7.

(1) Begin

```
Declare Function Remove(int  )
    int hash-val = HashFunc(k)
    while (hash_val != init and (ht[hash_val]==DelNode::getNode()
            or ht[hash_val]!= NULL and ht[hash_val]→k != k)).
        if(init==-1)
            init=hash_val
        hash_val=HashFunc(hash_val)
    if(hash_val !=init && ht[hash_val]!=NULL)
        delete ht[hash_val]
        ht[hash_val]=DelNode::getNode()
End.
```

(2) Assume hash(x)=hash(y)=hash(z)=i. And assume x was inserted first, then y and then z. In open addressing:

table[i]=x, table[i+1]=y. table[i+2]=z.

Now, assume you want to delete x. and set it back to NULL.

When later you will search for z. you will find that hash(z)=i and table[i]=NULL. and you will return a wrong answer. z is not in the table.

To overcome this, you need to set table[i] with a special marker indicating to the search function to keep looking at index (i+1) because there might be element there which its hash is also i;

(3) This is a method for resolving collisions and trying to find another open slot to hold the item that caused the collision.

(4) An element of key k hashes to slot h(k).

Compute h(k) to determine which list to traverse.

If ht[h(k)] contains a null pointer, initiatize this entry to point to a linked list that contains k alone. If ht[h(k)] is a non_empty list, we add k at the beginning of this list.