Jinjinqin 442801

Handle class:

```
public class Handle {
    private int index;
    public Handle(){    }
    public Handle(int index) {this.index = index;}
    public void setIndex(int index){this.index = index;}
    public int getIndex(){return this.index;}}
```

Explain:use handle to store the index in the array of T

PriorityQueue class:

```
import java.util.ArrayList;
class PriorityQueue<T> {
    private ArrayList<Combine<T>> pq;
    // constructor
    public PriorityQueue(){
        pq = new ArrayList<Combine<T>>();
        pq.add(null);}
         // Return true iff the queue is empty.
    public boolean isEmpty() {return (pq.size()==1);}
   //return leftchild's key if it exists
    private int leftchild(int pos){
        if(2*pos < pq.size())
            return pq.get(2*pos).getKey();
        else
            return Integer.MAX_VALUE;
    }
  //return rightchild's key if it exists
    private int rightchild(int pos){
        if((2*pos + 1) < pq.size())
            return pq.get(2*pos+1).getKey();
        else
            return Integer.MAX_VALUE;
    }
    //return it's parent
    private int parent(int pos){
        return pq.get(pos/2).getKey();
    }
```

Explain: Given a position(integer),return the key of children or parent. Use ArrayList to store the Combine(a class I created).

```
    private boolean isleaf(int pos){
        return ((pos > (pq.size() - 1)/2)&& pos < pq.size());
    }
```

```
Handle insert(int key, T value)
{
    int pos = pq.size();
    Handle handle = new Handle(pos);
    pq.add(new Combine<T>(key, handle, value));
    while(pos > 1){
        int pKey = parent(pos);
        if(key < pKey){
            int pPos = pos/2;
            swap(pos,pPos);
            pos = pPos;
            }
        else
            return handle;
    }//bubble up until it's smaller than its parent.
    return handle;
}
```

Explain:T is a combine class, contains key, handle, value, so we use handle to access (key, value) pair. Insert to the end, and bubble up to proper place.

```
public void swap(int i, int j){
    Combine<T> iItem = pq.get(i);
    Combine<T> jItem = pq.get(j);
    pq.remove(i);
    pq.add(i,jItem);
    pq.remove(j);
    pq.add(j, iItem);
    pq.get(i).handle.setIndex(i);
    pq.get(j).handle.setIndex(j);
}//swap item i and item j and set the handle right

public void Heapify(int index){

    if(!isleaf(index)){
        int lkey = leftchild(index);
        int rkey = rightchild(index);

        //if it has only 1 child, the non-exist one
        //is Integer.Max_value, and won't affect the next step
        //j is the smallest child's index
        int j = (lkey <= rkey)?(2*index):(2*index + 1);

        //swap when the smallest child is smaller than itself
        if(pq.get(j).getKey() < pq.get(index).getKey()){
                swap(index,j);
                //Heapify at the new position
                if(!pq.isEmpty())
                    Heapify(j);
            }
        }
    }
```

Explain:swap(i,j) in the arrayList. And heapify from the current place to the bottom, and be careful the pq may only have 1 node or it's empty, so add some if statements.

```
    // Return the smallest key in the queue.
    public int min(){return pq.get(1).getKey();}

    // Extract the (key, value) pair associated with the smallest
    // key in the queue and return its "value" object.
    public T extractMin()
    {
        //the smallest is the first item in the pq Heap.
        int n = pq.size() - 1;
        T min = pq.get(1).getValue();

        //swap the 1st and the last one, set the handle -1, and remove
the last one.
        swap(1,n);
        pq.get(n).getHandle().setIndex(-1);
        pq.remove(n);

        //heapify from the top
        if(!this.isEmpty())
            Heapify(1);

        return min;
    }
```

Explain:extractmin and start to heapify at 1st node. And I set the handle.index to be -1, so I won't use the handle again.

```
    // Look at the (key, value) pair referenced by Handle h.
    // If that pair is no longer in the queue, or its key
    // is <= newkey, do nothing and return false.  Otherwise,
    // replace "key" by "newkey", fixup the queue, and return
    // true.
    //
    public boolean decreaseKey(Handle h, int newkey)
    {
        //the position
        int pos = h.getIndex();
        if(pos > 0 && pos < pq.size() && pq.get(pos).getKey() > 
newkey){
            pq.get(h.getIndex()).setKey(newkey);

            while(pos > 1){
                int pKey = parent(pos);
                if(newkey < pKey){
                    int pPos = pos/2;
                    swap(pos,pPos);
                    pos = pPos;
                }
                else
                    return true; //find a proper place in the tree
            }
            return true; //it's in the 1st position now.
        }

        //can't decrease key,
        else
            return false;
    }
```

