Zongming Liu (47357124)
j4v9a

Q1:

```
void insert(Node *a, Node *b) {
  if (b == null) {
  print "error, the second list is empty!";
  return;
  }
  if (a == null) {
  return;
  }
  Node *temp = a;
  while (temp->next != null) {
  temp = temp->next;
  }
  temp->next = b->next;
  b->next = temp;
  return;
}
```

Q2:

(a)
if the structure is an array, then a single array will represent a street, with the identifier of the array being name of the street. The objects stored in each array would be the type of double, in an increasing order, representing house numbers on that street.

To find next-door house numbers given a street name and a house number, first, we need to find the array representing the street name. Then, we will use a for loop, to traverse the array to find the position of the element containing the given house number. Lastly, output the numbers stored in position-1 and position+1. Regarding the time complexity, this algorithm would take big-Theta(n) time, since the worst case should probably be T(n) = 2n+1, and if we take c=3, then T(n) belongs to big-Oh(n), however, if we take c=1, then T(n) belongs to big-Omega(n). Therefore, it would take big-Theta(n) time.

To add a new house after the given house, first, we need to find the array representing the street name. Then, we will use a for loop, to find the given house with number k. If there is already a house with number k+1, we give the new house any number between k and k+1. Otherwise, we just assign the number k+1 to the new house. In the meantime, we need to create a new array with the size of given array's size+1. Then, we just copy everything in the old array to the new one, with the new house. Regarding the time complexity, this algorithm would take big-Theta(n) time. Because T(n) is approximately equal to 4n. If we take c=3, then T(n) belongs to big-oh(n), whereas, if we take c = 5, then T(n) belongs to big-Omega(n). Thus, it would take big-Theta(n) time too.

Zongming Liu(47357124)
j4v9a

(b)
if my structure is a singly-linked list, then a list will represent a street, with each node representing each house on that street.

To find next-door house numbers given a street name and a house number, first, we need to traverse the list with given number, this would take n in the worst case; further, to compare the data of each node with our given number, it would take n in the worst case; in the meantime, we need to make our pointer to point to the next node each time we do not find the given house, this would take n-1 in the worst case. Thus, T(n) is approximately equal to 3n-1, which indicates that T(n) belongs to big-Theta(n). To be more specific, if c=1, then T(n) belongs to big-Omega(n); else if c=4, then T(n) belongs to big-Oh(n). Therefore, T(n) belongs to big-Theta(n).

To add a new house after the given house, first, we need to find the given house, which will take approximately 3n-1, then, we make the given house point to our new house, and our new house point to the house originally pointed by the given house. Therefore, this algorithm would take big-Theta(n) as well! On one hand, if c=1, then T(n) belongs to big-Omega(n); on the other hand, if c=4, then T(n) belongs to big-Oh(n). Thus, T(n) belongs to big-Theta(n).

(c)
I would choose array if I should pick one. Given they have the same time complexity, array is easier to traverse through, and it is not that hard to insert a new object into an array, though we have to create a new one and copy everything in the old one to the new one, clearly linked list is more efficient on this, since we can just change the object our given house pointing to.

Q3:

(a)
lg(32^n)=5n

(b)
2^(lg(n^2 * m^2)-lg(m^2))=n^2

(c)
-lg(1/8)=3

(d)
log base p (1/p)=-1

(e)
64^(lg(n^2))=n^12

Q4:

Zongming Liu(47357124)
j4v9a

log(n) -> n^(1/2) -> n -> n*log(n) -> lg(n!) -> sigma i^(3) from 1 to n -> n^(2*lg(n)) -> 2^n -> 2^(2^(n)) -> n^n

first, I take the limit n to infinity of log(n)/n^(1/2), which equals to 0, which means that log(n) <= n^(1/2). Then, it's easy to see that n^(1/2) <= n by exponents' differences. Also, given that log(n) belongs to big-Omega(1), so we know that n <= n*log(n). Further, lg(n!)/sigma i^(3) from 1 to n with n goes to infinity outputs 0, so lg(n!) <= than the i^3 one; meanwhile, lg(n!)/n*log(n) with n goes to infinity outputs a constant, which means they have same rate of growth. Moreover, i^3 could be seen as n^3 since it is the highest order, so we compare the exponent of n^3 and n^(2*lg(n)), it is obvious that 3 <= 2*lg(n), thus i^3 one <= n^(2*lg(n)). For the rest, I all took the limit with n goes to infinity of their fractions, and got the result.

Q5:

(b)
big-Theta(n^2), given that T(n)= 24n^2+80n+24, taking the highest order, which is 24n^2, then, we drop the constant, which left us with n^2.

(c)
big-Theta(2^(n)), taking the highest order of the polynomial, which should be 2^(n+c), then we omit the constant c, get the result.

(d)
big-Theta(n^3), given that the formula given will produce c*n^3.

(e)
big-Theta(2^(n/2)), by expanding T(n) to T(n)=2^((n-1)/2)*T(1)+c, and we got the answer.

(f)
big-Theta(lg(n)), by expanding T(n) to T(n)= T(1) + 3*lg(n), and we got the answer.

Q6:

(a)
proof - Let c=1, then we know for sure that 54n^3+17 belongs to big-Omega(n^3); However, let c=72 and n0=1, then we know 54n^3+17 belongs to big-Oh(n^3). Therefore, 54n^3+17 belongs to big-Theta(n^3).

(b)
proof - We use L'Hospital's Rule to examine 54n^3+17/c*(n^2), with c being any positive integer, taking the limit n to infinity, and we get infinity, which means that 54n^3+17 will belong to big-Omega(n^2) of all time, since there does not exist a positive c that could make

54n^3+17/c*(n^2) equals to zero, with n goes to infinity.

(c)
proof - first, let us prove that T(n) belongs to big-Oh(n^d), if we take a positive integer c that is significantly than ad, then we could say that T(n) belongs to big-Oh(n^d); on the other hand, if we take a positive integer c that is significantly than ad, then we could say that T(n) belongs to big-Omega(n^d), therefore, T(n) belongs to big-Theta(n^d).

Q7:

(a)
T(n)=lg(n), the 'double result= pow(x*x, n/2)' will use lg(n), other lines would use constant time as usually, therefore, the time complexity of this function is lg(n).

(b)
T(n)=n^2, given that there are two for loops, the highest order term would be n^2 after removing the less order terms. Therefore, the time complexity of this function is n^2.