Part 1:

```
StringTable table = createTable(patternSeq, matchLength);
```

```
StringTable table = new StringTable(patternSeq.length());
```

Explain:Create a table with a load factor of 1/4 and a size of power of 2.

Read in String patternSeq and divide into many substrings with a length of matchLength, insert the substrings into the table.

My createTable is as below:

```
public Record[] table;

    public StringTable(int maxSize)
    {
      int n=0;
      while(maxSize/2!=0){
            n++;
            maxSize/=2;
      }
      n+=3;
      table=new Record[(int) Math.pow(2,n)];
      }
```

Explain:

1) **In StringTable class, declare a public record array table.**
2) **The load factor a=1/4, and the table size should be the power of 2, so after setting stephash as an odd number, it will become a good hash function.**
3) **MaxSize =patternSeq.length(), find the closest power of 2 of MaxSize. If MaxSize=31(case 1), the closest power of 2 is 32, so the table size should be 128(pow(2,7)).**

My basehash and stephash are:

```
int baseHash(int hashKey)
    {
      // Fill in your own hash function here
      int m=table.length;
      double A=(Math.sqrt(5)-1)/2;
      int base=(int) Math.floor(m*(hashKey*A-Math.floor(hashKey*A)));
      return base;
    }

    int stepHash(int hashKey)
```

```
    {
      // Fill in your own hash function here
      int m=table.length;
      double A=Math.sqrt(2)/2;
      int step=(int) Math.floor(m*(hashKey*A-Math.floor(hashKey*A)));
     if(step%2==0)
         step++;
      return step;
    }
```

Explain:

1) **m is the table.length, for case 1 ,it's 128.**
2) **Hashkey is the return result of toHashKey, it's a big integer.**
3) **The toHashKey function use the char variable's integer value(ASCAL) and its position in the string to calculate the hashkey, so every substring can have the same hashkey value.**
4) **My basehash and stephash use multiplication hash function,(int)m*(kA-math.floor(kA)),but the step hash needs to be odd, so ,if it's even, return step++**
5) **baseHash and stepHash use different irrational A**

My insert function:

```
public boolean insert(Record r)
    {
      int hashKey=toHashKey(r.key);
      int base=baseHash(hashKey);
      int step=stepHash(hashKey);
      int maxsize=table.length;
      int k=base%maxsize;
      for(int i=0;i<maxsize;i++){
            if(table[k]==null){
                  table[k]=r;
                  return true;
            }
            else
            {
                  base=base+step;
                  k=base%maxsize;
            }
       }
      return false;
    }
```

Explain:

1) Use the toHashKey to calculate the hashKey of the record r.key, it's a big integer
2) Use the basehash and stephash to calculate the base and the step
3) Look for s0(k)=base, if it's null, we insert the record r.key into it, and return true.
4) If it's not null, we calculate base+step,base+2*step.....until we find a place to insert the r.
5) If the whole table is full, we end the loop and return false

My remove function:

```
public void remove(Record r)
    {
      if (find(r.key)!=null){
            r.key=new String("deleted");
      }
    }
```

My find function:

```
public Record find(String key)
    {
      int hashKey=toHashKey(key);
      int base=baseHash(hashKey);
      int step=stepHash(hashKey);
      int m=table.length;
      int k=base%m;
      for(int i=0;i<m;i++){
            if(table[k]!=null){
                  if(!((table[k].key).equals("deleted")))
                  {
                        if((table[k].key).equals(key))
                              return table[k];
                  }
                  else{
                  base=base+step;
                  k=base%m;
```

```
                }
            }
            else
                return null;
        }
        return null;
    }
```

Explain:

1) Use the same method to get the hash function, if it's null, means it's not there,just return null
2) If it's not null, could be the wrong key, the right key, and deleted, if it's the right key, we return the key and the position, if it's not, we search for the next one.
3) When we remove an item,first find the key,if we find it, reset it to be "deleted".

findMatches use the same method, get the substring of the corpusseq, and calculate the hash function, because the hash function is related to the string itself(the char value and the position in the string), so if the substring of corpusSeq and the substring of patternSeq are the same.They have the same hash function, we just search the exact place in the table to see if it's there.

maskTable use the same method, get the substring of maskTable and find them in the patternseq's table, if the patternseq's table contains the substring from the maskTable, we deleted the string in patterseq's table.

Part 2:

1)Modify the record class, add item int hashKey to store every string's to hashkey(s),and in find,remove function,instead of comparing two strings directly, we compare their hashKey to see if they are the same, if they are not the same,look for the next position.

2)if their hashkeys are the same, we then compare the two strings(Notice,I think even if the two hashkeys are the same, the 2 strings still can be different)

```
public class Record {

    public String key;
```

```
    public ArrayList<Integer> positions;
    public int hashKey;  //add an item

public Record find(String key)
    {
      int hashKey=toHashKey(key);
      int base=baseHash(hashKey);
      int step=stepHash(hashKey);
      int m=table.length;
      int k=base%m;
      for(int i=0;i<m;i++){
            if(table[k]!=null){

      if(!((table[k].key).equals("deleted"))&&table[k].hashKey==hashKey)
                  {
                        if((table[k].key).equals(key))
                              return table[k];
                  }
                  else{
                  base=base+step;
                  k=base%m;
                  }
            }
            else
                  return null;
      }
      return null;
    }
```

Part 2 double hash table:

In this part, I changed StringTable and insert function:

1)Declare a public integer n to represent the current inserted number.

So,every time compare n and size of table to decide whether or not to double the table.

2)StringTable function: create a new table with a size of 2

3)Insert function:insert record r,before insert, judge if n<m/4, if it's,we insert the r into the table and n++,then return true

4)If,n=m/4 or n>m/4,create an array old to copy all the items in table to the array,and reset table to be null, then, create a new table with size 2*m,so, the hashfunction changes too, but because we always calculate basehash and stephash with the current table.length, so we don't need to change anything here. Then we insert all the old items to the new table, at the meanwhile, n++ after we insert a new item.

5)In remove function,we also need to add "n--", after we delete an item.That is release a slot in the table.

```
public boolean insert(Record r)
    {
          int m=table.length;
     if(n<m/4){
          int hashKey=toHashKey(r.key);
          int base=baseHash(hashKey);
          int step=stepHash(hashKey);
          int k=base%m;
          r.hashKey=hashKey;
          for(int i=0;i<m;i++){
               if(table[k]==null){
                    table[k]=r;
                    n++;
                    return true;
                    }
               else{
                    base=base+step;
                    k=base%m;
               }
```

```
        }
    }
    else{
        Record[] old=new Record[m];
        int j=0;
        for(int i=0;i<m;i++)
            if(table[i]!=null){
                old[j]=table[i];
                j++;
                table[i]=null;
                }
        table=new Record[2*m];
        n=0;
        for(int i=0;i<m;i++)
            if(old[i]!=null)
            insert(old[i]);
        insert(r);
        return true;
        }
    return true;
}
```