a)

My code for finding the smallest weight among all the weighted inbound edges for each node in the graph consists of 3 parts: the mapper, the reducer, and driver functions.

The mapper takes a key-value input of Object-Text and outputs a key-value pair of Text-IntWritable. The algorithm is as follows:

1. The StringTokenizer takes each line of input and breaks it down into separate column values; I use the toString() method to convert each value into strings.
2. Because I am not interested in the 1st column(src), I skip it with itr.nextToken().
3. Then, I set the next value to "tgt", the key of the output.
4. Finally, I use the while loop to iterate through the values and set "weight" to the next token each time; this way, the last value in each row will eventually end up to be the "weight".
5. I convert this value to an integer so that I can write it out as an IntWritable when I output it as value along with "tgt"(which is the key).

Now we have key-value mappings of "tgt"-"weight" pairs. The reducer's job is to return only those pairs who have minimum "weight" values for each "tgt". The algorithm is as follows:

1. For each "weight" value, I check if the value is less than a minimum value. If it is less, I replace the minimum value with that value. I set the default "min" value to 10000, a large value to initiate this process.
2. After iterating through each key-value mapping, I set the final "result" value to "min". This is the absolute minimum value for each key.
3. I write the key-value mappings as output.

The final step is to setup the driver which puts it all together.

1. I first set up the "job" and set the main class.
2. Next, I set the Mapper, Combiner, and Reducer classes.
3. Then, I set the key and value type classes of the output.
4. Finally, I set the input and output paths from the arguments.

b)

I will use a similar approach to design an algorithm that joins an unordered collection of two kinds of records. The difference is that I will need to implement additional classes to ensure that the student name always precedes the department name. So I will start with that.

First, I need to create a 'TaggedKey' class of the WritableComparable interface that will be used to wrap the keys when joining entries. 'TaggedKey' will have two components, 'joinkey' and 'tag'. The class will employ the compareTo function which will match the 'joinkey' and the 'tag' values. This ensures that the order in which the joins occur are determined by 'tag'.

However, I still want only the 'joinkey' to be considered when determining which reducer to go through; thus, I write a custom partitioner class using .hashcode() to make sure that all outputs of the mapper with the same 'joinkey' pass through the same reducer.

Also, it would be helpful to use a comparator class that groups values based solely on the ‘joinkey’. I would accomplish this by referencing the superclass TaggedKey and returning the match between the ‘joinkey’ values of taggedKeys.

Now I can proceed to the Mapper:

- The mapper will take an input of LongWritable-Text and output Taggedkey-Text.
- Before I proceed to the map function, it would be helpful to write a setup method that achieves the following:
    - Designate variables that split and join the data on separator = “, “ using Guava Splitter and Joiner.
- The map function will
    - Create and populate a list with split & converted-to-string values
    - Designate to ‘joinkey’ value the ‘key’ that will be used to join
        - 3rd value in the list if 1st value == ‘Student’
        - 2nd value in the list if 1st value == ‘Department’
    - Designate ‘tag’ value the order at which the join will happen
        - 1 if 1st value == ‘Student’
        - 2 if 1st value == ‘Department’
    - Set taggedKey to (joinkey, tag)
    - Set String variable content to the joined String of list after removing the 1st value and the ‘joinkey’
    - Map (taggedKey, content) using context.write();

For the Reducer:

- The reducer will take TaggedKey-Text as inputs and output NullWritable-Text.
- For each key:
    - After creating a StringBuilder, I will first append the ‘joinkey’ to it.
    - Then I will iterate over the Text values and append each value using the “, “ separator to the StringBuilder.
    - After removing the last two characters of the builder(to remove the final “, “), I will convert the builder to a String.
    - I will set the Text variable Finaltext to the output from above and set the NullWritable variable nullKey to NullWritable.get();
    - I will use context.write(nullKey, Finaltext) to reduce and initialize the Stringbuilder by setting its length to zero.

Finally, for the Driver:

- I will first set up the “job” and set the main class.
- Next, I will set the mapper, reducer, partitioner, and comparator classes.
- Then, I will set the key and value type classes of the output.
- Lastly, I will set the input and output paths.