Explain:use handle.index to locate the node in pq list, and update the key if necessary.

```
    // Get the key of the (key, value) pair associated with a
    // given Handle. (This result is undefined if the handle no
longer
    // refers to a pair in the queue.)
    //
    public int handleGetKey(Handle h)
    {
        if(h.getIndex() <= (pq.size() - 1)&&h.getIndex()!=-1)
            return pq.get(h.getIndex()).getKey();
        return 0;
    }
```

Explain:I use getIndex != -1 to judge if the handle is not defined, since if the value is extracted out, I set the handle value to be -1.

```
    // Get the value object of the (key, value) pair associated with
a
```

```
    // given Handle. (This result is undefined if the handle no
longer
    // refers to a pair in the queue.)
    //
    public T handleGetValue(Handle h)
    {
       if(h.getIndex() < pq.size() && h.getIndex() > 0)
          return pq.get(h.getIndex()).getValue();
       return null;
    }
```

Explain: if handle.index < 0(because I set it to be -1 ,or just be created and not assgined any value, means it's 0, I return null.

```
    // Print every element of the queue in the order in which it
appears
    // in the implementation (i.e. the array representing the heap).
    public String toString()
    {
       String ans = "";
       if(!this.pq.isEmpty()){
          for(int i = 1; i < this.pq.size(); i++)
             ans += this.pq.get(i).toString();
          return ans;
          }
       return ans;
    }

    //pqNode
    public class Combine<T>{
       private int key;
       private Handle handle;
       private  T value;

       public Combine(int key, Handle handle, T value){
          this.key = key;
          this.handle = handle;
          this.value = value;
       }

       public int getKey(){return this.key;}
       public Handle getHandle(){return this.handle;}
       public T getValue(){return this.value;}

       public void setKey(int newKey){this.key = newKey;}
```

```
        public String toString(){return "(" + key + ", " +
value.toString() +")\n";}
    }
```

Explain: In order to associate the handle and the key, value pair together, and I can't change value(class vertex), because in pq class, I use T so I can do nothing to get access T's variables, so I create a Combine<T> as pq list's pqNode.

```
}
```

Shortest path class:

```
class ShortestPaths {

    public class Path{
        public ArrayList<Integer> path = new ArrayList<Integer>();
        }
```

Explain: Use path to remember the past path from the start to the current node, I create a class of path , because for every vertex, it all need a path, so it should be many paths for many vertex.and I don't know how many vertex there are until the constructor of shortestPath takes in G, so use a path class here.So I use ArrayList<Path> allPath = new ArrayList<Path>(); to store all the paths for all the vertex. It's an array of arraylist. And itself is an arraylist too, since I don't know how many vertex there are.

```
    public Path[] allPath;
    public ShortestPaths(Multigraph G, int startId,
           Input input, int startTime)
```

Explain: I did part 1 and part 2, part 1 passed, part 2 passed 2 in 3, use startTime ==0 to decide which function to call.

```
    {
        /*
         * passed part1 test query 1 2 3
         */
        if(startTime == 0){
        //pq is a heap of vertex
        PriorityQueue<Vertex> pq = new PriorityQueue<Vertex>();

        //handles to store the position in the pq
        Handle handles[] = new Handle[G.nVertices()];
        allPath = new Path[G.nVertices()];
        //initializing
        for(int i = 0; i < G.nVertices(); i ++){
            if(G.get(i).id() != startId){
```

```
            handles[i] = pq.insert(Integer.MAX_VALUE, 
G.get(i));
            
            //every airport has a integer arraylist to store the 
edges,
            //at first, the path which is an arraylist is empty.
            allPath[i] = new Path();
         }
      }
      //the begin point
      handles[startId] = pq.insert(0, G.get(startId));
      allPath[startId] = new Path();
      
      while(!pq.isEmpty()){
         //pop out the smallest (uDis, u.vertex) pair
         int uDis = pq.min();
         Vertex parent = pq.extractMin();
         
         if(uDis == Integer.MAX_VALUE)
            break;
         
         //get children of u
         Vertex.EdgeIterator children = parent.adj();
         while(children.hasNext()){
            //children is edge:(id,from,to,weight)
            Edge edge = children.next();
            Vertex child = edge.to();
            int weight = edge.weight();
            int flightId = edge.id();
            //update the path if needed
            if(pq.decreaseKey(handles[child.id()], uDis + 
weight)){
               ArrayList<Integer> childPath = 
allPath[child.id()].path;
               ArrayList<Integer> parentPath = 
allPath[parent.id()].path;
               childPath.clear();
               //copy its parent's past path to it
               for(int i = 0; i < parentPath.size(); i ++){
                  childPath.add(parentPath.get(i));
               }
               //parent's past path + (parent to child) becomes 
child's new path
               childPath.add(flightId);
            }
            }
        }
        }
```

Explain:nothing special here, except I use path arraylist to remember the past path. If decreasekey is true, I need to update the path, by replacing the path with parent's path + parent-->children. And update the value too.

