TIME COMPLEXITY ANALYSIS

For the time complexity analysis, I used different sized data and calculated running times of the 4 seperate programs like this:

```java
long start;
long end;
start = System.nanoTime();
LinkedList<Integer> mylist1;
mylist1 = new LinkedList<>();
for(int x=0; x<=10; x++){
    mylist1.add(x);
}
end = System.nanoTime();
System.out.println(end-start);
start = System.nanoTime();
LinkedList<Integer> mylist2;
mylist2 = new LinkedList<>();
for(int x=0; x<=100; x++){
    mylist2.add(x);
}
start = System.nanoTime();
LinkedList<Integer> mylist3;
mylist3 = new LinkedList<>();
for(int x=0; x<=1000; x++){
    mylist3.add(x);
}
end = System.nanoTime();
System.out.println(end-start);
end = System.nanoTime();
System.out.println(end-start);
```

-For the first homework assignment, these were the results when I ran the program 3 times:

```
86400
142800
5417700
```

```
78800
211700
8393700
```

```
88500
199000
5386200
```

So, this means with every x10 extendition of the data, program takes x10 more time. That means the relation is linear, so T(n) = Θ(n)

And this is the theoritical time complexity of the add method:

```java
@Override
@SuppressWarnings("unchecked")
public boolean add(Object e) {
    int i;
    if (size>=capacity) {
        capacity=capacity+1;
    }
    if (data==null) {
        data = (E[])new Object[this.capacity];
    }
    E[] tempArr = (E[])new Object[capacity];
    for (i = 0; i < size; i++) {
        tempArr[i]=data[i];
    }
    tempArr[size]=(E)e;
    size=size+1;
    data=tempArr;
    return true;
}
```

$\Theta(1)$

$\Theta(n)$

$\Theta(1)$

$T(n) = \Theta(n)$

-For the version with arraylist, these were the results when I ran the program 3 times:

```
162600
403600
575400
```

```
103800
282200
547800
```

```
257300
266600
598500
```

So, this means with every x10 extendition of the data, program takes about the same time. That means this method always takes constant time. So, T(n) = Θ(1)

And this is the theoritical time complexity of the add method:

```
this.houses1.add(house);
this.length1 = this.length1 - house.getLength();
this.total += house.getLength();
this.side1[index1][0] = house.getPosition();
this.side1[index1][1] = house.getLength();
this.side1[index1][2] = house.getHeight();
this.index1++;
System.out.println("House added successfully!");
```

Θ(1)

-For the version with linkedlist, these were the results when I ran the program 3 times:

```
100300
435300
652800
```

```
105800
385200
766300
```

```
123800
457000
745900
```

So, this means with every x10 extendition of the data, program takes about the same time. That means this method always takes constant time. So, T(n) = Θ(1)

And this is the theoritical time complexity of the add method:

```
this.houses1.add(house);
this.length1 = this.length1 - house.getLength();
this.total += house.getLength();
this.side1[index1][0] = house.getPosition();
this.side1[index1][1] = house.getLength();
this.side1[index1][2] = house.getHeight();
this.index1++;
System.out.println("House added successfully!");
```

Θ(1)

-For the version with ldlinkedlist, these were the results when I ran the program 3 times:

```
70300
6311300
6605300
```

```
64500
5914800
6221200
```

```
66300
6820300
7020500
```

So, these results were rather more complicated than the previous ones. T(n) was Θ(n^2) somewhere, and it was Θ(1) elsewhere. So the result actually is O(n).

And this is the theoritical time complexity of the add method:

```
@Override
@SuppressWarnings("unchecked")
public boolean add(Object e) {
    int i;
    if (size>=capacity) {
        capacity=capacity+1;                    // Θ(1)
    }
    if (data==null) {
        data = (E[])new Object[this.capacity];  // Θ(1)
    }
    E[] tempArr = (E[])new Object[capacity];
    for (i = 0; i < size; i++) {                // Θ(n)
        tempArr[i]=data[i];
    }
    tempArr[size]=(E)e;                         // Θ(1)
    size=size+1;
    data=tempArr;
    return true;
```

$T(n) = \Theta(n)$