap {  
 *private* ArrayList<Comparable> elements;  
 */\*\*  
 \* Constructs an empty heap.  
 \*/  
 public* MinHeap() {  
 elements = *new* ArrayList<Comparable>();  
 elements.add(*null*);  
 }  
 */\*\*  
 \* Returns the index of the left child.  
 \*  
 \* @param index the index of a node in this heap  
 \* @return the index of the left child of the given node  
 \*/  
 private static int* getLeftChildIndex(*int* index) {  
 *return* 2 \* index;  
 }  
*/\*\*  
 \* Returns the index of the right child.  
 \*  
 \* @param index the index of a node in this heap  
 \* @return the index of the right child of the given node  
 \*/  
private static int* getRightChildIndex(*int* index) {  
 *return* 2 \* index + 1;  
}  
 */\*\*  
 \* Returns the index of the parent.  
 \*  
 \* @param index the index of a node in this heap  
 \* @return the index of the parent of the given node  
 \*/  
 private static int* getParentIndex(*int* index) {  
 *return* index / 2;  
 }  
 */\*\*  
 \* Adds a new element to this heap.  
 \*  
 \* @param newElement the element to add  
 \*/  
 public void* add(Comparable newElement) {  
 *// Add a new leaf* elements.add(*null*);  
 *int* index = elements.size() - 1;  
 *// Demote parents that are larger than the new element  
 while* (index > 1 && getParent(index).compareTo(newElement) > 0) {  
 elements.set(index, getParent(index));  
 index = getParentIndex(index);  
 }  
 *// Store the new element in the vacant slot* elements.set(index, newElement);  
 }  
 */\*\*  
 \* Gets the minimum element stored in this heap.  
 \*  
 \* @return the minimum element  
 \*/  
 public* Comparable peek() {  
 *return* elements.get(1);  
 }  
 */\*\*  
 \* Removes the minimum element from this heap.  
 \*  
 \* @return the minimum element  
 \*/  
 public* Comparable remove() {  
 Comparable minimum = elements.get(1);  
 *// Remove last element  
 int* lastIndex = elements.size() - 1;  
 Comparable last = elements.remove(lastIndex);  
 *if* (lastIndex > 1) {  
 elements.set(1, last);  
 fixHeap();  
 }  
 *return* minimum;  
 }  
 */\*\*  
 \* Turns the tree back into a heap, provided only the root  
 \* node violates the heap condition.  
 \*/  
 private void* fixHeap() {  
 Comparable root = elements.get(1);  
 *int* lastIndex = elements.size() - 1;  
 *// Promote children of removed root while they are smaller than last  
 int* index = 1;  
 *boolean* more = *true*;  
 *while* (more) {  
 *int* childIndex = getLeftChildIndex(index);  
 *if* (childIndex <= lastIndex) {  
 *// Get smaller child  
 // Get left child first* Comparable child = getLeftChild(index);  
 *// Use right child instead if it is smaller  
 if* (getRightChildIndex(index) <= lastIndex  
 && getRightChild(index).compareTo(child) < 0) {  
 childIndex = getRightChildIndex(index);  
 child = getRightChild(index);  
 }  
 *// Check if larger child is smaller than root  
 if* (child.compareTo(root) < 0) {  
 *// Promote child* elements.set(index, child);  
 index = childIndex;  
 } *else* {  
 *// Root is smaller than both children* more = *false*;  
 }  
 } *else* {  
 *// No children* more = *false*;  
 }  
 }  
 *// Store root element in vacant slot* elements.set(index, root);  
 }  
 */\*\*  
 \* Returns the number of elements in this heap.  
 \*/  
 public int* size() {  
 *return* elements.size() - 1;  
 }  
 */\*\*  
 \* Returns the value of the left child.  
 \*  
 \* @param index the index of a node in this heap  
 \* @return the value of the left child of the given node  
 \*/  
 private* Comparable getLeftChild(*int* index) {  
 *return* elements.get(2 \* index);  
 }  
 */\*\*  
 \* Returns the value of the right child.  
 \*  
 \* @param index the index of a node in this heap  
 \* @return the value of the right child of the given node  
 \*/  
 private* Comparable getRightChild(*int* index) {  
 *return* elements.get(2 \* index + 1);  
 }  
 */\*\*  
 \* Returns the value of the parent.  
 \*  
 \* @param index the index of a node in this heap  
 \* @return the value of the parent of the given node  
 \*/  
 private* Comparable getParent(*int* index) {  
 *return* elements.get(index / 2);  
 }  
}

Didalam class ini, berisikan array dengan nama MinHeap lalu di dalamnya juga ada method Comparable, selain itu juga ada method getRightChild, getParent, getLeftChild.

Method comparable ini untuk mengambil data data (element) dari dalam array MinHeap.

Yang pada akhirnya akan membentuk sebuah HeapTree.

Gambarnya adalah sebagai berikut :



Bagaimana pada akhirnya HeapTree ini dapat terbentuk?

Pada awalnya, kita membuat satu node kosong diakhir, setelah itu kita menurunkan parent ke slot kosong tersebut sampai pada akhirnya sampai pada child pertama dari root, dikarenakan child pertama dari root ini kosong maka akan kita insertkan data baru.

Atau juga bisa dengan cara mengosongkan rootnya lalu menaikan node terakhir menjadi root, dikarenakan rootnya merupakan dari node terakhir tadi berarti angkanya besar. Maka dari itu nanti dari root itu akan di swap lagi hingga menjadi node terakhir lagi (ex : Pengambilan node terakhir semisal dari child yang sebelah kiri bisa berakhir menjadi child disebelah kanan, begitupun sebaliknya).