Part 2:

This part I passed test 1 and 3, failed 2.

```
        /**
        query extra 1 2 3 test, passed 1 and 3, failed 2.
        **/
        else{
            //pq is a heap of vertex
            PriorityQueue<Vertex> pq = new PriorityQueue<Vertex>();

            //handles to store the position in the pq
            Handle handles[] = new Handle[G.nVertices()];
            allPath = new Path[G.nVertices()];
            //initializing
            for(int i = 0; i < G.nVertices(); i ++){
                if(G.get(i).id() != startId){
                    handles[i] = pq.insert(Integer.MAX_VALUE,
G.get(i));

                    //every airport has a integer arraylist to store
the edges,
                    //at first, the path which is an arraylist is empty.
                    allPath[i] = new Path();
                }
            }
            //the begin point
            handles[startId] = pq.insert(0, G.get(startId));
            allPath[startId] = new Path();
        while(!pq.isEmpty()){
```

Explain: udis is the cost time from the starttime.

```
            //pop out the smallest (uDis, u.vertex) pair
            int uDis = pq.min();
            Vertex parent = pq.extractMin();

            if(uDis == Integer.MAX_VALUE)
                break;
            int parentArrivalTime = uDis;
            //get children of u
            Vertex.EdgeIterator children = parent.adj();

            while(children.hasNext()){
                //children is edge:(id,from,to,weight)
                Edge edge = children.next();
                int departFromParent =
input.flights[edge.id()].startTime;
                int endToChildren =
input.flights[edge.id()].endTime;

                int truecost = 0;
                int waitTillDepart = 0;
                /*
                 * judge if we take this flight or the flight the
next day
                 * waitTillDepart is the time we wait for the flight
taking off
                 */
```

Explain: hard to say, I'm confused, I didn't understand teacher's instruction, just use my method here, divide true cost for every edge to 2 parts: the waiting time till the flight takes off in the parent's airport, use if to judge if it's another day. Calculate the real time it takes from parent to children, add them together is the true cost.

```
                if((departFromParent - 45) > (parentArrivalTime +
startTime)%2400){
                    waitTillDepart = (departFromParent  -
(parentArrivalTime + startTime)%2400)%2400;

                }
                else
                    waitTillDepart = (departFromParent  -
(parentArrivalTime + startTime)%2400)%2400 +2400;
                /*
```

```
                 * actualFlightTime is the actual flying time
                 */
                int actualFlightTime = (endToChildren -
departFromParent + 2400)%2400;

                /*
                 * add together
                 */
                truecost = waitTillDepart + actualFlightTime;



                Vertex child = edge.to();
                int weight = edge.weight();
                int flightId = edge.id();

                //update the path if needed
                if(pq.decreaseKey(handles[child.id()],
truecost+uDis)){
                   ArrayList<Integer> childPath =
allPath[child.id()].path;
                   ArrayList<Integer> parentPath =
allPath[parent.id()].path;
                   childPath.clear();
                   //copy its parent's past path to it
                   for(int i = 0; i <  parentPath.size(); i ++){
                      childPath.add(parentPath.get(i));
                   }
                   //parent's past path + (parent to child)
becomes child's new path
                   childPath.add(flightId);
             }
             }
          }

       }
   // your code here
   }

   //
   // returnPath()
   // Return an array containing a list of edge ID's forming
   // a shortest path from the start vertex to the specified
   // end vertex.
```

```
    //

    public int [] returnPath(int endId)
    {
```

Explain: every vertex(airport) has a arraylist to store the past path.allPath is an array of path class, and path class is an arraylist stores all the edges’ id.

```
    // your code here
    ArrayList<Integer> pforend = allPath[endId].path;
    int[] ans = new int[pforend.size()];
    for(int i = 0 ; i < ans.length; i++){
        ans[i] = pforend.get(i);
    }
    return ans;
    }
}
```